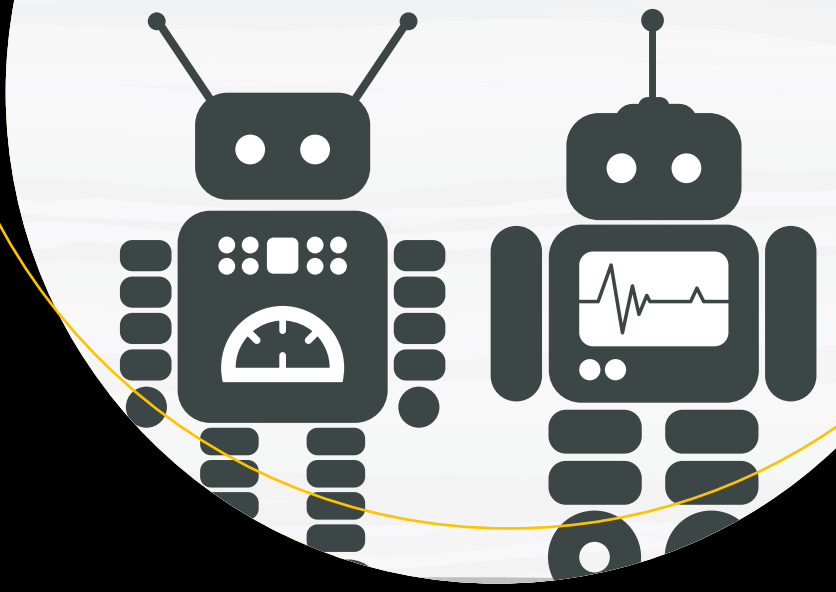

Developing Bots with Microsoft Bots Framework

Create Intelligent Bots using MS Bot Framework and Azure Cognitive Services

Srikanth Machiraju
Ritesh Modi

Apress®

Developing Bots with Microsoft Bots Framework

Create Intelligent Bots using MS Bot Framework and Azure Cognitive Services

Srikanth Machiraju
Ritesh Modi

Apress®

Developing Bots with Microsoft Bots Framework

Srikanth Machiraju
Hyderabad, Andhra Pradesh, India

Ritesh Modi
Hyderabad, Andhra Pradesh, India

ISBN-13 (pbk): 978-1-4842-3311-5
ISBN-13 (electronic): 978-1-4842-3312-2
https://doi.org/10.1007/978-1-4842-3312-2

Library of Congress Control Number: 2017962439

Copyright © 2018 by Srikanth Machiraju and Ritesh Modi

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Nikhil Karkal
Development Editor: James Markham
Technical Reviewer: Puneet Jindal
Coordinating Editor: Sanchita Mandal
Copy Editor: April Rondeau
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please email rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3311-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Contents

About the Authors ix

About the Technical Reviewer xi

Introduction xiii

Target Audience xv

Chapter 1: Conversations as Platforms 1

Types of User Interfaces 2

Drawbacks of Conventional UI 3

Conversations as Platform 5

Introduction to Microsoft Bot Framework 7

Meet a Few Bots 9

Summarize 9

Your Face 10

Azure Bot Service 11

LUIS Bot 13

QnA Bot 13

Proactive Bot 14

Direct Line 15

IOT and Bots 16

Other Bot Frameworks 16

Bot Abuse 17

Summary 17

Chapter 2: Develop Bots Using .NET Core 19

Designing Bot Applications 20

Setting Up the Development Environment 23

Testing the Bot 26

Debugging the Bot Application 27

Bot Application Life Cycle 28

Bot Architecture 31

Bot Authentication 32

Building a Bot 33

Deploy Bot to Azure 38

Register the Bot 42

Configure Channels 45

Configuring Skype Bot 47

Configuring Web Chat 50

Summary 52

Chapter 3: Develop Bots Using Node.js 53

Setting Up a Development Environment 54

Build Hello World Bot Using VS Code 54

Debugging Using VS Code 60

Building Bots with Conversations 61

Dialogs 61

Prompts 62

Messages 67

State 68

Deploying to Azure 69

Summary 73

Chapter 4: Channels 75

Channels and Channel Data 75

Channel Data 78

Build a Chat Bot Using an Email Client 80

Build a Chat Bot Using Slack Channel and API 87

Multi-dialog Bot Using Slack and Slack Channel Data 89

Onboarding a Slack Bot 95

Remote Debugging Slack Bot on Development Machine 96

Summary 97

Chapter 5: Bot Conversations 99

Understanding Conversations 100

Messages 100

Activity 101

Relationship Between Channels, Conversation, User, and Bot 102

Message Under the Hood 103

Conversation Under the Hood 104

Building Bots with Conversations 105

Attachments 105

Hero Card 111

Thumbnail Card 113

Carousal 114

Buttons 116

Prompts 116

Summary 121

Chapter 6: Skype Calling Bot 123
Introducing Skype Calling Bots 124
Use Cases for Skype Calling Bots 124
Enabling Calling for Your Bot 125
Building a Skype Calling Bot 126
Sequence of Events 130
Debugging Skype Calling Locally Using Ngrok 139
Speech-to-Text Using Bing Speech API 141
Summary 149
Chapter 7: Storing State 151
Stores for Bot State 152
State Service 153
Storing and Retrieving State Using StateClient 155
Storing and Retrieving State with Dialogs 158
More Control over State with Dialogs 162
Custom State Data Store 165
Overview of Cosmos DB 166
Cosmos DB as Custom State Data Store 166
Table Storage as Custom State Data Store 173
Summary 180
Chapter 8: Dialogs 181
The Dialog Model 181
IBotData 182
IBotTouser 182
IDialogStack 182
IBotContext 183

Dialog Stack 183
Dialog Context 183
Root Dialog 183
Building a Simple Dialog Bot 184
SimpleDialog.cs 184
MessagesController.cs 188
Creating Multi-Dialog Bots 189
Scenario 190
Solution 191
RootDialog.cs 192
Synonym.cs 194
Antonym.cs 194
Support.cs 194
MessagesController.cs 195
FormFlow 195
Building a Simple FormFlow Bot 196
FormBuilder 199
Customizing the Prompts 199
Customizing the Order of Prompts 200
Conditional Fields 200
Summary 202
Chapter 9: Natural Language Processing 203
Cognitive Services 204
LUIS 204
Intents 204
Entities 205
Utterances 205
Features 206
LUIS Development Lifecycle 206

Sample Application 211
Creating Intelligent Bots 215
Creating Intelligent Bots Without Dialogs 215
Summary 232
Chapter 10: Azure Cognitive Services 233
Introduction to Microsoft Cognitive Services 234
Getting Started 238
Building Smart Bots with Bing Web Search 244
Query Parameters 247
Bing Search Request 249
Handling Errors 253
Optical Character Recognition with Computer Vision API 256
Summary 260
Chapter 11: Bot Operations 261
Application Insights 261
Getting Started 262
Enable Bot Analytics 271
Advanced Analytics 277
Summary 278

Index 279

About the Authors

Srikanth Machiraju has over nine years of experience as a developer, architect, technical trainer, and community speaker. He is currently working with Microsoft Hyderabad designing and preaching modern application development using cutting-edge platforms and technologies. Prior to Microsoft, he worked with BrainScale as a corporate trainer and senior technical analyst on application design, development, and migration to cloud. He is a tech-savvy developer who is passionate about embracing new technologies and sharing his learning via blogs or community engagements. He has also authored *Learned Windows Server Containers*, blogs at https://vishwanathsrikanth.wordpress.com, and is active on LinkedIn at https://www.linkedin.com/in/vishsrik/.

Ritesh Modi is an ex-Microsoft Commercial software engineering senior technology evangelist. He is an architect, senior evangelist, cloud architect, published author, speaker, and a leader known for his contributions to Datacenter, Azure, Bots, Blockchain, cognitive services, DevOps, artificial intelligence, and automation. Currently, he is working as principle consultant with Infront Consulting. He is the author of multiple books, including `Azure for Architects` and `DevOps with Windows Server 2016`—both available on Amazon and B&N. He co-authored another book titled `Introducing Windows Server 2016 Technical Preview` with the Windows Server team. He has spoken at more than 15 conferences, including TechEd and PowerShell Asia conference, and is a published author for *MSDN* magazine.

He has more than a decade of experience in building and deploying enterprise solutions for customers. He has more than 25 technical certifications. His interests and hobbies include writing books, playing with his daughter, watching movies, and learning new technologies. His blog is available at https://automationnext.wordpress.com and his Twitter handle is `@automationnext`. He is based out of Hyderabad, India.

About the Technical Reviewer



Yogesh Sharma is an IT consultant based in Pune, India. He specializes in helping organizations with quality and cost improvement via migration and automation to various cloud offerings. His recent interest lies in collaborative learning. He would like to acknowledge Prachi, Celestin, and the entire editorial team for the opportunity and support.

Introduction

Bots are the new face of the user experience. Conversational user interfaces (CUI) provide a plethora of options to make the user experience richer, innovative, and appealing with Email, SMS, Image, Voice, or Video to communicate with the application. Modern web or desktop applications will soon be replace or augmented with intelligent bots that can be connected from anywhere using any device. Bots can use artificial intelligence and user data to provide richer insights and a personalized experience.

The Microsoft Bot framework has made the bot-building experience easy, and by using this framework we can build rich, scalable, and intelligent bots that can be connected from anywhere using an impressively vast list of platforms like Email, Skype, SMS, Facebook Chat and so on. This book explains how to develop intelligent bots using the Microsoft BOT framework, Visual Studio, Microsoft Azure, and Microsoft Cognitive Services. The preliminary chapters of the book deal with helping developers learn the basics of bot development using Visual Studio, .Net, C#, and Node.js. You will learn basic development and debugging skills, publishing to Azure, and configuring bots using the bot developer portal. The advanced section of the book deals with building intelligent bots using scalable storage, conversation flows, Microsoft Cognitive Services like LUIS, Bing Search, Vision and Voice API. This section also explains configuring analytics and other common Bot Operations.

The book is divided into the following sections:

Part 1 focuses on the need for a new communication platform and how conversation user interfaces (CUIs) break the barriers of building user interfaces; it also describes the current trends and future focus of this upcoming CUI and bot platform. This part also focuses on salient features of the MS Bot framework, available versions and features, and how the industry is embracing the change, and compares it with other competitive technologies and roadmap.

Part 2 teaches how to design and develop simple Skype bots on the Windows platform using the MS Bot framework, Skype, Visual Studio, .NET, and Azure.

In this part, the readers will also learn how to build bots using open source platforms like Node JS and VS Code. Readers will be shown how to manage the complete lifecycle of a Skype bot, like design, development, testing, and pushing to production.

Part 3 goes into prospective features of the Microsoft Bot framework, like channels and channel data and using rich text, buttons, media, and actions in chat messages. It also explains building a bot using the Skype Calling API and speech-to-text conversion, managing user data using Bot State Service, and using different conversation flows, like dialog model and form-flow model.

Part 4 delves into the details of building bots by integrating with Azure Cognitive Services, like Bing Search, OCR, and LUIS. We also focus on how to perform common operations, analytics, and diagnostics on bots in production environments.

Target Audience

The target audience for this book is C#/Node.js developers and architects who design and build modern applications using a Microsoft stack like Azure Cloud, Visual Studio, and code. Developers who want to get up to speed by learning the cutting-edge technologies that enrich the user experience and cater to multiple form factors. Architects/developers who wish to learn to build and design scalable and reliable messaging platforms that offer rich conversation experiences with the use of attachments, rich text, and voice for communication with enterprise applications. Developers can learn to integrate bots with machine learning and Cognitive Services offered by Azure. Business analysts and UX specialists can also learn to design trendy user interfaces by using bots and Azure ML that can be connected using any device and provide an enriched user experience to end customers. The target audience of this book do not need any prior bot-building experience.

CHAPTER 1

Conversations as Platforms

Have you ever had the experience of ordering pizza using an application that remembers your favorite pizza and orders to your current location automatically? Or have you ever booked a cab just by typing in a chat window or by using voice inputs and had a cab show up at your door step? If you have seen either of those, what you have experienced is the new generation of smart applications called bots (a short form of robots). Bots provide richer and more personalized experiences in our day-to-day activities, thereby making our lives much better. If you have not experienced this firsthand, you have yet to witness the next revolution in IT after the worldwide web, mobile, and data. Bots are much smarter than mobile applications; in some cases they can be smarter than you. Bots are designed to perform human-like interactions and exhibit human-like intelligence. Chat bots are not new; we have had platforms that help build chat-based applications for quite a few years (like Skype SDK), but what makes the new generation of bots special is their integration with artificial intelligence.

Over the years, smart devices and smart phones have become such an integral part of our life that they now hold lots of useful personalized information, like your favorite color, calendar, contacts, favorite restaurants, and so on. New-generation bots are designed to use the data and context surrounding the data with machine-learning (ML) and deep-learning technologies to give you a richer experience. A few decades ago, using ML or deep-learning technologies in a commercial application was highly complicated because they involve lots of new learnings and come with heavy computing and memory requirements. With the advent of cloud and serverless computing, the use of machine learning, data analytics, and advanced algorithms like facial recognition, voice recognition, and search is just a click away. The focus of this chapter will be on introducing the benefits of building conversations as a platform for all kinds of business needs; the Microsoft Bot framework, one of the top-class, end-to-end suites for building smarter, richer bots; and the various bot intelligence services and platforms available.

The following topics will be discussed in this chapter:

- Types of user interfaces
- Drawbacks of conventional user interface
- Conversations as platform
- Introduction to Microsoft Bot framework
- Meet a few bots

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_1

- Azure Bot Service
- Direct Line
- IOT and bot scenarios
- Other bot frameworks
- Bot abuse

Types of User Interfaces

User interfaces represent the face of any application. They should describe what the application can do and should be easy to use. It can be as simple as a command line where the customer interacts via commands or much more sophisticated, like a mobile application that can accept voice inputs from the user. Historically, user interfaces were more command-line or custom-input devices with a pre-defined set of commands printed on them; they could only perform a limited set of operations, like the interface shown in Figure 1-1. These types of interfaces had a limited set of responses and were not designed with the intelligence to respond to an unexpected random request. Some of them do not even retain any context of the user's previous conversations, which could be used to improve conversations with the returning user.

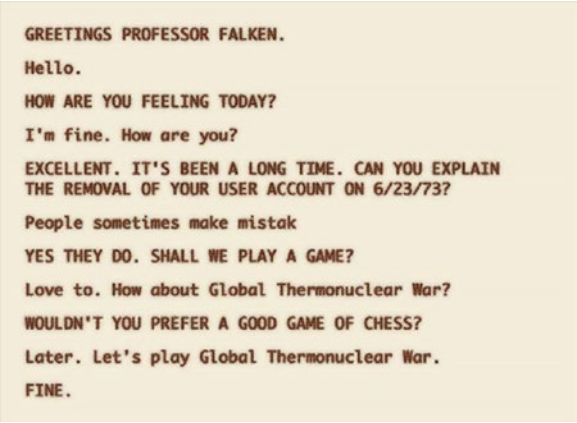

Figure 1-1. *WarGames: David Lightman talking with Joshua*

The most common user interface that we see today is called a graphical user interface (GUI), which was popularized by Xerox, Apple, and Microsoft during the 1980s.

Figure 1-2 shows the first GUI with a bitmapped screen. It was developed by Xerox in the year 1973 and was called Xerox PARC. In 1981, Xerox introduced Star and Workstation, which adapted Xerox PARC and influenced future innovations.

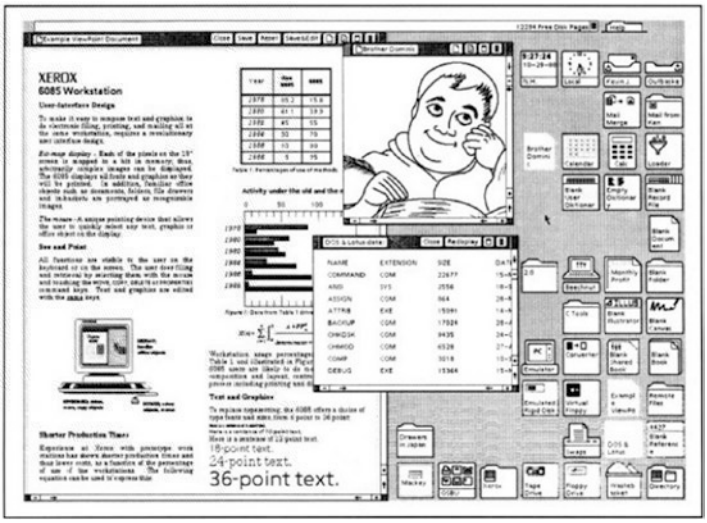

Figure 1-2. *Xerox Star user interface*

GUIs have evolved over the past few decades and are more sophisticated and easy to develop today. Modern applications are based on GUI, which are designed for desktop operating systems, web browsers, mobile, or kiosks. This, so far, is the easiest and most user-friendly form of interface, where the user interacts via button clicks (or touch) on a clickable element and uses an alpha-numeric keyboard, which can be used to input text, numbers, or symbols. Users are accustomed to these types of user interfaces, but haven't they become monotonous? Why would you want to ask the user to enter personal information like address, phone number, or favorite color every time; why would you need a call-center operator to answer the same questions from different customers? Why would you want to ask the user to install a mobile application or log in to your web application to perform a daily task like booking a cab, ordering your favorite food etc? Bots help you bring in a customized experience with natural language recognition and artificial intelligence—like voice-, image-, or video-based communication—to an application you are already using.

Drawbacks of Conventional UI

There was a reason we moved away from command-based interfaces to GUI, but what made us shift back to the old way? GUIs have their own shortcomings, and it is a challenge to build an effective user interface, be it for mobile or desktop. Not every GUI-based application makes optimum use of the screen space. For example, the GUIs in Figure 1-3 try to show all the available features or options on one screen, and for a first time user this can be overwhelming. It takes a while to figure out where to click and how to get the work done. These types of user interfaces force the business to include a readme document or 24/7 customer service, which can help users with queries about the portal's usage.

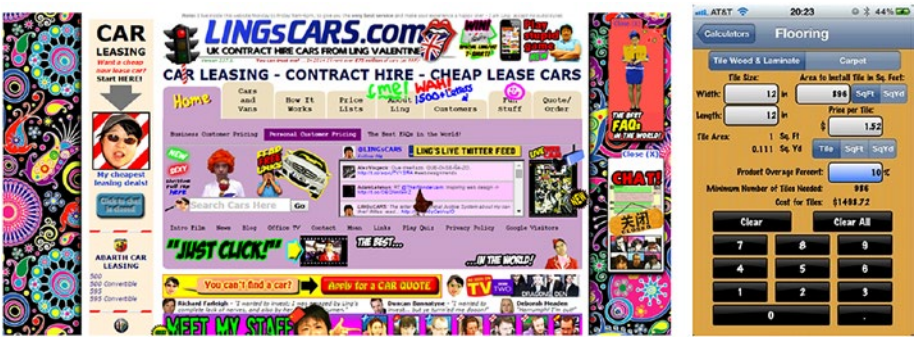

Figure 1-3. *Example of suboptimal screen design in a web and mobile application*

Mobile applications are constrained by screen space. The small screen size poses a challenge to building an effective application that is self-explanatory and covers all the operational aspects of the business. This restriction forces the stakeholders to design the application with only a few features on mobile and in parallel run a web version with a full feature set. It is also difficult to go through 20 clicks and a form-filling process on mobile for a few kinds of businesses. Web-based GUIs also have their limitations. Normally, the team involved in building these applications is huge and involves a UX designer, who works on the HTML + CSS, a development team, which builds the application logic, and database teams, which design the database schema. Bots, on the other hand, gel into existing chat interfaces like Skype, Slack, or Facebook chat, thus reducing the effort put into the application UI design so that the team can focus on building the application logic. For desktop applications, there is the additional challenge of the underlying software; this could be Java runtime, .NET, or node.js, which needs to be installed on the user's machine for the application to run. Desktop applications always target a specific operating system and version of runtime. It is highly difficult to build an application that works across a multitude of operating systems. Also, in a typical GUI application, the user moves from one screen to another; sometimes this makes it difficult for the user to understand and accept the flow and structure of the application (Figure 1-4). The infrastructure costs that are incurred when setting up the machines that run these applications or backend support systems is also huge. Chat bots do not need an enormous machine that hosts the interface; they can be designed to use existing applications to interact with business, reducing the application's running costs.

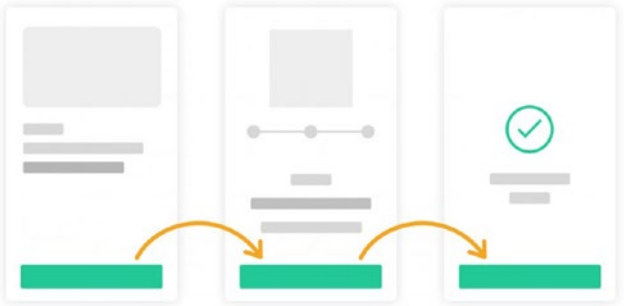

Figure 1-4. *Example of user experience in a typical GUI-based application*

Source: https://yourstory.com/2016/12/2017-year-conversational-user-interface/

The processing capabilities of today's computers have increased, most of the computations can be done at a large scale in real-time, and almost every task is achievable just by using a mobile phone or device. Why not change the way we interact with applications? Can we interact with these applications in a completely different way just by providing voice or text as inputs in a chat window, like you are talking to a friend? Can we build intelligent application interfaces that can use the context of my location, history, and personal preferences to finish tasks on my behalf; for example, a smart application that can track the upcoming football games of my favorite team and reserve a seat for me based on how my calendar is scheduled. Modern chat-based applications called bots are the key to the preceding questions. Bots are intelligent, context aware, and can hold more human-like conversations with the user.

Conversations as Platform

A conversational user interface (CUI) is any user interface that allows you to perform human-like interactions (Figure 1-5). What exactly is a human-like interaction? Human-like interactions use more natural language, which can be text, image, voice, or a combination of those. Human-like interactions are a conversational approach in which the application can understand and respond to what the user is saying regardless of the language used in the input; it can be in your own regional language in the most natural way, like the one shown in Figure 1-5. Conversation-based user interfaces make us realize that it is rather easy to book a flight or buy a shoe using more human-like interactions with a bot that is aware of my preferences and can even complete the payment on my behalf. It is also easier to build a conversation-based application because it does not contain any rich text, styling, or images to woo the user; all we need is to build an intelligent and simple conversation pattern that resembles a human.

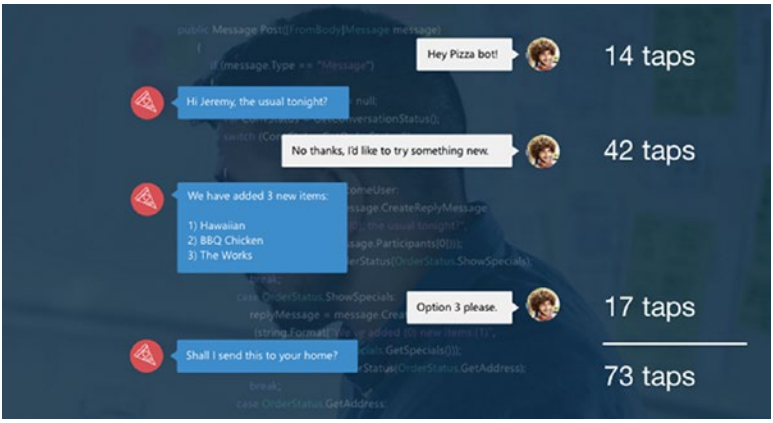

Figure 1-5. *Example of typical conversation-based user interface*

The most selling feature of CUIs is that everything happens right in front of you. You do not have to switch screens, navigate, or even scroll up and down to figure out a way to interact with the application. A simple gesture like Yes, No, or OK can get things done for you. CUIs are expected to be intelligent and context aware, and it makes no or less sense to have a CUI with limited set of responses.

A bot is a CUI-based software application that is automated to do a predefined set of tasks using human interactions (text and voice) through any medium, like a browser, desktop application, or phone. Bots are here to stay and are going to replace mobile apps. Like mobile applications replaced a lot of web-based applications, bots might replace mobile applications in the future with applications that can be interacted with using voice, text, or image. There are two categories of bots that are being built today: chat bots and AI bots. Chat bots are generally rule based. It is easier to build chat bots than AI bots, and they also consume less infrastructure and have lower costs. Most chat bots serve a single purpose, like a bot that imitates a pizza-ordering helpline with a limited set of menus.

The second type of bots are the artificial intelligence or AI bots. These bots are like chat bots, with the only difference being that they are backed by an artificial intelligence algorithm(s) that can predict your next action in an application based on usage pattern, or can recommend a similar item based on current selection and by analyzing what other users have bought together. Imagine just saying, "Repeat pizza order," and having a pizza delivered to your current location from your favorite pizza store. Most AI bots contain natural language processing and deep-learning capabilities. AI bots can sense the tone of the conversation and respond accordingly, which chat bots cannot do, as they are programmed with default responses irrespective of the tone and context. If you ask a chat bot an unrelated or random question, it might just deny the request or not respond at all, but an AI bot would try to analyze the question and answer as human. It would learn from the interactions so that it could respond to similar random questions. One more key distinguishing feature of AI bots is the source of data that drives decision making. Let us say you have a meeting scheduled for tomorrow 9 a.m. at Place A and you received an invite for lunch at Place B, which is about 10 miles away. An AI bot should be intelligent

enough to reject the invite because the traffic conditions will prevent you from reaching the lunch on time. It should also be able to auto-respond on your behalf based on the decision taken by the bot.

Bots are still emerging. As of today, there are only a few bots, which mimic web applications or mobile apps. The beauty of bots is that we do not have to build something from scratch or worry about installing it on a client's machine. Bots can be integrated into existing message platforms like Skype, Facebook Messenger, Slack, and so on. A few years ago, technologies like artificial intelligence and machine learning were out of reach to most developers because of the effort involved in learning the language and the semantics. Microsoft Cognitive Services has helped us overcome these challenges by introducing a multitude of intelligence-based APIs that are effective and easy to consume.

Introduction to Microsoft Bot Framework

The Microsoft Bot framework is a complete suite to build intelligent and intuitive bots that will be reachable via familiar communication tools like Skype, Slack, Teams, Office 365 Email, and other popular ones without any additional effort required. Communicating with a bot resembles a human-to-human communication and occurs in various forms, like sending a text or an email. The framework consists of a powerful Bot Builder SDK, Bot connector service, a developer portal, and a bot directory. The Microsoft Bot SDK is available for both Node.js and C# developers; for other languages, developers can use the REST API to build intelligent bots. The framework provides support for user management, session management, state management, authentication, and conversation models like dialogs, activities, cards, or attachments. The Bot SDK is open source.

To add more human-like conversation features to your bot, you can integrate with Microsoft Cognitive Services, which provides vision APIs for image processing, speech APIs for voice-to-text translation and vice versa, language APIs for language conversation, Search APIs for including Bing search in the results, and knowledge APIs for building recommendations based on a user's previous usage.

When you are done building your bot, you register it with the Bot registry and configure connectivity with a variety of channels before you finally publish. Figure 1-6 shows a few channels currently supported by the Microsoft Bot framework.

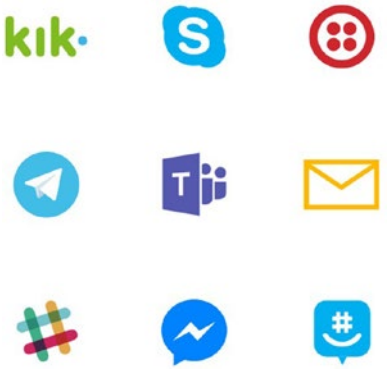

Figure 1-6. *Bot channels*

Microsoft provides a bot developer portal to connect your bot to various channels and test on those channels. The Bot Connector service uses the REST API and JSON schema to communicate with the Bot API. Once you have configured your channels and published your bot for testing, the bot ends up in a bot directory (Figure 1-7). A bot directory is a list of bots published by developers from across the globe. You can search for and connect to any bot available in the directory by using your favorite channel.

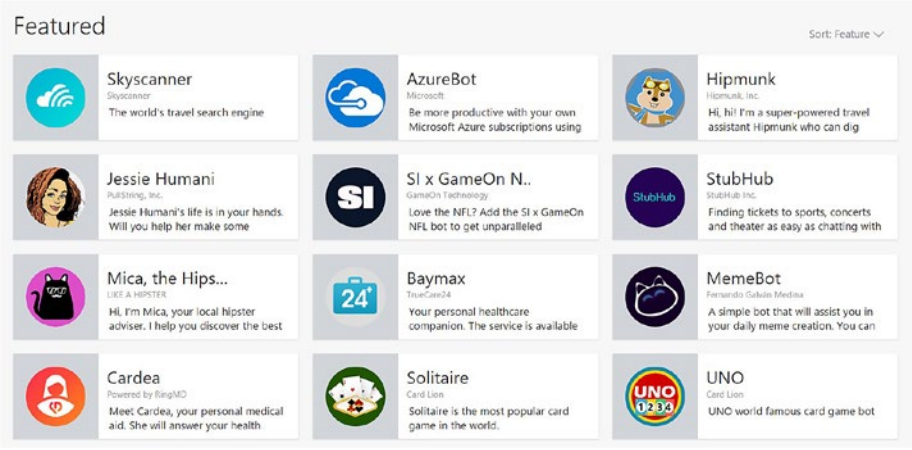

Figure 1-7. *Bot directory*

Every bot in a directory is configured with supported channels, so you can just click on any bot and connect using the configured channel. For example, the UNO Bot only contains Skype as a channel, so you can click on UNO Bot and then click on Add to Skype to add the bot to your Skype bot contacts list. The latest version of Skype, called Skype Preview, isolates the bots into a separate bot contacts category. You can also search

for bots using the Skype Search feature like any other user. The latest version of Skype Preview can be downloaded from the Windows App Store.

Meet a Few Bots

The bot directory consists of many featured bots that are designed for a specific cause—and some bots are just fun to chat with (specially the AI bots)! In this section, let us meet a few interesting bots and get a feel for the arena before we start designing bot applications. The bot directory is available under the Bot Directory tab in the Bot Developer Portal, found at `https://bots.botframework.com/`.

Summarize

If you are feeling too lazy or could not find the time to read a lengthy blog, Summarize Bot can help by providing a summary of the blog or any web page. The user can paste the link of the blog, and Summarize Bot will summarize the blog and list the important points. There are a couple of ways you can interact with Summarize Bot. From Skype Preview, you can search for Summarize Bot and add it to the contacts list, or you can visit the bot directory and use the web chat window. Figure 1-8 shows a sample conversation with Summarize Bot using Skype Preview.

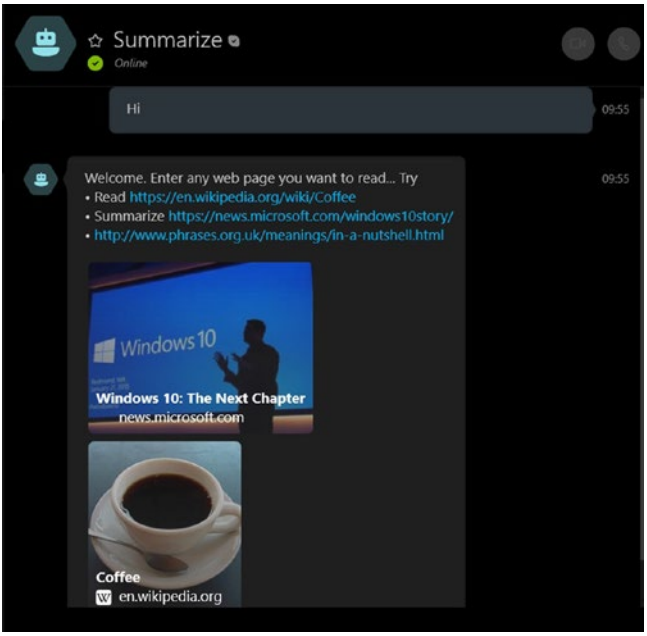

Figure 1-8. *Summarize Bot*

Figure 1-9 shows the response from Summarize Bot when any link—for example, `https://docs.botframework.com/en-us/azure-bot-service`—is sent as a request.

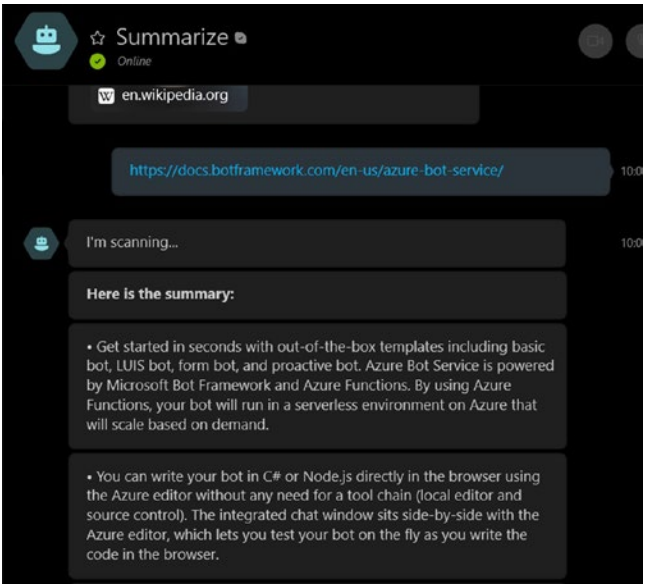

Figure 1-9. *Summarize Bot response*

Feedback from users is critical if an application is to be improved. Summarize Bot follows best practices by seeking information from the user as to whether it did well.

Your Face

Your Face is an AI-powered bot that uses Microsoft Cognitive Services to assess the face in the image and predict the age. **Your Face** is available on a variety of channels, like Skype, Telegram, Kik, email, GroupMe, and Facebook Messenger. Figure 1-10 shows an interaction with the bot using Microsoft Outlook. You can send any picture with a face as an attachment to `yourface_bot@outlook.com`.

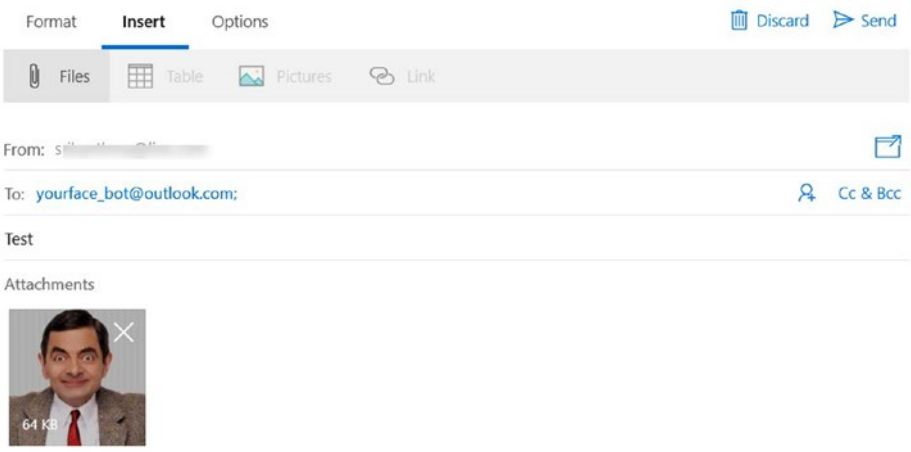

Figure 1-10. *Interacting with Your Face AI bot using email client*

Within a couple of minutes, the bot responds with an email. As you can see in Figure 1-11, the bot puts a name to the face since it is a familiar one; at the same time, it predicts the age of the face. You can try with any of your personal images and have some fun.

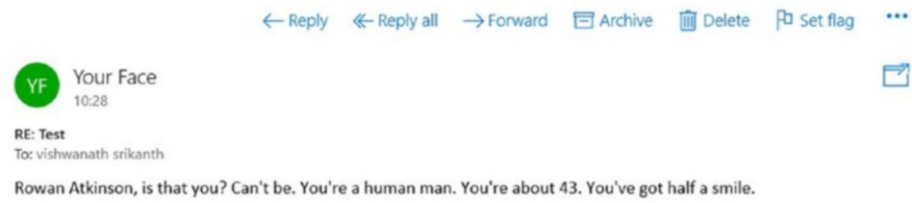

Figure 1-11. *Sample response from Your Face AI bot*

Azure Bot Service

Azure Bot Service is a PaaS (Platform as a Service) offering from Microsoft that is available as part of the Azure subscription. Azure Bot Service enables rapid application development powered by the Microsoft Bot framework and runs in a serverless environment on Azure. Azure Bot Service allows your bots to scale on demand and pay only for resources you consume. To create an Azure Bot Service, one would need an Azure subscription. You can buy one or create a free trial account for learning purpose from here: `https://azure.microsoft.com/en-us/free`.

Azure Bot Service provides an integrated development environment that helps you register the bot right from the Azure portal and allows you to author code with boilerplate templates. Figure 1-12 shows the bot configuration on Azure Bot Service.

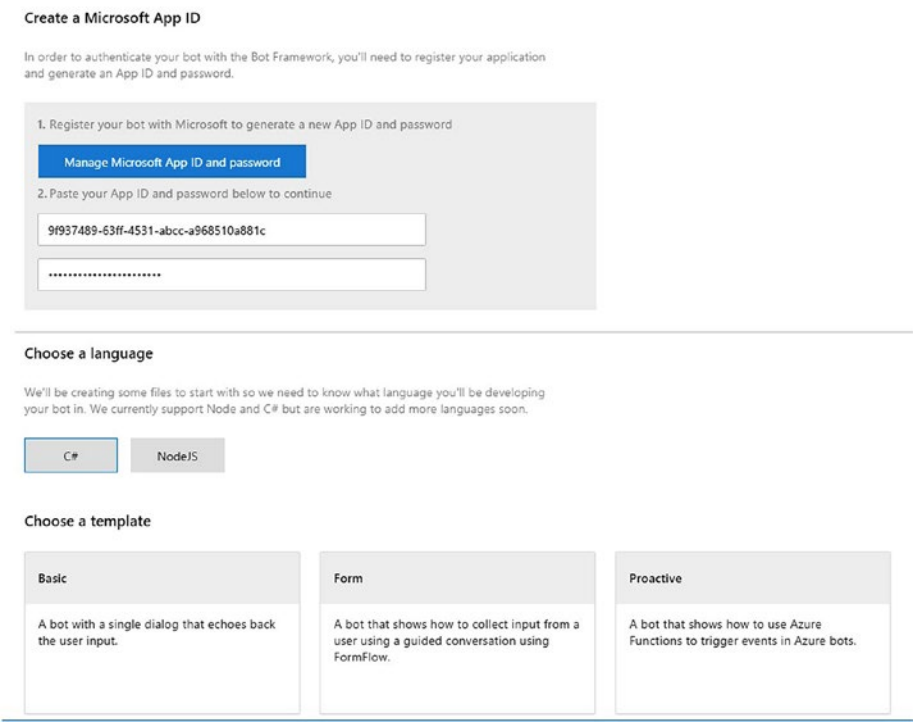

Figure 1-12. *Azure Bot Service*

You can select your favorite language and start developing bots by choosing any existing template. Azure Bot Service offers boilerplate templates, from simple bots to intelligent ones like those with natural language processing, proactive bots, and question and answer bots. Since it is on Azure, there is no overhead when managing servers or even patching. The bot can be scheduled to scale based on events powered by Azure Functions. By using Azure Functions, your bot runs on a completely serverless environment that scales on demand. Azure Bot Service provides in one place all the required resources for development, channel configuration, bot settings, a web-chat interface for testing, and a publishing service for publishing, as shown in Figure 1-13.

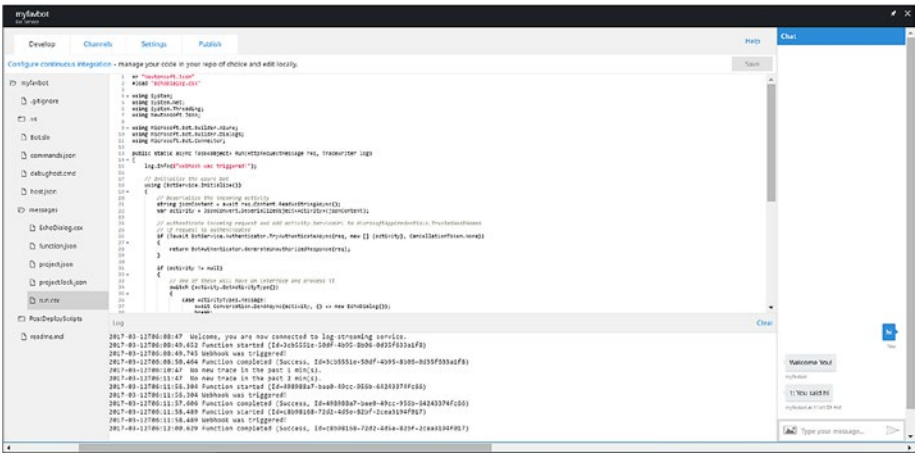

Figure 1-13. *Azure Bot Service development experience*

From Azure Portal, we can set up Continuous Integration from GitHub, Visual Studio Team Services, and many more. We can use the code from the web interface as a start, but remember that we cannot modify the code in Azure after setting up Continuous Integration. Continuous Integration and Delivery options let you deliver the application code at a rapid pace so the code gets built and deployment for every commit from the developer.

The following are a few intelligent bot templates available on Azure Bot Service.

LUIS Bot

For an intelligent bot, it is important to understand the user's conversation in the natural language. LUIS, which stands for *Language Understand Intelligent Service*, helps you identify the intent and entities in the conversation and map them to pre-defined HTTP endpoints. For example, in a statement like "get score about India versus England cricket," the bot should be able to get the intent, which is "get score" and the entities which are India, England, and Cricket LUIS enables you to design HTTP endpoints and map the user conversation to HTTP endpoints. For more information on integrating with LUIS, please visit `https://www.microsoft.com/cognitive-services/en-us/luis-api/documentation/home`. You will learn more about building bots with LUIS integration in Chapter 9.

QnA Bot

Most businesses have a QnA or FAQ section that helps users find answers to repetitive questions on the business model or on how to use the application. The bot template for QnA allows you to quickly create a FAQ or QnA bot that can answer a user's queries about your business via various channels using existing FAQ content as the knowledge base. QnA Maker (`https://qnamaker.ai/`) lets you ingest your existing FAQ content and

expose it as an HTTP endpoint. You can also build a new bot with an empty knowledge base. Figure 1-14 shows a sample QnA bot made from a FAQ URL knowledge base.

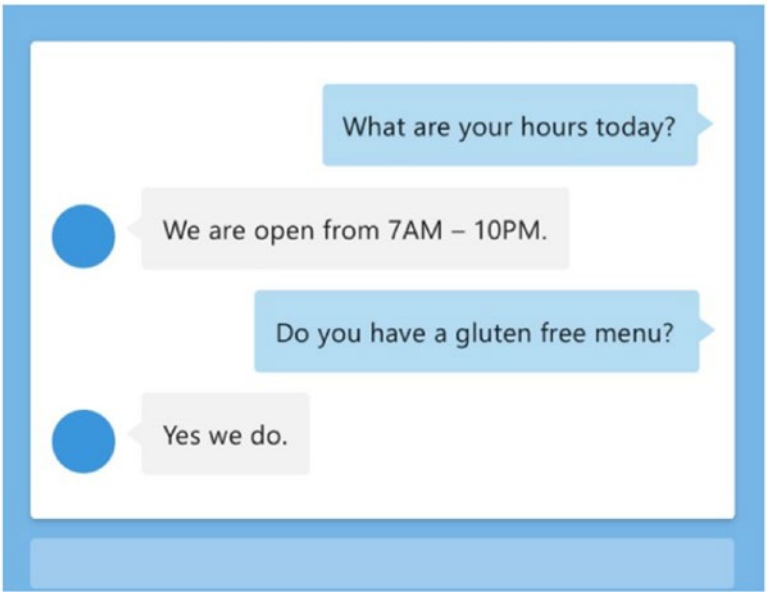

Figure 1-14. *Sample QnA bot*

Proactive Bot

The Proactive Bot template helps you in scenarios where you want the bot to initiate a conversation. The bot can initiate a conversation based on some triggered event, lengthy job, or external event like updating a cricket score on completion of a bowler's over. The Proactive Bot template uses Azure Functions to trigger an event when there is a message in the queue. Azure Functions then alerts the bot via the Direct Line API. The proactive bot template creates all the Azure resources you need for enabling the scenario. Figure 1-15 shows an overview of how multiple Azure resources communicate to trigger an proactive conversation.

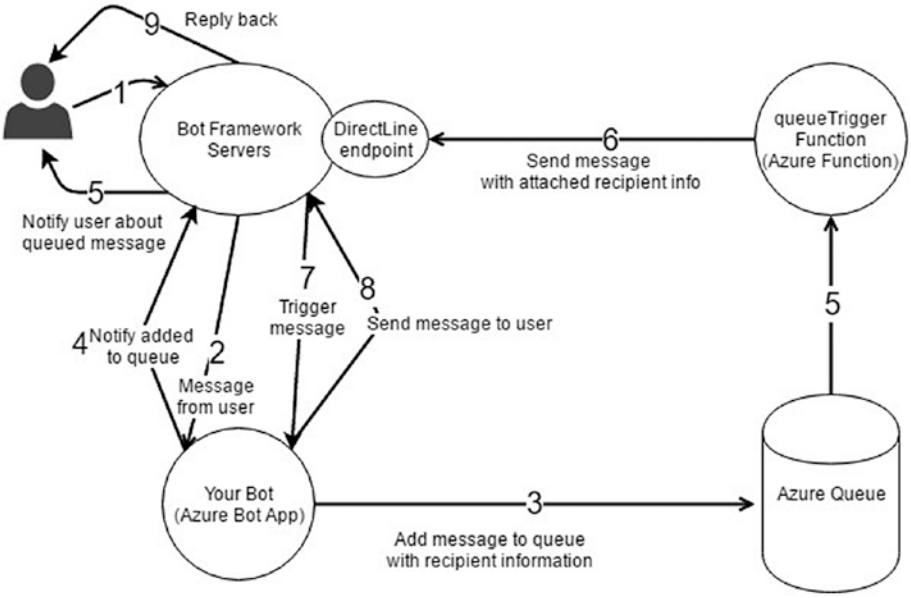

Figure 1-15. *Proactive Bot design*

Direct Line

The Direct Line API is a simple REST API that allows you to initiate a conversation with a Bot. This helps developers write their own client applications for web, mobile, or service-to-service. The API has the ability to authenticate using secret tokens or token patterns. Using the Direct Line API, you can send messages from your client to your bot via HTTP post messages. In addition, you can receive messages from the bot using Web Socket Stream or by polling mechanism. If you are planning to build any custom smart device that responds to a user's voice inputs or gestures, you can use the Direct Line API to send and receive messages from the bot, backed up by AI services from a custom IOT device (Figure 1-16).

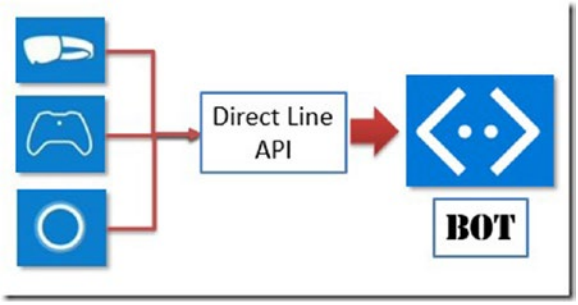

Figure 1-16. *High-level design for designing bots with custom clients*

IOT and Bots

Bots can integrate with IOT (Internet of Things) devices to make our lives much more sophisticated. Microsoft Azure provides services for building smart IOT devices, like Azure IOT hubs that can process millions of IOT assets. For example, imagine you are running a restaurant. A bot can proactively ping you with an image of a guest with the details like name, gender, age, recurring customer, favorite food (by using the Bing Search API and facial-recognition APIs) using a IOT device installed at the entrance. This model can be further extended by using the Azure ML, Stream Analytics, and Power BI to build a recommendations-based menu for recurring guests. Such is the power of Azure and the services provided by Microsoft as part of Azure.

Microsoft Cortana, Apple Siri, and Amazon Echo are a few examples of smartness being built into your handheld devices using various sources of information. With the Azure Bots framework and IOT hubs, you can use any existing device to build smart apps that accept voice input and respond with a voice or text response that includes information from a wide variety of sources.

Other Bot Frameworks

It is worth mentioning the other bot frameworks that are like the MS Bot framework and provide a similar feature set; for example, Wit.ai, API.ai, and Viv. The April 2016 Facebook Bot engine release is based on Wit.ai, which it acquired in 2015. Wit.ai runs on a server hosted on the cloud. The Facebook Bot engine is a wrapper that helps developers build bots for Facebook Messenger only. Wit.ai works like LUIS, as it helps extract intents and entities and it also comes with few pre-defined entities. API.ai also works along the lines of defining entities and intents, though the defining feature of API.ai is its reachability. API.ai bots can be exported as modules and integrated in Facebook, Kik, Slack, Alexa, and even Cortana. Viv.ai is from the authors of Apple Siri, and its focus is to process complex queries using a flexible mechanism. Viv follows the SQL query-processing approach as application logic, which involves breaking the request into constituents and combining them into an execution plan. Viv, as claimed by the makers, is more suitable for building virtual assistants than an enterprise bot.

Bot Abuse

Bots are aimed at replacing the mobile applications with interactions which are more human like. Some of these bots focus on delivering simple models, like QnA bots or an ordering system. But some bots are built using complex conversation pattern, and those bot responses are completely based on artificial intelligence, like Microsoft Tay (https://en.wikipedia.org/wiki/Tay_(bot)). Microsoft Tay is a Twitter-based bot that created controversy immediately after launch when it began to respond with inflammatory and offensive tweets over Twitter. This made Microsoft shut down the bot within 16 hours of launch. It was evident from the experience that artificial intelligence bots are prone to be attacked like any other application and are trained to respond in a derogatory way. It is the bot designer's responsibility to make sure the bot is tried and tested with all sorts of conversation patterns and that it responds safely when abused. Bot technology is still new, and it may take some time to define, practice, and standardize the best security standards or patterns to follow in order to avoid such circumstances. But it is undoubtedly a powerful platform that can enrich the life of every individual and organization if adapted appropriately.

Summary

GUI interfaces are restricted by screen space and often involve large teams that are responsible for designing better-looking apps, which is quite a challenge. Bots let your team focus on business logic and conversation design rather than a look and feel. User conversations with bots are intended to be more like human-to-human conversation. For a bot to respond in a more human-like way, it should be integrated with artificial intelligence or machine-learning algorithms. The Microsoft Bot framework allows you to build smart bot applications with voice, knowledge, and search-based intelligence in just a few steps. Microsoft offers the Bot Builder SDK for languages like C# and Node.js, Bot Connector Service, Bot developer portal, emulator, and bot directory to aid design, build, test, and manage bot-based applications. It also provides a REST API that can be used to build bots using other languages. Microsoft Cognitive Services provides a multitude of intelligence APIs that can be easily integrated with the MS Bot framework to add the necessary intelligence to bots. Azure Bot Service is a PaaS service that provides a singular platform with which to design, code, and deploy bots with artificial intelligence and infinite scalability from the Azure portal. Azure Bot Service can be a quick-start guide to quickly set up a bot application using existing boilerplate templates and set up CI and CD right from the Azure portal. Developers can also build their own smart clients or IOT devices that can communicate with bots directly using the Direct Line API.

CHAPTER 2

Develop Bots Using .NET Core

Building a bot is no different from building any other typical web application. You start by defining the user scenarios, then you design the user interface (UI) screens, where you capture the required information that flows from screen to screen. Building a bot falls along the same lines, except that the scenarios are implemented as conversations. The scenarios can be implemented in various ways in a bot application. User information can also be captured in various ways, like speech, text, image, or any form of gesture. The emphasis when designing a bot should be on making the intent discoverable and simple. The Microsoft Bot framework offers everything that is required to make a conversation intuitive and appealing. It is a complete suite that helps you build high-quality bots using existing tools like Visual Studio, VS Code, and programming languages like C#, Node.js, Java, PHP, and so on. As described in the previous chapter, building a bot consists of four main stages: designing and developing your bot using the Bot Builder SDK; configuring channels using Bot Connector; registering your bot via the Developer Portal, and making it discoverable via a bot directory. Microsoft provides bot framework libraries in C# and Node.js; for the rest of the languages, developers can use the HTTP Rest API. This chapter will focus on introducing bot-building strategies for C#/.Net developers, publishing to Azure, using the pre-built tools, testing your bots, and configuring channels using the MS Bot framework.

The following topics will be discussed in this chapter:

- Designing bot applications
- Setting up a development environment
- Understanding the bot life cycle and components
- Running and testing using Bot Emulator
- Deploying bot to Azure
- Registering bot using the Developer Portal
- Testing the use of bot channels like Skype and embedded web controls

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_2

Designing Bot Applications

How do we design a smart bot that stands out from alternative applications? It is very critical to learn the basic etiquette behind building a successful bot, but let us first review what makes a bot smart and successful. Every bot that is built using smart features like speech to text, natural language processing, or image processing need not be acknowledged as smart. A bot can still be liked by many if it can solve the user's problem in a few easy steps, is amicable to communicate with, and is reachable by all means used by your target audience. It need not always integrate with sophisticated features like Azure machine learning.

Here are a few best practices you should follow when designing a bot:

- A bot makes its intent clear in the first impression. Unlike web or mobile applications, there is not much surface area for a bot to explain its primary feature set, which makes it both difficult and important to let the user know what the bot can do.
- A bot should let the user navigate in a non-linear fashion. As in any type of application, users do not always navigate from Step 1 to Step N in a linear fashion; they might jump back and forth. Although it is difficult to achieve the same thing in a conversation, you should design your bot to allow the user to navigate in a non-linear fashion.
- Always provide help so that at any point the user can seek information on the current state and the next steps. Bot-style applications are new, so novice users might not understand how a bot operates. Providing timely help is very important to make the bot discoverable.
- Respond to the user's input immediately if there is any background processing involved. Provide an immediate acknowledgement and respond with an update on the background process later.
- A bot should always try to seek as little information as possible from the user. You can use the user's past conversations or personal data to fill in basic queries. For example, when searching for restaurants, you can show the restaurants near to the user's current location.
- A bot should always seek the user's permission before storing or accessing personal information. If requested by the user, the bot should permanently remove all the personal information belonging to the user.

What you should *not* do when designing a bot? See the following list:

- Do not force the user to provide a specific input for proceeding further. If you are expecting a specific input from the user, set maximum retries and fall back.

- Do not provide obvious responses to the user; for example: "Hi, Jack. Looks like you are driving back home; have a safe journey." Always provide information that is useful.
- Do not enforce natural language processing for every scenario that might impact the performance of the application. Whenever possible, ask specific questions and use regular expressions for parsing the message instead of AI APIs like LUIS.
- Do not use past conversation to confuse the user. For example, if the user is trying to book a flight, his past bookings might or might not help him. Using the conversation history in the context of current conversation is a conscious decision to make.

These items are only a few do's and don'ts to keep in mind while designing a bot. The areas where the bot type of application makes sense are varied; in most of cases, the designs vary by the scenario at hand. Use the preceding points as basic building blocks and improvise as required.

Now that we know what a smart bot should contain, let us learn the designing methodology. In any web application, we would start by building the screens. Let us say, for example, I want to build a bot that helps in setting doctor's appointments. I would design a screen something like that shown in Figure 2-1.

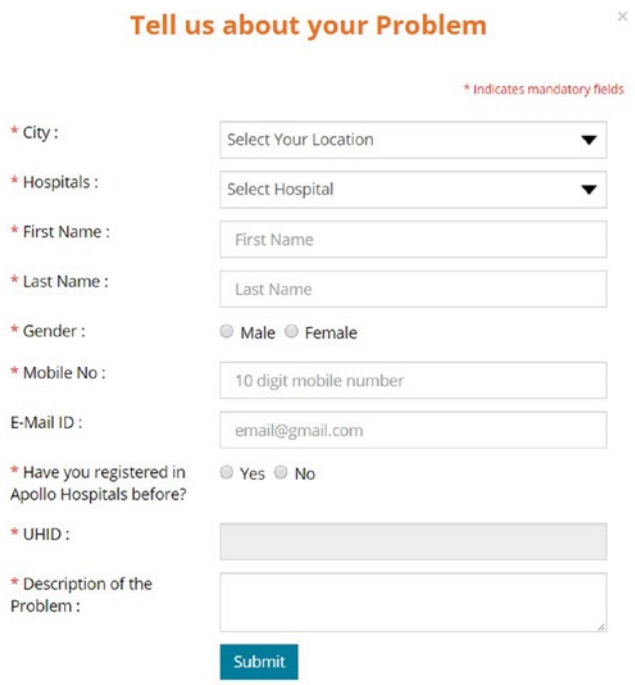

Figure 2-1. *Sample GUI for scheduling an appointment with a doctor*

In a bot application, we design conversations, or dialogs. Every bot application starts with a root dialog, where we introduce the features of the bot and show a menu of possible things the user can do. The subsequent dialogs are added one by one to the dialog stack. The dialog at the top of the stack is in control of the conversation until it exits the stack. The root dialog never exits the stack. This process is shown in Figure 2-2. Let us see how we can seek some of the information seen in the form in Figure 2-1 from the user via a series of questions.

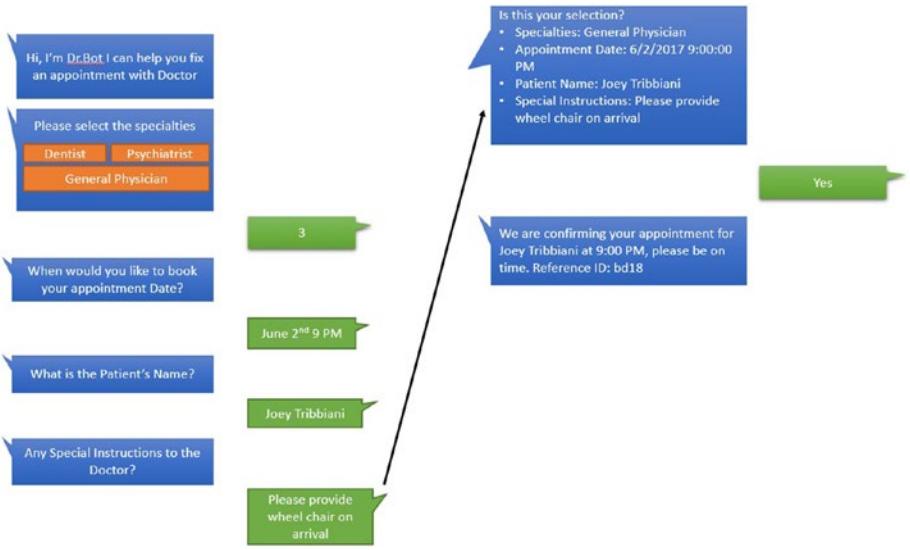

Figure 2-2. *Designing a conversation flow using dialogs*

The preceding conversation captures basic information required for booking an appointment.

Let us now design the navigation for the bot application. As shown in Figure 2-3, each step seeks information from the user and provides a way to navigate back in a non-linear fashion. At every step, the user can seek help regarding the current state. There can be any number of steps in a conversation; at the same time, it is important to keep it to a minimum in consideration of the user experience. Remember, this is one of the many ways of designing a bot. You can let the user provide voice input, upload images, or select from list of options, or you can extract more information from a user's input. For example, the statement "I want to book a **General Physician** for **myself** on **June 2nd at 9 p.m.**" contains all the information required to book an appointment. It can be parsed using LUIS to extract user intent, as marked in bold.

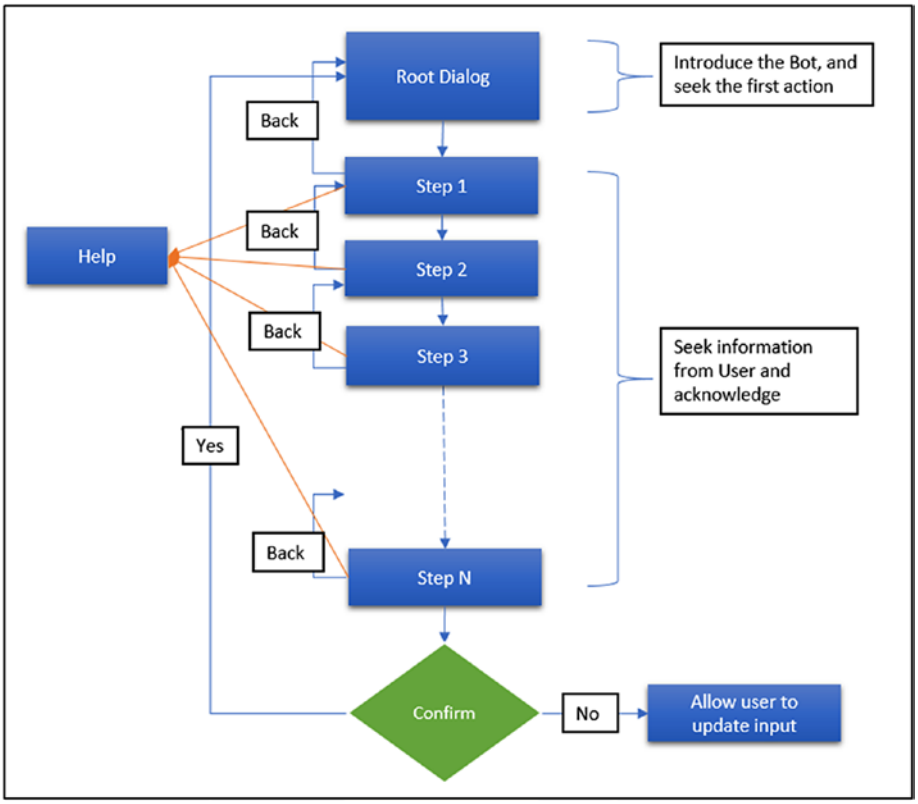

Figure 2-3. *Sample bot conversation flow chart*

The following sections teach the basic development aspects involved in building a bot application.

Setting Up the Development Environment

For developers who have invested time in learning .NET methodologies for building RESTful web APIs using C# and Visual Studio IDE, the development experience will be very familiar. You can continue using your favorite .NET IDEs, like Visual Studio (Community, Professional, or Enterprise), and Visual Studio code to build bot applications. If you are coming with no prior development experience, do not worry. This chapter teaches everything from scratch. To develop bots using .NET technologies, the following set of tools and SDKs are required:

- Visual Studio 2015/2017 (Community, Professional, or Enterprise)
- Visual Studio Template for C#

- Bot Builder SDK
- Bot framework emulator for testing

The following steps show how to set up the development environment for building bot applications:

- Download and install the Visual Studio Community version from `https://www.visualstudio.com/downloads/`. Since the MS Bot framework is still in the preview phase, the project template required for building bots is not part of the Visual Studio installation.
- Download the bot application template from `http://aka.ms/bf-bc-vstemplate`; the download contains a zip file named `BotApplication.zip`.
- Copy the zip to the Visual Studio Templates folder for C#. The default location for templates is %USERPROFIE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C#\. (This path is for VS 2015; for other version like VS 2017, use the Visual Studio 2017 Templates folder).
- Open Visual Studio as Administrator and click on New Project. Search for "Bot Application" using the search section at the top right corner. You should now be able to see the project template for bots in the Templates section, as shown in Figure 2-4.

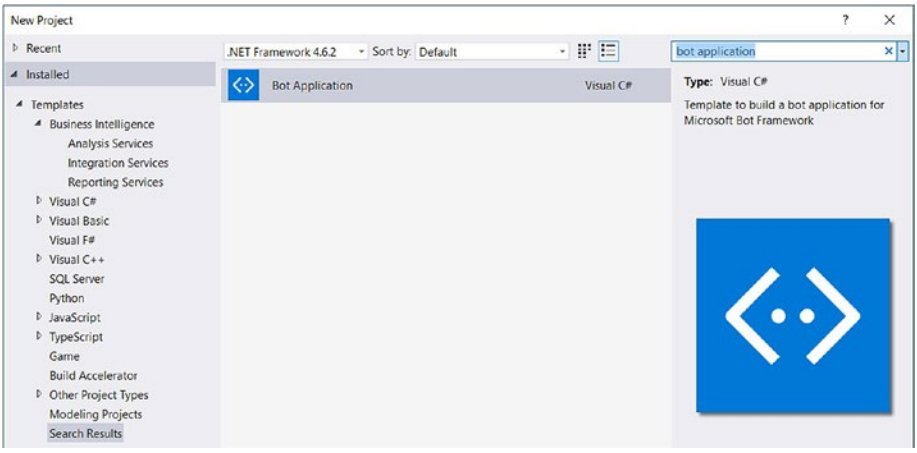

Figure 2-4. *Visual Studio bot application template*

The template creates a fully functional sample bot called EchoBot, which can be used to test the basic development approach. The sample application simply echoes the number of characters in any text entered by the user. Go through the following steps to understand how a bot application functions.

- Name the sample EchoBot and click OK to create your first sample bot application.
- Ensure the project builds successfully using Ctrl + Shift + B. Press Ctrl + F5 to run the bot (without debugging).
- Ensure Visual Studio automatically invokes page `http://localhost:3979/` using your default browser; the port number may vary from machine to machine.
- Ensure the landing page for EchoBot looks like the one in Figure 2-5.

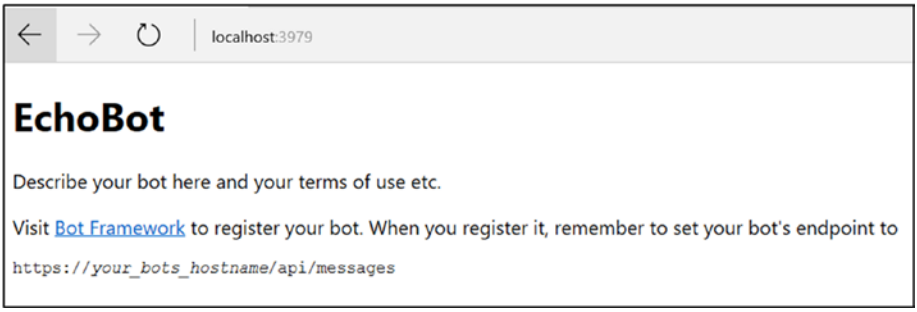

Figure 2-5. *Bot application start-up page*

There are a couple of ways to interact with bots; one way is by installing Bot Emulator and the other is by building a custom client application to interact with the bot. Testing the bot application with an emulator is the easiest and most recommended approach. Bot Emulator is a desktop application that is built to interact with and test bots. It is available for both Mac and Windows machines.

Download and install Bot Emulator from `https://emulator.botframework.com/`. If you encounter a security alert asking for permission (as shown in Figure 2-6) to allow Bot Emulator access to localhost, go ahead and approve by clicking on Allow Access.



Figure 2-6. *Windows Firewall security alert for Bot Emulator*

Figure 2-7 shows Bot Emulator's interface for testing and debugging bot applications during development.

