

Microsoft Bot Framework

Succinctly[®]

by Ed Freitas

Microsoft Bot Framework Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2017 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: John Elderkin

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the Succinctly Series of Books	6
About the Author.....	8
Acknowledgements	9
Introduction	10
Chapter 1 Bot Framework Overview	12
Introduction	12
Bot Builder SDK	12
Bot Connector	14
Getting started	15
Defining a bot application.....	19
Summary.....	20
Chapter 2 Our First Bot	21
Introduction	21
Thumbnail bot setup	21
Using the Webthumbnail.org API.....	28
The bot's capture logic.....	28
Summary.....	37
Chapter 3 Publishing Our Bot.....	39
Introduction	39
Registration basics.....	39
Publishing our Bot.....	41
Finishing the registration process	45
Re-publishing our bot.....	47
Adding Skype as a channel	49

Summary.....	51
Chapter 4 The QPX Express API	53
Introduction	53
The QPX Express API	53
Signing up for QPX Express	54
Setting up our VS project	58
AirfareAlertBot core logic	62
Summary.....	63
Chapter 5 Airfare Alert Bot.....	64
Introduction	64
Web.config	64
WebApiConfig.cs	65
Mapping airports to IATA codes	66
Handling conversational flow	72
Custom flow validations	78
Internal validation helpers	87
The bot's central hub	93
Running our bot	101
Wrapping up.....	108

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas currently works as a consultant on software development applied to customer success, mostly related to financial process automation, accounts payable processing, invoice data extraction, and SAP integration.

He has provided consultancy services and engineered, advised, and supported projects for global enterprises such as Agfa, Coca-Cola, Domestic & General, EY, Enel, Mango, and the Social Security Administration. He's also been invited to various companies such as Shell, Capgemini, Cognizant, and the European Space Agency.

Ed was recently involved in analyzing 1.6 billion rows of data using Redshift (Amazon Web Services) to gather valuable insights on client patterns. He holds a master's degree in computer science. He also created [Pluglock](#).

When not working, he enjoys soccer, running, traveling, life hacking, learning, and spending time with his family. You can reach him at <https://www.edfreitas.me>.

Acknowledgements

My thanks to all the people who contributed to this e-book, especially Tres Watkins, Darren West, and all the Syncfusion team that helped make it a reality. Graham High, the manuscript manager, and James McCaffrey, the technical editor, thoroughly reviewed the e-book's organization, code quality, and overall accuracy. Thank you all.

Introduction

We are always challenged to stay current and learn new technologies in programming and software development, and that is one of the reasons I love it. If you are like me and have a passion to discover new things, embracing new frameworks is as compelling as it is exciting.

For me, one of the most exciting and impressive breakthroughs to come out of the Microsoft [Build](#) 2016 conference was the announcement of the [Microsoft Bot Framework](#), which allows us to build our own connected and intelligent bots.

Bots, or automated agents, are programs that let users interact with “smart” or “intelligent” solutions as though they are conversing with another person. In this way, bots have a certain degree of sci-fi and popular-culture iconology associated with them.

Bots are often indirectly related to a subset of [artificial intelligence](#) (AI) that exists within an evolution of computing that allows users to interact with machines in natural conversational styles in a remarkably human way.

Although the scope of AI is much broader than natural conversations, a certain degree of machine intelligence is necessary to create an automated agent able to coherently reply to user queries in a defined way and within a delimited scope of a particular subject.

Microsoft introduced the Bot Framework in order to lower the entry barrier for wider adoption and usage of bots throughout the enterprise and to allow developers to write them in a much faster way.

By providing a bridge to the most common communication platforms used nowadays and also through smart [Cognitive Services](#)—bringing some machine intelligence closer to the reach of developers in general—the Bot Framework is well positioned to become one of the top AI conversational frameworks available on the planet.

As Microsoft describes it, the Bot Framework allows you to build and connect intelligent bots with your users naturally, whether those users are on Skype, Slack, Office 365 mail, or other popular conversational services.

As enterprises start to embrace various forms of AI and it becomes more ubiquitous throughout the IT landscape, bots will be an increasingly important way for users to interact with service providers and organizations.

Developers will find that basic I/O language and dialog skills are still necessary when they want to write a bot, and they must be able to connect to communication platforms where users already exist. This is where the Bot Framework adds needed value to the developer community.

Throughout this e-book, we'll explore the Bot Framework to understand its capabilities and how we can use it to build interactive, conversational bots using common communication channels we already know and love so that we can provide our users a better experience. We'll explore some handy cases and write some cool code to go with it.

In order to maximize your learning potential, you should already have some basic knowledge of C#. Visual Studio 2015 will be used as the development environment.

By the end of this e-book, you should have a good understanding of the capabilities that the Bot Framework has to offer, and you should also feel comfortable writing bots on your own and putting your imagination to work.

Although this e-book will mostly focus on technical aspects, a good dose of creativity is required in order to create useful bots for compelling use cases.

The journey should be fun to follow, the examples easy to implement, and the skills useful for helping you understand what is possible with this amazing framework from Microsoft. Have fun!

Chapter 1 Bot Framework Overview

Introduction

The Microsoft Bot Framework is a collection of services and APIs that allows developers to build and deploy high-quality bots for their users by using existing and well-known conversational platforms.

The main aim of the Bot Framework is to help developers solve the most common problems they'll encounter when writing a bot—creating a basic I/O mechanism and infrastructure to go with it, adding language and dialog skills, handling performance, responsiveness, and scalability, and being able to connect to users in any conversational platform and language of their choice.

In essence, the Bot Framework provides developers what they need in order to build, connect, manage, and deploy bots that interact as naturally as possible wherever users might be, such as [Skype](#), [Slack](#), [Messenger](#), [Kik](#), [Office 365](#) mail, and other popular services.

The Bot Framework is made up of several components that work seamlessly together, such as the Bot Builder SDK, Bot Connector, and Bot Developer Portal. Integration with Azure App Services is provided out-of-the-box.

The Bot Framework also works nicely with the Language Understanding Intelligent Service ([LUIS](#)) from Microsoft [Cognitive Services](#), which is a great way to provide bots with language understanding capabilities by allowing users to experience natural conversations.

The main objective in this chapter is to briefly go through some of these components to get a better understanding of how the Bot Framework works—how each part ties together, and from there, how to use this as an initial step in getting started with the framework—in particular, how to get started and create the foundations for our bots.

Sounds exciting. Let's get rolling!

Bot Builder SDK

Bot Builder is an [open source SDK](#) hosted on GitHub that provides developers with the basic building blocks for writing bots that are interactive and connect to multiple conversational platforms.

Bot Builder includes native libraries in both C# and [Node.js](#), and it also provides a [RESTful](#) API that can be used from any other programming language.

Microsoft has divided the Bot Framework into several components because, as you will see, it's one thing to write a bot and quite another to write a great one.

Being able to write a great bot requires being able to manage conversational state in a relatively easy way.

One of the goals of the Builder SDK is to act as a conversational logic provider. In other words, when you build a bot, you'll have logic that is directly related to the actions that your bot will execute and you should have logic that handles the state of the conversation.

Handling the state of a conversation is not a trivial task, and this is the main issue this SDK solves—making it easier for developers to achieve conversational state awareness.

So, the logic of your bot should be divided into two sections—one section that contains the bot's own logic and another section that manages the state of the conversation and is handled by this SDK.

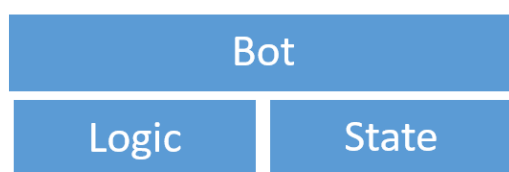


Figure 1.0: A Bot's Own Logic and Conversational State

In its most basic form, a bot should be able to get an input text string, process it, and return an output response. However, a bot that does only this is not really managing any conversational state at all.

Not handling state is fine for very simple bots, but really interesting ones—especially ones that handle multistep navigation, collect data, or require permissions—need a certain degree of complexity, to the point that if you want to manage on your own (inside your bot's own logic section), doing so can become extremely difficult and cumbersome.

Therefore, it is quite important to separate conversational state management from the rest of the bot's logic, and here is where this SDK comes in handy.

As the SDK also includes features for handling dialogs and form flow, the subset of the SDK focused on handling conversational state is often referred to as the Bot State service.

At a very high level, the SDK provides these features:

- A flexible dialog system.
- Built-in dialogs for simple things such as Yes/No, strings, numbers, and enumerations.
- Built-in dialogs that utilize powerful AI frameworks such as LUIS.
- Built-in statelessness that helps bots scale on demand.
- FormFlow: Being able to generate a form flow-based conversation from a C# class.

To get started with the SDK, first we must have a good understanding of the other major component of the Bot Framework, called the Bot Connector. Let's look at this.

Bot Connector

As its name implies, the Bot Connector is a service that allows you to connect your bot with multiple communication channels.

When you write a bot using the Bot Framework and publish it on the Internet using Azure App Services, the Bot Connector's role is to forward messages from your bot to a user and from the user back to your bot by using any of the communication platforms you might have chosen that are available and supported by the service.

In order to use the Bot Connector, you must have a Microsoft Account (Hotmail, Live, Outlook.com) so that you can log into the Bot Developer Portal, which will be used to register your bot.

You must also have an Azure-accessible RESTful endpoint exposing a callback for the Bot Connector service—this can be accomplished by deploying the bot to Azure App Services with Visual Studio (which we will explore later).

Finally, you must also have a developer account with at least one conversational platform (e.g., Skype) that will be used by the bot to communicate with users.

To get a better visual understanding of the Bot Connector, think of it as the central node in a graph that links all other nodes together. Each other node within the graph is a conversational platform supported by the Bot Framework that your bot can communicate with (i.e. send and receive messages).



Figure 1.1: The Bot Connector and Conversational Channels (Figure Courtesy of Petri.com)

These conversational platforms are channels (services) that the Bot Connector is able to connect in order to route messages and keep track of sessions that your bot will be using.

In other words, your bot is nothing more than a web service exposed through a public URL that relies on the Bot Connector to access various communication channels.

Here's another visual way of understanding what the Bot Connector does.

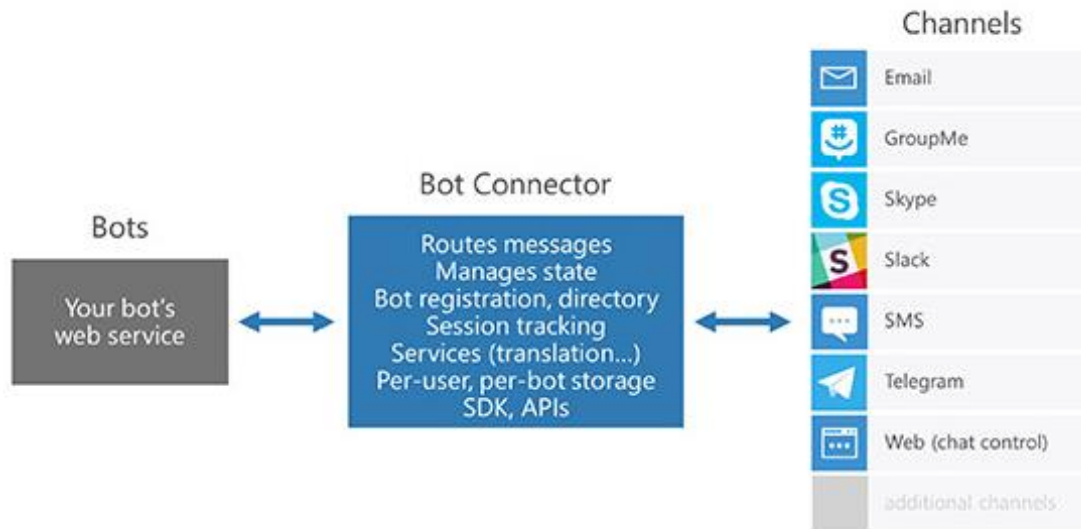


Figure 1.2: How the Bot Connector Binds Bots to Channels (Figure Courtesy of [Microsoft](#))

So, this gives us a clear indication of how the Bot Connector acts as a glue and ties channels to bots. It behaves as a sort of central communication hub between all parties.

