A
Project Report on

"Chat Bot using Natural Language Understanding"

Project submitted
In partial fulfilment of the requirements for the degree of

MASTER OF COMPUTER APPLICATION
By
SAIKAT BHATTACHARYA

Class Roll No: 001510503013
Exam Roll No: MCA186013
Registration No: 133675 of 2015 - 2016

Under the supervision of
DR. SUDIP KUMAR NASKAR

Department of Computer Science and Engineering
Faculty of Engineering and Technology
Jadavpur University
Kolkata – 700 032
India

A
Project Report on

"Chat Bot using Natural Language Understanding"

Project submitted
In partial fulfilment of the requirements for the degree of

MASTER OF COMPUTER APPLICATION
By
SAIKAT BHATTACHARYA

Class Roll No: 001510503013
Exam Roll No: MCA186013
Registration No: 133675 of 2015 - 2016

Under the supervision of
DR. SUDIP KUMAR NASKAR

Department of Computer Science and Engineering
Faculty of Engineering and Technology
Jadavpur University
Kolkata – 700 032
India

Jadavpur University

Faculty of Engineering and Technology

Department of Computer Science & Engineering

TO WHOM IT MAY CONCERN

This is to clarify that the project entitled ***"Chat Bot using Natural Language Understanding"*** *has been completed by* ***Saikat Bhattacharya.*** *This work is carried out under the supervision of* ***Dr. Sudip Kumar Naskar*** *in partial fulfilment for the award of the degree of* ***Master of Computer Application*** *of the department of* ***Computer Science and Engineering, Jadavpur University,*** *during the session 2017-2018. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said degree.*

(Signature of the head of the department)

Prof. Ujjwal Maulik, Head of the department of Computer Science and Engineering

(Signature of the project supervisor)

DR. Sudip Kumar Naskar

(Signature of the Dean of the faculty)

Prof. Chiranjib Bhattacharjee, Dean, Faculty Council of Engineering and Technology

Jadavpur University

Faculty of Engineering and Technology

Department of Computer Science & Engineering

CERTIFICATE OF APPROVAL

This is to clarify that the project entitled ***"Chat Bot using Natural Language Understanding"*** *has been completed by* ***Saikat Bhattacharya.*** *This work is carried out under the supervision of* ***Dr. Sudip Kumar Naskar*** *in partial fulfilment for the award of the degree of* ***Master of Computer Application*** *of the department of* ***Computer Science and Engineering, Jadavpur University,*** *during the session 2017-2018. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said degree.*

____ ____

(Signature of the internal examiner) (Signature of the external examiner)

DECLARATION OF ORIGINALITY AND COMPLIANCE OF ACADEMIC ETHICS

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

I declare that this is a true copy of my thesis, including any final revisions, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Candidate's Name:	Saikat Bhattacharya
Class Roll No:	001510503013
Exam Roll No:	MCA186013
Project Title:	Chat Bot using Natural Language Understanding

Date: ___ / ___ / ____

Place: ____________

____
(Signature of the candidate)

… full text …

ABSTRACT

Chatbots are poised to revolutionize User Interface design. Here's a quick summary of what chatbots are all about.

Chatbots, or conversational interfaces as they are also known, present a new way for individuals to interact with computer systems. Traditionally, to get a question answered by a software program involved using a search engine or filling out a form. A chatbot allows a user to simply ask questions in the same manner that they would address a human. The most well-known chatbots currently are voice chatbots: Alexa and Siri. However, chatbots are currently being adopted at a high rate on computer chat platforms.

The technology at the core of the rise of the chatbot is natural language processing ("NLP"). Recent advances in machine learning have greatly improved the accuracy and effectiveness of natural language processing, making chatbots a viable option for many organizations. This improvement in NLP is firing a great deal of additional research which should lead to continued improvement in the effectiveness of chatbots in the years to come.
A simple chatbot can be created by loading an FAQ (frequently asked questions) into chatbot software. The functionality of the chatbot can be improved by integrating it into the organization's enterprise software, allowing more personal questions to be answered, like "What is my balance?", or "What is the status of my order?".

Most commercial chatbots are dependent on platforms created by the technology giants for their natural language processing. These include Amazon Lex, Microsoft Cognitive Services, Google Cloud Natural Language API, Facebook Deep Text, and IBM Watson. Platforms where chatbots are deployed include Facebook Messenger, Skype, and Slack, among many others.

Table of contents

- **INTRODUCTION……………..……………………..……………….. 1 - 3**
  1. **What are chatbots? ………………..…………….……………… 1**
  2. **History of chatbots ………………..…………….……………… 1 - 2**
  3. **How do chatbots work? …………..……………….…............... 2**
  4. **The potential of chatbots …………..….,……….……………… 3**

- **Chapter - 1: Creating and testing Bots……..………………... 4 - 11**
  1. **Bot service ……………………..……………………………….. 4 - 6**
     - **1.1. Log into Azure …...…………………………….…..…............... 4**
     - **1.2. Create a new bot service …………………..……………............ 4 - 5**
     - **1.3. Test the bot …………………………..……………………….... 6**
     - **1.4. Bot settings overview …………..……………………………..... 6**
     - **1.5. Bot management ……..……………………………….............. 6**
  2. **App service settings ……………………………………….. 6 - 8**
     - **2.1. MicrosoftAppID and MicrosoftAppPassword …………............. 6**
     - **2.2. Edit a bot with online code editor ……………………….......... 6 - 8**
  3. **Create a Bot with the Bot Builder SDK for .NET …............ 8 - 11**
     - **3.1. Prerequisites …………………………………………………… 8**
     - **3.2. Create the bot …………………………………………………... 8**
     - **3.3. Verify that the project references the latest version of the SDK 9**
     - **3.4. Explore the code ……………………………………………..… 9 - 10**
     - **3.5. Test the bot ..……………………………………………….…… 10**
     - **3.6. Start the bot ……………………………………………….….... 11**
     - **3.7. Start the emulator and connect the bot ……………….……..... 11**
     - **3.8. Test the bot code result ………………………………….…........ 11**

- **Chapter - 2: Language Understanding (LUIS) ……………... 12 - 20**
  1. **What is a LUIS app? ……………………………………….. 12**
     - **1.1. Key LUIS concepts ……………………………………............... 12**
     - **1.2. Accessing LUIS ………………………………………………… 13**
     - **1.3. Author the LUIS model ……………………………..…............. 13**
     - **1.4. Identify Entities ………………………………………..………... 13**
     - **1.5. Improve performance ………………………………..………… 13**
  2. **Create new app with intents ……………………….…..… 14 - 16**
     - **2.1. Simple app with intents ……………………………………....… 14**
     - **2.2. Create a new app …………………………………………….... 14 - 16**
  3. **Create new app with intents and entities ………………..... 16 - 20**
     - **3.1. Simple app with intents and a simple entity ……………………... 16**
     - **3.2. Create a new app ……………………………………………….. 16 - 20**

- **Chapter - 3: Key concepts in Bot Builder SDK ………………. 21 - 30**
  1. **Connector ……………………………………………………… 21**
  2. **Activity ………………………………………………………… 21**
  3. **Dialog ……………………………………………………….... 21**
  4. **FormFlow ……………………………………………………… 21**
  5. **State …………………………………………………………… 21**
  6. **Naming conventions …………………………………………... 21**
  7. **Messages and Activities ……………………………………… 22 - 24**
     - 7.1. **Activities overview ………………………………………………… 22**
     - 7.2. **Add speech to messages …………………………………………... 22 - 23**
     - 7.3. **Add input hints to messages ……………………………………..... 24**
  8. **Dialogs ……………………………………………………..... 25 - 30**
     - 8.1. **Dialogs in the Bot Builder SDK for .NET ………………...….... 25 - 26**
     - 8.2. **Manage conversation flow ……………………………….…..... 26 - 28**
     - 8.3. **Scorable Dialogs ……………………………………….……... 28 - 30**

- **Chapter - 4: Including speech support in Bots ………………..... 31 - 33**
  1. **How to change the auto-generated app password …………... 32 - 33**
  2. **How to give speech input to the emulator …………………… 33**

- **Chapter - 5: App specifications ……………………………….. 34 - 43**
  1. **Name of the app ……………………………………………….. 34**
  2. **Functions ……………………………………………………… 34**
  3. **Intents ………………………………………………………….. 34 - 35**
  4. **Entities ………………………………………………………… 35 - 36**
  5. **Utterances ……………………………………………………… 36 - 40**
  6. **Adding Phrase lists …………………………………………… 40**
  7. **Testing ………………………………………………………….. 41**
  8. **Publishing …………………………………………………..... 41 - 43**

- **Chapter – 6: App Responses …………………………………... 44 - 50**
  1. **Intents extraction ……………………………………………… 45 - 48**
  2. **Intents and entities extraction ………………………………… 49 - 50**

- **CONCLUSIONS …...………..…………………………………….. 51**

- **REFERENCES ……………………………………………………….. 52**

INTRODUCTION

1. What are chatbots?

A chatbot is a program that communicates with a user.
It is a layer on top of, or a gateway to, a service. Sometimes it is powered by machine learning (the chatbot gets smarter the more the user interacts with it). Or, more commonly, it is driven using intelligent rules (i.e. if the person says this, respond with that).

The services a chatbot can deliver are diverse. Important life-saving health messages, to check the weather forecast or to purchase a new pair of shoes, and anything else in between.
The term chatbot is synonymous with text conversation but is growing quickly through voice communication… "Alexa, what time is it?" (other voice-chatbots are available!)
The chatbot can talk to the user through different channels, such as Facebook Messenger, Siri, WeChat, Telegram, SMS, Slack, Skype and many others.

Consumers spend lots of time using messaging applications (more than they spend on social media). Therefore, messaging applications are currently the most popular way companies deliver chatbot experiences to consumers.

Aside from buying shoes, here are a few more examples of companies using chatbots:

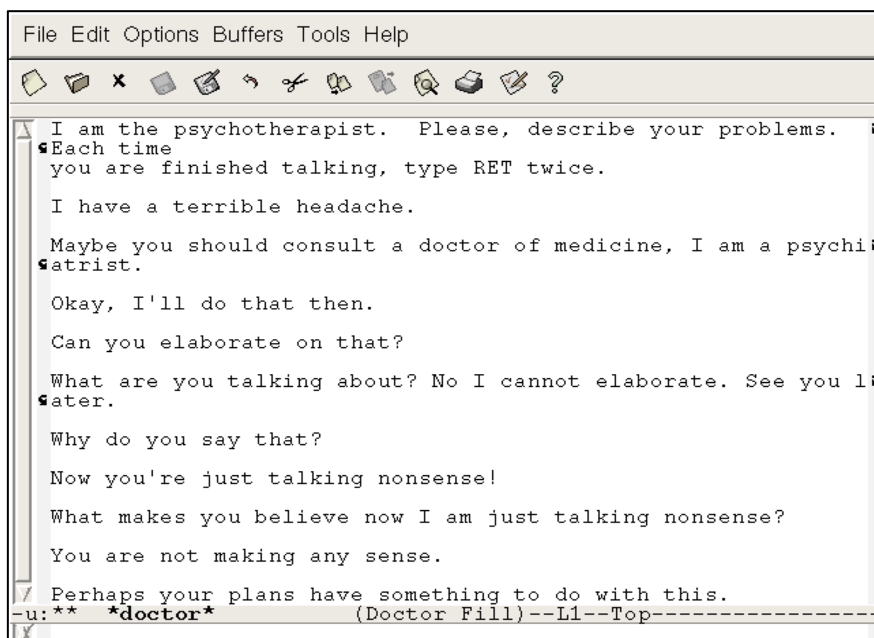
- Uber to book a taxi
- KLM to deliver flight information
- CNN to keep the user up-to-date with news content
- TechCrunch to keep the user up-to-date with techie content
- Pizza Hut to help the user order a pizza
- Sephora to provide beauty tips and a shopping experience
- Bank of America to connect customers and their finances

The possibilities are (almost) limitless.
So, from where did chatbots come?

2. The history of chatbots

It would not be fair to talk about the history of **chatbots** without mentioning Alan Turing and Joseph Weizenbaum. These men imagined computers talking like humans and, in 1950, had the foresight to develop a test to see if a person could distinguish human from machine: **the Turing Test**.

In 1966 a computer program called ELIZA was invented by Weizenbaum. It imitated the language of a psychotherapist from only 200 lines of code. One can still talk with it here: **Eliza**.



The first move away from text chatbots occurred in 1988 when Rollo Carpenter started the Jabberwacky project – a voice operated entertainment AI chatbot.

In the year 2000, Robert Hoffer from ActiveBuddy Inc. co-created the SmarterChild chatbot that used AOL Instant Messenger and MSN Messenger to build a relationship with over 30 million users. The chatbot provided access to news, weather, movie times and acted as a personal assistant using natural language comprehension.

Microsoft Research has spent decades working on Natural Language Processing (NLP) to develop their XiaoIce chatbot. With millions of followers in China, the chatbot can discern topic, sentiment and more through back and forth conversation with its users.
Recent developments in technology have given chatbots more power in interpreting natural language and machine learning, to both understand better, and learn over time.
Huge companies like Facebook, Apple, Google and Microsoft are contributing significant resources to deliver interactions between consumers and machines with commercially-viable business models.

3. How do chatbots work?

There are broadly two variants of chatbots.
One follows a set of rules, flows, and triggers to respond to very specific commands. A simple example might be a chatbot that tells the user the weather forecast for a location. A user might ask “weather forecast London” and the chatbot would find the answer and respond. This type of chatbot is only as smart as the developers who created it and thought of every eventuality of conversation.
The other variant uses machine learning to try to understand the sentiment and meaning of the language used, to not rely on pre-planned commands. A user might ask “what’s been happening in London lately?” and the chatbot might deliver the latest BBC News headlines for London. This type of chatbot learns from all the conversations it has had to improve accuracy and understanding over time.
The use of natural, everyday language in their responses creates the illusion that chatbots are simple creatures, but that could not be more wrong.

The complicated algorithms, analytics, optimisations, APIs, routeing, UX and everything behind the scenes is a direct result of the hard work by thousands of individuals involved in computer programming for the last 50 years.

4. The potential of chatbots

The near-future potential is quite apparent. No longer will consumers have to trawl through websites and search engines to find the information they need. Instead, they will be communicating with intelligent chatbots at every stage.

User – "Where is a good place to get coffee near me?
Search Chatbot – "There are three coffee shops near you rated five stars on xxx website".

User – "Add the highest rated coffee shop chatbot to this chat".
Coffee Chatbot – "Hello, this is xxx bot, what's up?"
User – "Send directions to your shop and order a flat white"
Coffee Chatbot – "No problem, directions are in your xxx map, do you want to pay using your xxx wallet?
User – "Yes"
Coffee Chatbot – "Ok, 3.99 has been paid, see you in 12 minutes. We have some delicious muffins just out of the oven too…"

PA Chatbot – "Hi, I noticed you are going for coffee, it looks like it is raining outside, want me to order you a taxi rather than walk?"
User – "Yes, leaving in 2 minutes"
PA Chatbot – "Ok, your driver is called Sammy and the car registration is xxx, he will meet you outside."

This type of chatbot interaction will be commonplace very soon.

Despite how impressive that sounds, it is done with technology that is still new. Communicating with chatbots will not just stop at businesses and brands.
Soon we will be using chatbots to communicate with other machines and connected devices. The internet of things (IoT) will connect everything to everything else. This is already happening with Amazon Echo and Google products.

The PA chatbot will be connected to the user's fridge and will notify him that his wife used up all the milk, and he should get more on the way home from work or offer to order it on Amazon for him.
Alternatively, perhaps the PA chatbot noticed it is raining, opened the garage door and had his autonomous car drive around the front to save him getting wet.

Chatbots, with the natural language and machine learning behind them, will lower our dependence on screens to receive feedback from a machine. Children of the very near future will joke about how we had screens on our phones and couldn't just talk to the machines we use.

CHAPTER - 1
Creating and testing Bots

Here, we are going to discuss step by step processes to successfully create a Bot Application and test it using proper software. At first, we need to learn a few things.

1. Bot Service

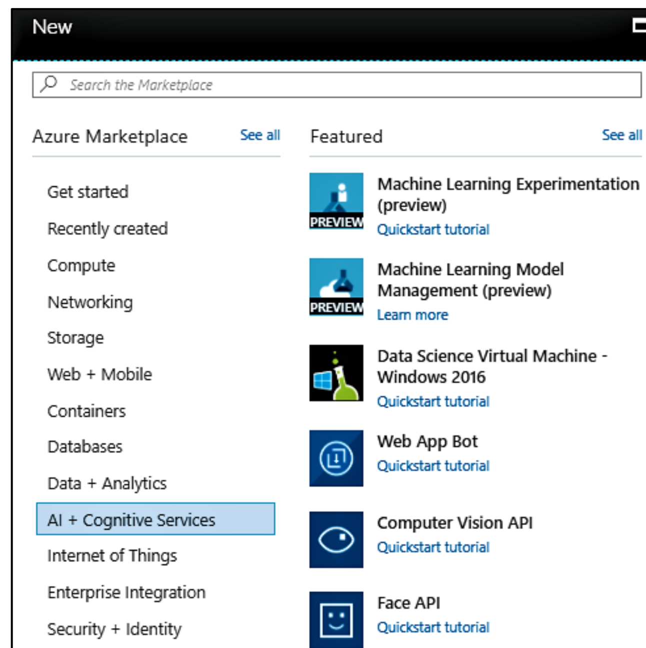
Bot Service provides an integrated environment that is purpose-built for bot development, enabling one to build, connect, test, deploy, and manage intelligent bots, all from one place. Bot Service leverages the Bot Builder SDK with support for .NET and Node.js.