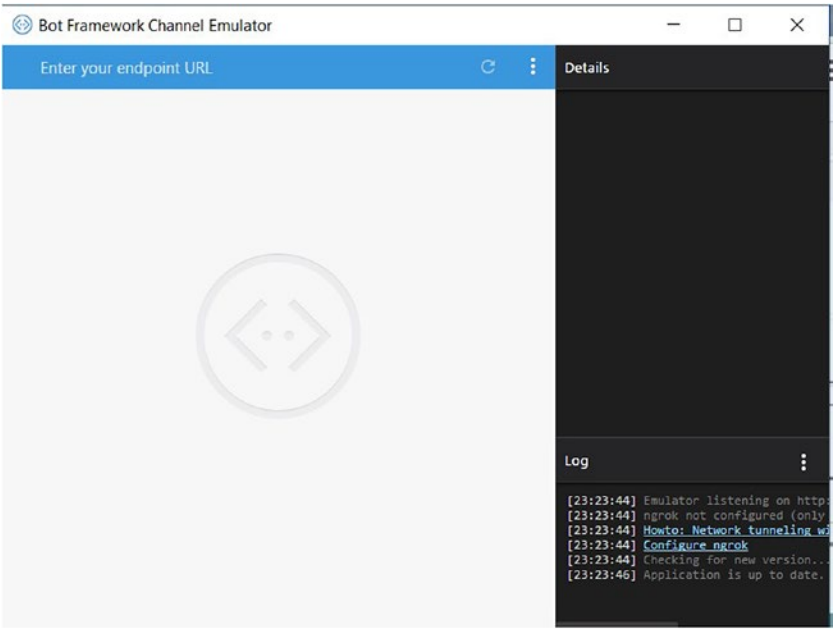

Figure 2-7. *Bot Emulator interface*

Testing the Bot

To interact with EchoBot using Bot Emulator ensure the application is running. If you are using Windows Machine and IIS Express, you can check for running applications from the system tray.

Copy the endpoint URL of our API to the emulator in the "Enter your endpoint URL" section. The endpoint URL is a combination of the base URL assigned to the API—for example, `http://localhost:3979` appended with `api/messages`. The complete endpoint URL, in this case, will be `http://localhost:3979/api/messages`, where `api` is part of the default route and `messages` is the name of the controller that contains the default endpoint that accepts the messages sent by the user. All of this is provided by the sample template.

Enter "Hello Echo Bot" in the conversation window and observe that the bot immediately responds with, "You sent Hello Echo Bot which was 13 characters."

The Log section on the emulator shows the communication log between the emulator and the bot (Figure 2-8). When you enter the bot endpoint URL in the URL section and hit Enter, the emulator tries to perform an initial handshake with our bot. The Log section shows the details of the communication between your bot and the emulator at this step.

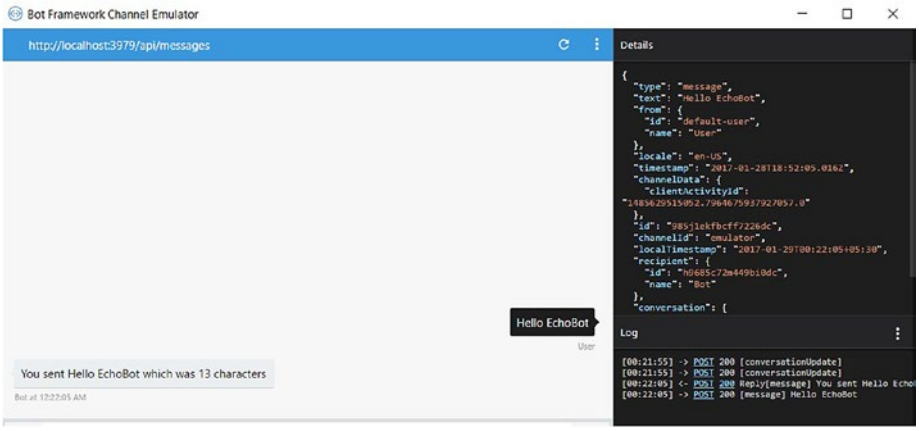

Figure 2-8. *Bot Emulator with communication logs*

Clicking on the HTTP response code, like 200, shows the payload sent/received during the conversation in JSON format.

Clicking on an HTTP header like POST on the Log section of the emulator shows the HTTP post request and response. We can also debug bot applications from Visual Studio like any other web API application.

Debugging the Bot Application

The application should be running in debug mode (Figure 2-9).

- Run the bot application in debug mode (press F5).
- Switch to Visual Studio and create a breakpoint on `MessageController` at Line 21.
- Use Bot Emulator to initiate a conversation.
- The process should now stop at the debug point when invoked from the emulator. Visual Studio also shows the exact message received by the application, as shown in Figure 2-9.



Figure 2-9. *Debugging bot application from Visual Studio*

We have successfully set up our development environment to build, develop, and test bot applications. Now, let us dig deeper into the project template and learn the life cycle of the bot.

Bot Application Life Cycle

The bot application is a simple web service that responds to messages posted by the client targeting the bot's endpoint, which means the life cycle of a bot application is very similar to that of a web API application. The application receives and responds to HTTP `POST` requests using the JSON message format. The JSON-to-object serialization is taken care of by the framework. Like any.NET web application, `Global.asax.cs` drives the bot life cycle. The following code shows the default code of `Global.asax.cs`:

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

    }
```

During the life cycle of a bot, the preceding class gets instantiated. This class provides a variety of events that can be used to handle any critical tasks before or after the life-cycle event (Figure 2-10).

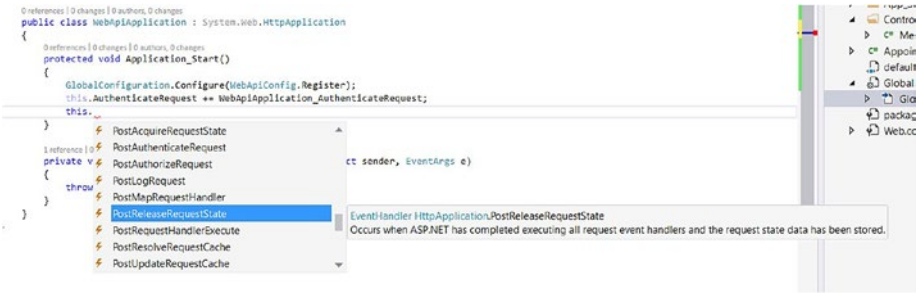

Figure 2-10. *Life cycle events of bot application*

In the preceding example, an `Application_Start` event is used to load the application configuration by invoking the `GlobalConfiguration.Configure()` method, which loads the configuration from the `WebApiConfig` class under the `App_Start` folder. The `register` method of the `WebApiConfig` class is used to configure the routes, services, handlers, formatters, and so on. The following excerpt from `WebApiConfig` configures the default routing for the bot application using HTTP attributes like `POST` and `GET` (these are the most commonly used HTTP attributes or verbs. For a full list of HTTP verbs, please visit https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html). This method can be used for registering handlers, configuring IoC containers, or for any bootstrapping activities.

```
// Web API routes
config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

The preceding code configures the server to forward the incoming messages from bot clients to the `Controllers` class. For example, if a user makes a `POST` request to `http://localhost:3979/api/messages`, forward the request to a method tagged as `POST` on `MessagesController`, and optionally the user can pass the `Id` parameter to the method (Figure 2-11). In a bot application, requests from any clients are always `POST` calls (unless a custom client is implemented). For `POST`-type calls, the request body is not included in the header; instead, the request body contains the complete request payload.



Figure 2-11. *Post request to bot application*

Controllers are endpoints or receivers of a user's message. There can be more than one controller in a bot application, and every controller should extend from `System.Web.Http.ApiController`. For bot applications using the default channels like Skype, Teams, and so on, there will always be one controller that accepts all types of messages from the user. The following code shows the `POST` method added by the project template by default.

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

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

The bot application instance constructs the `activity` parameter from the details of the HTTP request. The `activity` object contains the following details:

- **Channel type**, like Skype, emulator, email, or SMS
- **Unique ID** for the request
- **Activity type**, which classifies message as a conversation update, typing type, or ping type, depending on the type of message the response from the Bot can be constructed.

The details of this object are used to construct a response to the user, which is returned as `System.Net.Http.HttpResponseMessage`. The response can be anything from simple text or a hyperlink to an image or initiating an audio call. The bot responds to the user's `POST` requests infinitely until the application is terminated. The responses from the bot can be a reply or a prompt response seeking more information from the user.

Bot Builder SDK is the underlying framework that is used to build custom bots. The SDK is available as a NuGet package and can be added to the solution as shown next. It is a powerful framework for constructing bots that can be used to create freestyle interactions like the one we saw earlier and more guided ones like Dr.Bot, which we will build soon. The following entry in the `packages.config` section installs the NuGet package to the solution:

```
<package id="Microsoft.Bot.Builder" version="3.5.1" targetFramework="net46" />
```

You can also run the following NuGet command to install Bot Builder SDK from the package-manager console:

```
Install-Package Microsoft.Bot.Builder
```

Bot Builder SDK takes care of constructing a channel object to send constructive responses to the client(s). For example, the following code creates a connection back to the client using the service URL (the address of the client) and sends a simple string using the pre-built method provided by the framework.

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

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

Bot Architecture

Bots are designed to solve common business problems using a natural, conversation-style approach combined with machine learning for advanced intelligence. Microsoft's vision in introducing the bot framework is to allow developers to design applications that provide a personalized experience and texture of talking to human intelligence. To facilitate this, Microsoft has pooled a stream of existing technologies that are already familiar and available on most devices. Microsoft Cloud platform offers a multitude of opportunities, like Azure Active Directory, Logic apps, Big Data, IoT hubs, and Microsoft Cognitive Services, for integration with bots; these applications have the potential to provide a rich, reliable, and consistent conversation style experience to the users.

Microsoft Bot framework's architecture is quite simple: the basic building blocks are the channels, Bot Connector, bot application, and the optional artificial intelligence component, like Microsoft Cognitive Services (Figure 2-12).

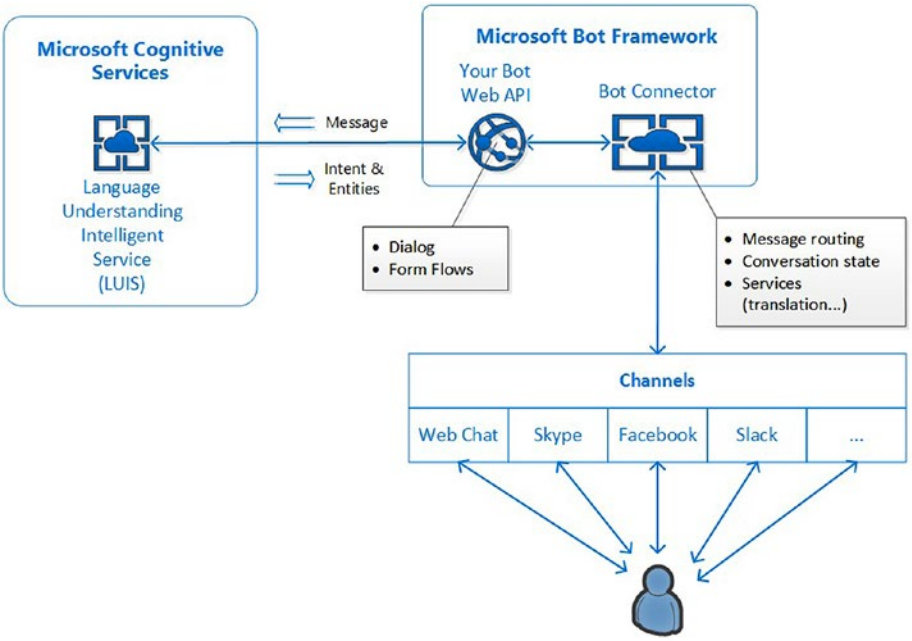

Figure 2-12. *Bot architecture*

Channels are what a client's application uses to connect to the bot; for example, Skype for Business. Bot Connector acts as an adapter between the various channels and your bot application, taking care of bot authentication, serialization with the wide variety of channels shown in Figure 2-12, conversation state, user-state management, and automatic language translation (nearly 30-plus languages). Bot Builder SDK is a free and open source framework that helps you build your bot using .Net/Node.js. The MS Bot framework provides built-in classes that provide a rich user experience, like clickable buttons, hero cards with image and text, custom dialogs, form flows, and audio-style conversations. Bot applications can be hosted anywhere on the web, although Azure is preferable because of the integration and monitoring capabilities provided out of the box.

Bot Authentication

When you create a bot application, the template adds an attribute to the `controller` class called `BotAuthentication`. Adding the `BotAuthentication` attribute to the controller enforces security and makes sure only authenticated and registered bots

can communicate with the bot application. The authentication is enforced using the attributes `MicrosoftAppId` and `MicrosoftAppPassword`. These two attributes are typically stored in the `Web.config` file of the bot application.

■ **Note** The attributes can be left blank during development, or you can also remove the `BotAuthentication` attribute from the `Controller` class(es), but before we publish the bot to the Developer Portal, the values should be replaced with the actual ones.

Every bot should be registered at the Developer Portal in order to obtain the authentication credentials. Once registered, the values should be replaced in the `Web.config` and republished to allow connections from bot clients. If you have configured dummy values during development, remember to add the same values in the `web.config` while connecting from the emulator, as shown in Figure 2-13. `MicrosoftAppId` acts as a unique identifier for the bot application, and `MicrosoftAppPassword` is autogenerated by the Developer Portal (Figure 2-13).

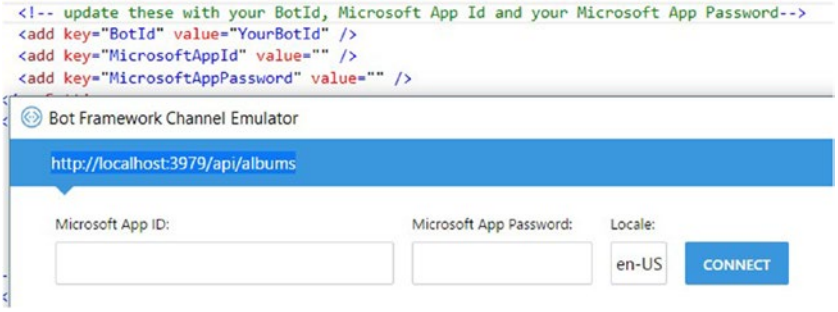

Figure 2-13. *Bot authentication using app ID and password*

Building a Bot

We earlier discussed the design methodologies for building a bot. In this section, we will implement the doctor's appointment bot using the form-flow feature provided by the MS Bot framework. You can create a new bot application and name it DockerAppointmentBot or use the EchoBot template we created earlier.

In a form flow–based bot application, the bot framework starts organizing dialogs in a stack. The stack starts with a root dialog, which will never exit. Replace the contents of `MessageController` with the following code to instantiate `RootDialog`.

```
[BotAuthentication]
public class MessagesController : ApiController
{
    internal static IDialog<Appointment> MakeRootDialog()
    {
        return Chain.From(() => FormDialog.FromForm(Appointment.BuildForm));
    }

    /// <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 != null)
        {
            // one of these will have an interface and process it
            switch (activity.GetActivityType())
            {
                case ActivityTypes.Message:
                    await Conversation.SendAsync(activity, MakeRootDialog);
                    break;

                case ActivityTypes.ConversationUpdate:
                case ActivityTypes.ContactRelationUpdate:
                case ActivityTypes.Typing:
                case ActivityTypes.DeleteUserData:
                default:
                    Trace.TraceError($"Unknown activity type ignored:
                    {activity.GetActivityType()}");
                    break;
            }
        }
        return new HttpResponseMessage(System.Net.HttpStatusCode.Accepted);
    }
}
```

The below code creates the root dialog and adds it to the conversation stack. The dialog gets created when the user initiates the first conversation. You can also choose to start the initial conversation by subscribing to a suitable activity type, like `conversationUpdate`.

```
internal static IDialog<Appointment> MakeRootDialog()
{
    return Chain.From(() => FormDialog.FromForm(Appointment.BuildForm));
}
```

The below code creates a new dialog from the Appointment class. The following are the contents of the Appointment class.

```
public enum Specialty { Dentist, GeneralPhysician, Psychiatist,
Cardiologist, PhysioTherapist }

    // Appointment is the simple form you will fill out to set up an
    appointment with Doctor.
    // It must be serializable so that the bot can be stateless. The order
    of fields defines the default order in which questions will be asked.
    // Enumerations shows the legal options for each field in the
    SandwichOrder, and the order is the order values will be presented
    // in a conversation.
    [Serializable]
    public class Appointment
    {
        [Prompt("When would you like to book your {&}?")]
        public DateTime AppointmentDate { get; set; }

        [Prompt("What is the {&}")]
        public string PatientName { get; set; }

[Prompt("What are the {&} you are looking for? {||}")]
        public Specialty? Specialties;

        [Prompt("Any {&} to the Doctor?")]
        public string SpecialInstructions { get; set; }

        public static IForm<Appointment> BuildForm()
        {
            OnCompletionAsyncDelegate<Appointment> processAppointment =
            async (context, state) =>
            {
                IMessageActivity reply = context.MakeMessage();
                reply.Text = $"We are confirming your appointment
                for {state.PatientName} at {state.AppointmentDate.
                ToShortTimeString()}, please be on time. " +
                             " Reference ID: " + Guid.NewGuid().ToString().
                             Substring(0, 5);

                // Save State to database here...

                await context.PostAsync(reply);
            };
```

```
            return new FormBuilder<Appointment>()
                    .Message("Welcome, I'm Dr.Bot ! I can help with fix an
                    appointment with Doctor.")
                    .OnCompletion(processAppointment)
                    .Build();
        }
    };
```

The `Appointment` class acts as a template for building the form. Each property of the class is converted into a form parameter that the user is expected to fill in, like with any web or mobile application. The prompt statement with every property is bound using the `prompt` attribute, as shown here:

```
        [Prompt("What are the {&} you are looking for? {||}")]
public Specialty? Specialties;
```

When presented to the user, {&} is replaced by the name of the property and {||} is replaced by possible options for the property. The values for the property `Specialties` are of enum type, like a drop-down of values. The user should select one before proceeding further. For primitive types like date, string, or number, the user can simply enter the text. Figure 2-14 shows the bot's conversation with a user.

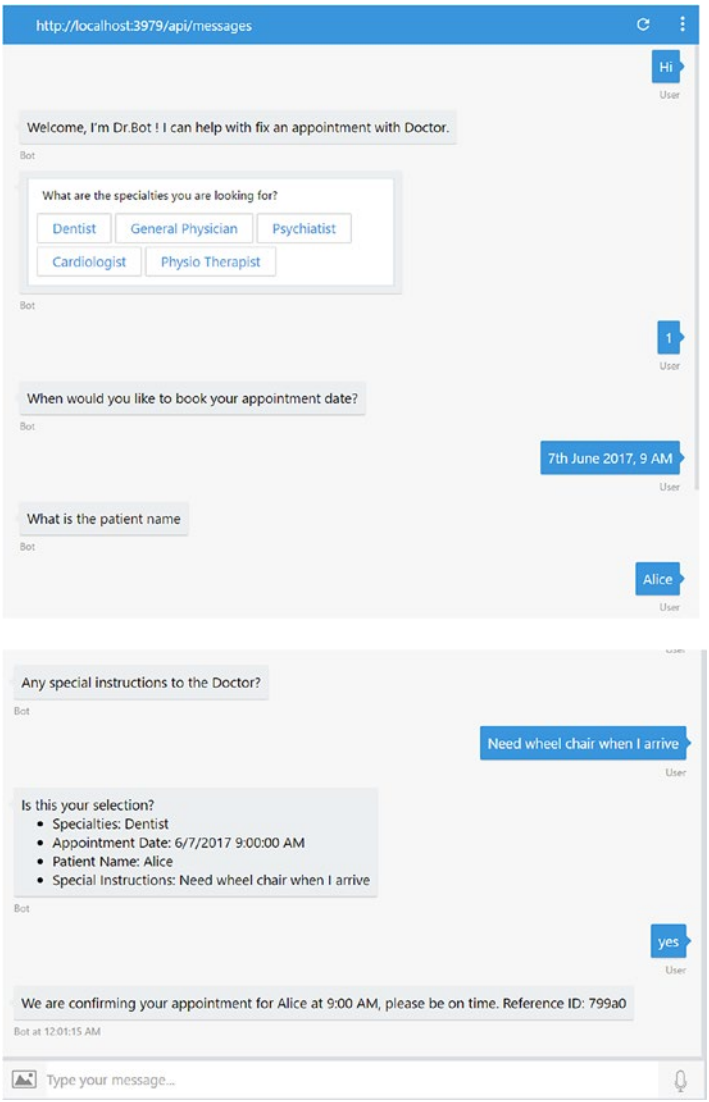

Figure 2-14. *Conversation with doctor appointment bot using Bot Emulator*

You will notice that we have followed all the best practices for the bot and achieved the required level of functionality with very little code. All the UI and navigation logic is automatically handled by the framework. We will learn more about the form flow in the next few chapters. Let us now learn how to deploy the bot to Azure so that our users can start interacting with it.

Deploy Bot to Azure

A bot application is a simple web service. To make your bot reachable by your users, it should be hosted somewhere. It could be any public cloud hosting platform, like Azure or AWS, or any machine on your network that is accessible to the internet. In this section, we will learn to publish our bot to Azure and test it using Bot Emulator, provided by the developer portal. Deploying applications on Azure requires an Azure subscription; for learning purposes Microsoft offers a free Azure subscription that can be used to develop and test bots. You can subscribe to this free subscription by visiting `https://azure.microsoft.com/en-in/free/`. This free subscription allows you to explore Azure by deploying anything for a period of 30 days, and you get $200 of free credit.

Azure App Service is a web application–hosting platform offered as part of the Azure subscription. App Service can be used to host any web, mobile, or web API application written in any language and targeting any device. App Service comes with instant scalability and built-in monitoring capabilities, and it's the most preferred platform for hosting web applications. Applications deployed on Azure App Service are managed by Azure. By running them in a managed environment unlike a virtual machine (VM), each application instance is isolated from the others and is instantly scalable.

Follow the below steps to deploy a bot application to Azure App Service (shown in Figure 2-15).

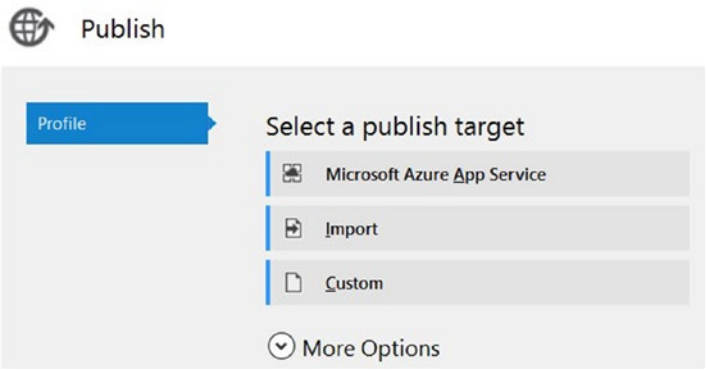

Figure 2-15. *Publishing bot application to Azure App Service*

1. Sign up for a free Azure account using your Live or Hotmail ID at the link provided earlier.
2. Right-click on the bot application and click on the Publish option.
3. Choose Microsoft App Service as the publish target.
4. You will be prompted to log in to your Azure account. Login with the credentials associated with your Azure account.

5. Select your subscription and then set **Resource Group** as View. Azure Resource Group deployment is a logical combination of one or more related resources that together form a product. Azure uses the term *Resource Group* to manage a set of resources together.

6. Click on **New** to create a new Resource Group.

7. The Create App Service window allows us to create an Azure App Service account, as shown in Figure 2-16.

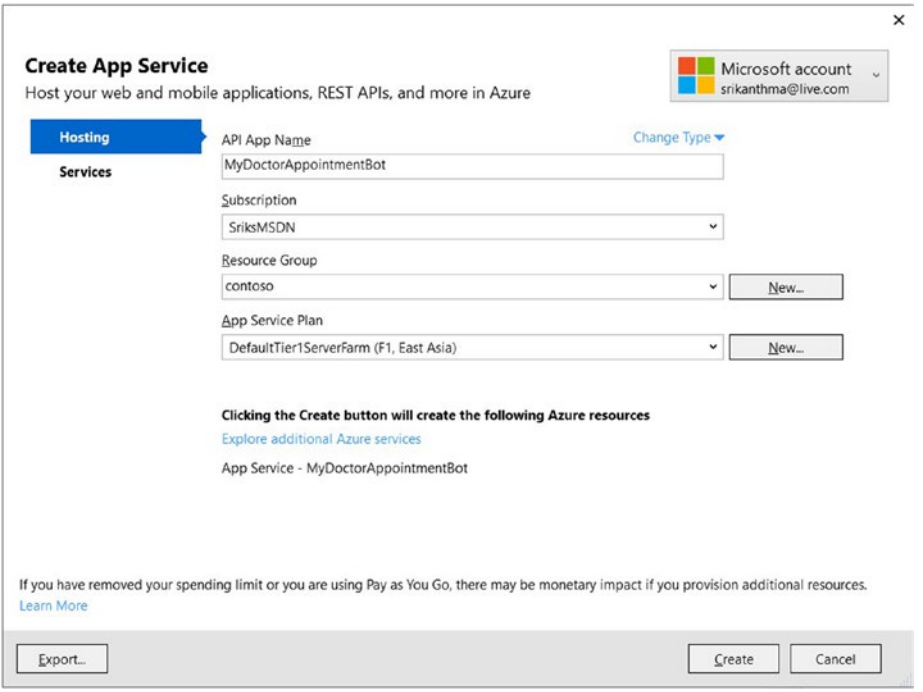

Figure 2-16. *Create an Azure App Service account*

8. Select an existing Resource Group or create a group by just typing in the name of the resource group.

9. Click **New** to create a new App Service plan. An App Service plan allows us to choose the size and cost of managed computing resources that will be used by Azure. We can also select the **Free** size during development; you will not be charged for computing resources and storage if the free size is selected, which makes it an ideal option while the application is still under development.

10. Press **OK** to confirm the App Service plan.

11. Click **Create** on the Create App Service window to start creating the required resources on Azure.

12. The Create App Service window shows the progress of deployment at the bottom left corner.

13. After successful creation of the required resources, Visual Studio opens the Publish window, as in Figure 2-17.

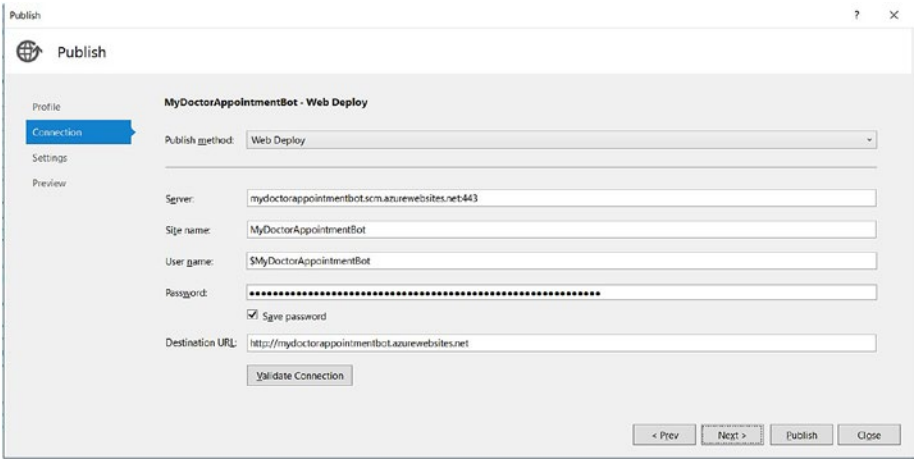

Figure 2-17. *Using Visual Studio Web Deploy to deploy bot application to Azure App Service*

14. The window contains properties already filled in that automatically connect to the resources we created in the previous steps. Click on **Validate Connection** to ensure the connection to the Azure App Service is successful. A green check mark confirms the connection is successful



. Click **Next**.

15. Select **Release** as configuration. We can also select **Debug** to remote-debug the deployed application. (Remote debugging will be covered in subsequent chapters). Click **Next** to continue.

16. Click on **Preview** to preview the files that will be published to Azure App Service, as shown in Figure 2-18.

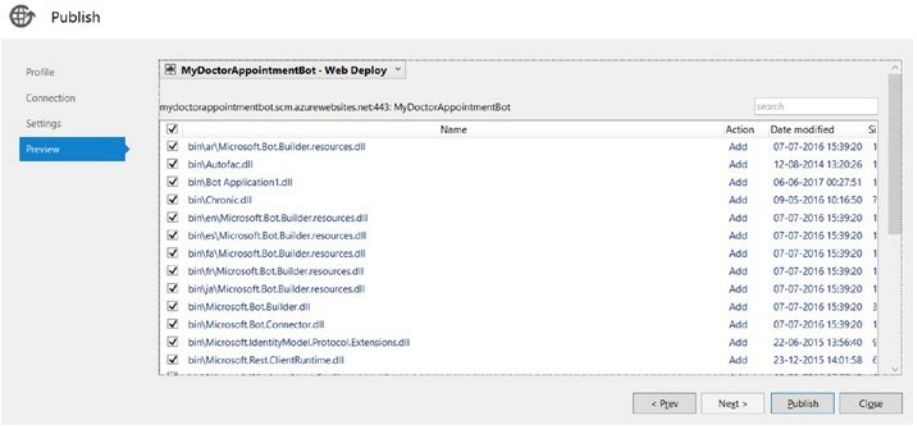

Figure 2-18. *Visual Studio Web Deploy or Publish preview*

17. Click on **Publish** to publish the binaries to Azure App Service.

18. Visual Studio shows the progress of Web Publish in the Web Publish Activity window. Click on **View ➤ Other Windows ➤ Web Publish Activity** to open the window if not visible automatically. Visual Studio also opens the published website `http://mydoctorappointmentbot.azurewebsites.net/` (the URL might vary based on the name you have chosen for the application) in your default browser.

19. Notice `azurewebsites.net` is the default suffix for every App Service. Azure also allows you to map a custom domain name to App Service; please visit `https://docs.microsoft.com/en-us/azure/app-service-web/web-sites-custom-domain-name` for more details.

We have successfully published the bot to Azure App Service. We can manage the bot application from Azure Portal (`https://portal.azure.com`). Log in to the portal and navigate to App Services. Click on the App Service name selected during deployment. In this case, the name is mydoctorappointmentbot. The portal shows the application's dashboard, with options to manage the application, like stop, delete, or restart the application. The portal also shows diagnostics and logging details by default (Figure 2-19).

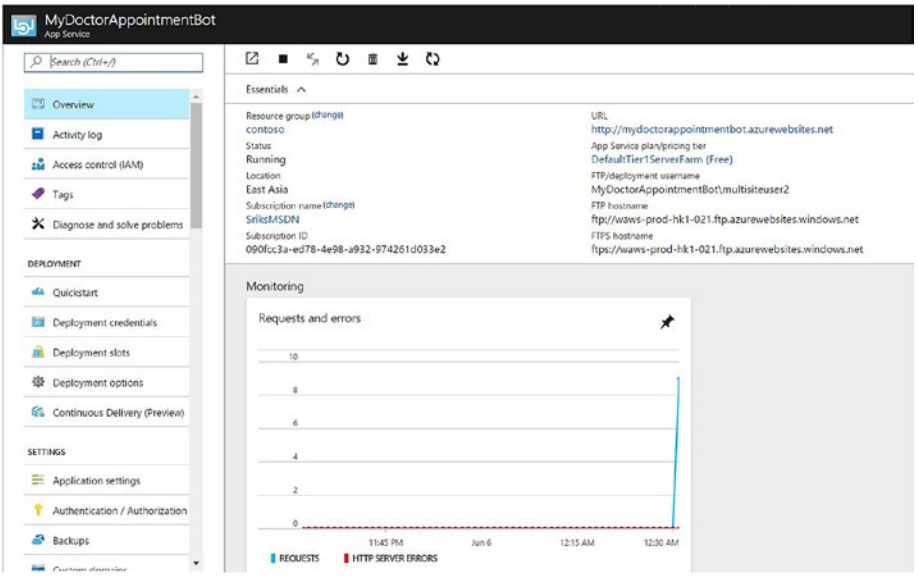

Figure 2-19. *Azure App Service dashboard for doctor appointment bot*

We cannot yet test the application using an emulator because the `MicrosoftAppId` and `MicrosoftAppPassword` attributes should be the actual ones in order to connect to the bot remotely. These two attributes can be created by registering our bot at the Developer Portal at `https://dev.botframework.com`.

Register the Bot

The Bot Developer Portal is for registering our bot and configuring Bot Connector Service channels so that users can communicate with our service using different platforms. Bot Connector Service works as an adapter between the channels we enable for users and the Azure-hosted bot application. Applications that the users can choose to connect to our bot can be configured using the Channels section on the Developer Portal. When the bot is registered with Microsoft Bot Connector Service, the portal provides the `MicrosoftAppId` and `MicrosoftAppPassword`, which are used to authenticate the conversation. The bot configuration (web.config for C# applications) also contains a property called `BotId`, which is used to store the URL in both the directory and the Developer Portal.

1. Log in to the Bot Developer Portal at `https://dev.botframework.com` using your Microsoft account.
2. Click on **Register Bot**.

3. Fill in the details about your bot, as shown in Figure 2-20. You can also upload an icon here. The details filled in here, including the icon, will show up in the bot directory. A bot directory is a contact list of all bots developed using the MS Bot framework. Remember: the bot handle should be a unique string across all bot registrations.

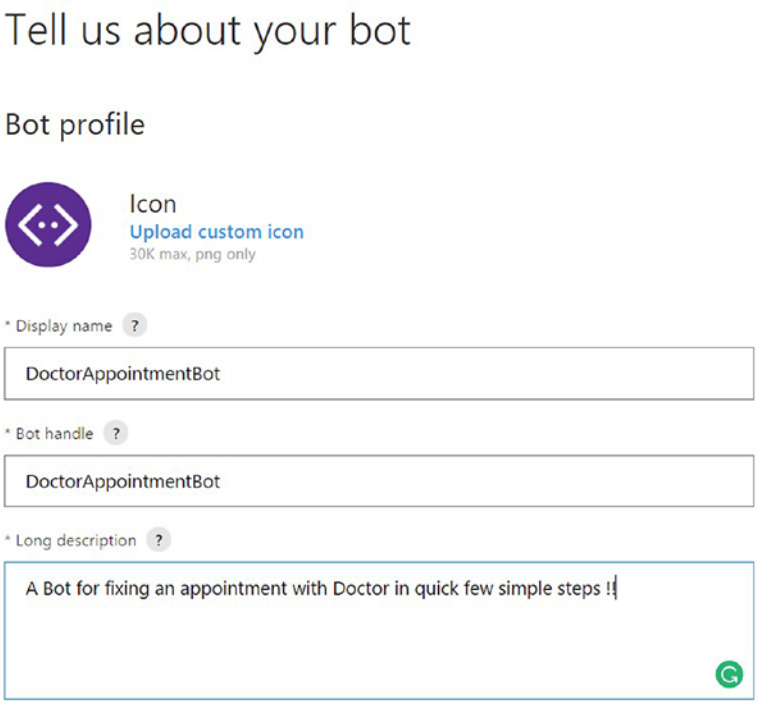

Figure 2-20. *Bot registration*

4. In the Configuration section, enter the messaging endpoint, as shown in Figure 2-21. The endpoint should be reachable from the internet, so we cannot use development URLs like `http://localhost:3979` while registering the bot. The bot's messaging endpoint should always be a valid HTTPS URL. Azure App Service offers SSL security for free, so we do not have to worry about configuring HTTPS for our bot.

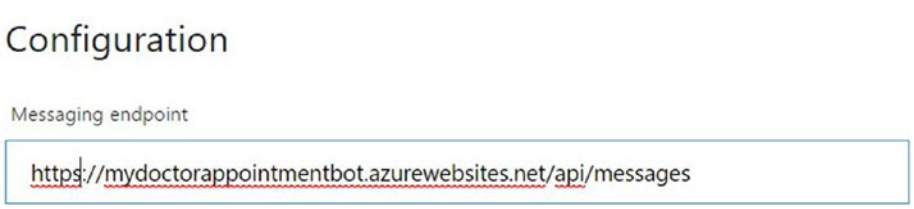

Figure 2-21. *Messaging endpoint configuration while registering a bot*

5. Click on **Create Microsoft App Id and Password**; you will be redirected to a different page (`https://apps.dev.microsoft.com`), which shows the app ID and app name, as shown in Figure 2-22. Click on **Generate app password to continue** to generate an app password.

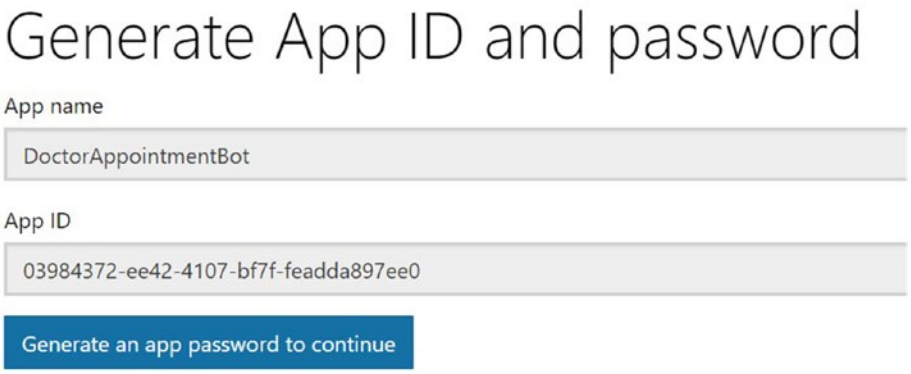

Figure 2-22. *Bot app ID and password*

6. Copy the password and paste it into your configuration file. This is the only time the app password is shown, so make sure you have a copy. If you lose it, you will be forced to regenerate the password, replace the old password in the configuration file, and republish the bot application.
7. Click on **Finish and go back to Bot Framework**.
8. Copy the app ID and paste it in the configuration file. The configuration file should be updated, as shown in Figure 2-23.

```
<appSettings>
  <!-- update these with your BotId, Microsoft App Id and your Microsoft App Password-->
  <add key="BotId" value="DoctorAppointmentBot" />
  <add key="MicrosoftAppId" value="03984372-ee42-4107-bf7f-feadda897ee0" />
  <add key="MicrosoftAppPassword" value="vxnGA15gmTK9n68goXe0b65" />
</appSettings>
```

Figure 2-23. *Updating Web.config with bot app ID and password*

9. Configure the Owners field with your Microsoft Account. We can ignore the Instrumentation key (we will learn to configure application insights later). Select the checkbox to confirm acceptance of the privacy statement, terms of use, and business conduct.

10. Finally, click **Register** to register the bot. The website greets you with a "Bot created" confirmation; click **OK** to proceed.

11. Before we configure channels for our bot, we should republish it with the updated Microsoft app ID and password. Switch back to Visual Studio, right-click on bot project, and click **Publish**; then, click **Publish** again to ensure the app published successfully.

Configure Channels

Now we have our bot service up and running on Azure and have registered our bot in the Developer Portal so that the connector service can relay messages to our service. It is now time to register the various channels by which the users will connect to our bot.

1. To configure a channel, Login again to the Developer Portal (`https://dev.botframework.com/`).

2. Navigate to the My Bots section.

3. Click on Bot we just created.

4. We can always test the connection to our bot using the Test section. Click on **Test** and try communicating with the bot, as shown in Figure 2-24.

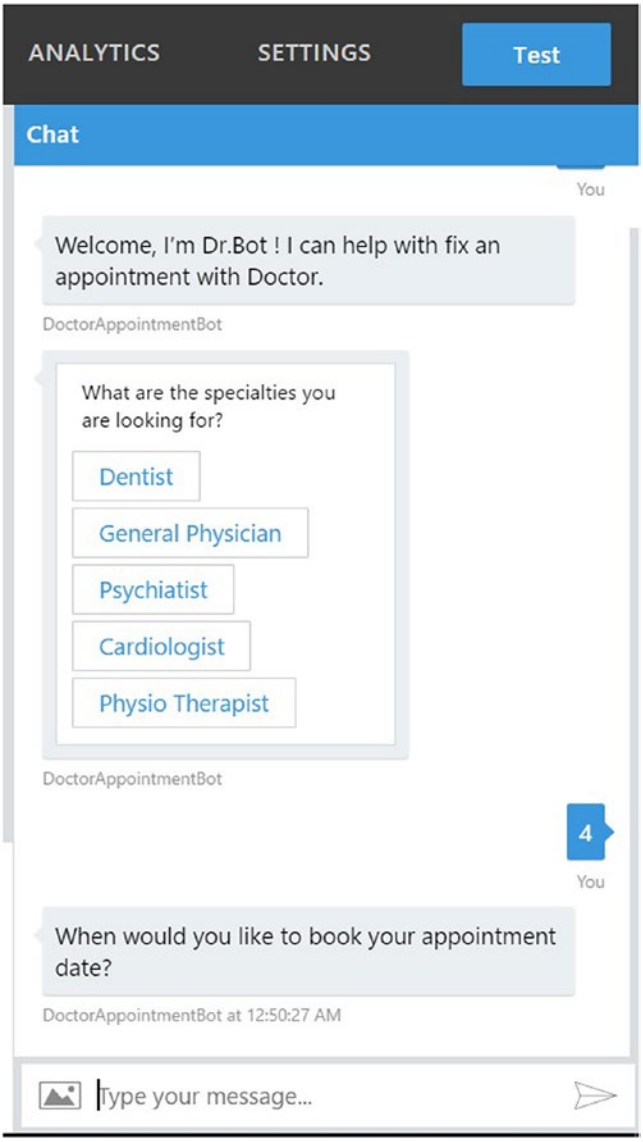

Figure 2-24. *Test the registered bot using built-in web-chat control in developer portal*

5. Click on the Channels section; notice that Skype and Web Chat come pre-configured and enabled by default. In preview mode, Skype Bot is limited to 100 subscribers. We should publish the bot to remove the limit.

Configuring Skype Bot

1. Figure 2-25 shows the list of channels or users applications available. All these applications are able to communicate with our bot. You can select any channel here to allow users to discover your bot via that channel.

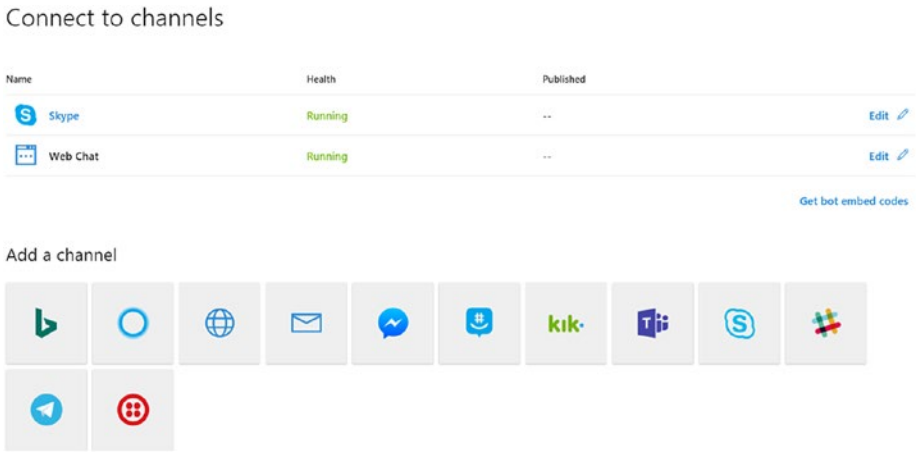

Figure 2-25. *Channel configuration in Bot Developer Portal*

2. Click on Skype, for example, to configure it as a channel for our bot. You will be redirected to a new page, where you will be presented with a consent form like the one shown in Figure 2-26. The consent page shows the bot's description and the different permissions and access levels our bot will have access to.

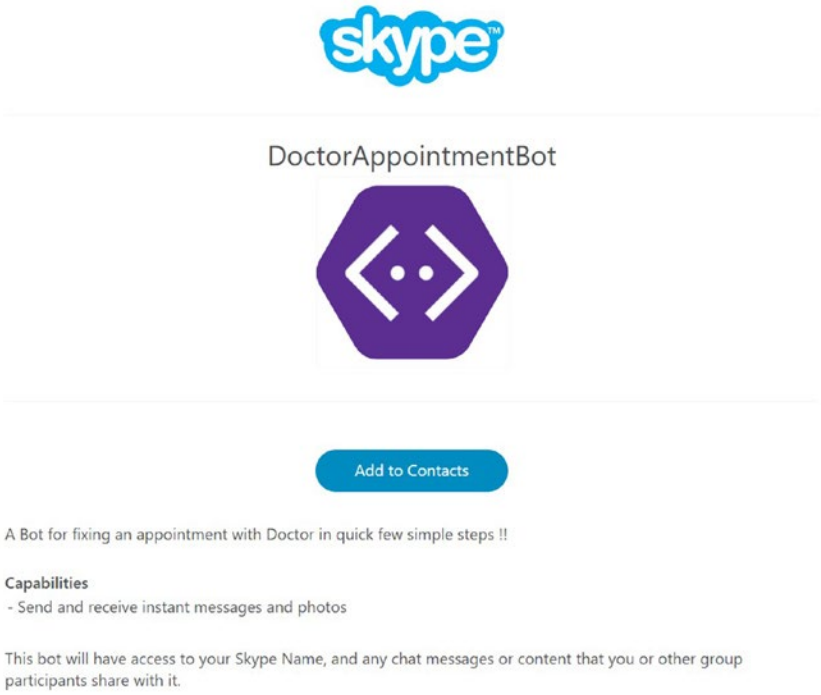

Figure 2-26. *Enable Skype channel for doctor appointment bot*

3. Click on **Add to Contacts** to add this to your Skype contacts. The site might further ask you to download Skype Preview.
4. Download and install Skype Preview if you do not have it installed on your machine yet.
5. Log in to Skype account to start using the bot we just developed and published. Figure 2-27 shows a conversation with our bot using Skype for Windows.

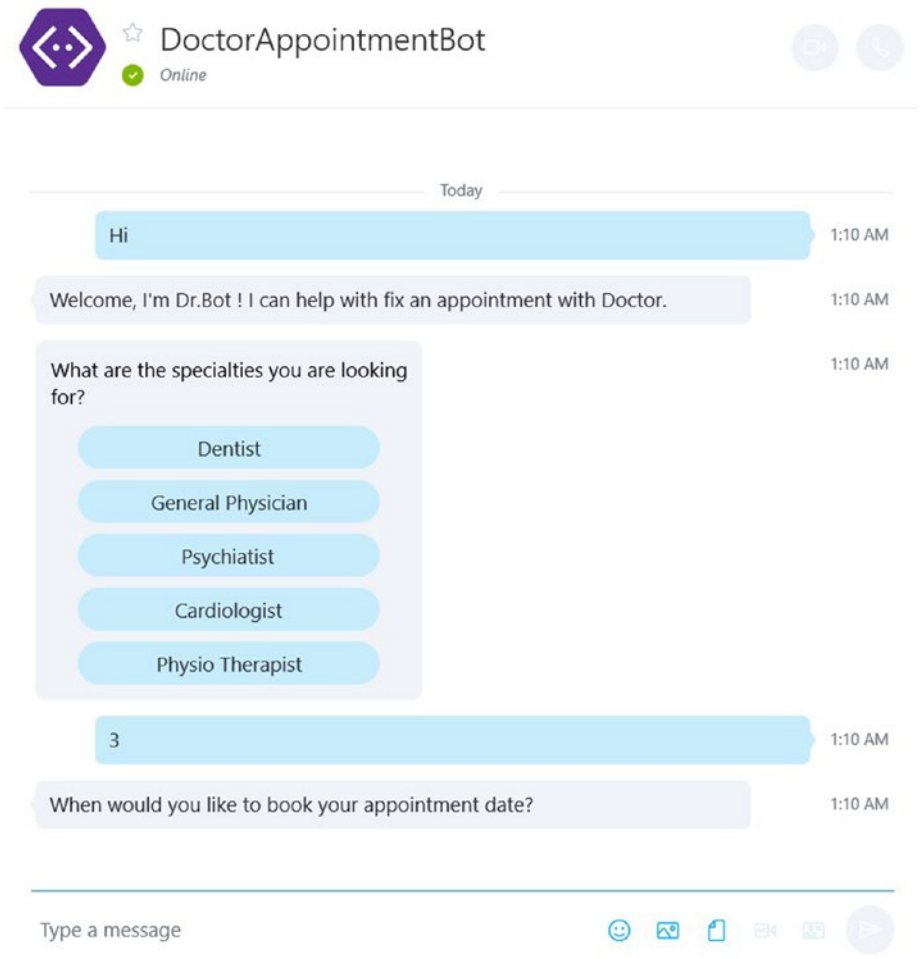

Figure 2-27. *Conversation with doctor appointment bot using Skype on Windows 10*

6. The Developer Portal allows you to reconfigure Skype settings whenever you like. Click on the **Edit** button next to the Skype channel configuration to go to a different page where you can configure what the user can do with your bot, like enabling text and pictures, enabling cards, enabling group messaging or calls, or even disabling the Skype-compatible bot entirely.

7. Click on **Bot embed codes** to see the embedded code that can be shared with others; in the pre-published state, the bot's embed code can be shared with 100 users only.



Figure 2-28. *Skype bot embed code*

Configuring Web Chat

A web chat is a chat widget you can embed in your website. Click on **Edit** next to the web chat to configure a web chat. Name the site; this option can be changed later. The page shows you a couple of hidden secret keys and an HTML iframe tag. Now, there are two ways you can embed web-chat controls in your web page. The easiest way is to paste the iframe tag into your website with the secret. This might just work, but it is not secure, because anyone can just reuse your web-chat bot by copying the iframe tag from your HTML page when it renders. The second way, which stops others from stealing your web bot, is by making an HTTP GET request to `https://webchat.botframework.com/api/tokens` by passing the secret in the HTTP header, this call returns an authorization token, which can only be used for one conversation. You can further style your embedded web chat by applying the style on the iframe tag directly.

You can add any other channel from the following list by just clicking on **Add**. Each of the channels has its own custom channel-configuration page, which can be used to configure settings specific for that channel. Figure 2-29 is the configuration page for Microsoft Teams.

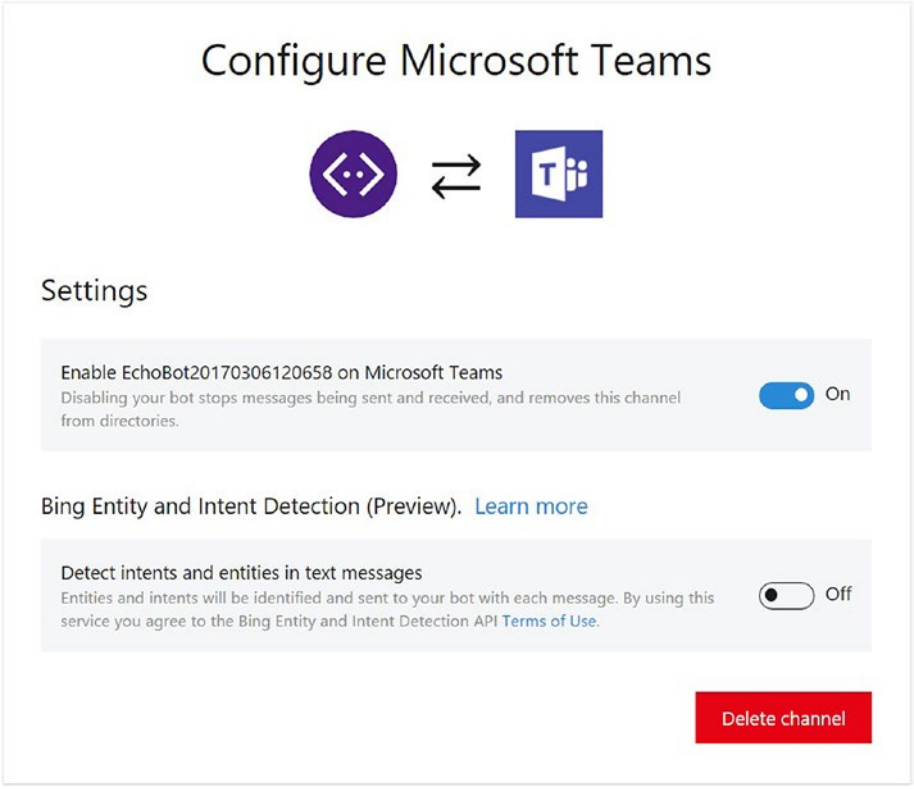

Figure 2-29. *Configuring Microsoft Teams channel*

Click on **Add to Teams** once you finish configuring the channel. The website opens Microsoft Teams for desktop, if available, or prompts you to download it, as we saw for Skype. As you can see in Figure 2-30, once the bot is published and tested, it is quite easy to configure multiple channels and increase its reachability.

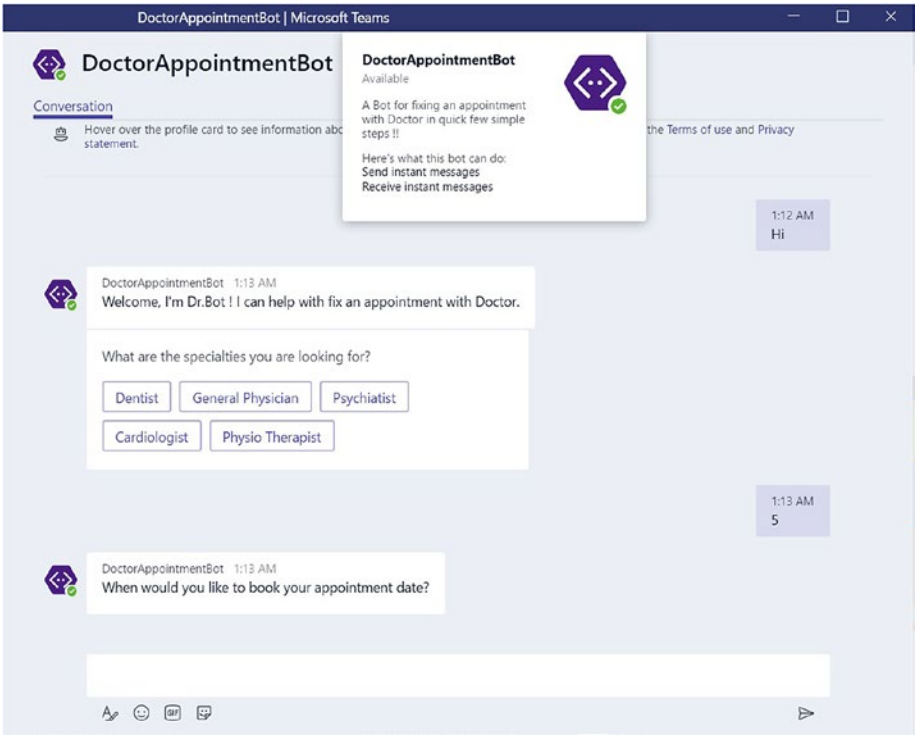

Figure 2-30. *Conversation with doctor appointment bot using Microsoft Teams*

If you are not finding any of your favorite channels, you need not worry; Microsoft offers a channel called Direct Line API, which can be used to write custom client applications. The Direct Line API is a REST API that allows developers to write their own client applications, web-chat controls, mobile apps, or service-service applications that will talk to the bot.

Summary

A bot application is a simple web service. There are three main parts of the bot framework: the Bot Builder SDK, Bot Connector Service, and Bot Developer Portal. The Bot Builder SDK is an open source free framework available in both C# and Node.js. The project template for building bots using C# is available as a separate download. Bots can be tested using Bot Emulator during development; it is a cross-platform tool. Bots can be deployed to any web-hosting platforms, like Azure or AWS. Bots should be registered on the Bot Developer Portal for authentication and to configure various channels for communication, like Skype, Teams, Slack, or Facebook. Microsoft Cognitive Services provides necessary intelligence to bots so they can understand and respond in more human-like ways using vision-, search-, knowledge-, speech-, language-, and location-processing capabilities.

CHAPTER 3

Develop Bots Using Node.js

Microsoft offers the Bot Builder SDK not just for the .Net framework but also for Node.js developers. Bot Builder for Node.js is an equally powerful framework for building interactive bots. The SDK can be used to build simple interactive and sophisticated bots with dialog prompts and artificial intelligence capabilities, like LUIS or Azure Cognitive Services. This chapter will focus on explaining the core concepts of Bot Builder for Node.js and help you get started with building, debugging, and deploying bots using VS Code on a Windows environment. The following points summarize the topics for this chapter:

- Setting up the development environment
- Build **Hello World** using VS Code
- Debugging using VS Code
- Building advanced bots with:
    - Dialogs
    - Prompts
        - Choice
        - Text
        - Confirmation
        - Number
        - Attachments
- Deploying to Azure from VS Code
- Setting up monitoring

© Srikanth Machiraju and Ritesh Modi 2018
S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_3

Setting Up a Development Environment

The following items should be set up in order to start developing bots:

- Node Package Manager Console
- VS Code or any Node.js code editor
- Bot Builder SDK
- Bot Emulator

First things first: to install the node modules required for building a bot we would need Node Package Manager, or NPM. NPM helps you download and install dependencies for building any application. To install the latest version, go to https://nodejs.org/en/. For authoring Node.js code, you can use any editor, like Cloud 9, Eclipse, Web Matrix, or any other tool. In this chapter, we will be using VS Code, a free, powerful, and lightweight editor from Microsoft that runs on Windows, MacOS, and Linux. VS Code comes with built-in support for authoring code in multiple languages, like JavaScript, TypeScript, Node.js, C#, and so on. It also comes pre-built with Intellisense support, debugging, and source control (Git) capabilities. You can download and install VS Code from https://code.visualstudio.com/. To aid developers in the authoring process, VS contains many extensions, which can be installed from VS Code.

Build Hello World Bot Using VS Code

In this section, we will build a simple bot that greets the user with "Hello World." Open NPM's command line in Administrator mode. Run the commands shown in Figure 3-1 to create a folder to store code artifacts.

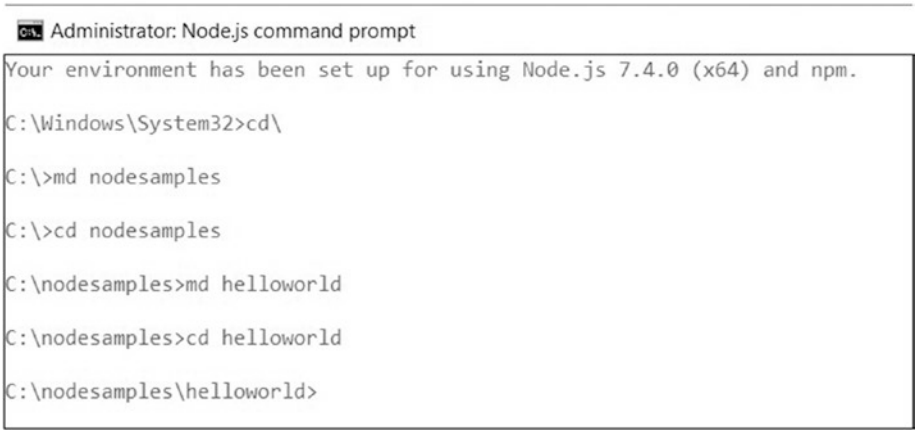

Figure 3-1. *Node Package Manager command prompt*

Run the following command to initialize the folder with a basic Node.js artifact (Figure 3-2):

```
npm init
```



Figure 3-2. *Initialize folder using NPM init*

`npm init` initializes the folder with the `package.json` file, which contains the application metadata. The next step is to install the dependencies required to build the sample bot. Run the following commands to install the Bot Builder and Restify modules using NPM.

```
npm install - -save botbuilder
npm install - -save restify
```

`BotBuilder` is the name of the Node.js module for building MS bots. `Restify` is a node.js module that helps you build or interact with RESTful web services. The preceding commands download and install the required dependencies in the current folder. Run the following command in the command prompt to open the current folder in VS Code:

```
code .
```

Notice in Figure 3-3 that there is a folder called `node_modules` that contains several open source modules. The subfolder named `botbuilder` contains the necessary code to help you build dialogs, cards, and buttons or add storage capabilities to bots. There is also a file named `package.json`, which contains the application's metadata and also dependencies required for the application. The contents of this file are added as part of the `npm init` command.

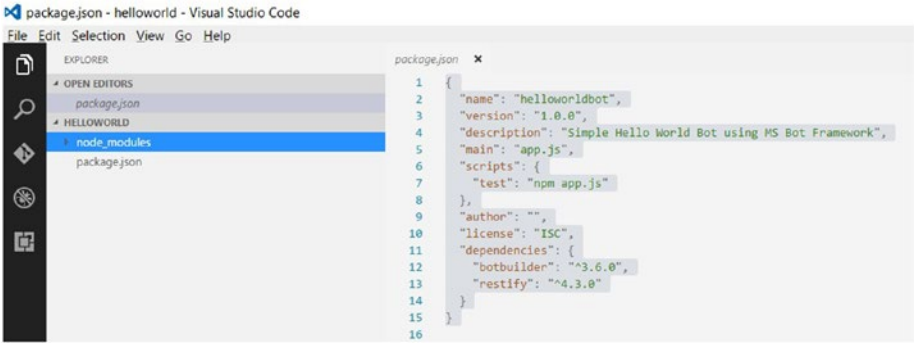

Figure 3-3. *Visual Studio Code*

Click on **Add New File** on the left side of the HelloWorld bar and name the file `app.js`. Add the following lines of code to the `app.js` file to include the modules:

```
var restify = require('restify');
var builder = require('botbuilder');
```

Bot Builder allows you to build bots for a variety of platforms. For example, you can build a bot that can be called from a command line or by using Bot Emulator. We should create a connector that will be used to interact with the bot. In the following code, we are creating a console connector:

```
var connector = new builder.ConsoleConnector().listen();
```

The logic to manage the bots lies within the `UniversalBot` class, which accepts a variety of connectors. In the following code, we pass the console connector just created to the `UniversalBot`:

```
var bot = new builder.UniversalBot(connector);
```

Now that we have the bot and the connector set up, we should be able to wire up the bot conversation. Bot conversations can be done in multiple styles, one of which is dialogs. In a typical web application, the user navigates through the application using the routing pattern defined by the application. Dialogs follow the same pattern; each dialog can be thought of as a route within the conversational approach. When the user sends a message, the framework takes care of routing the message to the active dialog. For example, the following code creates a dialog at the root of the conversation:

```
bot.dialog('/', function (session) {
    session.send('Hello World');
});
```

Run the command in Figure 3-4 from the command promt to run the bot. The bot waits for a gesture from the user and greets with "Hello World" (no matter what the user types), as that is the only active dialog.

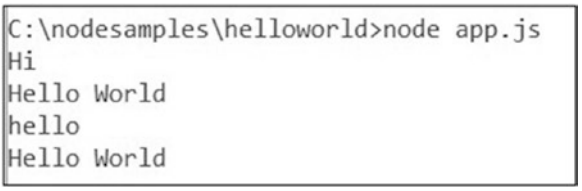

Figure 3-4. *Hello World bot application using Node.js*

Now, let us plug in a different connector to this bot application. Replace the contents of `app.js` with this code:

```
var restify = require('restify');
var builder = require('botbuilder');

//=========================================================
// Bot Setup
//=========================================================
// Setup Restify Server
var server = restify.createServer();
server.listen(process.env.port || process.env.PORT || 3978, function () {
   console.log('%s listening to %s', server.name, server.url);
});
```

```
// Create chat bot
var connector = new builder.ChatConnector({
    appId: process.env.MICROSOFT_APP_ID,
    appPassword: process.env.MICROSOFT_APP_PASSWORD
});
var bot = new builder.UniversalBot(connector);
server.post('/api/messages', connector.listen());
//=========================================================
// Bots Dialogs
//=========================================================
bot.dialog('/', function (session) {
    session.send("Hello World");
});
```

Notice that we have registered Restify to listen on port 3978 and the console connector is replaced with the `ChatConnector` class. Unlike the console client, the server will be listening on 3978 for post messages arriving at `api/messages`. To start the application, run the following command. The application starts listening for messages on port 3978 as shown in Figure 3-5.

```
C:\nodesamples\helloworld>node app.js
restify listening to http://[::]:3978
```

Figure 3-5. *Start bot application built using Node.js*

To connect to the chat connector bot, we would need an emulator; you can download and install the Bot Emulator for different types of operating systems from `https://docs.botframework.com/en-us/tools/bot-framework-emulator/`.

Open Bot Emulator and connect to `https://localhost:3978/api/messages` as shown in Figure 3-6. Click on **Connect** to confirm the connection. The Log window (Figure 3-6) shows the status of the connection to the application.

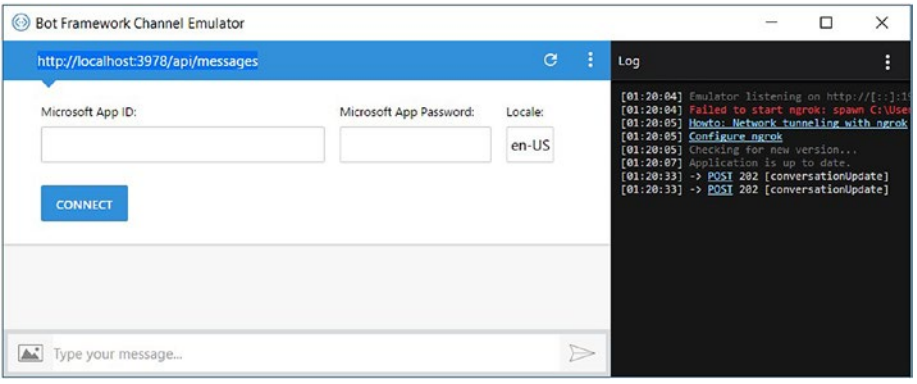

Figure 3-6. *Using Bot Emulator to connect to bot built using Node.js*

We can type any message in the text window at the bottom, and the application responds based on the routing logic, as shown in Figure 3-7. Bot Emulator also shows the details of the request response in the Log window. Click on the links for each request/response to see the JSON body.

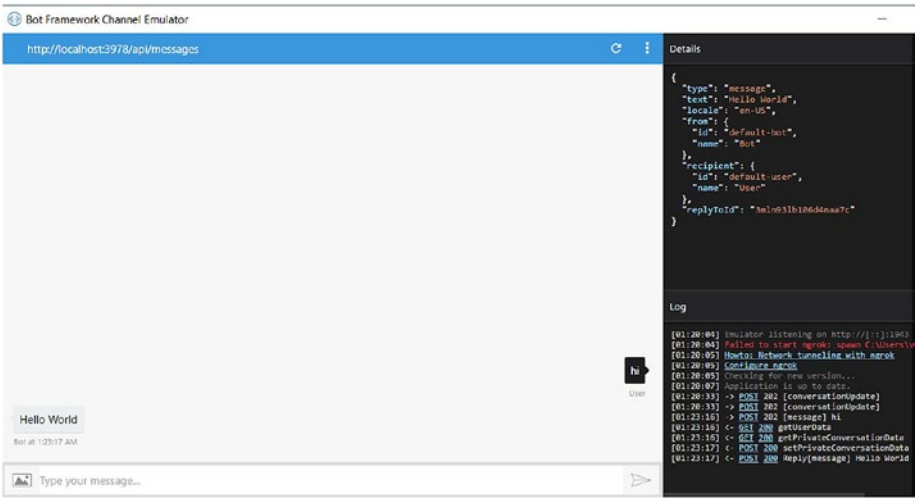

Figure 3-7. *Bot communications logs captured by Bot Emulator*

Additionaly, the node command window also shows logs of the coversations, as in Figure 3-8.

```
C:\nodesamples\helloworld>node app.js
restify listening to http://[::]:3978
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
session.beginDialog(/)
/ - waterfall() step 1 of 1
/ - session.send()
/ - session.sendBatch() sending 1 messages
```

Figure 3-8. *Bot communication logs captured by Node.js*

Debugging Using VS Code

VS Code provides smart Intellisense and debugging capabilities. To start debugging Node.js code within VS Code we need a launch configuration. Click on the Debug icon in VS Code and then select **Add Configuration** to select a launch configuration for our application, as in Figure 3-9.