All of this sounds exciting. Let's now explore some basic steps to get started.

Getting started

There are two native libraries for developing bots with the Bot Framework. One uses C# and the other uses Node.js.

Throughout this e-book, we'll learn by doing things, and the best way to achieve this is by creating different examples of various types of bots, each one focused on a specific use case. We'll be using C#, and all the development will be done using Visual Studio.

First, download and install [Visual Studio](#) 2015 if you don't already have it on your machine.

At the time of this writing, Visual Studio 2017 is in "release candidate" mode but has not yet been released. I recommend you use VS 2015 even if VS 2017 is available when you read this.

When you have it installed, make sure to update all Visual Studio extensions to their latest versions. This can be done by going to **Tools > Extensions and Updates > Updates**.

With all the VS updates installed, you'll need to download and install the [Bot Application template](#).

If directly clicking on the link doesn't start the download, type <http://aka.ms/bf-bc-vstemplate> directly into the address bar of a web browser.

Save this zip file to your Visual Studio templates folder, which is usually located under `%USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C#`, as we can see in the following figure. Please note that you should not unzip the file!

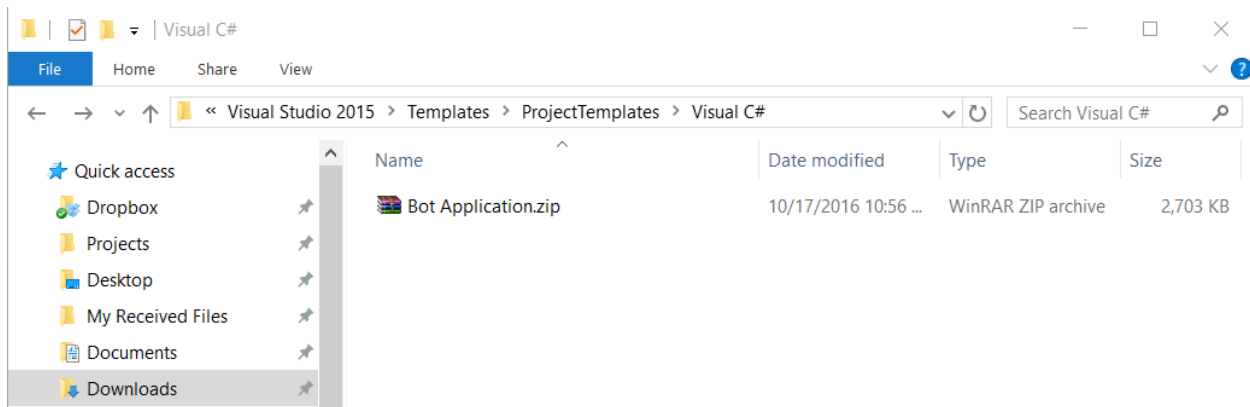


Figure 1.3: The Bot Application Template Saved in the VS Templates Folder

If you now open Visual Studio and type **bot** in the search bar when creating a new project, you'll see the Bot Application template ready to be selected. Cool! You can also find the template directly at the **Installed > Templates > Visual C#** area.

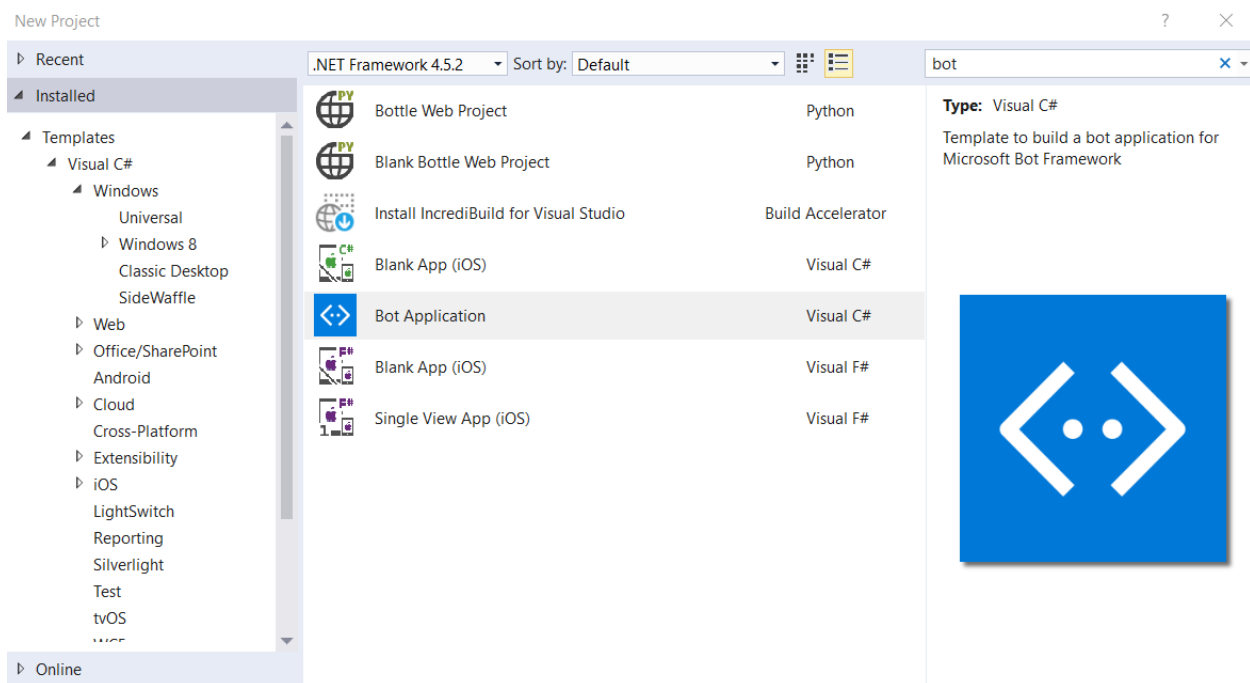


Figure 1.4: The Bot Application Template Ready in VS

We now have our Visual Studio Bot Application template ready, and you can create a test application with it (or you can do this later).

Before writing any code, you should also install the [Bot Framework Emulator](#) because it will be very useful when testing the application.

This emulator lets us make test calls to our application as if it were being called by the Bot Framework itself. Pretty awesome.

With this emulator, we can send requests and receive responses from our application to an endpoint and vice versa. This is quite a handy tool during development.

The Bot Emulator for Windows can be installed from <https://aka.ms/bf-bc-emulator>.

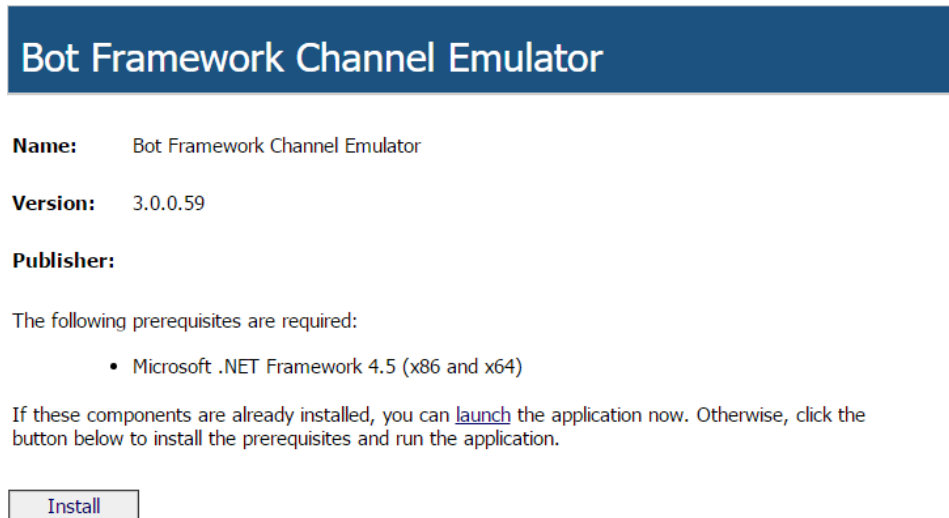


Figure 1.5: Bot Framework Emulator Install Site

When you click Install, a setup.exe file will be downloaded to your machine for you to run. Please note that when you run the installer, you should see something like the screen in Figure 1.6 (although the details might be a bit different by the time you read this).

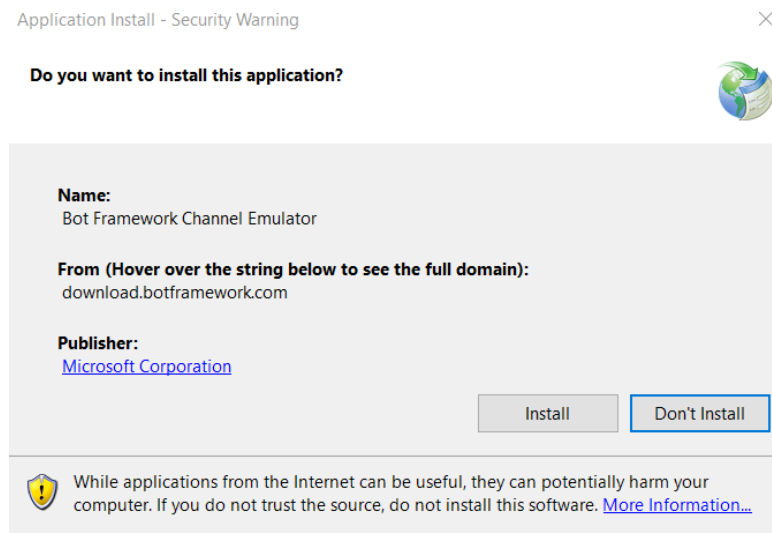


Figure 1.6: Bot Framework Emulator Application Setup

To start the installation process, click **Install**. The installation process takes less than a minute, and you should see it progress as in Figure 1.7.

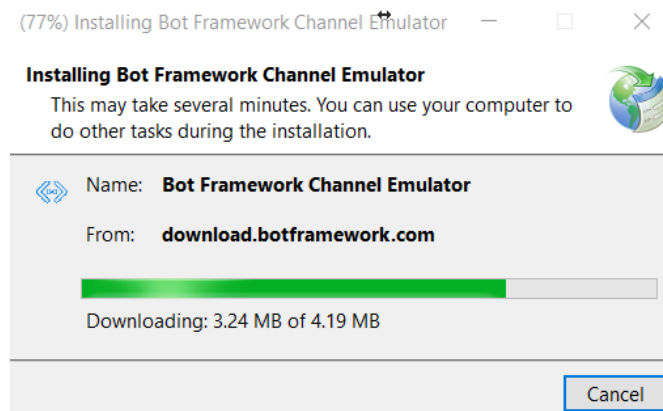


Figure 1.7: Bot Framework Emulator Application Installation

When the installation process has finished, the emulator automatically starts, and you should see something that resembles the following screen.