1.1. Log in to Azure

Log in to the Azure portal. https://portal.azure.com

1.2. Create a new bot service

1. Click the New button found on the upper left-hand corner of the Azure portal, then select AI + Cognitive Services > Web App bot.



2. A new blade will open with information about the Web App Bot. Click the Create button to start the bot creation process.
3. In the Bot Service blade, provide the requested information about the bot as specified in the table below the image.

Setting	Suggested value	Description
Bot name	The bot's display name	The display name for the bot that appears in channels and directories. This name can be changed at any time.
Subscription	The user's subscription	Select the Azure subscription the user wants to use.
Resource Group	myResourceGroup	The user can create a new resource group or choose from an existing one.
Location	The default location	Select the geographic location for the resource group. The location choice can be any location listed, though it's often best to choose a location closest to the customer. The location cannot be changed once the bot is created.
Pricing tier	F0	Select a pricing tier. The user may update the pricing tier at any time.
App name	A unique name	The unique URL name of the bot. For example, if the user names his/her bot myawesomebot, then the bot's URL will be http://myawesomebot.azurewebsites.net. The name must use alphanumeric and underscore characters only. There is a 35character limit to this field. The App name cannot be changed once the bot is created.
Bot template	Basic	Choose either C# or Node.js and select the Basic template for this quick start, then click Select. The Basic template creates an echo bot. Learn more about the templates.
App service plan/Location	The app service plan	Select an app service plan location. The location choice can be any location listed, though it's often best to choose a location closest to the customer. (Not available for Functions Bot.)
Azure Storage	The Azure storage account	The user can create a new data storage account or use an existing one. By default, the bot will use Table Storage.
Application Insights	On	Decide if one wants to turn Application Insights On or Off. If he/she selects On, he/she must also specify a regional location. The location choice can be any location listed, though it's often best to choose a location closest to the customer.
Microsoft App ID and password	Auto create App ID and password	Use this option if one needs to manually enter a Microsoft App ID and password. Otherwise, a new Microsoft App ID and password will be created for him/her in the bot creation process.

4. Click Create to create the service and deploy the bot to the cloud. This process may take several minutes.

Confirm that the bot has been deployed by checking the Notifications. The notifications will change from Deployment in progress... to Deployment succeeded. Click Go to resource button to open the bot's resources blade.

1.3. Test the bot

Now that the bot is created, test it in Web Chat. Enter a message and the bot should respond.

1.4. Bot settings overview

In the **Overview** blade, the user can find high level information about his/her bot. For example, the user can see his/her bot's **Subscription ID**, **pricing tier**, and **Messaging endpoint**.

1.5. Bot management

The user can find most of his/her bot's management options under the **BOT MANAGEMENT** section. Below is a list of options to help the user manage his/her bot:

Option	Description
Build	The Build tab provides options for making changes to the bot. This option is not available for **Registration Only Bot**.
Test in Web Chat	Use the integrated Web Chat control to help the user quickly test the bot.
Analytics	If analytics is turned on for the bot, the user can view the analytics data that Application Insights has collected for the bot.
Channels	Configure the channels the bot uses to communicate with users.
Settings	Manage various bot profile settings such as display name, analytics, and messaging endpoint.
Speech priming	Manage the connections between the LUIS app and the Bing Speech service.
Bot Service pricing	Manage the pricing tier for the bot service.

2. App service settings

The **Application Settings** blade contains detailed information about the bot, such as the bot's environment, ID, Application Insights key, Microsoft App ID, and Microsoft App password.

2.1. MicrosoftAppID and MicrosoftAppPassword

The user can find the **MicrosoftAppID** and **MicrosoftAppPassword** for his/her bot in the **Application Settings** blade.

2.2. Edit a bot with online code editor

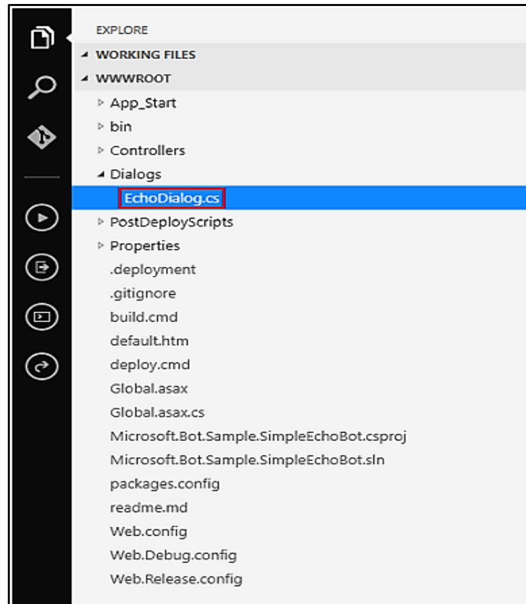
The user can use the online code editor to build the bot without needing an IDE. This topic will show how to open the bot code in the online code editor.

To edit a bot's source code in the online code editor, do the following for the specific type the user has.

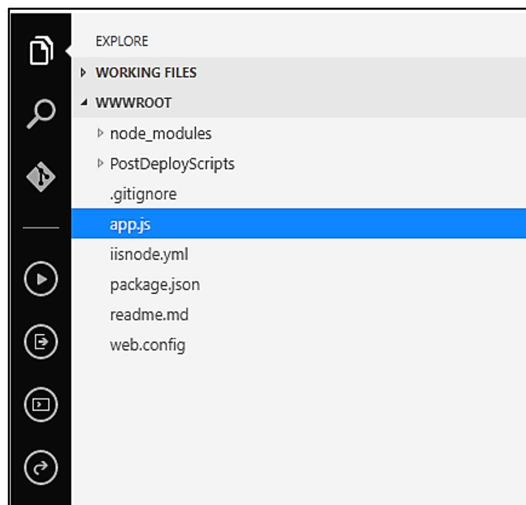
- **Web App Bot**

1. Sign into the Azure portal and open the blade for the bot.
2. Under the **BOT MANAGEMENT** section, Click **Build**.
3. Click **Open online code editor**. This will open the bot's code in a new browser window.Depending on the language of the bot, the file structure under the **WWWRoot** directory will be different.

For example, if the user has a C# bot, the **WWWRoot** may look something like this:



If the user has a Node.js bot, the **WWWRoot** may look something like this:



1. Make code changes. For example, for C# bots, the user can start with the Dialogs/EchoDialog.cs file. For Node.js bots, the user can start with the App.js file.
2. Save the changes. For C# bots that are on a Consumption plan and all Node.js bots, the bot is automatically updated once the source code is saved by clicking the Save button. For C# bots on an App service plan, open the Console blade and send the build.cmd command.
3. Switch back to Azure portal and click Test in Web Chat to test out the changes. If the user already has the **Web Chat** open for this bot, click **Start over** to see the new changes.

- **Functions Bot**

1. Sign into the Azure portal and open the blade for the bot.
2. Under the **BOT MANAGEMENT** section, Click **Build**.
3. Click **Open this bot in Azure Functions**. This will open the bot with the Azure Functions UI.
4. Make code changes. For example, update the function's messages code. The screen shot below shows the Messages code for a Node.js Functions Bot.
5. Save the code changes.
6. Switch back to Azure portal and click **Test in Web Chat** to test out the changes. If the user already has the **Web Chat** open for this bot, click **Start over** to see the new changes.

3. Create a Bot with the Bot Builder SDK for .NET

The Bot Builder SDK for .NET is an easy-to-use framework for developing bots using Visual Studio and Windows. The SDK leverages C# to provide a familiar way for .NET developers to create powerful bots.
This tutorial walks the user through building a bot by using the Bot Application template and the Bot Builder SDK for .NET, and then testing it with the Bot Framework Emulator.

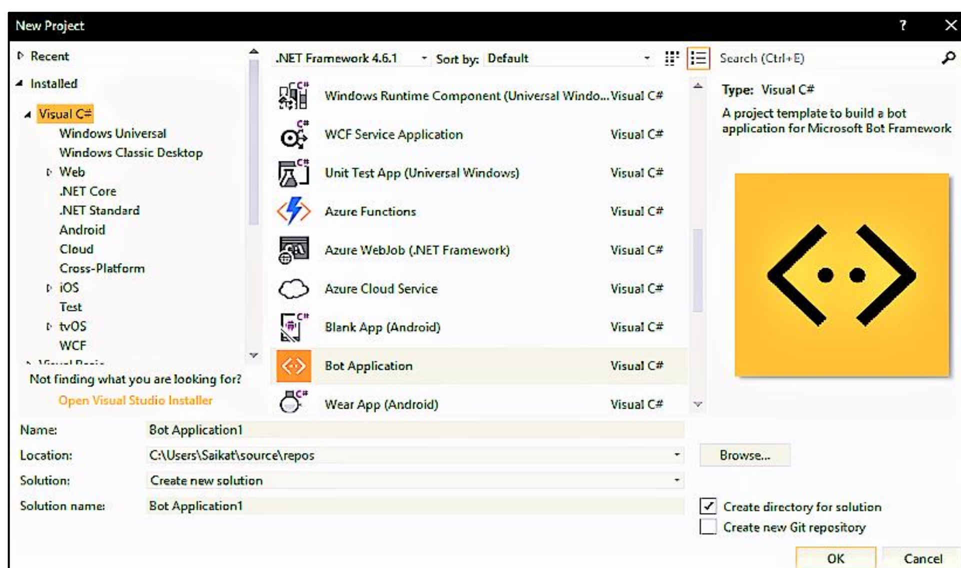
3.1. Prerequisites

Get started by completing the following prerequisite tasks:

1. Install **Visual Studio 2017** for Windows.
2. In Visual Studio, **update all extensions** to their latest versions.
3. Download the **Bot Application, Bot Controller, and Bot Dialog .zip** files. Install the project template by copying `Bot Application.zip` to the Visual Studio 2017 **project templates** directory. Install the item templates by copying `Bot Controller.zip` and `Bot Dialog.zip` to the Visual Studio 2017 **item templates** directory.

3.2. Create the bot

Next, open Visual Studio and create a new C# project. Choose the Bot Application template for the new project.



By using the Bot Application template, the user is creating a project that already contains all of the components that are required to build a simple bot, including a reference to the Bot Builder SDK for .NET, `Microsoft.Bot.Builder`.

3.3. Verify that the project references the latest version of the SDK

1. Right-click on the project and select **Manage NuGet Packages**.
2. In the **Browse** tab, type "Microsoft.Bot.Builder".
3. Locate the Microsoft.Bot.Builder package in the list of search results, and click the **Update** button for that package.
4. Follow the prompts to accept the changes and update the package.

Thanks to the Bot Application template, the project contains all of the code that's necessary to create the bot in this tutorial. The user won't actually need to write any additional code. However, before we move on to testing the bot, take a quick look at some of the code that the Bot Application template provided.

3.4. Explore the code

First, the Post method within **Controllers\MessagesController.cs** receives the message from the user and invokes the root dialog.

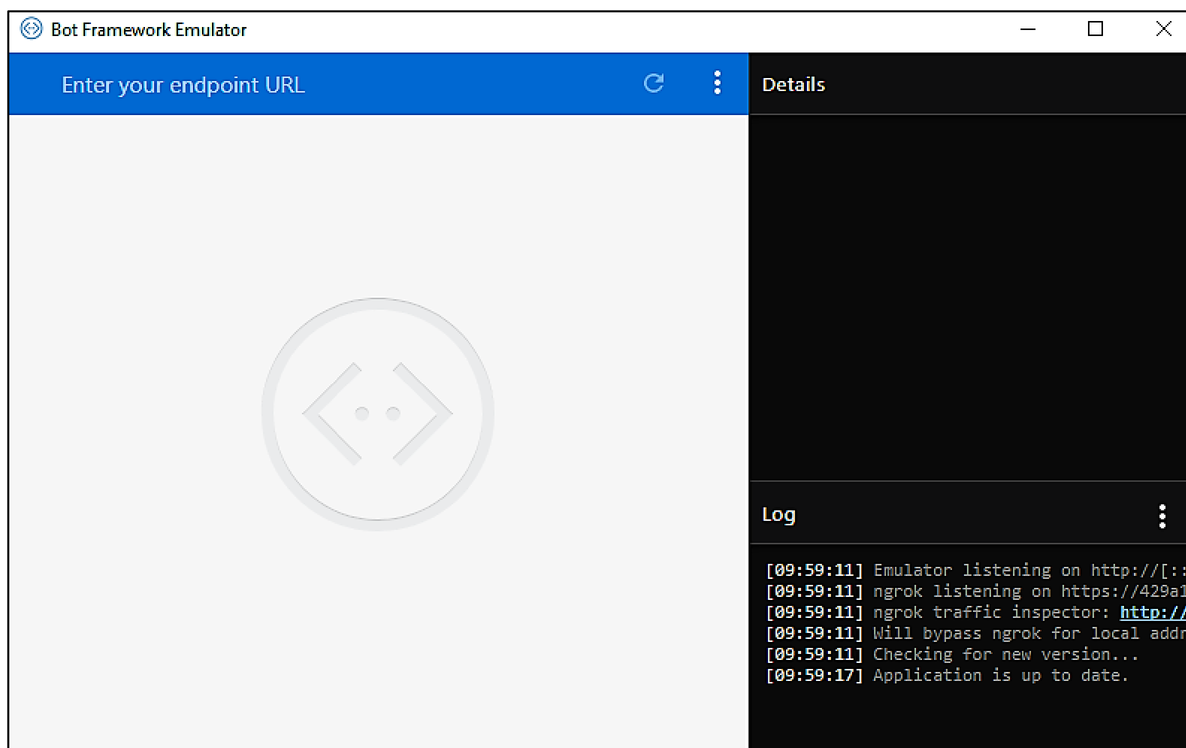
```
[BotAuthentication]
public class MessagesController : ApiController
{
    /// <summary>
    /// POST: api/Messages
    /// Receive a message from a user and reply to it
    /// </summary>
    public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
    {
        if (activity.Type == ActivityTypes.Message)
        {
            await Conversation.SendAsync(activity, () => new Dialogs.RootDialog());
        }
        else
        {
            HandleSystemMessage(activity);
        }
        var response = Request.CreateResponse(HttpStatusCode.OK);
        return response;
    }
    private Activity HandleSystemMessage(Activity message)
    {
            if (message.Type == ActivityTypes.DeleteUserData)
            {
            }
            else if (message.Type == ActivityTypes.ConversationUpdate)
            {
            }
            else if (message.Type == ActivityTypes.ContactRelationUpdate)
            {
            }
            else if (message.Type == ActivityTypes.Typing)
            {
            }
            return null;
        }
    }
}
```

The root dialog processes the message and generates a response. The MessageReceivedAsync method within **Dialogs\RootDialog.cs** sends a reply that echoes back the user's message, prefixed with the text 'the user sent' and ending in the text 'which was ## characters', where ## represents the number of characters in the user's message.

```
[Serializable]
public class RootDialog : IDialog<object>
{
    public Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
        return Task.CompletedTask;
    }

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

        // calculate something for us to return
        int length = (activity.Text ?? string.Empty).Length;

        // return our reply to the user
        await context.PostAsync($"You sent {activity.Text} which was {length} characters");

        context.Wait(MessageReceivedAsync);
    }
}
```

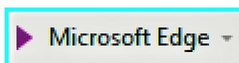
3.5. Test the bot

Next, test the bot by using the Bot Framework Emulator to see it in action. The emulator is a desktop application that lets the user test and debug the bot on localhost or running remotely through a tunnel. First, the user will need to download and install the emulator. After the download completes, launch the executable and complete the installation process. The emulator looks like this:

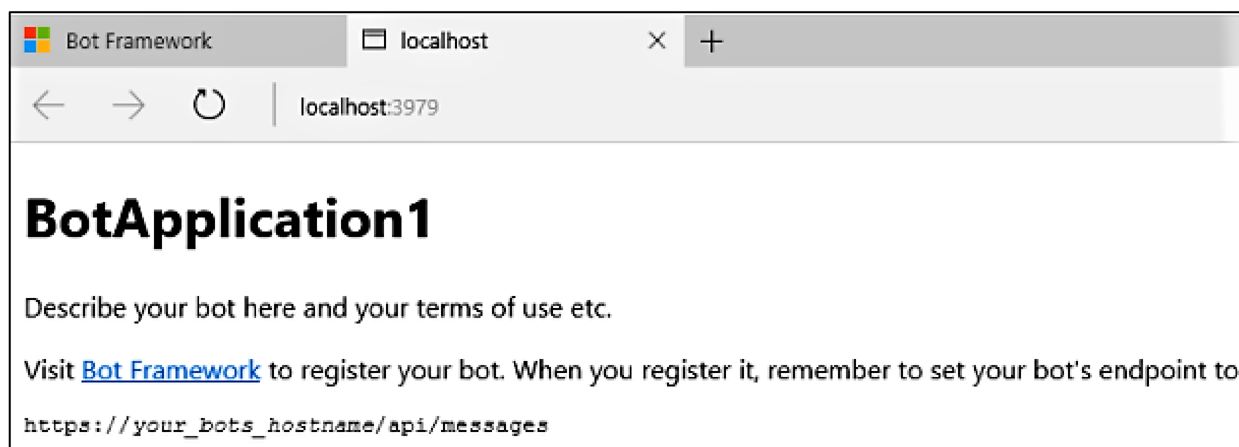


3.6. Start the bot

After installing the emulator, start the bot in Visual Studio by using a browser as the application host. This Visual Studio screenshot shows that the bot will launch in Microsoft Edge when the run button is clicked.