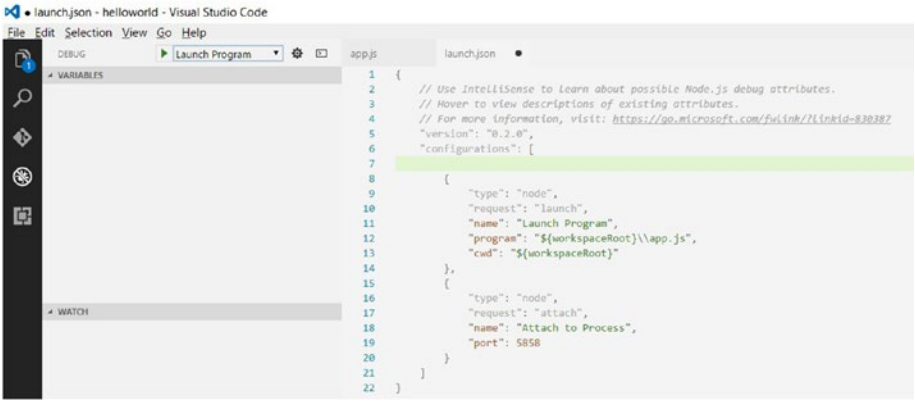

Figure 3-9. *VS Code launch configuration*

Press **F5** or click on the **Run** button to start the application in Debug mode. Alternatively, you can also run the application in Debug mode by running the following command:

```
Node –debug-brk app.js
```

You should be able to set up breakpoints and inspect objects within VS Code (Figure 3-10). You can see a `vscode` folder has been added to your project; it contains the launch configuration.

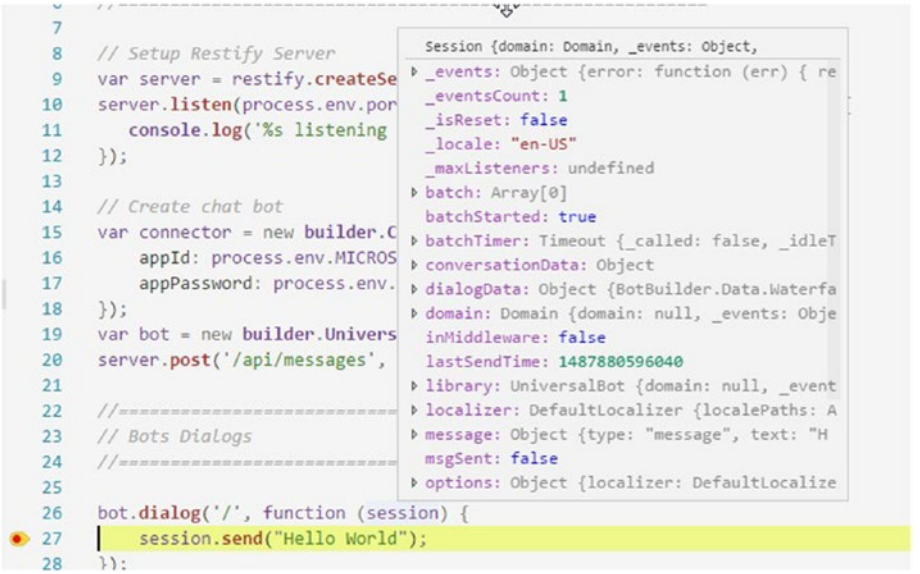

Figure 3-10. *Debugging bot applications using VS Code*

Building Bots with Conversations

Understanding how to create conversations is a critical piece of the Bot Builder framework. Bots use conversations to engage with the user, asking a sequence of questions or a continuous conversation until a logical conclusion is reached. In this section, we will learn to use dialogs and prompts to manage bot conversations with a user. As part of this section, we will also build a Pizza Bot that lets the user build his favorite pizza using dialogs and prompts.

Dialogs

Dialogs are like routes. For example, if we have to reach `index.html` on any website, like `http://fabrikamstore.com`, we invoke it in a browser, like `http://fabrikamstore.com/index` or just `http://fabrikamstore.com/`, which by default routes to the default page. Similarly, bots have a default route, like `'/'`. When the framework receives a message, it will be routed to the root for processing.

In the following code, the application routes the message to a route named `routeMenu`:

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send("Hello... I'm a pizza bot.");
        session.beginDialog('rootMenu');
    },
```

```
    function (session, results) {
        session.endConversation("Goodbye until next time...");
    }
]);
```

Here, we are passing a series of functions to the `UniversalBot` connector. Each function receives a `Session` object, which can be used to inspect the user details, send a reply to the user, or save data related to the user. `Session` objects can also be used to redirect the user to a different dialog. Within a dialog, we can create subconversations by using `session.beginDialog()`, which will search for other dialogs in the application using the string input (`rootMenu` in this case). The remaining functions in this series are called when each of the previous dialog calls with `session.endDialog()`. In this case, the result of the first function will be passed to the second function in the series. This style of conversation is called the Waterfall model. The Waterfall model is the most common form of dialog we will be using while building bots. But what if you want to collect inputs from the user to continue your conversation? Bot Builder framework contains a property called Prompts that helps you collect information from the user in a variety of forms.

Prompts

Bot Builder comes with a variety of built-in forms that can be used to collect input from the user. These prompts are implemented as dialogs, so responses to the prompts will be returned using the `session.endDialog()` or `session.endDialogWithresult()` methods. The framework maintains a stack of dialogs (or prompts, since prompts are also a type of dialog) that help the framework route the reply. Each dialog or prompt in the series receives the results or user responses from previous conversations. The following code shows a sample prompt that prompts the user to select one option from the menu to start a subconversation using `session.beginDialog()`:

```
// Add root menu dialog
bot.dialog('rootMenu', [
    function (session) {
        builder.Prompts.choice(session, "Choose an option:", 'Select
        Base|Select Toppings|Select Sides|Order Summary');
    },
    function (session, results) {
        switch (results.response.index) {
            case 0:
                session.beginDialog('basedialog');
                break;
            case 1:
                session.beginDialog('toppingsdialog');
                break;
            case 2:
                session.beginDialog('sidesdialog');
                break;
            case 3:
```

```
                session.beginDialog('ordersummary');
                break;
            default:
                session.endDialog();
                break;
        }
    },
    function (session) {
        // Reload menu
        session.replaceDialog('rootMenu');
    }
]).reloadAction('showMenu', null, { matches: /^(menu|back)/i });
server.post('/api/messages', connector.listen());
```

In the below code, we are using prompts. These show the user a list of menu options and seek an input from the user. Figure 3-11 shows how the menu is displayed to the user.

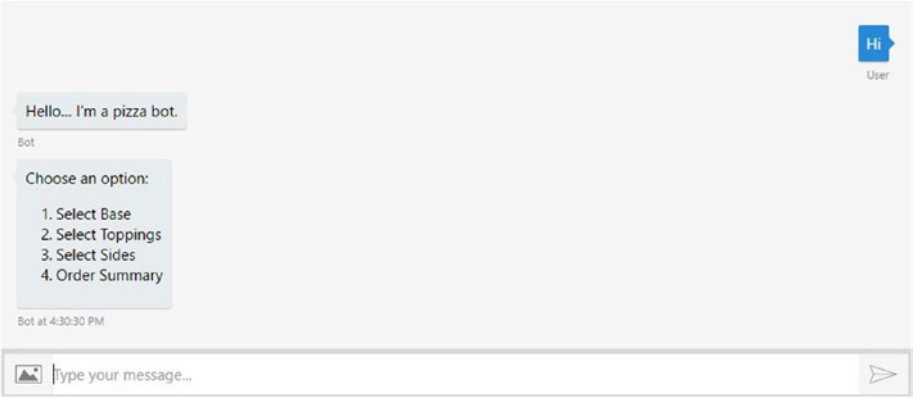

Figure 3-11. *Pizza Bot built using Node.js*

The user's response can be any one of the preceding options. The response is passed to the second function in the series. The option selected by the user will be available at `results.response.index`. Based on the option selected by the user, we can start a new conversation. If the user selects an option that is not present in the list, the framework is intelligent enough to respond with a default message, as shown in Figure 3-12.

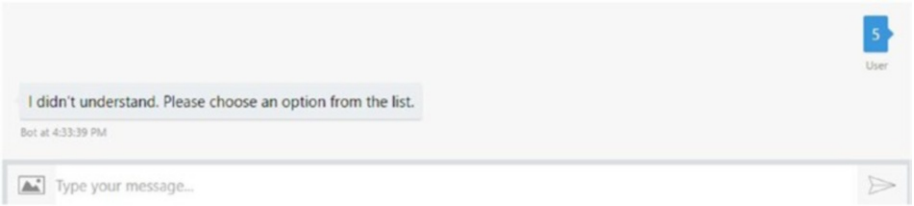

Figure 3-12. *Bot response for invalid option selected by user*

Prompts come in a variety of styles and can be presented to the user in different ways. The user interface (UI) of a prompt also depends on the channel being used.

Input Choice

`Prompt.choice()` asks the user to pick an option from a list. The user response will be returned as an `IPromptChoiceResult`. The list of choices can be presented to the user in a variety of styles using the `IPromptOptions.listStyle` property. The user can express their choice by entering either the number or its name. Bot Builder is smart enough to capture the menu option even if the user's response partially matches a menu's option. The following code shows a sample prompt that presents the pizza base options to the user using a button style:

```
// select base
bot.dialog('basedialog', [
    function (session, args) {
        builder.Prompts.choice(session, "Choose Thin Crust, Cheese Burst or
        Classic Hand Tossed", "thincrust|cheeseburst|classichandtossed", {
        listStyle: builder.ListStyle.button })
    },
    function (session, results) {
            session.endDialog("It's %s.", results.response.entity);
    }
]);
```

The options are presented to the user using buttons as defined in the `choice` method. If you are using a touch-enabled device to connect with the bot, you will notice that these options are clickable. The options in a menu can be separated by pipe symbol (|) as shown or declared as a string of arrays, such as `thincrust`, `cheeseburst`, and `classichandtossed` (Figure 3-13). Users can also choose by the number of the name of the option.



Figure 3-13. *Clickable menu created using choice style*

Text Input

The user's input is not always a choice; sometimes you might want the user to enter free-form text like their name, phone number, or email address. `Prompt.text` can be used to prompt the user to enter any text as input. The following code shows a code sample where the user can select any toppings for the selected pizza base:

```
// select toppings
bot.dialog('toppingsdialog', [
    function (session, args) {
        builder.Prompts.text(session, "Choose your toppings from Olives,
        Jalapeno, Onion, Bell Pepper, Corn");
    },
    function (session, results) {
            session.endDialog("It's %s.", results.response);
    }
]);
```

Confirm

The `prompts.confirm()` method can be used to confirm an action, like yes/no, from the user. The user's response will be returned as `IPromptConfirmResult`. The following code shows how to create a confirmation prompt:

```
// confirm order
bot.dialog('confirmorder', [
    function (session, args) {
        builder.Prompts.confirm(session, "Can I confirm your order?");
    },
```

```
    function (session, results) {
            session.endDialog("It's %s.", results.response);
    }
]);
```

The prompt looks like that shown in Figure 3-14.



Figure 3-14. *Confirmation dialog created using prompt style*

Users can respond using yes, y, or 1 for true and no, n, or 0 for false.

Number

The number prompt asks the user to enter a number. The following is sample code that does so:

```
builder.Prompts.number(session, "How many would you like to order?");
```

Time

The time prompt will ask the user to enter a time. The response can be relative, like "in 2 minutes," or absolute, like "June 5th 12 am." The MS Bot framework uses a library called Chronos to parse the user's response. The following code shows a sample prompt for asking for a time:

```
builder.Prompts.time(session, "What time would you want the order?");
```

The result returned by the entity is a prompt that should be parsed into a JavaScript object using `EntityRecognizer.resolveTime()` as shown here:

```
builder.EntityRecognizer.resolveTime([results.response]);
```

Attachment

The attachment prompt asks the user to upload a file attachment, like an image or video. The response will be of type `IPromptAttachmentResult`.

```
builder.Prompts.attachment(session, "Upload a picture for me to
transform.");
```

Messages

Bot Builder contains a `Message` builder class that can be used to send messages to the user using the `Session` object. A `Session` object is passed to every dialog whenever a bot receives a message and contains all the information required to send any number of messages to the user.

`Session.send()` can be used to send simple text messages, attachments, or cards to the user. The bot can call `Session.send()` any number of times before the user responds. If the bot sends multiple replies, the replies are grouped into a batch and sent to the user at once, thus preserving the order of the messages. The message can also contain template parameters like the one shown here:

```
session.send("hello there %s", name)
```

Bots can send simple text or much more sophisticated messages, like images, videos, or any other file attachments, to the user. The following sample shows how to send an attachment to the user in the response:

```
bot.dialog('/picture', [
    function (session) {
        session.send("You can easily send pictures to a user...");
        var msg = new builder.Message(session)
            .attachments([{
                contentType: "image/jpeg",
                contentUrl: " http://images.clipartpanda.com/pizza-clipart-
                pizza-clipart-1.jpg"
            }]);
        session.endDialog(msg);
    }
]);
```

Sometimes, in order to provide your user with a rich experience, the response sent to them should look attractive, actionable, and informative. Bot Builder allows you to design rich responses using cards, which are a combination of image, video, button, and text. The Bot Builder SDK contains helper methods to render cards to the user in a cross-platform way. A few helper card types provided by the SDK are Hero Card, Thumbnail Card, Receipt Card, and Sign-in Card. Remember that since these cards are UI rich, not all channels will support them in the same way. To determine the channel and customize the message schema based on the channel, we can use the `Message.sourceEvent()` method. The following code shows a Hero Card with a pizza-order summary:

```
// order summary - bot state
bot.dialog('ordersummary', [
    function (session, args) {
        var msg = new builder.Message()
                .address(session.message.address)
```

```
                .attachments([
                        new builder.HeroCard(session)
                        .title("Pizza Bot")
                        .subtitle("Order Summary")
                        .text("Pizza Base: " + session.userData.base + "
                        Pizza Toppings: " + session.userData.toppings + "
                        Sides: " + session.userData.sides)
                        .images([
                           builder.CardImage.create(session,
                           "http://images.clipartpanda.com/pizza-clipart-
                           pizza-clipart-1.jpg")
                        ])
                        .tap(builder.CardAction.openUrl(session,
                        "https://docs.botframework.com/en-us/node/builder/
                        overview/#navtitle"))]);

       // Send message through chat bot
       bot.send(msg);
    },
    function (session, results) {
       session.endDialog();
    }
]);
```

While the bot is busy composing a response, the user is unaware of the background processing, so it is our responsibility to show a progress indicator like we do in other types of applications. This is particularly important when you are starting an asynchronous operation to fetch data from a far resource that might take few seconds. `session.sendTyping()` shows an indicator to the user about the background processing. The amount of time the indicator stays visible depends on the channel; for example, it is three seconds for Slack and twenty seconds for Facebook. If you want to show an indicator periodically, a custom logic should be written.

State

While the bot is interacting with the user, there should be some means to store the conversation data or user-profile information to be reused later. Bot Builder SDK contains a built-in storage system. There are several ways of persisting data relative to the user or conversation. The following are properties of the `Session` object that allow you to store data at different levels.

- **User Data**: `session.userData` can be used to store information globally for the user across all conversations.
- **Conversations Data**: `session.conversationsData` stores information globally for a single conversation across the users. This data is visible to anyone who is part of the conversation, so we should be extra cautious with what we store here. This option is disabled by default and should be enabled using the bot's `persistConversationsData` setting.
- **Private Conversations Data**: `sessions.privateConversationsData` saves the user's conversation data. Since this data stores information across different dialogs in the stack, it is a nice option for saving anything related to a conversation and is organized per conversation per user.
- **Dialog Data**: `session.dialogData` persists data within a single dialog instance. This is essential for storing information in-between the steps of a Waterfall model.

The following code example shows the storing of a user response using the `session.userData` object so that we can reuse it when building the order summary:

```
// select toppings
bot.dialog('toppingsdialog', [
    function (session, args) {
        builder.Prompts.text(session, "Choose your toppings from Olives,
        Jalapeno, Onion, Bell Pepper, Corn");
    },
    function (session, results) {
            session.userData.toppings = results.response;
            session.endDialog("It's %s.", results.response);
    }
]);
```

Deploying to Azure

The completed sample for Pizza Bot is available under the code samples. In this section, we will learn how to deploy the Pizza Bot to Azure and test it using Bot Emulator. To complete the samples in this section, you will need an Azure Subscription; you might get a temporary one for free from https://azure.microsoft.com/en-in/free/.

A bot gets deployed as an Azure website. There are many ways to create an Azure website, such as from the portal (https://portal.azure.com). Use PowerShell if you are on a Windows machine. In this section, we will create an Azure website using Azure CLI, which is a cross-platform tool to manage Azure resources. The following steps show you how to deploy a bot on Azure using the command line.

Open the Node Package Manager console as Administrator and run the following command to install Azure CLI:

```
npm install azure-cli -g
```

To create a website, we must log in to Azure, so run the following command to log in using your subscription's credentials:

```
azure login
```

Run the following command to switch to Azure ARM mode:

```
azure config mode asm
```

Type the following command to create a new site and configure it for Node.js and Git:

```
azure site create - -git simplepizzabot
```

The below command creates an Azure website. Login to the Azure portal and set up the Git credentials so you can set up the continuous publish feature. Figure 3-15 shows how to set up deployment credentials.

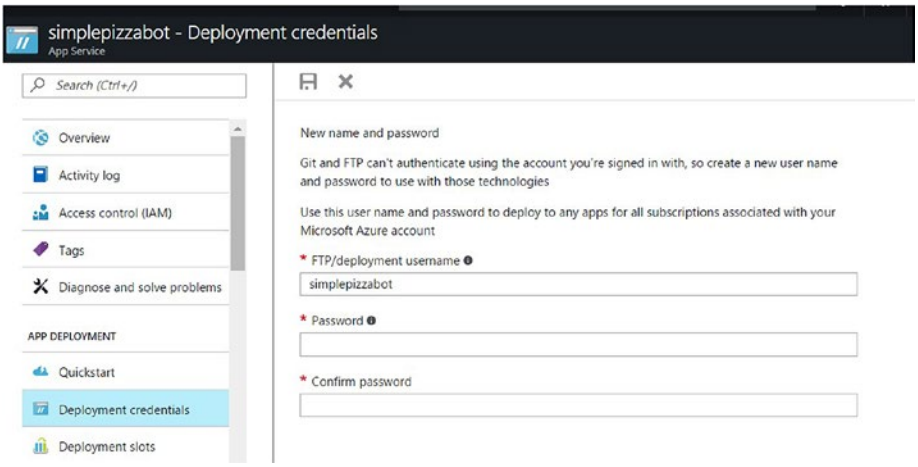

Figure 3-15. *Setting up deployment credentials for continuous publish*

Run the following commands to push the source code to the Git repository just created:

```
git add .
git commit -m "first commit"
git push azure master
```

You will be prompted to enter the deployment credentials—the username and the password chosen while setting up deployment credentials.

We should now be able to remotely test the application deployed on Azure using Bot Emulator. For bot applications deployed remotely, we should secure the communication channel so that only authentic clients can invoke the web endpoints; this happens via the MS Bot framework's Developer Portal. Let us quickly register the bot on Bot Developer Portal at `https://dev.botframework.com`. Login to Bot Developer Portal using a Windows Live Account or Hotmail account and register the bot as shown in Figure 3-16. For information on registering a bot on Developer Portal, see Chapter 2.

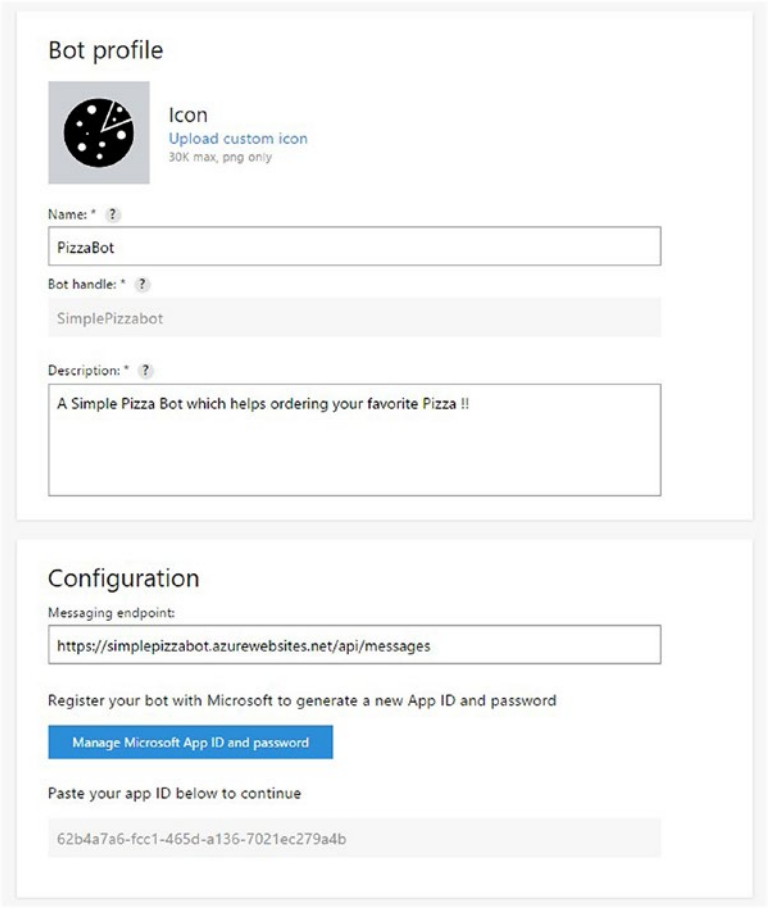

Figure 3-16. *Bot registration on Developer Portal*

Microsoft App ID and password serve as the credentials we need to connect to the Pizza Bot. The Node.js bot application deployed to Azure should be updated with the App ID and password so client applications can connect. Login to Azure Portal and add the relevant application settings to the Azure App Service, as shown in Figure 3-17.

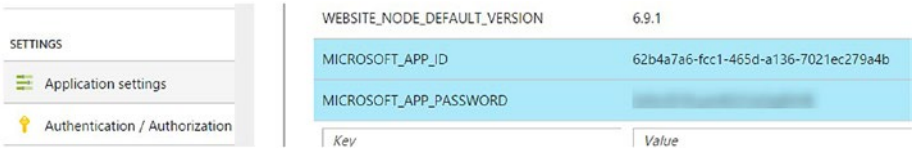

Figure 3-17. *Update Microsoft app ID and password in App Service settings*

We should now be able to test the bot using the embedded chat control on the Bot Developer Portal, or we could add it to Skype. Figure 3-18 shows the conversation with the bot, which is using Skype's interface remotely.

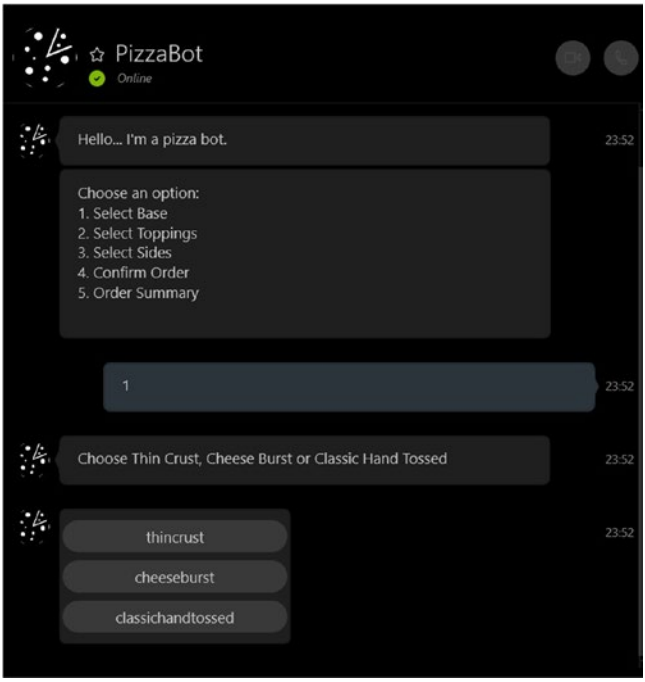

Figure 3-18. *Communicating with Pizza Bot via Skype*

You can now connect to the bot application using various channels, which are shown in Figure 3-19.



Figure 3-19. *Bot channels*

Summary

Microsoft offers a Bot Builder SDK so Node.js developers can build smart and intelligent bots. Bots can be authored using any Node.js editor; however, VS Code is the preferred one given its Intellisense, cross-platform, debugging, and built-in source-control capabilities. Bots have dialogs, which are like routes in a web application. A message from the user is routed to a dialog and then rerouted based on the result or return type of the dialog. The user's input can be collected using bot prompts. The appearance of the dialogs or prompts differs from channel to channel. The SDK offers a variety of options to save state using the `Session` object. The data can be saved globally or per user/conversation. Azure offers a cross-platform command-line tool call Azure CLI, which can be used to create and manage resources on Azure. VS Code contains built-in Git source control capabilities, which can be used to set up the continuous publish feature.

CHAPTER 4

Channels

Channels are the medium through which the conversation happens between your bot API and the client. There are numerous channels that a conversation can pass through. Though most of them implement the basic text-based conversation styles, like request/reply, there are a few characteristics that are specific to each channel. Bot developers can use the channel-specific characteristics via the underlying platform and provide an enriched experience to bot users. In this chapter, we will learn how to use the channels and channel data that exist in the Microsoft Bot framework's vast ecosystem. We will build a bot that uses the personalized features of the underlying channels by customizing the payload.

The following topics will be discussed in this chapter:

- Channels overview
- Build a chat bot using email client
- Build a chat bot using Slack channel and API:
  - Multi-dialog bot using Slack and Slack channel data
  - Onboard bot application to Slack
  - Remotely debug Slack bot on development machine

Channels and Channel Data

A bot's channel can be configured from the Bot Developer Portal (`https://dev.botframework.com`). The bot channels Web Chat and Skype are enabled for you by default. A few other channels available for you to explore are the following:

- Bing
- Cortana
- Facebook Messenger
- Kik
- Skype for Business

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_4

- Outlook
- Microsoft Teams
- Slack
- Telegram
- Direct Line
- Group Me
- Twilio (SMS)

To enable any of these channels, you should register your bot on the Developer Portal. Any of these channels can be enabled for your bot by clicking on the **Add Channel** buttonz in the Channels section of the Developer Portal, as shown in Figure 4-1.

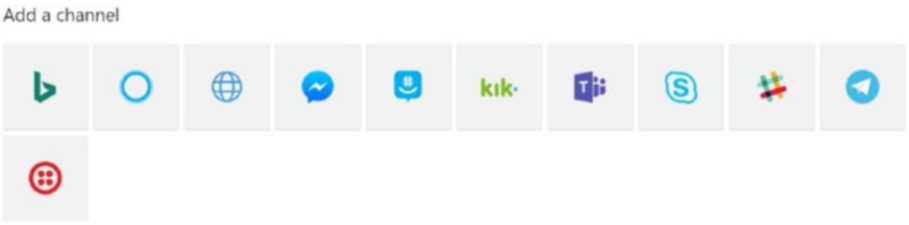

Figure 4-1. *Bot channels*

Bot channel configurations are different from one another. Most of the bots' registrations need an account per channel; for example, to register with an email client we would need an Office 365 email ID. A few other channels, like Slack (which we will see in the next section), require an application to be onboarded with the channel so that the channel can monitor the usage. The onboarding details provided by the channel are used to register the bot on the Developer Portal. Registering a channel for your bot should make your bot discoverable on the corresponding channel.

■ **Note** To enable a Skype for Business bot, you should be a Tenant Administrator on the Skype for Business online environment. For more details on enabling the Skype for Business channel, please visit `https://msdn.microsoft.com/en-us/skype/Skype-For-Business-Bot-Framework/docs/overview`.

Once a bot channel has been enabled, it shows up in the Connect to Channels section, as shown in Figure 4-2. The Developer Portal also allows you to update the channel configuration after enabling. If you find your bot misbehaving or hacked, you can disable the channel by clicking on the **Disable** button on the Channel Configuration page.



Figure 4-2. *List of channels configured for a bot*

The Health column in Figure 4-2 shows the recent issues discovered while the bot was running. Clicking on the issues shows more-detailed information on the failed scenarios, as shown in Figure 4-3. This can be used to troubleshoot channel-related issues in real-time.

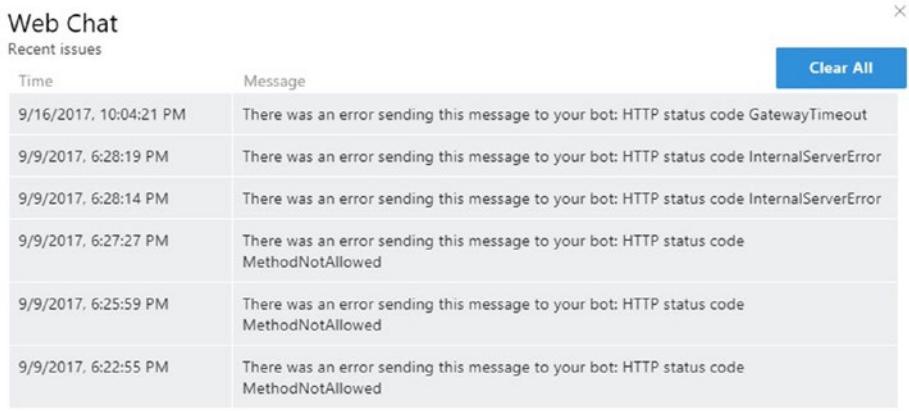

Figure 4-3. *Issues discovered by channel connector*

Clicking the **Get bot embed codes** link provides HTML snippets that can be embedded in any application host. Invoking the embedded code invokes the bot. There is no limit on the number of channels you can enable a bot on. You can also create your custom client for the bot by implementing the Direct Line API. Direct Line API is a REST API that can be used to send/receive messages with custom channel data to the bot application. There are also client libraries available for C# and Node.js. For more details on building a custom client using Direct Line API, please visit `https://docs.microsoft.com/en-us/bot-framework/rest-api/bot-framework-rest-direct-line-3-0-concepts`.

Channel Data

`ChannelData` is a special property of the activity object that is passed between the client and the bot API. The channel data is a dynamic object that can carry data in any schema. Channel data contains channel-specific information, and it can also be used to pass special information to the channel. The `ChannelData` property contains the data in the form of a JSON serialized string.

For example, Figure 4-4 shows the channel data coming from the Slack channel.

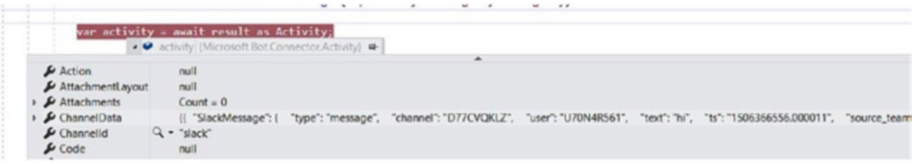

Figure 4-4. *Channel data from Slack*

Figure 4-5 shows the schema for the Slack channel data.

```
{{
  "SlackMessage": {
    "type": "message",
    "channel": "D77CVQKLZ",
    "user": "U70N4R561",
    "text": "hi",
    "ts": "1506366556.000011",
    "source_team": "T70FV929Z",
    "team": "T70FV929Z"
  },
  "ApiToken": "xoxb-245881152019-JAkMD7iZlhyKuI5EjFfSi6sl"
}}
```

Figure 4-5. *JSON schema for Slack channel data*

To implement channel-specific functionality, you can use the `ChannelId` property. The value shows the channel name, as shown in the Figure 4-5 (for example, slack). When replying to the user, you can use the channel-specific schema to implement custom features. The channel data can be sent in the response by designing a JSON-serializable class as shown in Figure 4-6.

```
[JsonObject]
public class EmailChannelData
{
    /// <summary>
    /// Gets or sets the HTML body.
    /// </summary>
    /// <value>
    /// The HTML body.
    /// </value>
    [JsonProperty("htmlBody")]
    public string htmlBody { get; set; }

    /// <summary>
    /// Gets or sets the subject.
    /// </summary>
    /// <value>
    /// The subject.
    /// </value>
    [JsonProperty("subject")]
    public string subject { get; set; }

    /// <summary>
    /// Gets or sets the importance.
    /// </summary>
    /// <value>
    /// The importance.
    /// </value>
    [JsonProperty("importance")]
    public string importance { get; set; }
}
```

Figure 4-6. *C# class for Email channel data matching the JSON schema*

Assign the value to the activity object's channel data property as follows:

```
Var emailChannelData = new EmailChannelData()
{
     htmlBody = $"<html><body style=\"font-family: Calibri; font-size:
     11pt;\">{weatherHTML}</body></html>",
     subject = $"Weather at {location}",
     importance = "normal"
};
```

```
if (activity.ChannelId != "email")
{
     reply.Text = emailChannelData.ToString();
}
```

Alternatively, you can build a JSON object by using the `Newtonsoft.JSON` library, as explained in the following code:

```
reply.Text = new JObject(
                new JProperty("htmlBody", $"<html><body style=\"font-
                family: Calibri; font-size: 11pt;\">{weatherHTML}
                </body></html>"),
                new JProperty("subject", $"Weather at {location}"),
                new JProperty("importance", "normal")).ToString();
```

Similarly, you can read the channel data by using the `Newtonsoft.JSON` library, as shown here:

```
// Read by path within JSON string
emailChannelData.SelectToken("htmlBody")?.ToString();
```

Build a Chat Bot Using an Email Client

Unlike other bot channels, which exercise standard conversation patterns through a chat window, the Email channel allows bots to send automated emails as per the business logic. An email client is one of the MS Bot framework's list of channels. The bot application is invoked when an email is sent to the email address used during channel configuration.

The following steps explain the process of building a bot application and configuring it for the Email channel. In this sample application, we will build a bot that responds to emails with weather information. Follow these steps:

1. Open Visual Studio as Administrator and create a new bot application; name it `email channel`.

2. Add the following class for serializing the Email channel data:

```
[JsonObject]
 public class EmailChannelData
 {
     /// <summary>
     /// Gets or sets the HTML body.
     /// </summary>
     /// <value>
     /// The HTML body.
     /// </value>
     [JsonProperty("htmlBody")]
     public string htmlBody { get; set; }
```

```
        /// <summary>
        /// Gets or sets the subject.
        /// </summary>
        /// <value>
        /// The subject.
        /// </value>
        [JsonProperty("subject")]
        public string subject { get; set; }

        /// <summary>
        /// Gets or sets the importance.
        /// </summary>
        /// <value>
        /// The importance.
        /// </value>
        [JsonProperty("importance")]
        public string importance { get; set; }
    }
```

3. Update the `MessageReceivedAsync` function in `RootDialog.cs` with the following code. The code creates a message with custom channel data in response to the message from the bot client. The code also identifies the channel before responding to the request using the `activity.ChannelId` property.

```
/// <summary>
/// Messages the received asynchronous.
/// </summary>
/// <param name="context">The context.</param>
/// <param name="result">The result.</param>
/// <returns></returns>
private async Task MessageReceivedAsync(IDialogContext
context, IAwaitable<object> result)
{
    var activity = await result as Activity;
    var reply = context.MakeMessage();
    string location = (activity.Text ?? string.Empty);
    var weatherHTML = this.getWeatherDetails(location);
    var emailChannelData = new EmailChannelData()
    {
        htmlBody = $"<html><body style=\"font-family: Calibri;
        font-size: 11pt;\">{weatherHTML}</body></html>",
        subject = $"Weather at {location}",
        importance = "normal"
    };
```

```
        if (activity.ChannelId != "email")
        {
            reply.Text = emailChannelData.ToString();
        }
        else
        {
            reply.ChannelData = emailChannelData;
        }
        // return our reply to the user
        await context.PostAsync(reply);
        context.Wait(MessageReceivedAsync);
    }
```

4. Add the following getWeatherdetails helper function to the RootDialog.cs. The function is used to get weather information from http://api.openweathermap.org.

```
    /// <summary>
    /// Gets the weather details.
    /// </summary>
    /// <param name="location">The location.</param>
    /// <returns></returns>
    private string getWeatherDetails(string location)
    {
        string htmlResponse = string.Empty;
        string response = string.Empty;
        try
        {
            using (var client = new HttpClient())
            {
                response = client.GetStringAsync($"http://
                api.openweathermap.org/data/2.5/weather?q={
                location}&appid=9aeafb54eb98a3b6380 4af59320
                66f8c&units=metric&mode=html").Result;
                htmlResponse += response;
            }
        }
        catch (Exception ex)
        {
            Debug.WriteLine(ex.ToString());
            htmlResponse = $"Invalid location, please only
            location in Email. For Ex: New Delhi, Current
            Location Passed: {location}, API Response:
            {response}";
        }
        return htmlResponse;
    }
```

5. Build and run the application. The application can be tested by using Bot Emulator.

6. Publish the application to Azure Portal (Figure 4-7).

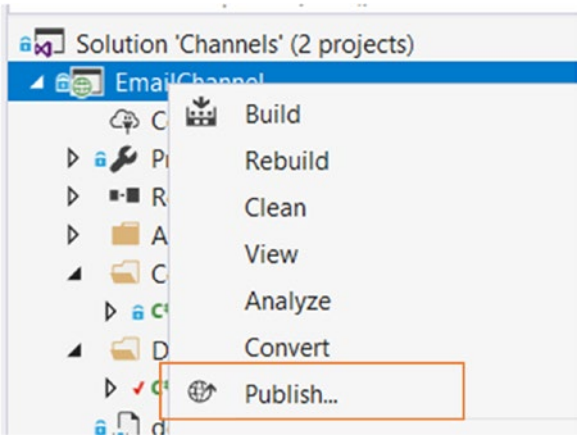

Figure 4-7. *Publish Email bot from Visual Studio*

7. Onboard the bot application to the Bot Developer Portal by filling in the details shown in Figure 4-8 on the bot's profile page.

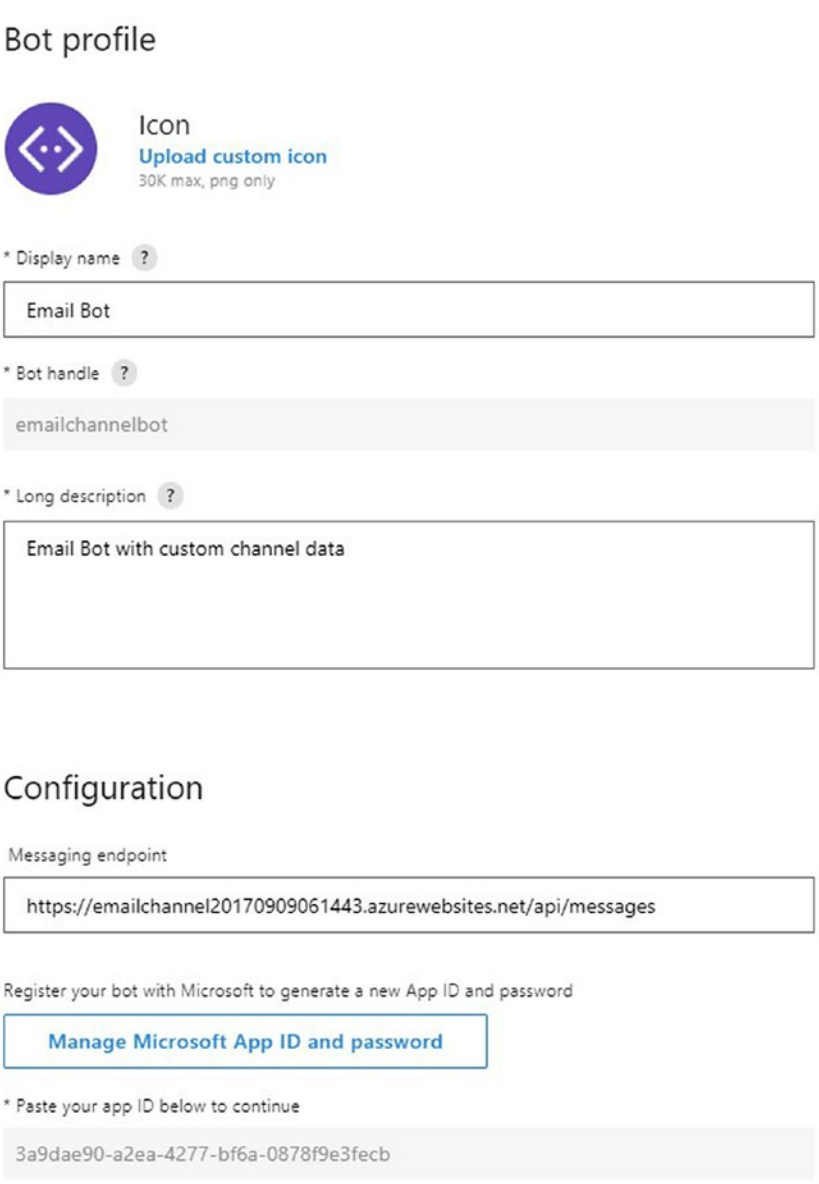

Figure 4-8. *Email bot registration*

8. Replace the messaging endpoint with the bot application's endpoint, provided by Azure, followed by `api/messages`.

■ **Note** During the onboarding process, you will be provided with a Microsoft App ID and password unique to your bot, which is required for authentication. Ensure you update the App ID and password in the `web.config` of the application and republish to Azure.

9. Click on **Email** in the Channels section and configure the email and password as shown in Figure 4-9. The email configured here will be used by your bot to listen for incoming messages.

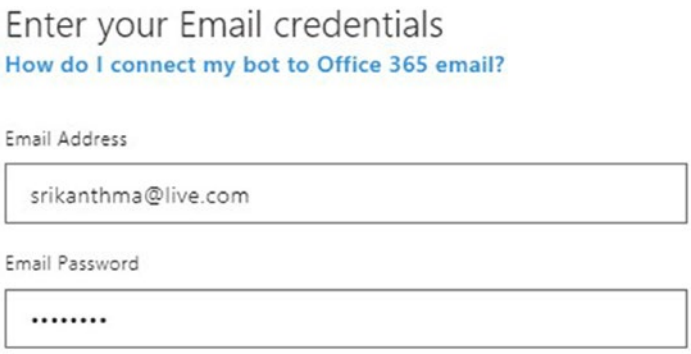

Figure 4-9. *Email (Office 365) channel configuration*

■ **Note** Email channel can only be used with Office 365 accounts; other email services are not currently supported.

10. Let us send an email to this address with the details of the location for which we seek weather information (Figure 4-10). Notice that we are not including any data other than location here.

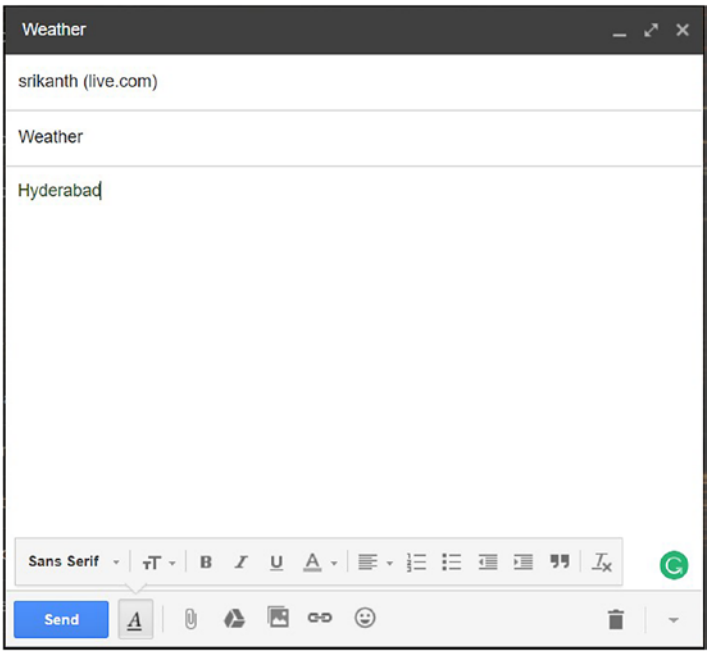

Figure 4-10. *Draft email to Weather email bot*

■ **Note** To build complex bots using the Email channel, we can use LUIS for processing the email body and to respond accordingly.

---

11. In no time you would receive an email with weather information from `openweathermap.org`, as shown in Figure 4-11.



Figure 4-11. *Email response from Weather email bot*

While building email bots, ensure you're not spamming or sending bulk emails by adhering to the code of conduct laid out at https://www.botframework.com/Content/Microsoft-Bot-Framework-Preview-Online-Services-Agreement.htm. Ensure the Email channel is disabled when the application is taken offline.

Build a Chat Bot Using Slack Channel and API

Slack (www.slack.com) is a team collaboration tool built by Stewart Butterfield. It is available on most of the desktop and mobile operating systems. Slack offers lots of collaboration features, like persistent chat rooms or channels organized by topic (not to be confused with bot channels), private groups, and direct messaging. All the content shared in the Slack tool is searchable; in fact, SLACK stands for Searchable Logs of All Conversation and Knowledge. Figure 4-12 shows a Slack desktop application on Windows 10.

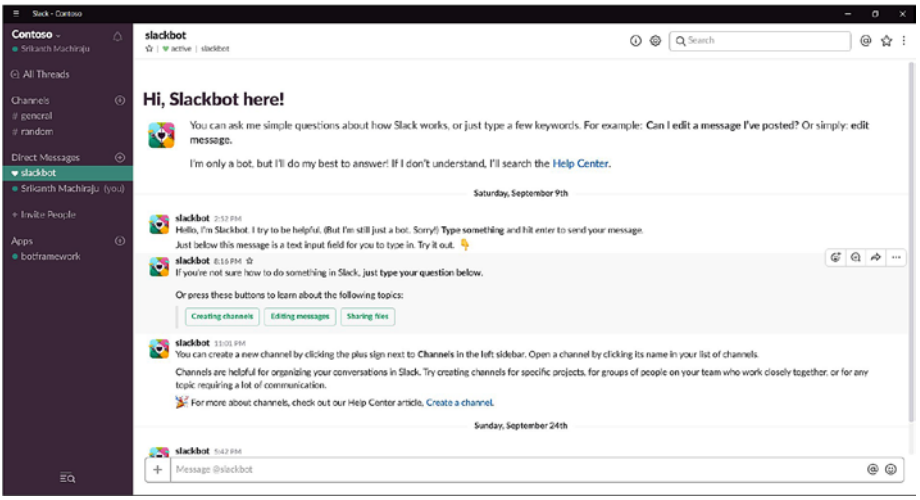

Figure 4-12. *Slack application on Windows OS*

Slack is one of the supported channels on the MS Bot framework. We can build a Slack bot by using the Slack API (https://api.slack.com) for formatting messages, linking attachments, and embedding images, action buttons, confirmation dialogs, and so on.

To enable a Slack bot in Developer Portal, we need an Application Client ID, Client Secret, and Verification Token, as shown in Figure 4-13. The following details are provided by Slack when the application is onboarded to their portal, but before this step we should onboard the bot to the MS Bot Developer Portal. This is because the Slack onboarding depends on the bot handle to uniquely identify our bot.

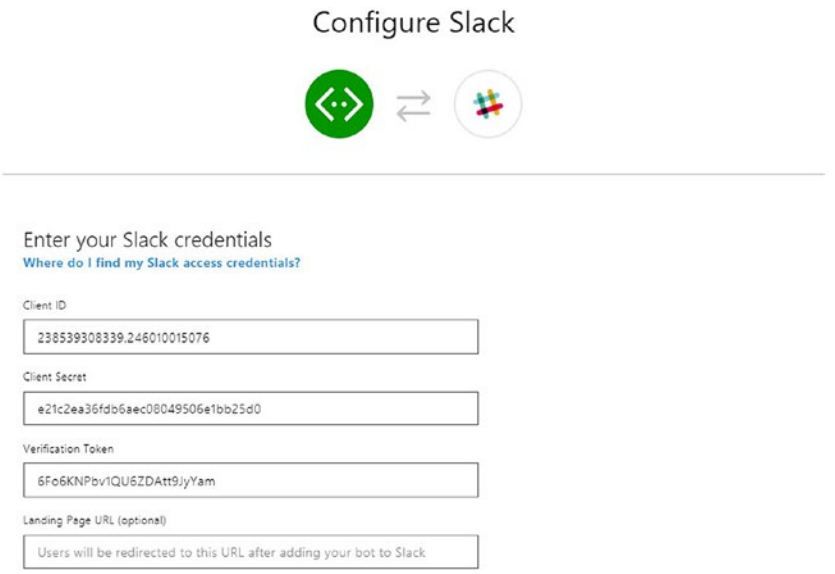

Figure 4-13. *Slack configuration in Bot Developer Portal*

In this section, we will build a Slack bot that helps us book a reservation in one of the famous hotels of Las Vegas. The application goes through a series of dialogs to ensure all the information is captured. The user is asked to choose a hotel from the list of destinations and to enter name, age, check-in date, and check-out date, and finally the bot responds with a hotel confirmation, as shown in Figure 4-14.



Figure 4-14. *Sample response from Slack Bot*

Multi-dialog Bot Using Slack and Slack Channel Data

Let us get started with building a Slack bot by going through the following steps. Open Visual Studio as Administrator and create a new bot application, naming it `Slack channel`.

In this section, we are going a build a series of dialogs. Rename the `RootDialog.cs` to `SlackDialog` and replace the `MessageReceivedAsync` method with the following code.

```
private async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
{
    var validDestinations = new List<string> { "palazzo", "bellagio",
    "mirage" };
    var activity = await result as Activity;
    var slackChannelData = JObject.FromObject(activity.ChannelData);
    var destination = slackChannelData.SelectToken("Payload.actions[0].
    value")?.ToString();
  if (!string.IsNullOrEmpty(destination) &&validDestinations.
  Contains(destination))
  {
       this.destination = destination;
       context.Call(new NameDialog(), this.NameDialogResumeAfter);
  }
  else
  {
        var reply = context.MakeMessage();
        reply.ChannelData = new SlackMessage
        {
           Text = "Hi Welcome to *Vegas tours Bot*, _book your Las Vegas
           stay from anywhere using Slack_ :hotel:",
           Attachments = new System.Collections.Generic.List<Models.
           Attachment>
           {
              new Models.Attachment()
              {
                 Title = "Which hotel do you want to stay at?",
                 Text = "Choose a Hotel",
                 Color = "#3AA3E3",
                 Callback = "wopr_hotel",
                 Actions = new List<Models.Action>()
                 {
                    new Models.Action()
                    {
                         Text = "The Palazzo (min. $350 per night)",
                         Name = "destination",
                         Type = "button",
                         Value = "palazzo"
                    },
```

```
                    new Models.Action()
                    {
                         Text = "Bellagio Hotel and Casino (min. $300 per
                         night)",
                         Name = "destination",
                         Type = "button",
                         Value = "bellagio"
                    },
                    new Models.Action()
                    {
                          Text = "The Mirage (min. $280 per night)",
                          Name = "destination",
                          Type = "button",
                          Value = "mirage",
                          Style = "Danger",
                          Confirm = new JObject(
                                        new JProperty("confirm",
                                            new JObject(
                                                new JProperty("title",
                                                "Are you sure, you may
                                                want to check the events at
                                                Palazzo?"),
                                                     new JProperty("text",
                                                     "You may want to check
                                                     events, restaurants at
                                                     the Palazzo?"),
                                                     new JProperty("ok_
                                                     text", "Yes"),
                                                     new JProperty("dismiss_
                                                     text", "No")
                                                ))))}
                       }
                    }
               }
               };
               await context.PostAsync(reply);
               context.Wait(MessageReceivedAsync);
          }
     }
```

The above code greets the user with formatted text, a stylish menu, and a smiley icon, as shown in Figure 4-15. The formatting is done using the Slack API; more details on basic formatting can be found at https://api.slack.com/docs/message-formatting#message_formatting.

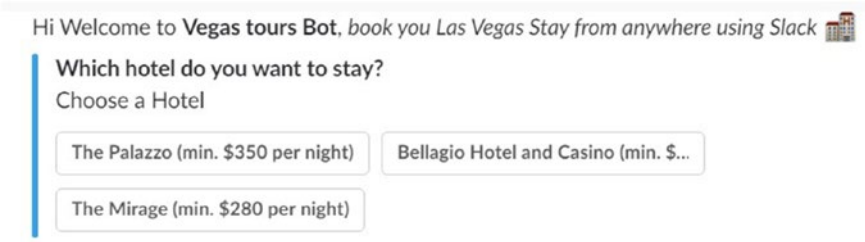

Figure 4-15. *Sample response from Slack bot with buttons*

We can also build confirmation popups that will show up upon the selection of a button, as shown for **The Mirage** in the below code. The confirmation popup looks like Figure 4-16 and appears when you click on **The Mirage**.

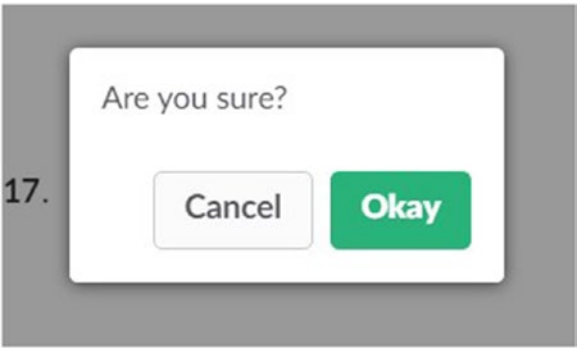

Figure 4-16. *Confirmation popup built using Slack API*

The following code extracts the data from the callback once the user makes a selection:

```
var slackChannelData = JObject.FromObject(activity.ChannelData);
var destination = slackChannelData.SelectToken("Payload.actions[0].value")?.
ToString();
if (!string.IsNullOrEmpty(destination) &&
validDestinations.Contains(destination))
  {
       this.destination = destination;
       context.Call(new NameDialog(), this.NameDialogResumeAfter);
  }
```

Here, we are using `Newtonsoft.Json` to extract a value from a JSON serialized string by crawling the path. Alternatively, you could design a C# class matching the schema of the response and deserialize the JSON string using the `activity.GetChannelData<T>` method on the activity object.

Upon successful extraction of the selection made by the user, the application directs the user to the next dialog. For each subsequent dialog, the callback is registered on the `SlackRootDialog` class to extract the selection made by the user. The following is the content of the next dialog:

```
[Serializable]
    public class NameDialog : IDialog<string>
    {
        public async Task StartAsync(IDialogContext context)
        {
            await context.PostAsync("What is your name?");

            context.Wait(this.MessageReceivedAsync);
        }

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

            /* If the message returned is a valid name, return it to the
            calling dialog. */
            if ((message.Text != null) && (message.Text.Trim().Length > 0))
            {
                /* Completes the dialog, removes it from the dialog stack,
                and returns the result to the parent/calling
                    dialog. */
                context.Done(message.Text);
            }
        }
    }
```

Figure 4-17 shows the conversations between the bot and the slack user.

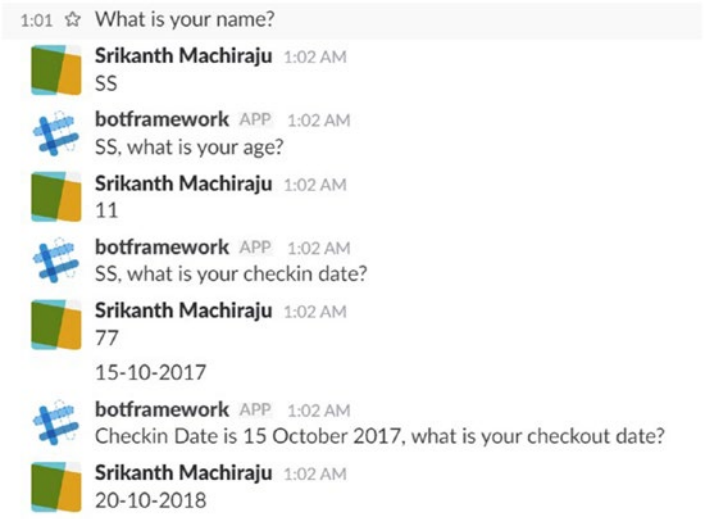

Figure 4-17. *Conversation with Slack bot*

The stint of conversations ends with a confirmation of the booking, as shown in Figure 4-18.



Figure 4-18. *Final response from Slack bot*

The following code shows the final response with embedded smiley, image, and link in the response message:

```
// Sample Image embedded in Slack Message
 string imageUrl = "https://www.palazzo.com/content/dam/palazzo/Suites/
 bella/Palazzo-Bella1-med_900x600.jpg.resize.0.0.474.316.jpg";
 // Sample Title URL
 string titleURL = "https://www.vegas.com/tours/";
 this.checkOutDate = await result;
 var reply = context.MakeMessage();
 reply.ChannelData = new SlackMessage
 {
     Text = $"Hi *{name}*, your stay is confirmed from \n Your stay
     is from *{checkInDate.ToLongDateString()} to {checkOutDate.
     ToLongDateString()}*. \n Have a pleasant stay :smiley:",
     Attachments = new System.Collections.Generic.List<Models.Attachment>
     {
          new Models.Attachment()
         {
         Title = "Here are few things you can do in Vegas !! :thumbsup_all:",
              Text = "Hotel Pics",
              ImageUrl = imageUrl,
              TitleLink = titleURL
         }
      }
  };
  await context.PostAsync(reply);
```

If you want to know the complete request/response schema, you can go to https://api.slack.com/docs/messages. Slack API also gives you a message builder that can help with the preview of a message for any given JSON response, as shown in Figure 4-19. To use the message builder, navigate to the below link and click on **Message builder** under the Messages section. The following link shows an example of the Slack message schema: https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-channeldata.

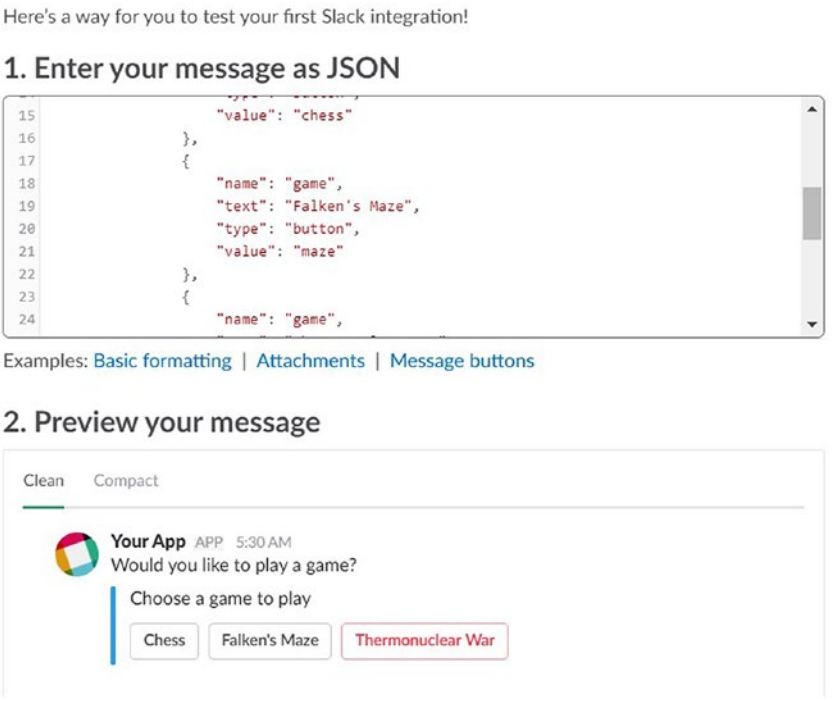

Figure 4-19. *Slack message builder*

Onboarding a Slack Bot

The steps for onboarding the Slack bot are quite different from the Email bot registration we saw earlier. The following steps are common for any bot registration:

1. Publish the bot application as an Azure App Service (or any other website-hosting environment).
2. Register the bot on the Developer Portal using the host name of the bot application. Remember the bot handle used during the registration process, as it will be used in the next steps.
3. Update the Microsoft App ID and password in the `web.config` and republish to the Developer Portal.

After completing the above steps for the sample project (just built), we should be able to enable the Slack channel by clicking on the Slack icon in the Channels section. But, before that, we should onboard our bot to Slack. The steps corresponding to connecting the bot to Slack are detailed at `https://docs.microsoft.com/en-us/bot-framework/channel-connect-slack`. Gather the credentials from the Slack portal and update the same on the Bot Developer Portal accordingly (Figure 4-20).



Figure 4-20. *Applying configuration in Bot Developer Portal using information from Slack API portal*

After completing the above steps, the bot application should be visible in your Slack application under the Apps section.

Remote Debugging Slack Bot on Development Machine

In order to debug Slack bot conversations from Visual Studio, the calls from the bot channels to the bot application running on Azure should be intercepted. Ngrok helps us create secure tunnels that can be used to connect a program running on your local machine (like Visual Studio) to a cloud service. You can download `ngrok.exe` from `https://ngrok.com/download` or from the code samples linked to this chapter.

To debug the application from Visual Studio, follow these steps:

1. Run the application in Debug mode from Visual Studio.
2. Run the following command at the path where `ngrok.exe` is placed. The bot should be running on port 3979; if not, change the port accordingly.

   ```
   ngrok.exe http 3979 -host-header="localhost:3979"
   ```

3. Ngrok provides both an HTTP and an HTTPS tunnel, as shown in Figure 4-21.

```
ngrok by @inconshreveable

Session Status                online
Version                       2.2.8
Region                        United States (us)
Web Interface                 http://127.0.0.1:4041
Forwarding                    http://a39e1f41.ngrok.io -> localhost:3979
Forwarding                    https://a39e1f41.ngrok.io -> localhost:3979
```

Figure 4-21. *Ngrok configuration for remote debugging*

4. Copy the HTTPS forwarding address (`https://a39e1f41.ngrok.io`).
5. Update the configuration endpoint on the Bot Developer Portal, as shown in Figure 4-22.

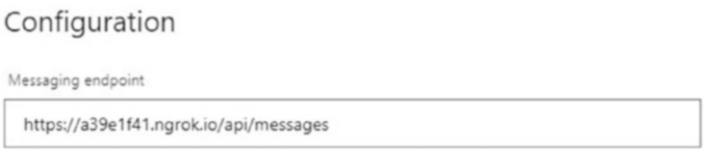

Figure 4-22. *Ngrok configuration on Bot Developer Portal*

6. Save changes.
7. Open the Slack application and start a conversation with your bot.

You should now be able to debug your application from Visual Studio in real-time and inspect the channel data. The MS Bot framework provides several options for customizing the conversation experience for channels like Facebook, Telegram, kik, and so on.

Summary

Channels and channel data can be used to customize the conversation experience by using the channel-specific features. For example, we can use the personalized features of Slack, like smileys, attachments, and notifications, by responding to the user with custom channel data. `ChannelData` is a property of the activity class that can be used to send data that only the channel can understand and interpret. The `ChannelData` property contains the data in the form of a JSON serialized string. Email is also a channel supported by MS Bot framework. The bot responds via email when an email is sent on the registered email ID. The Email channel can only be configured by using Office 365 accounts, as other email services are not currently supported.

CHAPTER 5

Bot Conversations

Conversations form the core of any bot implementation. Conversations refer to the process of exchanging ideas and content between multiple parties. When two or more people talk to each other, they have entered a conversation. Bot implementation is all about conversations. A bot mimics and simulates conversation as two humans would converse in their natural form. Bots are implemented to hear and talk to their users, provide information, guide them, act on their behalf, and keep providing updates about its current state. A conversation with a bot could be text or speech driven.

Conversations are the way to interact and stay engaged with a bot. MS Bot framework provides a feature-rich framework that supports a variety of conversation types. In this chapter, we will focus on the details of the messages that are exchanged between a bot and a user as part of a conversation, deep dive into message internals, and explore different types of conversation options available as part of MS Bot framework. MS Bot provides cards that help when exchanging rich content with users as part of a conversation. This chapter includes the following:

- Attachments
- Images
- Hero card
- Thumbnail card
- Adaptive card
- Carousel card
- Prompt dialogs

Microsoft offers these rich media conversations not just for .Net framework but also for Node.js developers. The examples used in this chapter will use C# as the choice of language; however, the code for Node.js will also be shown for each option.

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_5

Understanding Conversations

Conversations are a series of messages passed between bots and users. A conversation can take the following shapes:

- One-to-one conversation between a user and a Bot
- One-to-many conversations between a Bot and users
- Conversation between Bots

Almost all channels implement one-to-one conversation between a user and bot, but not all channels implement group conversations. Documentation of individual channels should be referred before implementing group conversation. Email is one of the channels that implements group conversation.

Messages

Messages are the core to bot conversations. Messages are passed from a sender to a recipient. Both user and bot are sender and recipient in a conversation.

There are multiple types of messages that can be transmitted between users and bots, as shown in Table 5-1.

Table 5-1. *List of Message Types in MS Bot Framework*

Message Type	Description
`contactRelationUpdate`	The channel sends this message to indicate that the user added or removed your bot from their contacts list in the channel. If the user added your bot to their contacts list, the message's action property is set to add; otherwise, it's set to remove.
`conversationUpdate`	The channel sends this message to indicate that one or more users has joined or left the conversation, or the topic name changed. For a list of users that joined the conversation, see the message's `addedMembers` property; otherwise, see the `removedMembers` property. You can use this message to welcome new users to the conversation.
`deleteUserData`	The channel sends this message to indicate that the user wants the bot to delete all of their personally identifiable information (PII) that the bot may have saved using the User State REST API. If you receive this message, you must delete the user's data. After you delete the user's data, you should send them a message indicating that it's been deleted.

(*continued*)

Table 5-1. (*continued*)

Message Type	Description
`message`	The bot or user sends this message to advance the conversation; for example, the user sends a message asking for information, and your bot replies with a message that answers the user's question. Most messages that you send and receive will be of this type.
`ping`	The bot is sent this message to verify that its URL is accessible. The bot should respond with HTTP status code 200 OK, and may respond with 401 Unauthorized or 403 Forbidden.
`typing`	The channel or bot sends this message to indicate to the other party that they're working on a reply. Not all channels support this message.

The most important type of message in the shown list is `message`. All general conversation between a user and bot happen using this type. Also, a bot can check for message types that are part of its implementation using an Activity object.

Activity

Messages are represented as `Activity` objects within a bot. A bot interacts with a message using an `Activity` object. This class is defined within the MS Bot framework. When a user sends a message, the bot receives the message in the form of an `Activity` object, and a bot in turn creates an `Activity` object to reply to the user. This object contains the complete context under which the bot is executed. Some of the important information encapsulated by `Activity` objects is shown in Table 5-2.

Table 5-2. *Properties in an Activity Object*

Property	Description
`conversation`	Conversation's ID to which the message is related
`from`	Sender's ID of the message
`locale`	Locale information of message origination
`recipient`	Receiver's ID of the message
`ID`	Message identifier
`replyToId`	When replying to a message, set this to the contents of the ID property from the user's message.
`type`	Type of message. Could be any valid value from Table 5-1.
`text`	The actual message content in the form of text passed to recipient
`textformat`	Allows adding markdown to text messages
`attachments`	Used for sending rich content in the form of audio, video, images, and rich cards.

Relationship Between Channels, Conversation, User, and Bot

Conversation happens over a channel. It is an absolute must for a conversation to be hosted on a channel. Connectors provide multiple types of channels, like Skype, Facebook, Slack, and more. Conversations, users and Bots can be identified using Bot framework provided identifier. A channel can identify and weave these entities together as a unit and enable conversation with the help of their IDs. Figure 5-1 shows the hierarchal relationship between channels, conversations, and messages.

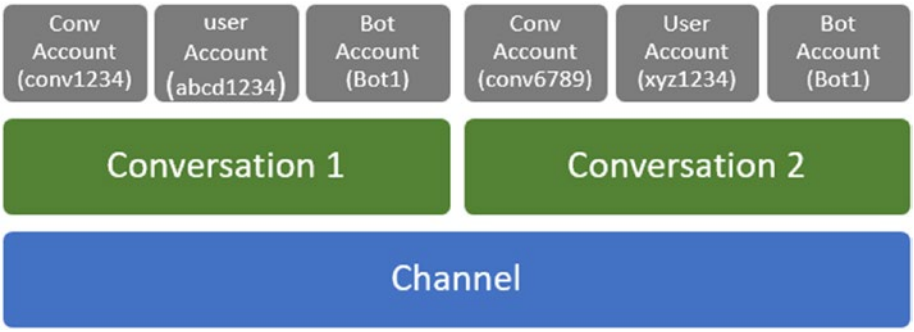

Figure 5-1. *Channels, conversations, and messages are related in a hierarchical manner*

The **`Conversation`** property shown earlier contains information about the conversation ID. The **`From`** property of the `Activity` object contains the ID and name of the message sender, the **`Recipient`** property contains the ID and name of message recipient, and the **`ID`** property contains the message identifier. Replies are made to the message using the **`replytoID`** property.

It is important to note that the conversation ID, Bot ID, and user ID remain constant for the entire session between a user and bot; however, for every message sent, the MS Bot framework generates a new message ID. A message ID is not generated when a reply to previous message is sent. In short, a message ID is constant for a pair of responses and not more.

Message Under the Hood

A message is sent from sender to recipient in JSON (JavaScript Object Notation) format. A sample message in JSON format is shown in Figure 5-2.

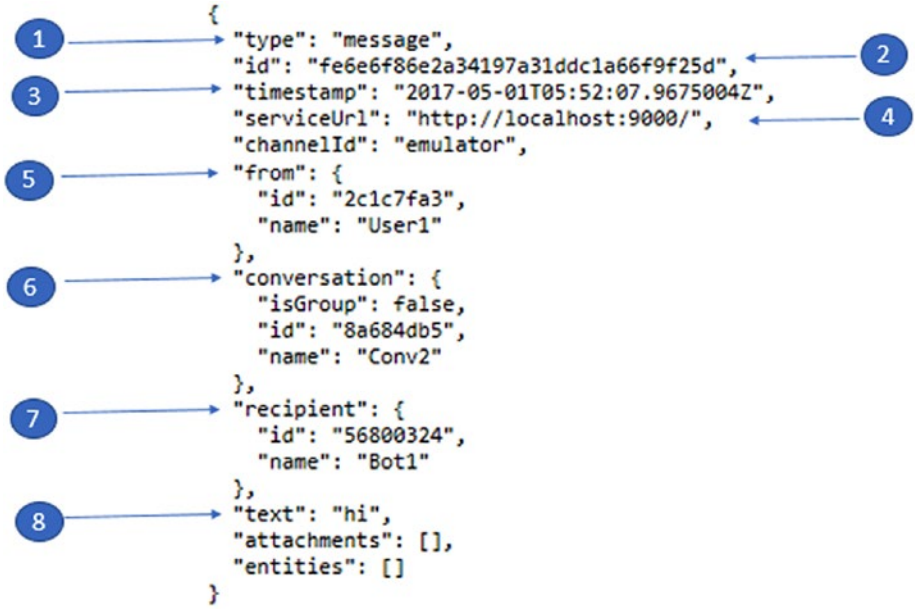

Figure 5-2. *Sample message in JSON format passed between a sender and a recipient. In this case, it's from a user to a bot.*

This message in JSON format is deserialized into an `Activity` object when it reaches a bot. The important properties are marked with numbers

1. The type of message passed to the bot. It is of type `message`.
2. The ID of the message. This ID is generated upon the initiation of the message. If the bot wants to reply to this message, it should use the `ReplyToID` property and set it to this ID.
3. Date and time of message.
4. The originating URL
5. Channel Account ID of user as sender of message
6. Channel Account ID of conversation
7. Channel Account ID of bot as recipient of message
8. Text sent by user to the bot

Conversation Under the Hood

A bot is primarily a REST (Representational State transfer) endpoint. It provides an endpoint through which users can interact with it. Typically, a conversation is initiated by a user, although it is possible for even a bot to start a conversation. When a user starts a conversation using a channel (Facebook, Skype, etc.), it sends an HTTPS POST message to `https://api.botframework.com/v3/conversations`. The request reaches the connector, and if the connector can establish the conversation with the bot as specified in the URL, a new conversation ID is generated. A new channel account for the conversation is created. By this time, a bot account, user account, and conversation account are established on the channel. A request is sent to the bot using `https://api.botframework.com/v3/conversations/{ConversationID}/activities`. This is the same conversation ID shown as item 2 in Figure 5-2. This request contains the identical message as shown in Figure x. The bot creates a new `Activity` object in response to the request and replies using the same conversation ID. The bot uses `https://api.botframework.com/v3/conversations/{ConversationID}/activities/{activityID}`. The "From" and "Recipient" information are swapped, but the conversation ID remains the same, as shown in the `replyToID` property in Figure 5-3.