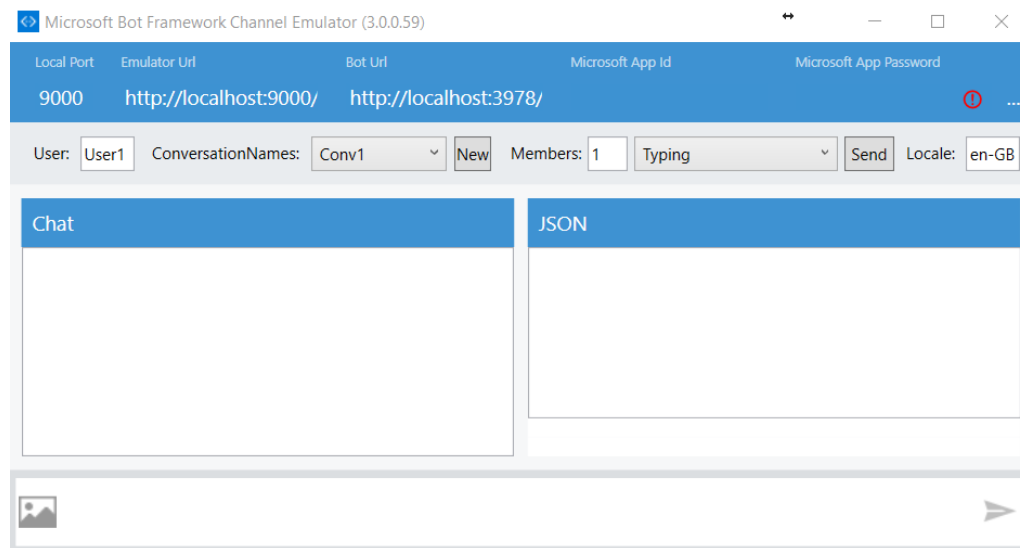


Figure 1.8: Bot Framework Emulator Running

For now, you might close or minimize this window, as we'll be using it later. If you close it, you'll always be able to find it under the Windows Start menu with the name Bot Framework Channel Emulator.

With this done, next let's get an Azure account. You'll need this when we deploy our bot application.

If you don't have an Azure account, you should sign in or sign up for Azure with a Microsoft account. You can do that by visiting <https://azure.microsoft.com>.

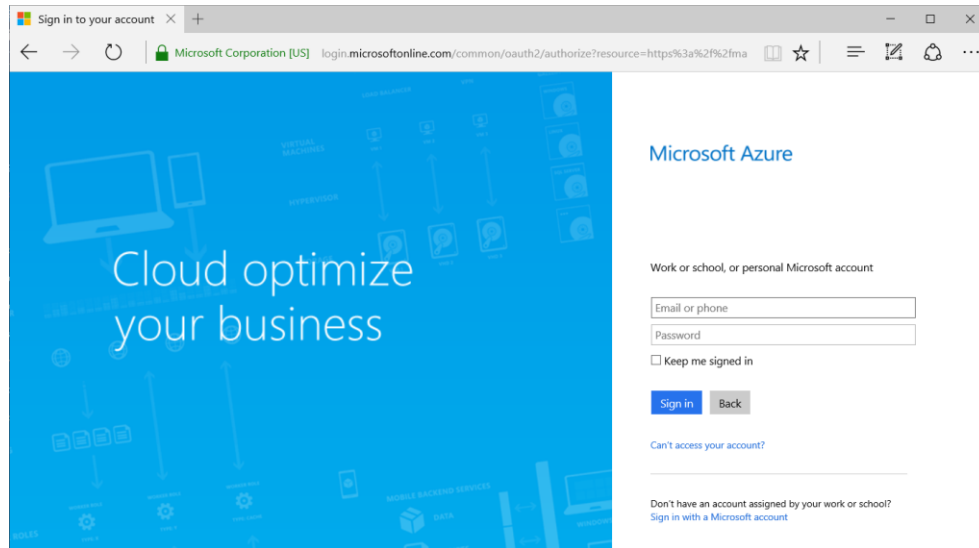


Figure 1.9: Microsoft Azure Sign-In Page

Assuming the Azure account is ready, let's create our first bot application.

Defining a bot application

It is exciting to have reached this point! We are now ready to create our first bot application. So, what will our bot do? Good question.

The hardest part of creating a bot is actually figuring out what the bot will do. In essence, a bot should be useful even though it might not necessarily be conversationally fluid.

By useful, I mean that a bot should be able to perform a task that is either not easily solvable by using a search engine, or it should perform a task that might prove tedious, labor intensive, or simply repetitive.

The underlying concept of a bot is that if you give it a command, it should give back a result that is easier and faster to get than if you look it up yourself on the Internet. There are a lot of bots out there that do all sorts of things, however not all of them adhere to this fundamental principle.

Almost any type of bot can be written—there's no technological barrier that prevents us from doing that. Even creating conversationally fluid ones is easier than ever with APIs such as [LUIS](#). The key factor is creating something useful that ultimately saves users time. Taking this into account, what can our first bot do?

I have an idea. Let's create a bot that gives us a thumbnail image of any website we type in.

For a moment, let's pretend we work in market research and want to keep track of how frequently our competitors' websites change. A great way to do this is by creating thumbnails of their main pages throughout different points in time.

Obviously, we can do this manually by going to the Internet and signing up for a myriad of sites that offer this service. However, doing this manually will become quite tedious, so it's definitely a task that should be automated with a bot. Now we've got something going. Awesome!

So, we've defined what our first bot application will do, which also fits nicely with the usefulness principle we've recognized, and we are ready to start building it. We'll do that in the next chapter.

Summary

This chapter introduced the Bot Framework, explained its parts and how to set it up, and defined the basic principle that an automated agent application using this framework should aim to accomplish.

With that outlined, we are ready to dive into the technical details of how to write bots and use them to carry out interesting and useful activities that could help us save time and prevent us from engaging in repetitive and tedious tasks.

In the next two chapters, we'll create our first real bot application. It won't be conversationally fluid, but it will nevertheless solve an interesting problem and be useful.

Following that, we'll create an even more exciting, interesting, and useful bot application—one that is more conversationally fluid.

Chapter 2 Our First Bot

Introduction

In the previous chapter, we explored the basics of the Bot Framework and its parts. At this point we have a good understanding of what the framework can allow us to achieve, but we still haven't really had any action with it.

That's about to change. Now, we'll create our first bot—a very interesting one. Not particularly smart to the extent that the word “bot” itself tends to imply, but one that performs a rather nontrivial and tedious task.

We'll start digging into the code right away, and as we go, we'll expand our understanding of the framework and use this later to build an even more complex and interesting bot app.

Thumbnail bot setup

Our first bot application will be a screenshot bot. We will provide the name of the website or URL we are interested in, then the bot will take a screenshot of the main page of that website and return a thumbnail image of it.

That's basically all the bot will do. Simple, but still fun—an interesting and time-saving tool.

In case you didn't save the Visual Studio project when we previously selected the Bot Application template, create a new VS project and select the template.

Let's name our bot **Web2ThumbnailBot** (you are free to use any name you wish, just be consistent and use the same name even when registering and publishing it).

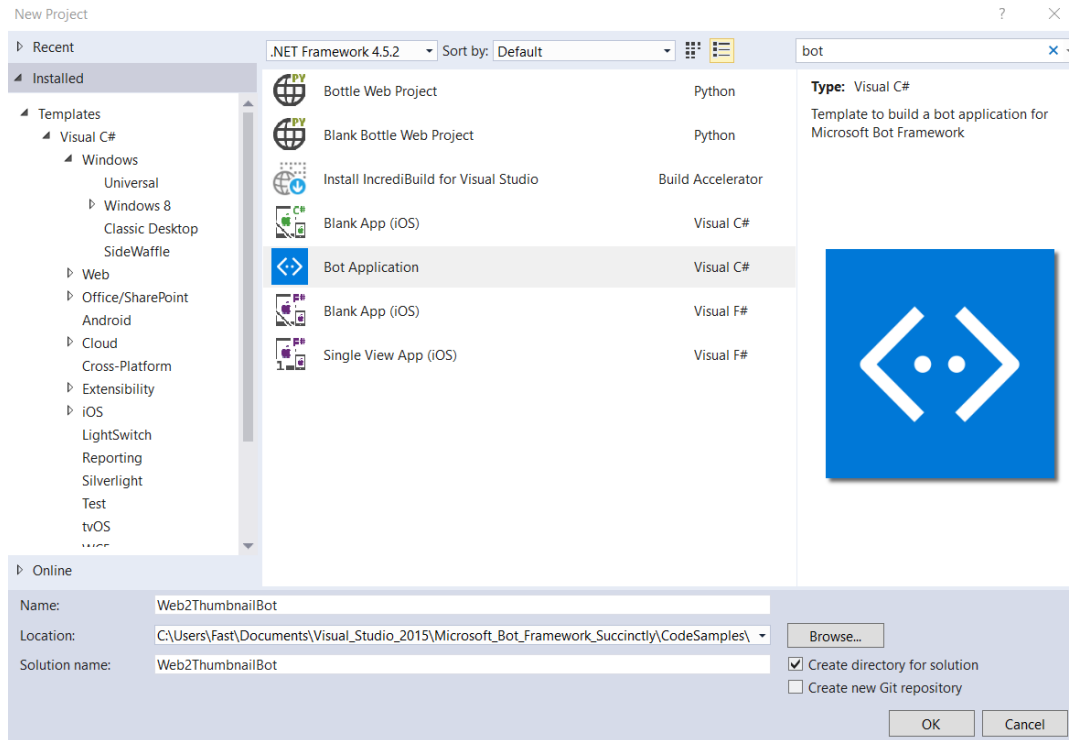


Figure 2.0: New VS Project with the Bot Application Template Selected

When the project has been created successfully, if we open the **Solution Explorer** within Visual Studio, we will immediately see that it is simply a Web API project.

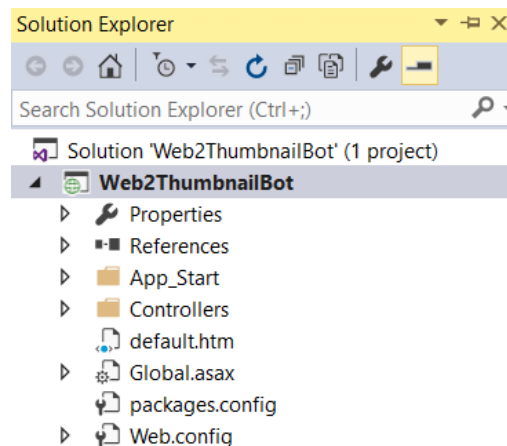


Figure 2.1: VS Project Structure

Before digging any further into the structure of the project, let's run the application in Debug mode by pressing **F5**.

Next, IIS Express is loaded and a browser window with a localhost site is launched.

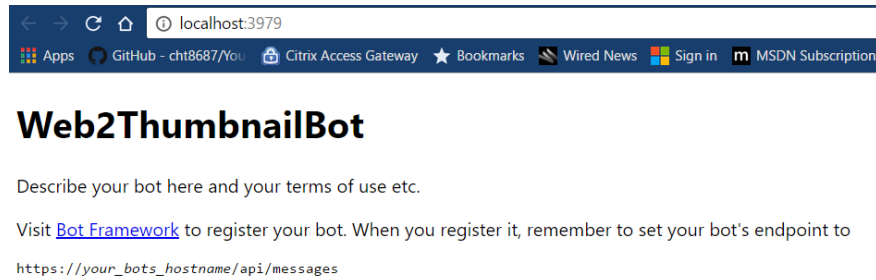


Figure 2.2: Bot Running on Localhost

Now that the bot is running for the first time, open the Bot Framework Channel Emulator and set the Bot URL to the same port where the bot site is running. In my case, that is `http://localhost:3979/api/messages`.

The port number might be different on your machine, but the rest of the URL (aside from the port number) needs to be the same.

Notice that the URL suffix `/api/messages` is required for the Bot Emulator to communicate with the bot app.

You can type any text. What we get by default is an echo response that replies with the string length that our message contains.