When the user clicks the run button, Visual Studio will build the application, deploy it to localhost, and launch the web browser to display the application's **default.htm** page. For example, here's the application's **default.htm** page shown in Microsoft Edge:



3.7. Start the emulator and connect the bot

At this point, the bot is running locally. Next, start the emulator and then connect to the bot in the emulator:

1. Type `http://localhost:port-number/api/messages` into the address bar, where **port-number** matches the port number shown in the browser where the application is running.
2. Click **Connect**. The user won't need to specify **Microsoft App ID** and **Microsoft App Password**. The user can leave these fields blank for now. He/she will get this information later when he/she registers his/her bot.

3.8. Test the bot code result

Now that the bot is running locally and is connected to the emulator, test the bot by typing a few messages in the emulator. The user should see that the bot responds to each message he/she sends by echoing back the message prefixed with the text 'You sent' and ending with the text 'which was ##characters', where ## is the total number of characters in the message that the user sent.

End of chapter

CHAPTER - 2
Language Understanding (LUIS)

Language Understanding (LUIS) allows the application to understand what a person wants in their own words. LUIS uses machine learning to allow developers to build applications that can receive user input in natural language and extract meaning from it. A client application that converses with the user can pass user input to a LUIS app and receive relevant, detailed information back.
Several Microsoft technologies work with LUIS:

- **Bot Framework** allows a chat bot to talk with a user via text input.
- **Bing Speech API** converts spoken language requests into text. Once converted to text, LUIS processes the requests.

1. What is a LUIS app?

A LUIS app is a domain-specific language model designed by one and tailored to his/her needs. He/she can start with a prebuilt domain model, build his/her own, or blend pieces of a prebuilt domain with his/her own custom information.
A model starts with a list of general user intentions such as **"Book Flight"** or **"Contact Help Desk".** Once the intentions are identified, one should supply example phrases called utterances for the intents. Then he/she label the utterances with any specific details he/she wants LUIS to pull out of the utterance.
Prebuilt domain models include all these pieces for him/her and are a great way to start using LUIS quickly.
After the model is designed, trained, and published, it is ready to receive and process utterances. The LUIS app receives the utterance as an HTTP request and responds with extracted user intentions. The client application sends the utterance and receives LUIS's evaluation as a JSON object. The client app can then take appropriate action.

1.1. Key LUIS concepts

- **Intents:** An intent represents actions the user wants to perform. The intent is a purpose or goal expressed in a user's input, such as booking a flight, paying a bill, or finding a news article. The users define and name intents that correspond to these actions. A travel app may define an intent named "BookFlight."
- **Utterances:** An utterance is text input from the user that the app needs to understand. It may be a sentence, like "Book a ticket to Paris", or a fragment of a sentence, like "Booking" or "Paris flight." Utterances aren't always well-formed, and there can be many utterance variations for a particular intent.
- **Entities:** An entity represents detailed information that is relevant in the utterance. For example, in the utterance "Book a ticket to Paris", "Paris" is a location. By recognizing and labeling the entities that are mentioned in the user’s utterance, LUIS helps the user choose the specific action to take to answer a user's request.

Intent	Sample User Utterance	Entities
BookFlight	"Book a flight to Seattle?"	Seattle
Booking	"Book me a flight ticket"	Flight
Installation	"I want to install Google Chrome"	Google Chrome

1.2. Accessing LUIS

LUIS has two ways to build a model: the Authoring REST-based APIs and the LUIS website. Both methods give the user and his/her collaborators control of his/her LUIS model definition. The user can use either the LUIS website or the Authoring APIs or a combination of both to build the model. This management includes models, versions, collaborators, external APIs, testing, and training.
Once the model is built and published, the user pass the utterance to LUIS and receive the JSON object results with the Endpoint REST-based APIs.

1.3. Author the LUIS model

Begin the LUIS model with the intents the client app can resolve. Intents are just names such as "BookFlight" or "OrderPizza."
After an intent is identified, one needs sample utterances that he/she wants LUIS to map to his/her intent such as **"Buy a ticket to Seattle tomorrow".** Then, label the parts of the utterance that are relevant to his/her app domain as entities and set a type such as date or location.
Generally, an **intent** is used to trigger an action and an **entity** is used as a parameter to execute an action.
For example, a **"BookFlight"** intent could trigger an API call to an external service for booking a plane ticket, which requires entities like the travel destination, date, and airline.

1.4. Identify Entities

Entity identification determines how successfully the end user gets the correct answer. LUIS provides several ways to identify and categorize entities.

- **Prebuilt Entities:** LUIS has many prebuilt domain models including intents, utterances, and prebuilt entities. One can use the prebuilt entities without having to use the intents and utterances of the prebuilt model. The prebuilt entities save time.
- **Custom Entities:** LUIS gives several ways to identify one's own custom entities including simple entities, composite entities, list entities, regular expression entities, and hierarchical entities.
- **Phrases:** LUIS provides phrase lists, which also help identify entities.

1.5. Improve performance

Once the application is published and real user utterances are entered, LUIS uses active learning to improve identification. In the active learning process, LUIS provides real utterances that it is relatively unsure of for one to review. He/she can label them according to intent and entities, retrain, and republish.
This iterative process has tremendous advantages. LUIS knows what it is unsure of, and his/her help leads to the maximum improvement in system performance. LUIS learns quicker and takes the minimum amount of time and effort. LUIS is an active machine learning at its best.

2. Create new app with intents

2.1. Simple app with intents

This simple app has two intentions. The first intent's purpose is to identify when a user wants store information such as hours, and location. The second intent's purpose is to identify every other type of utterance.

Once the type of utterance is identified, LUIS is done. The calling application or chat bot then takes that identification and fulfils the request -- in whatever way the app or chat bot is designed to do.

2.2. Create a new app

1. Log in to the LUIS website. Make sure to log in to the region where the user needs the LUIS endpoints published.
2. On the LUIS website, select **Create new app**.
3. In the pop-up dialog, enter the name MyStore.

Create new app

Name (Required)

MyStore

Culture (Required)

English

** Culture is the language that your app understands and speaks, not the interface language.

Description

Type app description

Done Cancel

4. When that process finishes, the app shows the **Intents** page with the **None** Intent.

Intents

Create new intent | Add prebuilt domain intent | Search intents

Name	Labeled Utterances
None	0

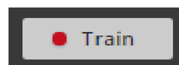
5. Select Create new intent. Enter the new intent name `GetStoreInfo`. This intent should be selected any time a user wants information about the store such as what is sold, what hours it is open, and how to contact.
   By creating an intent, the user is creating a category of information that he/she wants to identify. Giving the category a name allows any other application that uses the LUIS query results to use that category name to find an appropriate answer. LUIS won't answer these questions, only identify what type of information is being asked for in natural language.
6. Add seven utterances to the `GetStoreInfo` intent that is expected from a user to ask for, such as:

Example utterances
When do you open?
What are your hours?
Are you open right now?
What is your phone number?
Can someone call me please?
Where is your store?
How do I get to your store?

7. The LUIS app currently has no utterances for the **None** intent. It needs utterances that the user doesn't want the app to answer, so it needs to have utterances in the **None** intent. Do not leave it empty. Select Intents from the left panel. Select the **None** intent. Add three utterances that the user might enter but are not relevant to the app. If the app is about the store, some good **None** utterances are:

Example utterances
Cancel!
Good bye
What is going on?

   In the LUIS-calling application, such as a chat bot, if LUIS returns the None intent for an utterance, the bot can ask if the user wants to end the conversation. The bot can also give more directions for continuing the conversation if the user doesn't want to end it.
8. In the top right side of the LUIS website, select the Train button.



   Training is complete when one sees the green status bar at the top of the website confirming success.

9. In the top right side of the LUIS website, select the Publish button. Select the Publish to product slot. Publishing is complete when one sees the green status bar at the top of the website confirming success.
10. On the Publish page, select the endpoint link at the bottom of the page. This action opens another browser window with the endpoint URL in the address bar. Go to the end of the URL in the address and enter `When do you open next?`. The last query string parameter is `q`, the utterance query. This utterance is not the same as any of the example utterances in step 4 so it is a good test and should return the `GetStoreInfo` utterances.

- **What has this LUIS app accomplished?**

This app, with just two intents, identified a natural language query that is of the same intention but worded differently.

The JSON result identifies the top scoring intent `GetStoreInfo` with a score of 0.984749258. All scores are between 1 and 0, with the better score being close to 1. The `None` intent's score is 0.2040639, much closer to zero.

- **Where is this LUIS data used?**

LUIS is done with this request. The calling application, such as a chat bot, can take the top Scoring Intent result and either find information (not stored in LUIS) to answer the question or can send the user to the store's website page containing the information. There are other programmatic options for the bot or calling application. LUIS doesn't do that work. LUIS only determines what the user's intention is.

3. Create new app with intents and entities

3.1. Simple app with intents and a simple entity

This simple app has two intents and one entity . This app demonstrates how to pull data out of an utterance. In the utterance, `Send a message telling them to stop`, the intent (primary data) is to send a message and the simple entity (secondary data) is the content of the message, `telling them to stop`.

When the intent and entities of the utterance are identified, LUIS is done. The calling application or chat bot takes that identification and fulfils the request -- in whatever way the app or chat bot is designed to do.

3.2. Create a new app

1. Log in to the LUIS website. Make sure to log into the region where the user needs the LUIS endpoints published.
2. On the LUIS website, select **Create new app**.
3. In the pop-up dialog, enter the name `MyCommunicator`.

Create new app

Name (Required)

MyCommunicator

Culture (Required)

English

** Culture is the language that your app understands and speaks, not the interface language.

Description

Type app description

Done Cancel

When that process finishes, the app shows the Intents page with the None Intent.

- ## Create a new intent

1. On the **Intents** page, select **Create new intent**.
2. Enter the new intent name `SendMessage`. This intent should be selected any time a user wants to send a message.

By creating an intent, user is creating the primary category of information that he/she wants to identify. Giving the category a name allows any other application that uses the LUIS query results to use that category name to find an appropriate answer or take appropriate action. LUIS won't answer these questions, only identify what type of information is being asked for in natural language.

3. Add seven utterances to the `SendMessage` intent that one expects a user to ask for, such as:

Example utterances
Reply with I got your message, I will have the answer tomorrow
Send message of When will you be home?
Text that I am busy
Tell them that it needs to be done today
IM that I am driving and will respond later
Compose message to David that says When was that?
say greg hello

- ## Add utterances to None intent

The LUIS app currently has no utterances for the **None** intent. It needs utterances that the user doesn't want the app to answer, so it has to have utterances in the **None** intent. Do not leave it empty.

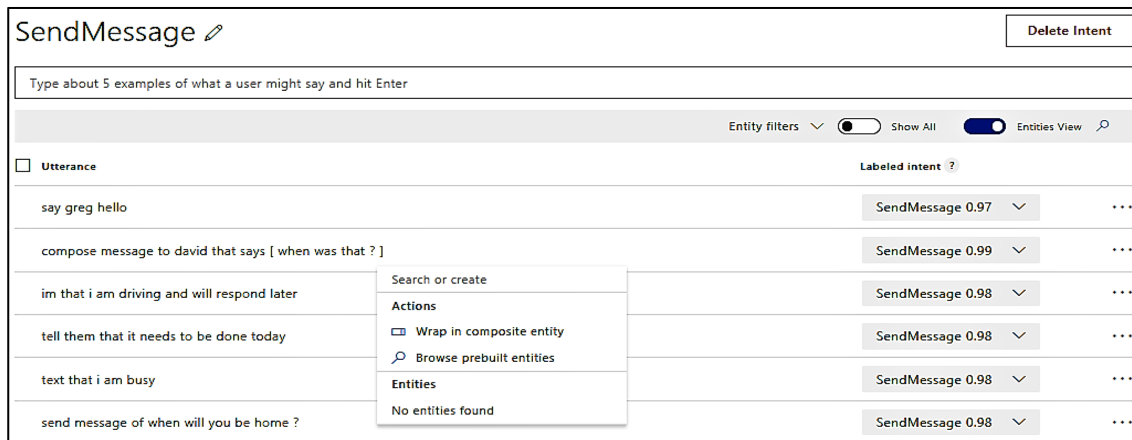
1. Select **Intents** from the left panel.
2. Select the **None** intent.
3. Add three utterances that the user might enter but are not relevant to the app. Some good **None** utterances are:

Example utterances
Cancel!
Good bye
What is going on?

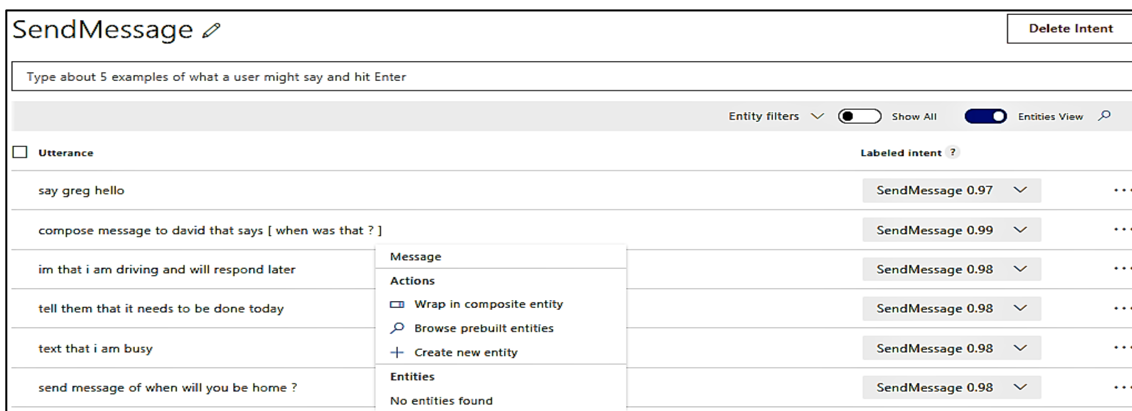
In the LUIS-calling application, such as a chat bot, if LUIS returns the **None** intent for an utterance, the bot can ask if the user wants to end the conversation. The bot can also give more directions for continuing the conversation if the user doesn't want to end it.

- ## Create a simple entity to extract message

1. Select **Intents** from the left menu.
2. Select `SendMessage` from the intents list.
3. In the utterance, `Reply with I got your message, I will have the answer tomorrow`, select the first word of the message body, `I`, and the last word of the message body, `tomorrow`. All these words are selected for the message and a drop-down menu appears with a text box at the top.



4. Enter the entity name `Message` in the text box.



5. Select **Create new entity** in the drop-down menu. The purpose of the entity is to pull out the text that is the body of the message. In this LUIS app, the text message is at the end of the utterance, but the utterance can be any length, and the message can be any length.
6. In the pop-up window, the default entity type is **Simple** and the entity name is Message. Keep these settings and select **Done**.

What type of entity do you want to create?

Entity name (Required)

Message

Entity type (Required)

Simple

A **simple entity** describes a single concept. For example, if the user's intent is GetWeather, you can use City as a simple entity to capture the city for the weather report.

Done Cancel

7. Now that the entity is created, and one utterance is labeled, label the rest of the utterances with that entity. Select an utterance, then select the first and last word of a message. In the drop-down menu, select the entity, Message. The message is now labeled in the entity. Continue to label all message phrases in the remaining utterances.

SendMessage

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Entity filters Show All Entities View

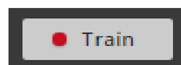
Utterance	Labeled intent ?
say greg Message	SendMessage ...
compose message to david that says Message	SendMessage ...
im that Message	SendMessage ...
tell them that Message	SendMessage ...
text that Message	SendMessage ...
send message of Message	SendMessage ...

The default view of the utterances is **Entities view**. Select the **Entities view** control above the utterances. The **Tokens view** displays the utterance text.

Train the LUIS app

LUIS doesn't know about the changes to the intents and entities (the model), until it is trained.

1. In the top right side of the LUIS website, select the **Train** button.



2. Training is complete when the user sees the green status bar at the top of the website confirming success.

App successfully trained on Mon, 12 Feb 2018 21:59:16 GMT

Publish the app to get the endpoint URL

In order to get a LUIS prediction in a chat bot or other application, the user needs to publish the app.

1. In the top right side of the LUIS website, select the **Publish** button.
2. Select the **Publish to production slot**.
3. Publishing is complete when the user sees the green status bar at the top of the website confirming success.

Query the endpoint with a different utterance

On the **Publish** page, select the **endpoint** link at the bottom of the page.

This action opens another browser window with the endpoint URL in the address bar. Go to the end of the URL in the address and enter `text I'm driving and will be 30 minutes late to the meeting`. The last querystring parameter is `q`, the utterance **query**. This utterance is not the same as any of the labeled utterances so it is a good test and should return the `SendMessage` utterances.

What has this LUIS app accomplished?

This app, with just two intents and one entity, identified a natural language query intention and returned the message data.

The JSON result identifies the top scoring intent `SendMessage` with a score of 0.987501. All scores are between 1 and 0, with the better score being close to 1. The `None` intent's score is 0.111048922, much closer to zero.

The message data has a type, `Message`, as well as a value, `i ' m driving and will be 30 minutes late to the meeting`. The chat bot now has enough information to determine the primary action, `SendMessage`, and a parameter of that action, the text of the message.

End of chapter

CHAPTER - 3

Key concepts in Bot Builder SDK

1. Connector

The Bot Framework Connector provides a single REST API that enables a bot to communicate across multiple channels such as Skype, Email, Slack, and more. It facilitates communication between bot and user by relaying messages from bot to channel and from channel to bot. In the Bot Builder SDK for .NET, the Connector library enables access to the Connector.

2. Activity

The Connector uses an Activity object to pass information back and forth between bot and channel (user). The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel.