```
{
  "type": "message",
  "timestamp": "2017-05-01T05:52:08.3765478Z",
  "from": {
    "id": "56800324",
    "name": "Bot1"
  },
  "conversation": {
    "id": "8a684db5",
    "name": "Conv2"
  },
  "recipient": {
    "id": "2c1c7fa3",
    "name": "User1"
  },
  "locale": "es-es",
  "text": "How are you!!",
  "replyToId": "fe6e6f86e2a34197a31ddc1a66f9f25d"
}
```

Figure 5-3. *Sample message in JSON format depicting a reply to an incoming message*

`ReplyToID` contains the same conversation ID that was in the incoming message. Now, if the user sends another message to the bot not in reply to their first message, a new message ID is generated, and rest of the information remains the same. This is shown in Figure 5-4.

```
{
  "type": "message",
  "id": "524146986c854215bd8cf4eebbb966ac",
  "timestamp": "2017-05-01T06:26:49.2522547Z",
  "serviceUrl": "http://localhost:9000/",
  "channelId": "emulator",
  "from": {
    "id": "2c1c7fa3",
    "name": "User1"
  },
  "conversation": {
    "isGroup": false,
    "id": "8a684db5",
    "name": "Conv2"
  },
  "recipient": {
    "id": "56800324",
    "name": "Bot1"
  },
  "text": "Hi",
  "attachments": [],
  "entities": []
`
```


Figure 5-4. *Sample message consisting of different message ID for every new message*

Building Bots with Conversations

It is quite easy to understand the concept of attachments with the help of an analogy. The following sections discuss this.

Attachments

Think about an email and its contents. An email consists of content in the form of text. It also at times contains rich content in the form of images, audio, video, and other formats, including binary format. This rich content is added to the email as attachments. It is part of the overall email content but is separated from the text content. Bots also provide features to add rich content to the message exchange. Rich media like audio, video, images, animation, and so forth can be added to the messages as attachments. Both the text and attachments are part of the message (activity) payload and are transmitted to bot and users.

MS Bot framework provides the `attachments` object to easily send and receive attachments, as shown in Figure 5-5.

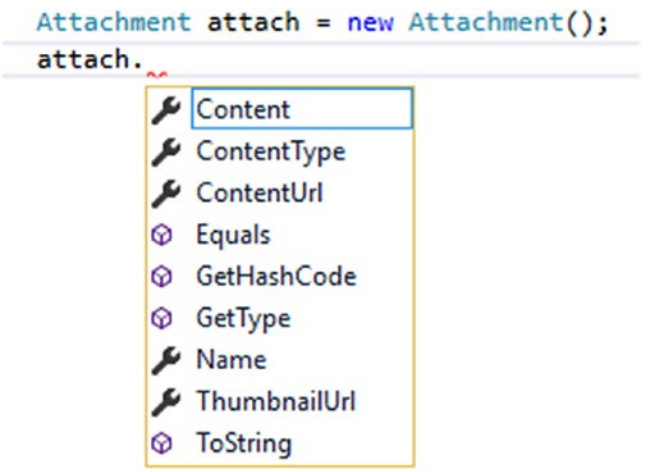

Figure 5-5. *Attachment object and its properties*

Adding and sending an attachment in a message is quite simple. Follows these steps:

1. Create an `attachment` object:

   ```
   Attachment attach = new Attachment();
   ```

2. Determine the content type of the attachment. The `ContentType` property of `attachment` accepts valid MIME (Multipurpose Internet Mail Extensions) types. If you're attaching an image, set the `contentType` property to the image media type and its subtype to PNG, GIF, or JPEG (for example, image/png):

   ```
   attach.ContentType = "image/png";
   ```

3. Set the `ContentUrl` or `ThumbnailUrl` property to the actual location of the media based on type of media and channel used:

   ```
   attach.ContentUrl = "https://upload.wikimedia.org/wikipedia/en/a/
   a6/Bender_Rodriguez.png";
   ```

4. Provide `name` for attachment:

   ```
   attach.Name = "SampleAttachment";
   ```

5. For channels that support inline images, you can set the `ContentUrl` property to a base64 binary value of the image (for example, …). The channel will display the image or the image's URL next to the message's text string:

```
attach.ContentUrl = "
AQ...."
```

6. Add it to the message's attachments array:

```
reply.Attachments = new List<Attachment> { attach };
```

After the attachment is attached, the message payload will correspond to the format, as shown in Figure 5-6. In this case, the bot is sending out a .png image as an attachment with some text associated with it.

JSON

```
{
  "type": "message",
  "timestamp": "2017-05-03T13:11:48.2292142Z",
  "from": {
    "id": "56800324",
    "name": "Bot1"
  },
  "conversation": {
    "id": "8a684dbb",
    "name": "Conv4"
  },
  "recipient": {
    "id": "2c1c7fa3",
    "name": "User1"
  },
  "text": "This is sample attachment!!!",
  "attachments": [
    {
      "contentType": "image/png",
      "contentUrl": "https://upload.wikimedia.org/wikipedia/en/a/a6/Bender_Rodriguez.png",
      "name": "SampleAttachment"
    }
  ],
  "replyToId": "2aed897b6a5247c2886f51fed0fb037e"
}
```

Figure 5-6. *Sample message in JSON format containing attachment details*

An interaction with this bot using Bot Emulator to view the image as an attachment is shown in Figure 5-7.

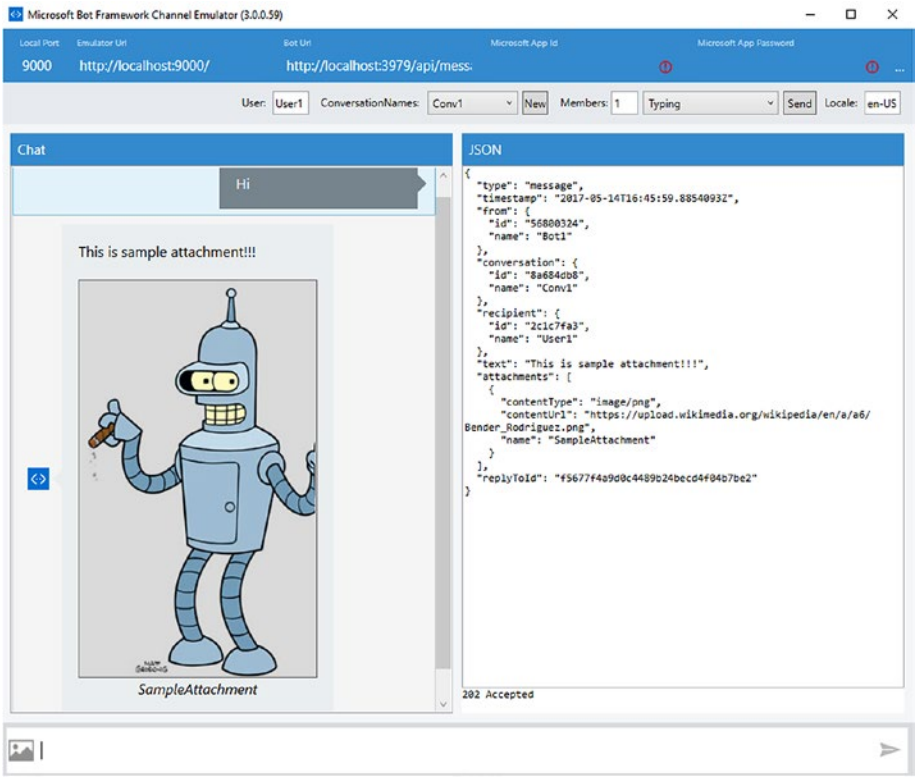

Figure 5-7. *Using Bot Emulator for viewing an image as an attachment received from a bot*

Receiving an attachment by bot in a message is also quite simple. Follow these steps:

1. Declare variables to hold return values:

```
Attachment attachment = null;
       long? contentLenghtBytes = 0;
```

2. Create `ConnectorClient` object to interact with channels:

```
ConnectorClient connector = new ConnectorClient (new Uri(activity.
ServiceUrl));
```

3. Check if incoming message has attachments:

```
if (activity.Attachments != null && activity.Attachments.Any())
```

4. It is possible that there are multiple attachments within a single incoming message. If there is more than one attachment, they can be looped through using the `attachments` property of the incoming message. Here, it is assumed that there is a single attachment within the incoming message:

```
var attachment = activity.Attachments.First();
```

5. When a user sends an attachment, the file is stored temporarily and should be downloaded by the bot. The bot should download the file using the classes available in the `System.Net` namespace. In this case, `HttpClient` is used and a new object of `HttpClient` is created:

```
HttpClient httpClient = new HttpClient ()
```

6. As seen before, each attachment object has a `ContentUrl` property. Using `HttpClient` and referring to the `ContentUrl` property, the file is downloaded in the bot context:

```
var responseMessage = await httpClient.GetAsync(attachment.
ContentUrl);
```

7. The downloaded file is read and can be stored in a `Stream` object:

```
Stream content1 = await responseMessage.Content.
ReadAsStreamAsync();
```

8. When the attachment is an image, an in-memory representation of it can be created:

```
Image img = System.Drawing.Image.FromStream(content1);
```

9. Details about the attachment can be obtained from the header. It is important to note that the `content` property contains all details about the file downloaded:

```
var contentLenghtBytes = responseMessage.Content.Headers.
ContentLength;
```

The full code is shown here:

```
Attachment attachment = null;
long? contentLenghtBytes = 0;

ConnectorClient connector = new ConnectorClient(new Uri(activity.
ServiceUrl));
```

```
if (activity.Attachments != null && activity.Attachments.Any())
 {
    attachment = activity.Attachments.First();
    using (HttpClient httpClient = new HttpClient())
    {
           var responseMessage = await httpClient.GetAsync(attachment.
           ContentUrl);

           contentLenghtBytes = responseMessage.Content.Headers.
           ContentLength;

            Stream content1 = await responseMessage.Content.
            ReadAsStreamAsync();

            Image img = System.Drawing.Image.FromStream(content1);

    }
}

Activity reply = activity.CreateReply($"You sent {attachment.ContentType}
which was {contentLenghtBytes.ToString()} characters");

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

An interaction with this bot using Bot Emulator to send an image as an attachment is shown in Figure 5-8.

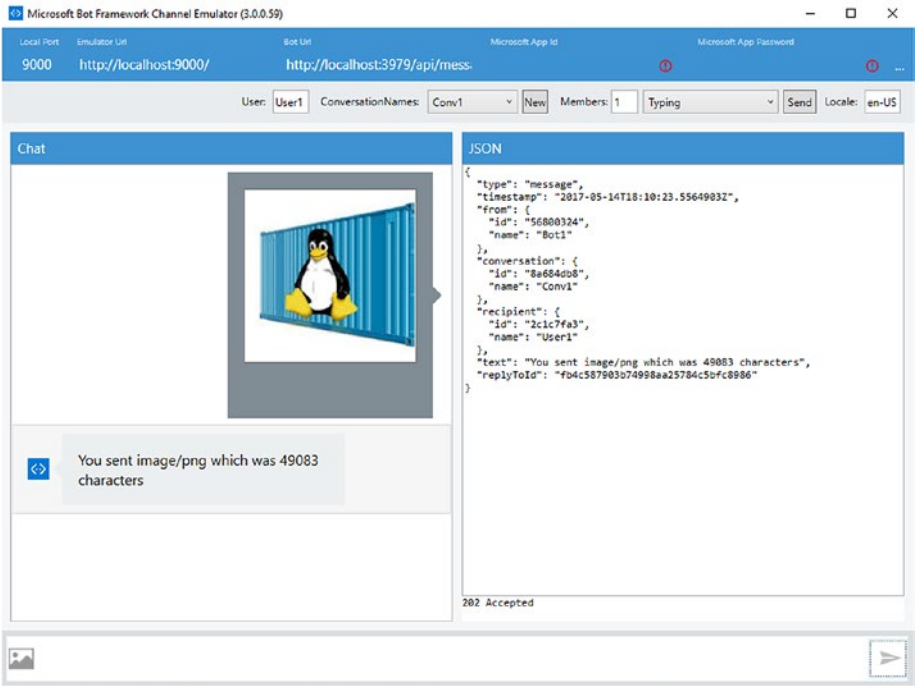

Figure 5-8. *Sending an image as an attachment using Bot Emulator*

Hero Card

The Hero card is a multipurpose card; it primarily hosts a single large image, a button, and a "tap action," along with text content to display on the card.

The properties used to configure a Hero card are listed in Table 5-3.

Table 5-3. *Properties Supported by Hero Card*

Property	Description
`Title`	Title of card
`Subtitle`	Link for the title
`Text`	Text of the card
`Images[]`	For a hero card, a single image is supported
`Buttons[]`	Hero cards support one or more buttons
`Tap`	An action to take when tapping on the card

A sample output of a Hero card is shown in Figure 5-9 in an emulator.

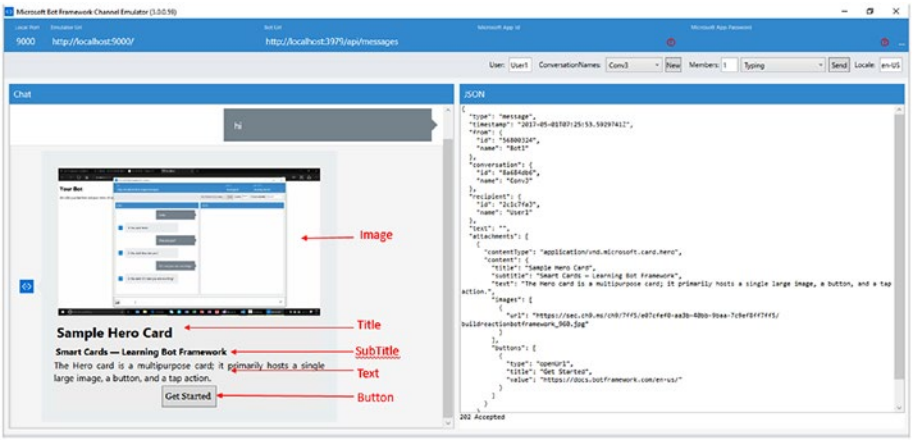

Figure 5-9. *Sample depicting Hero card in Bot Emulator*

The accompanying code has the complete project for a Hero card available:

```
    ConnectorClient connector = new ConnectorClient(new Uri(activity.
    ServiceUrl));
   var heroCard = new HeroCard
   {
       Title = "Sample Hero Card",
       Subtitle = "Smart Cards – Learning Bot Framework",
       Text = "The Hero card is a multipurpose card; it primarily hosts a
       single large image, a button, and a tap action.",
       Images = new List<CardImage> { new CardImage("https://
       sec.ch9.ms/ch9/7ff5/e07cfef0-aa3b-40bb-9baa-7c9ef8ff7ff5/
       buildreactionbotframework_960.jpg") },
       Buttons = new List<CardAction> { new CardAction(ActionTypes.OpenUrl,
       "Get Started", value: "https://docs.botframework.com/en-us/") }
   };

   Activity reply = activity.CreateReply();
   Attachment attach = new Attachment();

   attach = heroCard.ToAttachment();
   reply.Attachments = new List<Attachment> { attach };

   await connector.Conversations.ReplyToActivityAsync(reply);
   var response = Request.CreateResponse(HttpStatusCode.OK);
return response;
```

Thumbnail Card

The Thumbnail card is a multipurpose card; it primarily hosts a single small image, a button, and a "tap action," along with text content to display on the card. The difference between the Hero and Thumbnail cards is the size of image displayed. Thumbnail has smaller images compared to the Hero card.

The properties used to configure the Thumbnail card are listed in Table 5-4.

Table 5-4. *Properties Supported by Thumbnail Card*

Property	Description
`Title`	Title of card
`Subtitle`	Link for the title
`Text`	Text of the card
`Images[]`	Multiple image is supported
`Buttons[]`	Support one or more buttons
`Tap`	An action to take when tapping on the card

A sample output of a Thumbnail card is shown in Figure 5-10 in an emulator.

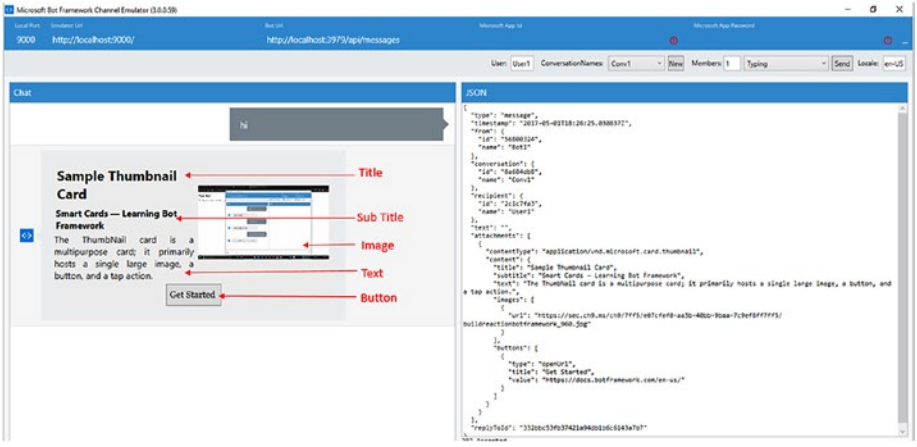

Figure 5-10. *Sample depicting Thumbnail card in Bot Emulator*

The accompanying code has the complete project for a Thumbnail card available:

```
ConnectorClient connector = new ConnectorClient(new Uri(activity.
ServiceUrl));
                var thumbNail = new ThumbnailCard
                {
```

```
            Title = "Sample Thumbnail Card",
            Subtitle = "Smart Cards – Learning Bot Framework",
            Text = "The ThumbNail card is a multipurpose card; it
            primarily hosts a single large image, a button, and a
            tap action.",
            Images = new List<CardImage> { new CardImage("https://
            sec.ch9.ms/ch9/7ff5/e07cfef0-aa3b-40bb-9baa-
            7c9ef8ff7ff5/buildreactionbotframework_960.jpg") },
            Buttons = new List<CardAction> { new
            CardAction(ActionTypes.OpenUrl, "Get Started", value:
            "https://docs.botframework.com/en-us/") }
        };

        Activity reply = activity.CreateReply();
        Attachment attach = new Attachment();

        attach = thumbNail.ToAttachment();
        reply.Attachments = new List<Attachment> { attach };

        await connector.Conversations.ReplyToActivityAsync(reply);

         var response = Request.CreateResponse(HttpStatusCode.OK);
    return response;
```

Carousal

A Carousal has multiple cards or images that display in turn. A Carousel is composed of other rich media cards, like Hero card, images, Thumbnail cards, and so on. Users can navigate the individual cards in Carousal and perform actions on them.

A sample output of a Carousel in Bot Emulator is shown in Figure 5-11.

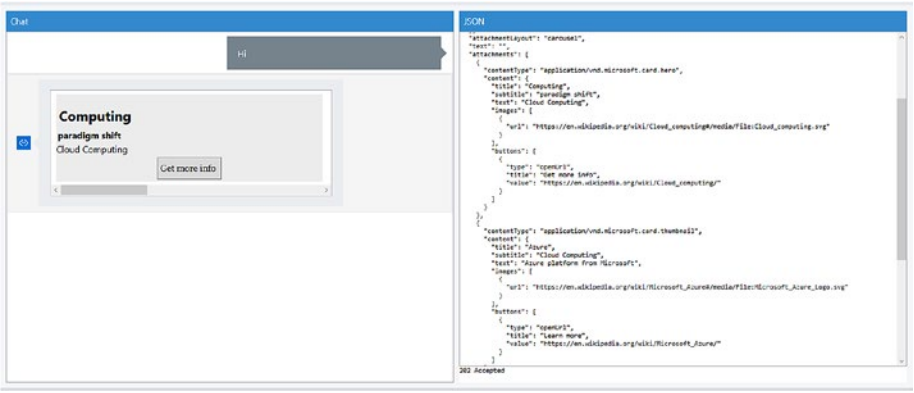

Figure 5-11. *Sample depicting using Carousel in Bot Emulator*

The accompanying code has the complete project for a Carousel available:

```
ConnectorClient connector = new ConnectorClient(new Uri(activity.
ServiceUrl));

                Activity reply = activity.CreateReply();
                reply.AttachmentLayout = AttachmentLayoutTypes.Carousel;
                reply.Attachments = new List<Attachment>()
                {
                    new HeroCard
                    {
                        Title = "Computing",
                        Subtitle = "paradigm shift",
                        Text = "Cloud Computing",
                        Images = new List<CardImage>() {new CardImage(url:
                        "https://en.wikipedia.org/wiki/Cloud_computing#/
                        media/File:Cloud_computing.svg") },
                        Buttons = new List<CardAction>() { new
                        CardAction(ActionTypes.OpenUrl, "Get more info",
                        value: "https://en.wikipedia.org/wiki/Cloud_
                        computing/") },
                    }.ToAttachment(),
                    new ThumbnailCard
                    {
                        Title = "Azure",
                        Subtitle = "Cloud Computing",
                        Text = "Azure platform from Microsoft",
                        Images = new List<CardImage>() { new CardImage(url:
                        "https://en.wikipedia.org/wiki/Microsoft_Azure#/
                        media/File:Microsoft_Azure_Logo.svg") },
                        Buttons = new List<CardAction>() { new
                        CardAction(ActionTypes.OpenUrl, "Learn more", value:
                        "https://en.wikipedia.org/wiki/Microsoft_Azure/") },
                    }.ToAttachment(),
                    new HeroCard
                    {
                        Title = "Chat Bot",
                        Subtitle = "Bot Framework",
                        Text = "Bot framework from Microsoft",
                        Images = new List<CardImage>() {new CardImage(url:
                        "https://en.wikipedia.org/wiki/Chatbot#/media/
                        File:Telegram_tourism_chatbot.png") },
                        Buttons = new List<CardAction>() { new
                        CardAction(ActionTypes.OpenUrl, "Get Details",
                        value: "https://en.wikipedia.org/wiki/Chatbot") },
                    }.ToAttachment(),
            };

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

Buttons

Buttons in media cards are the primary mechanism through which users interact with the bot. MS Bot framework provides the flexibility to change the button action depending on the requirements. Buttons can open a URL in a browser, post back the selection to the bot, download a file, or call a phone number. Most of the examples use the `openUrl` action for the buttons. For sending a response back to bot to take further action, the `imBack` or `postback` actions should be used. Table 5-5 provides the entire list of properties supported by this object.

Table 5-5. *Propeties Supported by Button Object*

CardAction.Type	CardAction.Value
`openUrl`	URL to be opened in the built-in browser
`imBack`	Text of the message to send to the bot (from the user who clicked the button or tapped the card). This message (from user to bot) will be visible to all conversation participants via the client application that is hosting the conversation.
`postBack`	Text of the message to send to the bot (from the user who clicked the button or tapped the card). Some client applications may display this text in the message feed, where it will be visible to all conversation participants.
`call`	Destination for a phone call in this format: **tel:123123123123**
`playAudio`	URL of audio to be played
`playVideo`	URL of video to be played
`showImage`	URL of image to be displayed
`downloadFile`	URL of file to be downloaded
`signin`	URL of OAuth flow to be initiated

Prompts

A prompt is a mechanism to get a quick answer from a user. It is a mechanism to encourage users to provide information. MS Bot framework provides prompts for this purpose. Prompts in MS Bot framework are implemented as dialogs, which is the main subject of the next chapter. However, they also help form rich conversation with users, and so it made sense to include prompts in this chapter. Users are encouraged to revisit this section along with the chapter on dialogs.

Prompts provide various options for asking for different types of input from users. These include the following:

1. Choice – Users can click on an option to select and submit it.
2. Confirm – Users are provided with Yes and No buttons, and they can click on a button to select and submit their response.
3. Text – Users are prompted to enter text content.
4. Number – Users are encouraged to enter numerical content.

When a prompt is used, it asks users to provide an answer using buttons or raw text. In both cases, MS Bot framework makes it easy for developers to accept the data and pass it to the parent dialog. This helps the parent dialog take appropriate action based on the user input. Prompts also provide a mechanism to ask users the same question multiple times if the answer does not satisfy the bot.

Prompt dialogs are complete dialogs by themselves and have their own lifecycle. They also encapsulate the logic to prompt questions, display buttons, and accept inputs, then process and pass them to the parent dialog.

A Choice prompt needs multiple items that can be iterated using the `IEnumerable` and `IEnumerator` interfaces. Enums and lists provide an easy way to group multiple items together for this purpose. Custom collections can also be used for this. An enum named Title is shown in the code example. This enum is used to generate the button that the user can select as part of their interaction with the bot.

```
    public enum title
    {
        Mr,
        Mrs,
        others
    }
PromptDialog.Choice(
                context: context,
                resume: NameFromUserMethod,
                options: Enum.GetValues(typeof(title)).Cast<title>().ToArray(),
                prompt: "How would you like to be addressed ? (Mr, Mrs or
                others):",
                retry: "I was clueless about this input!  Please try again!!");
```

The output of running code listing is shown in Figure 5-12.



Figure 5-12. *Example demonstrating using PromptDialog to accept user input*

A Confirm prompt displays Boolean true/False, Yes/no buttons, and users can select one of them as part of their selection. `_salvation` and `_name` are global class-level variables.

```
PromptDialog.Confirm(
                context: context,
                resume: GetCityFromUserMethod,
                prompt: $"You entered {_salvation} {_name} !! Click yes to
                confirm!!",
               retry: "I was clueless about this input!  Please try again!!");
```

The output of running this code is shown in Figure 5-13.



Figure 5-13. *Example demonstrating using PromptDialog to confirm user input*

A text prompt, as the name suggests, prompts users to provide content as text:

```
PromptDialog.Text(
                context: context,
                resume: CityFromUserMethod,
                prompt: "Please provide your full name:",
               retry: "I was clueless about this input!  Please try again!!");
```

Number prompts are used to provide numbers as content.

The complete code for understanding prompts is available as accompanying code for this chapter. The sample code is implemented as a dialog and prompts users to select their salutation through choices, prompts users to provide their full name through a text prompt, and asks them to confirm their salutation and full name through a confirm prompt.

Figure 5-14 shows an interaction with this bot using Bot Emulator to show prompts being used.

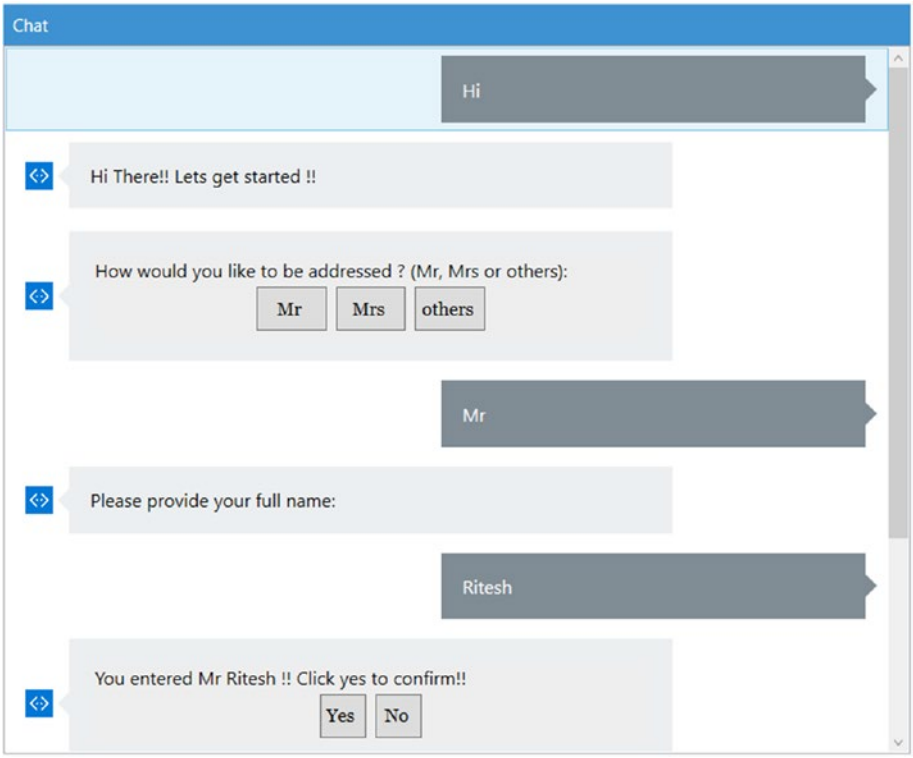

Figure 5-14. *Example showing usage of prompts for accepting input from user*

The following code is for the prompt example:

```
public enum title
    {
        Mr,
        Mrs,
        others
    }

    [Serializable]
    public class ParentDialog : IDialog<object>
    {
        private title _salvation;
        private string _name;
```

```
public async Task StartAsync(IDialogContext context)
{
    await context.PostAsync("Hi There!! Lets get started !!");
    context.Wait(GetStarted);
}

public virtual async Task GetStarted(IDialogContext context,
IAwaitable<object> result)
{
    var input = await result;
    PromptDialog.Choice(
        context: context,
        resume: NameFromUserMethod,
       options: Enum.GetValues(typeof(title)).Cast<title>().ToArray(),
        prompt: "How would you like to be addressed ? (Mr, Mrs or
        others):",
       retry: "I was clueless about this input!  Please try again!!");
}

public virtual async Task NameFromUserMethod(IDialogContext context,
IAwaitable<title> result)
{
    _salvation = await result;
    PromptDialog.Text(
        context: context,
        resume: CityFromUserMethod,
        prompt: "Please provide your full name:",
       retry: "I was clueless about this input!  Please try again!!");
}

public virtual async Task CityFromUserMethod(IDialogContext context,
IAwaitable<string> result)
{
    string _name = await result;
    PromptDialog.Confirm(
        context: context,
        resume: GetCityFromUserMethod,
        prompt: $"You entered {_salvation} {_name} !! Click yes to
        confirm!!",
       retry: "I was clueless about this input!  Please try again!!");
}

public virtual async Task GetCityFromUserMethod(IDialogContext
context, IAwaitable<bool> result)
{
    bool register = await result;
    if (register)
    {
```

```
                await context.PostAsync("Thanks for providing your details
                about your name !! ");
                await context.PostAsync("bye bye !! you may close this
                conversation !! ");
            }
            else
            {
                await context.PostAsync("please submit any key to get
                started !! ");
                context.Wait(GetStarted);
            }

        }
    }
```

Summary

Conversations are the core of writing bots. Conversations as a platform is the new phenomenon that is gaining traction, and bots are becoming the new face of organizations instead of traditional web pages. Bots are based on conversations. Conversations that are intuitive and intelligent are a must for any bot in order to keep users engaged. MS Bot framework provides numerous features to make conversations user friendly and intuitive. It provides facilities for sending and receiving attachments, images, and files. It also helps in providing advanced user-interface elements like buttons, Carousels, Hero cards, Thumbnail cards, and many others to ensure that bot authors can write engaging bots.

CHAPTER 6

Skype Calling Bot

The advent of smartphones is a revolution, and with internet-based voice calling (VOIP) available on all mobile phones, audio/video calling has become the cheapest, fastest, and easiest method of communication. Businesses should adapt and evolve as per the latest advances in technology and its application keeping in view the competition, customer's usage trends and expectations. Many businesses today have a mobile application in addition to their website, like Flipkart, Zomato, Uber, and so on. Today, businesses are embracing the next wave of revolution in the fields of application interface or communication, which is virtual assistants. Soon, you will see all businesses being operated by virtual assistants, accompanied by processes running top-notch, highly scalable artificial intelligence platforms. We have been learning to build these virtual assistants, which can communicate with the user using text, image, or touch inputs. Bots can also use speech/voice for conversations.

Bots use conversation as a medium to interact with the user (*Bot* here is referred to as an agent or interface between the user and the business, just like a website or a mobile application), so why not let the user and bot talk to each other instead of typing into the chat window? Imagine a user trying to log a service complaint; instead of typing in the whole description, the user can speak into the microphone, describing the issue. The bot, on the other hand, can respond by speaking to the user after interpreting the user's voice. The user here is communicating with a bot in real-time with an impression of human-to-human conversation.

Microsoft Bot framework provides the Skype Calling SDK, which can be used to build bots with calling features. Skype Calling SDK allows developers to design bot APIs with calling, DTMF input, intent recognition, and text input by using MS Bot framework–supported languages like C# and Node.js. Skype calling is available only on the Skype channel and is an opt-in feature. Skype calling bots are the new generation of intelligent bots that reshape the user experience by allowing them to communicate with applications using voice from any device, in any language, with ease. Bots can also play pre-recorded audio to the user; for example, a welcome message or waiting tone. Speaking to a bot is one side of the communication, but how do you make the bot understand and respond? The inputs from a user can be simple Yes/No or choosing from a given set of options, but for it to understand speech the Bot should be able to perform textual analytics. Microsoft Cognitive Services provides an intelligent speech-recognition API called Bing Speech API with which developers can build bots that can convert speech to text and text to speech and perform voice recognition. In this chapter, we will learn to build a next-generation, smart, callable IVR bot that can convert speech to text. The code samples in this chapter are built using C#, but the concepts apply to Node.js as well.

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_6

The following topics will be discussed in this chapter:

- Introducing Skype calling bots
- Use cases for building callable bots
- Enabling calling for your bot
- Building a Skype calling bot using C# and VS
- Subscribing and handling events in Skype calling bot
- Debugging Skype calling bot using Ngrok
- Speech-to-text conversion using Bing Speech API

Introducing Skype Calling Bots

The steps that help create a Skype calling bot are very much like those for any typical bot. The most important aspect you need to understand while designing a Skype calling bot is the sequence of events. Skype calling bot conversations are initiated by a caller by calling a bot contact from the user's contact list. The call is translated to an HTTP REST call to our bot application by the Skype channel. The translation also embodies the user's input, like the DTMF option (option selected from dial pad), voice as stream of bytes, or anything else. The Skype Bot API contains two callable endpoints, `callback` and `call`, and the Skype channel ensures it calls the correct endpoint as per the lifecycle. For example, when the bot-calling conversation is initiated by the caller, the `call` endpoint is invoked first followed by the `callback`—this handling of lifecycle events is taken care of by the framework, making development easy.

■ **Note** Any bot application can also be invoked by using the MS Bot framework's Direct Line API.

For the Skype calling option to be activated, it should be enabled on the Developer Portal. This is usually done when you register the bot in the portal, and this is the only means by which you can test your bot before publishing it to the wider audience.

Use Cases for Skype Calling Bots

The following are few use cases where Skype calling bots can be built:

- A typical use case for designing Skype calling bots is to automate the customer-service system. In any customer-service system, the user connects to a customer agent who is responsible for solving the customer's query, but over a period the queries and the responses tend to follow a pattern, leaving the agent repeating the same response again and again. For example, if the customer's query is related to performing a hard reset for your Smart TV, the steps to do so are same for any customer.

- A Skype bot can also act as an intelligent listener in a group conversation. Let us say you want to have a listener in your regular group calls who can summarize the contents of the call; it can be automated by building a Skype bot and adding it to the group call. Skype bots can be integrated with Cognitive Services to perform speech-to-text conversion, key-phrase extraction, sentiment analysis, and topic and language detection. It can also do the job of language translation in text format for the participants on the call who speak different dialects.
- A Skype calling bot can act as an authentication platform when combined with Bing Speech API's voice-recognition feature. For example, you can use Skype calling in two-factor authentications where the bot provides an authentication code after recognizing the user's voice.
- A Skype calling bot can also help in speech analysis. For example, we can build a bot that converts speech to text and performs sentiment analysis, biased/unbiased categorization, and so on.

Enabling Calling for Your Bot

A Skype calling bot's registration process is essentially no different from the steps we saw earlier, although there is one additional mandatory step to enable Skype calling for your bot.

Follow these steps to enable Skype calling for your bot:

- Log in to the Developer Portal at `https://dev.botframework.com` and navigate to the My Bots section.
- Select the bot for which you intend to enable Skype calling
- Click on the Edit  icon on the Skype channel.
- Click on the Calling tab.
- Select "Enable Calling" and "IVR – 1:1 IVR audio calls" and click Save (Figure 6-1).

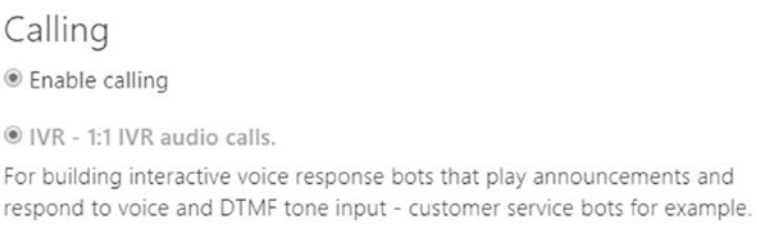

Figure 6-1. *Enable Skype calling feature in Developer Portal*

- To disable, select "Disable Calling" and click Save.

Building a Skype Calling Bot

There are many scenarios where you might want to provide a rich user experience by building voice-based interaction. In this section, we will build a Skype IVR bot that registers a customer's complaints related to products made by an imaginary organization, Contoso. IVR stands for *Interactive Voice Response*. In a typical IVR-based system, the computer reads out a predefined set of options to the user and seeks information from the user in the form of keyboard input or voice, which is converted into commands or actions. It is also possible to generate dynamic menus in an IVR system. In this example, we will stick to a simple IVR bot with a fixed menu that seeks information from the user using keyboard-based inputs. IVR-based bots can be helpful in building a guided conversation that resembles a workflow.

■ **Note** For this example, let us assume Contoso is an organization that sells electronics like smart TVs, laptops, refrigerators, and washing machines to its consumers. Any such organization is usually backed up by a customer-service team that is reachable by a toll-free number. In this chapter, we will learn to build scenarios around automating the job of a customer-service agent.

Let us get started with building our Contoso bot by following these steps:

- Open Visual Studio and create a new project using Bot Template, and let us call it `ContosoIVRBot`.
- The Skype Calling API is available as an extension to the Bot Builder SDK called `Microsoft.Bot.Builder.Calling`. The extension should be installed on the project using the NuGet Installer. Right-click on the project references and then click on Manage NuGet Packages. Search and install the package named `Microsoft.Bot.Builder.Calling` as shown in Figure 6-2.

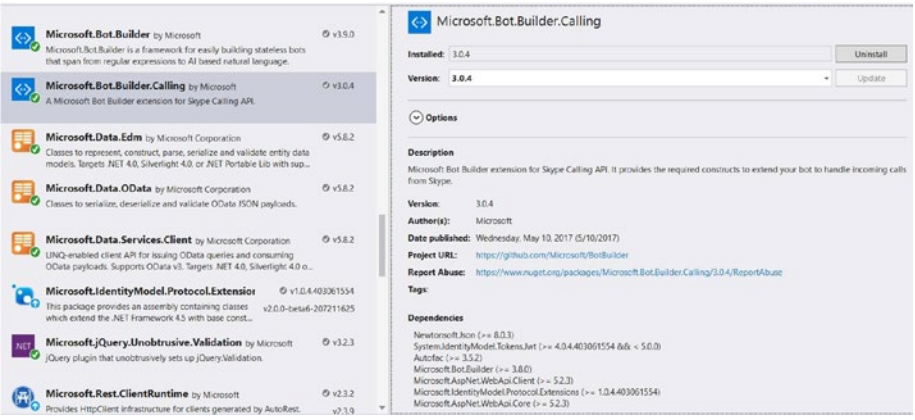

Figure 6-2. *Installing Bot Builder Calling SDK using NuGet installer*

- Alternatively, you can also run the following command in the NuGet package console:

```
Install-Package Microsoft.Bot.Builder.Calling -Version 3.0.4
```

- The default template provides a controller that is used for text conversation, which we have been using in the previous examples. The Skype Calling controller has a different workflow. To build a Skype calling bot, we should create a controller that can accept incoming calls and call back. For this purpose, let us add a new controller to the ContosoIVR project under the `Controllers` folder and call it `CallingController`. Although both `MessageController` and `CallingController` extend from the same base class `ApiController`, the lifecycle of the calling controller is different from that of a message controller, as the message controller responds to the post messages from the user while the calling controller is controlled by a series of events. Copy the following contents into the `CallingController`:

```
[BotAuthentication]
[RoutePrefix("api/calling")]
public class CallingController : ApiController
{
    public CallingController() : base()
    {
        CallingConversation.RegisterCallingBot(callingBotService
        => new IVRBot(callingBotService));
    }

    [Route("callback")]
    public async Task<HttpResponseMessage>
    ProcessCallingEventAsync()
    {
        return await CallingConversation.SendAsync(this.Request,
        CallRequestType.CallingEvent);
    }

    [Route("call")]
    public async Task<HttpResponseMessage>
    ProcessIncomingCallAsync()
    {
        return await CallingConversation.SendAsync(this.Request,
        CallRequestType.IncomingCall);
    }
}
```

Add the following `using` statements to the `CallingController` to refer the required libraries:

```
using Microsoft.Bot.Builder.Calling;
using Microsoft.Bot.Connector;
```

A few important things to notice in the preceding `CallingController` are the `Class` decoration and attributes. The route prefix attribute allows you to define a route to reach your bot's callable endpoint, in this case `api/calling`. This is a very important step, as the route defined here is also used for debugging by enabling a few settings at the Developer Portal, which we will learn later. Hence, if there is change in the route, the bot's registration should be updated accordingly.

The `CallingConversation` class is at the root of the series of events. It allows you to register a service that can handle the events that will be raised during a Skype call conversation. In this example, we are using a class called `IVRBot`, which hosts our event handlers. If the calling bot is not registered, the application would compile successfully but you would not be able to receive any calls.

The following route handles the incoming call and invokes the `OnIncomingCallReceived` event on our registered bot service (`IVRBot`, in this case):

```
        [Route("call")]
        public async Task<HttpResponseMessage> ProcessIncomingCallAsync()
        {
            return await CallingConversation.SendAsync(this.Request,
            CallRequestType.IncomingCall);
}
```

The following `callback` event is a process for any subsequent callbacks from the user. Depending on the state of the conversation, the appropriate event handler on the calling bot service is invoked.

```
       [Route("callback")]
        public async Task<HttpResponseMessage> ProcessCallingEventAsync()
        {
            return await CallingConversation.SendAsync(this.Request,
            CallRequestType.CallingEvent);
 }
```

The following code shows the skeleton of the `IVRBot` service that registers the event handlers for our callable bot:

```
/// <summary>
/// IVR Bot class
/// </summary>
/// <seealso cref="System.IDisposable" />
/// <seealso cref="Microsoft.Bot.Builder.Calling.ICallingBot" />
public class IVRBot : IDisposable, ICallingBot
{
```

```
/// <summary>
       /// Initializes a new instance of the <see cref="IVRBot"/> class.
       /// </summary>
       /// <param name="callingBotService">The calling bot service.</param>
       public IVRBot(ICallingBotService callingBotService)
       {
           if (callingBotService == null)
           {
               throw new ArgumentNullException(nameof(callingBotService));
           }

           this.CallingBotService = callingBotService;

           // Registering Call events
           this.CallingBotService.OnIncomingCallReceived +=
           OnIncomingCallReceived;
           this.CallingBotService.OnPlayPromptCompleted +=
           OnPlayPromptCompleted;
           this.CallingBotService.OnRecordCompleted +=
           OnRecordCompletedAsync;
           this.CallingBotService.OnRecognizeCompleted +=
           OnRecognizeCompleted;
           this.CallingBotService.OnHangupCompleted += OnHangupCompleted;
       }

       /// <summary>
       /// Gets the calling bot service.
       /// </summary>
       /// <value>
       /// The calling bot service.
       /// </value>
       public ICallingBotService CallingBotService
       {
           get; private set;
       }
}
```

The important piece of code you need to notice is the interface `ICallingBot` that is implemented by the `IVRBot`. The root registration class `CallingConversation` expects an object of a class that implements `ICallingBot`; hence, the interface:

```
namespace Microsoft.Bot.Builder.Calling
{
    //
    // Summary:
    //     The calling bot interface.
    public interface ICallingBot
    {
```

```
        ICallingBotService CallingBotService { get; }
    }
}
```

`ICallingBot` hosts the instance of `CallingBotService` (as just shown), which contains the events that are fired during a Skype call conversation. The interface containing the events raised during a Skype call conversation is shown in Figure 6-3.

```
namespace Microsoft.Bot.Builder.Calling
{
    public interface ICallingBotService
    {
        event Func<IncomingCallEvent, Task> OnIncomingCallReceived;
        event Func<AnswerOutcomeEvent, Task> OnAnswerCompleted;
        event Func<HangupOutcomeEvent, Task> OnHangupCompleted;
        event Func<PlayPromptOutcomeEvent, Task> OnPlayPromptCompleted;
        event Func<RecognizeOutcomeEvent, Task> OnRecognizeCompleted;
        event Func<RecordOutcomeEvent, Task> OnRecordCompleted;
        event Func<RejectOutcomeEvent, Task> OnRejectCompleted;
        event Func<WorkflowValidationOutcomeEvent, Task> OnWorkflowValidationFailed;

        string ProcessCallback(string content, Task<Stream> additionalData);
        Task<string> ProcessCallbackAsync(string content, Task<Stream> additionalData);
        string ProcessIncomingCall(string content);
        Task<string> ProcessIncomingCallAsync(string content);
    }
}
```

Figure 6-3. *Events of calling bot interface*

Sequence of Events

It is important to understand the sequence of the events when designing a free flow or guided conversation in a Skype calling bot. Every action from the user is converted into an HTTPS REST–based call to our endpoint, and the bot should reply with workable actions like accepting the DTMF inputs or voice from the user. The list of actions sent to the user to execute is called the *workflow*. If the bot fails to send a proper action to the user, the call will be disconnected/dropped, which means there is no way the bot can communicate back to the user. The Skype channel does not provide any details of the caller, like Skype username or email ID; Skype only sends a unique identifier for each participant in the conversation. The number of participants on the call can be more than one (excluding the bot). The Skype channel respects the user's privacy and security guidelines by not exposing the Skype caller name or user information.

Figure 6-4 explains a typical scenario in a voice-based conversation with a bot.

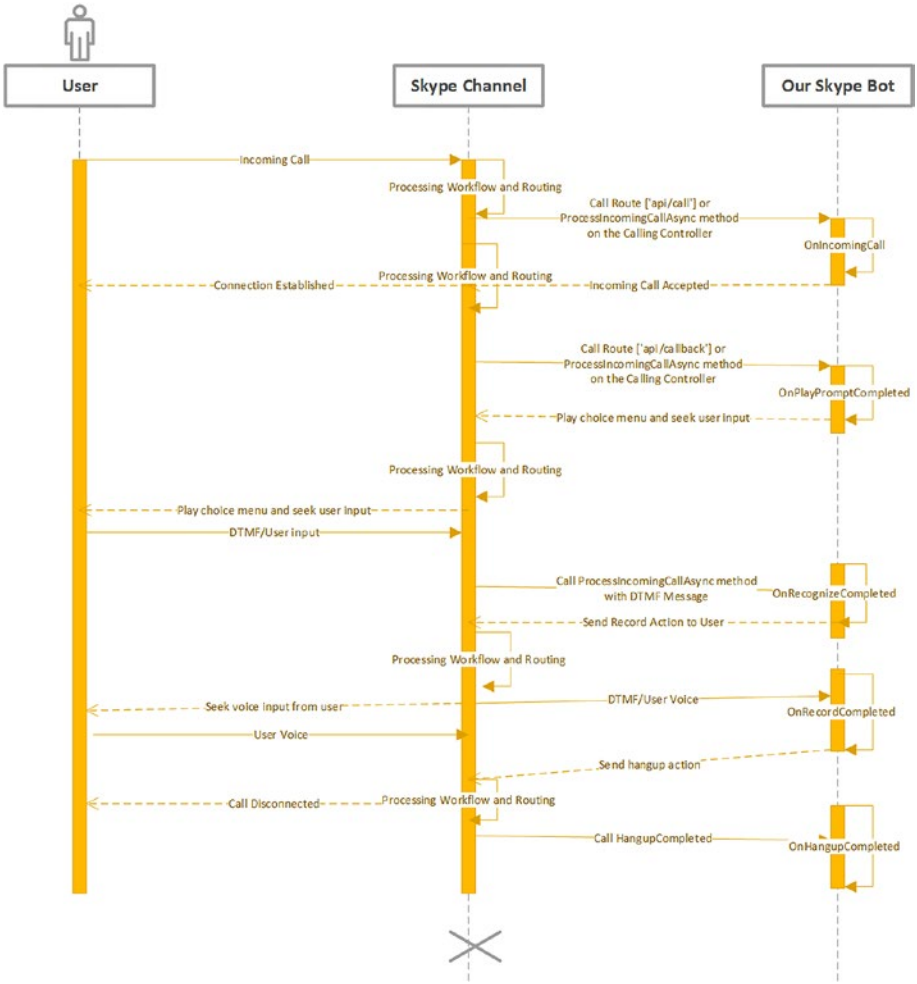

Figure 6-4. *Sequence of events in a callable bot application*

Table 6-1 lists the critical events in a `CallingBotService` that should be handled by the bot.

Table 6-1. *CallingBotService Events*

Event	Description
`OnIncomingCallReceived`	This event is raised when the bot receives a call. The reply action could be answer/reject.
`OnAnswerCompleted`	This event is raised when caller is acknowledged by the answer action. This also means that the connection is established for any further communication. The bot should respond with an action or list of actions in every call.
`OnRejectCompleted`	This event is raised when bot rejects the call, and the reject action is acknowledged by the user. In this case, there is no need to respond with any workflow action.
`OnPlayPromptCompleted`	This event is raised when a play prompt action is completed. The arguments of this event contain the outcome of the play prompt.
`OnRecognizeCompleted`	This event is raised when bot gets the outcome of a recognize option (like a DTMF response). The arguments of this event contain the DTMF response status and the actual outcome.
`OnRecordCompleted`	This event is raised when the bot gets the outcome of the record action. The arguments of this action contain the content stream.
`OnHangupCompleted`	This event is raised when bot gets the outcome of the hangup action, and this also means that the connection is no longer available.
`OnWorkflowValidationFailed`	This event is raised when any workflow is failed by the bot platform during validation.

In addition to the preceding events, there are a few more design aspects that should be considered before authoring a bot. Bots are stateless, but the bot conversations should be state-ful; every action should be saved to provide a rich and personalized conversation experience for the user. There are several ways to do this. You can save the state to a temporary storage, like a dictionary, and push it to more persistent storage, like DocumentDB, Azure Storage, or SQL Database. In this example, I'm using a dictionary, which saves the user's state and pushes the state to Azure table storage at the end of the conversation. See here:

```
/// <summary>
/// The call state map
```

```
/// </summary>
private IDictionary<string, CallState> callStateMap = new Dictionary<string,
CallState>();
```

Bot applications are multi-user environments. There could be multiple users interacting with a bot at the same time; hence, there are multiple user states while the application is running. To identify or differentiate between two user states, we should have a unique identifier. The bot channel assigns a default identifier to each conversation that can be used to isolate the user state. The value of the identifier is passed to every event handler during the conversation and can be used to retrieve the previous state from storage, like the language option chosen by the user, user preferences, or a chosen menu option.

Having reviewed the core functionality of each event, let us now build our Skype calling bot. As a first step, we should accept the call from the user and play a welcome message, which can be done in the `OnIncomingCallReceived` event. Copy the following code to the `OnIncomingCallReceived` event handler. As explained earlier, `incomingCallEvent.IncomingCall.Id` is the unique identifier for the user.

```
this.callStateMap[incomingCallEvent.IncomingCall.Id] = new
CallState(incomingCallEvent.IncomingCall.Participants);
            telemetryClient.TrackTrace($"IncomingCallReceived -
            {incomingCallEvent.IncomingCall.Id}");
            incomingCallEvent.ResultingWorkflow.Actions = new List<ActionBase>
            {
                new Answer { OperationId = Guid.NewGuid().ToString() },
                GetPromptForText(IVROptions.WelcomeMessage)
            };
            return Task.FromResult(true);
```

The method `GetPromptForText` converts any given text to an audio prompt. The bot calling service provides the text-to-speech conversion out of the box. You also have the option to customize the voiceover, like the gender of the speaker and culture. You can also play a recorded audio instead of text-to-speech. The `Prompt` class provides various configurable options, like emphasize any given word, customize pronunciation, or configure a delay between the speech.

```
private PlayPrompt GetPromptForText(string text)
        {
            var prompt = new Prompt { Value = text, Voice = VoiceGender.
            Female };
            return new PlayPrompt { OperationId = Guid.NewGuid().ToString(),
            Prompts = new List<Prompt> { prompt } };
        }
```

Figure 6-5 shows the options for configuring the prompts played to the user.



Figure 6-5. *Properties of Prompt class*

Once the prompt is played to the user, the bot channel invokes the next event: `OnPlayPromptCompleted`.

■ **Note** You can also register the `OnAnswerCompletedEvent`. This event is helpful when you want to establish a connection to the persistent storage and load the previous state. In this case, I'm using the `OnPlayPromptCompletedEvent` to send a recognize workflow action to the user.

In this method, you can set up the next workflow action for the user; for example, choosing from a list of options.

```
private Task OnPlayPromptCompleted(PlayPromptOutcomeEvent
playPromptOutcomeEvent)
        {
            // Add Choices for the caller
            telemetryClient.TrackTrace($"PlayPromptCompleted -
            {playPromptOutcomeEvent.ConversationResult.Id}");
            Recognize recognize = SetupInitialMenu();
            playPromptOutcomeEvent.ResultingWorkflow.Actions = new
            List<ActionBase> { recognize };
            return Task.FromResult(true);
        }
```

The `recognize` action lets the user choose an option from the list of key–value pairs. Once the key matches the user's input, the value is sent as an argument to the next event handler. The following code plays the prompt "*If your complaint is regarding Mobile, press 1, for TV, press 2, for Refrigerator, press 3*" to the user and waits for user's input until the timeout value assigned to the property `InitialSilenceTimeoutInSeconds` runs out.

```
private Recognize SetupInitialMenu()
        {
            var callerChoices = new List<RecognitionOption>()
            {
                new RecognitionOption() { Name = "1", DtmfVariation = '1' },
                new RecognitionOption() { Name = "2", DtmfVariation = '2'},
                new RecognitionOption() { Name = "3", DtmfVariation = '3'},
                new RecognitionOption() { Name = "#", DtmfVariation = '#'}
                // for navigating back
            };

            // create recognize action for caller
            var recognize = new Recognize
            {
                OperationId = Guid.NewGuid().ToString(),
                PlayPrompt = GetPromptForText(IVROptions.MainMenuPrompt),
                BargeInAllowed = true,
                InitialSilenceTimeoutInSeconds = 10,
                Choices = callerChoices
            };
            return recognize;
        }
```

■ **Note** `GetPromptForText` is used throughout the example; it converts any given text into a voice prompt. All the text messages are stored as constants in a static class called `IVROptions`.

Notice that the voice prompt has the exact same number of options as in our recognizable options. If the options are not in sync with the prompt played, the outcome will be a failure, or you might end up taking a wrong action. The option `BargeInAllowed` is set to true to allow users to select an option before the prompt completes. If you intend to capture a sequence of numbers from the user, like an Order ID or phone number, you can use the `CollectDigits` property in the `recognize` class. Remember: You can either ask the user to enter a single digit as input or a sequence of digits; you cannot use both. Every `recognize` action is assigned a unique operation Id, which can be used to identify or correlate the outcome during the validation and response event. The bot calling service allows you to send multiple actions to the user. The `OperationId` will also be helpful when identifying an action in the response body when multiple actions are specified.

In response to the above workflow action, the user should select a number from the dial pad, which is available in the list of recognizable options. Once the user makes a selection, an `OnRecognizeCompleted` event is raised. The outcome of the event could be success or failure. In case of failure, you can either replay the previous prompt to the user

and ask them to retry or prompt an error message and hang up. In this case, I'm playing an unsupported option prompt and replaying the initial menu:

```
private Task OnRecognizeCompleted(RecognizeOutcomeEvent
recognizeOutcomeEvent)
        {
            if (recognizeOutcomeEvent.RecognizeOutcome.Outcome != Outcome.
            Success)
            {
                telemetryClient.TrackTrace($"RecognizeFailed -
                {recognizeOutcomeEvent.ConversationResult.Id}");
                var unsupported = GetPromptForText(IVROptions.
                OptionMenuNotSupportedMessage);
                var recognize = SetupInitialMenu();
                recognizeOutcomeEvent.ResultingWorkflow.Actions = new
                List<ActionBase> { unsupported, recognize };
                return Task.FromResult(true);
            }

// outcome success
telemetryClient.TrackTrace($"RecognizeCompleted - {recognizeOutcomeEvent.
ConversationResult.Id}");
            var prompt = GetPromptForText(IVROptions.RecordMessage);
            var record = new Record
            {
                OperationId = Guid.NewGuid().ToString(),
                PlayPrompt = prompt,
                MaxDurationInSeconds = 60,
                InitialSilenceTimeoutInSeconds = 5,
                MaxSilenceTimeoutInSeconds = 2,
                RecordingFormat = RecordingFormat.Wav,
                PlayBeep = true,
                StopTones = new List<char> { '#' }
            };
            this.callStateMap[recognizeOutcomeEvent.ConversationResult.
            Id].ChosenMenuOption = recognizeOutcomeEvent.RecognizeOutcome.
            ChoiceOutcome.ChoiceName.ToString();
            this.callStateMap[recognizeOutcomeEvent.ConversationResult.Id].
            Id = recognizeOutcomeEvent.ConversationResult.Id;
            recognizeOutcomeEvent.ResultingWorkflow.Actions = new
            List<ActionBase> { record };
            return Task.FromResult(true);
        }
```

In the event of success, the record workflow is set up for the user. The user is asked to voice his message, which will be captured by the bot. According to the above configuration, the maximum possible duration of a message is 60 seconds, and the user has 5 seconds after the play beep to start recording or else it will result in a

timeout. The user can press the # key to end the recording; you can use any other key as an indicator to end the recording, but the # key is the recommended and conventional approach. The recording format can be set to WAV, MP3, or WMA. As part of this event handler, I'm also capturing the user's choice so that I can use it in the next stages if required.

■ **Note** Since this is sample code, I'm using an in-memory dictionary to store the state. This is not recommended in a production scenario because you might lose state if the application crashes. You must design it to sync the state to a persistent store more often.

The user now records his message after the beep and presses # on the dial pad to end the recording. As a result, an event `OnRecordCompletedAsync` is raised. The event's arguments contain the recorded voice in the form of Stream, which can be saved to any persistent storage like Azure blob storage. You can extend the conversation to any extent in this way and update the state as the conversation progresses; to end the conversation at any point you should raise a `Hangup` action as shown here:

```
private async Task<bool> OnRecordCompletedAsync(RecordOutcomeEvent
recordOutcomeEvent)
        {
            telemetryClient.TrackTrace($"RecordCompleted -
            {recordOutcomeEvent.ConversationResult.Id}");
            recordOutcomeEvent.ResultingWorkflow.Actions = new List<ActionBase>
                {
                    GetPromptForText(IVROptions.Ending),
                    new Hangup { OperationId = Guid.NewGuid().ToString() }
                };

            // Message from User as stream of bytes format
            var recordedContent = recordOutcomeEvent.RecordedContent.Result;
            this.callStateMap[recordOutcomeEvent.ConversationResult.Id].
            RecordedContent = recordedContent;

            // save call state
            var azureStorgeContext = new AzureStorageContext();
            await azureStorgeContext.SaveCallStateAsync(this.
            callStateMap[recordOutcomeEvent.ConversationResult.Id]);

            recordOutcomeEvent.ResultingWorkflow.Links = null;
            this.callStateMap.Remove(recordOutcomeEvent.ConversationResult.Id);
            return await Task.FromResult(true);
        }
```

In this example, the recorded content is saved to Azure blob storage, and the call state along with the blob's URL are saved to Azure table storage. Unlike web or mobile applications, bot speech conversations cannot be reinitiated unless designed to do so;

you must ensure during your design phase that you are allowing the user to retry in case of a failed outcome using the event arguments. See here:

```
if (recordOutcomeEvent.RecordOutcome.Outcome != Outcome.Success)
            {
                // write code to Hangup or Ask user to repeat
                return await Task.FromResult(false);
            }
```

The following event handler is invoked after the call is disconnected. This should be the point where you dispose of any open connections to resources like storage or save any pending state.

```
private Task OnHangupCompleted(HangupOutcomeEvent hangupOutcomeEvent)
        {
            telemetryClient.TrackTrace($"HangupCompleted -
            {hangupOutcomeEvent.ConversationResult.Id}");
            hangupOutcomeEvent.ResultingWorkflow = null;
            return Task.FromResult(true);
        }
```

And, finally, you unregister the event handlers in the `Dispose` method:

```
public void Dispose()
        {
            if (CallingBotService != null)
            {
                CallingBotService.OnIncomingCallReceived
                -= OnIncomingCallReceived;
                CallingBotService.OnPlayPromptCompleted
                -= OnPlayPromptCompleted;
                CallingBotService.OnRecordCompleted
                -= OnRecordCompletedAsync;
                CallingBotService.OnRecognizeCompleted
                -= OnRecognizeCompleted;
                CallingBotService.OnHangupCompleted -= OnHangupCompleted;
            }
        }
```

The other pieces of the code, like logging events to Application Insights and Azure blob/table storage, follow the best practices in application design. You can download the code from GitHub to understand how telemetry works. Now that we have a fully functional Skype calling bot, let us learn how to debug and test it.

Debugging Skype Calling Locally Using Ngrok

When we developed chat-based bots in previous chapters, we learned how to debug using Bot Emulator. The same approach isn't possible with Skype calling bots because calling is only available on the Skype channel, which means the only way you can test the bot is by calling from Skype (besides the Direct Line API). Debugging your code on the development machine is an important step, as it is critical to know how your user experiences the workflow and how well your bot can capture the user's voice. It creates a serious hit on the branding of your bot if anything goes wrong after it is published. Direct Line API is a complex approach to test calling bots, but there is a simple yet convincing method of debugging a Skype calling bot that does not involve Direct Line API. You can call from your Skype application on mobile/PC through to your local PC's Visual Studio instance using Ngrok. Ngrok is a tiny executable that can be used to create an HTTPS or HTTP (or SSH) tunnel to any port on your localhost, which means any calls on a public internet-facing endpoint can be routed to the local port by running Ngrok on your local machine. Let us learn how to debug a Skype calling bot. You can download the Ngrok executable from `https://ngrok.com/`.

Follow these steps to set up debugging for a Skype calling bot in Visual Studio:

1. Register the bot at the Bot Developer Portal: `https://dev.botframework.com`.
2. Unzip Ngrok to any folder on your local PC and run the following command:

   ```
   ngrok.exe http 3979 -host-header="localhost:3979"
   ```

   (Assuming you're running the application on port 3979)
3. Ngrok should show up online and provide a forwarding IP address mapping on HTTP/HTTPS endpoints, as shown in Figure 6-6.

```
ngrok by @inconshreveable

Session Status                online
Version                       2.2.8
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://c5a82e95.ngrok.io -> localhost:3979
Forwarding                    https://c5a82e95.ngrok.io -> localhost:3979

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Figure 6-6. *Ngrok TunnelCopy the HTTPS endpoint (`https://c5a82e95.ngrok.io`) in this case*

4. In the Bot Developer Portal for your bot, click on the "Skype edit" option on the Skype channel and enable the options shown in Figure 6-7.