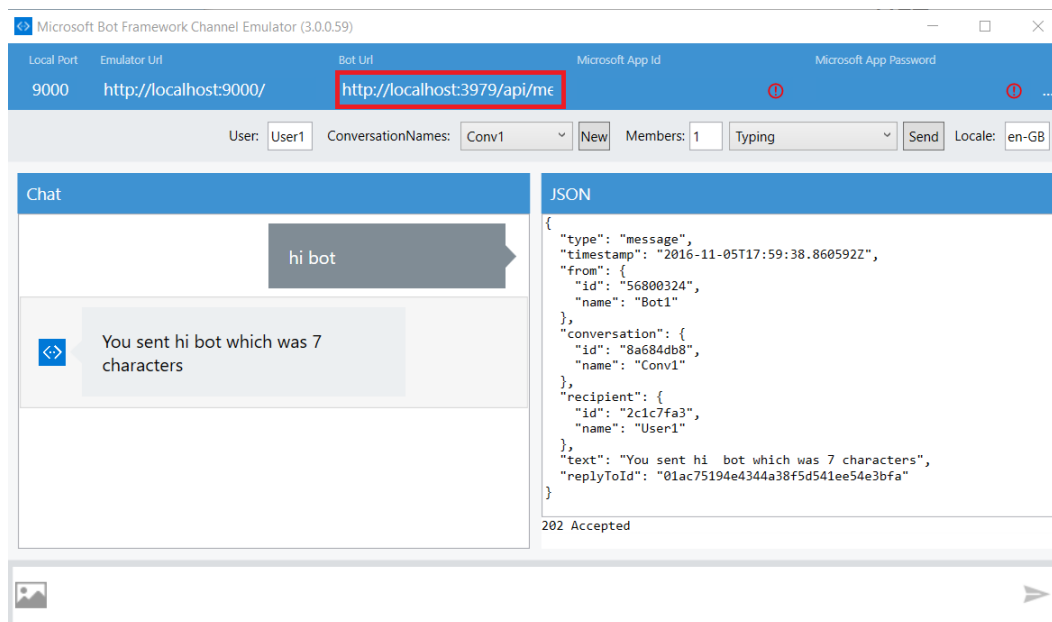


Figure 2.3: Echo Response on Bot Emulator

Very straightforward so far, but now it's time to dive a bit deeper and explore what's happening behind the scenes.

The Bot Framework Emulator is under active development and the UI is likely to be somewhat different from the one shown in this e-book. However, you shouldn't have much trouble with any UI changes.

Let's go back into the **Solution Explorer** in Visual Studio and expand the **Controllers** folder. Inside this folder is a file called `MessagesController.cs` that defines all the methods and properties that control our bot.

Now, let's examine the code. I've only slightly modified the output message (to make the response shorter) after running the prebuilt code the first time—you'll see a comment in the following code.

Code Listing 2.0: MessagesController.cs

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;
using Microsoft.Bot.Connector;

namespace Web2ThumbnailBot
{
    [BotAuthentication]
    public class MessagesController : ApiController
    {
        /// <summary>
        /// POST: api/Messages
        /// Receive a message from a user and reply to it.
        /// </summary>
        public async Task<HttpResponseMessage> Post([FromBody]Activity
activity)
        {
            if (activity.Type == ActivityTypes.Message)
            {
                ConnectorClient connector = new ConnectorClient(new
Uri(activity.ServiceUrl));
                // Calculate something for us to return.
                int length = (activity.Text ?? string.Empty).Length;

                // Reply to the user - I made the response shorter :)
                Activity reply = activity.CreateReply(
                    $"{activity.Text} is {length} characters");
                await
connector.Conversations.ReplyToActivityAsync(reply);
            }
            else
            {
                HandleSystemMessage(activity);
            }
        }
    }
}
```



```

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

    private Activity HandleSystemMessage(Activity message)
    {
        if (message.Type == ActivityTypes.DeleteUserData)
        {
            // Implement user deletion here.
            // If we handle user deletion, return a real message.
        }
        else if (message.Type == ActivityTypes.ConversationUpdate)
        {
            // Handle conversation state changes, like members being
            // added and removed. Use Activity.MembersAdded and
            // Activity.MembersRemoved and Activity.Action for info.
            // Not available in all channels.
        }
        else if (message.Type == ActivityTypes.ContactRelationUpdate)
        {
            // Handle add/remove from contact lists.
            // Activity.From + Activity.Action represent what
            // happened.
        }
        else if (message.Type == ActivityTypes.Typing)
        {
            // Handle knowing that the user is typing.
        }
        else if (message.Type == ActivityTypes.Ping)
        {
        }
        return null;
    }
}

```

In this class, the **Post** method takes care of the request received from the user and sends a reply back with the length of the message received. This is possible because we are checking that the **Activity.Type** is of type **ActivityTypes.Message**.

Notice that it is also possible to handle other **ActivityTypes** that might occur while the bot is running. These are handled within the **HandleSystemMessage** method.

See also how the reply to the message received is handled with an asynchronous method called **ReplyToActivityAsync**. In essence, all the bot's communication is done asynchronously—this is important to keep in mind.

Finally, if we take a step back and analyze the sample code automatically generated by the Bot Application template, we can see that we have the most important foundation for our bot, which is basic communication handling.

Now we simply need to create the necessary logic for generating website thumbnails.

We can write this logic inside a separate .cs file, then invoke a method from the new class we will create from the **Post** method.

Let's name our new project file **Thumbnails.cs** and get rolling. We can create this file as a standard C# class within the **Controllers** folder of our VS project.

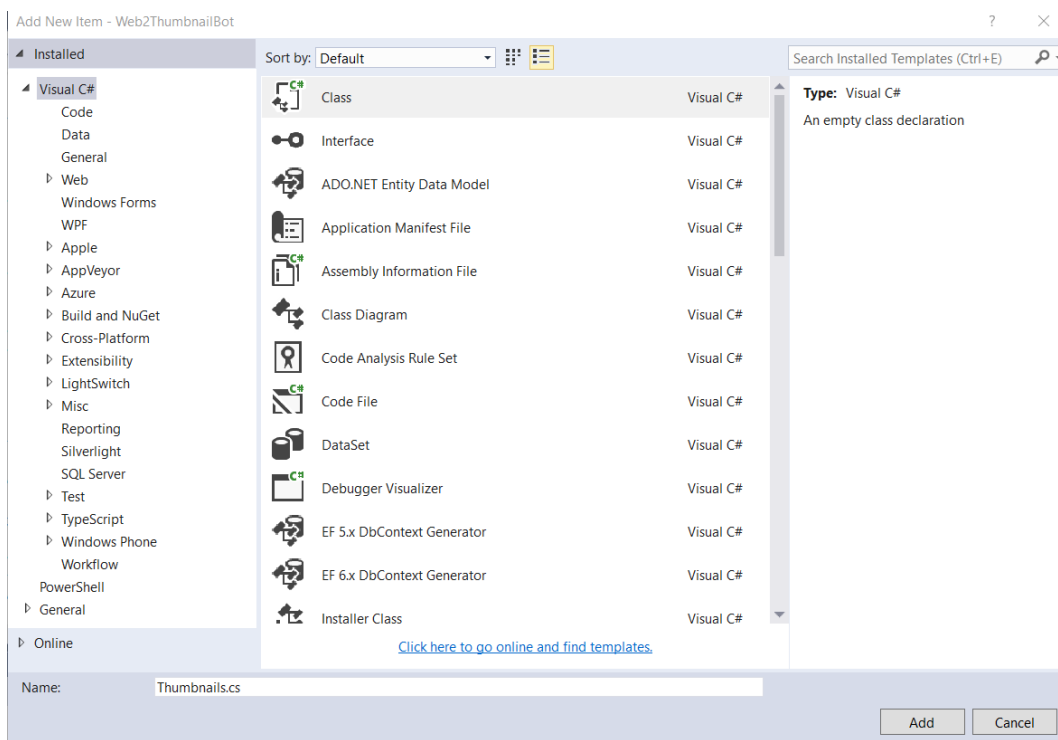


Figure 2.4: Adding the Thumbnails.cs Class

To generate a site's thumbnail, we can use an existing API or write our own code. If something has already been created, let's not reinvent wheel and instead use what is out there.

There are several libraries and APIs that allow us to take screenshots of a website such as [Selenium WebDriver](#) and [PhantomJS](#). However, we won't be using either of them, as their implementations can be quite tricky. Instead, we'll be using a simple yet effective and reliable external API called [Webthumbnail.org](#).

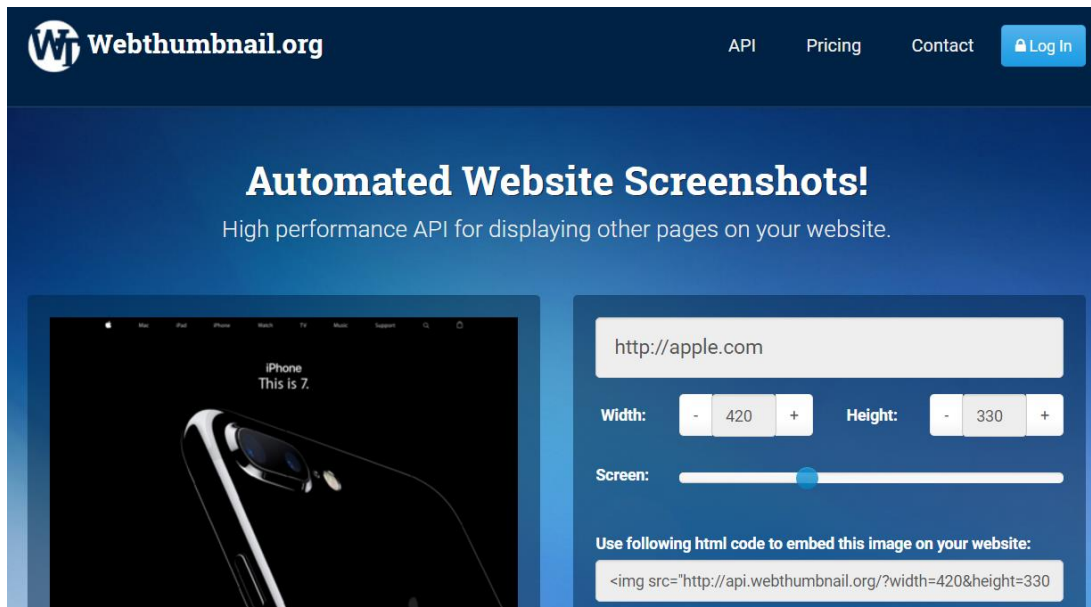


Figure 2.5: Webthumbnail.org

The site's API is incredibly simple and reliable. All we need to use this API is make an HTTP request to this URL:

`http://api.webthumbnail.org/?width=...&height=...&screen=...&url=...`

Simply replace the `...` with integer values no greater than 1024 for *width* / *height* and 1920 for the *screen* parameter. The *url* parameter works with or without the *http* or *https* prefixes.

The API details can be found [here](#).

How to use the API

You can easily use Webthumbnail.org by hotlinking images directly in your website's html code. To do that use the `` tag.

The API endpoint is available at <http://api.webthumbnail.org>.

The SSL is also supported, for secure http use <https://api.webthumbnail.org> endpoint.

This is an url format for hotlinking screenshot images generated by Webthumbnail.org:

`http://api.webthumbnail.org/?width=...&height=...&screen=...&url=...`

Param	Default	Min	Max	Description
width	100	100	1024	Thumbnail width in pixels.
height	100	100	1024	Thumbnail height in pixels.
screen	1024	1024	1920	Browser screen width, correct values are: 1024, 1280, 1650, 1920.
url	-	-	-	The url address of the website you want to capture. Url works with and without http:// and https:// prefixes.

Figure 2.6: How to Use the Webthumbnail.org API

I've never seen such a simple and effective API. Now let's write some code that uses it.

Using the Webthumbnail.org API

In order to use this API, we'll need to install the RestSharp library, which is available as a NuGet package.

We can do this by right-clicking the **References** item in the **Solution Explorer** of our Visual Studio project and then selecting the option **Manage NuGet Packages**.

Once we've done this, we select **RestSharp** from the list in the **Visual Studio NuGet Package** window, type **rest** in the search bar, then click **Install** to install the package. This will bring the required references and any dependencies into the project.

Once installed, you can always update this library to the newest version by clicking **Update**.

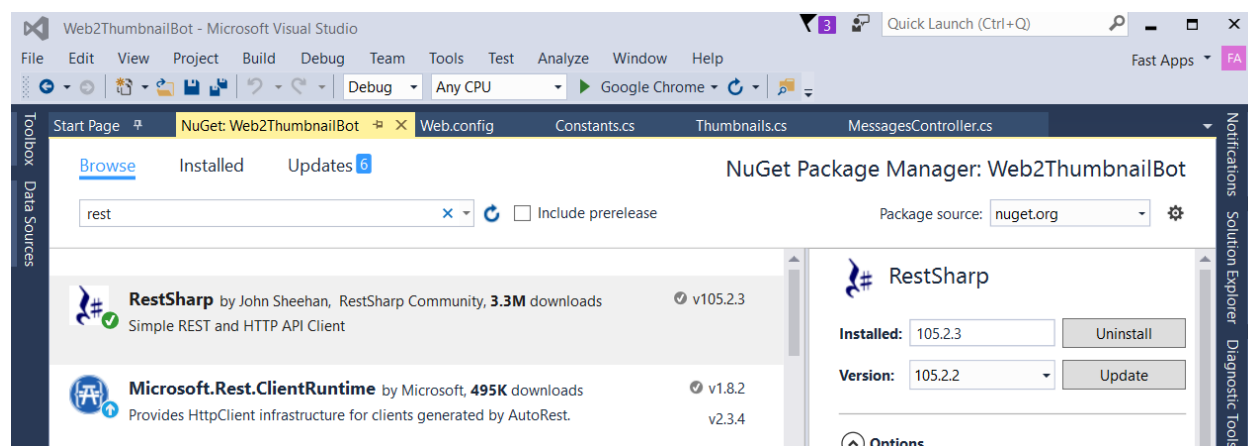


Figure 2.7: The RestSharp NuGet Package

Now that RestSharp has been installed, we are ready to start implementing the logic that our bot will use to capture screenshots using the Webthumbnail.org API. We'll do this next.

The bot's capture logic

One of the reasons this API was chosen (besides its simplicity and reliability) is that the Bot Framework doesn't give us the ability to keep files on disk—it is basically a Web API application.

So, we invoke the Webthumbnail.org API and get back a URL that points to a thumbnail image of the site we requested. This is perfect because we can then return the URL back to the user as a response card.

Let's write this logic in our Thumbnails.cs file to keep all the logic related to the creation of the thumbnail and create the response card in one place—separate from the main bot logic.

Code Listing 2.1 shows how the code will look.

Code Listing 2.1: Thumbnails.cs