3. Dialog

When a user creates a bot using the Bot Builder SDK for .NET, he/she can use dialogs to model a conversation and manage conversation flow. A dialog can be composed of other dialogs to maximize reuse, and a dialog context maintains the stack of dialogs that are active in the conversation at any point in time. A conversation that comprises dialogs is portable across computers, which makes it possible for the bot implementation to scale. In the Bot Builder SDK for .NET, the Builder library enables the user to manage dialogs.

4. FormFlow

A user can use FormFlow within the Bot Builder SDK for .NET to streamline of building a bot that collects information from the user. For example, a bot that takes sandwich orders must collect several pieces of information from the user such as type of bread, choice of toppings, size, and so on. Given basic guidelines, FormFlow can automatically generate the dialogs necessary to manage a guided conversation like this.

5. State

The Bot Builder Framework enables the bot to store and retrieve state data that is associated with a user, a conversation, or a specific user within the context of a specific conversation. State data can be used for many purposes, such as determining where the prior conversation left off or simply greeting a returning user by name. If one stores a user's preferences, he/she can use that information to customize the conversation the next time he/she chats. For example, one might alert the user to a news article about a topic that interests her or alert a user when an appointment becomes available. For testing and prototyping purposes, one can use the Bot Builder Framework's in-memory data storage.

6. Naming conventions

The Bot Builder SDK for .NET library uses strongly-typed, Pascal-cased naming conventions. However, the JSON messages that are transported back and forth over the wire use camel-case naming conventions. For example, the C# property **ReplyToId** is serialized as **replyToId** in the JSON message that's transported over the wire.

7. Messages and Activities

7.1. Activities overview

The Connector uses an Activity object to pass information back and forth between bot and channel (user). The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel.

- ### Activity types in the Bot Builder SDK for .NET

The following activity types are supported by the Bot Builder SDK for .NET.

Activity.Type	Interface	Description
message	ImessageActivity	Represents a communication between bot and user.
conversationUpdate	IconversationUpdateActivity	Indicates that the bot was added to a conversation, other members were added to or removed from the conversation, or conversation metadata has changed.
contactRelationUpdate	IContactRelationUpdateActivity	Indicates that the bot was added or removed from a user's contact list.
typing	ItypingActivity	Indicates that the user or bot on the other end of the conversation is compiling a response.
ping	n/a	Represents an attempt to determine whether a bot's endpoint is accessible.
deleteUserData	n/a	Indicates to a bot that a user has requested that the bot delete any user data it may have stored.
endOfConversation	IendOfConversationActivity	Indicates the end of a conversation.
event	IeventActivity	Represents a communication sent to a bot that is not visible to the user.
invoke	IinvokeActivity	Represents a communication sent to a bot to request that it perform a specific operation. This activity type is reserved for internal use by the Microsoft Bot Framework.
messageReaction	ImessageReactionActivity	Indicates that a user has reacted to an existing activity. For example, a user clicks the "Like" button on a message.

7.2. Add speech to messages

If one is building a bot for a speech-enabled channel such as Cortana, he/she can construct messages that specify the text to be spoken by the bot. He/she can also attempt to influence the state of the client's microphone by specifying an input hint to indicate whether the bot is accepting, expecting, or ignoring user input.

- ### Specify text to be spoken by the bot

Using the Bot Builder SDK for .NET, there are multiple ways to specify the text to be spoken by the bot on a speech-enabled channel.

One can set the `Speak` property of the message, call the `IDialogContext.SayAsync()` method, or specify prompt options `speak` and `retrySpeak` when sending a message using a built-in prompt.

- **IMessageActivity.Speak**

If one is creating a message and setting its individual properties, he/she can set the Speak property of the message to specify the text to be spoken by the bot. The following code example creates a message that specifies text to be displayed and text to be spoken and indicates that the bot is accepting user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.AcceptingInput;
await connector.Conversations.ReplyToActivityAsync(reply);
```

- **IDialogContext.SayAsync()**

If one is using dialogs, he/she can call the SayAsync() method to create and send a message that specifies the text to be spoken, in addition to the text to be displayed and other options. The following code example creates a message that specifies text to be displayed and text to be spoken.

```
await context.SayAsync(text:  "Thank  you  for  your  order!", speak:  "Thank  you  for  your
order!");
```

- **Prompt options**

Using any of the built-in prompts, one can set the options speak and retrySpeak to specify the text to be spoken by the bot. The following code example creates a prompt that specifies text to be displayed, text to be spoken initially, and text to be spoken after waiting a while for user input. It uses SSML formatting to indicate that the word "sure" should be spoken with a moderate amount of emphasis.

```
PromptDialog.Confirm(
                Context: context,
                Resume: AfterResetAsync,
                promptOptions: new PromptOptions<string>(prompt: "Are you sure that you want
                to cancel this transaction?", speak: "Are you <emphasis level=\"moderate\">
                sure </emphasis> that you want to cancel this transaction?",retrySpeak: "Are
                you <emphasis  level=\"moderate\">sure</emphasis> that you want to cancel this
                transaction?")
                );
```

- **Speech Synthesis Markup Language (SSML)**

To specify text to be spoken by the bot, one can use either a plain text string or a string that is formatted as Speech Synthesis Markup Language (SSML), an XML-based markup language that enables him/her to control various characteristics of the bot's speech such as voice, rate, volume, pronunciation, pitch, and more.

- **Input hints**

When one sends a message on a speech-enabled channel, one can attempt to influence the state of the client's microphone by also including an input hint to indicate whether the bot is accepting, expecting, or ignoring user input.

7.3. Add input hints to messages

By specifying an input hint for a message, one can indicate whether his/her bot is accepting, expecting, or ignoring user input after the message is delivered to the client. For many channels, this enables clients to set the state of user input controls accordingly. For example, if a message's input hint indicates that the bot is ignoring user input, the client may close the microphone and disable the input box to prevent the user from providing input.

- **Accepting input**

To indicate that the bot is passively ready for input but is not awaiting a response from the user, set the message's input hint to `InputHints.AcceptingInput`. On many channels, this will cause the client's input box to be enabled and microphone to be closed, but still accessible to the user. For example, Cortana will open the microphone to accept input from the user if the user holds down the microphone button. The following code example creates a message that indicates the bot is accepting user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.AcceptingInput;
```

- **Expecting input**

To indicate that the bot is awaiting a response from the user, set the message's input hint to `InputHints.ExpectingInput`. On many channels, this will cause the client's input box to be enabled and microphone to be open. The following code example creates a message that indicates the bot is expecting user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.ExpectingInput;
```

- **Ignoring input**

To indicate that the bot is not ready to receive input from the user, set the message's input hint to `InputHints.IgnorningInput`. On many channels, this will cause the client's input box to be disabled and microphone to be closed. The following code example creates a message that indicates the bot is ignoring user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.IgnoringInput;
```

- **Default values for input hint**

If one does not set the input hint for a message, the Bot Builder SDK will automatically set it for the user by using this logic:

- If the bot sends a prompt, the input hint for the message will specify that the bot is expecting input.
- If the bot sends single message, the input hint for the message will specify that the bot is accepting input.
- If the bot sends a series of consecutive messages, the input hint for all but the final message in the series will specify that the bot is ignoring input, and the input hint for the final message in the series will specify that the bot is accepting input.

8. Dialogs

8.1. Dialogs in the Bot Builder SDK for .NET

When one creates a bot using the Bot Builder SDK for .NET, he/she can use dialogs to model a conversation and manage conversation flow. Each dialog is an abstraction that encapsulates its own state in a C# class that implements IDialog. A dialog can be composed with other dialogs to maximize reuse, and a dialog context maintains the stack of dialogs that are active in the conversation at any point in time.
A conversation that comprises dialogs is portable across computers, which makes it possible for the bot implementation to scale. When one uses dialogs in the Bot Builder SDK for .NET, conversation state (the dialog stack and the state of each dialog in the stack) is automatically stored to his/her choice of state data storage. This enables the bot's service code to be stateless, much like a web application that does not need to store session state in web server memory.

Consider this echo bot example, which describes how to change the bot that uses dialogs to exchange messages with the user.

- **MessagesController.cs**

In the Bot Builder SDK for .NET, the Builder library enables one to implement dialogs. To access the relevant classes, import the Dialogs namespace.

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

Next, add this EchoDialog class to **MessagesController.cs** to represent the conversation.

```
[Serializable]
public class EchoDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
    }

    public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity>
    argument)
    {
        var message = await argument;
        await context.PostAsync("You said: " + message.Text);
        context.Wait(MessageReceivedAsync);
    }
}
```

Then, wire the EchoDialog class to the Post method by calling the Conversation.SendAsync method.

```
public virtual async Task<HttpResponseMessage> Post([FromBody] Activity activity)
{
    // Check if activity is of type message
    if (activity != null && activity.GetActivityType() == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new EchoDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }
    return new HttpResponseMessage(System.Net.HttpStatusCode.Accepted);
}
```

- **Implementation details**

The Post method is marked async because Bot Builder uses the C# facilities for handling asynchronous communication. It returns a Task object, which represents the task that is responsible for sending replies to the passed-in message. If there is an exception, the Task that is returned by the method will contain the exception information.

The Conversation.SendAsync method is key to implementing dialogs with the Bot Builder SDK for .NET. It follows the dependency inversion principle and performs these steps:

1. Instantiates the required components
2. Deserializes the conversation state (the dialog stack and the state of each dialog in the stack) from IBotDataStore
3. Resumes the conversation process where the bot suspended and waits for a message
4. Sends the replies
5. Serializes the updated conversation state and saves it back to IBotDataStore

When the conversation first starts, the dialog does not contain state, so Conversation.SendAsync constructs EchoDialog and calls its StartAsync method. 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).

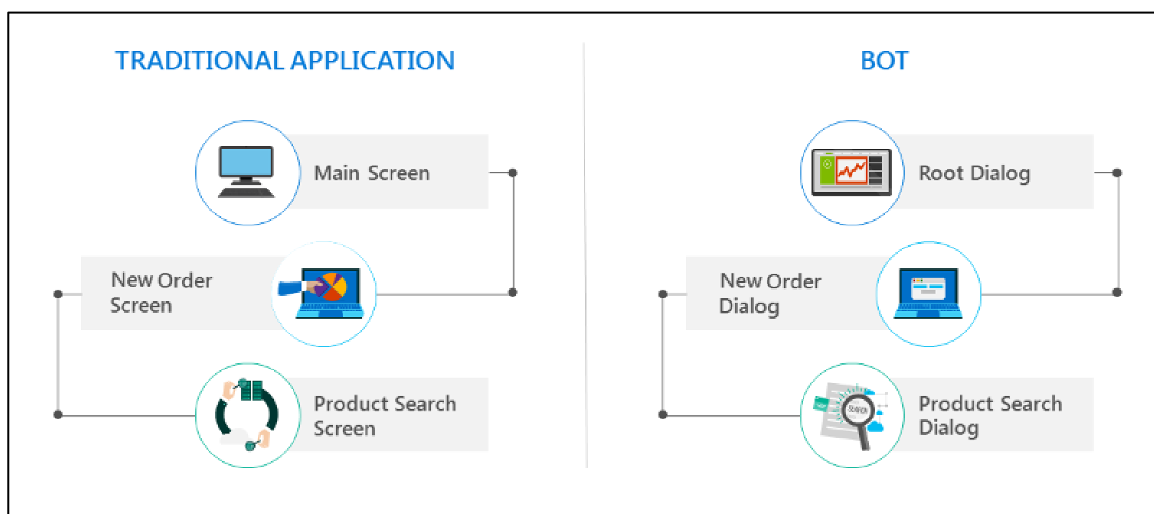
The MessageReceivedAsync method waits for a message, posts a response, and waits for the next message. Every time IDialogContext.Wait is called, the bot enters a suspended state and can be restarted on any computer that receives the message.

A bot that's created by using the code samples above will reply to each message that the user sends by simply echoing back the user's message prefixed with the text 'You said: '. Because the bot is created using dialogs, it can evolve to support more complex conversations without having to explicitly manage state.

8.2. Manage Conversation flow

- **Manage conversation flow with dialogs**

This diagram shows the screen flow of a traditional application compared to the dialog flow of a bot.



In a traditional application, everything begins with the **main screen**. The **main screen** invokes the **new order screen**. The **new order screen** remains in control until it either closes or invokes other screens. If the **new order screen** closes, the user is returned to the **main screen**.

In a bot, everything begins with the **root dialog**. The **root dialog** invokes the **new order dialog**. At that point, the **new order dialog** takes control of the conversation and remains in control until it either closes or invokes other dialogs. If the **new order dialog** closes, control of the conversation is returned back to the **root dialog**.
This article describes how to model this conversation flow by using dialogs and the Bot Builder SDK for .NET.

▪ Invoke the root dialog

First, the bot controller invokes the "root dialog". The following example shows how to wire the basic HTTP GET call to a controller and then invoke the root dialog.

```
public class MessagesController : ApiController
{
    public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
    {
            // Redirect to the root dialog.
            await Conversation.SendAsync(activity, () => new RootDialog());
            ...
    }
}
```

▪ Invoke the 'New Order' dialog

Next, the root dialog invokes the 'New Order' dialog.

```
[Serializable]
public class RootDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        // Root dialog initiates and waits for the next message from the user.
        // When a message arrives, call MessageReceivedAsync.
        context.Wait(this.MessageReceivedAsync);
    }

    public virtual async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<IMessageActivity> result)
    {
        var message = await result; // We've got a message!
        if (message.Text.ToLower().Contains("order"))
        {
            // User said 'order', so invoke the New Order Dialog and wait for it to finish.
            // Then, call ResumeAfterNewOrderDialog.
            await context.Forward(new NewOrderDialog(), this.ResumeAfterNewOrderDialog, message,
CancellationToken.None);
        }
        // User typed something else; for simplicity, ignore this input and wait for the next
message.
        context.Wait(this.MessageReceivedAsync);
    }

    private async Task ResumeAfterNewOrderDialog(IDialogContext context, IAwaitable<string>
result)
    {
        // Store the value that NewOrderDialog returned.
        // (At this point, new order dialog has finished and returned some value to use within the
root dialog.)
        var resultFromNewOrder = await result;

        await context.PostAsync($"New order dialog just told me this: {resultFromNewOrder}");

        // Again, wait for the next message from the user.
        context.Wait(this.MessageReceivedAsync);
    }
}
```

- **Dialog lifecycle**

When a dialog is invoked, it takes control of the conversation flow. Every new message will be subject to processing by that dialog until it either closes or redirects to another dialog.

In C#, one can use context.Wait() to specify the callback to invoke the next time the user sends a message. To close a dialog and remove it from the stack (thereby sending the user back to the prior dialog in the stack), use context.Done(). One must end every dialog method with context.Wait(), context.Fail(), context.Done(), or some redirection directive such as context.Forward() or context.Call(). A dialog method that does not end with one of these will result in an error (because the framework does not know what action to take the next time the user sends a message).

8.3. Scorable Dialogs

- **Global message handlers using scorables**

Users attempt to access certain functionality within a bot by using words like "help," "cancel," or "start over" in the middle of a conversation when the bot is expecting a different response. One can design the bot to gracefully handle such requests using scorable dialogs.

Scorable dialogs monitor all incoming messages and determine whether a message is actionable in some way. Messages that are scorable are assigned a score between [0 – 1] by each scorable dialog. The scorable dialog that determines the highest score is added to the top of the dialog stack and then hands the response to the user. After the scorable dialog completes execution, the conversation continues from where it left off.

Scorables enable one to create more flexible conversations by allowing the users to 'interrupt' the normal conversation flow one finds in regular dialogs.