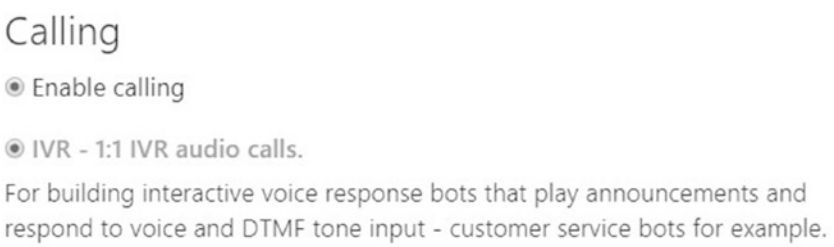

Figure 6-7. *Enabling Skype calling*

5. Configure the `Webhook` endpoint as shown in Figure 6-8.

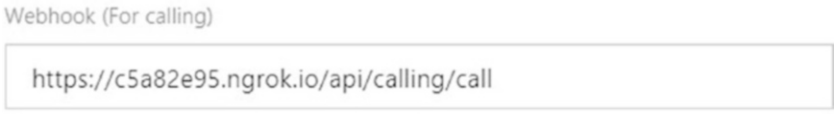

Figure 6-8. *Webhook configuration for debugging on a local development environment*

6. `https://{ngrokendpoint}}/api/calling/call` (ensure HTTPS endpoint is used). For example,
7. Update the `CallbackUrl` endpoint on the `web.config` with the `ngrok` endpoint, as shown below.

```
<add key="Microsoft.Bot.Builder.Calling.CallbackUrl"
value="https://c5a82e95.ngrok.io/api/calling/callback" />
```

Remember that the call back on the Developer Portal ends with call, and the call back on the bot's `web.config` ends with callback. That's all you need to set up debugging. You can now press F5 in Visual Studio to configure break points and debug your Skype bot. Before you do any of that, ensure you have added the bot to your Skype contacts; this can be done from the Bot Developer Portal. After the contact is added, you should also see the calling option enabled on Skype.

■ **Note** It takes few minutes for the configuration to work.

When you make a call, the Skype Bot channel receives the call and forwards it to the webhook. The webhook running our machine forwards the call to the application running on our localhost, and that's how we get to debug our application. Figure 6-9 shows the series of HTTPS calls made on the webhook that are forwarded to Visual Studio.

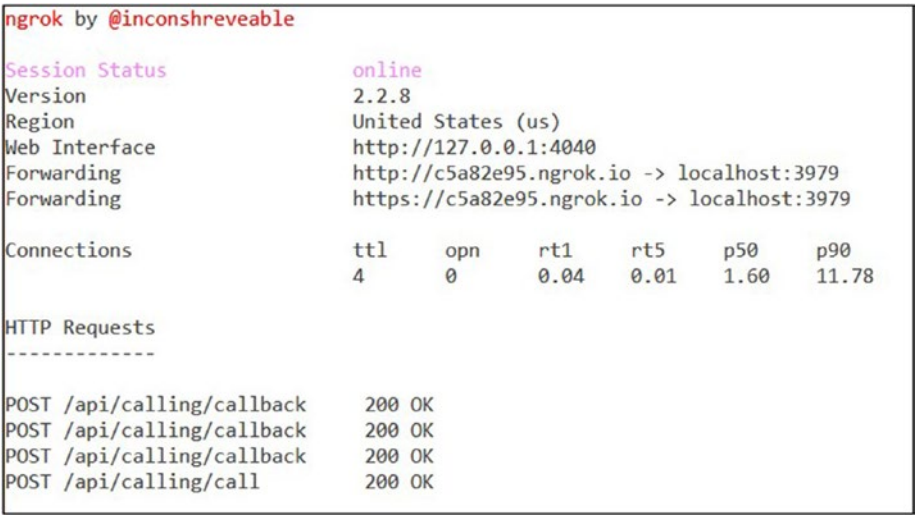

Figure 6-9. *HTTP requests for Skype calling with Ngrok configuration*

Figure 6-10 shows the breakpoint hit when the first request is made by the Skype caller.

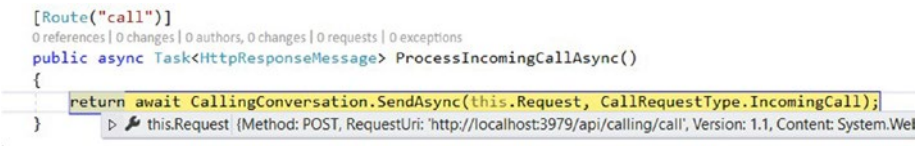

Figure 6-10. *Debugging Skype calling bots using Ngrok configuration*

Speech-to-Text Using Bing Speech API

Skype calling is an amazing feature. It takes very little code to set up a calling bot that allows your callers to record their voice in any language or culture from any type of device. But in most of the cases, a recorded voice is not useful as-is. If you want to perform any analysis—like s search operation, language conversion, or to find out the speaker's intent by applying text analytics—it should converted to text format. In its plain form, the audio file should be downloaded and played back anytime you want to analyze. Speech-to-text conversion was not something achievable with a few lines of code before Microsoft launched Cognitive Services. With Cognitive Services, you just need an API

key and a few lines of code to extract text from any type of audio file. In this section, we will learn to extend our Contoso IVR bot by converting the audio recording to text and building a simple Power BI dashboard that shows trends in customer complaints.

The Bing Speech API is one of the many services under Microsoft Cognitive Services with which you can build smart applications with capabilities like speech recognition and speech-to-text from an audio stream in real-time or from an audio file. The Bing Speech API also helps you build applications that can talk back to your users using text-to-speech—this is an out-of-the-box feature in Microsoft Bot framework. The Bing Speech API can be consumed using the REST API or C# SDK; in this example we will use the C# SDK. The API can also be used when building bots with Node.js.

Bing Speech API's billing is calculated in terms of transactions per month; with each pricing tier there is a limit on the number of calls per second (Figure 6-11).

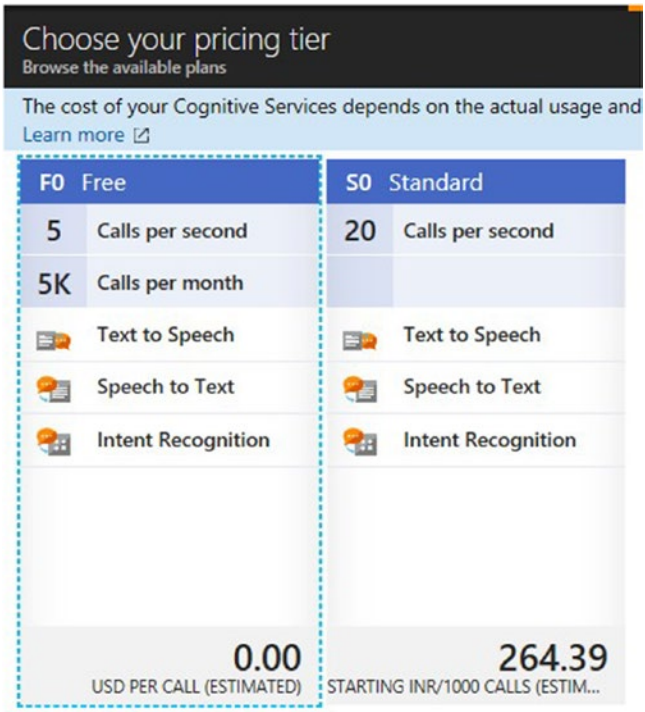

Figure 6-11. *Bing Speech API pricing options*

To get started we need to obtain an API key, which you can get by creating a Bing Speech API account in the Azure Subscription by navigating to `https://ms.portal.azure.com/#create/Microsoft.CognitiveServicesBingSpeech`. See Figure 6-12.

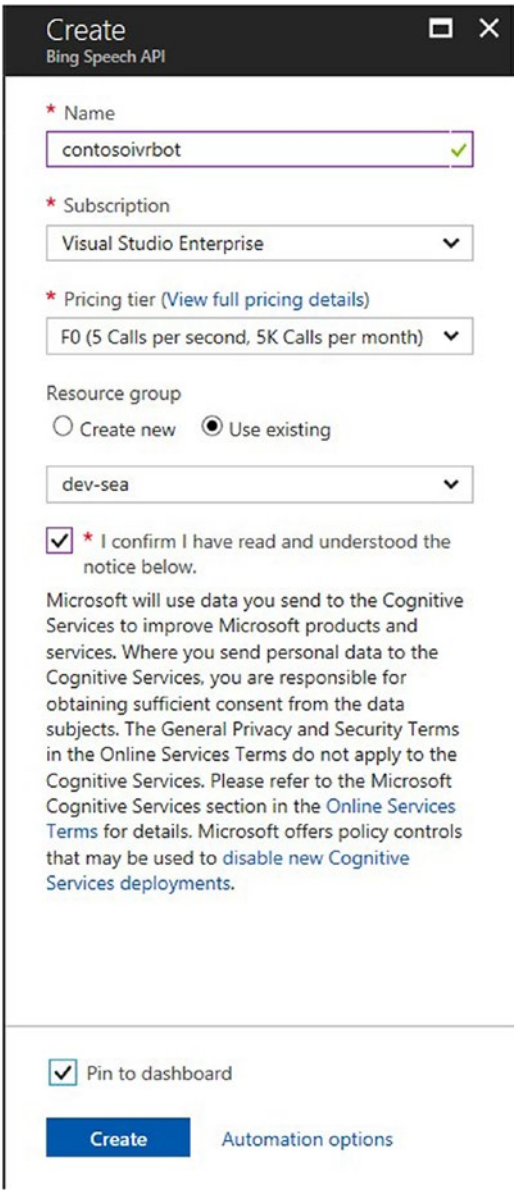

Figure 6-12. *Azure portal blade showing Bing Speech API account creation*

You can also get a 15-day trial key by adding Bing Speech API (Figure 6-13) to your subscription here: https://azure.microsoft.com/en-us/try/cognitive-services/my-apis/.

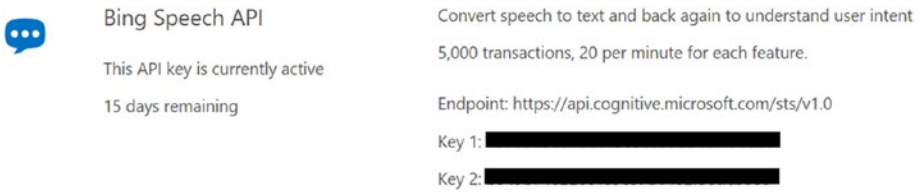

Figure 6-13. *Bing Speech API keys*

In our example, we will use the `Microsoft.ProjectOxford.SpeechRecognition` NuGet package to interact with the Bing Speech API (Figure 6-14). Right-click on the references and add the following packages to the project.



Figure 6-14. *Bing Speech API NuGet packages*

Add the following code to the `OnRecordCompletedAsync` event handler after the recorded content is available:

```
BingSpeech bs = new BingSpeech(
                async t => {
                    // save call state
                    this.callStateMap[conversationId].SpeechToTextPhrase = t;
                    await azureStorgeContext.SaveCallStateAsync(this.
                    callStateMap[recordOutcomeEvent.ConversationResult.Id]);
                    this.callStateMap.Remove(conversationId);
                },
                t => telemetryClient.TrackTrace("Bing Speech to Text failed"));
            bs.CreateDataRecoClient();
            bs.SendAudioHelper(recordedContent);
```

`BingSpeech` is a custom class. The constructor of `BingSpeech` accepts two call backs, one for success and one for failure, which will be invoked appropriately after the speech-to-text conversion. To convert speech to text, we should create a connection to the Bing Speech API (https://api.cognitive.microsoft.com/sts/v1.0) and subscribe

to the events that are invoked by SDK. The following functions take care of creating the connection by reading the API key from the `web.config`:

```
public void CreateDataRecoClient()
        {
            this.SubscriptionKey = ConfigurationManager.AppSettings["Microso
            ftSpeechApiKey"].ToString();
           this.dataClient = SpeechRecognitionServiceFactory.CreateDataClient(
                SpeechRecognitionMode.ShortPhrase,
                this.DefaultLocale,// for example: 'en-us'
                this.SubscriptionKey);
            this.dataClient.OnResponseReceived += this.
            OnResponseReceivedHandler;
            this.dataClient.OnConversationError += this.OnConversationError;
        }
```

Speech Recognition mode supports two options:

- Short Phrase mode: Supports an utterance up to 15 seconds long
- Long Diction mode: Supports an utterance up to 2 minutes long

The response from the API can be subscribed to using two types of events: `OnResponseReceived` and `OnPartialResponseReceived`. The first event is raised when the API finishes the speech-to-text conversion completely; the second event, `OnPartialResponseReceived`, is raised when some part of the conversion is available. `OnPartialResponseReceived` is more suitable for long dictations. For long dictations, `OnResponseReceived` is raised multiple times based on the sentence pauses detected by the server. The event `OnConversationError` is invoked when the server detects an error during conversion.

The following method sends the audio stream to the server:

```
public void SendAudioHelper(Stream recordedStream)
        {

            int bytesRead = 0;
            byte[] buffer = new byte[1024];
            try
            {
                do
                {
                    // Get more Audio data to send into byte buffer.
                   bytesRead = recordedStream.Read(buffer, 0, buffer.Length);

                    // Send audio data to service.
                    this.dataClient.SendAudio(buffer, bytesRead);
                }
                while (bytesRead > 0);
            }
```

```
            catch (Exception ex)
            {
                Debug.WriteLine("Exception ------------ " + ex.Message);
            }
            finally
            {
                // We are done sending audio. Final recognition results will
                arrive in OnResponseReceived event call.
                this.dataClient.EndAudio();
            }
        }
```

The following method is invoked when a response is available, which in turn calls the registered callback:

```
private void OnResponseReceivedHandler(object sender,
SpeechResponseEventArgs e)
        {
            StringBuilder phraseResponse = new StringBuilder();
            for (int i = 0; i < e.PhraseResponse.Results.Length; i++)
            {
                phraseResponse.AppendLine(string.Format("{0} ",
                e.PhraseResponse.Results[i].DisplayText));
            }
            this._callback.Invoke(phraseResponse.ToString());
        }
```

`PhraseResponse` contains a list of phrases with a confidence level assigned to each phrase, as shown in Figure 6-15. Confidence level helps in making a guided decision on the converted text.

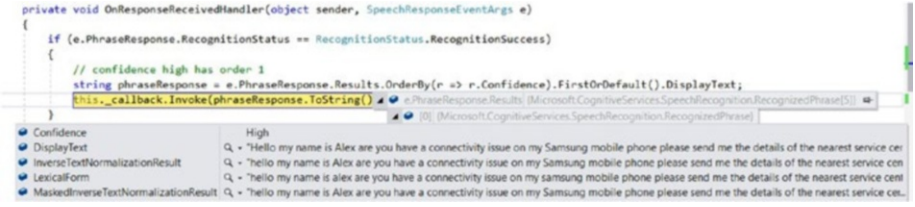

Figure 6-15. *Response from Bing Speech-to-Text API request*

The following piece of code is invoked, with the text converted from the speech:

```
 async t =>
{
// save call state
this.callStateMap[conversationId].SpeechToTextPhrase = t;
```

```
await azureStorgeContext.SaveCallStateAsync(this.
callStateMap[recordOutcomeEvent.ConversationResult.Id]);
this.callStateMap.Remove(conversationId);
}
```

The application stores the converted text in an Azure Storage account; the storage account connection string is available in the `web.config`. Refer to https://docs.microsoft.com/en-us/azure/storage/storage-create-storage-account to create a storage account. You can see the speech-to-text converted data when connected to the Azure Storage account configured in the `web.config` (you can connect using Visual Studio Cloud Explorer or download Azure Storage Explorer). Figure 6-16 shows a table with text extracted from speech.

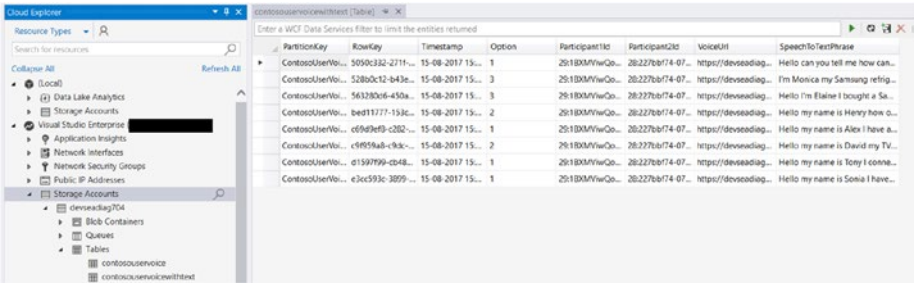

Figure 6-16. *Visual Studio Azure Storage Explorer*

What if you want to play back the text to the user? In a few cases, you might like to have the user confirm if the message has been correctly converted to text. This too is possible by creating a proactive message and sending it through by direct conversation. Adding the following code after the STT response is received does the job.

```
var url = "https://skype.botframework.com";

var userAccount = new ChannelAccount(caller.Identity, caller.DisplayName);

var botAccount = new ChannelAccount(callee.Identity, callee.DisplayName);

// authentication
var account = new MicrosoftAppCredentials(ConfigurationManager.
AppSettings["MicrosoftAppId"].ToString(),
    ConfigurationManager.AppSettings["MicrosoftAppPassword"].ToString());
var connector = new ConnectorClient(new Uri(url), account);

MicrosoftAppCredentials.TrustServiceUrl(url, DateTime.Now.AddDays(7));
```

```
var conversation = await connector.Conversations.CreateDirectConversation
Async(botAccount, userAccount);
IMessageActivity message = Activity.CreateMessageActivity();
message.From = botAccount;
message.Recipient = userAccount;
message.Conversation = new ConversationAccount(id: conversation.Id);
message.Text = string.Format("The following complaint has been registered
thanks for calling: {0}, Complaint Id: {1}", t, conversationId);
message.Locale = "en-us";
await connector.Conversations.SendToConversationAsync((Activity)message);
```

The above code creates a direct connection to the user by using the participant IDs sent by Bot Connector. The message back and the service URL should be signed before sending the message to the connector. This is done to let our bot account trust the service URL, and so the outgoing call is authenticated. Remember: we are not sending the message directly to the user; instead, it is sent to the Bot Connector, which then sends the message to the user, as shown in Figure 6-17.

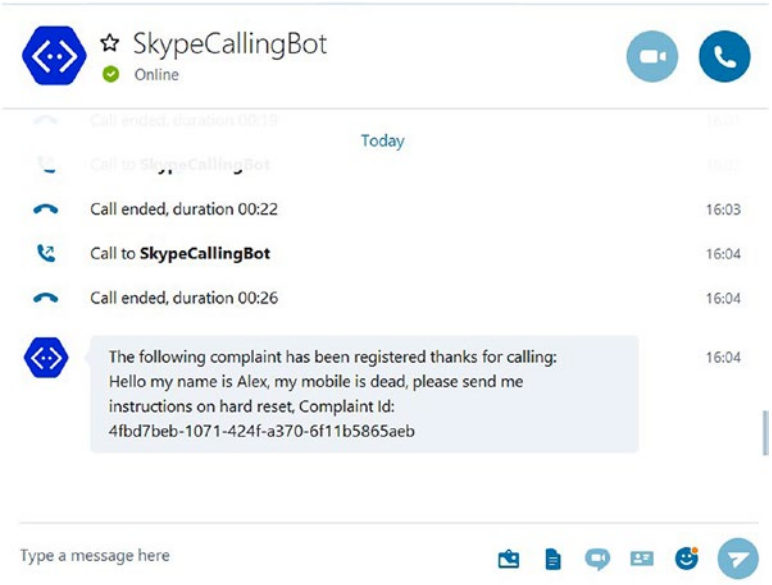

Figure 6-17. *Successful conversation with Skype calling bot from Skype on Windows 10*

As per the current design, our bot ends the call before the preceding message is passed to the user. Instead, you can wait for the user to confirm or edit the complaint before you hang up. The complaint ID is a unique identifier for the complaint that the caller can use to reply back.

The information is stored in the Azure Storage account and can be used for business analytics. The sample Power BI report in Figure 6-18 (also available in the sample code) helps you analyze the trends and complaints by category and search through the text that is captured in the storage account.

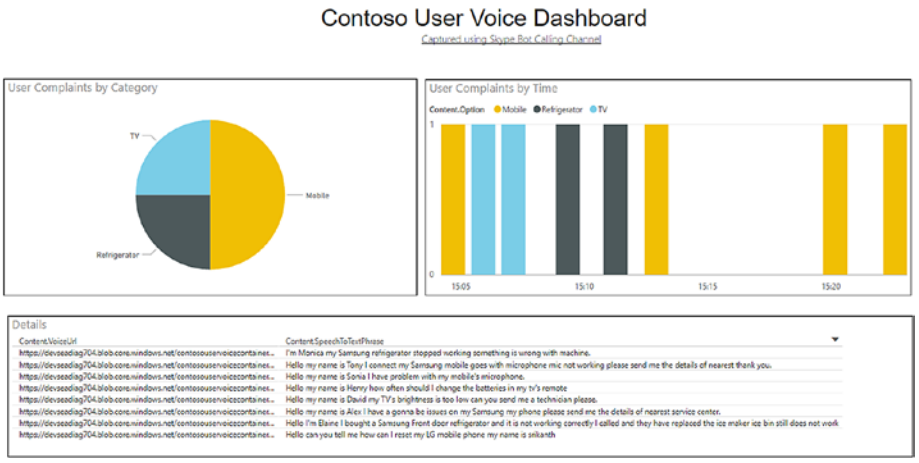

Figure 6-18. *Power BI dashboard with live status of calls received by Contoso in various categories*

Azure gives you a plethora of options with which to channel your data into the right medium to bring in more intelligence, which can then help drive your business in a better and more profitable way. It could be as simple as creating a Power BI report, making your data searchable using Azure Search, maintaining better customer relationships by sending notifications using Azure Mobile Push Notifications, and so on. The preceding example can be extended by sending the converted text to LUIS for intent recognition. With Bing Speech and bots, you can build intelligent bots that capture feedback in a sophisticated fashion and make your business reachable by a larger audience.

Summary

Microsoft Bot framework provides a Skype Calling SDK that can be used to build bots with calling features. The Skype Calling SDK allows developers to design bot APIs with calling and voice support, DTMF input, intent recognition, and text input by using Bot framework–supported languages like C# and Node.js. Skype calling is available only on the Skype channel and is an opt-in feature. The Skype calling feature should be enabled from the Bot Developer Portal. The Skype Calling API is available as a REST API and

NuGet package called `Microsoft.Bot.Builder.Calling`. The Skype Calling SDK contains built-in support for converting text to speech or playing any audio file to the user.

The response from the user can be as simple as selecting a menu option by using the dial pad or their voice. Microsoft Cognitive Services provides an intelligent speech-recognition API called Bing Speech API that developers can use to build bots that can convert speech to text and text to speech as well as perform voice recognition. The `Microsoft.ProjectOxford.SpeechRecognition` NuGet package is used to interact with Bing Speech API; it can also be interacted with via REST API. Bing Speech API's billing is calculated in terms of transactions per month.

It is important to understand the sequence of the events in order to design a free flow or guided conversation in a Skype calling bot. Every action from the user is converted into an HTTPS REST–based call to our bot endpoint, and the bot should reply with workable actions, like accepting DTMF inputs or voice from the user. The list of actions sent to the user to execute is called the workflow. If the bot fails to send a workflow, the call is dropped. The call can be intentionally dropped by sending the hang-up workflow.

You can debug Skype calling bots by initiating a call from your Skype application on mobile/PC through to your local PC's Visual Studio instance using the ngrok executable. Bing Speech API supports two types of voice-recognition modes: short phrase and long dictation. Short Phrase supports an utterance up to 15 seconds long while long dictation mode supports up to 2 minutes. The extracted `PhraseResponse` contains a list of phrases with a confidence level assigned to each phrase—High, Normal, and Low. The extracted text from speech can be sent back to the user in text form by using proactive messaging.

CHAPTER 7

Storing State

State management is one of the more complex subjects in software computing related to services on the internet. The internet uses HTTP protocol by default, which is stateless in nature. It means that a request to one URI on the internet using HTTP protocol is different than a subsequent request. The changes in the first request are not reflected in next request. All the requests are independent of each other, and subsequent requests do not have knowledge about the previous requests. This is the nature of internet protocols; however, to write rich applications and services that can remember what the previous request did, multiple frameworks, patterns, and platforms have been developed. The purpose of these frameworks and patterns is to provide state management either on the client side or the server side—or on both—and thus save a developer from undergoing the pain of writing their own state-management routines.

There are two types of services implemented on the internet—stateful and stateless.

Stateful refers to a programming paradigm in which the services actively hold the values of variables between multiple requests. The values stored in the variables are available in subsequent requests.

Stateless is also a programming paradigm in which the services do not hold and manage values in variables between multiple requests. All necessary values needed to execute the services are sent by the client along with every request.

There are pros and cons of both stateful and stateless services. However, stateless services have an edge over stateful services since they are highly scalable and consume fewer resources. They are scalable because they do not need to take care of recreating the current state of a request in times of disaster; they can simply be moved and deployed to another server without any dependency on the current state. Also, not managing a state means a lesser dependence on volatile memory.

To overcome these challenges, stateful services have an option to store the state in a separate process or server instead of in-memory. This helps the service to move around on servers without losing the current state, and it can also connect to a previously stored state easily. However, this arrangement has its own cost. Generally, managing state in different processes or servers has performance penalties that are not always desirable.

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_7

MS Bot framework provides a centralized mechanism to manage state for bots. By default, bots do not have to worry about managing their state. Bots can be implemented as stateless services, and Bot framework will ensure that whatever state is needed by a bot is available to it anytime and every time. This chapter focuses on state management in bots and will discuss in detail the following topics:

- Types of state management available in MS Bot framework
- Introduce and interact with state service
- State management in bots with dialogs
- Storing custom state information
- Storing state in Cosmos DB (formerly known as DocumentDB) and table storage

Stores for Bot State

The MS Bot framework provides three stores that you can use to store state, as follows:

- User data state
- Conversation data state
- Private conversation state

It is to be noted that every channel provides a separate state store. There could be a User data state specific to the Facebook channel and another related to the Slack or Skype channels.

The User data store should be used to store user-specific data on a channel. Data that relates to a user, like name, age, email, and so on should be stored in this data store. This data will remain the same irrespective of the number of conversations the user has with a bot. PII data can be stored in the User data store.

The Conversation data store is used to store data related to a conversation on a channel but is not specific to any user. This data is visible to all parties involved in a conversation. User-specific PII data should not be stored in this store.

The Private Conversation data store should be used to store user-specific data on a channel specific to a conversation. PII-related user data can be stored in this data store.

It is important to note that state data, no matter the store, is available for the lifetime of a session. If the user closes the client application, like Slack or Facebook, the session is torn down and so is the state for that session. If the same user comes back on the same channel, he/she will not find the previous state available.

Another important point to remember is that both the Conversation and the Private Conversation data stores are available for the lifetime of a conversation within a session. If the user closes the conversation, the state in both stores is lost. However, data stored in the User store is not dependent on any specific conversation. User data continues to be available even if the user switches or creates a new conversation. All stores lose data when the session expires or is closed.

All three stores can store up to 32 KB of data individually for a channel. The maximum amount of data that you can store in each store for a user or conversation is 32 KB. For example, you can store 32 KB of data for User A on channel ABC, 32 KB of data for User A in a private conversation on channel ABC, and 32 KB of data for Conversation 1 on channel ABC.

Typically, you use the stores to save a user's preferences so you can tailor the conversation to them the next time you chat. For example, you can use this information to alert them about a news article that might interest them or alert them when an item of their interest is available again for buying. A hiking bot might want to save previous hike information so it can suggest new hikes. It's up to you how you use the stores.

State Service

The state stores are available to a bot as set of REST APIs. MS Bot framework makes it easier to interact with state APIs with the `StateClient` object.

The REST API for the User data store is available at `https://api.botframework.com/v3/botstate/{channelId}/users/{userId}`.

The REST API for the Conversation data store is available at `https://api.botframework.com/v3/botstate/{channelId}/conversations/{conversationId}`.

The REST API for the Private Conversation data store is available at `https://api.botframework.com/v3/botstate/{channelId}/conversations/{conversationId}/users/{userId}`.

`StateClient` provides `Get`, `Set`, and `Delete` methods that encapsulate calls to these REST APIs.

Figure 7-1 shows a snapshot of the getter properties available from the `StateClient` object.

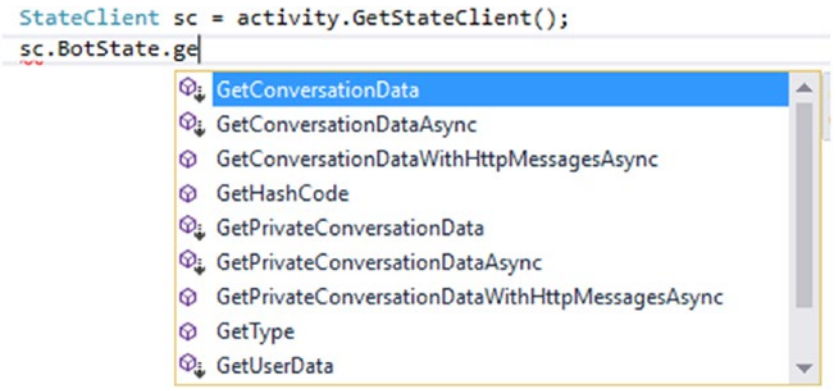

Figure 7-1. *Getter properties available from StateClient object*

Figure 7-2 shows a snapshot of the setter properties available from the `StateClient` object.

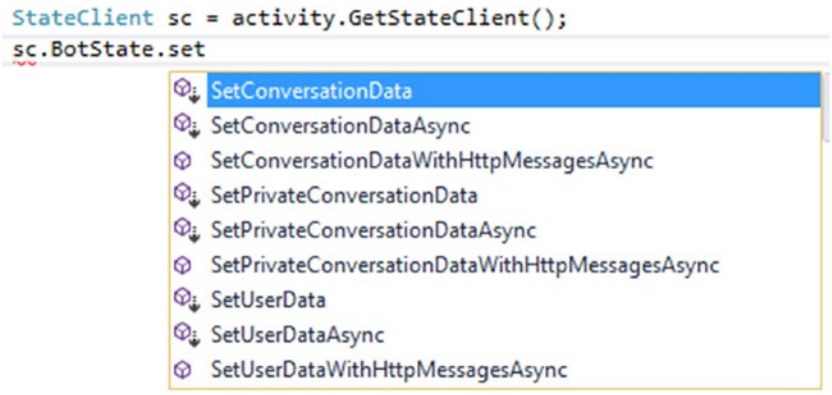

Figure 7-2. *Setter properties available from StateClient object*

Figure 7-3 shows a snapshot of delete properties that are available from the `StateClient` object.

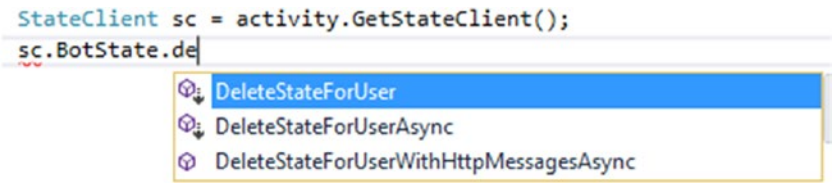

Figure 7-3. *Delete properties available from StateClient object*

There are three variants for each API. There is a synchronous method, an asynchronous method, and a method that adds custom HTTP headers before invoking the REST API. Internally, the synchronous method calls the asynchronous method which in turn calls the method that adds HTTP headers.

The bot saves state data by using one of the `Set` methods, and subsequent requests can access the same using one of the `Get` methods.

Storing and Retrieving State Using StateClient

Adding and retrieving state in a bot is quite simple. Follow these steps:

1. Get an instance of the `StateClient` object from `activity` object, which was introduced in Chapter 4. `StateClient` helps in managing state. The `GetStateClient` method gets the `StateClient` object:

   ```
   StateClient sc = activity.GetStateClient();
   ```

2. `StateClient` maintains an object `BotState` that implements the `IBotState` interface. It is this object that helps in getting, setting, and deleting state from the state store.

   To get the User data state, the code shown next should be used. Retrieval of User state needs channel ID and user ID information, which are available from `activity` object:

   ```
   BotData userData = sc.BotState.GetUserData(activity.ChannelId,
   activity.From.Id);
   ```

   To get the Conversation data state, the code shown next should be used. Retrieval of Conversation state needs channel ID and conversation ID information:

   ```
   BotData conversationData = sc.BotState.
   GetConversationData(activity.ChannelId, activity.Conversation.Id);
   ```

   To get the Private Conversation state, the code shown next should be used. Retrieval of this state needs channel ID, conversation ID, and user ID information:

   ```
   BotData privateConversationData = sc.BotState.GetPrivateConversa
   tionData(activity.ChannelId, activity.Conversation.Id, activity.
   From.Id);
   ```

3. Get state value from state store by name. In this case, properties named `myUserString`, `myConversationString`, and `myPrivateConversationString` are queried from three different state stores. It will return empty if these properties do not exist in state stores.

   ```
   myUserString = userData.GetProperty<string>("myUserString") ?? "";

   myConversationString = userData.GetProperty<string>("myConversation
   String") ?? "";

   myPrivateConversationString = userData.GetProperty<string>("myPri
   vateConversationString") ?? "";
   ```

4. The next code checks if the property is empty. If it is empty, then a new property is created in a `BotState` object and is finally added to the User data store. This will ensure that subsequent requests will get a valid property value instead of returning empty.

   To set the User data state, the code shown next should be used. Information about channel ID and user ID along with an updated `BotState` object should be provided while updating the store.

```
if (myUserString == "")
{

userData.SetProperty<string>("myUserString", "userData: " +
activity.Text + " from user data \n\n");

sc.BotState.SetUserData(activity.ChannelId, activity.From.Id,
userData);

str.Append("userData: " + activity.Text + " from user data
\n\n");
 }
        else {
str.Append( "userData: " + myUserString + " from user data
\n\n");
 }
```

   To set the Conversation data state, the code shown next should be used. Information about channel ID and conversation ID along with an updated `BotState` object should be provided while updating the store.

```
       if (myConversationString == "")
       {

userData.SetProperty<string>("myConversationString",
"conversationData: " + activity.Text + " from conversation
data  \n\n");

sc.BotState.SetConversationData(activity.ChannelId, activity.
Conversation.Id, userData);

str.Append("conversationData: " + activity.Text + " from conversation
data \n\n");
        }
```

```
        else {
              str.Append("conversationData: " +
              myConversationString + " from conversation data
              \n\n");
 }
```

To set the Private Conversation data state, the code shown next should be used. Information about channel ID, conversation ID, and user ID along with an updated `BotState` object should be provided while updating the store.

```
      if (myPrivateConversationString == "")
       {
 userData.SetProperty<string>("myPrivateConversationString",
"privateConversationString: " + activity.Text + " from private
conversation data \n\n");
sc.BotState.SetPrivateConversationData(activity.ChannelId,
activity.Conversation.Id, activity.From.Id, userData);

str.Append("privateConversationString: " + activity.Text + " from
private conversation data \n\n");
       }
       else {
str.Append("privateConversationString: " +
myPrivateConversationString + " from private conversation data
\n\n")
       }
```

5. Depending on the choice of state store or any of its combinations, the values stored within a session are available either across conversations or just within a conversation.

Figure 7-4 shows the sample bot displaying stored values from all three types of stores.

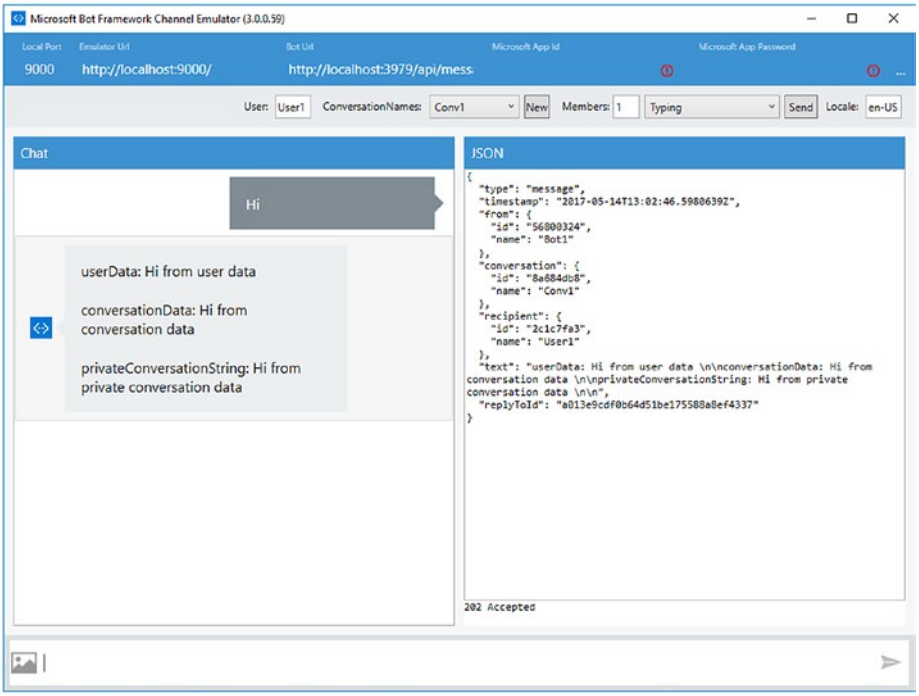

Figure 7-4. *Bot Emulator showing values from different stores in a bot application*

Storing and Retrieving State with Dialogs

Adding and retrieving state in a bot that uses dialog is also quite simple if you follow these steps:

1. As discussed briefly in Chapter 2, dialogs are an abstraction manifested as a C# class containing its own state and behavior and implementing an `IDialog` interface. A conversation can consist of multiple dialogs, and the framework maintains them using a dialog context via a stack. The dialog stack is stored in a state service provided by Bot Connector Service. Create a new project using the Bot Application template in Visual Studio, as shown in Figure 7-5.

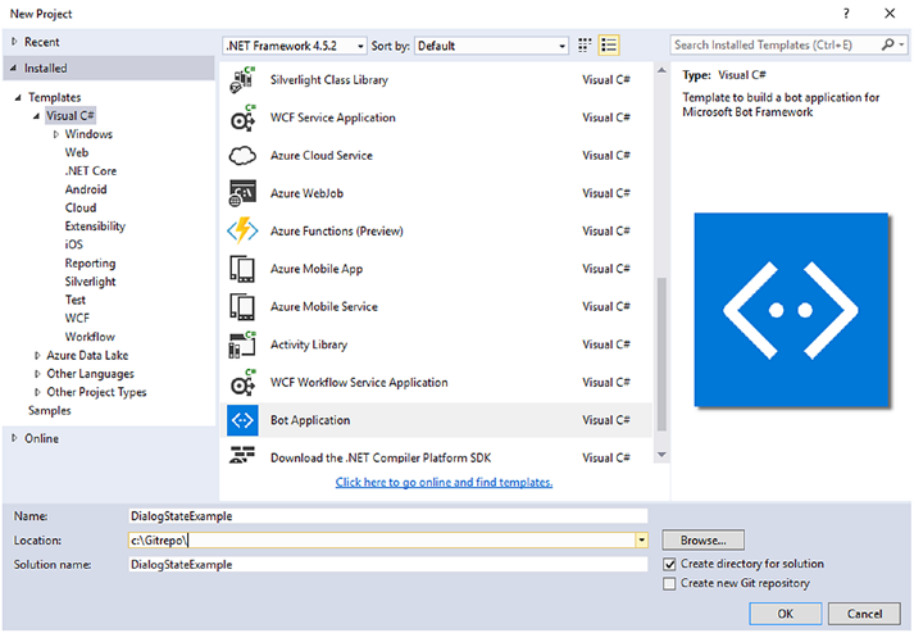

Figure 7-5. *Select Bot Application template in Visual Studio for creating a new bot*

2. Add a new class through which a dialog can be implemented. This example uses a class named `StateSampleDialog`. This class should implement the `IDialog<object>` interface. Necessary namespaces should be added to the class file for name resolution. The entry point within a dialog is through the `StartAsync` method. We will discuss dialogs in detail in the next chapter, and readers should revisit this section for a better understanding of state management in dialogs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace DialogStateExample
{
    [Serializable]
    public class StateSampleDialog :IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
```

```
        {

        }
    }
}
```

3. Dialogs should be serializable. MS Bot framework by default sterilizes the state of all dialogs and stores in the Private Conversation store before returning to the user. When a subsequent request reaches the bot, the state is retrieved from the state store and fills all public class variables with previously stored values. Two public variables are declared within the class:

```
public string userName;
public string cityName;
```

4. A set of asynchronous functions are implemented to demonstrate the default state management provided by dialogs. This dialog starts a conversation by asking the name of the user and their city. It saves both the name and city in `username` and `cityName` variables declared earlier. `StartAsync` is the starting function that wires up `NameFromUserMethod`, which then wires up `CityFromUserMethod`. The last method, `GetCityFromUserMethod`, displays the values entered by the user as part of the conversation. The values are automatically serialized and stored in the state store, then retrieved when recreating the dialog.

```
public async Task StartAsync(IDialogContext context)
        {
            await context.PostAsync("Hi There!! Lets get started
            !!");
            context.Wait(GetStarted);
        }

        public virtual async Task GetStarted(IDialogContext
        context, IAwaitable<object> result)
        {
            var name = await result;

            await context.PostAsync("Please provide your name !!");
            context.Wait(NameFromUserMethod);
        }

        public virtual async Task NameFromUserMethod(IDialogContext
        context, IAwaitable<IMessageActivity> result)
        {
            var incomingMessage = await result;
```

```
        userName = incomingMessage.Text;

        await context.PostAsync("Please provide your City
        name !!");
        context.Wait(CityFromUserMethod);
    }

    public virtual async Task CityFromUserMethod(IDialogContext
    context, IAwaitable<IMessageActivity> result)
    {
        var incomingMessage = await result;
        cityName = incomingMessage.Text;

        await context.PostAsync($"Hello {userName} !! do you
        want to know which city you entered ??");
        context.Wait(GetCityFromUserMethod);
    }

    public virtual async Task GetCityFromUserMethod(IDialogCo
    ntext context, IAwaitable<IMessageActivity> result)
    {

        var answer = await result;

        await context.PostAsync($"Hello {userName} !! You
        entered {cityName} ??");
        context.Wait(GetStarted);
    }
```

5. Change the implementation of `MessageController` code.
Replace the next lines of code

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

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

with the following:

```
await Conversation.SendAsync(activity, () => new
StateSampleDialog());
```

6. Whenever, a request comes to the bot, it is passed to `StateSampleDialog`. Here, the dialog is resurrected from state and continues executing from where it was serialized.

More Control over State with Dialogs

In the previous example, all class-level variables were stored in a state store irrespective of their requirements. If there is a need to have finer control over the values stored in the state stores, the dialog context provides access to all three bot stores. These can be directly interacted with, and a bot can choose the values it wants to store in different types of stores based on its needs.

This section will provide the steps to follow in order to create a bot with dialog that controls all variables that are stored in state stores.

1. Create a new project using the Bot Application template in Visual Studio.

2. Add a new class through which a dialog can be implemented. This example uses a class named `StateSampleDialog`. This class should implement the `IDialog<object>` interface. Necessary namespaces should be added to the class file for name resolution. The entrypoint within a dialog is through a `StartAsync` method. We will discuss dialogs in more detail in the next chapter, and readers should revisit this section for a better understanding of state management in dialogs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace DialogStateExample
{
    [Serializable]
    public class StateSampleDialog :IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {

        }
    }
}
```

3. Dialogs should be serializable, as explained in the previous section. It is to be noted that no class-level variables are declared within this bot.

4. A set of asynchronous functions are implemented to demonstrate the custom state management using the dialog context. This dialog starts a conversation by asking the name of the user and then their city. The name of the user is stored in the User data store as the `username` property. This store is available from the `context` object:

```
context.UserData.SetValue("username", name.Text);
```

The bot then asks the user's city name. It saves the city name in the Conversation data store as the `cityname` property, and it also retrieves the username saved earlier in the User data store:

```
string userName = context.UserData.Get<string>("username");

context.ConversationData.SetValue("cityname", city.Text);
```

The bot then goes on to ask the user about his rating for the current conversation. It saves this rating in the Private Conversation data store as the `rating` property, and it also retrieves the user name and city name saved earlier in the User data and Conversation data stores respectively.

```
string userName = context.UserData.Get<string>("username");

string cityName = context.ConversationData.
Get<string>("cityname");

       context.PrivateConversationData.SetValue("rating", rating.
       Text);
```

Finally, the bot asks the user to close the conversation, and as a result it retrieves the `username` property from the User data store, `cityname` property from the Conversation data store, and `rating` from the Private Conversation data store and displays them to the user.

The code for these functions is listed here:

```
        public async Task StartAsync(IDialogContext context)
        {
            await context.PostAsync("Hi There!! Lets get started
            !!");
            context.Wait(GetStarted);
 }
public virtual async Task GetStarted(IDialogContext context,
IAwaitable<object> result)
        {
            var name = await result;
```

```
        await context.PostAsync("Please provide your name !!");
        context.Wait(NameFromUserMethod);
    }

    public virtual async Task NameFromUserMethod(IDialogContext
    context, IAwaitable<IMessageActivity> result)
    {
        var name = await result;

        context.UserData.SetValue("username", name.Text);

        await context.PostAsync("Please provide your City
        name !!");
        context.Wait(CityFromUserMethod);
    }

    public virtual async Task CityFromUserMethod(IDialogContext
    context, IAwaitable<IMessageActivity> result)
    {
        var city = await result;

        string userName = context.UserData.
        Get<string>("username");

        context.ConversationData.SetValue("cityname", city.Text);

        await context.PostAsync($"Hello {userName} !! you
        stay in {city.Text}..Provide a rating (1-10) for this
        conversation");
        context.Wait(RatingFromUserMethod);
    }

    public virtual async Task RatingFromUserMethod(IDialogCon
    text context, IAwaitable<IMessageActivity> result)
    {
        var rating = await result;

        string userName = context.UserData.
        Get<string>("username");
        string cityName = context.ConversationData.
        Get<string>("cityname");

        context.PrivateConversationData.SetValue("rating",
        rating.Text);

        await context.PostAsync($"Hello {userName} !! Enter
        'over' to know your details !!");
        context.Wait(closingConversation);
    }
```

```
public virtual async Task closingConversation(IDialogContext
context, IAwaitable<IMessageActivity> result)
{
    var rating = await result;

    string userName = context.UserData.
    Get<string>("username");
    string cityName = context.ConversationData.
    Get<string>("cityname");
    string score = context.PrivateConversationData.
    Get<string>("rating");

    await context.PostAsync($"Hello {userName} !! you stay
    in {cityName}.. and you have rated this conversation
    with a score of {score} .. Thank you!!");
    context.Wait(NameFromUserMethod);
}
```

5. Change the implementation of the `MessageController` code. Replace the next lines of code

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

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

with the following:

```
await Conversation.SendAsync(activity, () => new
StateSampleDialog());
```

6. Whenever a request comes to the bot, it is passed to `StateSampleDialog`. Here, the dialog is resurrected from state and continues executing from where it was serialized

Custom State Data Store

By default, MS Bot framework stores the state in cache using the `CachingBotDataStore` object. There are use cases in which the bot would like to store state in DocumentDB or Azure Tables to have more control over it. Bot framework is an extensible framework and allows extensions for a custom state data store. By using a custom data store, the state

is no longer written in cache but rather in the custom store as configured by the MS Bot framework. In this chapter, two examples will be described, one using DocumentDB as the state data store and the other using Table storage.

MS Bot framework provides two classes as part of the Bot Builder SDK:

1. `DocumentDBBotDataStore`
2. `TableBotDataStore`

Both the classes implement the `IBotDataStore` interface responsible for declaring methods for reading, writing, and deleting data from the state store.

Cosmos DB will be used as one of the custom bot state stores, and so a small overview on it is provided next. DocumentDB and Cosmos DB are used interchangeably in this chapter. Any reference to DocumentDB can also be read as Cosmos DB.

Overview of Cosmos DB

Azure Cosmos DB is a fully managed, globally distributed, horizontally scalable in storage and throughput, multi-model database service backed up by comprehensive SLAs. Azure Cosmos DB is the next generation of Azure DocumentDB. Cosmos DB was built from the ground up with global distribution and horizontal scale at its core—it offers turn-key global distribution across any number of Azure regions by transparently scaling and replicating your data wherever your users are. You can elastically scale throughput and storage worldwide and pay only for the throughput and storage you need. Cosmos DB guarantees single-digit millisecond latencies at the 99th percentile anywhere in the world, offers multiple well-defined consistency models to fine-tune for performance, and offers guaranteed high availability with multi-homing capabilities—all backed by industry-leading service-level agreements (SLAs).

Cosmos DB is truly schema-agnostic; it automatically indexes all the data without requiring you to deal with schema and index management. Cosmos DB is multi-model—it natively supports document, key–value, graph, and columnar data models. With Cosmos DB, you can access your data using the NoSQL APIs of your choice—DocumentDB SQL (document), MongoDB (document), Azure Table Storage (key–value), and Gremlin (graph) are all natively supported. Cosmos DB is a fully managed, enterprise-ready, and trustworthy service. All your data is fully and transparently encrypted and secure by default. Cosmos DB is ISO, FedRAMP, EU, HIPAA, and PCI compliant as well.

Cosmos DB as Custom State Data Store

The high-level steps for using a custom state data store are as follows:

1. Identify the data store. This example uses Cosmos DB as its data source.
2. Provision the data store if it does not already exist.
3. Create or open a bot project using Bot Application template.

4. Add code to wire up and use Cosmos DB as the data store provided by Bot Builder SDK.
5. Write dialogs normally without any knowledge of location and type of state store.

Here are the steps in more detail:

1. In this example, Cosmos DB is used as the state store.
2. As mentioned before, Cosmos DB is a managed Azure service. To use Cosmos DB, having a valid Azure subscription is a pre-requisite. Log in to Azure as a valid user and create a new Cosmos DB service. Provide a name, select the appropriate subscription, choose DocumentDB as the API type, and set the resource group to provision a Cosmos DB database. This is shown in Figure 7-6.

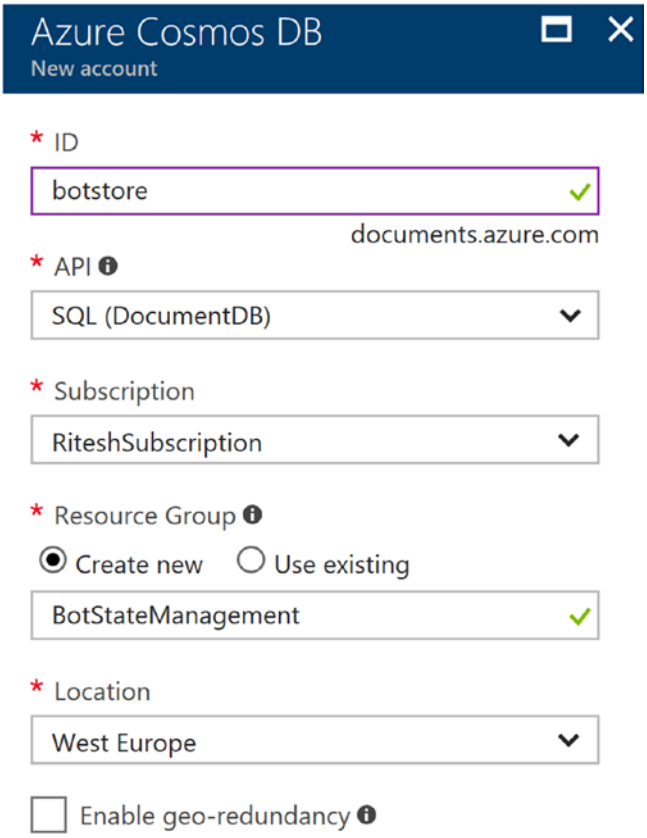

Figure 7-6. *Creating Cosmos DB service on Azure*

After Cosmos DB is provisioned, navigate to its Overview blade and copy the service endpoint URL as shown in Figure 7-7. Client applications can use this URL to connect to this instance of Cosmos DB.

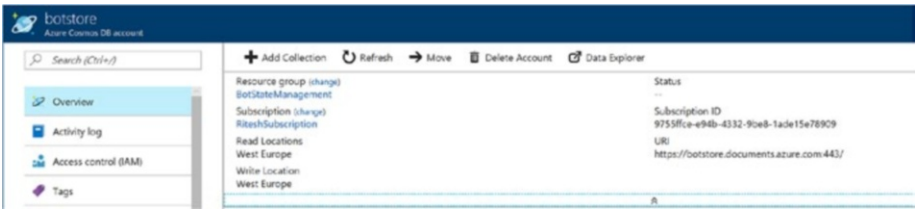

Figure 7-7. *Noting down the document's service endpoint*

Apart from URL, the primary authentication key should also be noted down in order for client applications to connect to Cosmos DB. The primary key is available from Keys blade, as shown in Figure 7-8. Both URL and authentication key are needed to successfully connect to Cosmos DB.



Figure 7-8. *Taking note of Cosmos DB primary key used to connect to it from bot application*

3. This example will reuse the same solution created in the previous section for managing state using dialog, but with more control. The Bot template ensures that the necessary assemblies related to Bot Builder and Connector SDKs are already referenced by the project. If these are not available, then they are downloaded and installed by NuGet.

   Bot Builder SDK provides the `DocumentDbBotDataStore` class, which helps in working with DocumentDB as a state store. This class is part of the `Microsoft.Bot.Builder.Azure` namespace. If this class is not available on your system, it means that this version of Bot Builder Azure SDK is not available and should be downloaded and installed.

Navigate to Tools ➤ NuGet Package Manager ➤ Package Manager Console and run this command:

```
Install-Package Microsoft.Bot.Builder.Azure
```

It is used for installing assemblies containing classes related to Azure services, as shown in Figure 7-9. In this case, version 3.2.1 is the version downloaded and installed for the `Microsoft.Bot.Builder.Azure` assembly.

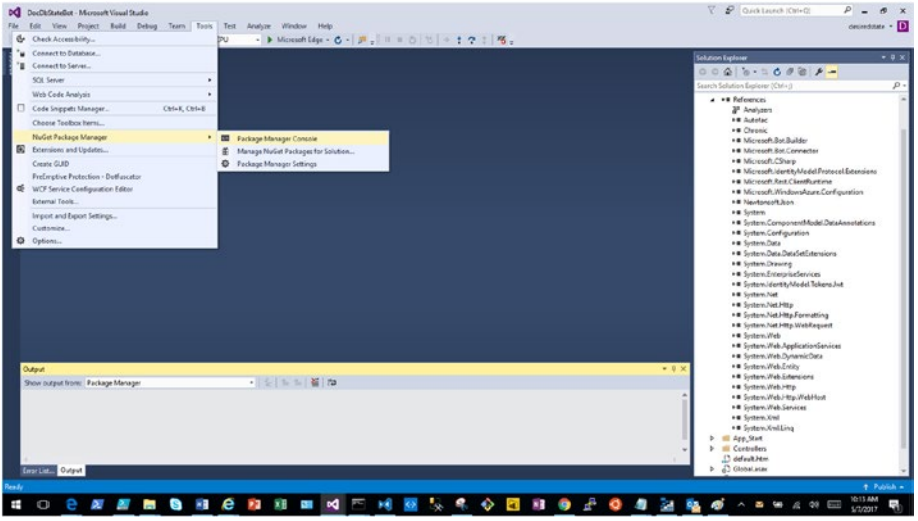

Figure 7-9. *Opening Package Manager Console in Visual Studio*

4. Open the `global.asax` file and update it with the code within the `Application_Start` function. `DocumentDBBotDataStore` implements the `IBotDataStore` interface, providing necessary functions to wire up data stores to the state store. MS Bot framework utilizes Autofac Inversion of Control (IoC) to create objects and services. By default, Bot framework uses `CachingBotdataStore` as its data store, which should be overridden with `DocumentDbBotDataStore` to store state data in Cosmos DB rather than in the default cache. An instance of `ContainerBuilder` is created (part of the Autofac library), and an Azure module is registered with it, and Cosmos DB is registered as the new state store. Finally, the container is updated to reflect the changes made to available services.

At this stage, if the name of the Cosmos DB collection is not provided, Bot framework will provide a default name `botcollection`. The `DocumentDbBotDataStore` constructor accepts `databaseId` and `collectionId` parameters, and users can provide their own values. They would be created if they do not pre-exist.

```
            Uri docDbServiceEndpoint = new
            Uri(ConfigurationManager.AppSettings["DocumentDbService
            Endpoint"]);

            string docDbEmulatorKey = ConfigurationManager.AppSet
            tings["DocumentDbAuthKey"];

            var builder = new ContainerBuilder();

            builder.RegisterModule(new    AzureModule(Assembly.
            GetExecutingAssembly()));

            var store = new DocumentDbBotDataStore(docDbService
            Endpoint,     docDbEmulatorKey);

      builder.Register(c => store)
                .Keyed<IBotDataStore<BotData>>(AzureModule.Key_
                DataStore)
                .AsSelf()
                .SingleInstance();

            builder.Update(Conversation.Container);

GlobalConfiguration.Configure(WebApiConfig.Register);
```

5. Update the imported namespace in the `global.asax` file:

```
using System;
using System.Web.Http;
using System.Configuration;
using System.Reflection;
using Autofac;
using Microsoft.Bot.Builder.Azure;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Connector;
```

6. Update the `web.config` file and add both the Cosmos DB service endpoint URL and the principal authentication key as additional name–value pairs in the `appSettings` section. These values are referenced in the `Application_Start` code:

```
<add key="DocumentDbServiceEndpoint"  value="https://botstore.
documents.azure.com:443/" />

    <add key="DocumentDbAuthKey" value="xxxxxxxxxxxxxxxxxxxxxxxx
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    xxxxxxxxxxxxxxxxxxxxxxxx==" />
```

Run the bot and complete a conversation with it using Bot Emulator, as shown in Figure 7-10.

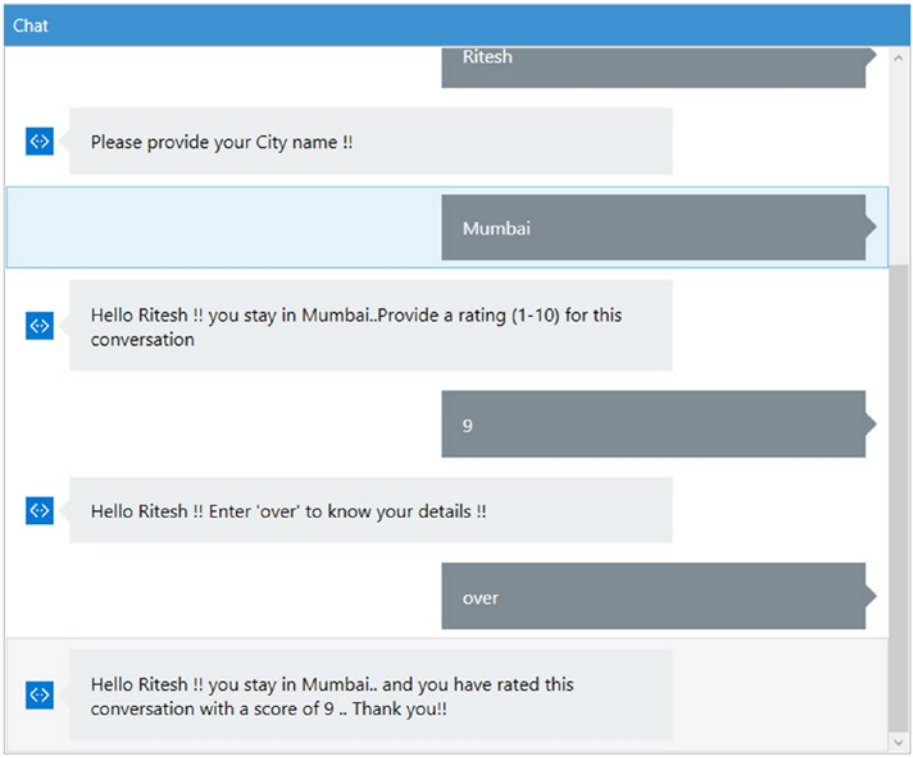

Figure 7-10. *Complete sample bot showing usage of different stores*

Now, to verify that the bot state is stored in the Cosmos DB collection, navigate to Cosmos DB service on Azure. Figure 7-11 shows data in JSON format stored in the User data store; Figure 7-12 shows data stored in the Conversation data store; and Figure 7-13 shows data stored in the Private Conversation data store.

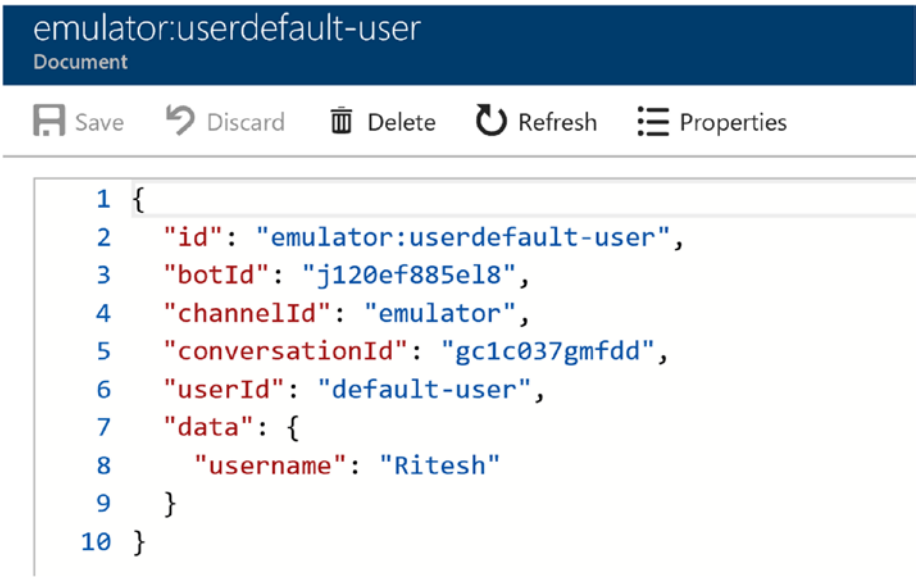

Figure 7-11. *User state data in Cosmos DB collection*



Figure 7-12. *Conversation state data in Cosmos DB collection*

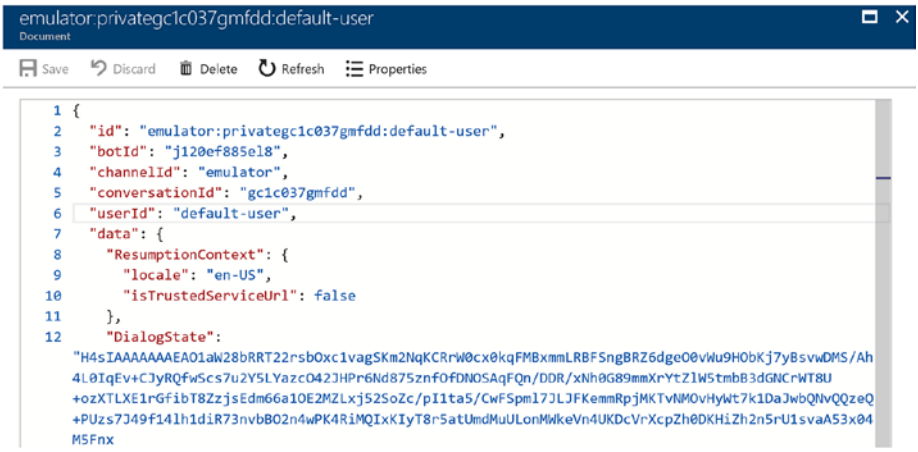

Figure 7-13. *Private Conversation state data in Cosmos DB collection*

Table Storage as Custom State Data Store

The high-level steps for using a Table storage state data store remain the same as those for the Cosmos DB data store:

1. Identify the data store. This example uses Table storage as its data source.
2. Provision the data store if it does not already exist.
3. Create or open a bot project using the Bot Application template
4. Add code to wire up and use Table storage as the data store provided by Bot Builder SDK.
5. Write dialogs normally without any knowledge of location or type of state store.

Here are the steps in more detail:

1. In this example, Azure Table storage is used as the state store. Azure Table storage is a service that stores structured NoSQL data in the cloud, providing a key–attribute store with a schema-less design. That Table storage is schema-less helps in creating a flexible database design to accommodate data in different shapes quite easily without any changes.
2. As mentioned before, Table storage is a managed Azure service. To use Table storage, you must have a valid Azure subscription. Log in to Azure as a valid user and create a new Storage account. Provide a name and select the appropriate subscription and resource group to provision a storage account. This is shown in Figure 7-14.

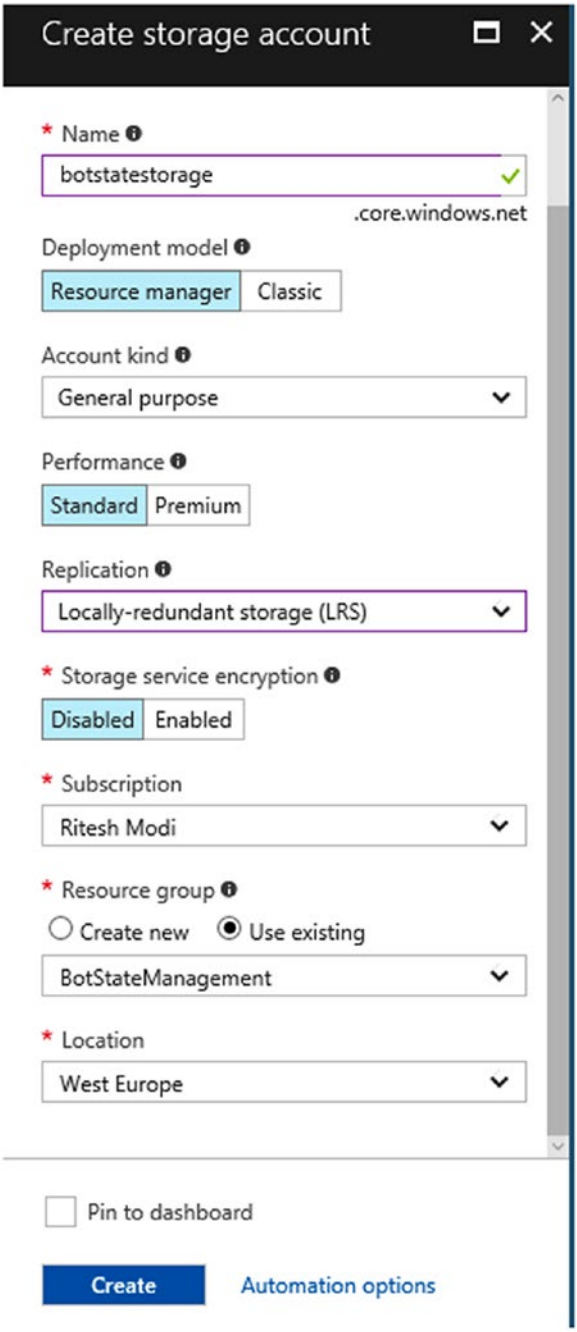

Figure 7-14. *Creating a new Azure Storage account*

After your Storage account is provisioned, navigate to its Access Keys blade and copy the Connection String against Key1 as shown in Figure 7-15. Client applications can use this connection string to connect to this Storage account.