```
using Microsoft.Bot.Connector;
using System.Collections.Generic;
using System.Threading.Tasks;
using RestSharp;

namespace Web2ThumbnailBot
{
    public class Thumbnails
    {
        public static async Task ProcessScreenshot(ConnectorClient
            connector, Activity msg)
        {
            Activity reply = msg.CreateReply($"Processing: {msg.Text}");
            await connector.Conversations.ReplyToActivityAsync(reply);

            await Task.Run(async () =>
            {
                string imgUrl = GetThumbnail(msg.Text);
                reply = CreateResponseCard(msg, imgUrl);

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

        public static string GetThumbnail(string url)
        {
            string r = Constants.cStrApiParms + url;

            RestClient rc = new RestClient(Constants.cStrThumbApi);
            RestRequest rq = new RestRequest(r, Method.GET);

            IRestResponse response = rc.Execute(rq);

            return Constants.cStrThumbApi + r;
        }

        public static Activity CreateResponseCard(Activity msg, string
            imgUrl)
        {
            Activity reply = msg.CreateReply(imgUrl);

            reply.Recipient = msg.From;
            reply.Type = "message";
            reply.Attachments = new List<Attachment>();

            List<CardImage> cardImages = new List<CardImage>();
        }
    }
}
```

```

        cardImages.Add(new CardImage(imgUrl));

        ThumbnailCard plCard = new ThumbnailCard()
        {
            Subtitle = msg.Text,
            Images = cardImages
        };

        Attachment plAttachment = plCard.ToAttachment();
        reply.Attachments.Add(plAttachment);

        return reply;
    }
}

```

There are some string constants used in the previous code that are kept in a Constants.cs file.

Code Listing 2.2: Constants.cs

```

namespace Web2ThumbnailBot
{
    public class Constants
    {
        public const string cStrHttp = "http://";
        public const string cStrHttps = "https://";
        public const string cStrThumbApi =
            "http://api.webthumbnail.org/?";
        public const string cStrApiParms =
            "width=1024&height=1024&screen=1024&url=";
    }
}

```

We will be using a value of 1024 for the *width*, *height*, and *screen* parameters of the API. These have been hard-coded into the **cStrApiParms** string.

The **Thumbnails** class is responsible for making the call to the Webthumbnail.org API, then returning the URL of the thumbnail created as a reply to the user.

The main method of the class is called **ProcessScreenshot**, and it does two things: it invokes the **GetThumbnail** method, then passes the **imgUrl** variable to the **CreateResponseCard** method, creating a response card that contains the URL of the thumbnail taken and also a small image of the screenshot.

The **CreateResponseCard** method then returns an **Activity** object that is passed on to the **ReplyToActivityAsync** method in order to send the response back to the user.

A [response card](#) is simply a message displayed as a card that contains both text and an image. Now let's quickly analyze the **CreateResponseCard** method.

Code Listing 2.3: The CreateResponseCard Method

```
public static Activity CreateResponseCard(Activity msg, string imgUrl)
{
    Activity reply = msg.CreateReply(imgUrl);
    reply.Recipient = msg.From;
    reply.Type = "message";
    reply.Attachments = new List<Attachment>();

    List<CardImage> cardImages = new List<CardImage>();
    cardImages.Add(new CardImage(imgUrl));

    ThumbnailCard plCard = new ThumbnailCard()
    {
        Subtitle = msg.Text,
        Images = cardImages
    };

    Attachment plAttachment = plCard.ToAttachment();
    reply.Attachments.Add(plAttachment);
    return reply;
}
```

We can see that an **Activity** instance is created to be used as the **reply** that will be sent back to the user. This **reply** object has an **Attachments** property to which a **ThumbnailCard** object is added.

The **ThumbnailCard** object contains a list of **CardImage** objects. Multiple **CardImage** instances can be attached to a **ThumbnailCard** object, but in this case we'll attach only one **cardImage**.

The **CreateResponseCard** method is responsible for creating a response that contains a small image of the captured screenshot along with its URL, which will be sent back to the user.

Now, let's have a quick look at the **ProcessScreenshot** method.

This method first sends a reply back to the user saying that the request is being processed. This is done by invoking the first **ReplyToActivityAsync** method.

Code Listing 2.4: The ProcessScreenshot Method

```
public static async Task ProcessScreenshot(ConnectorClient
    connector, Activity msg)
{
    Activity reply = msg.CreateReply($"Processing: {msg.Text}");
    await connector.Conversations.ReplyToActivityAsync(reply);

    await Task.Run(async () =>
    {
        string imgUrl = GetThumbnail(msg.Text);
    });
}
```

```

        reply = CreateResponseCard(msg, imgUrl);

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

```

After that, we call the **GetThumbnail** method. Its result is passed on to the **CreateResponseCard** method, which returns the final response to the user—the second call to the **ReplyToActivityAsync** method.

We have wrapped this up into a **Task** because these operations are “glued” together, and the final response depends on the result of both the **GetThumbnail** and **CreateResponseCard** methods. They constitute one indivisible processing block that should be executed as a single task, but asynchronously, so that it doesn’t block the bot from processing other requests.

In order to wrap this up, we need to remove the original call to **activity.CreateReply** and replace it with **ProcessResponse** within the **Post** method of **MessagesController.cs**, as in Code Listing 2.5.

Code Listing 2.5: The Updated Post Method in MessagesController.cs

```

public async Task<HttpResponseMessage> Post([FromBody]Activity
    activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        ConnectorClient connector = new ConnectorClient(new
            Uri(activity.ServiceUrl));

        // Return our reply to the user.
        await ProcessResponse(connector, activity);
    }
    else
    {
        HandleSystemMessage(activity);
    }
    var response = Request.CreateResponse(HttpStatusCode.OK);
    return response;
}

```

Code Listing 2.6 shows what the **ProcessResponse** method looks like.

Code Listing 2.6: The ProcessResponse Method in MessagesController.cs

```

private async Task ProcessResponse(ConnectorClient connector,
    Activity input)
{

```



```

Activity reply = null;
string exMsg = string.Empty;

if (IsValidUri(input.Text, out exMsg))
{
    await Thumbnails.ProcessScreenshot(connector, input);
}
else
{
    reply = input.CreateReply(
        "Hi, what is the URL you want a thumbnail for?");

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

```

We can see that this method checks whether the incoming request is a valid URL, and, if so, it invokes the **static ProcessScreenshot** method from the **Thumbnails** class. Otherwise, it returns a default response.

Here's the full code for the updated MessagesController.cs file including the **IsValidUri** method.

Code Listing 2.7: The Updated Full MessagesController.cs File

```

using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;
using Microsoft.Bot.Connector;

namespace Web2ThumbnailBot
{
    [BotAuthentication]
    public class MessagesController : ApiController
    {
        /// <summary>
        /// POST: api/Messages
        /// Receive a message from a user and reply to it.
        /// </summary>
        public async Task<HttpResponseMessage> Post([FromBody]Activity
            activity)
        {
            if (activity.Type == ActivityTypes.Message)
            {
                ConnectorClient connector = new ConnectorClient(new
                    Uri(activity.ServiceUrl));
            }
        }
    }
}

```

```

        // Return our reply to the user.
        await ProcessResponse(connector, activity);
    }
    else
    {
        HandleSystemMessage(activity);
    }
    var response = Request.CreateResponse(HttpStatusCode.OK);
    return response;
}

public bool CheckUri(string uri, out string exMsg)
{
    exMsg = string.Empty;
    try
    {
        using (var client = new WebClient())
        {
            using (var stream = client.OpenRead(uri))
            {
                return true;
            }
        }
    }
    catch (Exception ex)
    {
        exMsg = ex.Message;
        return false;
    }
}

private bool IsValidUri(string uriName, out string exMsg)
{
    uriName = uriName.ToLower().Replace(
        Constants.cStrHttps, string.Empty);
    uriName = (uriName.ToLower().Contains(Constants.cStrHttp)) ?
        uriName : Constants.cStrHttp + uriName;

    return CheckUri(uriName, out exMsg);
}

private async Task ProcessResponse(ConnectorClient connector,
    Activity input)
{
    Activity reply = null;
    string exMsg = string.Empty;

    if (IsValidUri(input.Text, out exMsg))

```

```

        {
            await Thumbnails.ProcessScreenshot(connector, input);
        }
        else
        {
            reply = input.CreateReply(
                "Hi, what is the URL you want a thumbnail for?");