- **Create a scorable dialog**

First, define a new dialog. The following code uses a dialog that is derived from the IDialog interface.

```
public class SampleDialog : IDialog<object>
{
   public async Task StartAsync(IDialogContext context)
   {
       await context.PostAsync("This is a Sample Dialog which is Scorable. Reply with anything to
       return to the prior dialog.");

       context.Wait(this.MessageReceived);
   }

   private async Task MessageReceived(IDialogContext context, IAwaitable<IMessageActivity>
   result)
   {
       var message = await result;

       if ((message.Text != null) && (message.Text.Trim().Length > 0))
       {
           context.Done<object>(null);
       }
       else
       {
           context.Fail(new Exception("Message was not a string or was an empty string."));
       }
    }
}
```

To make a scorable dialog, create a class that inherits from the ScorableBase abstract class. The following code shows a SampleScorable class.

```
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Builder.Internals.Fibers;
using Microsoft.Bot.Builder.Scorables.Internals;

public class SampleScorable : ScorableBase<IActivity, string, double>
{
    private readonly IDialogTask task;

    public SampleScorable(IDialogTask task)
    {
        SetField.NotNull(out this.task, nameof(task), task);
    }
}
```

The ScorableBase abstract class inherits from the IScorable interface. One will need to implement the following IScorable methods in the class:

- PrepareAsync is the first method that is called in the scorable instance. It accepts incoming message activity, analyzes and sets the dialog's state, which is passed to all the other methods of the IScorable interface.

```
protected override async Task<string> PrepareAsync(IActivity item, CancellationToken token)
{
    // TODO: insert your code here
}
```

- GetScore will only trigger if HasScore returns true. One will provision the logic in this method to determine the score for a message between 0 - 1.

```
protected override double GetScore(IActivity item, string state)
{
    // TODO: insert your code here
}
```

- In the PostAsync method, define core actions to be performed for the scorable class. All scorable dialogs will monitor incoming messages, and assign scores to valid messages based on the scorables' GetScore method. The scorable class which determines the highest score (between 0 - 1.0) will then trigger that scorable's PostAsync method.

```
protected override Task PostAsync(IActivity item, string state, CancellationToken token)
{
    //TODO: insert your code here
}
```

- DoneAsync is called after the scoring process is complete. Use this method to dispose of any scoped resources.

```
protected override Task DoneAsync(IActivity item, string state, CancellationToken token)
{
    //TODO: insert your code here
}
```

- ## Create a module to register the IScorable service

Next, define a Module that will register the SampleScorable class as a component. This will provision the IScorable service.

```
public class GlobalMessageHandlersBotModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        base.Load(builder);

        builder
            .Register(c => new SampleScorable(c.Resolve<IDialogTask>()))
            .As<IScorable<IActivity, double>>()
            .InstancePerLifetimeScope();
    }
}
```

- ## Register the module

The last step in the process is to apply the SampleScorable to the bot's Conversation Container. This will register the scorable service within the Bot Framework's message handling pipeline. The following code shows to update the Conversation.Container within the bot app's initialization in **Global.asax.cs**

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        this.RegisterBotModules();
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }

    private void RegisterBotModules()
    {
        var builder = new ContainerBuilder();
        builder.RegisterModule(new ReflectionSurrogateModule());

        //Register the module within the Conversation container
        builder.RegisterModule<GlobalMessageHandlersBotModule>();

        builder.Update(Conversation.Container);
    }
}
```

End of chapter

CHAPTER - 4

Including Speech Support in Bots

In the previous chapter, we created the app using LUIS. On this chapter, we are going to give speech support to the existing app by registering our bot in Microsoft Azure Portal by creating a **‘Web App Bot’.**

When we are done creating the **Web App Bot**, we receive a **Microsoft App ID** and **Microsoft App Password**, and these two can be found in the **‘Application Settings’** option.

BotId	HelperApp1	☐ Slot setting
MicrosoftAppId	4472925a-e0c5-4759-aff1-5fdc9031fb2d	☐ Slot setting
MicrosoftAppPassword	eaveVWW304(!{?jcoZSYS57	☐ Slot setting
BotStateEndpoint		☐ Slot setting
BotOpenIdMetadata		☐ Slot setting
UseTableStorageForConversationState	true	☐ Slot setting
BotDevAppInsightsKey	419c08ab-b0fb-43ab-b7b1-306582fcf51d	☐ Slot setting
BotDevAppInsightsName	HelperApp1amh6kj	☐ Slot setting
BotDevAppInsightsAppId	1034e38e-fff5-4838-bcd0-3d4ce1c07ac9	☐ Slot setting
LuisAPIKey	9a353943d75b4e5689cd9a2dc3ab50c8	☐ Slot setting
LuisAppId	f1998763-012c-4f93-8cd3-4712f1acd48c	☐ Slot setting
LuisAPIHostName	westus.api.cognitive.microsoft.com	☐ Slot setting

We can also find more three important fields, they are:

1. **LuisAPIKey:** It is the luis subscription key.
2. **LuisAppId:** It is the luis app id to which this web app bot is linked.
3. **LuisAPIHostName:** It is the region where the app is published.

Now, the bot is registered with the given ID and Password, so we can also use speech now. We just need to give the ID and Password to the emulator fields when we run the app and then we can give speech as inputs.

The app ID and app Password are auto-generated. But the password can be changed. Now, we are going to see how to do it.

1. How to change the auto-generated app password

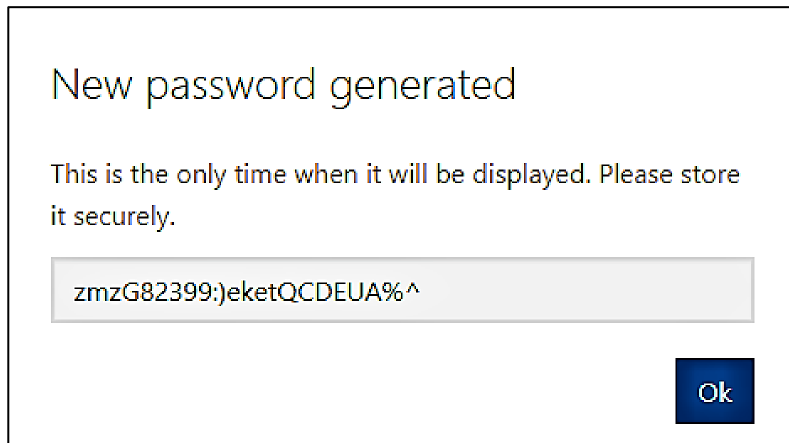
The Microsoft App Password is automatically generated. But one can also change the auto-generated password, but it will randomly regenerate if one wants to change it.

In order to change the password,

1. Click on the 'Settings' option, then under the 'Configuration' menu, click on the 'Manage' button situated beside the Microsoft App ID.

2. A page will open, where all the information of the app will be displayed along with the password. In this page, under the 'Application Secrets' menu, select the option 'Generate New Password'.

3. After clicking on the button, a small alert box will be opened carrying the new password along with a message "New password generated". But the password will be shown only once, so the password must be copied as soon as it is created and paste it in the 'Microsoft App Password' field.

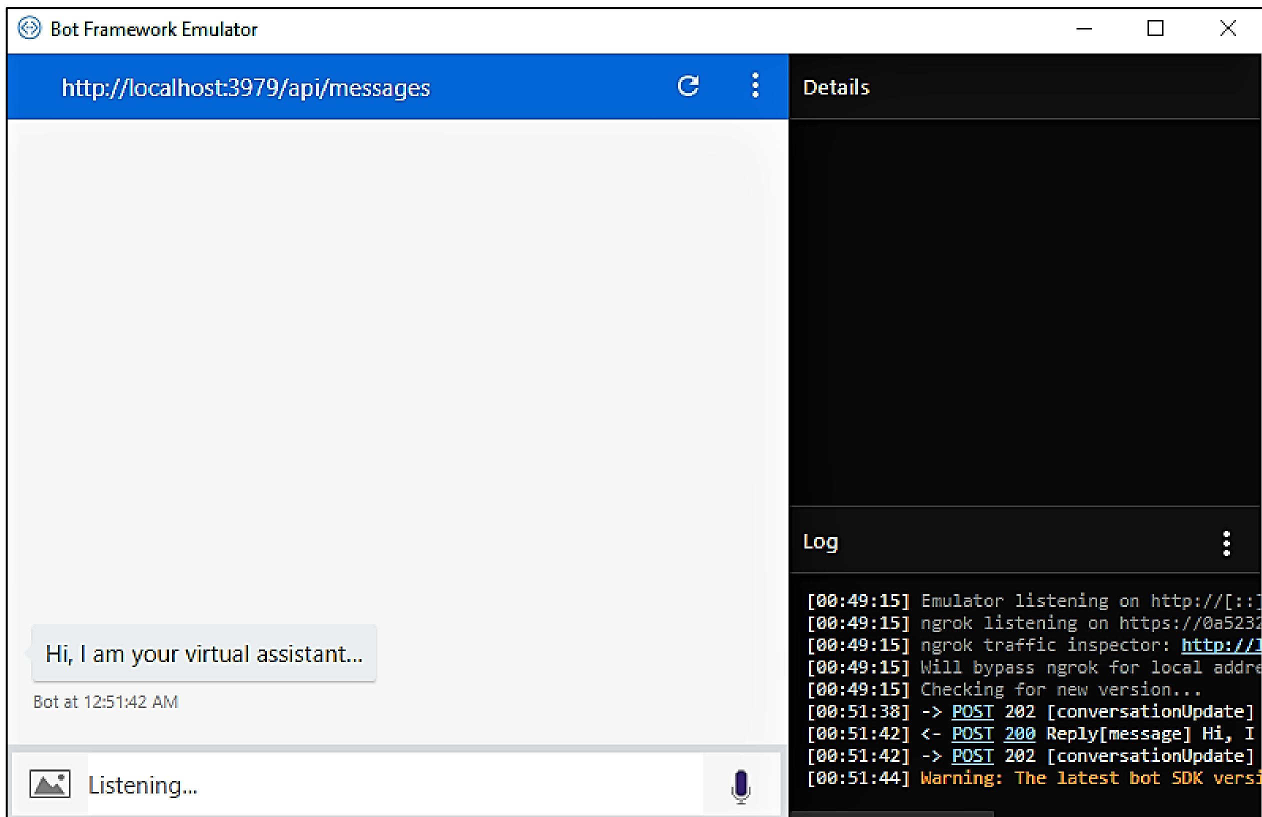


There it is, the app is successfully registered with an app ID and app Password. So, now, text as well as speech will work.

2. How to give speech input to the emulator

Now that we know how to include speech to our bot, we need to know how we can give speech input to the running app in the emulator.

To give speech input, one just needs to give the correct app credentials in the Microsoft App ID and Microsoft App Password fields in the emulator and click on the speech icon situated at the lower-right corner of the emulator, and then wait a little, after 2/3 seconds, the emulator will prompt us that it is listening, then just start talking, the words will automatically be transformed into their text equivalents and we will be able to see the message we just sent when we stop talking.



End of chapter

CHAPTER - 5
App Specifications

The app is just a demo of how the LUIS works and depending on the intents and entities, how the Bot responses back to the user.

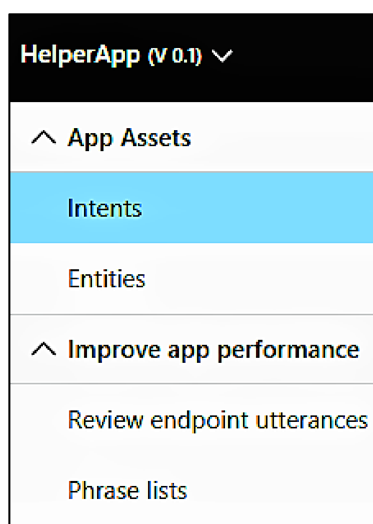
1. Name of the app: HelperApp.

2. Functions

This app has four main functions. They are:

1. This app helps to book tickets of movie, flight, or railway.
2. This app helps to shop online in online shopping stores.
3. This app helps to change or reset the passwords of different platforms.
4. This app helps to install different softwares.

In order to create this app, I have logged into my profile and created an app named 'HelperApp'.



3. Intents

I have created four main intents, they are:

1. Booking
2. Installation
3. PassChange
4. Shopping

Booking: This intent is used to guide the user to book movie, flight or railway tickets.
Installation: This intent is used to search the software the user wants to install.
Passchange: This intent is used to guide the user how to change or reset the passwords of different platforms.
Shopping: This intent is used to help the user to shop online in different online stores like flipkart, amazon, etc.

To make the app more interactive, I have created some basic intents which is often very likely to occur depending on the user's behaviour. They are:

1. Greet
2. BidBye
3. ThanksGiving
4. Help

Greet: This intent is used to greet the user with some randomly generated responses so that the user feels like he/she is having a conversation with another human.

Bidbye: This intent is used to bid bye to the user when the user wants nothing more from the bot. Its purpose is the same as the intent Greet.

ThanksGiving: This intent is used to welcome the user when the user shows gratitude to the bot. Here, also some randomly generated responses come into play. Purpose is same as Greet and Bidbye.