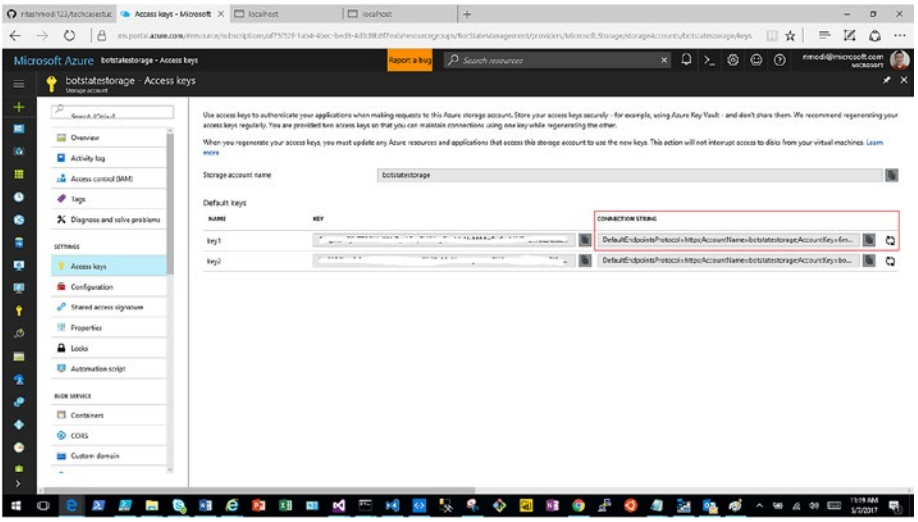

Figure 7-15. *Taking note of Azure Storage account connection string for bot to connect to it*

3. This example will reuse the same solution created in the previous section for managing state using dialog, but with more control. The Bot Application template ensures that the necessary assemblies related to the Bot Builder and Connector SDKs are already referenced within the project. If these are not available, they can be downloaded and installed using the NuGet Package Manager and provider.

   Bot Builder SDK provides a `TableBotDataStore` class that helps in working with `TableBotDataStore` as the state store. This class is part of the `Microsoft.Bot.Builder.Azure` namespace. If this class is not available on your system, it means that this version of Bot Builder Azure SDK is not available and should be downloaded and installed.

   Readers are advised to check out the previous section on DocumentDB for installing assemblies using Visual Studio Package Manager.

4. Open the `global.asax` file and update with the code within the `Application_Start` function. `TableBotDataStore` implements the `IBotDataStore` interface, providing necessary functions to wire up Table storage to the bot's state store. The steps are the same as we saw using DocumentDB, with the only difference being the use of `TableBotDataStore` as the data store instead.

```
        string storageConnectionString = ConfigurationManager.App
        Settings["storageConnectionString"];

        var builder = new ContainerBuilder();

        builder.RegisterModule(new AzureModule(Assembly.
        GetExecutingAssembly()));

        var store = new TableBotDataStore(storageConnectionString);
        builder.Register(c => store)
            .Keyed<IBotDataStore<BotData>>(AzureModule.Key_
            DataStore)
            .AsSelf()
            .SingleInstance();

        builder.Update(Conversation.Container);

    GlobalConfiguration.Configure(WebApiConfig.Register);
```

5. Update the imported namespace in the `global.asax` file as shown in this section:

```
using System;
using System.Web.Http;
using System.Configuration;
using System.Reflection;
using Autofac;
using Microsoft.Bot.Builder.Azure;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Connector;
```

6. Update the `web.config` file and add the Azure Storage account connection string as an additional name–value pairs in the `appSettings` section. These values are referenced and used from the `Application_Start` code in `global.asax`.

```
<appSettings>
    <!-- update these with your BotId, Microsoft App Id, and your
    Microsoft App Password-->
    <add key="BotId" value="YourBotId" />
    <add key="MicrosoftAppId" value="" />
    <add key="MicrosoftAppPassword" value="" />
    <add key="storageConnectionString" value="DefaultEndpointsPro
    tocol=https;AccountName=botstatestorage;AccountKey=xxxxxxxxxx
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    xxxxxxxxxxxxxx==;EndpointSuffix=core.windows.net" />
  </appSettings>
```

7. Download and install Azure Storage Account Explorer from `http://storageexplorer.com/` to easily verify the data stored as state by `TableBotDataStore` in Azure table storage after running the sample bot. The Azure account should be added first to the explorer. This is shown in Figure 7-16.

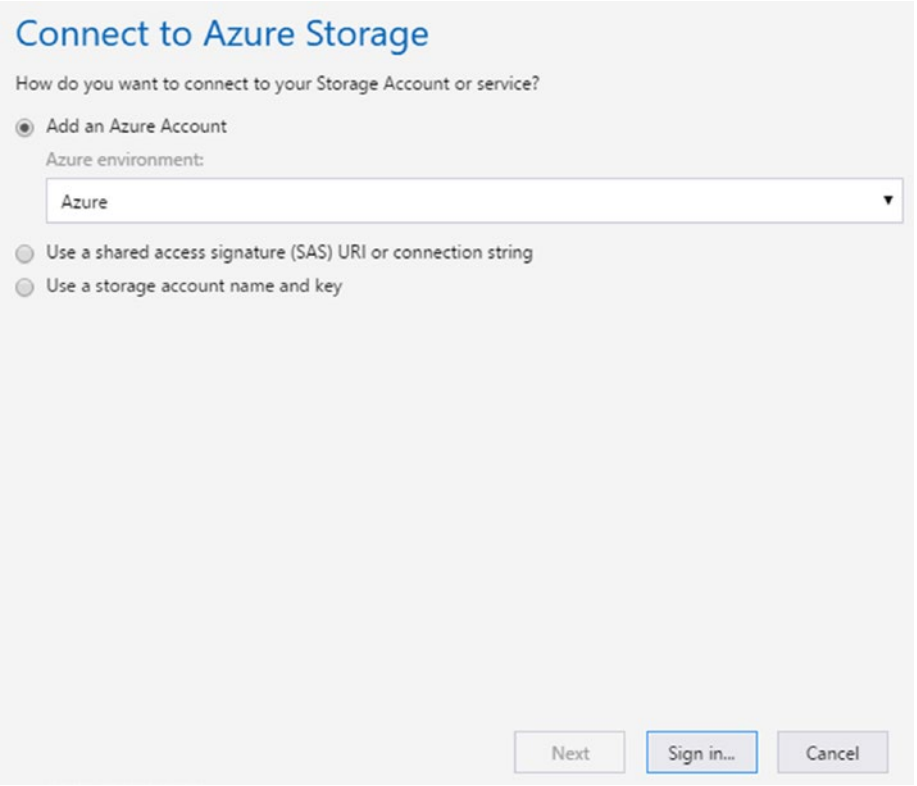

Figure 7-16. *Connecting Storage Explorer to our storage account*

Initially, the Table will be empty, as shown in Figure 7-17. This is because we have not run our bot even once.

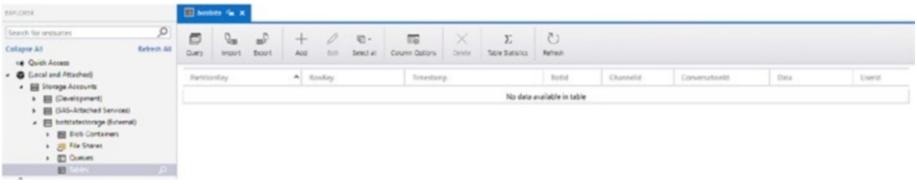

Figure 7-17. *Storage Explorer showing data in Table storage (there is no data to start with)*

8. Run the bot and complete a conversation with it using Bot Emulator, as shown in Figure 7-18. Now, if you navigate to the DocumentDB service on Azure, the bot state data should be available, as shown in Figure 7-19.

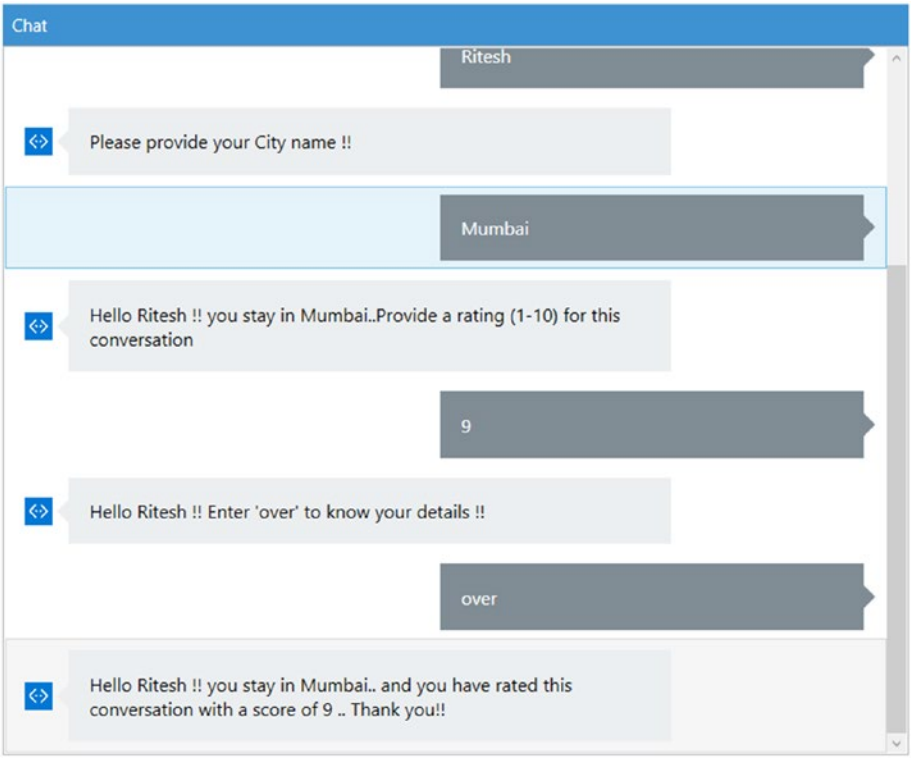

Figure 7-18. *Sample bot showing use of Table storage for storing state*

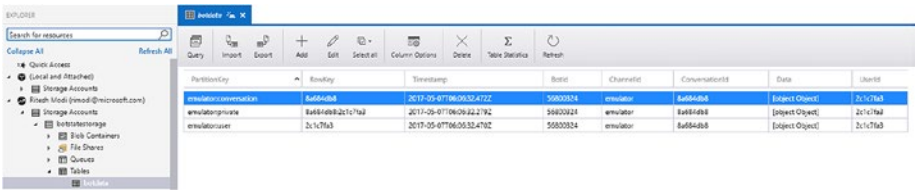

Figure 7-19. *Storage Explorer showing data in Table storage (after bot stored state)*

The data in Table storage is encoded by default. The schema for Table storage can be viewed using the explorer, and it will show the different columns created by the Bot SDK to store bot state data. This is shown in Figure 7-20.

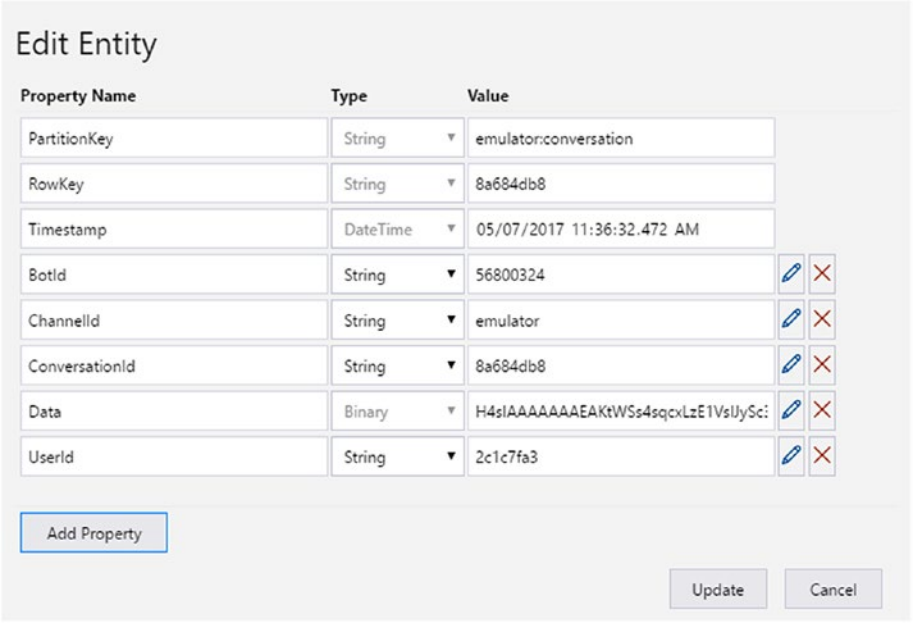

Figure 7-20. *Storage Explorer showing Table storage schema*

Summary

Microsoft offers the Bot Builder SDK so that Node.js and C# developers can build smart and intelligent bots. The SDK offers a variety of options to save state using the `StateClient` object. The data can be saved per globally or per user/conversation. Any serious bot would need to remember its users to provide personalization services and remember them the next time they come back. They would also need some transient data store to store intermediary data within a conversation to facilitate better conversations with its users. Bot SDK helps in easing this task by providing centralized state management. Users do not have to manage the infrastructure for this state, and it is available out of the box. To author advanced features in a bot, state features are generally used. It is must-have knowledge and a great toolkit for any serious bot developer.

CHAPTER 8

Dialogs

Bots are all about "Conversation as a platform." Bots help build relationships with their users. They do not perform monologues or just provide details iteratively. Just like the way you converse with different people in daily life, bots have meaningful conversations with their users. And just like the way you have different conversation with different people, bots also have different contexts and intents while having different conversations with their users.

Conversations are the core element for bots. Conversation happen when there is a dialog between people. This dialog could be one to one or one to many. Dialogs have a context, opening, state, and closing. There can be multiple dialogs with the same person. Dialogs can move from one topic to another and return back to the original topic.

Every application has some sort of user interface. While some have wizards, other have screens that transition from one screen to another. Bots also have a user interface. In fact, dialogs *are* the User Interface for bots. Dialogs enable the Bot developer to logically separate various areas of bot functionality and guide conversational flow. For example, Bot can contain a dialog that has logic to search airline tickets, another dialog to buy airline tickets, and a separate dialog to book a hotel room.

MS Bot framework provides a user interface in terms of prompts, carousel, buttons, and other elements apart from text, though text forms the majority of a dialog's UI.

Bot framework provides a couple of ways to author bots. We have seen in previous chapters that a complete bot can be written within the controller itself. Another approach is to use the Dialog framework, which provides a rich plumbing infrastructure to manage the dialog's stack and state as well as the current dialog, and to transition from one dialog to another and other dialog management–related activities.

The Dialog Model

The Dialog model provides access to a rich framework provided by Bot Builder SDK to manage the entire lifecycle of multiple dialogs, interactions between dialogs, the current dialog, the state machine for all dialogs, and the loading and saving of the dialog state. Figure 8-1 shows the class hierarchy for dialogs in Bot SDK.

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_8

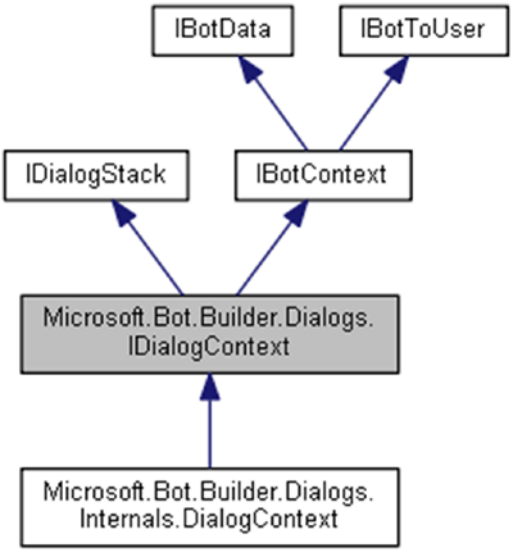

Figure 8-1. *Dialog class and interface hierarchy*

IBotData

This interface provides function definitions for managing dialog state—retrieving and saving bot state data to different stores, as follows:

- UserData
- PrivateConversationData
- ConversationData

IBotTouser

This interface provides function definitions for creating and sending messages to users.

IDialogStack

This interface is responsible for providing function definitions to manage the stack of dialogs in the conversational process. It provides definitions that help in transitioning from one dialog to another and also provides the hook that suspends the current dialog until an external event has been sent to the bot.

IBotContext

The `IBotContext` interface inherits `IBotData` and `IBotToUser` interfaces and provides properties to retrieve `conversationData`, `UserData`, and `PrivateConversationData` states.

Dialog Stack

Dialog stack, as the word signifies, means a stack of dialogs. A stack is a group of items where some items are kept on top of other items. Dialog stack means creating a stack in which dialogs are placed on top of other dialogs. There is always a root dialog when the conversation starts with the user. This root dialog can further invoke other dialogs, and they are placed on top of the calling dialog in a stack. The dialog that is on top of the stack is the current dialog in progress and controls the conversation. Every new message sent by the user will be subject to processing by that dialog until it either closes or redirects to another dialog. When a dialog ends its execution, it's removed from the stack, and the previous dialog in the stack assumes control of the conversation. Dialog stacks are a means and mechanism for MS Bot framework to know the hierarchy of dialogs in conversation, the current dialog, and the last dialog. Typically, when having a guided conversation, multiple dialogs are involved, and the dialog stack helps in managing all constituent dialogs.

Dialog Context

Dialog context is the core concept when working with dialogs. *Context* is the entry point into the Dialog framework. By the time a message from the user reaches a method within a dialog, the context is already populated with the dialog stack, the current dialog in the stack, and the state for the current dialog. Dialog context is pre-populated by Bot framework and available to every method and function in a dialog. The context helps in retrieving the current bot state, transitioning between dialogs, and managing a stack of dialogs.

Root Dialog

Every bot should start its conversation with users using a root dialog. This root dialog acts as the first point of contact and is responsible for creating new dialogs, since conversation flow starts with user. This root dialog should create newer dialogs or implement the bot conversation. As a best practice, based on nature and the topic of conversation, newer Dialogs should be created using single-responsibility intuition.

Building a Simple Dialog Bot

In this section, a bot with a single dialog will be created. It will provide the steps required to create such a bot and will also explain in detail the concepts around it.

Follow these steps to create a dialog-based bot:

1. Create a new bot project.
2. Add the `Microsoft.Bot.Builder.Dialog` namespace in the `MessagesController.cs` file.
3. Add a new class file and add the `Microsoft.Bot.Builder.Dialog` and `Microsoft.Bot.Connector` namespaces.
4. Derive the new class from the `IDialog` interface and implement its only method: `StartAsync`.
5. Change the code of the `MessagesController` class to invoke your root dialog.

In this sample, we will create a bot that will be responsible for accepting a word and returning its synonyms. For returning synonyms, REST APIs provided by Oxford Dictionaries are used.

Create an account with Oxford Dictionaries by going to https://developer.oxforddictionaries.com. There is a free plan you can use to test its usage.

After your account is created, a new application ID and key should be generated. These will be used in the bot to authenticate the Oxford Dictionaries API. Both application ID and key should be saved in a secure location and should not be shared with anyone.

SimpleDialog.cs

Add a new class file and name it `SimpleDialog`.

Add the `Microsoft.Bot.Builder.Dialogs` and `Microsoft.Bot.connector` namespaces to this file. It provides access to all classes and interfaces from the Bot Builder framework that help in creating and using a dialog.

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

Derive the new class from the `IDialog<string>` interface. There are two implementations of this interface, one with C# generics and another without it. Each bot can return a value when it has completed its execution, and this datatype helps in identifying the type of return value coming from the bot.

Decorate the new bot class with the `[Serializable]` attribute. Dialog state is saved automatically by Bot framework, and thereby the framework mandates that bot dialogs should be serializable. Without this attribute, the framework would not be able to save dialog state to its state store.

Implement the single method `StartAsync` provided by the `IDialog` interface. This method gets an `IDialogContext` object by default and will be used to send messages to the user, suspend the current dialog, navigate to another dialog, and store and load state.

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

        }
```

`Context.wait` suspends the current execution, saves the state, and waits for inputs from the user. As soon as it gets an input from the user, it executes the function that is provided to it as a parameter while suspending the current execution. This function acts as a continuation delegate and gets executed when a new message arrives.

The implementation of the `MessageHandler` function is the core of the `SimpleDialog` dialog. It is here that the main logic of invoking the Oxford Dictionaries REST API is executed and its return value is captured, iterated over, and returned back to the user.

Even this function is passed `IDialogContext` by the Bot framework, and it also gets the incoming activity containing the user-provided text.

```
private async Task MessageHandler(IDialogContext context,
IAwaitable<IMessageActivity> result)
```

The incoming activity is stored in the `message` variable, and the bot informs and confirms with the user using the `PostAsync` method:

```
var message = await result;
await context.PostAsync("You said: " + message.Text);
```

A `StringBuilder` typed variable to hold `returnMessage` is declared next:

```
StringBuilder returnMessage = new StringBuilder();
```

The Oxford Dictionaries REST API needs language to be sent as part of its URL. It is set to "EN." The REST API for synonyms is available at `https://od-api.oxforddictionaries.com/api/v1/entries/{source_lang}/{word_id}/synonyms`.

The `word_id` is the word that the user is interested in finding synonyms for:

```
StringBuilder returnMessage = new StringBuilder();
string language = "en";
string word_id = message.Text;

string url = "https://od-api.oxforddictionaries.com:443/api/v1/entries/" +
language + "/" + word_id + "/synonyms";
```

Next, a new managed object of type `HttpClient` is created. It will get disposed of automatically after the `using` block has finished its execution. Oxford Dictionaries REST API expects the application ID and application key to be part of the request header, and

they are added to newly created `HttpClient` object. The `HttpClient` object is needed to invoke the LUIS endpoint. The request is sent using the `GetStringAsync` method from this class. It should be provided with the complete URL to the REST endpoint along with text from the user. The user text is available from the `text` property of the `activity` object.

```
using (var httpClient = new HttpClient())
            {
                httpClient.DefaultRequestHeaders.Add("ContentType",
                "application/json");
                httpClient.DefaultRequestHeaders.Add("app_id", "294b41ad");
                httpClient.DefaultRequestHeaders.Add("app_key",
                "6ae4a7a761884936995f7d875ef479eb");
```

The REST API is invoked using the `GetClientAsync` method, and the returned JSON value is converted into a dynamic .NET object using the `JSONConvert` class. This class is available as part of the `Newtonsoft.JSON` namespace. The return value from the REST API contains deep-nested arrays and objects. To navigate to synonyms, multiple `foreach` loops are used and the synonyms are appended to the `StringBuilder` object.

```
try
{

    var response = await httpClient.GetStringAsync(new Uri(url));

   dynamic rr = JsonConvert.DeserializeObject(response);
    foreach (var obj in rr.results)
    {
        foreach (var obj1 in obj.lexicalEntries)
        {
            foreach (var obj2 in obj1.entries)
            {
                foreach (var obj3 in obj2.senses)
                {
                    foreach (var obj4 in obj3.synonyms)
                    {
                        returnMessage.Append(obj4.text);
                        returnMessage.Append(" , ");
                    }
                }
            }
        }
    }

}
catch (Exception ex)
{
```

```
    await context.PostAsync(ex.ToString());
}
```

Finally, the response containing the string from the StringBuilder object is returned to the user:

```
await context.PostAsync("Synonyms for your word : " + message.Text + " are ");
await context.PostAsync(returnMessage.ToString());
context.Wait(MessageHandler);
```

The code for this function is shown here:

```
private async Task MessageHandler(IDialogContext context,
IAwaitable<IMessageActivity> result)
        {
            var message = await result;
            await context.PostAsync("You said: " + message.Text);

            StringBuilder returnMessage = new StringBuilder();
            string language = "en";
            string word_id = message.Text;

            string url = "https://od-api.oxforddictionaries.com:443/api/v1/
            entries/" + language + "/" + word_id + "/synonyms";

            using (var httpClient = new HttpClient())
            {
                httpClient.DefaultRequestHeaders.Add("ContentType",
                "application/json");
                httpClient.DefaultRequestHeaders.Add("app_id", "294b41ad");
                httpClient.DefaultRequestHeaders.Add("app_key",
                "6ae4a7a761884936995f7d875ef479eb");

                try
                {

                   var response = await httpClient.GetStringAsync(new Uri(url));

                   dynamic rr = JsonConvert.DeserializeObject(response);
                    foreach (var obj in rr.results)
                    {
                        foreach (var obj1 in obj.lexicalEntries)
                        {
                            foreach (var obj2 in obj1.entries)
                            {
                                foreach (var obj3 in obj2.senses)
                                {
                                    foreach (var obj4 in obj3.synonyms)
```

```
                                {
                                    returnMessage.Append(obj4.text);
                                    returnMessage.Append(" , ");
                                }
                            }
                        }
                    }
                }

            }
            catch (Exception ex)
            {

                await context.PostAsync(ex.ToString());
            }

        }
        await context.PostAsync("Synonyms for your word : " + message.
        Text + " are ");
        await context.PostAsync(returnMessage.ToString());
        context.Wait(MessageHandler);
    }
```

MessagesController.cs

Add the `Microsoft.Bot.Builder.Dialogs` namespace to this file. It provides access to all classes and interfaces from the Bot Builder framework that help in creating and using a dialog.

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

Open and replace the following code in `MessagesController.cs` within the `Controllers` folder:

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

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

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

with the following:

```
await Conversation.SendAsync(activity, () => new SimpleDialog());
```

The `post` method in `MessagesController` is the entry point to the bot, and the `Conversation.SendAsync` method is the entry point to the Dialog framework. The method is a state machine and is responsible for creating a new instance of Dialog, loads the last saved state from the state store, finds the current suspended location, and resumes the conversation from there, replying to the user and saving the new state to the state store. It also calls the bot's implementation of the `StartAsync` method, and from there the dialog takes control of the conversation.

Interacting with this bot using Bot Emulator is shown in Figure 8-2.

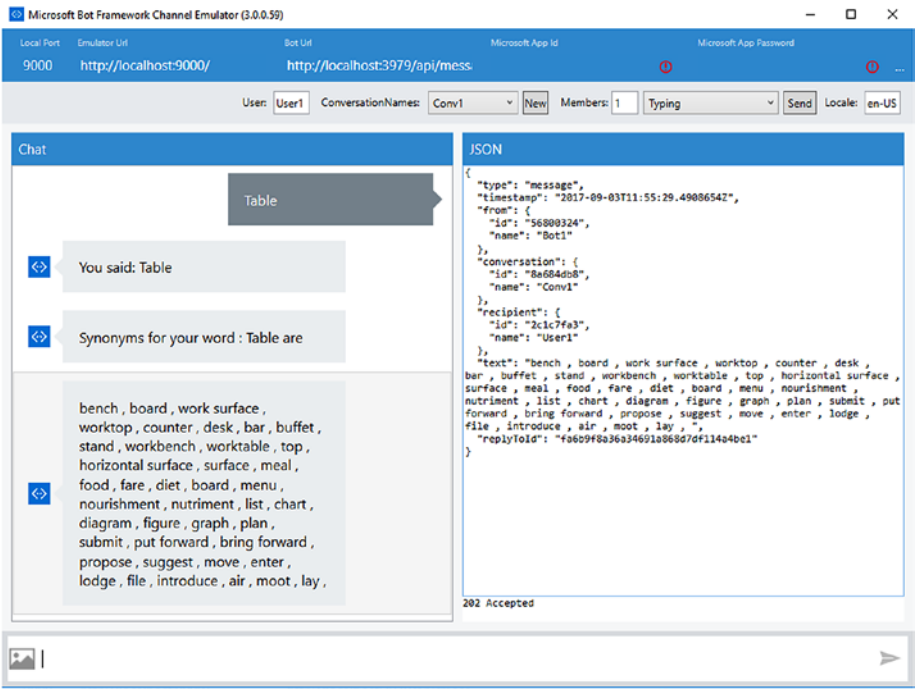

Figure 8-2. *Interacting with bot using Bot Emulator (without dialogs)*

Here, the user provides "Table" as input, and all synonyms from the Oxford Dictionary are returned as response.

Creating Multi-Dialog Bots

Now that we understand single-bot implementations, it's time to understand the technicalities for implementing bots with multiple dialogs.

There are three different ways in which multiple dialogs can be implemented:

- Nested dialogs – In this implementation, one Dialog calls another dialog, and second dialog calls further dialogs in turn.

- Façade dialogs – In this, a dialog calls multiple dialogs, each responsible for its own functionality.
- Combined dialogs – Here, both nested and façade dialogs are used for bot functionality.

In this section, an example of using combined dialogs will be illustrated and explained.

There are two primary ways in which bots can interact with each other. Bot framework provides the following:

- `IDialogContext.Forward` method calls `StartAsync` method on another dialog and passes the message as well.
- `IDialogContext.Call` method calls `StartAsync` method on another dialog but does not pass any messages to it. The target dialog should initiate its own conversation to get its conversation going.

When a dialog is initiated into a conversation using either the call or the forward method, it should either call the `IDialogContext Done` method to inform Bot framework that it has completed its execution successfully and return any return value, or call the `IDialogContext Fail` method to inform Bot framework that it has failed in its execution and return an exception to its parent dialog.

Scenario

To explain the multiple-bot scenario, a dictionary bot will be created with two dialogs, as follows:

- A dialog responsible for providing antonyms
- Another dialog responsible for providing synonyms

A user will initiate the conversation, and the bot will ask the user to either type *Antonym* or *Synonym*. Based on the user input, either the antonym dialog or the synonym dialog will be used.

There will also be a root dialog that will act as the entry point to the bot dialogs and be responsible for further creating and transitioning to other dialogs. The root dialog uses the forward method of `IDialogContext` to invoke both antonym and synonym dialogs.

One of the dialogs, called a "support" dialog, will be created to help users open a ticket for any issue with the dictionary bot. The root dialog uses the forward method of `IDialogContext` to invoke both antonym and synonym dialogs. The root dialog uses the call method of `IDialogContext` to invoke the support dialog.

Solution

The picture shown in Figure 8-3 depicts the dialog stack for the sample application. The source code for is available as `MultiDialogExample` within accompanying code bundle.

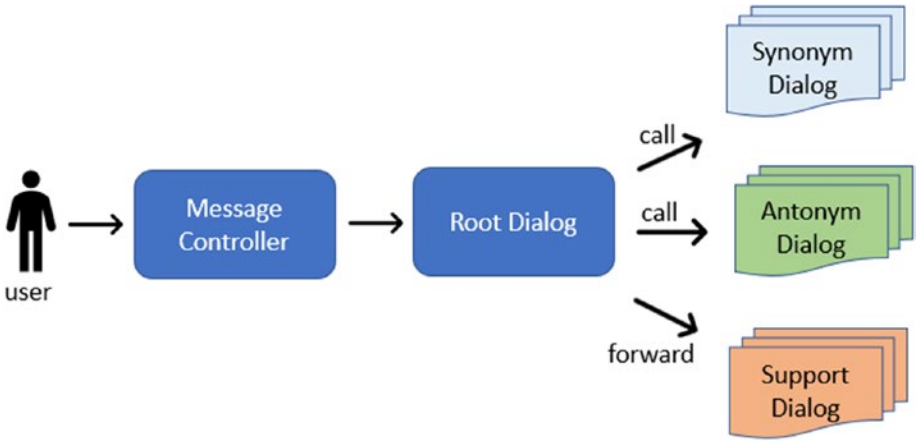

Figure 8-3. *Dialog composition for multi-dialog bot application*

The Synonym dialog created in the previous section while discussing the simple dialog will be reused in this example. Moreover, few more dialogs will be created for this example, as follows:

- Root dialog that will orchestrate creating the appropriate dialog based on user input
- Antonym dialog for fetching all antonyms of a word
- Support dialog for generating a support ticket for any problems with the not

Both antonym and synonym dialogs will be invoked using the call method, and the support dialog will be used using the forward method.

RootDialog.cs

RootDialog is the first point of contact for our lexicon bot. MessagesController creates the RootDialog and invokes its StartAsync method. Within this method, the dialog expects the user to initiate the conversation by suspending the current dialog, using the IDialogContext.wait method to pass in the continuation handler start as a parameter.

```
public async Task StartAsync(IDialogContext context)
        {
            context.Wait<IMessageActivity>(start);
        }
```

As soon as the user initiates and provides its input (e.g., Hi/Hello), the bot reloads the root dialog and its state and executes the start method. In this method, the bot asks the user to provide input—antonym or synonym—and again suspends the current dialog, using the IDialogContext.wait method to pass in the continuation handler DecisionMaker as its parameter.

```
private async Task start(IDialogContext context,
IAwaitable<IMessageActivity> result)
{
    await context.PostAsync("Please provide your input !! Synonym or
    Antonym");
    context.Wait<IMessageActivity>(DecisionMaker);
}
```

DecisionMaker is the heart of RootDialog. Here, the user input is evaluated and a decision is made about the next dialog to be created and passed over the control. There is also a support dialog that will be invoked if the user provides *support* as a keyword. Readers should notice that for the support dialog the forward method of IDialogContext is used while for the antonym and synonym dialogs the call method is used.

The call method takes two parameters—an instance of the next dialog to be made the current dialog in the stack and a message handler that will be executed when the next dialog has finished its execution either successfully or failure. The bool datatype in the call method signifies the return value from the next bot. This value should be set in the target dialog to announce whether it was successful in its execution or not, and the parent dialog should check this value to decide on next steps. As a good practice, it is always advised to return an appropriate type from child dialogs to know their execution status.

The forward method takes four parameters—an instance of the next dialog to be made the current dialog in the stack, a message handler that will be executed when the next dialog has finished its execution either successfully or unsuccessfully, the message itself from the user, and a cancellation token. The difference between the call and forward methods is that a message is passed along with the invocation of the forward method.

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

```
            if ("support" == message.Text.ToLower())
                await context.Forward(new Support(), SupportHandler,
                message, System.Threading.CancellationToken.None);
            else if ("synonym" == message.Text.ToLower())
                context.Call<bool>(new Synonym(), SynonymHandler);
            else if ("antonym" == message.Text.ToLower())
                context.Call<bool>(new Antonym(), SynonymHandler);
            else
               context.Wait(start);
        }
```

Next, two handlers—one for handling the return from the synonym and antonym dialogs and another for handling the return from the support dialog. Synonym and antonym use the same handler because of the similarity in their return values and its handling. This handler gets `IDialogContext` as its first argument and a result argument of type `bool`. The parent dialog gets the return value from the child dialog using this argument. Both synonym and antonym dialogs are responsible for displaying and returning results to its user, and the root dialog is only tasked to ask questions. Here, the root dialog again asks the user about their choice.

```
private async Task SynonymHandler(IDialogContext context, IAwaitable<bool>
result)
{
    bool message = await result;
    await context.PostAsync("Please provide your input !! Synonym or
    Antonym");
    context.Wait<IMessageActivity>(DecisionMaker);
}
```

The support dialog returns an integer value representing a ticket number. This is captured and returned to the user.

```
private async Task SupportHandler(IDialogContext context, IAwaitable<int>
result)
        {
            var ticketNumber = await result;

            await context.PostAsync($"Thanks for contacting our support
            team. Your ticket number is {ticketNumber}.");
            context.Wait(start);
        }
```

Synonym.cs

This is a child dialog invoked using the `IDialogContext call` method. This implementation is the same as that of the simple dialog that was discussed in last section. Please refer to the previous section for more details.

Antonym.cs

This is a child dialog invoked using the `IDialogContext call` method and is very similar to the synonym dialog. The only difference between the two implementations are the Oxford Dictionaries REST API URL and schema for the return value. The REST URL for antonym is https://od-api.oxforddictionaries.com/api/v1/entries/{source_lang}/{word_id}/antonyms, and the returned JSON schema contains antonyms instead of synonyms. Notice the call to the `Done` method and passing of a return value using the same.

```
dynamic rr = JsonConvert.DeserializeObject(response);
                foreach (var obj in rr.results)
                {
                    foreach (var obj1 in obj.lexicalEntries)
                    {
                        foreach (var obj2 in obj1.entries)
                        {
                            foreach (var obj3 in obj2.senses)
                            {
                                foreach (var obj4 in obj3.antonyms)
                                {
                                    returnMessage.Append(obj4.text);
                                    returnMessage.Append(" , ");
                                }
                            }
                        }
                    }
                }

await context.PostAsync("Synonyms for your word : " + message.Text + " are ");
            await context.PostAsync(returnMessage.ToString());
            context.Done(true);
```

Support.cs

This is a child dialog invoked using the `IDialogContext forward` method. This is invoked when the user provides the *support* keyword. This is a simple dialog that generates a new random number and returns it back to the parent dialog as a ticket number. Notice the call to the `Done` method and the passing of the return value using the same.

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

```
{
    context.Wait(MessageHandler);

}

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

    var ticketNumber = new Random().Next(0, 20000);

    await context.PostAsync($"Your message '{message.Text}' was
    registered. Once we resolve it; we will get back to you.");

    context.Done(ticketNumber);
}
```

MessagesController.cs

The is the entry class for the bot. It implements the `POST` method and accepts incoming activities from the user. It creates the `RootDialog` and passes control over to it:

```
await Conversation.SendAsync(activity, () => new RootDialog());
```

FormFlow

The Dialog model provides access to a rich framework provided by Bot Builder to give complete control over the bot implementation. Dialogs offer complete flexibility to implement the bot's conversational flow and its business logic; however, writing a guided conversation can be a difficult proposition both in terms of effort required and complexity. Conversations can be quite complex. At any point in a conversation, there are multiple options and directions in which the conversation can flow. Users may ask for help, support, confirmation for previous inputs, and more. It can be a daunting task to implement all of this functionality using dialogs, especially when all of it needs to be implemented from scratch.

There are two ways in which FormFlow can generate conversation bots, as follows:

- Using C# classes – used in this book
- Using JSON schema

Bot Builder SDK for C# provides the FormFlow framework with which writing guided conversation–based bots can be simplified, and it can also help you save time. FormFlow allows you to design a guided conversation–based bot using simple guidelines in terms of C# enums and properties. Dialogs are generated from these guidelines and

wired together to form a complete conversation. Needless to say, there are constraints while using FormFlow because of its inherit nature, and the bot designer will lose some flexibility when creating a bot. FormFlow dialogs can be combined with general dialogs to form more complex conversations.

All the information that a bot should ask the user for in a guided conversation should be provided by the bot designer to FormFlow. The information should be available within a C# class as properties. Each property in this class should be based on following data types:

- Integral (sbyte, byte, short, ushort, int, uint, long, ulong)
- Floating point (float, double)
- String
- DateTime
- Enumeration
- List of enumerations

Building a Simple FormFlow Bot

The first step in creating a FormFlow bot is to declare a class with properties. If properties are based on enums, those Enums should be defined as well. In this example, a simple Notebook configuration information is sought by the bot from its users.

The `FormFlowSimple` class defines the form along with its properties. The properties are based on Enum declarations within the same namespace. The form is used to order Notebook based on the custom configuration.

The `Microsoft.Bot.Builder.FormFlow` namespace should be imported into both `messagesController.cs` and the custom form class.

The code for the form is shown next. Enums for delivery options, RAM options, screen-size options, disk options, CPU options, and operating system options are declared. A class containing public properties for these enums are declared. There is additionally a `string` property to hold the credit card number. The class also implements a `BuildForm` static function that is responsible for building the form and returning a dialog containing the entire conversation flow. This function creates a new instance of `FormBuilder. FormBuilder` implements Fluent API and helps in chaining multiple form-based calls together, including the start message.

```
namespace FormFlowExample
{

    public enum Deliveryoptions {
        CashOnDelivery, CreditCard, DebitCard, Wallet, BitCoin
    }

    public enum RamOptions
    {
        TwoGB, FourGB, EightGB, SixteenGB
    }
```

```
    public enum ScreenSizeOptions
    {
        Small, Medium, Large
    }

    public enum DiskOptions
    {
        SSD, HybridDrive, MechanicalHardDrive
    }

    public enum CpuOptions
    {
        AMD, IntelCoreI5, IntelCoreI7, IntelPentium, IntelCeleron
    }

    public enum OperatingSystemOptions
    {
        DOS, Windows, Linux, Mac, Chrome
    }

    [Serializable]
    public class FormFlowSimple
    {
        public OperatingSystemOptions? OS;
        public CpuOptions? cpu;
        public DriveOptions? drive;
        public ScreenSizeOptions? screen;
        public RamOptions? ram;
        public Deliveryoptions? delivery;
        public string CreditCardNumber;

        public static IForm<FormFlowSimple> BuildForm() {
            return new FormBuilder<FormFlowSimple>().Message("Welcome to
            NoteBook builder..").Build();
        }
    }
}
```

The next step is to connect the form to the bot. This happens within the `messagesController.cs`, which contains the Post REST API. An additional function is defined in this class that is responsible for creating a `FormDialog`. `FormDialog` is passed the static `BuildForm` function in our `Form` class. `FormDialog` invokes this function, which in turn builds the entire form and returns back to `FormDialog`. Fluent API is used again, and the `Do` method is called, which accepts a message handler that is called after the user confirms all their input and has completed the conversation. In this function, if there is an exception, the user is notified; otherwise, a success message is sent.

```
internal static IDialog<FormFlowSimple> MakeRootDialog()
        {
            return Chain.From(() => FormDialog.FromForm(FormFlowSimple.
            BuildForm)).Do(async (context, order) =>
            {
                try
                {
                    var completed = await order;
                    // Actually process the sandwich order...
                    await context.PostAsync("Your laptop order is processed
                    and will be delivered soon !");
                }
                catch (FormCanceledException<FormFlowSimple> e)
                {
                    string reply;
                    if (e.InnerException == null)
                    {
                       reply = $"Bot was at {e.Last} stage, try again later !";
                    }
                    else
                    {
                        reply = "Apologies, there was an issue with the Bot!
                        Please try again";
                    }
                    await context.PostAsync(reply);
                }
            });

        }
```

This function is called by the `Post` method, similar to other dialogs using the `Conversation.SendAsync` method. Here, instead of creating a dialog, the `MakeRootDialog` function is called.

FormFlow provides multiple features, like each property name is used to ask the user, prompts are displayed to the user in order of their property declaration in the class, each option within enums is divided into multiple words based on capital letters, a confirmation message containing all chosen options is run by to user (see Figure 8-4), and so on. More details about these features can be found at https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-formflow and https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-formflow-advanced.

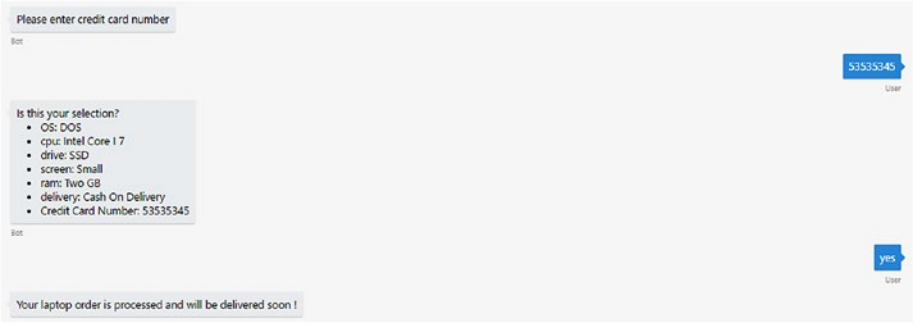

Figure 8-4. *FormFlow asks for confirmation as final step in conversation*

FormBuilder

The previous section introduced FormFlow, and FormBuilder is the main engine that helps in FormFlow bots. FormBuilder provides a rich and versatile framework and APIs that help with customizing almost every aspect of the FormFlow. This section will deep-dive into some of these customization aspects of FormBuilder; however, not everything can be covered in a small section. Some of the most important aspects will be covered.

We will continue with the same example created in the previous section. The accompanying code contains the FormBuilderCustomization project for this section.

Customizing the Prompts

Generally, FormFlow prompts for text based on property names. For example, when asking for a credit card number, FormFlow would prompt, "Please enter your Credit card Number" for simple properties and "Please select a ram" for enum-based properties. The prompt text can be customized using prompt and template attributes on top of properties.

The prompt for a simple property can be changed using the `Prompt` attribute:

```
[Prompt("Please enter your Credit card Number in international format !!")]
public string CreditCardNumber;
```

The prompt for an enum-based property can be changed using the `Template` attribute. `EnumSelectOne` allows a single selection and many more options. Check out the online documentation to find out more about these options. {||} helps show the list of choices for the property.

```
[Template(TemplateUsage.EnumSelectOne, "What is your preferred operating
system? {||}")]
public OperatingSystemOptions? OS;
```

Customizing the Order of Prompts

The prompts are displayed to the user in order of property declaration. Instead of displaying the prompts based on their order of declaration, the order can be customized using the `Field` method, as shown next. Instead of relying on the default prompt layout, individual fields have been used to define the order of prompts. In this example, the order of prompts will be ram, cpu, drive, screen, os, delivery:

```
public static IForm<FormFlowSimple> BuildForm()
        {
            return new FormBuilder<FormFlowSimple>()
                .Message("Welcome to NoteBook builder..")
                .Field(nameof(ram))
                .Field(nameof(cpu))
                .Field(nameof(drive))
                .Field(nameof(screen))
                .Field(nameof(OS))
                .Field(nameof(delivery))
                .Field(nameof(CreditCardNumber), IsCreditCard)
                .Build();
        }
```

Conditional Fields

By default, all properties are shown to the user by means of prompts. However, there are situations where a field should be shown based on a condition. For example, if the user has opted to use credit card as payment, then the credit card number prompt should be shown; otherwise, not. The second parameter of the `Field` method takes in a boolean true/false value. The field is prompted only if its value is true. By default, the value is true. A function is declared that checks the current state of the `delivery` property. If the value is the same as that of `Credit Card`, the prompt for the credit card number is shown. See here:

```
public static IForm<FormFlowSimple> BuildForm()
        {
            return new FormBuilder<FormFlowSimple>()
                .Message("Welcome to NoteBook builder..")
                .Field(nameof(ram))
                .Field(nameof(cpu))
                .Field(nameof(drive))
                .Field(nameof(screen))
                .Field(nameof(OS))
                .Field(nameof(delivery))
                .Field(nameof(CreditCardNumber), IsCreditCard)
                .Build();
        }
```

```
private static bool IsCreditCard(FormFlowSimple state)
{
    return state.delivery == Deliveryoptions.CreditCard;
}
```

When implementing a conditional field, the credit card number prompt is not shown to the user, as shown in Figures 8-5 and 8-6.

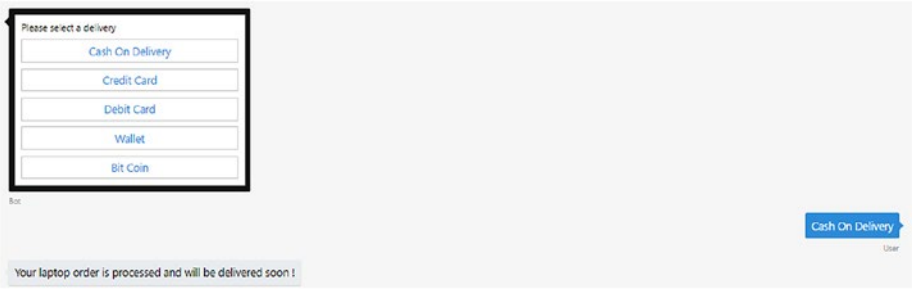

Figure 8-5. *Bot asking user for payment selection*



Figure 8-6. *Utilizing conditional fields based on certain conditions being true*

Summary

Bot framework provides multiple options for authoring conversational bots. Each option has its advantages and disadvantages. The Bot framework enables the writing of the entire logic within the `messagesController` API itself or allows you to divide logic within multiple dialogs and FormFlow. While dialog provides maximum control and flexibility when authoring bots, FormFlow makes authoring conversational bots faster and easier. Architects should evaluate their requirements and adopt an option. Moreover, these options can be used together to create a conversational bot. For example, a LUISDialog can be used in conjunction with FormFlow in a multi-dialog scenario. There are lots of moving pieces for both dialogs and FormFlow, and this chapter provides the beginners' introduction to them. Readers should also check for online documentation for changes to these elements with passage of time.

CHAPTER 9

Natural Language Processing

Natural language processing (NLP) is a field of computer science that is a subset of artificial intelligence and that helps computers understand human language as it is written and spoken. Until recently, it was humans who were trying to understand computer languages to talk to them in their way via scripting languages like BASH, JavaScript, and PowerShell and programming languages like Java, C#, and others. In recent times, a new phenomenon is gaining momentum—machines learning and understanding the language of humans.

Generally, computers are great for computational processing. If you tell a computer to find the square of a number as big as 25 digits, it will have an answer in less than a fraction of a second. However, if, at the same time, you tell it to find differences between two almost identical images, or to guess the emotions of a person in a picture, it can be difficult and challenging for them to have an answer immediately. On the contrary, humans will take minutes if not hours to find the same square number but only seconds to discern the emotions of the person in the picture or the differences between two images.

Artificial intelligence helps replicate the same neural functionality within computers so they can become capable of understanding and finding answers like emotions just like the human brain does. This could relate to vision, speech, emotions, or human language.

Every human has their own proficiency and vocabulary in written and spoken language. An English sentence conveying a message can be written in multiple ways while conveying the same idea. Everyone has their own way of expressing the same thing. A user searching for a flight for vacation will use different vocabulary, punctuation, and phrasing compared to any other individual. Understanding the idea and motive behind user-provided text in their own language is a difficult problem to solve, but NLP can help.

NLP helps by creating generic language models through which it is easier for computers to understand the idea and motive of user interactions that use written language. NLP helps make intelligent systems that can understand the intent behind a user's text and provide them with an appropriate response.

In this chapter, we will go through Microsoft Cognitive Services' offering that is specific to NLP, known as Language Understanding Intelligent Services (LUIS). Important concepts related to LUIS like intents and entities, along with the LUIS process of creating and publishing them, will be the core of the chapter. A sample bot solution will be undertaken that will interact with LUIS to find the motive behind a user's interaction to demonstrate the usage of LUIS with bots. MS Bot framework provides multiple framework features to interact with LUIS, which will also be discussed in depth in this chapter. Finally, best practices related to LUIS will also be provided.

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_9

Cognitive Services

Microsoft provides a suite of intelligent API services under an umbrella known as Cognitive Services. These services enable natural and contextual interaction by using tools that augment users' experiences via the power of machine-based intelligence. Cognitive Services provides a collection of powerful artificial intelligence algorithms for vision, speech, language, and knowledge. It helps build intelligent systems with powerful algorithms that use just a few lines of code and work across devices and platforms, such as iOS, Android, and Windows. More about Cognitive Services can be found at https://azure.microsoft.com/en-gb/services/cognitive-services/.

LUIS

The NLP implementation is known as LUIS. It offers a fast and effective way of adding language understanding to applications. It is a service exposed as a REST API, and users can consume this service from any platform. It provides an HTTP endpoint that will take in sentences sent by users, find the intention behind them, and identify the subject and objects in them. LUIS helps boost developers' productivity by providing a set of powerful tools in the form of dashboards and a web-based environment to define models and by exposing them through a simple user experience and a comprehensive set of APIs. These APIs are part of Azure Services, and users must have an active Azure subscription to consume these services. An account at https://www.luis.ai/ should also be created using your Microsoft (earlier Live) account. Luis.ai provided the browser-based interface to manage the lifecycle of LUIS applications.

LUIS is not aware of the business problem that you as a developer are trying to solve. Developers should make LUIS aware of the problem they are trying to solve by providing inputs to it. LUIS needs a few inputs to create intelligent, language-aware applications. These inputs in LUIS's terms are as follows:

- Intents
- Entities
- Utterances
- Features

Intents

The primary function of LUIS is to find intents from user-provided text. Users can provide any text with any vocabulary and punctuation. Developers should provide LUIS with the intents they are interested in. Intents are the motives, ideas, or goals of the users interacting with the application. These motives can be divided into two categories:

1. Searching for something – Examples of these motives are finding weather conditions next week in local city or town, finding information about flight tickets, etc.
2. Acting or taking action – Examples include booking a flight ticket, reserving a seat in a restaurant, etc.

Intents are verbs or actions that users are trying to execute using the application. A set of intents that correspond to the user's actions should be provided as inputs to LUIS. If you are building an application for a travel company, it might include intents like find a flight between two locations, book a flight, cancel a flight, reschedule a flight, find connected flights, and more.

Intents have scores in LUIS. It means that LUIS can identify multiple intents for the same utterance; however, the score will distinguish between them in terms of their strength. Intents with higher scores show a higher confidence level about its identification, while a low score denotes a lower confidence level.

Entities

Entities are equally important and should be fed into LUIS along with intents. It is good to know the user's intent, but it will fail miserably if an action cannot be taken on the identified intent. The action depends on the user-provided parameters for their intent. For example, if a user is booking a flight ticket, then flight information, source location, destination location, date of travel, class of ticket, and food preferences should be identified in order to take any meaningful action. LUIS can identify these entities from user-provided sentences. Entities are the core data elements for applications. Entities are like nouns, objects, and subjects that provide additional context for the meaning of a conversation and subsequent action. If a user typed *book a flight to Paris from London for December 25, 2017* the intent would be to book a flight, and the entities would be Paris, London, and December 25, 2017. These entities would represent source and destination locations along with the date.

Entities can be pre-built and provided by LUIS, like age, datetime, numbers, dimension, and so on, or they can be custom, which we will use heavily in this chapter. Custom entities are the ones defined by developers for their business problem and solution. Pre-built entities are already trained and do not need further training.

Utterances

The lexical meaning of utterance is "the action of saying or expressing something." After intents and entities have been defined, LUIS should be fed with sample utterances so as to train itself to identify those intents and entities. Just providing intents and entities to LUIS is not enough. LUIS needs to be trained to find them in incoming sentences. To enable the training of LUIS, sample utterances should be provided. Based on these, LUIS will adapt itself to understand the utterances and to find intents and entities based on them. Examples of utterances for the previous travel-agent example include "Book a flight from London to Paris tomorrow," "Book a flight from London to Paris today," "I want to fly on December 22, 2017 from Mumbai to Hongkong." Enough utterances should

be provided so that LUIS is able to identify the majority of a user's incoming text and to successfully identify appropriate intents and entities. In the end, it is utterances that come from users when they use applications.

Features

Features refers to additional metadata, traits, or attributes for an intent or entity. They help in improving the overall efficiency and effectiveness of the NLP algorithmn that runs behind the scene. Phrase lists can be added to features and will provide additional information while training LUIS; scores can be made significantly higher using them.

LUIS Development Lifecycle

LUIS is an Azure service with generic API endpoints to create LUIS applications. It provides a web-based user interface to create your own endpoints that can take user sentences as an input and output the intents and entities. The user's related endpoints are auto-generated when a LUIS application is published. These newly generated endpoints are customized based on the model, which is defined in terms of intents, entities, and utterances. Intelligent applications should consume this newly published REST API that is aware of your intents and entities.

However, before a model can be published as an endpoint, it should be created and trained. In this section, we will go through the process of creating a LUIS application and publishing it as endpoint for consumption.

The high-level process that should be followed to create a LUIS application is as follows:

- Create application
- Add intents
- Add entities
- Add utterances
- Train and test
- Publish

Create Application

The first step in using LUIS is to create an application. LUIS is available at `https://www.luis.ai`. Sign in to LUIS with an appropriate account and create a new app. The LUIS app is the logical boundary, and each app, when published, generates endpoints that can be consumed in bot applications (see Figure 9-1).

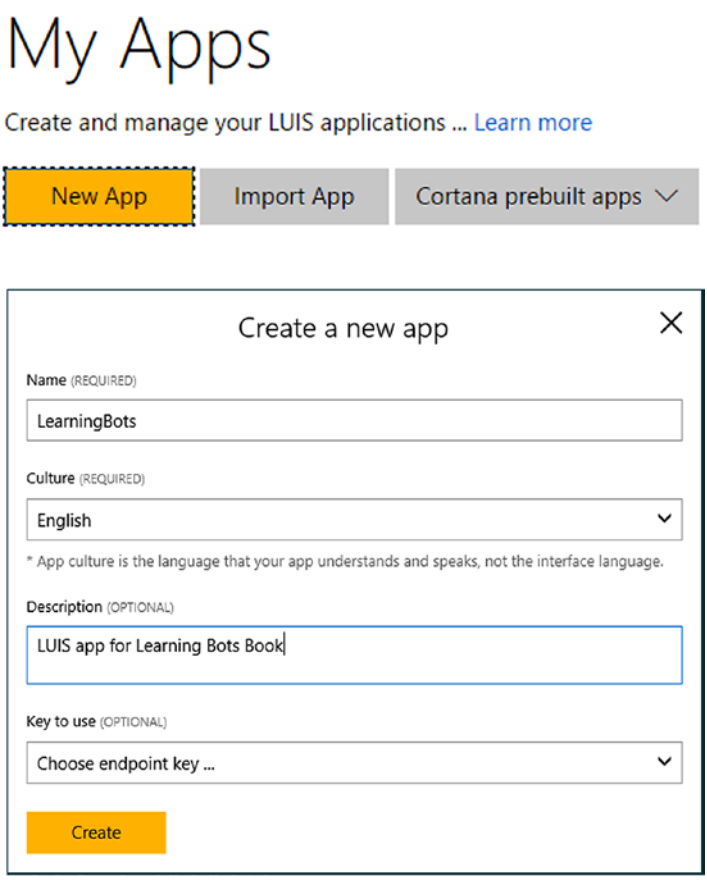

Figure 9-1. *Steps to create a new LUIS app*

Add Intents

After the app is provisioned, the first step should be creating intents. Open the recently created app and click Intents in the left panel, as shown in Figure 9-2.

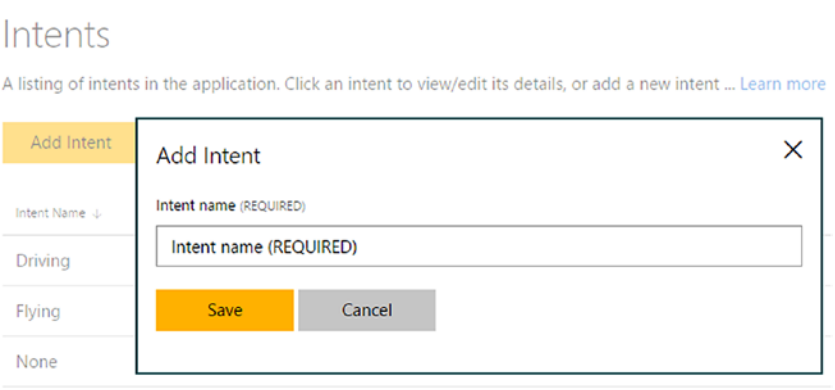

Figure 9-2. *Adding a new intent in LUIS app*

A *None* intent is created for all apps by default. There are two additional intents created that will be used in later sections of this chapter.

Add Entities

After intents have been defined, the next step should be creating entities. Entities can be the following:

- Simple: a generic entity
- Hierarchical: entities in a parent–child relationship. A parent can have multiple children (sub-entities).
- Composite: a compound of two or more separate entities combined, forming a composite and treated as a single entity. Composite entities are best suited to establishing and inferring relationships between multiple entities. For example, "fly from London to Paris via Globe Airways" has three simple entities—London, Paris, and Globe Airways. Without composite entities, LUIS would still be able to find these three entities, but there would be no way to establish and infer a relationship between them. Composite entities help us establish relationships among multiple entities.
- List: a customized list of entity values to be used as keywords or identifiers to recognize entities within utterances.

Click on Entities on the left panel to create new entities. This is shown in Figure 9-3.

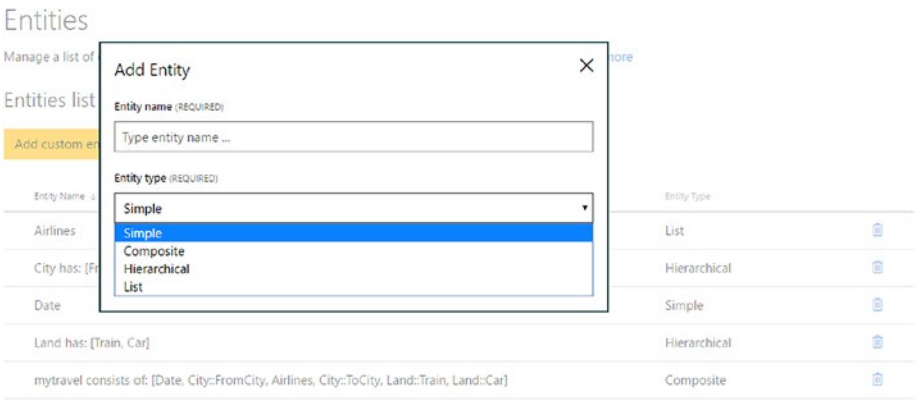

Figure 9-3. *Adding a new entity in LUIS app*

Add Utterances

Utterances help in identifying intents and entities with higher confidence. Each intent can be identified through multiple utterances. Utterances can be added using the user interface of an intent. Click on any intent and add utterances to it. While adding them, select the entities that are part of the utterances. This is shown in Figure 9-4.

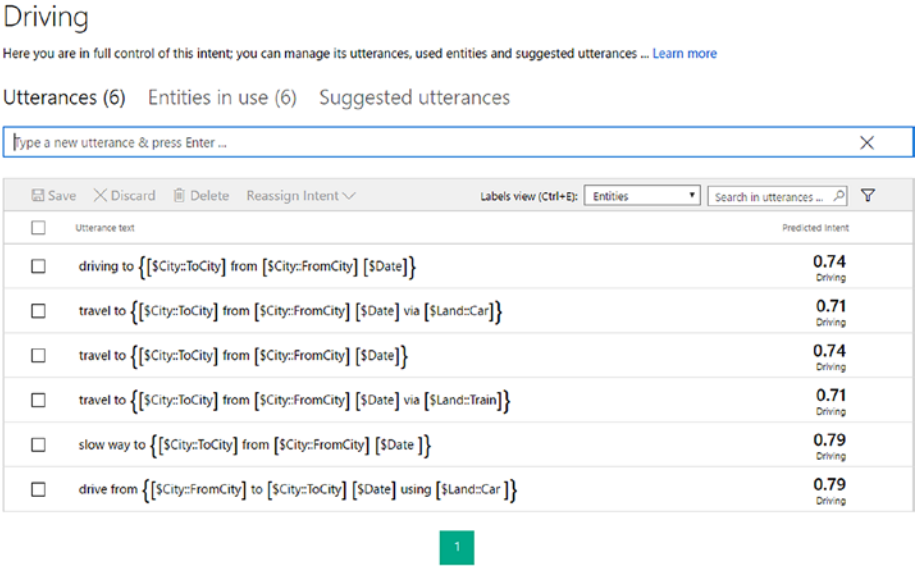

Figure 9-4. *Mapping intents and entities within LUIS app*

Train and Test

After intents, entities, and sufficient utterances are added, other LUIS components like features can be used to augment the performance of the overall model and increase predictability. However, after the model is completed, the next step is to train it. Training a model means to take the available intents, entities, features, and utterances and process them through LUIS-provided algorithms to build the neural networks that will help in identifying the intents and entities in incoming sentences. Developers just need to click on the "Train & Test" link available in the left panel to start the training process.

Testing can be done on a trained model by providing additional utterances and checking the intents and entities identified as output with their individual scores. If there is any deviation from the actual meanings, more utterances should be added, more features should be utilized, and the model should be retrained to understand any outliers and deviations. LUIS provides a testing feature for applications, shown in Figure 9-5.

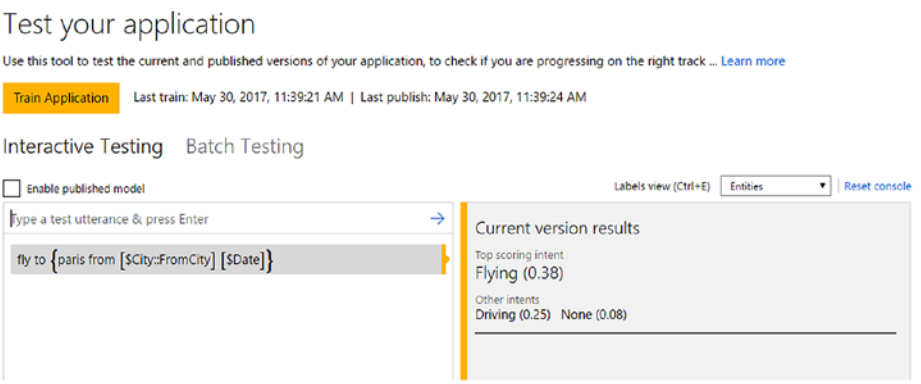

Figure 9-5. *Testing intents and entities using sample queries*

Publish

Once developers are confident about the predictability scores for the intents and entities after training and retraining the model, they can publish their app. This will publish the endpoints necessary for the application to consume.

To publish your application, you need a key and application ID. Publishing will generate a new endpoint and URL. This URL should be used by the application to interact with LUIS. The format of the URL is as follows:

```
https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/<<ApplicationID>>?
subscription-key=<<Subscription key>>&timezoneOffset=0&verbose=true&q=
```

Both subscription key and application ID must be provided by developers.

Click on the "Publish App" link on the left panel to open the Publish window. This is shown in Figure 9-6.

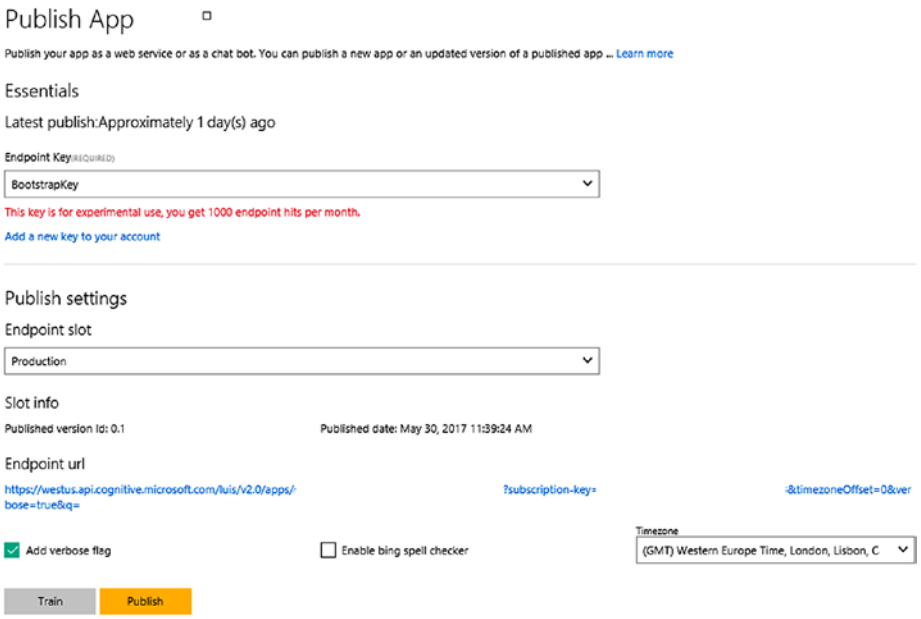

Figure 9-6. *Publishing LUIS app for use in bot application*

Sample Application

We will now create a sample application for LUIS named LearningBots and then create a bot that will consume LUIS's services using the custom endpoint published. The goal of this sample is to understand LUIS modeling and the options available to integrate with a bot. The same sample will be implemented using different features provided by MS Bot framework. The purpose of the bot is to show how LUIS can help implement intelligent bots and find intents and entities based on which the bot can take appropriate actions. For sample purposes, two intents are created, as shown in Figure 9-7, and the None intent is available by default. None intent is used when no other intent matches incoming data.

- Flying – used when flying from one city to another using a flight
- Driving – used when either driving on the road or riding a train from one city to another)



Figure 9-7. *Intents created for the sample application*

The entities defined for the sample application are as follows:

- FromCity and ToCity are simple entities for capturing information about source and destination cities. This is used for both flying and driving intents.
- Train and Car are simple entities for capturing information about the medium of transportation on the ground. This is used for the driving intent.
- Date is a simple entity for capturing input about the date of travel. This could have been a prebuilt entity as well.
- A list entity names airlines. The list includes "Airline1," "Abc Airline," "World Airlines," and Globe Airways." It is solely used for the flying intent.
- Two hierarchical entities, City and Land, with City containing FromCity and ToCity simple entities. City is used for both intents while Land is used only for driving intent.
- A composite entity MyTravel comprising Date, Airlines, City, and Land entities. City and Land in turn have further child entities.

All entities for the sample application are shown in Figure 9-8.

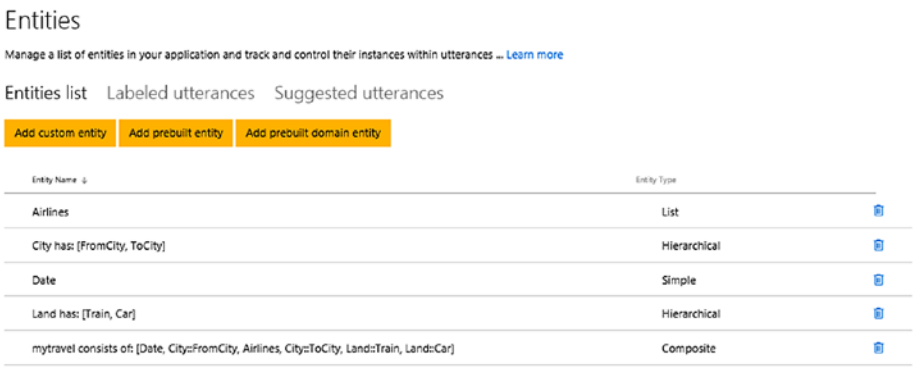

Figure 9-8. *Sample entities for bot application*

Utterances act as training data and help find the intents by training the LUIS model. Utterances as shown in Figure 9-9 are used for the flying intent.

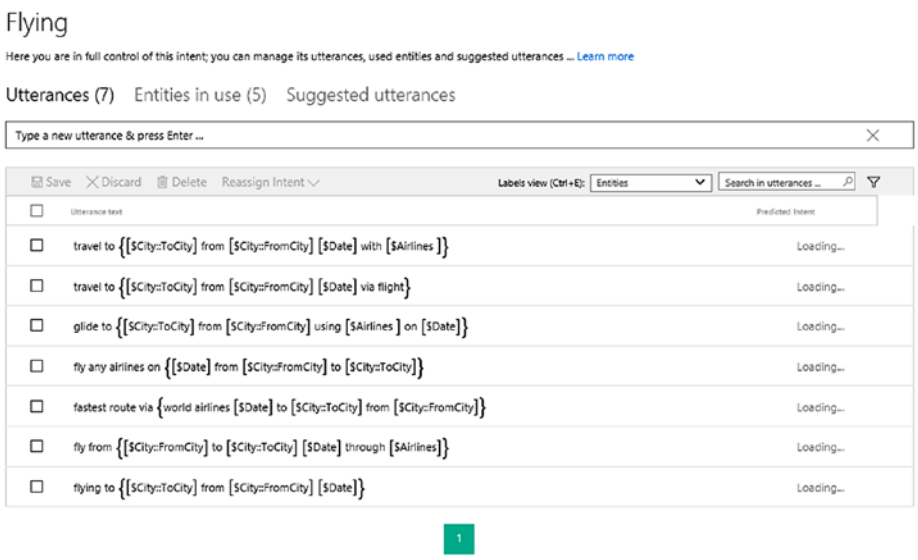

Figure 9-9. *Utterances mapping entities and flying intent for sample application*

Utterances as shown in Figure 9-10 are used for the driving intent.

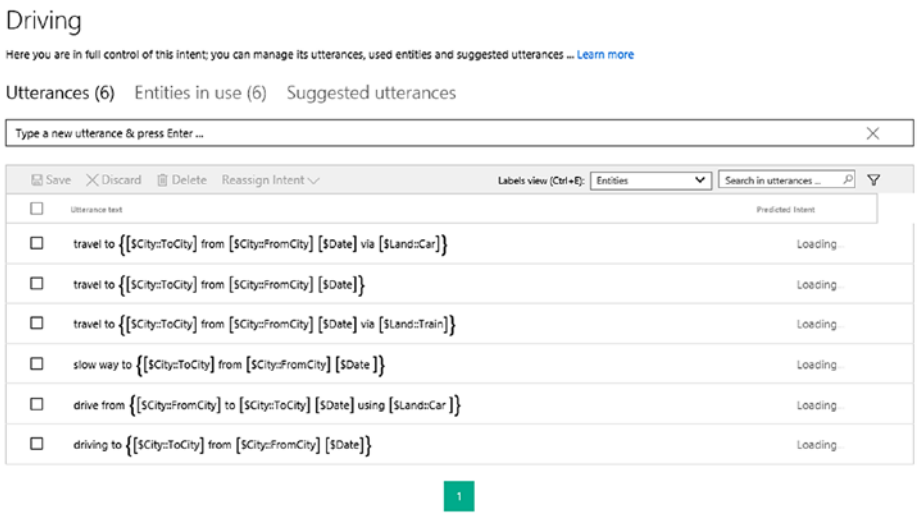

Figure 9-10. *Utterances mapping entities and driving intent for sample application*

Readers should notice that each utterance has multiple brackets representing composite, hierarchical, list, and simple entities.

The LUIS application should be trained and published such that it can be consumed by bots using REST endpoints. Publishing would generate the endpoint URL. Copy the URL and use it in a web browser by appending query text so as to view the JSON document that is generated as a LUIS response containing the intents and entities. This is shown in Figure 9-11. Do not worry if the JSON document does not contain all the intents and entities related to your application. At this stage, we are more interested in getting the JSON structure obtained from LUIS. This JSON document would eventually be needed in order to generate a C# class from it. Copy the JSON document to the clipboard. Replace the subscription key and application ID with valid values.