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

    private Activity HandleSystemMessage(Activity message)
    {
        if (message.Type == ActivityTypes.DeleteUserData)
        {
            // Implement user deletion here.
            // If we handle user deletion, return a real message.
        }
        else if (message.Type == ActivityTypes.ConversationUpdate)
        {
            // Handle conversation state changes, like members being
            // added and removed.
            // Use Activity.MembersAdded and Activity.MembersRemoved
            // and Activity.Action for info.
            // Not available in all channels.
        }
        else if (message.Type == ActivityTypes.ContactRelationUpdate)
        {
            // Handle add/remove from contact lists.
            // Activity.From + Activity.Action represent what
            // happened.
        }
        else if (message.Type == ActivityTypes.Typing)
        {
            // Handle knowing that the user is typing.
        }
        else if (message.Type == ActivityTypes.Ping)
        {
        }

        return null;
    }
}

```

Let's run the application with Visual Studio and use the Bot Emulator to interact with it. We will test with my own website, edfreitas.me, but you can use any other.

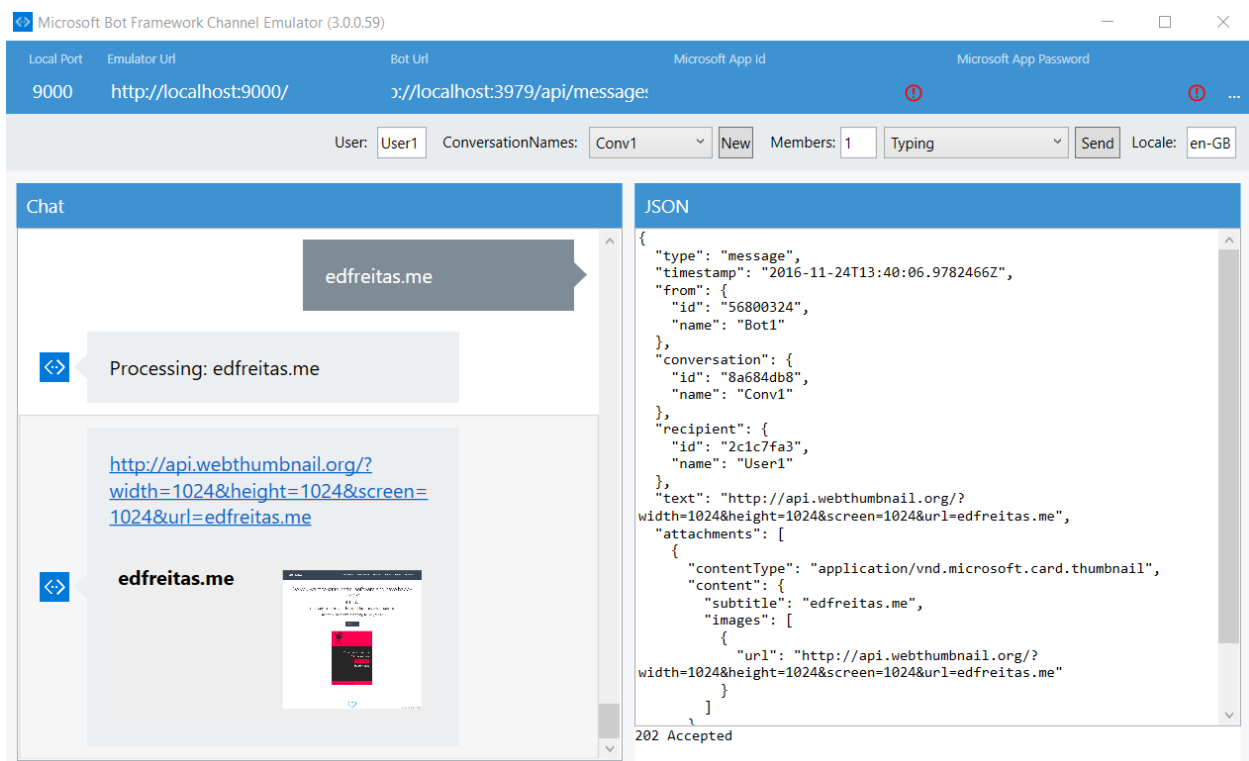


Figure 2.8: Bot Execution Results

We can see that we requested the bot to fetch a screenshot of edfreitas.me, and a **ThumbnailCard** object response was returned that contains the URL of the thumbnail taken and a small picture of the screenshot.

Figure 2.9 shows how the actual thumbnail taken looks in the browser if we click on the URL provided in the response.

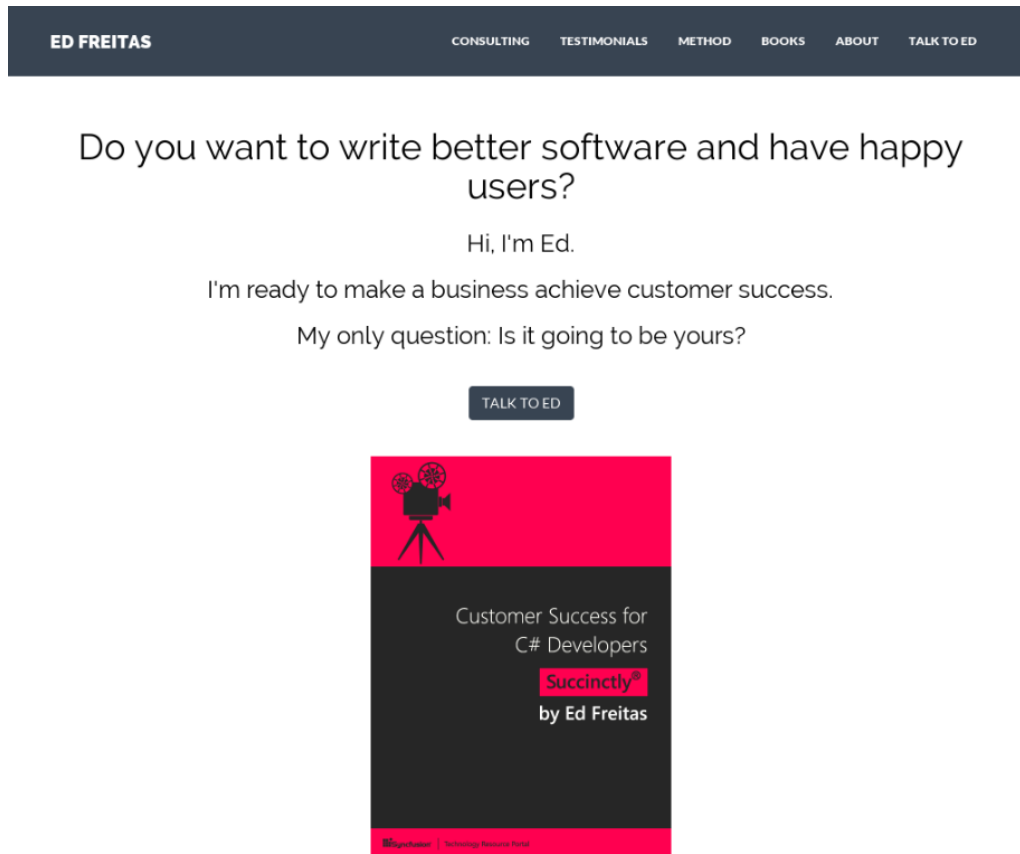


Figure 2.9: Thumbnail Taken by the Bot as Seen on the Browser

Awesome! Our bot application actually does what it was set to do, which is impressive considering that the code base is quite small, and it's also interesting because we were able to combine all this functionality into just a couple of classes. How cool is that?

By making the **ProcessScreenshot** method asynchronous, we pretty much guarantee that each request is independent from any other, and multiple requests can be made simultaneously without blocking the application to process other requests. That's great, and there's still room for further improvement. For example, we could add more asynchronous capabilities and threading or even explore other thumbnail libraries (or create one of our own). That's beyond the scope of this chapter, but it is definitely something worth exploring in your free time if you want to have a bit of extra fun.

You can find the full source code (VS project) for this bot [here](#).

Summary

Throughout this chapter, we've explored how to write a small, simple, yet interesting bot with a defined goal and practicality in mind.

It's not particularly the brightest and smartest of the bots out there, but it does perform an interesting task that could be quite tedious to perform manually. So, it's definitely worth our time and attention.

We've also looked at how to streamline the bot by making it capable of handling requests asynchronously, which makes it far more responsive than having it deal with one request at a time in a sequential and synchronous manner.

We've also noticed that when creating bots, we should keep the bot focused on being a bot. Anything that doesn't correspond to the bot itself might well be outsourced to an external service or API.

In other words, a bot should be a coordinator app and not a full-blown engine that does lots of things internally, as that would add complexity both in terms of tasks to manage and asynchronous execution.

In the next chapter, we'll look at how we can register and publish our bot to Skype.

Beyond that, the chapters that follow will focus on creating a bot that is conversationally fluid by using FormFlow.

We are starting to scratch the surface of bot development, and we can already start to appreciate its beauty and complexity. Most importantly, this should be a fun, immersive, engaging, and motivating experience.

Chapter 3 Publishing Our Bot

Introduction

In the previous chapter, we created a simple but useful bot that takes screenshots from websites. In short, we did something nice that solved a problem and automated a repetitive task.

In this chapter we'll explore each of the steps required for registering our application on the Bot Developer Portal, and then we'll publish it on Skype.

There won't be any coding involved at all, just steps related to the registration of the bot and publishing process, which are actually quite straightforward and easy to follow. Sounds awesome, so let's not wait any further. Rock and roll bot!

Registration basics

Every time we compile our bot code and run it, a browser window will appear that reminds us to register our bot—see [Figure 2.2](#).

Typically, I don't recommend registering your bot until you have tested it thoroughly and are confident it works on your local machine.

The idea of not publishing until we are ready is what inspired the Bot Emulator, which is able to connect to the bot's localhost endpoint and simulate that it's actually live and published. This is great for debugging and testing.

If the emulator did not exist, we would have to publish our bot every time we wanted to test something, which is definitely not ideal. Not all bot frameworks out there have an emulator, so the Microsoft Bot Framework is a huge time saver.