Help: This intent is used to help the user at any time to remind them of what the bot can do and its limitations.

None: This is the default intent. It is triggered whenever the user utterances do not match with any other intents.

App Assets
Intents
Entities
Improve app performance
Review endpoint utterances
Phrase lists

Intents

Create new intent | Add prebuilt domain intent | Search intents

Name	Utterances
BidBye	6
Booking	20
Greet	4
Help	4
Installation	10
None	3
PassChange	12
Shopping	15
ThanksGiving	5

4. Entities

I have created four entities. They are:

1. BookingOption
2. PasswordOption
3. ShoppingSite
4. Software

- **BookingOption:** It is the object the user wants to book tickets for.
  Values: Movie, Flight, and Railway.
- **PasswordOption:** It is the platform in which the password needs to be changed.
  Values: Gmail account, Facebook account, Github, Skype, System, etc.
- **ShoppingSite:** It is the shopping site in which the user wants to shop.
  Values: flipkart, amazon, myntra, jabong, etc.
- **Software:** It is the software the user wants to install.
  Values: Google Chrome, Microsoft Visual Studio, etc.

App Assets
Intents
Entities
Improve app performance
Review endpoint utterances
Phrase lists

Entities

Create new entity | Manage prebuilt entities | Add prebuilt domain entity | Search entities

Name	Type	Labeled Utterances
BookingOption	Simple	16
PasswordOption	Simple	9
ShoppingSite	Simple	12
Software	Simple	8

5. Utterances

There are plenty of utterances for each of the intents and these utterances have entities within them.

- **Functional Intents**

▪ **Booking**

There are plenty of utterances for this intent. The entity **'BookingOption'** is used in the utterances. It recognizes whether any user wants to book movie, flight, or railway tickets.

Booking | Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
book some BookingOption tickets for me	Booking 0.99
i want some tickets	Booking 0.96
i want some tickets of a BookingOption	Booking 0.98
i want some tickets of BookingOption	Booking 0.97
i want to book BookingOption tickets	Booking 0.99
i want to book another ticket	Booking 0.95
i want to book BookingOption tickets	Booking 0.99
i want to book BookingOption tickets	Booking 0.99
i want to book BookingOption tickets	Booking 0.99
i want to book one more BookingOption ticket	Booking 0.96

Booking | Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
i want to book BookingOption tickets	Booking 0.99
i want to book some BookingOption tickets	Booking 1.00
i want to book some BookingOption tickets	Booking 1.00
i want to book some BookingOption tickets	Booking 0.99
i want to book some tickets	Booking 1.00
i want to book BookingOption tickets	Booking 0.99
i want to buy some tickets	Booking 0.96
i want to buy some tickets of a BookingOption	Booking 0.99
i want to buy some tickets of BookingOption	Booking 0.98
book me a BookingOption ticket	Booking 0.94

Installation

There are plenty of utterances for this intent. The entity **'Software'** is used in the utterances to recognize the name of the software the user wants.

Installation

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
i want to install a software	Installation 0.99
i want to install Software	Installation 0.94
i want to install Software	Installation 0.96
i want to install Software	Installation 0.96
i want to install Software	Installation 0.96
i want to install Software	Installation 0.97
i want to install Software	Installation 0.95
install Software for me	Installation 0.98
install Software for me	Installation 0.95
software installation	Installation 0.88

Passchange

There are plenty of utterances as well for this intent. The entity **'PasswordOption'** is used in the utterances to get the platform in which the user wants to change or reset the password.

PassChange

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
can you show me how to change the password of PasswordOption	PassChange 0.98
change password	PassChange 0.98
change the PasswordOption password	PassChange 0.98
change the PasswordOption password	PassChange 0.98
i want to change PasswordOption password	PassChange 0.98
i want to change PasswordOption password	PassChange 0.95
i want to change PasswordOption password	PassChange 0.97
i want to change my PasswordOption password	PassChange 0.98
i want to change my PasswordOption password	PassChange 0.98
i want to change my password	PassChange 0.98

PassChange

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
i want to change my PasswordOption password	PassChange 0.98
password change	PassChange 0.98

▪ Shopping

There are plenty of utterances for this intent as well. The entity **'ShoppingSite'** is used in the utterances to get the online shopping site in which the user wants to shop online.

Shopping

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors Entity Entities view

Utterance	Labeled intent
i want shopping from ShoppingSite	Shopping 1.00
i want some shopping	Shopping 0.95
i want to buy something from ShoppingSite	Shopping 0.97
i want to buy something in ShoppingSite	Shopping 0.97
i want to buy something on ShoppingSite	Shopping 0.96
i want to do some shopping	Shopping 0.97
i want to do some shopping on ShoppingSite	Shopping 1.00
i want to return something in ShoppingSite	Shopping 0.97
i want to return something on ShoppingSite	Shopping 0.97
i want to shop from ShoppingSite online	Shopping 0.98

Shopping

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors Entity Entities view

Utterance	Labeled intent
i want to shop from ShoppingSite online	Shopping 1.00
i want to shop from ShoppingSite	Shopping 0.93
i want to shop online	Shopping 0.93
shopping in ShoppingSite	Shopping 1.00
want to shop in ShoppingSite	Shopping 0.94

• Non-functional Intents

▪ Greet

Some basic utterances are used in this intent. There is no entity for this intent.

Greet

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors Entity Entities view

Utterance	Labeled intent
hello	Greet 0.90
hey	Greet 0.97
hi	Greet 0.97
yo	Greet 0.97

- **BidBye**

Some basic utterances are used in this intent. There is no entity for this intent.

BidBye

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
goodbye	BidBye 0.92
i want nothing more	BidBye 0.85
ok bye	BidBye 0.93
ok finish now	BidBye 0.91
see you later	BidBye 0.89
stop	BidBye 0.92

- **ThanksGiving**

Some basic utterances are used in this intent. There is no entity for this intent.

ThanksGiving

Delete Intent

Type about 5 examples of what a user might say and hit Enter

Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
good work	ThanksGiving 0.89
thank you	ThanksGiving 0.97
thank you for the help	ThanksGiving 0.85
thanks	ThanksGiving 0.90
well done	ThanksGiving 0.89

- **Help**

Some basic utterances are used in this intent. There is no entity for this intent.

Help

Delete Intent

Type about 5 examples of what a user might say and hit Enter

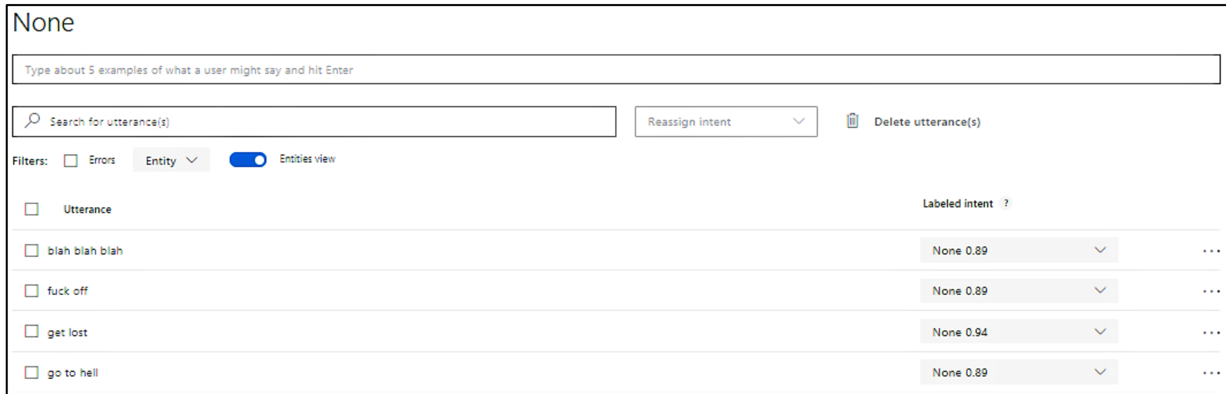
Search for utterance(s) | Reassign intent | Delete utterance(s)

Filters: Errors | Entity | Entities view

Utterance	Labeled intent
help	Help 0.92
help me	Help 0.92
show me available options	Help 0.88
what can you do	Help 0.87

- **Default Intent**
  - **None**

This is the default intent. We do not have to add any utterances here. But sometimes, to improve performance, some utterances can be added to this intent. The utterances should be completely different from the utterances used in other intents and should not have any link with the app functions.

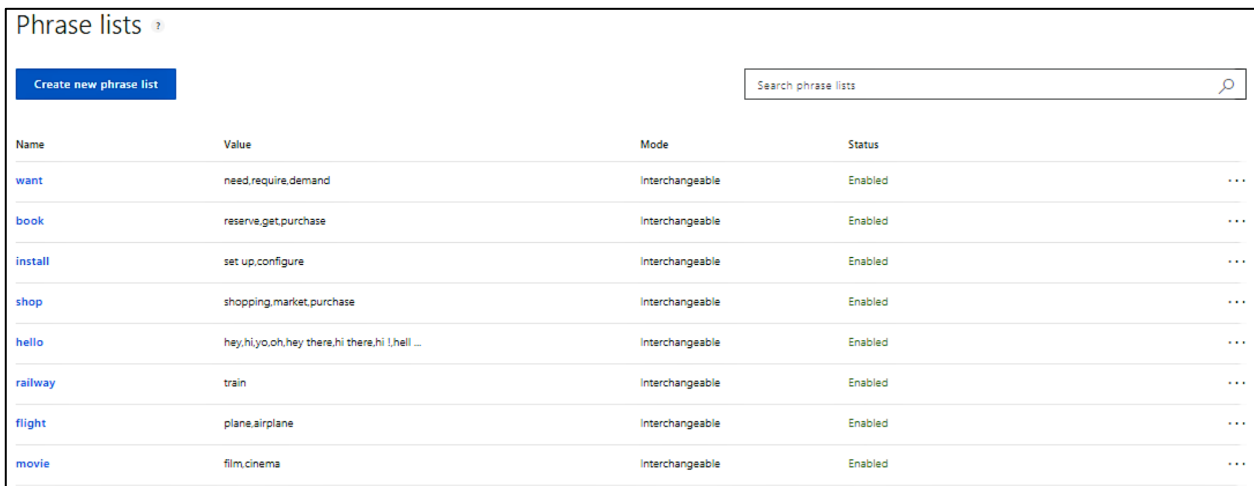


6. Adding Phrase lists

Phrase lists are added in any app to give the user permission to use different synonyms of the words used in the utterances instead of the words themselves. This improves the app performance.

- **Phrase lists**

I have created some phrase lists. They are given below:



Phrase lists

Create new phrase list

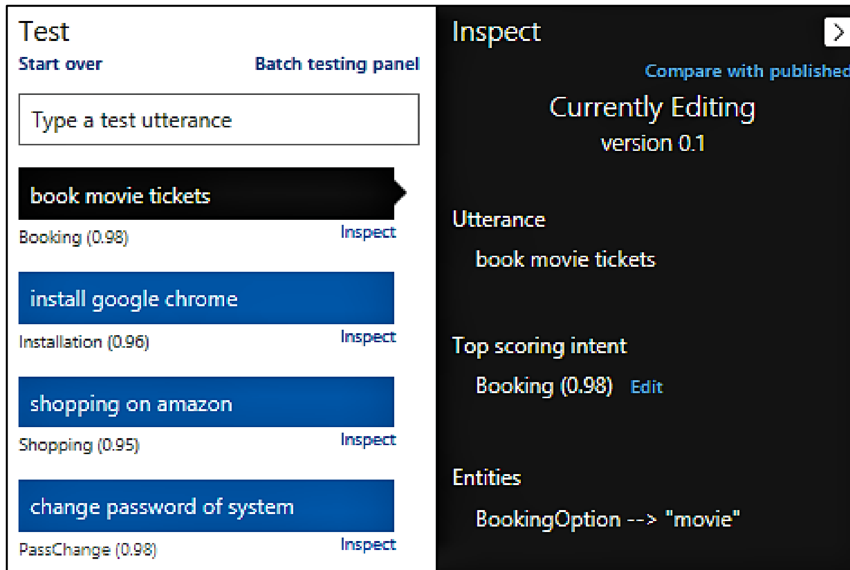
Search phrase lists

Name	Value	Mode	Status
want	need,require,demand	Interchangeable	Enabled
book	reserve,get,purchase	Interchangeable	Enabled
install	set up,configure	Interchangeable	Enabled
shop	shopping,market,purchase	Interchangeable	Enabled
hello	hey,hi,yo,oh,hey there,hi there,hi !,hell ...	Interchangeable	Enabled
railway	train	Interchangeable	Enabled
flight	plane,airplane	Interchangeable	Enabled
movie	film,cinema	Interchangeable	Enabled

These phrases are interchangeable. So, if one uses any of these words instead of the word with which the app is trained, then the same action will take place. So that’s how it makes our app more interactive and responsive.

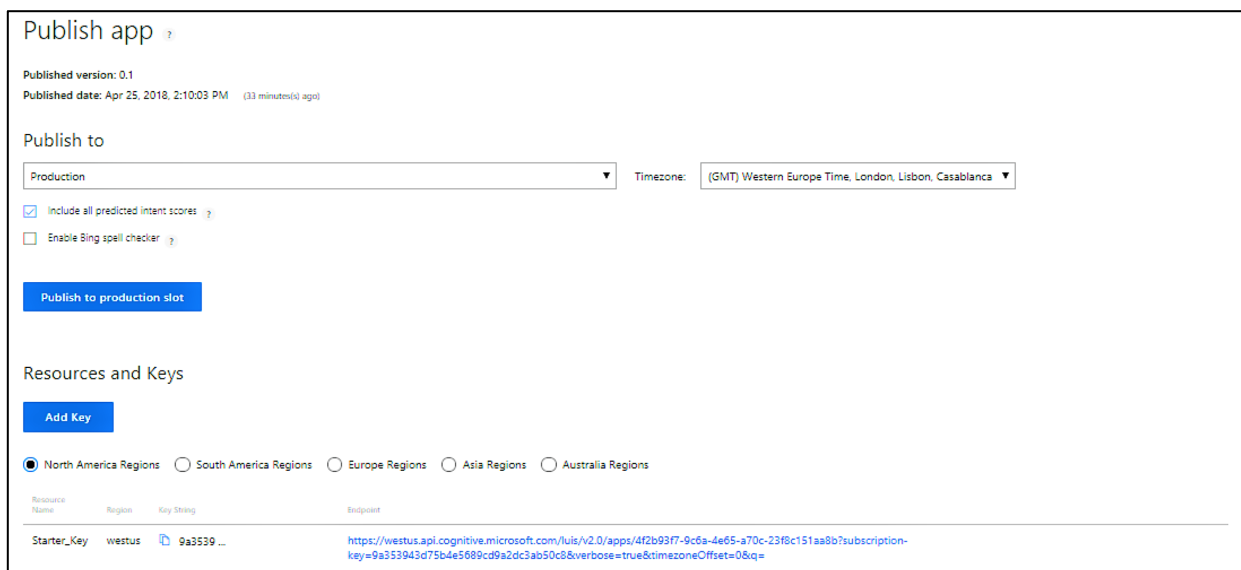
7. Testing

Now, the app can be tested. In the upper right corner, there is a **'Test'** Button to test the app. This is shown below:



8. Publishing

In order to use the app, we have to publish it to a particular region. In the lower right side, there is a URL, which contains the LUIS app ID and also the LUIS subscription key. This is shown below:



When we click on the URL, a page will be opened with a blank response page, which is the response of the app without any query. The URL and the response are shown below:

- **URL**

The URL is shown below:

https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/4f2b93f7-9c6a-4e65-a70c-23f8c151aa8b?subscription-key= 9a353943d75b4e5689cd9a2dc3ab50c8 &verbose=true&timezoneOffset=0&q=

- **Response without query**

The response without any query is shown below:


```
{"query":null,"intents":[],"entities":[]}
```


The URL consists of an element named **'q'**, this stands for the query. We can enter any query to see the generated response. This is shown below:

- **Response with query**

When any query is entered after the element 'q', then the response will change according to the query. this is shown below:


```
{
  "query": "i want to install google chrome",
  "topScoringIntent": {
    "intent": "Installation",
    "score": 0.9596489
  },
  "intents": [
    {
      "intent": "Installation",
      "score": 0.9596489
    },
    {
      "intent": "Shopping",
      "score": 0.0127906939
    },
    {
      "intent": "Booking",
      "score": 0.0101504466
    },
    {
      "intent": "PassChange",
      "score": 0.00532666361
    },
    {
      "intent": "BidBye",
      "score": 0.004148946
    },
    {
      “intent”: “None”,
      “score”: 0.00397643
    },
    {
      “intent”: “ThanksGiving”,
      “score”: 0.00257727085
    },
    {
      “intent”: “Help”,
      “score”: 0.002468674
    },
    {
      “intent”: “Greet”,
      “score”: 0.000668771158
    }
  ],
```

```
  “entities”: [
    {
      “entity”: “google chrome”,
      “type”: “Software”,
      “startIndex”: 18,
      “endIndex”: 30,
      “score”: 0.93492496
    }
  ]
}
```

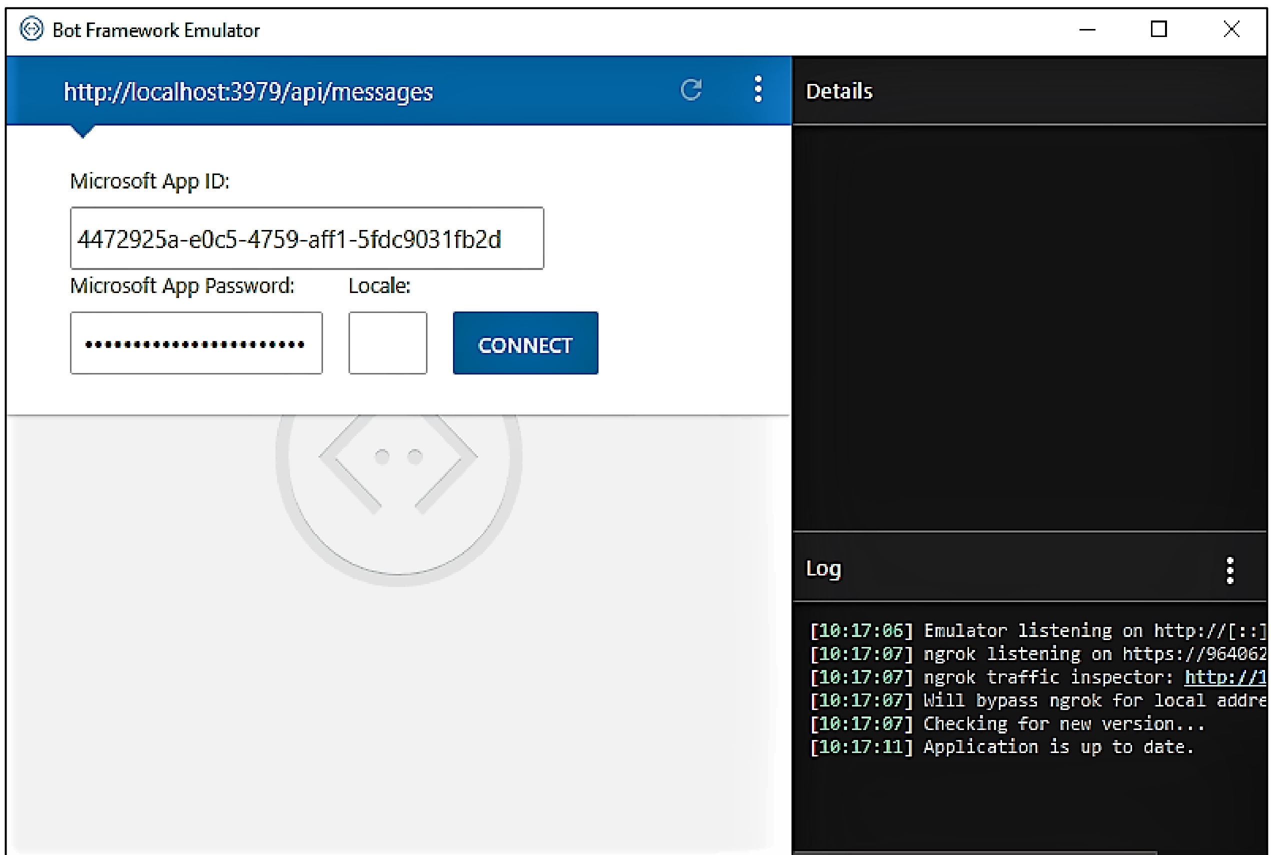
As it is seen from the above example that the top scoring intent and the top scoring entity is present in the list at the top most position, so that particular intent and entity are triggered against the entered query without any doubt.

End of chapter

CHAPTER - 6

App Responses

We will now see the bot running in action. In order to run it, we have to use the Bot Framework Emulator and give the corresponding Microsoft App Id and Microsoft App Password and click on ‘Connect’.



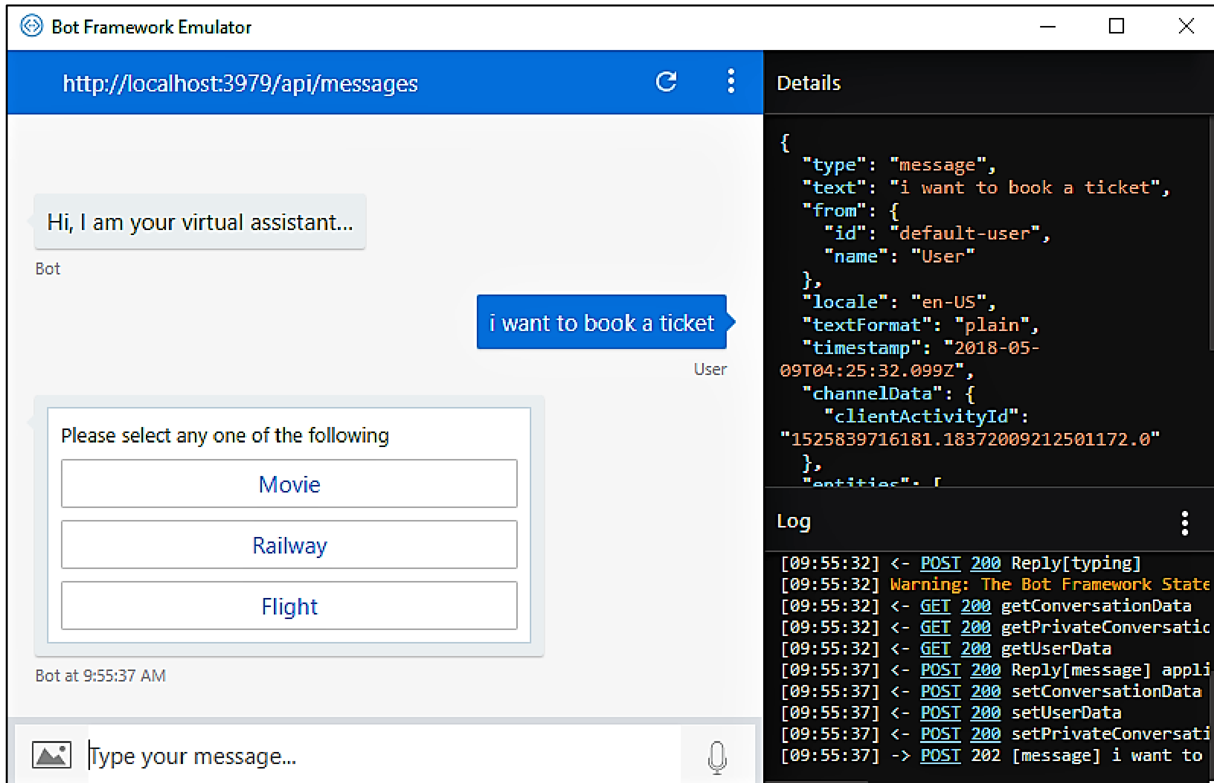
After clicking the ‘Connect’ button, we can start our chatting. If one wants to give input through speech, the he/she needs to press the speech icon located in the lower-right side of the emulator and then simply say whatever he/she wants, the speech will automatically be translated to text and will be displayed in the box as it is while giving normal text input.

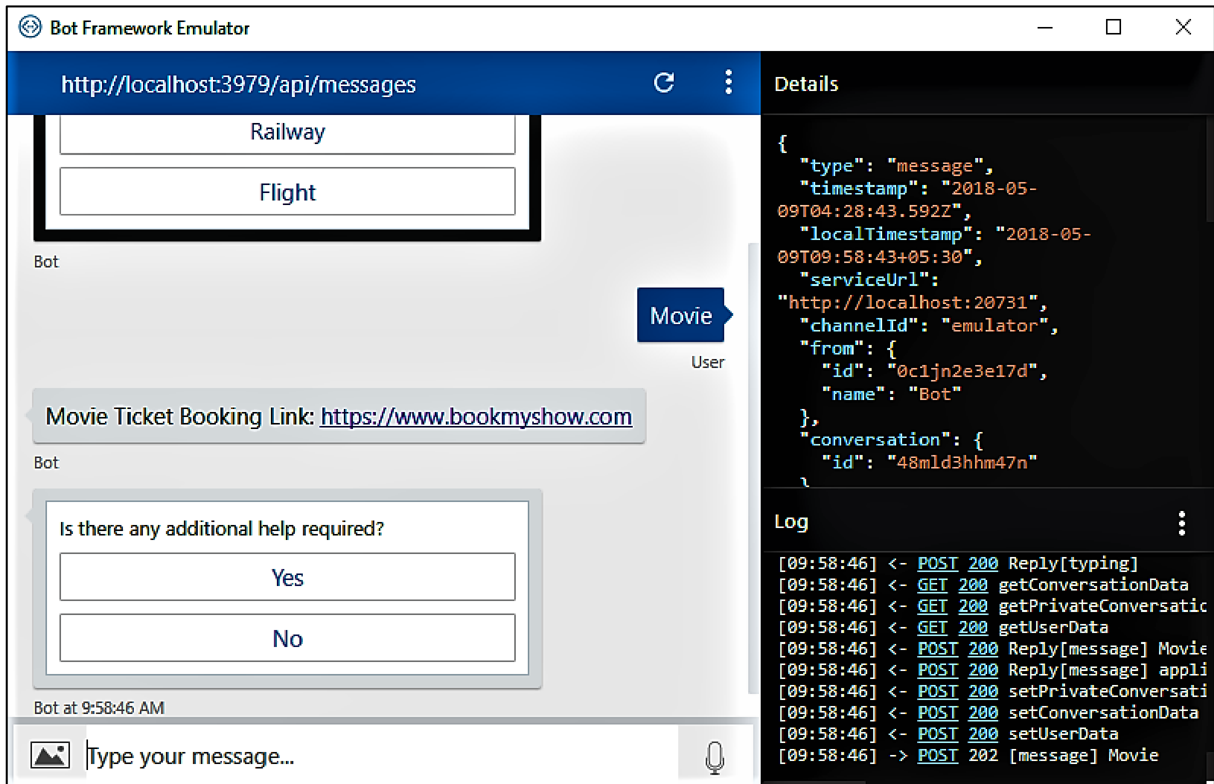
1. Intents extraction

In this case, there will be no entities in the user entered utterances.

- **Booking**

For this intent, the bot will ask the user what ticket he/she wants.

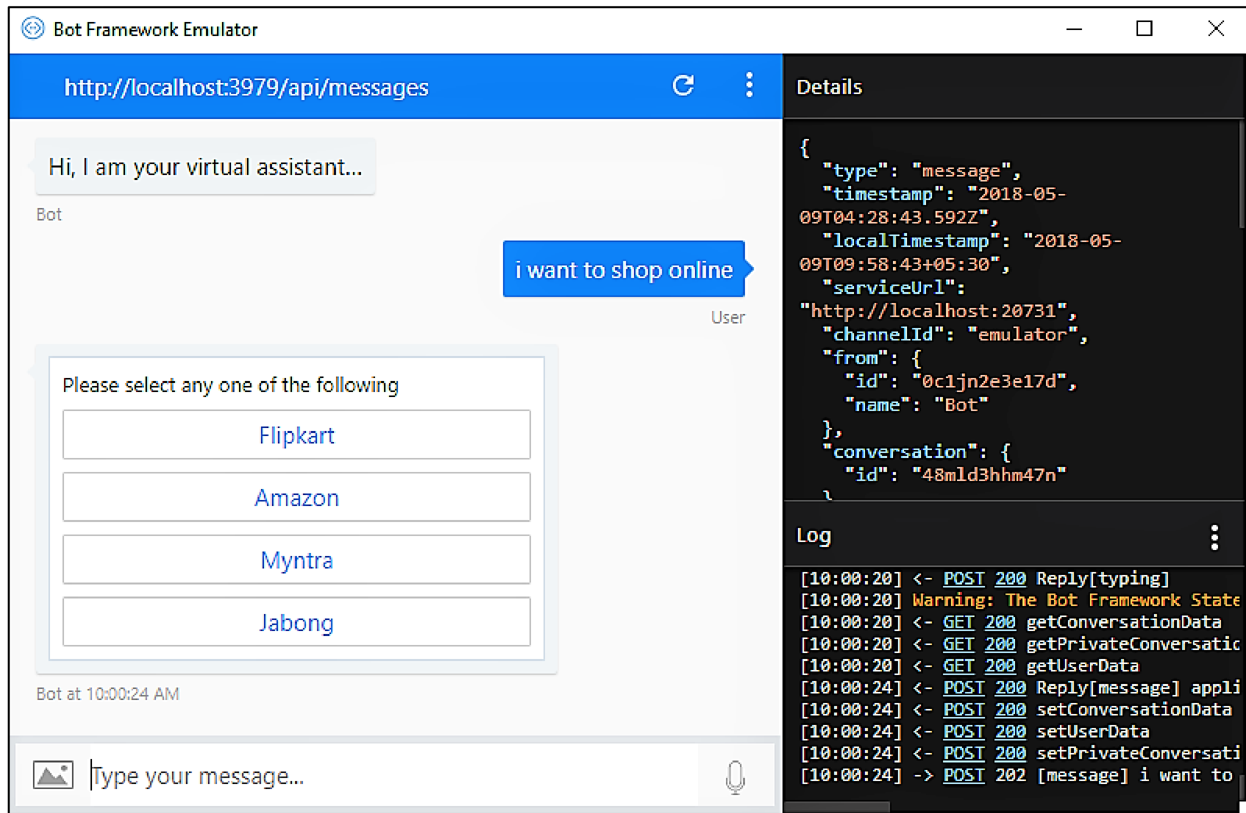


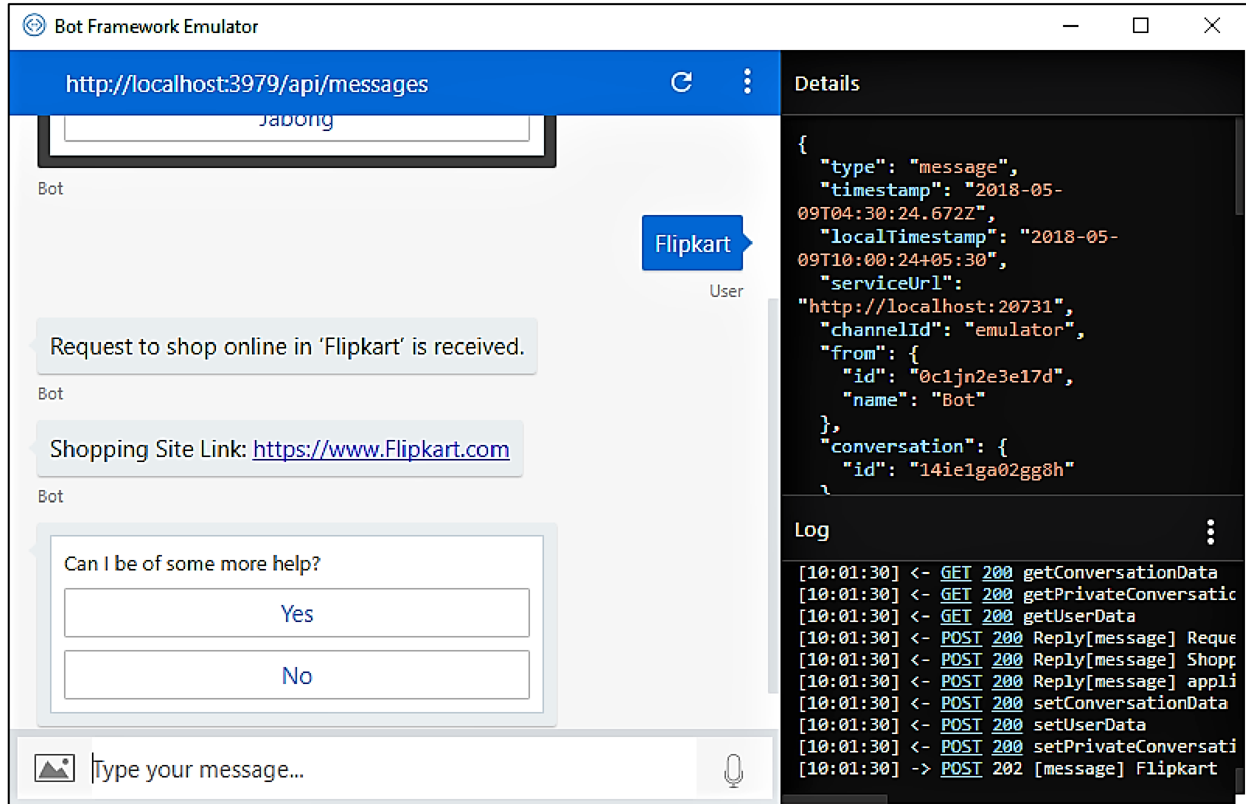


As it is seen, the bot asks the user to select one option, and when the user selects one option, the corresponding link is displayed and after clicking on the link, we can go and do our job.

- <u>**Shopping**</u>

For this intent, the bot will ask the user which portal he/she is looking for.

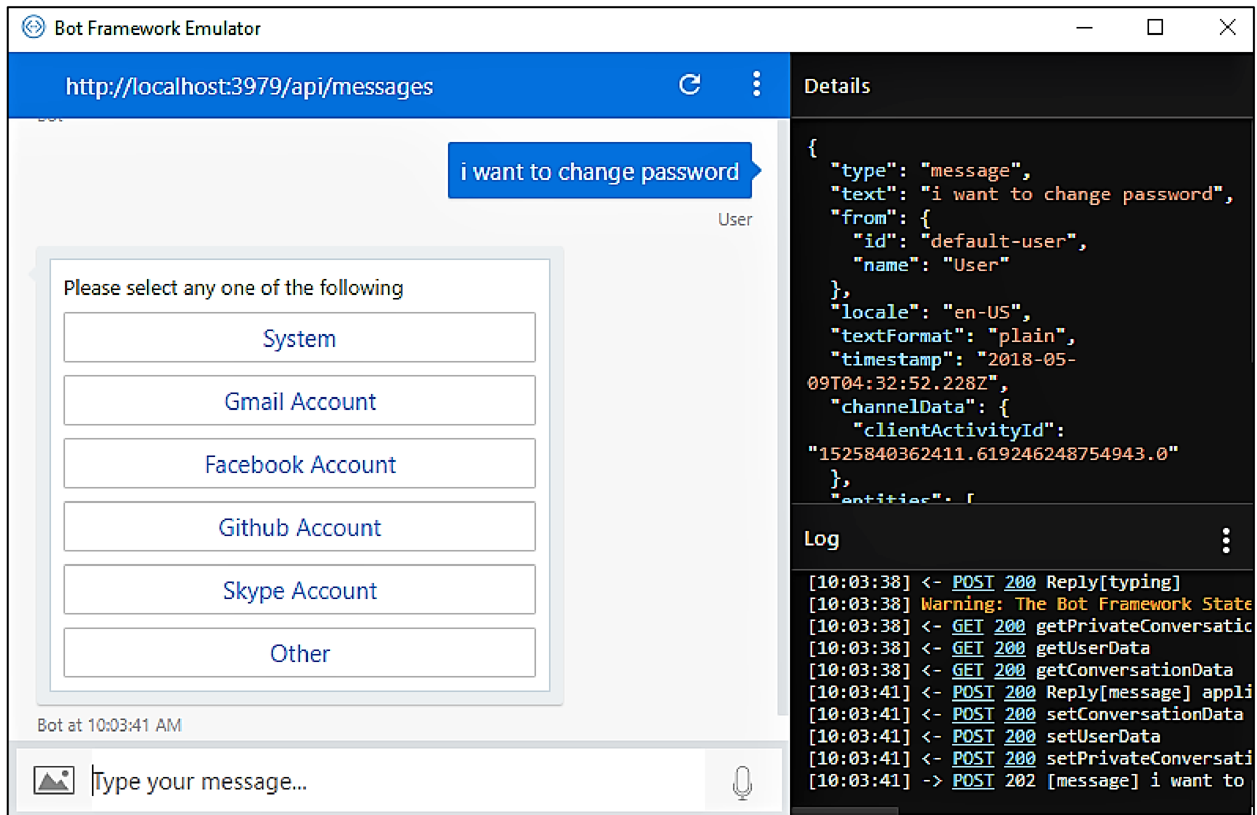


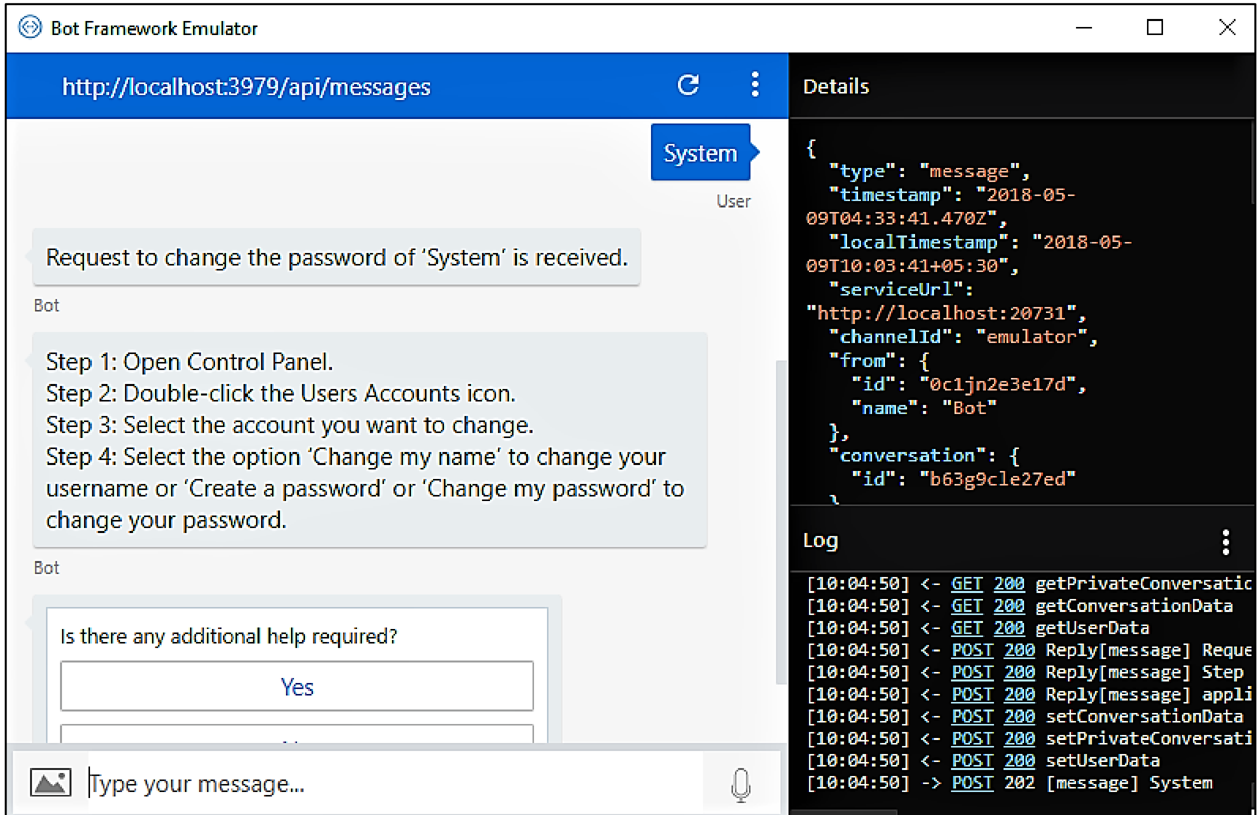


As it is seen, the bot asks to choose an option and when the user chooses one option, the corresponding link is opened and after clicking on the link, the link is opened.

- **PassChange**

For this intent, the bot will ask the user which platform he/she wants to change the password of.

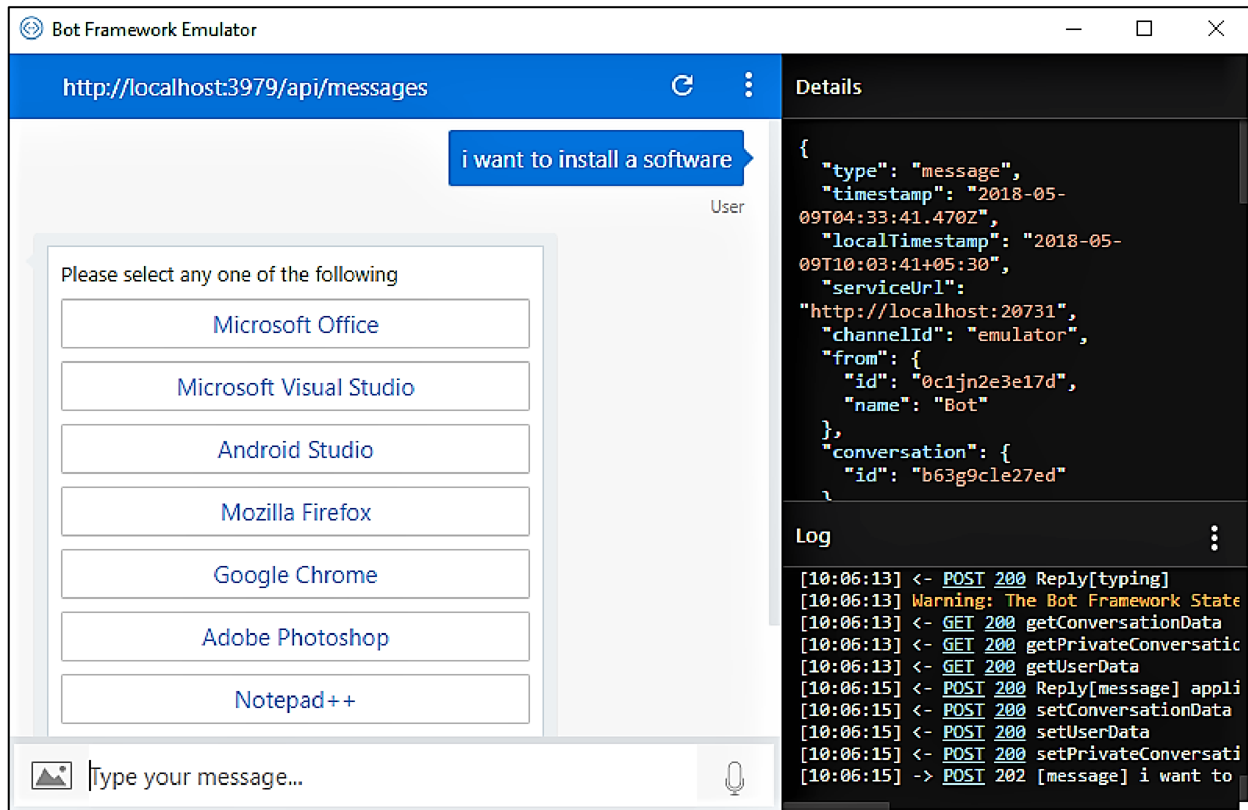


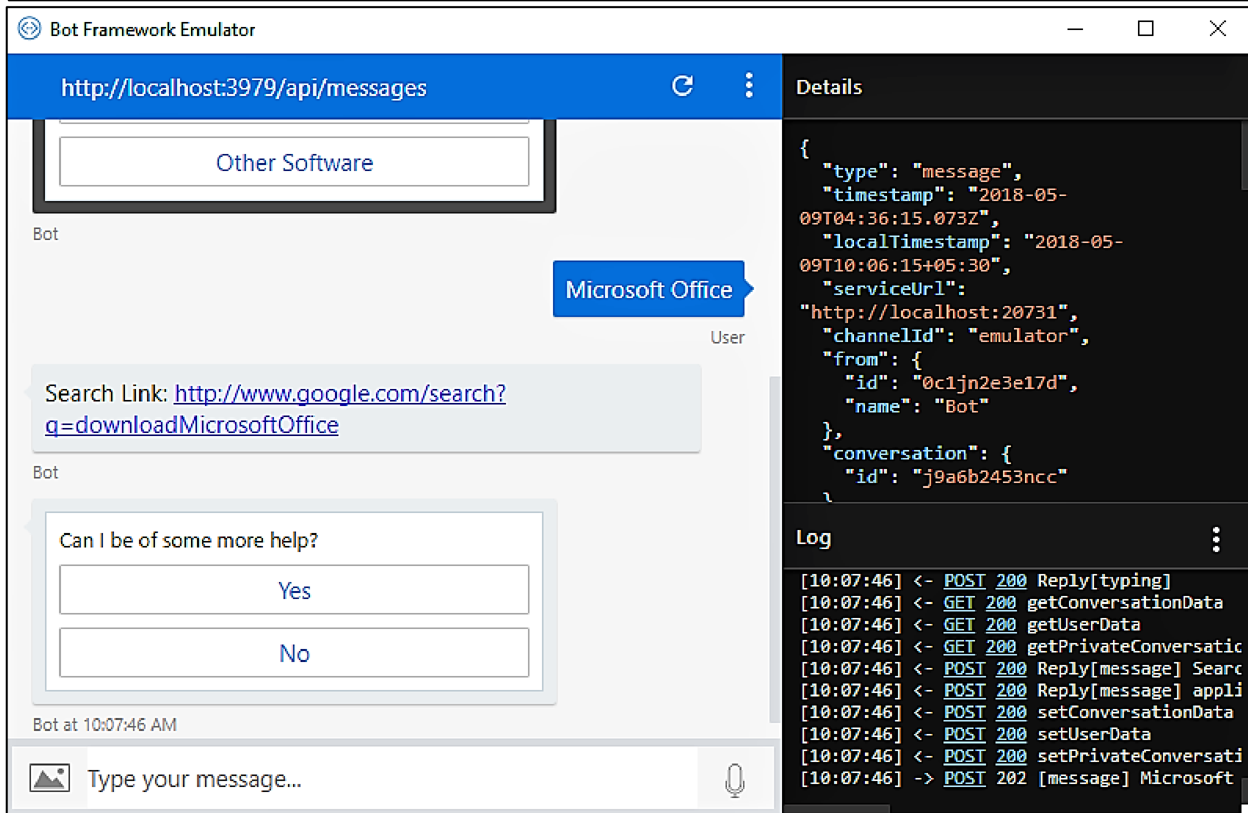


As it is seen the user asks the user to choose one option, and when the user chooses one option, the corresponding information is shown to the user.

- **Installation**

For this intent, the bot will ask the user which software he/she wants to install.



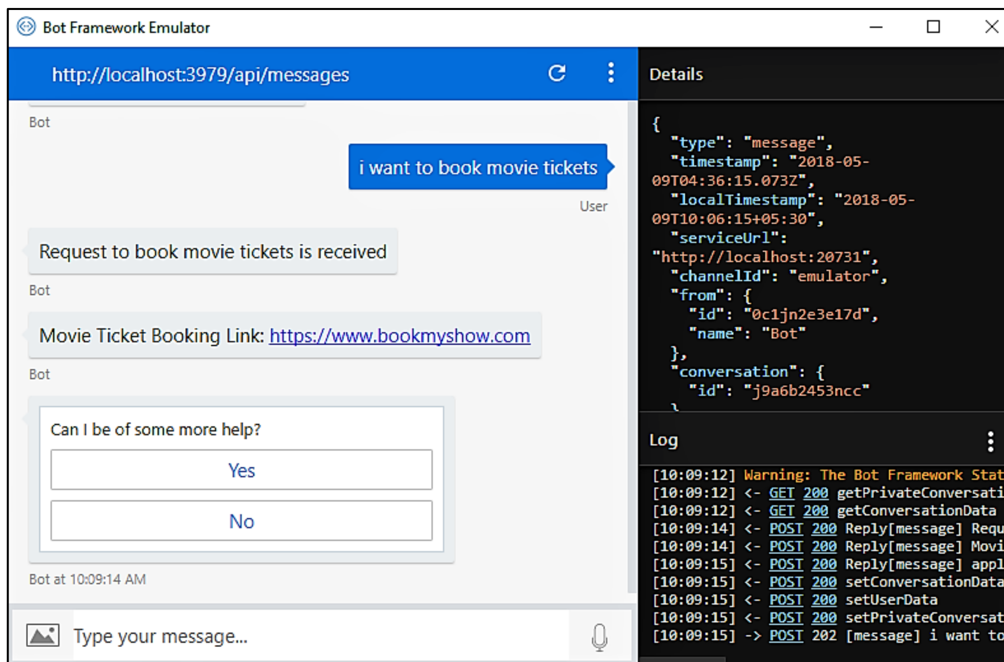


As it is seen the bot asks the user to select one option, and when the user selects one option, the corresponding search link is given, after clicking on this link, the user can download the required software.

2. Intents and Entities extraction

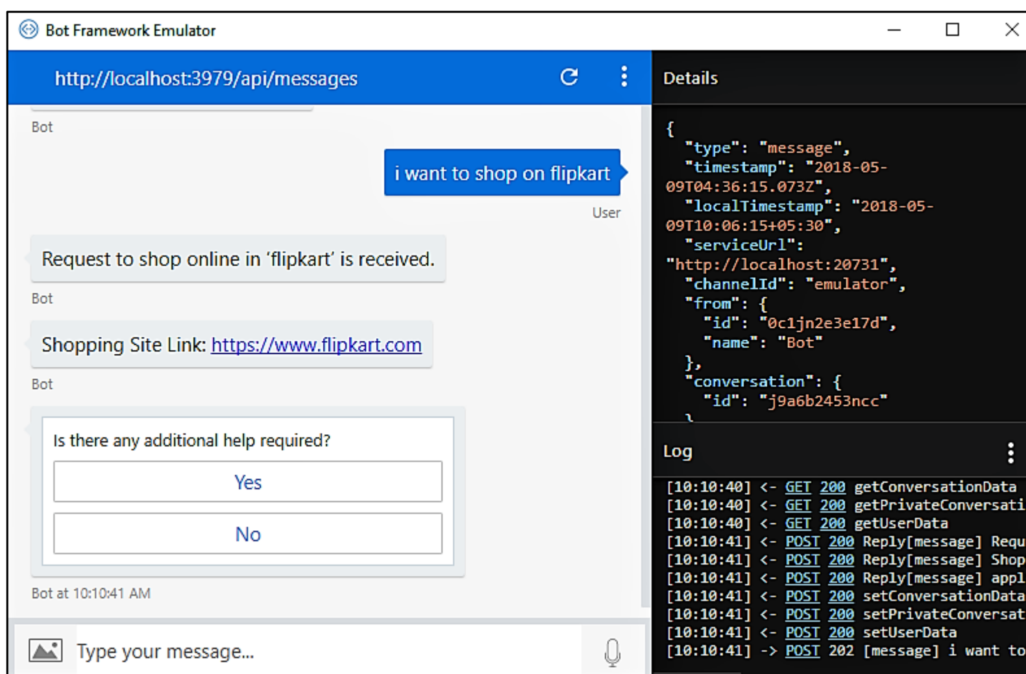
In this case, there will be entities in the user entered utterances.

- **Booking**



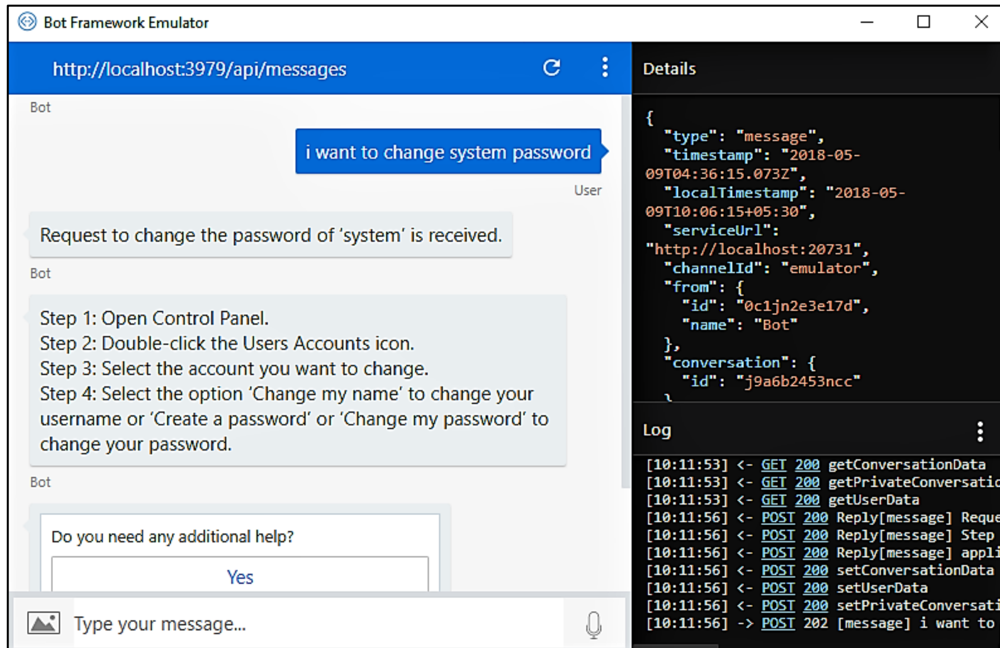