```
https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/<<Application
ID>>?subscription-key=<<Subscription key>>&timezoneOffset=0&verbose=true&q=t
ravel to london from edinburgh tomorrow
```



Figure 9-11. *JSON payload used to generate plain old simple C# class*

Creating Intelligent Bots

Bots need to invoke HTTP capabilities to consume LUIS endpoint services. Furthermore, LUIS provides a response comprising intents and entities in JSON format. Any client capable of parsing JSON documents can take advantage of LUIS. Although it is possible to parse raw JSON, it is a better practice to use objects deserialized from JSON documents.

By now, we know that bots can be created with or without dialogs. In this section, we will create two bots—one using dialogs and one without dialogs. Both bots would consume the same LUIS application.

Creating Intelligent Bots Without Dialogs

In this approach of creating a bot with LUIS without dialogs, the developer is responsible for invoking the LUIS endpoint explicitly. Developers are also responsible for sending requests and parsing JSON responses from LUIS. They can create objects by deserializing the JSON document or they can work with raw JSON. Working with objects is much easier than parsing raw JSON. The developer also needs to write the logic and code for how he or she wants to redirect the code execution based on incoming intents and entities. Generally, these are not needed when using dialogs. This approach should generally be avoided, especially when MS Bot framework provides inherent support to work with LUIS using dialogs. This example will demonstrate the process of invoking the LUIS endpoint, sending requests containing sentences from the user, obtaining the response containing intents and entities, and displaying them within the bot. Once you have entities and intents available, you can split the code into multiple paths and take necessary action.

Create a new bot project. Add a new class structure representing JSON documents from LUIS. Create or open a C# class file in Visual Studio, then go to Edit ➤ Paste Special ➤ Paste JSON As Classes. The JSON document should already be there in the clipboard. This will generate the entire class structure. The class should similar to that shown in Figure 9-12.

```
public class LuiSObject
{
    0 references
    public string query { get; set; }
    0 references
    public Topscoringintent topScoringIntent { get; set; }
    2 references
    public Intent[] intents { get; set; }
    1 reference
    public Entity[] entities { get; set; }
    0 references
    public Compositeentity[] compositeEntities { get; set; }
}
1 reference
public class Topscoringintent...
2 references
public class Intent
{
    1 reference
    public string intent { get; set; }
    2 references
    public float score { get; set; }
}
1 reference
public class Entity
{
    1 reference
    public string entity { get; set; }
    1 reference
    public string type { get; set; }
    0 references
    public int startIndex { get; set; }
    0 references
    public int endIndex { get; set; }
    0 references
    public float score { get; set; }
    0 references
    public Resolution resolution { get; set; }
}
1 reference
public class Resolution...
1 reference
public class Compositeentity...
1 reference
public class Child...
```

Figure 9-12. *Class generated from JSON payload, used in bot application*

Open and replace the following code in `MessagesController.cs` within `Controllers` folder:

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

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

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

with the following:

```
ConnectorClient connector = new ConnectorClient(new Uri(activity.
ServiceUrl));

LuiSObject Data = new LuiSObject();
StringBuilder sb = new StringBuilder();
using (HttpClient client = new HttpClient())
{
       string RequestURI = "https://westus.api.cognitive.microsoft.com/luis/
       v2.0/apps/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx?subscription-key=xx
       xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx&verbose=true&timezoneOffset=0&q=" +
       activity.Text;

       HttpResponseMessage msg = await client.GetAsync(RequestURI);

       if (msg.IsSuccessStatusCode)
          {
              var JsonDataResponse = await msg.Content.ReadAsStringAsync();
              Data = JsonConvert.DeserializeObject<LuiSObject>(JsonData
              Response);

Intent itemsMax = Data.intents.Where(x => x.score == Data.intents.Max(y =>
y.score)).First();

              sb.Append("intent is " + itemsMax.intent);

              foreach (var d in Data.entities) {
              sb.Append("Entity type " + d.type + " has value " + d.entity +
              Environment.NewLine);
            }
       }
}
```

```
Activity reply = activity.CreateReply($"You sent {sb.ToString()} ");
await connector.Conversations.ReplyToActivityAsync(reply); )
```

1. Create `ConnectorClient` object. This is the main object through which a bot can communicate with channels and eventually users. Also, create an instance of newly generated `LuiSObject` class. This is needed for populating values from the deserialized JSON document received as a response from LUIS.

   ```
   ConnectorClient connector = new ConnectorClient(new
   Uri(activity.ServiceUrl));

   LuiSObject Data = new LuiSObject();
   StringBuilder sb = new StringBuilder();
   ```

2. Create an instance of `HttpClient` object. This is needed to invoke the LUIS endpoint. The request is sent using the `GetAsync` method from this class. It should be provided with the complete URL to the LUIS endpoint along with text from the user. The user text is available from the `text` property of the `activity` object.

   ```
   HttpClient client = new HttpClient()
   string RequestURI = "https://westus.api.cognitive.
   microsoft.com/luis/v2.0/apps/xxxxxxxx-xxxx-xxxx-
   xxxx-xxxxxxxxxxxx?subscription-key=xxxxxxxxxxxxxxxxx
   xxxxxxxxxxxxxxx&verbose=true&timezoneOffset=0&q=" +
   activity.Text;

   HttpResponseMessage msg = await client.
   GetAsync(RequestURI);
   ```

3. If the LUIS endpoint responds with success code (200 OK), get the JSON content and deserialize it a into `LuiSObject` instance.

   ```
   var JsonDataResponse = await msg.Content.
   ReadAsStringAsync();
          Data = JsonConvert.DeserializeObject<LuiSObje
          ct>(JsonDataResponse);
   ```

4. By now, the `LuiSObject` instance has all the identified intents and entities from LUIS. It is up to the developer now how he wants to use them. For the purpose of this sample, the intent with the highest score is extracted using a LINQ query, and all entities are looped through to fill the `StringBuilder` object.

```
Intent itemsMax = Data.intents.Where(x => x.score ==
Data.intents.Max(y => y.score)).First();

sb.Append("intent is " + itemsMax.intent);

foreach (var d in Data.entities) {
        sb.Append("Entity type " + d.type + " has
value " + d.entity + Environment.NewLine);
}
```

5. Finally, the content from the `StringBuilder` object is sent as the response to the user from the bot.

Figure 9-13 uses Bot Emulator to show an interaction with this bot demonstrating the driving intent.

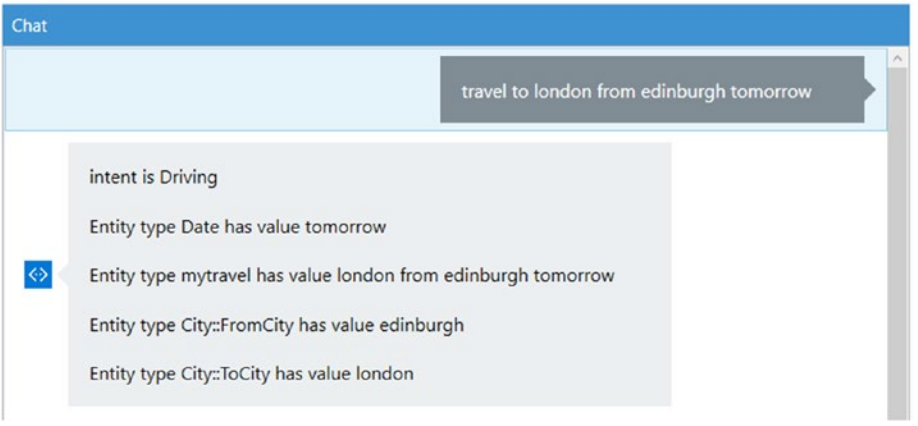

Figure 9-13. *Interacting with sample LUIS-based bot using Bot Emulator (driving intent)*

Here, the intent is driving and entities found were tomorrow, Edinburgh, London, and a composite entity "London from Edinburgh tomorrow."

Figure 9-14 shows the flying intent getting used in Bot Emulator.

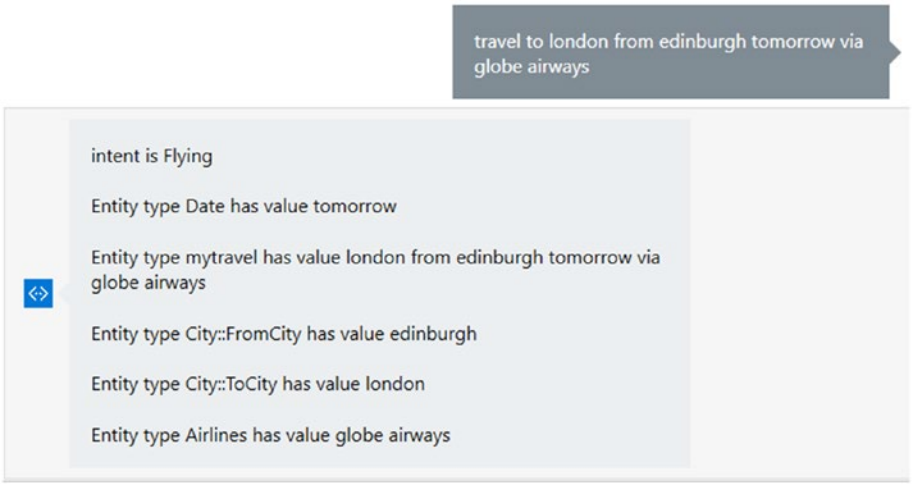

Figure 9-14. *Interacting with sample LUIS-based Bot using Bot Emulator (flying intent)*

Here, the intent is flying and entities found were tomorrow, Edinburgh, London, globe airways, and a composite entity "London from Edinburgh via globe airways."

Creating Intelligent Bots with Dialogs

As seen in the previous chapter, a dialog models a conversational process where the user exchanges a series of messages with the bot. The MS Bot framework comes with an out of the box dialog, `LuisDialog`, that is integrated with LUIS. Using this dialog helps the developer by eliminating any need to write code explicitly for invoking a LUIS endpoint. They do not even have to think about JSON parsing or creating classes that can help with deserializing JSON. These things are automatically taken care of by `LuisDialog`. This is the recommend approach when working with and integrating LUIS with Bots. This example will demonstrate the process of creating a bot using LuisDialog, handling intents and entities, and passing control to sub-dialogs for each intent. It will also show how to use global handlers using the Scorable interface.

1. Create a new bot project in Visual Studio. The first step should be to update the Bot Builder assemblies with the latest version. Open Package Manager Console from Tools ➤ NuGet Package Manager (Figure 9-15) and execute the following:

   ```
   Install-Package Microsoft.Bot.Builder
   ```

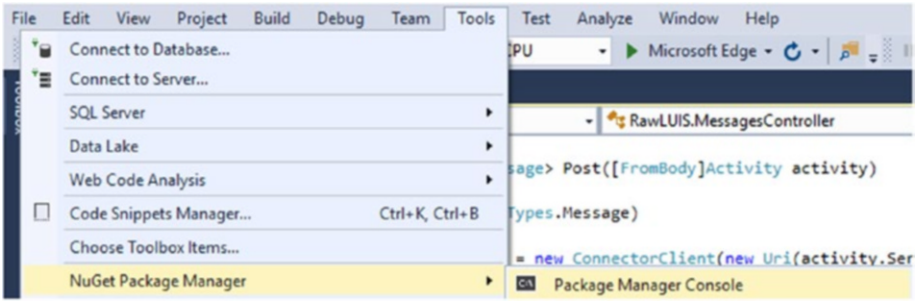

Figure 9-15. *Installing Bot SDK NuGet packages in Visual Studio solution*

2. After the package is updated, create a new class file that would represent the root dialog for the bot. This dialog will also be responsible for integrating with LUIS.

3. Name the class `LearningBotDialog`. Add `Serializable` attribute to it.

4. Add necessary `using` statements, including the following:

   a. `Microsoft.Bot.Builder`

   b. `Microsoft.Bot.Builder.Luis`

   c. `Microsoft.Bot.Builder.Luis.Models`

5. `LuisDialog` should inherit from the `LuisDialog<object>` class.

6. Add the `LuisModel` attribute to the class. Also, provide the LUIS app ID and subscription ID as parameters to `LuisModel`. `LuisDialog` will use both app ID and subscription ID to invoke the LUIS endpoint internally.

```
namespace LUISDialog
{
    [LuisModel("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" ,
    "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx")]
    [Serializable]
    public class LearningBotDialog :LuisDialog<object>
    {
    }
```

7. Write a method for each intent that is available in the LUIS application, with each method decorated with the `LuisIntent` attribute. This attribute takes a string parameter representing the name of the intent. In effect, each of these methods acts as a handler for the intent it is associated with. This association is made using the `LuisIntent` attribute. Since we had three intents, there are three methods, each associated with an intent.

```
[LuisIntent("")]
public async Task None(IDialogContext context,
LuisResult result)
{
    await context.PostAsync("I do not understand
    you!!");
    context.Wait(MessageReceived);
}

[LuisIntent("Flying")]
public async Task Fly(IDialogContext context,
LuisResult result)
{
    await context.Forward(new FlyingDialog(),
    AfterMessagehandler, result, CancellationToken.
    None);
}

[LuisIntent("Driving")]
public async Task Ground(IDialogContext context,
LuisResult result)
{
    await context.Forward(new GroundDialog(),
    AfterMessagehandler, result, CancellationToken.
    None);

}
```

The `None` method sends a response directly to the user because it does not match any custom intent, while the `Fly` and `Ground` methods associated with the flying and driving intents forward the request to another dialog, and, while forwarding the request, it sends the `LuisResult` object along with it. It is to be noted that the `LuisResult` class is available from the `Microsoft.Bot.Builder.Luis.Models` namespace. This class contains all intents and entities identified by LUIS, along with their scored.

It is important to understand here that users' requests do not land in any of the methods shown before. These methods are invoked after LUIS has processed and identified the intents and entities. The request from the user is passed on to the `LuisDialog` base class, which connects and invokes the LUIS endpoint along with the user-provided content. LUIS identifies intents and entities and sends the response back to the `LuisDialog` base class, which fills up the `LuisResult` object. The `LuisDialog` base class then, based on the `LuisIntent` attribute value, identifies the appropriate method handler to fire it. It also sends the `LuisResult` object to the handler as a parameter. In effect, the custom `Dialog` class inheriting from the `LuisDialog` base class acts as a state machine, and based on the returning intent value an appropriate handler is invoked.

8. There is an additional method, `AfterMessageHandler`, that is invoked after the sub-dialogs have finished their processing.

```
private async Task AfterMessagehandler(IDialogContext
context, IAwaitable<bool> result)
{
    bool message = await result;

    context.Wait(MessageReceived);
}
```

The `LuisDialog` class provides `MessageReceived`, a generic implementation of message handler. The `Context.Wait` method ensures that this handler is invoked, which in turn connects and queries LUIS with the user text, processes incoming intents and entities, and invokes the appropriate intent handler.

Both `Fly` and `Ground` methods forward requests to their corresponding sub-dialogs—`FlyingDialog` and `GroundDialog`. Next, it's time to implement them.

9. Create a new class and name it `FlyingDialog`. Add the `Microsoft.Bot.Builder`, `Microsoft.Bot.Builder.Luis`, and `Microsoft.Bot.Builder.Luis.Models` namespaces and inherit the newly created class from the `IDialog<bool>` interface. Also, attribute the class with the `Serializable` attribute. Note that this class does not implement or inherit the `LuisDialog` class. This is a general dialog that we saw in the previous chapter. Implement the `StartAsync` method, available from the `IDialog` interface, and a handler after receiving inputs from its parent dialog. This dialog is invoked when the flying intent is identified by LUIS.

```
namespace LUISDialog
{
    [Serializable]
    public class FlyingDialog : IDialog<bool>
    {
        public async Task StartAsync(IDialogContext
        context)
        {
            await context.PostAsync("Thanks for
            booking with us!! please confirm your
            details");
            context.Wait<LuisResult>(BookFlightTrip);
        }

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

            StringBuilder sb = new StringBuilder();

            var itemsMax = message.Intents.Where(x
            => x.Score == message.Intents.Max(y =>
            y.Score)).First();
            sb.Append("Intent is " + itemsMax.Intent
            + Environment.NewLine + "\n");

            foreach (var d in message.Entities)
            {
                sb.Append("Entity type " + d.Type + "
                has value " + d.Entity + Environment.
                NewLine + "\n");
            }

            await context.PostAsync(sb.ToString());)

            context.Done<bool>(true);
        }
    }
}
```

Within the `StartAsync` method, the bot waits for user inputs, which in this case come from the parent `learningBotsDialog` dialog. The handler `BookFlightTickets` is invoked when `learningBotsDialog` forwards the request to `FlyingDialog`.

Within `BookFlightTickets`, the `LuisResult` object is looped and both intents and entities are extracted. Finally, they are returned as text to the user using the `StringBuilder` object. It is to be noted here that within this method actions like the actual booking of tickets can be executed.

When using multiple dialogs and weaving them together explicitly, a call to `context.Done` should be made to flag that the current dialog has finished its processing and that it can be removed from the stack.

10. Create a new class, `GroundDialog`, similar to `FlyingDialog`. Add the `Microsoft.Bot.Builder`, `Microsoft.Bot.Builder.Luis`, and `Microsoft.Bot.Builder.Luis.Models` namespaces and inherit the newly created class from the `IDialog<bool>` interface. Also, attribute the class with the `Serializable` attribute. Implement the `StartAsync` method, available from the `IDialog` interface, and a handler after receiving inputs from its parent dialog. This dialog is invoked when the driving intent is identified by LUIS.

```
namespace LUISDialog
{
    [Serializable]
    public class GroundDialog : IDialog<bool>
    {
        public async Task StartAsync(IDialogContext
        context)
        {
            await context.PostAsync("Thanks for
            booking with us!! please confirm your
            details");
            context.Wait<LuisResult>(BookGroundTrip);
        }

        private async Task
        BookGroundTrip(IDialogContext context,
        IAwaitable<LuisResult> result)
        {
            var message = await result;
            StringBuilder sb = new StringBuilder();

            var itemsMax = message.Intents.Where(x
            => x.Score == message.Intents.Max(y =>
            y.Score)).First();
            sb.Append("Intent is " + itemsMax.Intent
            + Environment.NewLine + "\n");
```

```
                foreach (var d in message.Entities)
                {
                    sb.Append("Entity type " + d.Type + "
                    has value " + d.Entity + Environment.
                    NewLine + "\n");
                }
                await context.PostAsync(sb.ToString());
                context.Done<bool>(true);
            }
        }
    }
```

11. Open and replace the following code in `MessagesController.cs` within the `Controllers` folder:

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

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

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

with the following:

```
await Conversation.SendAsync(activity, MakeRoot);
```

Here, the `MakeRoot` handler is invoked by MS Bot framework and then starts the process of creating the root dialog.

12. Implement the `MakeRoot` method in the `MessagesController.cs` class. This method is responsible for creating the root dialog and putting it on top of the dialog stack. The root dialog for the current sample is `LearningBotDialog`.

```
private static IDialog<object> MakeRoot()
{
    return Chain.From(() => new LearningBotDialog());
}
```

13. The final step in this example is to implement global handlers. It is possible to implement generic bot features, like responding to Hi, Hello, help, and so forth, using LUIS by creating a greetings intent and populating it with appropriate utterances. However, generally users input these ancillary keywords throughout a conversation, so it's important that even after such interruptions bots can continue with their conversation from where they left off. Scorable helps in implementing such functionality, which we already saw in the previous chapter.

14. Implement a class, in this case `GreetingsScorable`, implementing all methods of the `ScorableBase` abstract class. The Bot framework always checks for scorables before passing the request to the root dialog. The first method that Bot framework invokes is the `PrepareAsync` method. This method returns the actual text message back to the Bot framework to show which intent it wants to handle these messages from the user. The Bot framework calls the `HasScore` method, which returns true, to flag that this scorable is interested in getting invoked when any ancillary text is submitted by the user. If the `HasScore` method returns true, the `GetScore` method is invoked to evaluate the score. Since we have only one `Scorable` class, the `GetScore` method returns 1.0. The score from the `GetScore` method can range from 0 to 1. Eventually, the `PostAsync` method is invoked, which in turn calls `HelpDialog` and passes the text to be sent to the user. `HelpDialog` responds back to the user and removes itself from the dialog stack. For more details on scorables, please refer to the previous chapter.

```
namespace LUISDialog)
{
    public class GreetingsScorable:
    ScorableBase<IActivity, string, double>
    {

        private IDialogTask _task;
        public GreetingsScorable(IDialogTask task)
        {
            SetField.NotNull(out _task, nameof(task),
            task);
        }
        protected override Task DoneAsync(IActivity
        item, string state, CancellationToken token)
        {
            return Task.CompletedTask;
        }
```

```
protected override bool HasScore(IActivity
item, string state)
{
    return state != null;
}

protected override double GetScore(IActivity
item, string state)
{
    return 1.0;
}

protected override async Task
PostAsync(IActivity item, string state,
CancellationToken token)
{
    var message = item as IMessageActivity;

    if (message != null)
    {
        var incomingMessage = message.Text.
        ToLowerInvariant();
        var messageToSend = string.Empty;

        if (incomingMessage == "hello")
            messageToSend = "Thanks for
            pinging me!! provide details
            about your travel";

        if (incomingMessage == "thank you")
            messageToSend = "You are welcome
            and can always return back for
            further queries and travel
            bookings!";

        if (incomingMessage == "goodbye")
            messageToSend = "See you later
            but remember to get back if you
            want to travel again!!";

        if (incomingMessage == "hi")
            messageToSend = "Thanks for
            pinging me!! provide details
            about your travel";

        if (incomingMessage == "help")
            messageToSend = "Please provide
            from and To location, date of
```

```
                    travel to start the conversation
                    !!";

                var abc = new
                HelpDialog(messageToSend);
                var interruption = abc.Void<object,
                IMessageActivity>();
                this._task.Call(interruption, null);
                await this._task.PollAsync(token);
            }
        }

        protected override async Task<string>
        PrepareAsync(IActivity item,
        CancellationToken token)
        {

            var message = item.AsMessageActivity();

            if (message != null && !string.
            IsNullOrWhiteSpace(message.Text))
            {
                var msg = message.Text.
                ToLowerInvariant();

                if (msg == "hello" || msg == "thank
                you" || msg == "goodbye" || msg ==
                "hi" || msg == "help")
                {
                    return message.Text; )
                }
            }

            return null;

        }
    }
}
```

15. Implement the HelpDialog dialog as a normal general dialog that implements the IDialog<object> interface and its sole method, StartAsync.

```
namespace LUISDialog
{
    [Serializable]
    public class HelpDialog  : IDialog<object>
```

```
    {
        private readonly string _messageToSend;

        public HelpDialog(string message)
        {
            _messageToSend = message;
        }

        public async Task StartAsync(IDialogContext
        context)
        {
            await context.PostAsync(_messageToSend);
            context.Done<object>(null);
        }

    }
}
```

16. The final change needed is to stitch `Scorable` into the request pipeline. This is done using the `ContainerBuilder` IOC container available from Autofac. This change should be done in the `Application_Start` method of `Global.asax`. Ensure that this function has the following code:

```
GlobalConfiguration.Configure(WebApiConfig.Register);
var builder = new ContainerBuilder();
builder.RegisterType<GreetingsScorab
le>().As<IScorable<IActivity, double>>().
InstancePerLifetimeScope();
builder.Update(Conversation.Container);
```

    A new instance of `ContainerBuilder` is created, and `GreetingScorable` is registered with it. Implementing this functionality in the `Application_Start` method ensures that the effect is application wide and for all requests.

17. Interacting with this bot using Bot Emulator is shown next in Figures 9-16 and 9-17.

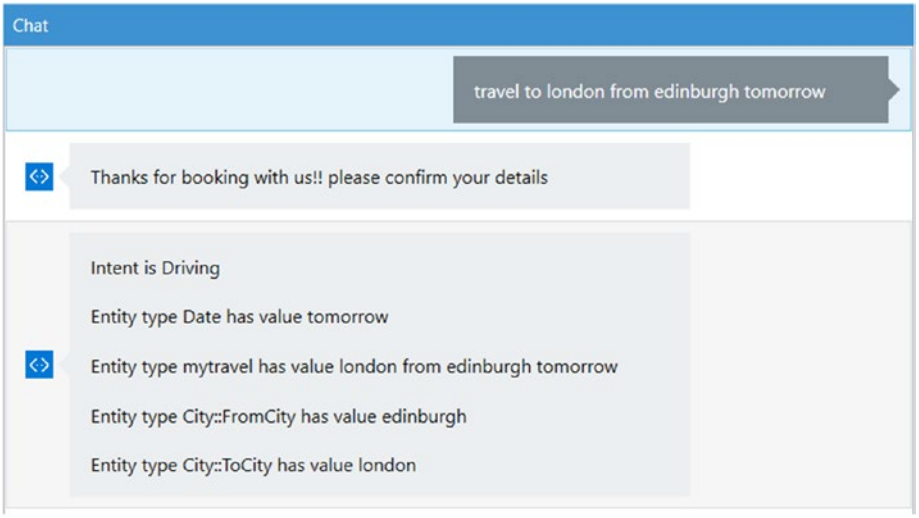

Figure 9-16. *Interacting with sample LUIS-based bot using Bot Emulator (driving intent)*

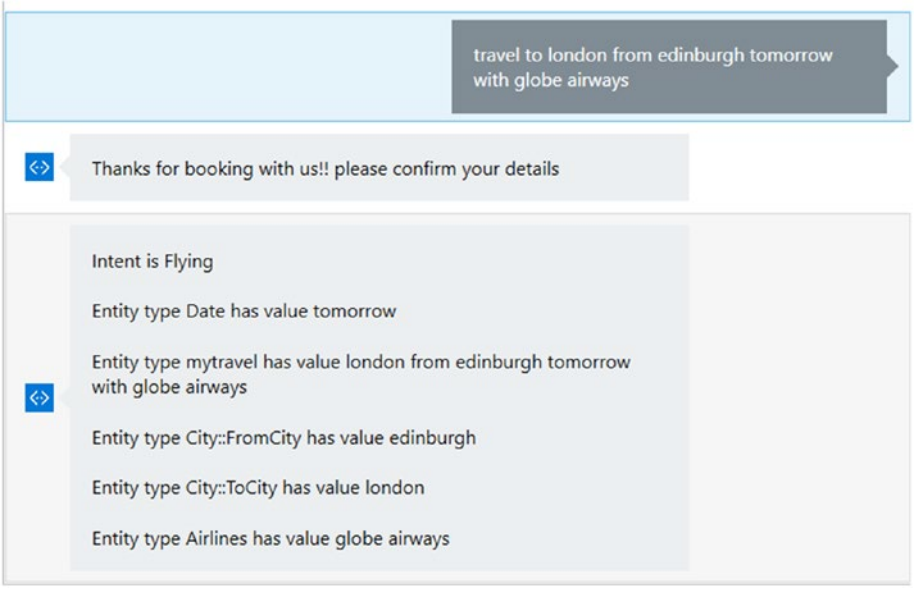

Figure 9-17. *Interacting with sample LUIS-based bot using Bot Emulator (flying intent)*

Here, the intent is driving and entities found were tomorrow, Edinburgh, London, and a composite entity, "London from Edinburgh tomorrow."

Here, the intent is flying and entities found were tomorrow, Edinburgh, London, globe airways, and a composite entity, "London from Edinburgh with globe airways."

The use of scorables is shown in Figure 9-18.

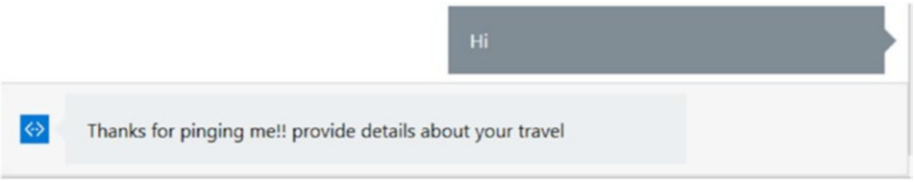

Figure 9-18. *Usage of scorables in Bot framework*

When the user types *Hi*, *help*, *goodbye*, *hello*, and *thank you*, the scorable kicks in and takes care of the request but also ensures that any conversation that was interrupted because of it continues from where it left off.

Summary

Bots are gaining prominence because of two important factors—availability on multiple channels and ability to converse with users in their natural language form. Bots should be intelligent enough to understand the meaning of inputs provided by the user. This makes the bot intelligent and also ensures that users are satisfied using the bot. Without intelligence, bots will simply become a static search interface that takes commands in an exact static format. In this chapter, LUIS was introduced, which is part of Azure Cognitive Services. It helps in defining entities and intents, which subsequently can be consumed by bots. Bot framework provides constructs that help in easily wiring bots with LUIS endpoints. There are also advanced features available in Bot framework, like scorables, which help intercept asides within conversations and return back to the original conversation. LUIS and bots go hand-in-hand, and writing bots cannot be envisaged without incorporating LUIS.

CHAPTER 10

Azure Cognitive Services

Bots can be broadly divided into two categories: chat bots and smart bots. Chat bots respond using a predefined set of rules and hence the responses are limited. For example, if you build a bot for reserving a table in a restaurant, the bot would always ask the basic questions of date and time, number of people, and seating preference (indoor/outdoor). If you throw a random request at the chat bot, it might not respond with a meaningful message or might just respond with a generic message. Smart bots are more intelligent. They work with a wide variety of information and generate more human-like responses. Smart bots are designed to learn from the conversation and provide more-useful answers as the conversation progresses, leaving the impression of a human-to-human conversation. Bots can be designed to create smarter responses by using cutting-edge artificial intelligence algorithms. Authoring AI algorithms is a complex task, requiring a varied skill set and lots of analysis to build an AI algorithm that can perform tasks like natural language processing and sentiment analysis or generate recommendations. Microsoft Cognitive Services provides a basket of AI algorithms that can be integrated into any application. These algorithms were developed by an expert team and cater across the fields of computer vision, speech, text analysis, natural language processing, knowledge extraction, and web search. In this chapter, we will learn to build smarter bots using Microsoft Cognitive Services and come to understand the capabilities of the evolving list of powerful AI algorithms.

The following topics will be discussed in this chapter:

- Introduction to Microsoft Cognitive Services
- Getting started, APIs, language support
- Building a Bing web search bot
- Building an OCR bot

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_10

Introduction to Microsoft Cognitive Services

Microsoft Cognitive Services is an exhaustive list of intelligent APIs that can be easily integrated into any type of application. Formerly known as Project Oxford, Microsoft Cognitive Services is built on top of Azure Machine Learning (ML). Microsoft Cognitive Services contains highly complicated, state-of-the-art, intelligent ML algorithms exposed as uniform and simple-to-use REST APIs and available as SDKs for a few languages. REST APIs can be used in any type of application written in any language just by adding a few lines of code. With Cognitive Services, Microsoft is aiming to increase the productivity of every individual and organization by allowing developers to build applications with rich intelligence by which applications can hear, see, speak, and think like humans. The goal is to simplify and merge the very complicated ML-based algorithms into the world of mainstream computing, thus bringing the benefits of AI to every individual.

Microsoft Cognitive Services lets you build all the smartness into the bot application, which can differentiate your bots from the rest while all the complexity is handled within the APIs. Bots can integrate with Microsoft Cognitive Services to provide a personalized and rich conversation experience for the user. They allow you to perform complex operations like image processing, pattern matching, recommendations, speech-to-text, tagging images, and so on using simple API calls. Microsoft Cognitive Services is broadly divided into vision, speech, language, knowledge, and search APIs. Each of these categories contains discrete APIs catering to various needs. While you are designing your bots for intelligence or personalization, it is recommended you go through the following exhaustive list of APIs to see which one fits your needs or which among these can bring a more personalized and rich conversation experience to your bot. For example, if I'm building a bot that automatically places an order for my monthly list of prescribed medicines, I would try to use OCR to read text from digital prescriptions, handwriting recognition to read text from handwritten prescriptions, speech-to-text to accept voice inputs from the user, and text analytics and Bing search APIs to provide better information on the entities extracted from the user's conversation.

Before you know which APIs to use, you should know the breadth of services provided by MCS. Table 10-1 lists what each API provides.

Table 10-1. *Microsoft Cognitive Services API Classification*

Vision	Computer Vision API	This API helps you extract actionable information from the images based on the uploaded image or the image URL. The Vision API can analyze visual data and extract textual/objects/analytics in the form of a JSON response. For example: • Identifying objects and living beings in an image • Categorizing images based on visual aspects into groups like Indoor, Dark, Sky. Vision API supports 86 different categories. • Identifying images as clipart, non-clipart, or ambiguous • Recognizing faces, gender, and age in images with human faces • Generating descriptions of an image • Recognizing handwritten text and converting it into digital format • Generate thumbnails from an image for different form factors • Perceive color schemes in an image • Flag adult and racy content Extracting text from image using optical character recognition (OCR)
	Emotion API	This API can be used to identify emotions like happiness, sadness, surprise, anger, fear, contempt, disgust, or neutral in an uploaded image or URL with human faces. Each human face will also have a bounding box to help relate the response. The emotions for each human face are normalized in such a way that they sum up to 1, so you should always consider the emotion with higher confidence. The Emotion API works across all cultures.
	Face API	This API can be used for face detection and face recognition in an image. The response contains the coordinates of the bounded rectangle (left, right, top, bottom) that indicates the face within an image and a few face-related attributes, like age, gender, pose, head pose, facial hair, and glasses. It can also perform one-to-one face verification, check for similar-looking faces, and do face grouping, face identification from a database of faces, and face storage.
	Video API	This API allows you to detect faces and emotions in a video or URL of a video. You can also stabilize a video or generate a thumbnail.

(*continued*)

Table 10-1. (*continued*)

Speech	Bing Speech API	This API helps you enable voice input for your applications. The API allows you to convert speech to text and vice versa. It supports around 29 different languages (at the time of writing) for speech to text and around 40 languages for speech to text.
	Speaker Recognition API	This API provides speaker identification and verification services. Using speaker identification, you can provide sophisticated voice-based authentication to your users. Each user should enroll by recording a specific phrase; during authentication the voice for the same phrase is compared with the voice and phrase (and a few other features) recorded during enrollment. For speaker identification, each speaker should be enrolled by speaking any phrase; for any given voice input the API compares the voice using the enrolled list and returns an identity if found.
Language	Bing Spell Check API	This API can be used to provide contextual grammar and spelling corrections using the web-based spell checker built using Azure ML. Traditionally, spell check is performed using dictionary-based rule sets, as in Microsoft Word, but Bing spell check uses web documents to provide real-time spelling and contextual grammar checks. It supports many sophisticated features like slang, word breaking, and so forth, which is otherwise not possible with regular spell checkers.
	Text Analytics API	This API can be used to perform sentiment analysis from text, extract key phrases from a sentence using the Natural Language Processing Kit, extract topics from a discussion or user review, or do language detection from a handwritten text (around 120 languages are supported).
	Web Language Model API	This API can be used to build a language model using the web-scale corpus collected by Bing in the en-us market. The data set is available as the XML Web Service; it is divided into four categories, namely Body Text, Title Text, Anchor Text, and Query Text.
	Linguistics Analysis API	This API can be used to identify the structure of text. The following services are provided by the API: 1. Sentence separation and tokenization 2. Part-of-speech tagging 3. Constituency parsing

(*continued*)

Table 10-1. (*continued*)

Knowledge	Recommendations API	This API can be used to recommend items to your customer based on their activity and the trained catalog of items in your store. The API supports scenarios like frequently bought together, recommend related items, and recommend items based on customer's past activity.
	Entity-Linking Intelligence Service	This API can be used to identify entities within a specific paragraph and context. For example, in the sentence "The Bermuda Triangle, also known as the Devil's Triangle, is a loosely-defined region in the western part of the North Atlantic Ocean," the API recognizes entities like "Bermuda Triangle," "Devil's Triangle," and "North Atlantic Ocean." This can be combined with Bing Search API to link discovered entities with contextual information from the web.
	Academic Knowledge API	This API can be used to interpret search queries related to academic intent using the Microsoft Academic Graph knowledge base. The knowledge base is divided into field of study, author, institution, paper, venue, and event. The MAG knowledge base is continuously indexed and mined by using Bing.
Search	Bing Search API	This API provides search capabilities like Bing.com that can be integrated into any application. Using this API, you can perform searches related to web pages, images, videos, and news. Bing Search API also provides content-specific APIs for images, web, videos, and news for more specific search.
	Bing Auto-Suggest API	This API can be used to provide auto-suggest capabilities for search queries to your users using data from the web. The suggested terms are picked from the query searches performed by other users as well as from user intent.

Apart from these, Microsoft also allows you to take an early look at their biding Cognitive Services, like Project Prague, Project Johannesburg, and Project Wollongong; for more details on each of these projects visit https://labs.cognitive.microsoft.com/.

Getting Started

Like any other Azure service, getting started with Azure Cognitive Services is quite easy. All you need is a Windows Live Account and an Azure subscription. You can also acquire a free account for MCS by visiting https://www.microsoft.com/cognitive-services/en-US/subscriptions. The free account provides access to each of the APIs just described for a period of 30 or 90 days depending on the type of service (Figure 10-1).

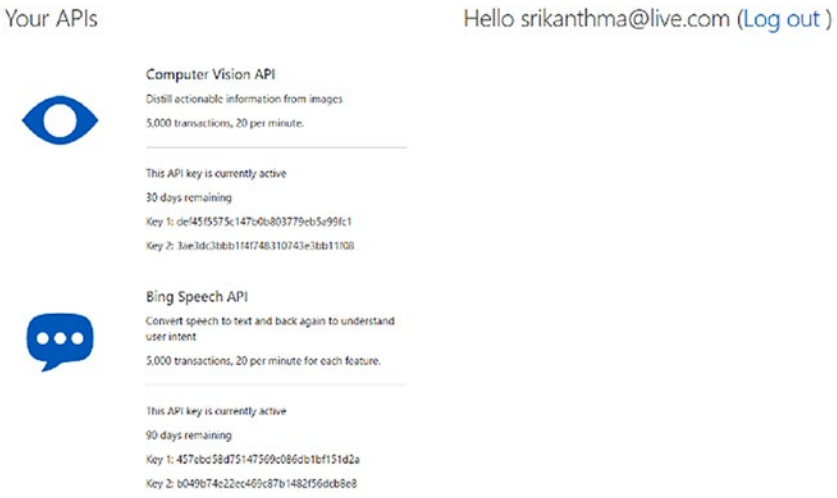

Figure 10-1. *Cognitive Service API keys*

Each API contains the threshold limit set for the service, Key 1, and Key 2. For example, using the free version you can perform a total of 5000 transactions in a period of 90 days and only 20 per minute. While this may be sufficient for learning, to build a production bot we should create an account using the following steps:

1. Sign into the Azure Portal (https://portal.azure.com).
2. Click +New.
3. Select AI + Cognitive Services and discover the list of APIs available under MCS. Click on "See All" to see the complete list of APIs (Figure 10-2).

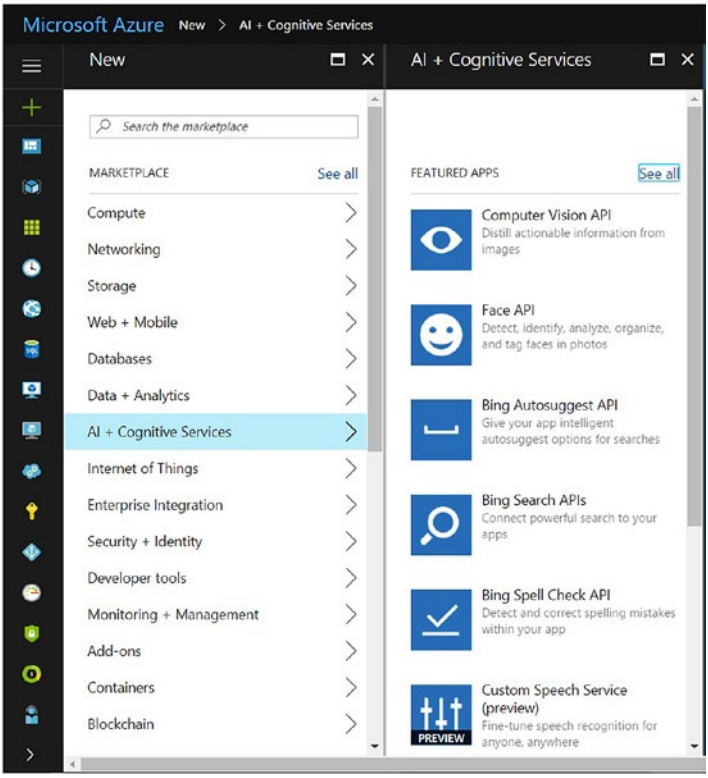

Figure 10-2. *AI + Cognitive Services on Azure Portal*

4. Click on the API of your choice to create an account.
5. On the Create page, provide the following information:

    a. **Account name** – The descriptive name of the account. It is recommended to use the name of the product, location, and environment in the account name. For example, `contoso_ea_dev_bingsearch`.

    b. **Subscription** – Choose the Azure subscription to be used to associate with the account. To select a subscription, you should have at least contributor access.

    c. **Pricing tier** – Each API comes with multiple pricing tiers; the cost of the service depends on the pricing tier chosen here. Each pricing tier also comes with a threshold value. If the API reaches the threshold, the service is throttled. Choose a pricing tier that fits the needs of your bot. You can upgrade/degrade your service tier after provisioning an account.

d. **Resource group** – Choose an existing resource group or create a new one. The resource group defines the logical group of resources the API will fall under and also the location.

e. Finally, read and agree to the Microsoft terms and conditions before clicking Create.

Figure 10-3 shows a Bing Search Cognitive Services account created using Azure Portal.

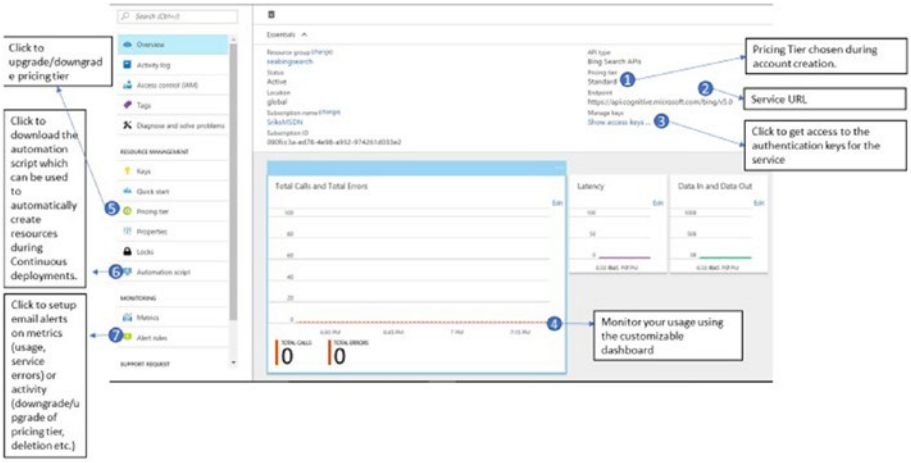

Figure 10-3. *Bing Search Service Azure dashboard*

You can test the service using the API testing console at `https://dev.cognitive.microsoft.com/docs/services/56b43eeccf5ff8098cef3807/operations/56b4447dcf5ff8098cef380d` or by using any HTTP REST client, like fiddler or Postman. Figure 10-4 shows testing the Bing Web Search API using the account just created on an Open API Testing console. The `ocp-Apim-Subscription-key` can be obtained by clicking on the "Show access keys" option.

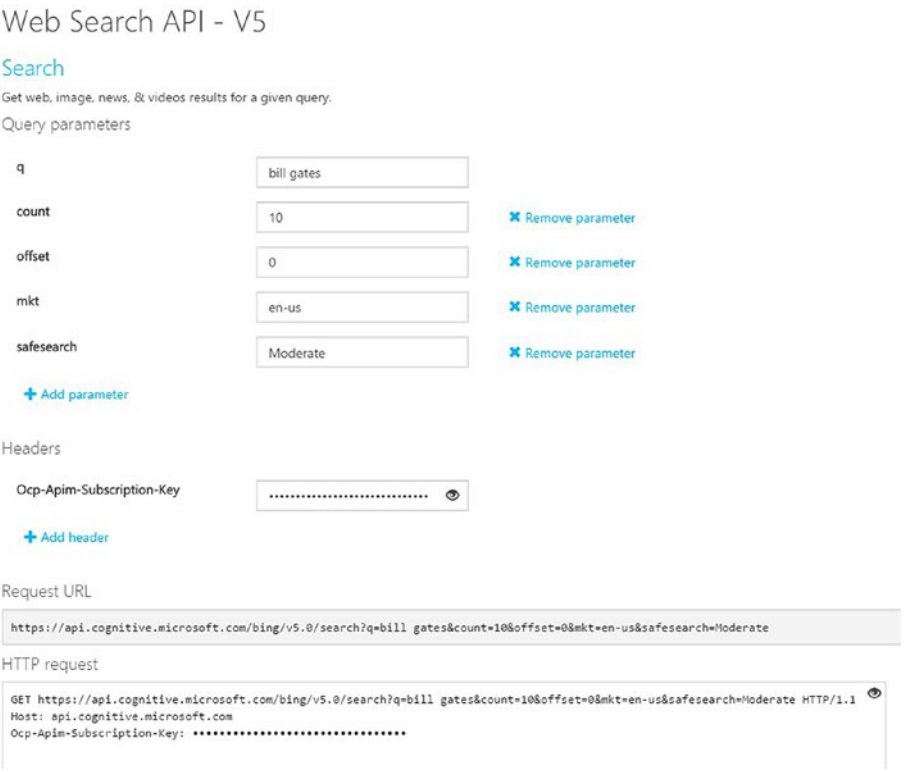

Figure 10-4. *Bing Web Search API testing console*

The response for this request looks like Figure 10-5.

Response status

200 OK

Response latency

882 ms

Response content


```
Vary: Accept-Encoding
BingAPIs-TraceId: 339A2248BBEE49FCA5068FD1849E375C
X-MSEdge-ClientID: 0EB1CE0F7E5C635613B6C4867F036211
X-MSAPI-UserState: 4ec9
BingAPIs-Market: en-US
X-MSEdge-Ref: Ref A: 339A2248BBEE49FCA5068FD1849E375C Ref B: BAYEDGE0421 Ref C: Sun May 21 07:04:38 2017 PST
apim-request-id: 17d5c992-aba0-4acb-8005-cf5202c4f78f
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
x-content-type-options: nosniff
Cache-Control: max-age=0, private
Date: Sun, 21 May 2017 14:04:39 GMT
P3P: CP="NON UNI COM NAV STA LOC CURa DEVa PSAa PSDa OUR IND"
Content-Length: 55488
Content-Type: application/json; charset=utf-8
Expires: Sun, 21 May 2017 14:03:38 GMT

{
  "_type": "SearchResponse",
  "webPages": {
    "webSearchUrl": "https://www.bing.com/cr?IG=86426E46676545218CB2851DA1E47873&CID=0EB1CE0F7E5C635613B6C4867F036211&rd=1&h=W8h
ngOM84FSaRr8SkjXGZZHmLQ9nwPfNLzvKn-f8V2Q&v=1&r=https%3a%2f%2fwww.bing.com%2fsearch%3fq%3dbill%2bgates&p=DevEx,5534.1",
    "totalEstimatedMatches": 565000,
    "value": [
      {
        "id": "https://api.cognitive.microsoft.com/api/v5/#WebPages.0",
        "name": "Bill & Melinda Gates Foundation - Official Site",
        "url": "http://www.bing.com/cr?IG=86426E46676545218CB2851DA1E47873&CID=0EB1CE0F7E5C635613B6C4867F036211&rd=1&h=1JbeG5FmF
O2xt80Ud1j66P54vrvsIcPJEduZjjLluEQ&v=1&r=http%3a%2f%2fwww.gatesfoundation.org%2f&p=DevEx,5089.1",
        "about": [
          {
            "name": "Bill & Melinda Gates Foundation"
          },
```


Figure 10-5. *Sample Bing Search response*

The same request can be made from the Postman HTTP client, as shown in Figure 10-6. The Postman HTTP client can be downloaded from `https://www.getpostman.com/apps`.

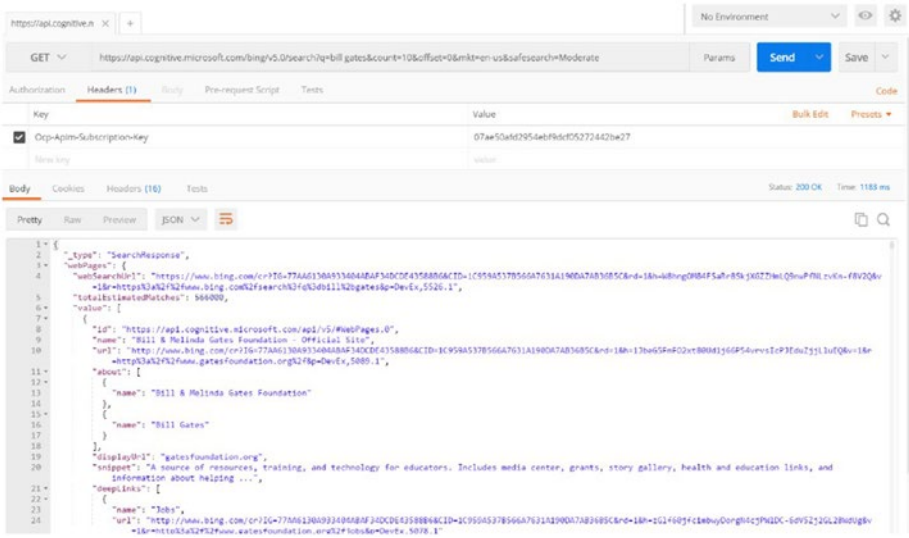

Figure 10-6. *Invoking Bing Web Search API from Postman*

Figure 10-7 shows where MCS fits into the lifecycle of a bot's conversation.

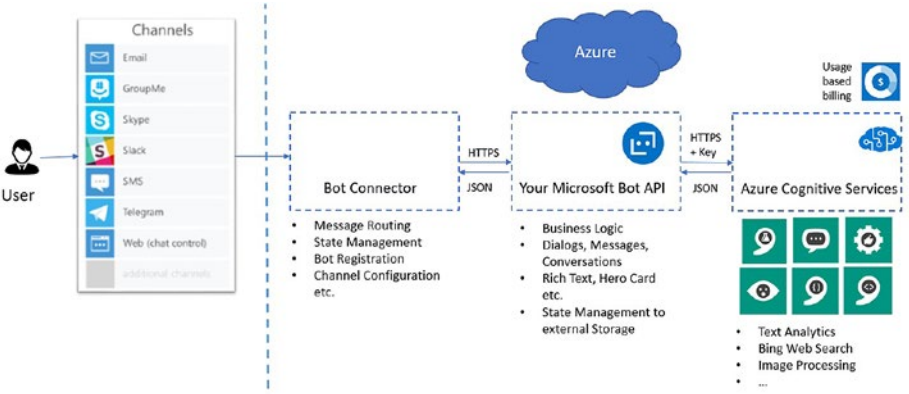

Figure 10-7. *Microsoft Cognitive Services and Bot framework ecosystem*

Building Smart Bots with Bing Web Search

Bing Search APIs allow you to add intelligent search capabilities to your bot, providing the experience of a Bing-style web search from the bot's interface. Using the Bing Search API, the user's bot conversations can be easily converted to search queries and the responses can be influenced by knowledge acquired from the web pages, images, video, and news related to the user's conversation. Search APIs provide fine-grained control over the request. With them, we can search the web for location-based information, apply filters to eliminate adult content, and order the results by a specific date. The Bing Search API consolidates the searches from web pages, images, videos, and news for any given search query. The Search API is segregated into more-specific content APIs, like Image Search API, News Search API, and Video Search API. If you are looking specifically for a particular type of content, you should use the content-specific API instead. For example, if you are building a bot that searches for relevant images based on the text entered by the user as part of the conversation, you can use the Image Search API instead. Bing Search API is a super set of all the content-specific APIs; however, for a search query Bing Search only returns relevant results for each of the content types. For example, for a query like "rainforest colorful birds," the Bing Search API might return images, web pages, and news but not videos, because it did not find anything relevant to the search query, whereas the content-specific APIs would still return videos for the same search query—they just might not be very relevant.

Let us build a simple web-search bot using Microsoft Cognitive Services' Bing Search API.

- Open Visual Studio as Administrator and create a new bot project called `SearchBot`.
- Create a new class called `SearchDialog` in the project.
- The following code shows the template for the `SearchDialog` class:

```
namespace SearchBot
{
    using System;
    using System.Threading.Tasks;
    using Microsoft.Bot.Builder.Dialogs;

    [Serializable]
    public class SearchDialog : IDialog<Object>
    {
        public Task StartAsync(IDialogContext context)
        {
            throw new NotImplementedException();
        }
    }
}
```

IDialogs is an interface that is used to model a conversation for the user; it is more flexible for authoring new styles of conversations. In the below code, an object of SearchDialog is instantiated and the StartAsync method is called when a new conversation starts. IDialogContext contains a stack of dialogs active in the current conversation.

- Add the following code to the StartAsync method so that it waits for the user's response:

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

The method MessageReceiveAsync acts as a handler whenever a new message arrives from the user:

```
public async Task MessageReceiveAsync(IDialogContext dialogContext,
IAwaitable<IMessageActivity> argument)
{
    throw new NotImplementedException();
}
```

IAwaitable<IMessageActivity> contains the conversation message from the user. Before we make a call to the Bing Search API, we need to get a Cognitive Services subscription key from https://azure.microsoft.com/en-us/try/cognitive-services/my-apis/.

The link shows the keys and the usage limits for each of the cognitive services subscribed to, as shown in Figure 10-8. Copy Key 1; Key 2 is used as a spare key. The key acts as an authentication token for our requests to the Bing Search API.

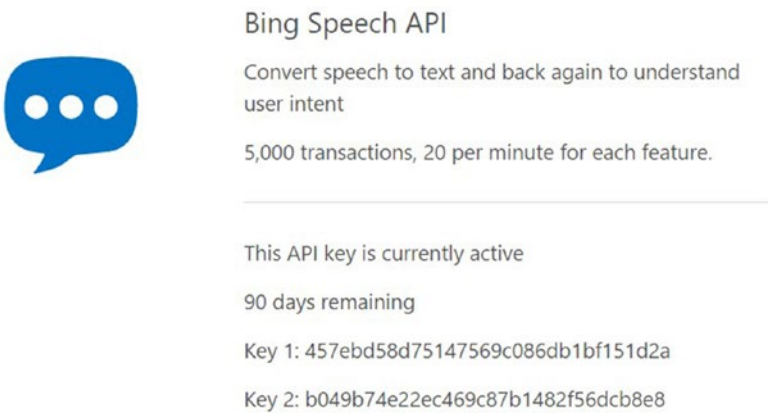

Figure 10-8. *Bing Speech API*

To fetch the web-search results, we should make a GET request to the Bing Search API endpoint: https://api.cognitive.microsoft.com/bing/v7.0/search.

■ **Note** At the time of writing, v7 is still in preview. The REST endpoint for v5 is https://api.cognitive.microsoft.com/bing/v5.0/search.

To interact with the Bing Search API, we can use any HTTP client library that is capable of making HTTPS-based REST calls. In this example, we will be using the HttpClient class from the System.Net.Http assembly.