With our coding and debugging sessions behind us, we are now ready to register our bot. However, I highly recommend that you read this excellent [article](#) from the official Bot Framework documentation, which is helpful if you run into any authentication problems when testing your bot.

If you point your browser to the Bot Framework's main [page](#), there's an option clearly labeled **Register a bot**, so let's explore this further.

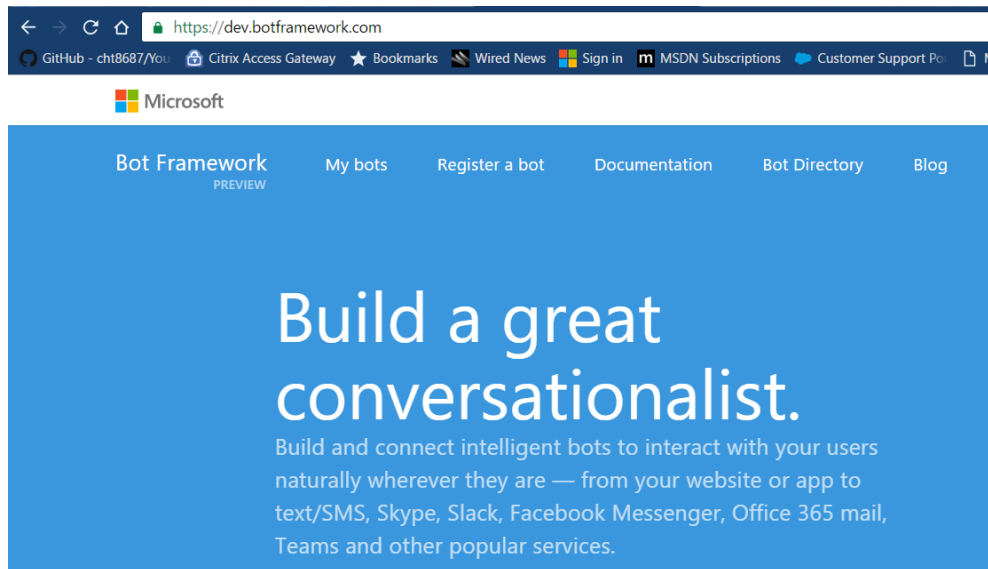


Figure 3.0: MS Bot Framework Main Page

Let's click the [Register a bot](#) option. When you click on it, you'll be prompted to sign in with a Microsoft (Hotmail, Live, or Outlook.com) account.

If you have not signed in to the Bot Framework before, you might be asked to subscribe to a regular newsletter. You might also be asked to select a country or region. Opting out is an option.

If you do opt-in, you'll receive a regular email newsletter with updates about the framework—definitely recommended if you want to stay current on any developments and news from the Microsoft team behind this awesome project.

When you've signed in, this is what the registration page looks like—very straightforward.

Figure 3.1: Bot Registration Page

As you can see, there are three basic fields that are mandatory. The first field represents the bot's name, which will be used to identify the bot within the Bot Directory (if we later decide to make it public). It cannot be longer than 35 characters.

The second field is the bot's handle, which will be used as part of the bot's public URL. It only allows alphanumeric and underscore characters, and it cannot be changed after registration.

The third field is the bot's description. The first 46 characters are displayed on the bot's card on the Bot Directory. The rest of the description is displayed under the bot's details.

Let's add some details to our bot.

Tell us about your bot

Bot profile

Icon
Upload custom icon
30K max, png only

Name: * ?
Web2ThumbnailBot

Bot handle: * ?
Web2ThumbnailBot

Description: * ?
A website thumbnail creation bot for the book Microsoft Bot Framework Succinctly from Syncfusion

Configuration

Messaging endpoint:
https://web2thumbnailbot.azurewebsites.net/api/messages

Figure 3.2: Registration Page with Our Bot's Info

Cool! So we have now entered the basics details, but we are not done yet registering our bot.

If we scroll down the page, we will see that we are also being asked to enter a messaging endpoint and an App ID, which we don't have yet because we haven't actually published the bot.

Let's leave this page open in our browser and switch over to Visual Studio to publish our bot and carry on with the registration process.

Publishing our Bot

If you don't have Visual Studio running, open it and load your bot project. Once the project is loaded, right-click on the project's name and select the **Publish** option from the context menu.

You will see the screen shown in Figure 3.3. Here we are being asked to select a publish target, which in our case will be Microsoft Azure App Service. Let's choose that one.

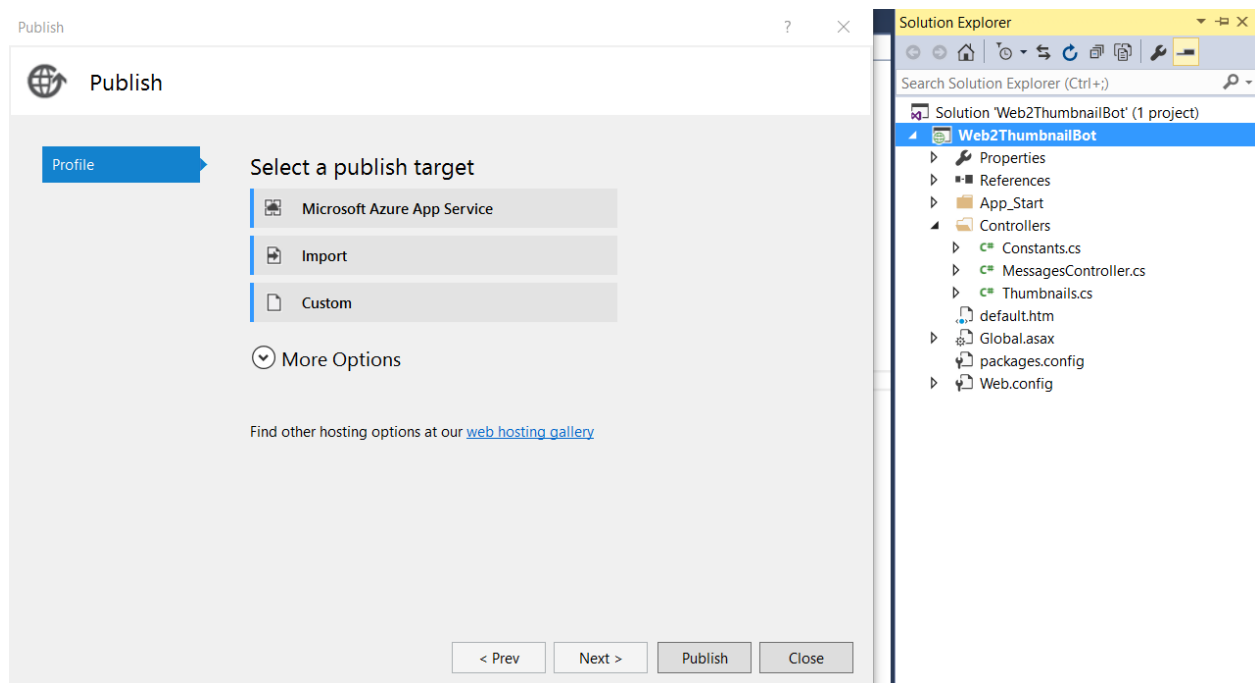


Figure 3.3: Initial Bot Publishing Screen within Visual Studio

When you've chosen that option, the publishing wizard will ask us to use an existing Azure App Service to host our bot or to create a new one. You will see the screen shown in Figure 3.4.

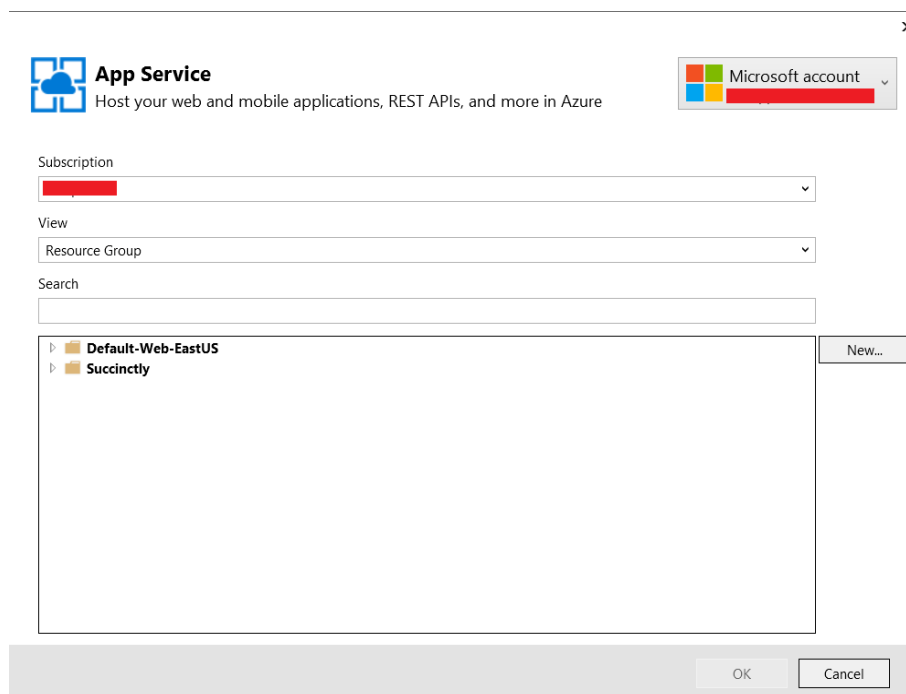


Figure 3.4: App Service Initial Screen within Visual Studio

I already have a couple of App Services created in Azure that I do not wish to use for the bot. I'd like the bot to have its own App Service, not shared with other applications I've created, so I'll choose to create a new one by clicking **New**, which leads to the screen in Figure 3.5.

Create App Service
Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account

Hosting
Services

Web App Name [Change Type](#)
Web2ThumbnailBot

Subscription
[Redacted]

Resource Group
Succinctly

App Service Plan
Succinctly [New...](#)

Clicking the Create button will create the following Azure resources

App Service - Web2ThumbnailBot

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... Create Cancel

Figure 3.5: App Service Creation Screen within Visual Studio

I've given my App Service the same name as the bot (also the same name as the VS project) in order to keep naming conventions simple and consistent. You may choose to do this differently; however, I strongly recommend simplicity and using the same name.

Even though I want to create a new App Service, I've decided to reuse some existing Azure resources that I've used for apps I've built previously, such as the Resource Group and Service Plan.

Finally, in order to create the Azure App Service that will host the bot, let's click **Create**.

Awesome! The App Service has been created, and the publishing process can continue. Next, we are presented with a final window that contains information about our bot and how it will be deployed to Azure.

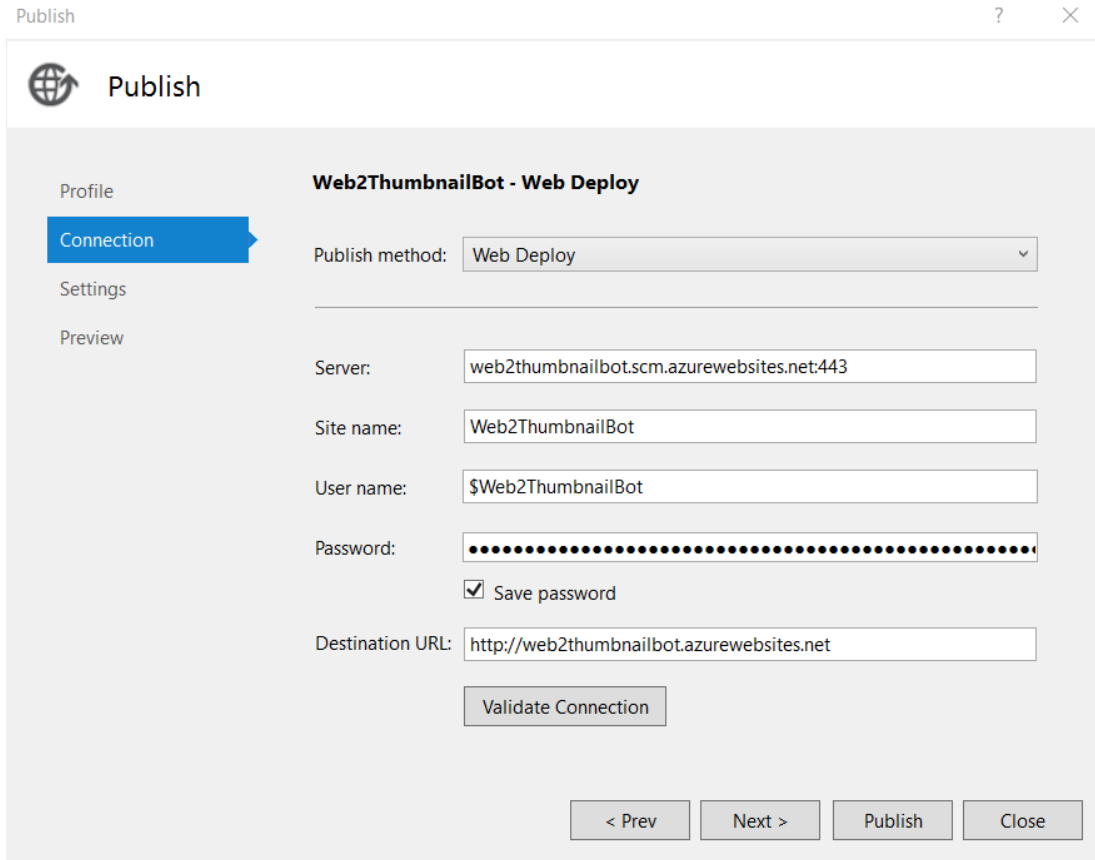


Figure 3.6: Web Deploy Publishing Screen within Visual Studio

The values are automatically filled in by Visual Studio with the details that the Azure App Service has provided, which means you may quickly test the connection to Azure by clicking **Validate Connection** before clicking **Publish**.