As it is seen that the word 'movie' is recognized as an entity. So the bot extracts that intent and directly shows the link without asking the options.

- **Shopping**



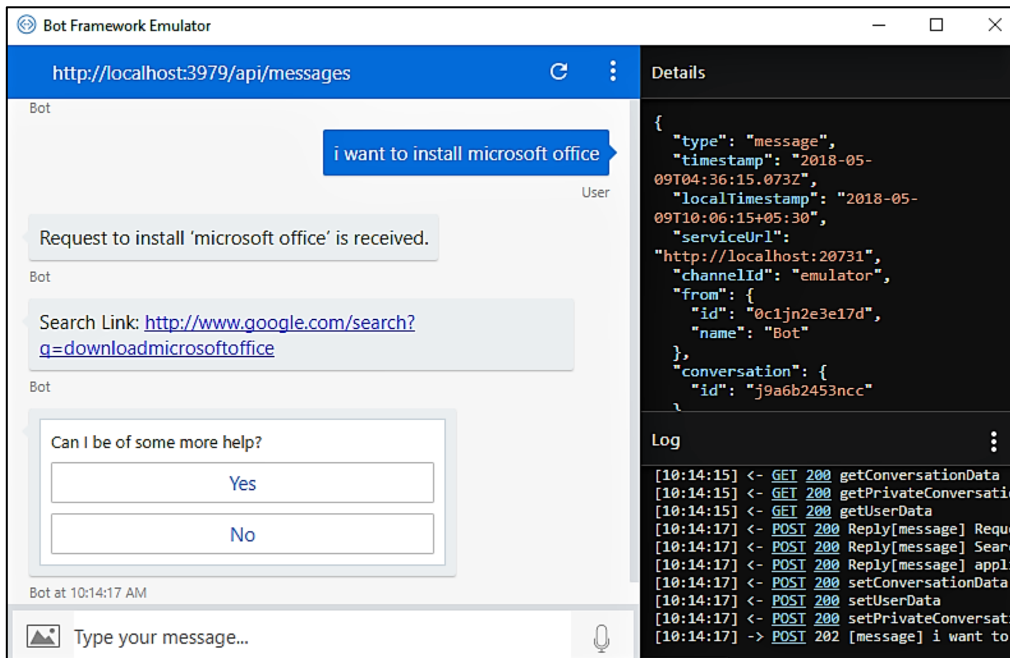
As it is seen the word 'flipkart' is recognized as an entity. So the bot extracts it and directly shows the link.

- **PassChange**



As it is seen that the word 'system' is recognized as an entity. So the bot extracts the entity and directly show the information corresponding to it.

- **Installation**



As it is seen that the word 'microsoft office' is recognized as an entity. So the bot extracts it and shows the link.

End of chapter

CONCLUSIONS

• Why is now the time of the chatbots?

In early 2015, people started using messaging applications more than they use social networks.
This is a significant shift and a huge turning point in how consumers consume information. Up until 2015, to market a business online, one would use social networks – as this is where his/her consumers were. Now, there is a better place to concentrate resources (Tweet this).

Businesses that seize opportunity are the ones that follow consumers the fastest.
Think back to 5 or so year ago.
“There’s an app for that” – said everyone.

Now it is probably too late for a business to create an app, similar functionality can probably be better delivered elsewhere. I certainly do not think any sane person would form an app-building start-up.
It is not just consumer trends.

Another contributing factor is the commercial opportunity, and therefore, interest from large (wealthy) companies. The platforms that enable the delivery of chatbot experiences are opening up to larger audiences and more innovative ways of creating an ROI and user interaction are being rapidly developed.

It is the culmination of the consumer behaviour (moving to messaging apps) and the technology being ready, along with a greater cultural shift in consumer behaviour.
People have been using messaging apps (and SMS) to talk with friends and family for long enough to feel confident in using the same practices to communicate with a business. This coincides with businesses now having the tools and technology to effectively communicate through the apps in a way consumers require.

• Are chatbots the future?

A question definitely comes in this context, that is “Are chatbots the future?”.
Well the answer depends on the Organization requirements. Many organizations use bots instead of humans as they are faster and more responsive than humans, but to be honest a bot can never replace humans in all aspects because they run through programs, and no matter how good the code is, there will always be some flaw in the code as we can never achieve perfect machine learning.

Chatbot technology will adapt to us and creating personal chatbots will be as easy as changing the settings on one’s Facebook account, or adding an inbox filter to one’s email. It will know one’s surroundings, his/her personal history, his/her culture and language. It will become useful in ways we cannot yet comprehend or imagine.

REFERENCES

[1] ABSTRACT:

https://chatbotslife.com/a-chatbot-abstract-1cd002e7a480

[2] INTRODUCTION:

https://blog.ubisend.com/discover-chatbots/what-is-a-chatbot-introduction

[3] Chapter - 1: Creating and testing Bots (Azure Bot Service Documentation):

https://docs.microsoft.com/en-us/azure/bot service/?view=azure-bot-service-3.0

[4] Chapter - 2: Language Understanding (LUIS) (LUIS Service Documentation):

https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/Home

[5] Chapter - 3: Key concepts in Bot Builder SDK:

https://docs.microsoft.com/en-us/azure/bot-service/dotnet/bot-builder-dotnet-concepts?view=azure-bot-service-3.0

[6] CONCLUSIONS:

https://blog.ubisend.com/discover-chatbots/what-is-a-chatbot-introduction

[7] C# tutorial:

https://www.tutorialspoint.com/csharp/index.htm

[8] C# fundamentals:

- Svetlin Nakov, Veselin Kolev & Co. FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#
- Andrew Stellman, Head First C#

[9] Bot repositories:

- https://github.com
- https://bitbucket.org