```
using (var httpClient = new HttpClient())
{
  var queryString = HttpUtility.ParseQueryString(string.Empty);

  // Request headers
  httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key",
"{SubscriptionKey}");
}
```

The subscription key fetched goes as part of the request headers. It acts as an authentication key and also helps to track the number of requests made. Remember all the Microsoft Cognitive Service API calls are metered, and there is a limit on the number of API calls that can be made as part of a free subscription. With the free subscription, Bing Search API allows you to perform 1,000 transactions per month and up to 7 per second.

The search query from the user will be used as a request header. Bing Search API allows a few more request headers, as mentioned in the following table. Each of the headers adds more context to the search request, like user's location and client IP, which can be used to personalize the search response. Table 10-2 shows request headers and query parameters that can be used with Bing Web Search v7.

Table 10-2. *Bing Speech API Request Headers*

Header	Required	Description
`Ocp-Apim-Subscription-Key`	Yes	The authentication key obtained by subscribing for the service.
`X-MSEdge-ClientID`	No (Optional)	Bing uses this header to provide a consistent user experience to users for requests originating from the same user on the same device. Bing also uses this ID to improve the result ranking by using the activity generated by the user.
`X-Search-ClientIP`	No (Optional)	The IPv4 or IPv6 address of the user's device. This is used to discover the user's location. Bing uses the location to determine safe search behavior.
`X-Search-Location`	No (Optional)	A semicolon-separated list of key–value pairs that describe the client's geographical location. This information is used by Bing to provide safe search behavior and location-relevant information

Query Parameters

To make a Bing Search from a bot application we should create an `HTTP GET` request, which comprises the URL, request headers, and query parameters. Table 10-3 shows the query string parameters that can be appended to the `GET` request to form a search request. Each parameter value should be URL encoded before being appended to the request.

Table 10-3. *Query String Parameters*

Parameter	Required/Optional	Description	DataType
`Q`	Required	Search term	String
`Mkt`	Optional	The market you want to associate the request with. It takes the format of `<languageCode>-<countryCode>`; for example, en-US. For full list of market codes, visit `https://docs.microsoft.com/en-us/rest/api/cognitiveservices/bing-web-api-v7-reference#market-codes`.	String
`Cc`	Optional	A two-character country code indicating where the results come from. Bing uses the cc code combined Accept-Language to show results with nearest match. `Cc` and `mkt` are mutually exclusive—only one of them should be included in a request.	String
`Count`	Optional	The number of search results to be included in the response. The actual number of results can be lesser than the specified count. You can combine this with `offset` to page the results.	Positive Integer
`Answercount`	Optional	The number of answers you want the response to include. The answers are filtered based on ranking. This can be combined with responseFilter parameter below to filter further.	Positive Integer
`Freshness`	Optional	Filters results by age in terms of day, week, or month.	String
`Offset`	Optional	This zero-based offset can be used to skip a few results. The default is zero (0). This parameter can be used along with `Count` to page the results.	Positive Integer

(*continued*)

Table 10-3. (*continued*)

Parameter	Required/Optional	Description	DataType
`Promote`	Optional	A comma-delimited list of results that you want to promote irrespective of ranking. For example: `news, video` will add news and video to the search result irrespective of the `answerCount` set on the same request. This parameter is to be used along with `answerCount` only. For example: If you want to get top two ranked answers and you want to promote videos within that, you should set `answerCount` to 2 and `promote` to video.	String
`responseFilter`	Optional	A comma-delimited list of results that you want to include in the result. Use `promote` to force include any specific type of response.	String
`safeSearch`	Optional	Use as a filter to exclude adult content. The possible values are `Off`, `Moderate`, and `Strict`.	String

Bing Search Request

The following code shows how to make a Bing Search API request from a bot application built in C#:

```
using (var httpClient = new HttpClient())
            {
                var queryString = HttpUtility.ParseQueryString(string.Empty);

                // Request headers
                httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-
                Key", "ff72d7edc09741c7b350906980ab502a");
                // User agent for PC, this should be extracted from client's
                request to Bot application and
                // forwarded to the Cognitive Services API
                httpClient.DefaultRequestHeaders.Add("User-Agent",
                "Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0;
                Trident/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 822)");
                // Bing-generated ID to identify the request
                httpClient.DefaultRequestHeaders.Add("X-Search-ClientIP",
                "999.999.999.999");
                // User's location
```

```
            httpClient.DefaultRequestHeaders.Add("X-Search-Location",
            "lat:17.4761950,long:78.3813510,re:100");

            // Query parameters
            queryString["q"] = message.Text;
            // user's search query
            queryString["count"] = "10";
            // number of results to include
            queryString["offset"] = "0";
            // number of pages to skip
            queryString["mkt"] = "en-us";
            // market to associate the request with
            QueryString["safesearch"] = "Moderate";
            // safe search other possible values are Off and Strict
            queryString["freshness"] = "Day";
            // freshness of the content by Day, Week, Month
            queryString["answerCount"] = "2";
            //
            queryString["promote"] = "Video, News";
            // freshness of the content by Day, Week, Month
            var uri = "https://api.cognitive.microsoft.com/bing/v7.0/
            search?" + queryString;
            var responseMessage = await httpClient.GetAsync(uri);
            var responseContent = await responseMessage.Content.
            ReadAsByteArrayAsync();
            var response = Encoding.ASCII.GetString(responseContent, 0,
            responseContent.Length);
            dynamic data = JsonConvert.DeserializeObject<object>
            (response) ;

        }
```

Figure 10-9 shows the response for the preceding request:

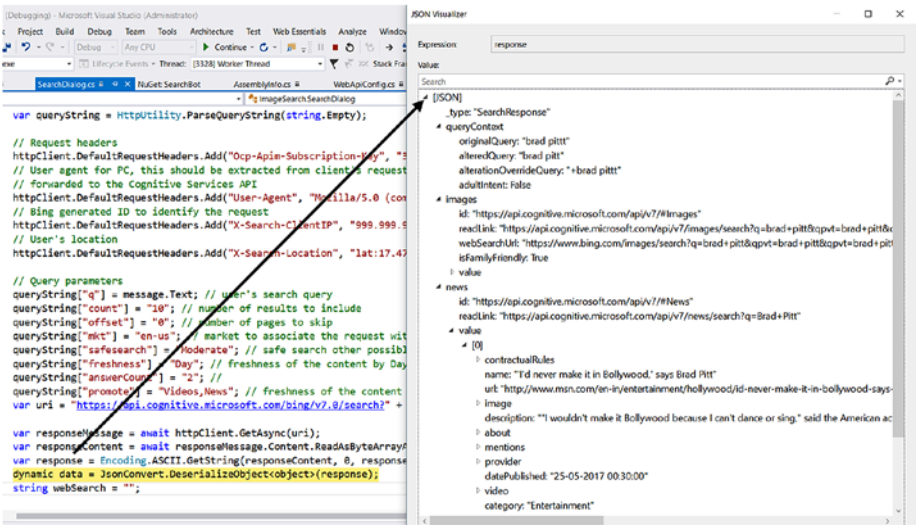

Figure 10-9. *Bing Search response*

To parse the response, you can use any JSON parser for C#, like Newtonsoft.json: https://www.nuget.org/packages/newtonsoft.json/.

The following code shows how the response is parsed to convert image-search results to Hero cards:

```
var replyMessage = dialogContext.MakeMessage();
replyMessage.Attachments = new List<Attachment>();

                foreach (var webPage in data.images.value)
                {
                    string name = webPage.name;
                    string url = webPage.thumbnailUrl;
                    string displayUrl = webPage.contentUrl;
                    string snippet = webPage.encodingFormat;
                    string hostPageDisplayUrl = webPage.hostPageDisplayUrl;
                    DateTime datePublished = webPage.datePublished;
                    replyMessage.Attachments.Add(new ThumbnailCard
                    {
                        Title = name,
                        Subtitle = "Date Published: " + datePublished.
                        ToLongDateString(),
                        Images = new List<CardImage> { new
                        CardImage(displayUrl) },
                        Buttons = new List<CardAction> { new
                        CardAction(ActionTypes.OpenUrl, "Know more", value:
                        hostPageDisplayUrl ) }
                    }.ToAttachment());
                }

                await dialogContext.PostAsync(replyMessage);
```

Figure 10-10 shows the user experience of this code from Bot Emulator.

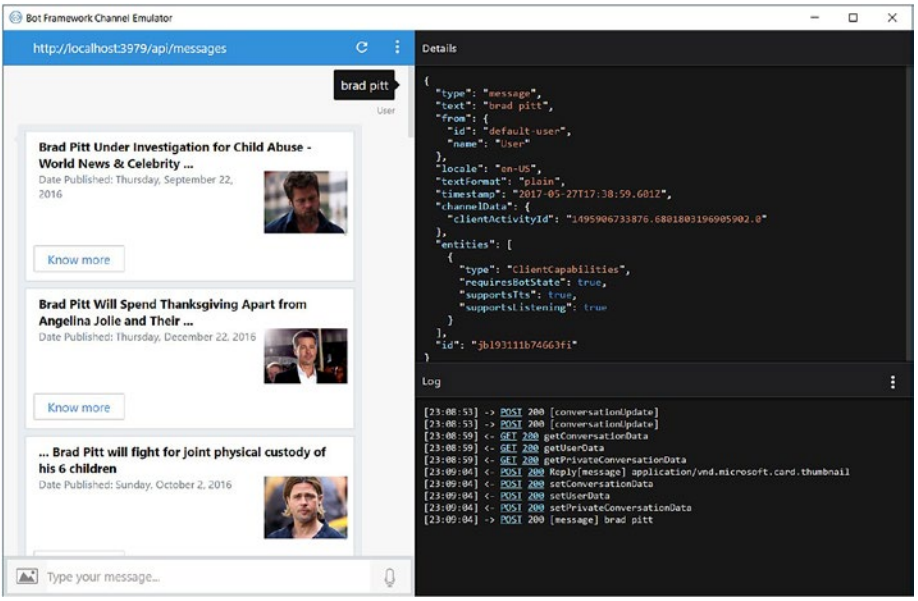

Figure 10-10. *Bing image search*

If you look through the lens of the HTTP watch tool fiddler, this is how the request looks:

```
GET https://api.cognitive.microsoft.com/bing/v7.0/search?q=brad+pittt&count=
10&offset=0&mkt=en-us&safesearch=Moderate&freshness=Day&answerCount=2&promote=
Videos%2cNews HTTP/1.1
Ocp-Apim-Subscription-Key: 35c667a6d90049678156ae7d8a064a4d
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0;
Trident/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 822)
X-Search-ClientIP: 999.999.999.999
X-Search-Location: lat:17.4761950,long:78.3813510,re:100
Host: api.cognitive.microsoft.com
Connection: Keep-Alive
```

Handling Errors

It is very important to consider resiliency when building applications that rely heavily on services whose availability are not under our control. Though Microsoft guarantees the availability of services as per their SLA, there could be instances where the service might fail to respond. In these cases, the errors should be captured and acted upon accordingly. A few scenarios where the service might fail to respond are as follows:

- Invalid/expired subscription key
- Missing a required parameter in the request/invalid request
- Quota expired
- Unexpected server error

In these cases, our bot application should capture the error and convert it into a user-friendly message so that users are aware of the unavailability and can act accordingly. If the request fails while connecting to Microsoft Cognitive Services, the response object will contain all the necessary details. The request can fail as a result of several errors, in which case the error-response object will contain a list of error objects, as shown in Figure 10-11.

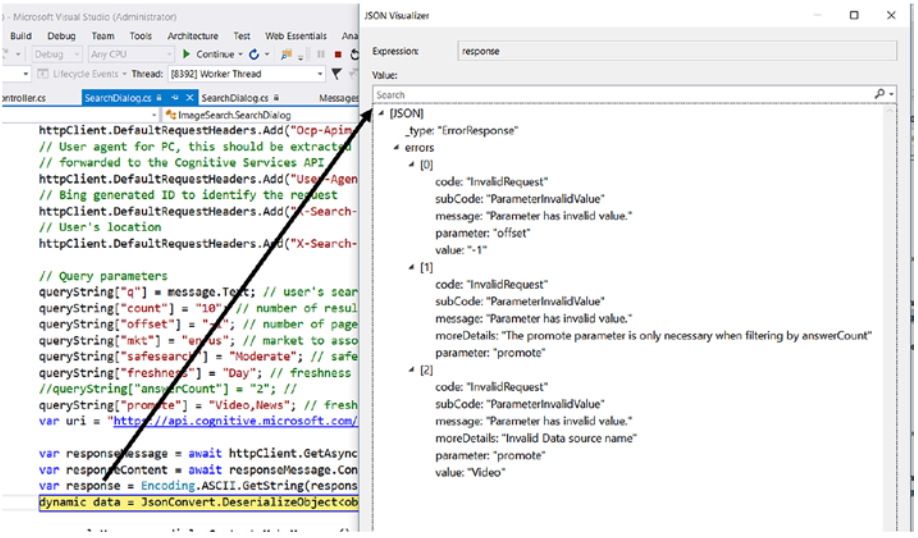

Figure 10-11. *Microsoft Cognitive Services error response*

The request in the figure has three incorrect parameter values, and aptly the response type is marked as `ErrorResponse`, so as a developer you should use this field to differentiate between a valid response and an error response. Additionally, each of the error object shows the name of the parameter, value passed for the parameter, and the message. For more details on error codes and sub-error codes, visit https://docs.microsoft.com/en-us/rest/api/cognitiveservices/bing-web-api-v7-reference#error-codes.

Every user who subscribes to Cognitive Service is granted access, with a few threshold limits called quota per month (QPM) and quota per second (QPS). As a bot application developer, it is your responsibility to design your application in such a way that you always stay within these defined limits. The service responds with HTTP response code 403 if the service exceeds QPM. One way of handling this is to have a pool of subscriptions and to retry the request with a different subscription key. The service responds with HTTP response code 429 if the service exceeds QPS, and the service also includes a `Retry-After` header, which can be used to retry after a specified number of seconds. Cognitive Services is smart enough to differentiate between DOS (denial of service) attack and QPS violations. If the service detects DOS, it responds with 200 OK and an empty response.

Bing Search contains a generic search option called `search` and content-specific API endpoints for images, news, and videos so on. Table 10-4 shows all the available endpoints and what they provide for v7. If you're building a bot for a specific response type, like a Bing image search, you should query the content-specific API rather than the generic search API. The generic search API responds with all types of content types matching the search query, which can be filtered further by using attributes like `responseFilter` and `promote`.

Table 10-4. *Bing Search API Endpoints*

Endpoint	Details
https://api.cognitive.microsoft.com/bing/v7.0/search	The response includes links to images, web pages, videos, and web searches that are related to the user's search query.
https://api.cognitive.microsoft.com/bing/v7.0/videos/search	Returns videos relevant to user's query
https://api.cognitive.microsoft.com/bing/v7.0/videos/details	Returns videos along with relevant details like related videos
https://api.cognitive.microsoft.com/bing/v7.0/videos/trending	Returns trending videos related to the query. This can be combined with a market parameter to get trending videos by market
https://api.cognitive.microsoft.com/bing/v7.0/images/search	Returns images relevant to user's query
https://api.cognitive.microsoft.com/bing/v7.0/images/details	Returns images along with relevant details like web pages that include the image
https://api.cognitive.microsoft.com/bing/v7.0/images/trending	Returns trending images related to the query
https://api.cognitive.microsoft.com/bing/v7.0/Suggestions	Returns a list of intent-based auto-suggest queries that other users have searched; this can be used to provide a rich search experience to users
https://api.cognitive.microsoft.com/bing/v7.0/news	Returns top news articles by category, like sports, politics, technology, etc.
https://api.cognitive.microsoft.com/bing/v7.0/news/search	Returns top news articles related to the user's search query
https://api.cognitive.microsoft.com/bing/v7.0/news/trendingtopics	Returns trending topics from social networks
https://api.cognitive.microsoft.com/bing/v7.0/SpellCheck	Spell-check service lets you perform a spelling and grammar check on a string

Apart from the above list, Bing Search also supports computation- and time zone–related search queries. For example, the following query returns a computational response as shown in Figure 10-12:

```
https://api.cognitive.microsoft.com/bing/v7.0/search?q=how+many+litres+
in+one+gallon&mkt=en-us
```

```
[JSON]
    _type: "SearchResponse"
  queryContext
      originalQuery: "how many litres in one gallon"
  webPages
  computation
      id: "https://api.cognitive.microsoft.com/api/v7/#Computation"
      expression: "1 US gallon"
      value: "3.7854 liters"
  rankingResponse
```


Figure 10-12. *Bing computation search query*

Optical Character Recognition with Computer Vision API

Search forms a very critical component of any data-oriented application. Azure Search is a popular PaaS service from Microsoft that can be used to build applications with search capabilities that use various forms of data sources. It helps to perform a quick search from any form of application using a simple-to-use REST API or C# SDK. Data can be ingested for search from various relational databases, like SQL, and non-relational databases, like JSON documents. Text can be extracted from PDFs and MS Word documents and stored in Azure blob storage for Azure Search. Images also contain text, like the one shown in Figure 10-13. For example, you might want to create a bot application that allows doctors to search through patients' lab records. The data comes from various labs in the form of digital documents (JPEG, PNG). But what if you want to search text within images? Extracting text embedded in images is quite challenging, Cognitive Services helps us solve this problem by offering a simple API called OCR. OCR (optical character recognition) is a state-of-the-art Intelligent Vision API that can be used to detect and extract text from images. The service detects over 20 languages today and can auto-detect the language from the uploaded image.

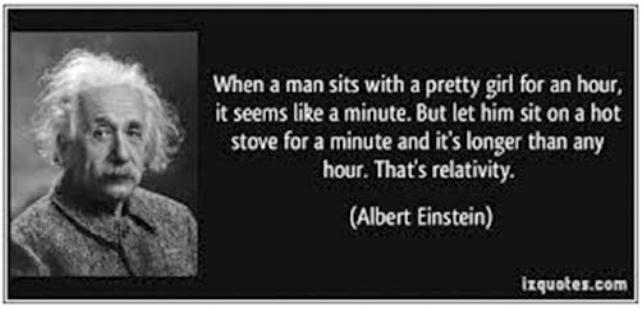

Figure 10-13. *Sample image with text. Source: izquotes.com*

Users can upload an image or provide a URL of the image. The image containing the text should adhere to the following restrictions:

- The size of the image must be between 40 × 40 and 3200 × 3200 pixels.
- The image cannot be greater than 10 MB.

The accuracy of the text depends on the quality of the image. In the following example, I'm building a Skype bot that my users can use to upload images with text. The extracted text is then uploaded to the Azure Search index, which makes it instantly searchable. Let us start by building the bot application. The following code shows the sample code for making a request to the OCR vision API:

```
string _apiUrlBase = "https://westus.api.cognitive.microsoft.com/vision/
v1.0/ocr";
var message = await argument;
string ret = "";
if (message.Attachments != null && message.Attachments.Any())
{
  var attachment = message.Attachments.First();
  using (HttpClient httpClient = new HttpClient())
  {
        // Skype & MS Teams attachment URLs are secured by a JwtToken, so we
        need to pass the token from our bot.
        if ((message.ChannelId.Equals("skype", StringComparison.
        InvariantCultureIgnoreCase) || message.ChannelId.Equals("msteams",
        StringComparison.InvariantCultureIgnoreCase))
                        && new Uri(attachment.ContentUrl).Host.
                        EndsWith("skype.com"))
       {
           var token = await new MicrosoftAppCredentials().GetTokenAsync();
           httpClient.DefaultRequestHeaders.Authorization = new Authentication
           HeaderValue("Bearer", token);
       }

        var responseMessage = await httpClient.GetAsync(attachment.ContentUrl);
        using (var httpClient1 = new HttpClient())
       {
                        //set up HttpClient
          httpClient1.BaseAddress = new Uri(_apiUrlBase);
          httpClient1.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key",
          "5c3911e21d464982a8f5f1272d294cc3");
          HttpContent content = responseMessage.Content;
          content.Headers.ContentType = new MediaTypeWithQualityHeaderValue
          ("application/octet-stream");
          var response = await httpClient1.PostAsync(_apiUrlBase, content);
          var responseContent = await response.Content.ReadAsByteArrayAsync();
          ret = Encoding.ASCII.GetString(responseContent, 0,
          responseContent.Length);
```

```
            dynamic image = JsonConvert.DeserializeObject<object>(ret);
        }
    }
}
```

The above code ensures that the request contains an image as an attachment and uploads the image as a series of bytes with the appropriate media type as the content header. Figure 10-14 shows the sample response from the service. The response contains a region array, which identifies the regions where the text was available on the image, along with the coordinates of the region, which are specified in the `boundingBox` attribute. Each region contains lines, and lines are made up of words. Again, each line and box contains the `boundingBox` attribute, which specifies the coordinates of the words within the image.

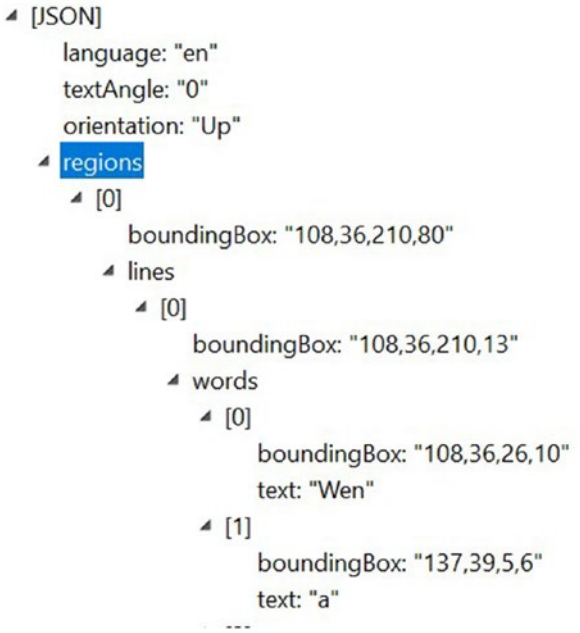

Figure 10-14. *OCR response*

The following is the sample code that parses the response from all the regions available on the image and extracts the complete sentence. Figure 10-15 shows the result.

```
string temp = "";
foreach (var regs in image.regions)
{
    foreach (var lns in regs.lines)
    {
        foreach (var wds in lns.words)
```

```
        {
            temp += wds.text + " ";
        }
    }
}
await context.PostAsync($"The text found is  '{temp}' ");
```

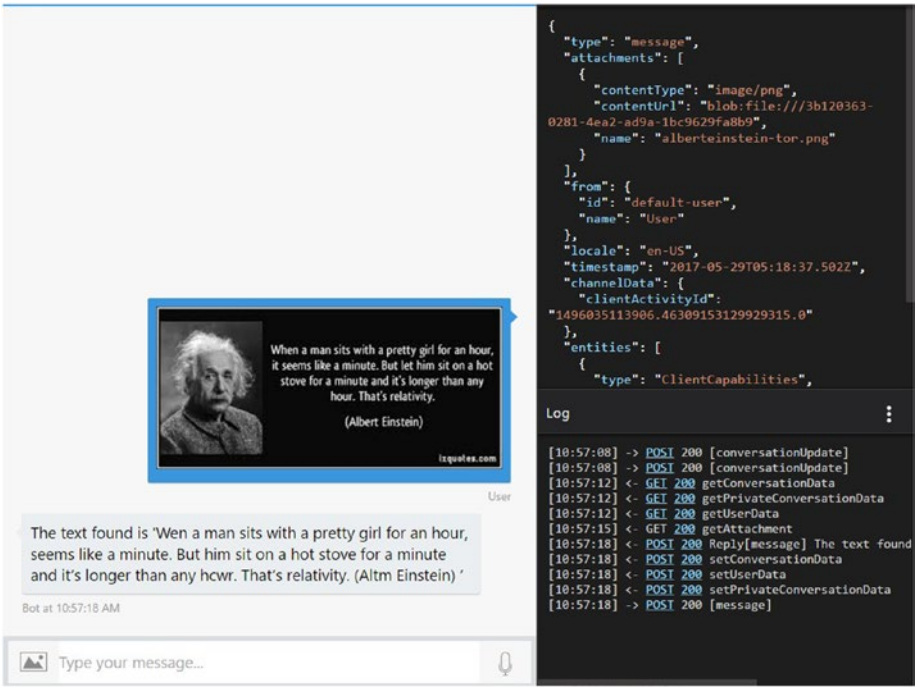

Figure 10-15. *Conversation with OCR bot using Bot Emulator*

In some cases, an image can contain text oriented at an angle. OCR can handle such cases by rotating the image before extracting the text, which can be done by setting `detectOrientation` to true in the request headers. If OCR rotates the image before extracting text, the angle of rotation (when rotated clockwise around the center of the original image) for each text is included in the response. Post rotation, the `orientation` property included in the response should provide the direction of the text within the image (up, down, left, and right) relative to the rotated image or text angle. If the image contains texts at multiple angles, only one of them will be extracted.

The text extracted in the form of words, lines, and regions can be combined to make a complete sentence, which can then be used in various ways; for example, it can be ingested into the Azure Search index, as shown in the following code. This allows your users to search images based on the text contained within them or perform trend or common data analysis based on the extracted text. For example, if it is a lab report, I can group the information by blood groups and do more interesting analysis, like classifying vitamin deficiency among various age groups and so on.

```
// Azure search index endpoint
string _searchAPIBaseURI = "https://mcs-ocr.search.windows.net/indexes/
ocrindex/docs/index?api-version=2016-09-01";

// request body for inserting data into Azure Search
JObject postData = new JObject();
dynamic indexData = new JObject();
indexData["@search.action"] = "upload";
indexData.id = Guid.NewGuid().ToString();
indexData.value = temp;
postData["value"] = new JArray(indexData);

using (var httpClient2 = new HttpClient())
{
    httpClient2.BaseAddress = new Uri(_searchAPIBaseURI);
    httpClient2.DefaultRequestHeaders.Add("api-key",
    "3A3D68FC446694903A0CCB62E5C94EC9");
    var response = await httpClient2.PostAsJsonAsync<JObject>
    (_searchAPIBaseURI, postData);
}
```

Summary

Microsoft Cognitive Services (formerly called Project Oxford) is a collection of APIs built on top of Azure ML to provide rich and intelligent services to developers. It enables developers to integrate into their bots intelligent algorithms built for emotion and video detection, facial expression detection, speech and vision recognition, natural language processing, and many more. The pricing model is scoped individually for each service. Based on the pricing model chosen, each service contains QPS (quota per second) and QPM (quota per month) values that act as thresholds for the service. More details on pricing are available at https://azure.microsoft.com/en-in/pricing/details/cognitive-services/.

Bot applications can easily integrate with MCS and access different services, like image analysis, natural language processing, or speech-to-text conversion, using HTTP/REST calls. Based on the channel chosen for your bot, you can choose any API; for example, if you are creating a bot for Skype for Business users, you can use the speech-to-text conversion service for voice calls.

Developers can learn to integrate with any service by creating a trial account at http://microsoft.com/cognitive-services/en-US/subscriptions. In order to create a new service, you can log in to Azure Portal at https://portal.azure.com and search for Cognitive Services.

CHAPTER 11

Bot Operations

Bots are essentially web APIs and need a container in which to be deployed. The container could be a virtual machine or a managed hosted platform as a service (PaaS). Often, bots are hosted on a certain PaaS. Azure provides app services that form a rich platform for hosting bots created using MS Bot framework. Azure also provides Bots as a Service, and Azure functions can be used to write bots quickly.

Authoring a bot solves one large piece of the overall puzzle. The other big piece is to manage your bot appropriately. Monitoring, administrating, and managing are equally important to the success of the bot in the long run.

This chapter focuses on the monitoring and management aspects of bots.

Microsoft provides Application Insights as a universal service for monitoring and providing rich information about applications, including bots.

Application Insights

Application Insights is Microsoft's flagship service for managing applications across platforms, languages, operating systems, and locations. Application Insights provides many rich monitoring and management capabilities; it can be used for any kind of application on any platform. It provides real-time streaming of incoming data, analytics, application maps, performance information, availability results, and more.

Application Insights installs a small footprint binary in the application. The role of this agent is to gather instrumentation and log information from the application and its ecosystem and send it across to the service hosted on Azure. The service provides rich reports and dashboard features for advanced visualization of incoming data and to generate insights. Moreover, the raw data can further be consumed by multiple platforms like Power BI, webhooks, and the generation of alerts. The architecture of Application Insights is shown in Figure 11-1.

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2_11

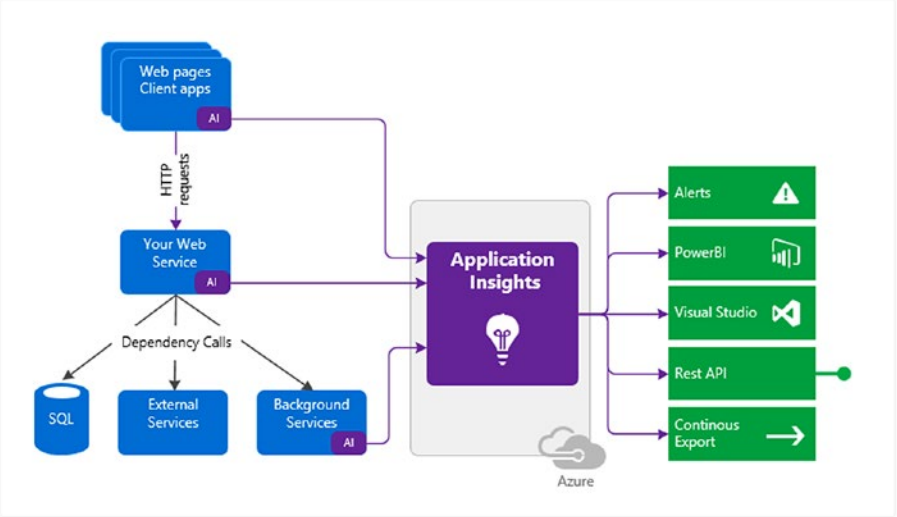

Figure 11-1. *Application Insights architecture overview*

Since Application Insights is an Azure resource, having a valid working Azure subscription is a pre-requisite for working with it. The free Azure trial subscription is available using multiple channels and can be explored.

More information about Application Insights is available at `https://docs.microsoft.com/en-us/azure/application-insights/app-insights-overview`.

Getting Started

In this section, we will add the Application Insights feature and binaries to an existing bot application. We created the `SimpleDialogExample` bot in previous chapters and will use it to show how to monitor and manage using Application Insights.

Open the project `SimpleDialogExample`, right-click on the project, and choose "Application Insights Telemetry," as shown in Figure 11-2.

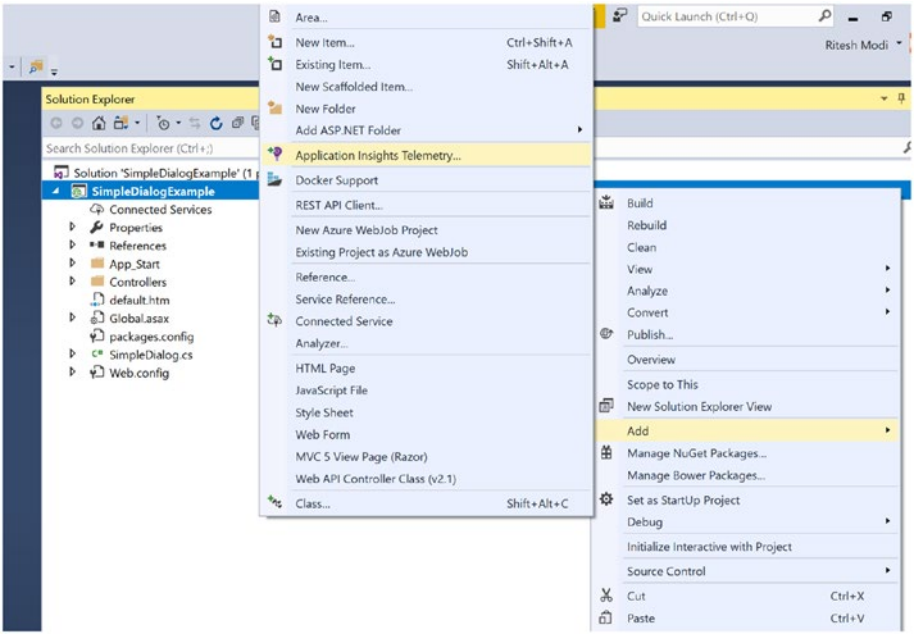

Figure 11-2. *Adding Application Insights feature to bot application in Visual Studio*

This will initiate the process of enabling Application Insights for the bot, as shown in Figure 11-3.

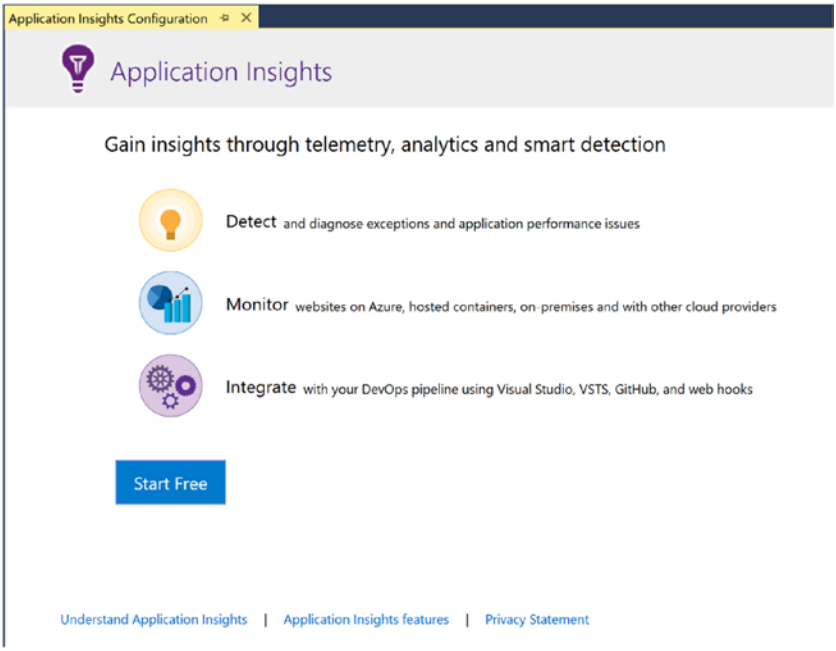

Figure 11-3. *Application Insights start screen in Visual Studio*

Click on the Start Free button to get started. It will show any existing Microsoft account or work or school account. Select an appropriate account, subscription, and resource. It also shows cost information for data transmission to Azure. This is shown in Figure 11-4. The agent at the application side can stop sending telemetry information after 1 GB/month, or they can continue to send.

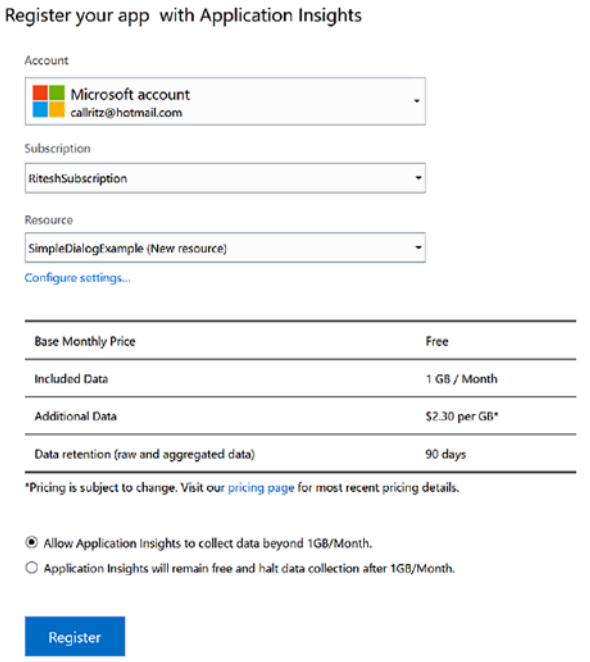

Figure 11-4. *Configuring Application Insights in Visual Studio*

The resource can be further configured as shown in Figure 11-5 by clicking on the "Configure" link and providing a custom resource group name, application insight instance name, and location. The location should ideally be the same as that of the location of the app service hosting the bot.

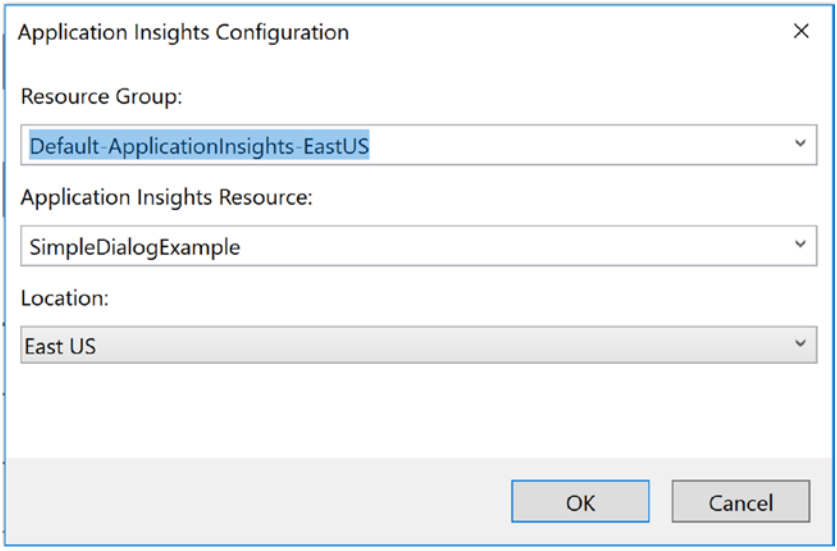

Figure 11-5. *Customizing Application Insights configuration in Visual Studio*

Clicking on "Register" will download the necessary Application Insights binaries and add to the project the required references. The dashboard shows that Application Insights is getting added to the project, as shown in Figure 11-6.

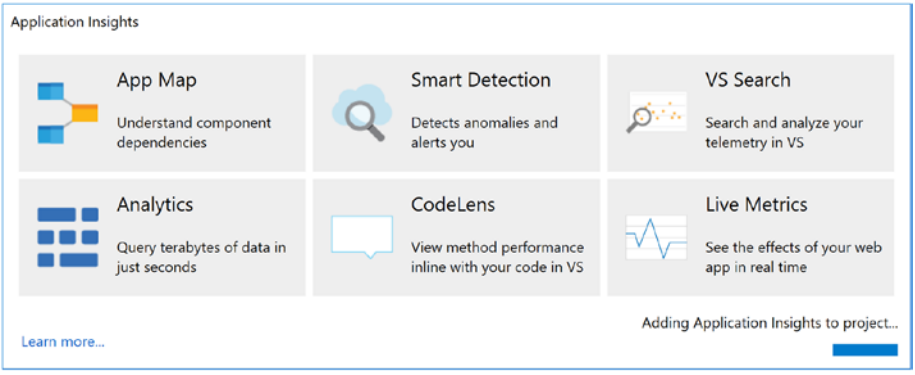

Figure 11-6. *Application Insights dashboard in Visual Studio*

The post Application Insights installation screen is shown in Figure 11-7. Trace collections from the bot can be further configured from this screen, but it can be configured later as well.

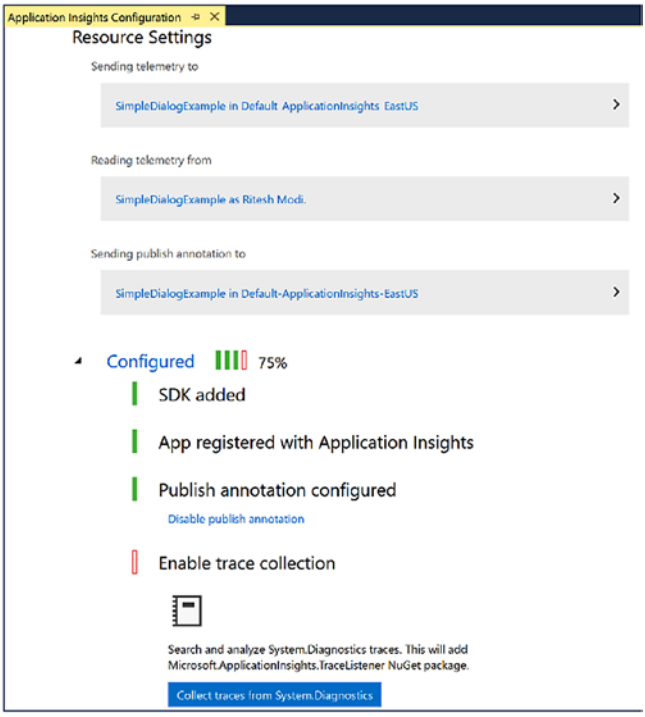

Figure 11-7. *Application Insights post-installation screen and status*

The project structure gets updated with Application Insights artifacts, as shown in Figure 11-8.

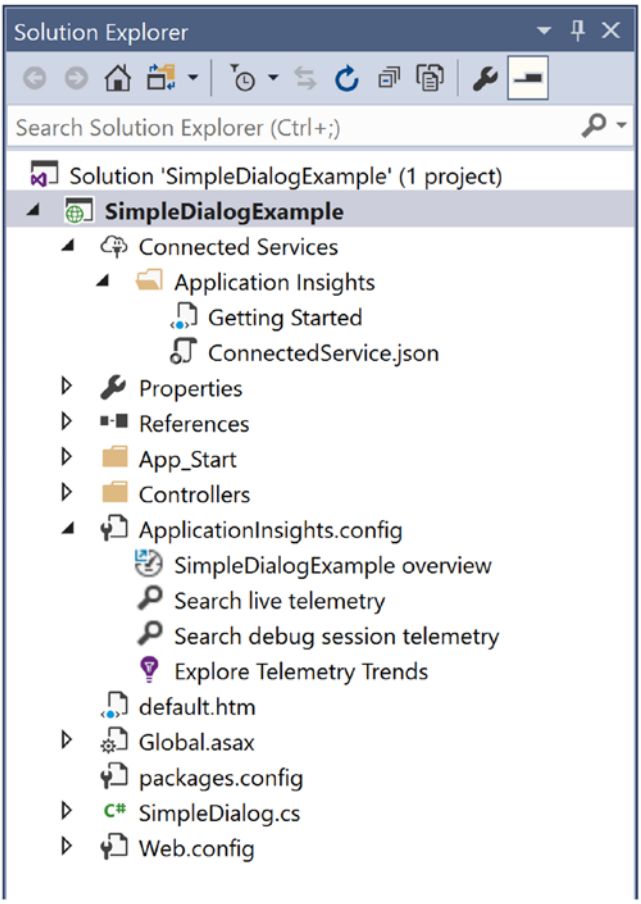

Figure 11-8. *Bot solution structure after installation of Application Insights in Visual Studio*

The Application Insights configuration, portal, and other telemetry information is available by right-clicking on the `Application Insights` folder, as shown in Figure 11-9.

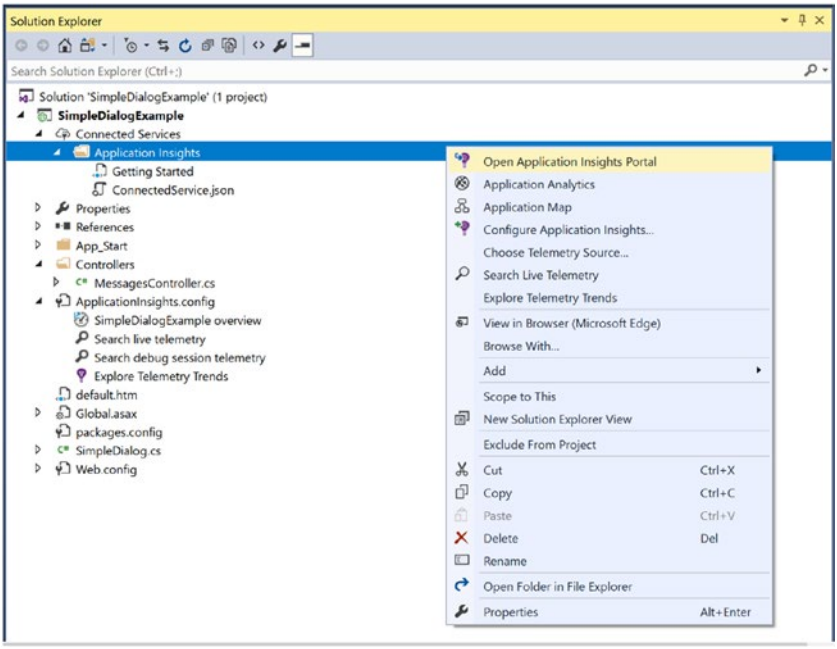

Figure 11-9. *Opening Application Insights portal from Visual Studio*

Now, if we log in to our Azure subscription, an Application Insights resource is provisioned based on the configuration information provided earlier. The Azure Portal showing Application Insights is shown in Figure 11-10.

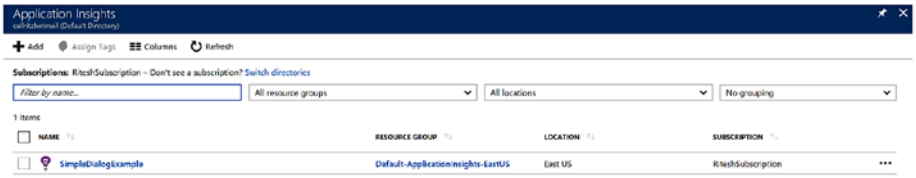

Figure 11-10. *Application Insights provisioned in Azure*

Clicking on an Application Insights resource will take you into its dashboard on Azure Portal, as shown in Figure 11-11.

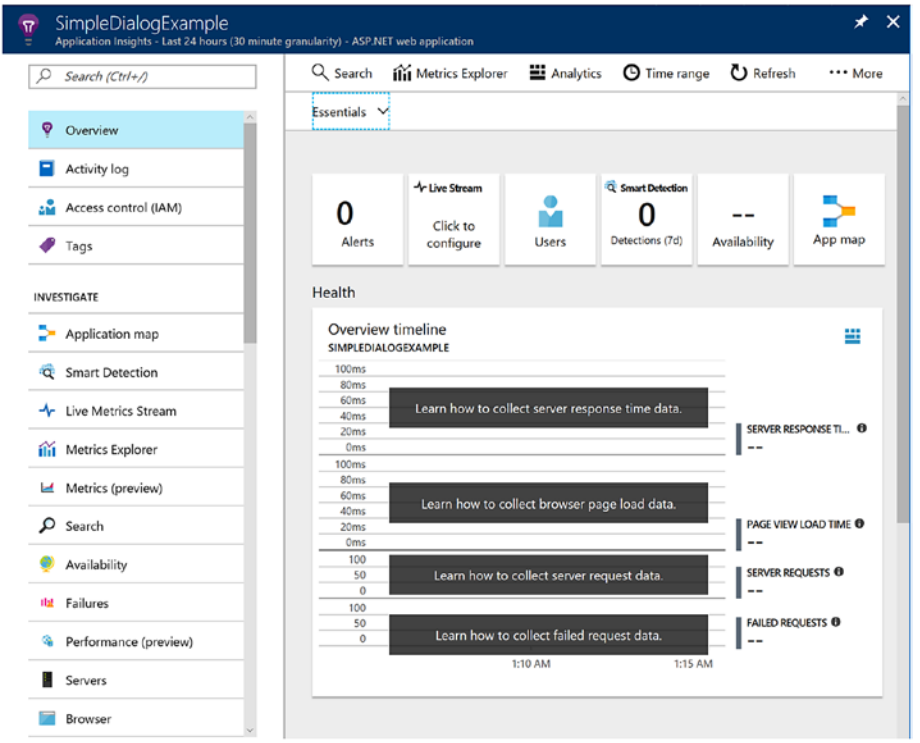

Figure 11-11. *Application Insights configuration options in Azure*

Further configuration can be performed using the Configure Application Insights menu. Trace logs can be enabled from here, as shown in Figure 11-12.

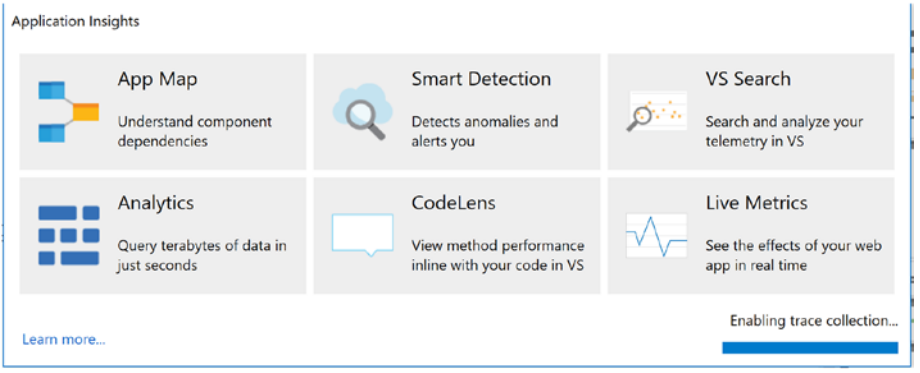

Figure 11-12. *Enabling trace collection using Visual Studio*

Live telemetry information can be viewed using the "Search Live Telemetry" option, as shown in Figure 11-13.

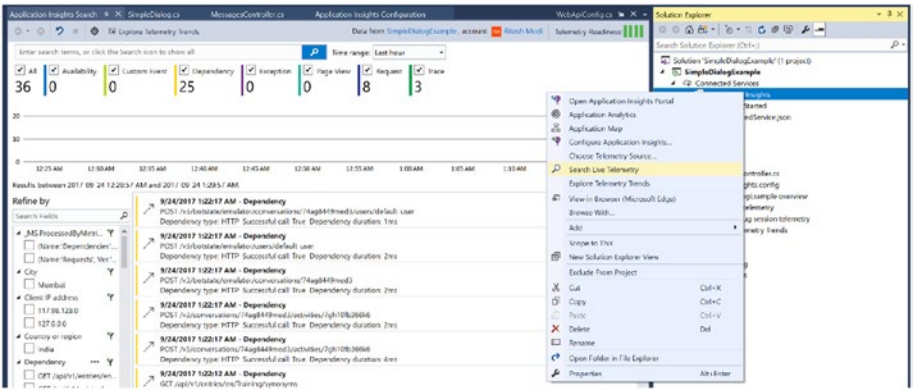

Figure 11-13. *Application Insights telemetry information in Visual Studio*

Important point to realize here is that all telemetry and insight information is available within Visual Studio as well as centrally in Azure Portal. Apart from this, this data can be consumed by custom applications for further custom analysis.

Configuring Application Insights is a first but important step for monitoring the following:

- Request rates, response times, and failure rates
- Exceptions
- Load performance
- User and session counts
- Performance counters such as CPU, memory, and network usage
- Custom events and metrics

However, a bot needs additional monitoring capabilities, and that is the topic of discussion of the next section.

Enable Bot Analytics

Now that basic monitoring is in place, it's time to concentrate on bot-specific monitoring. Microsoft provides a centralized registry of all bots at `https://dev.botframework.com/`. Registering your bot at this location helps it get discovered and provides advanced monitoring functionality.

The first step to enabling advanced analytics is to register the bot on this portal.

Log in to https://dev.botframework.com/ with valid credentials to register your bot. Click on the Create a Bot button from the My Bots menu and fill in the displayed form.

The Configuration section is important, and new AppID and password should be generated, as shown in Figure 11-14.

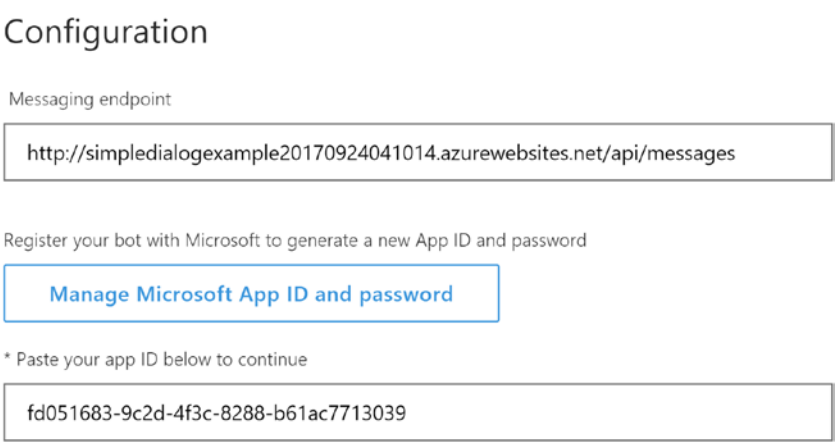

Figure 11-14. *Registering bot at central bot registry*

The Analytics section should be filled with the following:

- Application Insights instrumentation key
- Application Insights API key
- Application Insights application ID

All three values are available from Azure Portal within the Application Insights resource instance.

The Application Insights instrumentation key is available from the Properties menu of Application Insights, as shown in Figure 11-15.

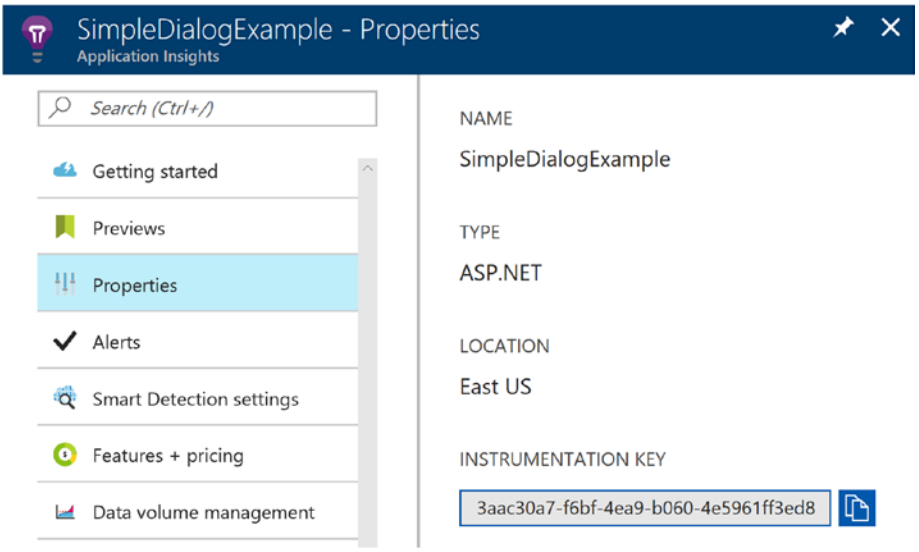

Figure 11-15. *Taking note of Application Insights instrumentation key*

Both the Application Insights API key and the Application Insights application ID are available from the API Access menu in Application Insights (Figure 11-16). You will have to generate a new key by clicking on the Create API Key button (Figure 11-16).

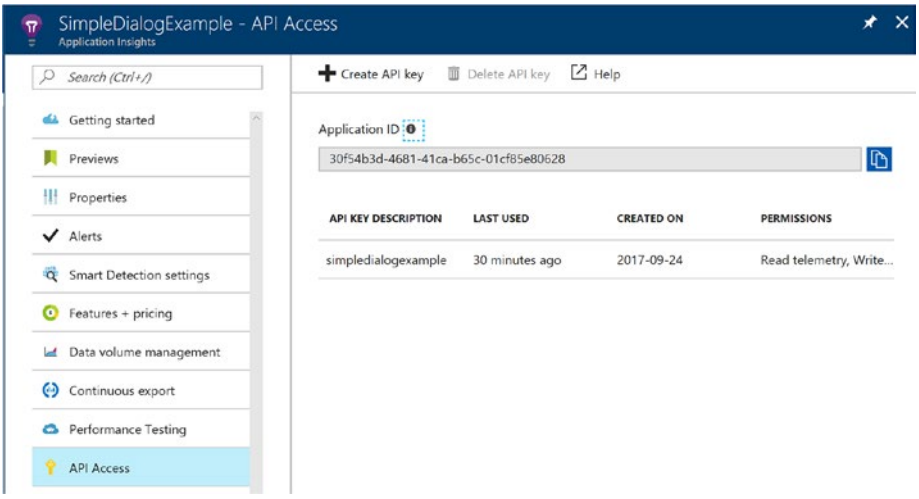

Figure 11-16. *Taking note of Application Insights application ID in Azure Portal*

Both application ID and key should be used to further configure the bot while registering it at Developer Portal, as shown in Figure 11-17.

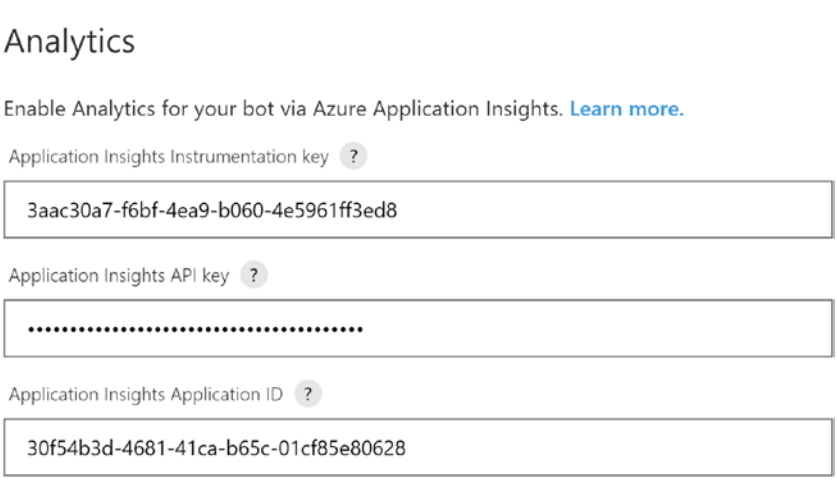

Figure 11-17. *Enabling analytics at bot registration portal*

After registering the bot, the dashboard shows the default channels configured for the bot, as shown in Figure 11-18.

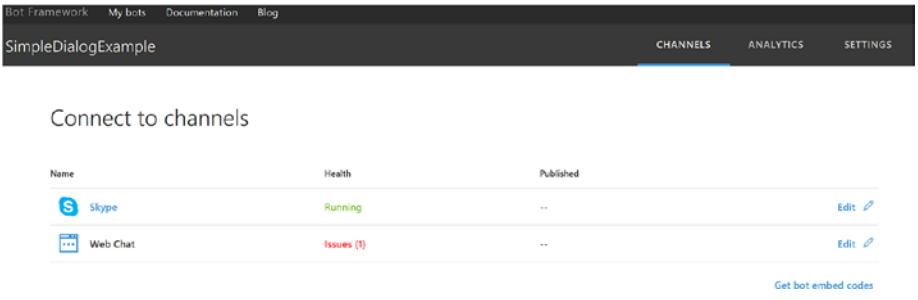

Figure 11-18. *Bot registration dashboard*

The Analytics menu should be enabled now, and clicking on it will take you to its dashboard. Analytics is an extension of Application Insights. Application Insights must be enabled and configured before using the Analytics dashboard. While Application Insights is broad-based and provides service- and application-level instrumentation and analytics, Analytics provides a bot's conversation-level analytics and instrumentation information.

Clicking on the down arrow button on the menu will display the monitoring of all channels, as shown in Figure 11-19. This should be configured based on the channels the bot is hosted on.

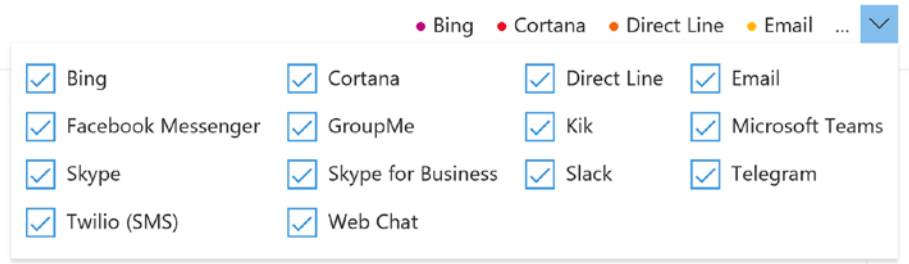

Figure 11-19. *Analytics configuration for available channels for bot appliction*

The administrator can also select the time period for which the data is of interest, as shown in Figure 11-20.

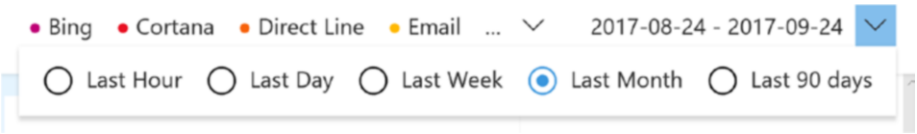

Figure 11-20. *Analytics configuration for data availability time period*

Further, it also provides the number of messages and users using the bot, as shown in Figure 11-21.

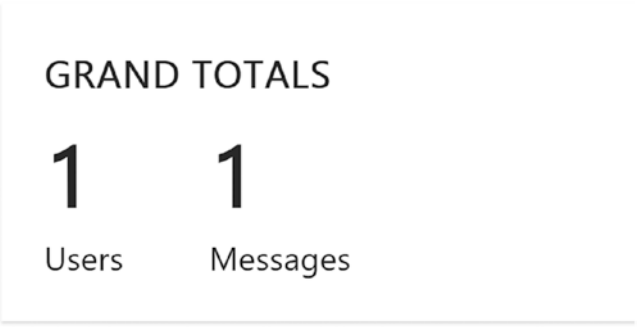

Figure 11-21. *Dashboard showing number of users and messages interacting with bot*

It also provides a nice view of user retention, as shown in Figure 11-22. It shows how many users who were involved in a conversation sent follow-up messages after the first message. The chart is a rolling ten-day window.

RETENTION - % USERS WHO MESSAGED AGAIN (LAST 10 DAYS)

		Days later									
Date	Users	1	2	3	4	5	6	7	8	9	10
9/13/2017	0	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
9/14/2017	0	0%	0%	0%	0%	0%	0%	0%	0%	0%	
9/15/2017	0	0%	0%	0%	0%	0%	0%	0%	0%		
9/16/2017	0	0%	0%	0%	0%	0%	0%	0%			
9/17/2017	0	0%	0%	0%	0%	0%	0%				
9/18/2017	0	0%	0%	0%	0%	0%					
9/19/2017	0	0%	0%	0%	0%						
9/20/2017	0	0%	0%	0%							
9/21/2017	0	0%	0%								
9/22/2017	0	0%									

Figure 11-22. *Report showing number of users who interacted with bot continuously*

The Users chart, as shown in Figure 11-23, shows the number of users on various channels.



Figure 11-23. *Chart showing number of users consuming bot on different channels*

The messages chart, as shown in Figure 11-24, shows the number of messages on various channels.



Figure 11-24. *Chart showing number of messages received by bot on different channels*

Advanced Analytics

Even after configuring both Application Insights and Bot Analytics, there are use cases that are still not covered from a monitoring and insights perspective. Imagine a bot is using LUIS features, and the bot's stakeholders are interested in knowing about the following:

- How well are LUIS intents getting used!
- What are the conversion ratios?
- Execute some sentiment analysis to find users' satisfaction using bot and company services.

There is an open source tool called ibex available at `https://github.com/CatalystCode/ibex-dashboard`. It is an open source tool built using Node.js and react. It is built on top of Application Insights and so it is a must to be provisioned before deploying this tool.

It provides relevant information on intent usage, sentiment analysis, channel usage and activity, and the rate of messages, as shown in Figure 11-25.

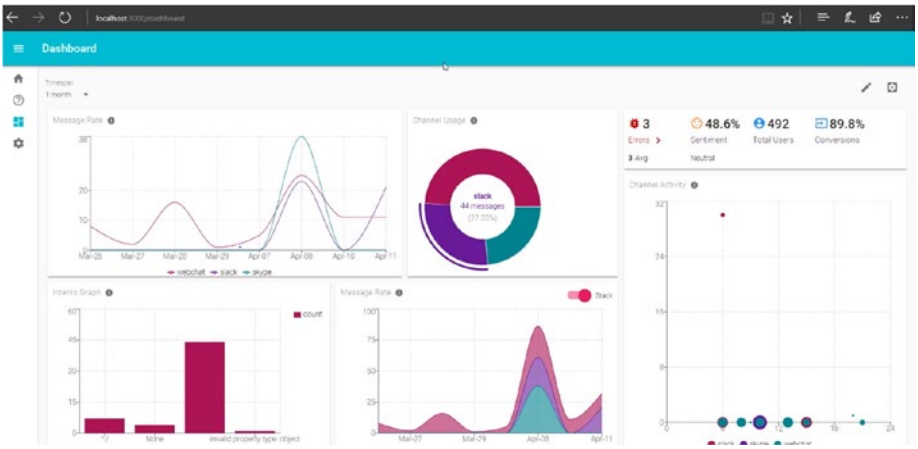

Figure 11-25. *Bot advanced analytics tool iBex dashboard*

Summary

Bots have become the new face of organizations, and it is extremely important to monitor and manage them well. It can seriously dampen an organization's value if the bot does not perform and interact as expected. Moreover, what cannot be measured, cannot be improved. Ensuring that the bot keeps learning from its past data and telemetry information and keeps adapting and improving itself is the key for its success in engaging its users and customers. Azure provides Application Insights and advanced bot analytics to ensure that organizations can fetch and view appropriate telemetry information from the bot's platform and the bot itself to find any deviation and degradation. It helps to take proactive as well as reactive steps to keep bots alive and kicking. This is an important activity, and if you are doing any serious bot development and deployment, the monitoring of your bot should not be ignored.

Index

A

Autofac Inversion of Control (IoC), 169
Azure App Service, 38
Azure bot service
  boilerplate templates, 12
  configuration, 11
  development experience, 13
  LUIS bot, 13
  proactive Bot, 14
  QnA bot, 13
Azure cognitive services
  AI + cognitive services, 239
  API classification, 235–236
  API keys, 238
  Bing Web Search, 240–241
  Bot framework ecosystem, 243
  chat and smart bots, 233
  handling errors
    API endpoints, 254–255
    Bing computation search query, 256
    bot application, 253
    error-response object, 253
    optical character recognition, 256–260
    QPM and QPS, 254
  Postman HTTP client, 242
  REST APIs, 234
Azure Functions, 12
Azure machine learning (ML), 234

B

Bing Speech API, 123, 141, 150
  account creation, 143
  audio stream to server, 145
  Azure Storage account, 147
  billing, 150
  keys, 144
  NuGet package, 144
  OnConversationError event, 145
  OnRecordCompletedAsync event handler, 144
  PhraseResponse, 146
  Power BI report, 149
  pricing options, 142
  registered callback, 146
  speech recognition mode, 145
  STT response, 147
  on Windows 10, 148
Bing Web Search
  Bing Speech API, 245, 247
  bot application, 249–250
  bot's interface, 244
  image search, 251–252
  MessageReceiveAsync method, 245
  Microsoft Cognitive Services, 244
  query parameters, 247–249
  search response, 251
Bot Builder SDK, 19, 31, 52
Bot Connector Service, 19, 52, 148
Bot Developer Portal, 52
Bot Emulator, 110, 139, 189
  attachment, 108, 111
  Carousel in, 114
  Hero card, 112
  Thumbnail card, 113
BotId property, 42
Bot operations
  abuse, 17
  analytics configuration, 274–275
  Application Insights
    architecture, 261–262
    artifacts, 267–268
    configuration, 265, 271
    dashboard, 266

© Srikanth Machiraju and Ritesh Modi 2018

S. Machiraju and R. Modi, *Developing Bots with Microsoft Bots Framework*,
https://doi.org/10.1007/978-1-4842-3312-2

Bot operations (*cont.*)
Azure Portal, 269–270, 273
bot registration, 274
channels, 8
chat and AI, 6
chat-based applications, 5
conversations, 181
CUI-based software application, 6
customizing application, 266
directory, 8–9
enabling trace collection, 270
face AI bot, 10–11
features, 181
frameworks, 16
IOT, 16
LUIS features, 277
open Application Insights portal, 268–269
properties menu, 272–273
registering bot, 272
Search Live Telemetry option, 271
SimpleDialogExample, 263
start screen, 263–264
summarize, 9
time period, 275
user interface, 181
users and messages, 275–277
Bots, .NET Core, 19
application life cycle, 28
activity object, 30
best practices, 20
bootstrapping activities, 29
Bot Builder SDK, 31
conversations, 22–23
events, 29
Global.asax.cs, 28
NuGet package, 31
POST request, 29–30
pre-built method, 31
root dialog, 22
architecture, 31–32
authentication, 32–33
building, 33
Appointment class, 35
conversation, 37
conversationUpdate, 34
prompt attribute, 36
RootDialog, 33–34
configure channels, 45–46
Developer Portal, 47
Skype bot, 47–50
web chat, 50–52
deploy to Azure, 38
Azure App Service, 38
creating account, 39
diagnostics and logging details, 41
doctor appointment bot, 42
publishing, 38
steps, 38–41
Visual Studio Web Deploy, 40
development environment
applications, 24–25
Bot Emulator, 25–26
debugging bot application, 27–28
testing bot, 26–27
tools and SDKs, 23–24
registering, 42
BotId property, 42
configuration section, 43–44
Developer Portal, 42
ID and password, 44–45

C

CachingBotDataStore object, 165
ChannelData property, 97
Channels, 75
bot, 75–77
channel data, 78, 80
chat bot using email client, 80
application, 80
Bot Emulator, 83
configuration, 85
getWeatherdetails helper function, 82
MessageReceivedAsync function, 81, 82
openweathermap.org, 86
serializing, 80–81
chat bot using slack channel and API
Developer Portal, 87–88
multi-dialog bot, 89–94
onboarding, 95–96
remote debugging on development machine, 96–97
sample response, 88
on Windows 10, 87

configuration, 76
Get bot embed codes link, 77
issues, 77
Chat bot
using email client, 80
application, 80
Bot Emulator, 83
configuration, 85
getWeatherdetails helper function, 82
MessageReceivedAsync function, 81, 82
openweathermap.org, 86
serializing, 80–81
using slack channel and API
Developer Portal, 87–88
multi-dialog bot, 89–94
onboarding, 95–96
remote debugging on development machine, 96–97
sample response, 88
on Windows 10, 87
Cognitive services, 204
Conventional UI, drawbacks, 3–5
Conversational user interface (CUI), 5–7
Conversations, 99
activity, 101
bots with
attachments, 105–111
buttons, 116
Carousal, 114–115
Hero card, 111–112
prompts, 116–119, 121
Thumbnail card, 113–114
channels, user and bot, 102
messages, 100–101, 103
under hood, 103, 104
Creating intelligent bots
with dialogs, 220–223, 225–230, 232
without dialogs, 215, 217–220
CUI-based software application, 6

D

Dialog bot, building, 184
MessagesController.cs, 188–189
SimpleDialog.cs, 184, 186–188
Dialog context, 183
Dialog model, 181
IBotContext, 183
IBotData, 182
IBotTouser, 182
IDialogStack, 182
Dialog stack, 183
Direct Line API, 15–16

E

EchoBot, 26

F

FormBuilder, 199
conditional fields, 200
customizing the prompts, 199, 200
FormFlow, 195
data types, 196
features bot, 199
FormFlow bot, 196–199

G, H

Generic language models, 203

I, J, K

Interactive Voice Response (IVR), 126
Internet-based voice calling (VOIP), 123
Internet of Things (IOT), 16
IVRBot class, 128
IVROptions, 135

L

Language Understand Intelligent Service (LUIS), 203, 204
entities, 205
features, 206
intents, 204
add entities, 208
add intents, 207
add utterances, 209
create application, 206
publish, 210
sample application, 211–214
train and test, 210
utterances, 205

M

Microsoft Bot framework, 7–8
Microsoft Cognitive Services, 52
Multi-dialog Bots, creating, 189
  antonym.cs, 194
  combined dialogs, 190
  Façade dialogs, 190
  MessagesController.cs, 195
  nested dialogs, 190
  RootDialog.cs, 192–193
  scenario, 190
  solution, 191
  support.cs, 195
  synonym.cs, 194

N

Natural language processing (NLP), 203
  cognitive services, 204
  creating intellegent bots, 215, 218–227, 229–230, 232
  LUIS, 204–214
Ngrok, 139–141
Node.js
  App Service settings, 72
  attachment prompt, 66
  Azure, 69–73
  bot channels, 73
  Bot Emulator, 56, 58–59
  bot registration, 71
  Confirm() method, 65
  debugging, VS code, 60
  deployment credentials, 70
  development environment, 54
  dialogs, 61
  input choice, 64
  IPromptOptions.listStyle property, 64
  LUIS/Azure Cognitive Services, 53
  Message builder class, 67–68
  Message.sourceEvent() method, 67
  node_modules, 56
  NPM command prompt, 54
  NPM init, 55
  number prompt, 66
  Pizza Bot built, 63
  prompts, 62–63
  routing pattern, 57
  session.beginDialog(), 62
  start bot application built, 58
  state, 68–69
  text Input, 65
  time prompt, 66
  UniversalBot class, 57
  user interface (UI), 64
  Visual Studio Code, 54–59
  Waterfall model, 62
node_modules, 56
NoSQL APIs, 166

O

Optical character recognition (OCR), 235, 256–260

P

Platform as a service (PaaS), 261
Proactive Bot template, 14

Q

Quota per month (QPM), 254
Quota per second (QPS), 254

R

Recipient property, 102
ReplyToID property, 103
Rootdialog, 183

S, T

Service-level agreements (SLAs), 166
Skype calling bots, 123
  building, 126
    for Business bot, 76
    callback event, 128
    CallingController, 127–128
    CallingConversation class, 128
    Contoso, 126
    Controllers folder, 127
    interface ICallingBot, 129–130
    IVR, 126
    IVRBot service, 128, 129
    OnIncomingCallReceived event, 128
  debugging using Ngrok, 139–141
  enabling calling, 125
  overview, 124

sequence of events, 130
Application Insights and Azure blob/table storage, 138
callable bot application, 131
CallingBotService events, 132
CollectDigits property, 135
Dispose method, 138
GetPromptForText method, 133
InitialSilenceTimeoutIn Seconds property, 134
multi-user environments, 133
OnIncomingCallReceived event, 133
OnPlayPromptCompleted event, 134
OnRecognizeCompleted event, 135
OnRecordCompletedAsync event, 137
prompt class, 133–134
recognize action, 134
workflow, 130
speech-to-text using Bing Speech API, 141
account creation, 143
audio stream to server, 145
Azure Storage account, 147
keys, 144
NuGet package, 144
OnConversationError event, 145
OnRecordCompletedAsync event handler, 144
PhraseResponse, 146
Power BI report, 149
pricing options, 142
registered callback, 146
speech recognition mode, 145
STT response, 147
on Windows 10, 148
use cases, 124–125
Skype Calling SDK, 149
StartAsync method, 159
StateClient object, 153–154, 180
Stateful service, 151
Stateless service, 151
State management, 151
control over state with dialogs, 162
asynchronous functions, 163
Bot Application template, 162
cityname property, 163
MessageController code, 165
rating property, 163
StartAsync method, 162
StateSampleDialog, 162
username property, 163
Cosmos DB, 165
custom state data store, 166–173
IBotDataStore interface, 166
overview, 166
table storage, 173, 175–179
state service, 153–154
stores for, 152
Conversation data, 152
Private Conversation data, 152
User data, 152
storing and retrieving state
using dialogs, 158–160, 162
using StateClient, 155–157

U

User interface (UI), 2–3, 19

V, W, X, Y, Z

Virtual machine (VM), 38