After clicking **Publish**, Visual Studio will start its magic. It will build the solution and deploy it to its corresponding App Service on Azure. This can take a couple of minutes. When that's done, you will see the **Output** pane shown in Figure 3.7.

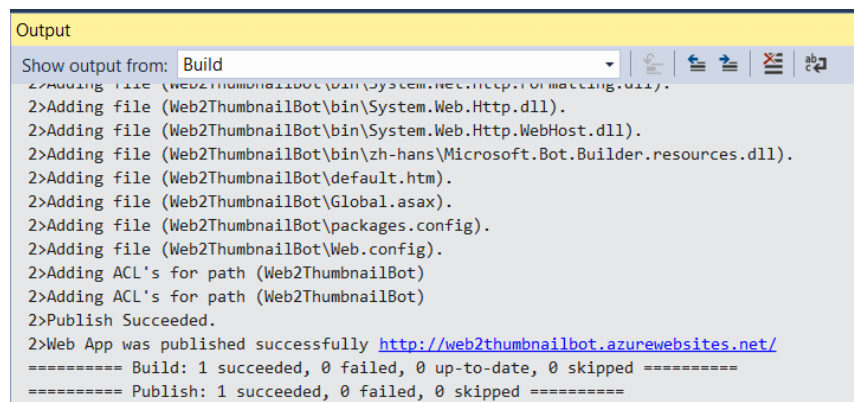


Figure 3.7: Output Pane in Visual Studio—Bot Published

At this stage, the bot has been published successfully and Visual Studio will open a browser window that looks like Figure 3.8.

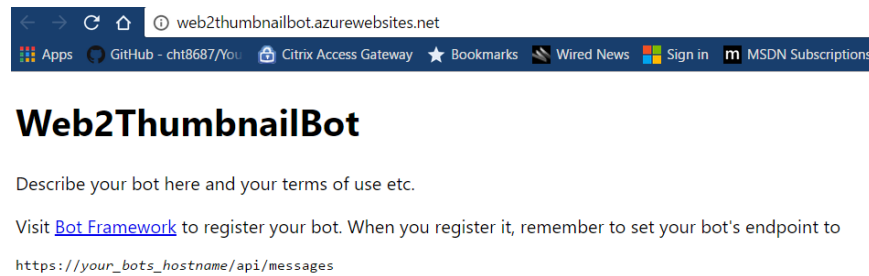


Figure 3.8: Bot Running on Azure

Notice that unlike when we ran the code in Debug mode, the browser window we have open now points to the URL where our bot resides on Azure.

With our bot published and running on Azure, we can continue the registration process.

Finishing the registration process

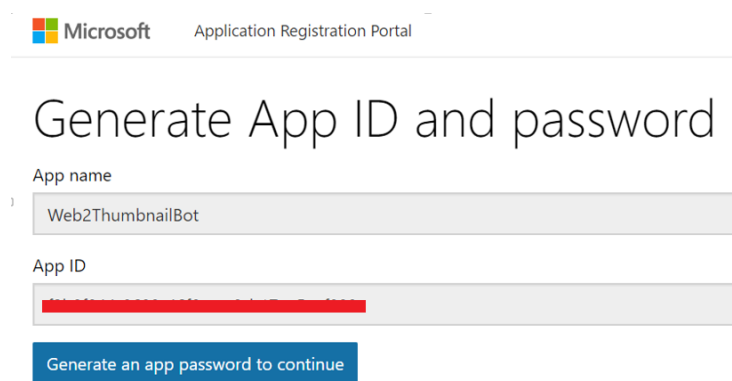
With our bot published, we can continue the registration process within the developer portal.

We now have the endpoint that was requested of us, but we still do not have an App ID. So, let's enter the endpoint first.

A screenshot of the Bot Framework registration configuration page. The page is divided into two main sections: 'Configuration' and 'Admin'. In the 'Configuration' section, there is a 'Messaging endpoint' field with the value 'https://web2thumbnailbot.azurewebsites.net/api/messages'. Below this is a button 'Create Microsoft App ID and password'. Underneath the button is a text input field for the 'Microsoft App ID from the Microsoft App registration portal', which is highlighted with a red border and labeled as a required field. The 'Admin' section below contains an 'Owners' field with a redacted value and an 'Instrumentation key' field with the placeholder text 'Instrumentation key (Azure App Insights key)'.

Figure 3.9: Additional Registration Details

Next, click **Create Microsoft App ID and password**. This will take us to the next screen.



Microsoft Application Registration Portal

Generate App ID and password

App name

Web2ThumbnailBot

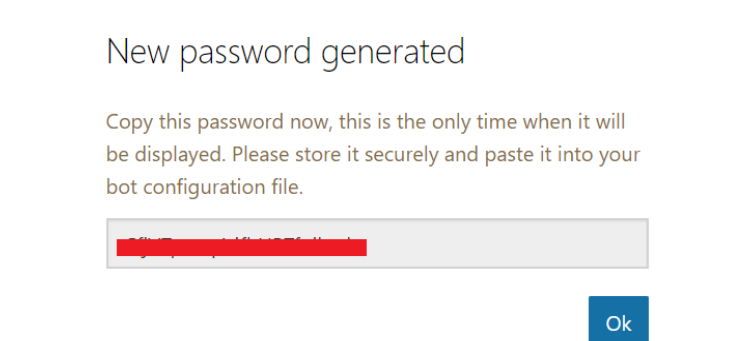
App ID

[Redacted]

[Generate an app password to continue](#)

Figure 3.10: The Generate App ID Screen

The ID is automatically generated. Next, we need to click **Generate an app password to continue**. This will create the application password we will need to copy and paste into the Web.config file in Visual Studio, along with the App ID. After that, we'll again publish our bot, but we'll do this later.



New password generated

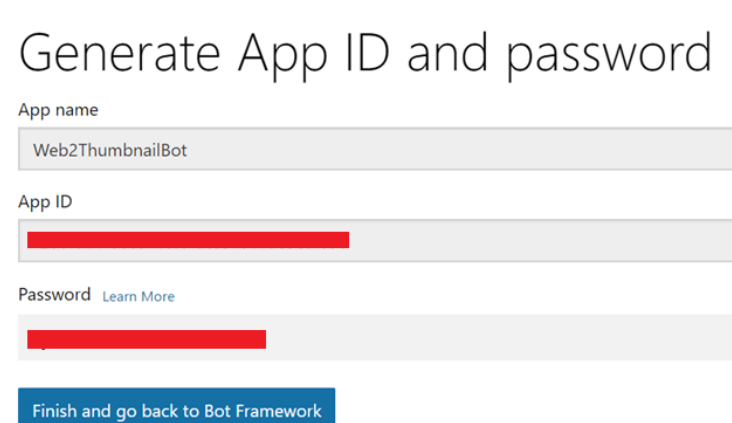
Copy this password now, this is the only time when it will be displayed. Please store it securely and paste it into your bot configuration file.

[Redacted Password]

[Ok](#)

Figure 3.11: The Generated Password Screen

When you get this pop-up window, copy and save the value of the generated password, then click **OK**. This will bring us back to the original screen.



Generate App ID and password

App name

Web2ThumbnailBot

App ID

[Redacted]

Password [Learn More](#)

[Redacted]

[Finish and go back to Bot Framework](#)

Figure 3.12: The Generated App ID and Password Screen

To finish this process, click **Finish and go back to Bot Framework**. This will bring you back to the main registration screen.

There, type your email in the **Owners** text box, accept the terms and conditions, then finish the process by clicking **Register**.

If you have an Azure App Insights key, you may enter this value as well, but it is not required.

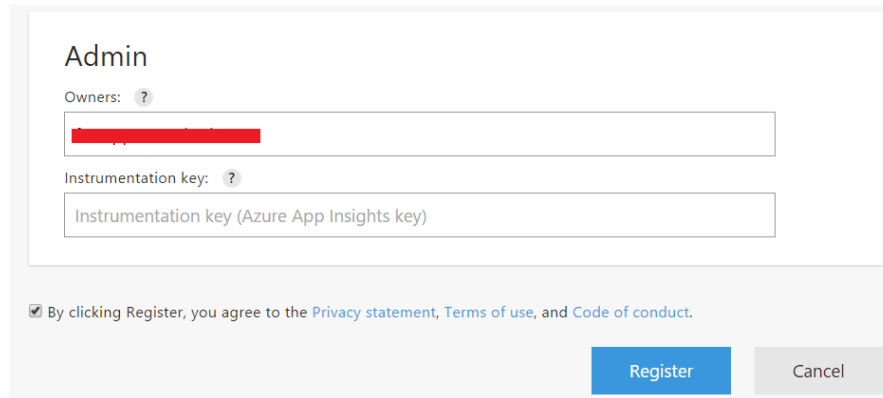


Figure 3.13: Final Registration Step

This will register the bot on the Bot Developer Portal. However, it will not publish it on the Bot Directory, nor will it enable channels such as Skype. This will lead to the screen in Figure 3.14.

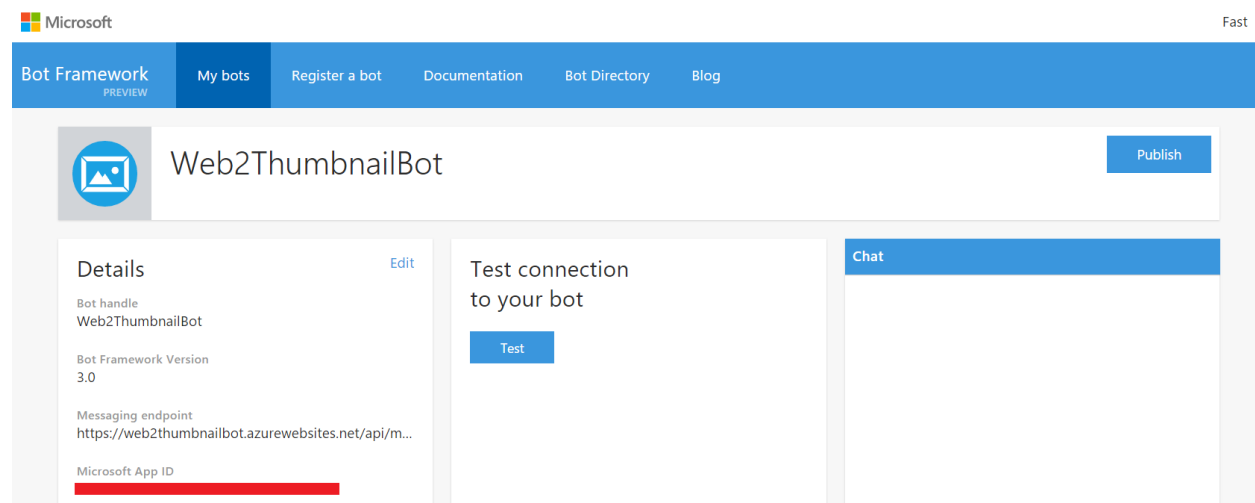


Figure 3.14: The Bot's Page on the Developer Portal

Exciting—our bot is now registered!

Re-publishing our bot

We now have the App ID and password from the Bot Developer Portal. We need to add this to our Visual Studio project in the Web.config file and re-publish.

If we don't re-publish, our bot won't be able to communicate with the Bot Service or use any channels we might add to it. So, let's go to Visual Studio, look for the Web.config file in our solution, and open it.

Code Listing 3.0: The AppSettings Part of the Web.config File

```
<appSettings>
  <!-- update these with your BotId, Microsoft App ID, and your Microsoft App
  Password-->
  <add key="BotId" value="" />
  <add key="MicrosoftAppId" value="" />
  <add key="MicrosoftAppPassword" value="" />
</appSettings>
```

Let's enter the values we have for the **BotId**, **MicrosoftAppId**, and **MicrosoftAppPassword**.

The **BotId** is the *bot handle*, the **MicrosoftAppId** is the *App ID*, and the **MicrosoftAppPassword** is the autogenerated *password* we previously copied.

If you used the same bot name on the developer portal as you did for the Visual Studio project and the Azure App service, the **BotId** will be the name of the Visual Studio project.

When you've entered the values, save the changes in Visual Studio and publish the bot again.

This time the publishing screen will look slightly different, and it's a one-click process because only a minimal change needs to be synced to the Azure App Service where our live bot is hosted.

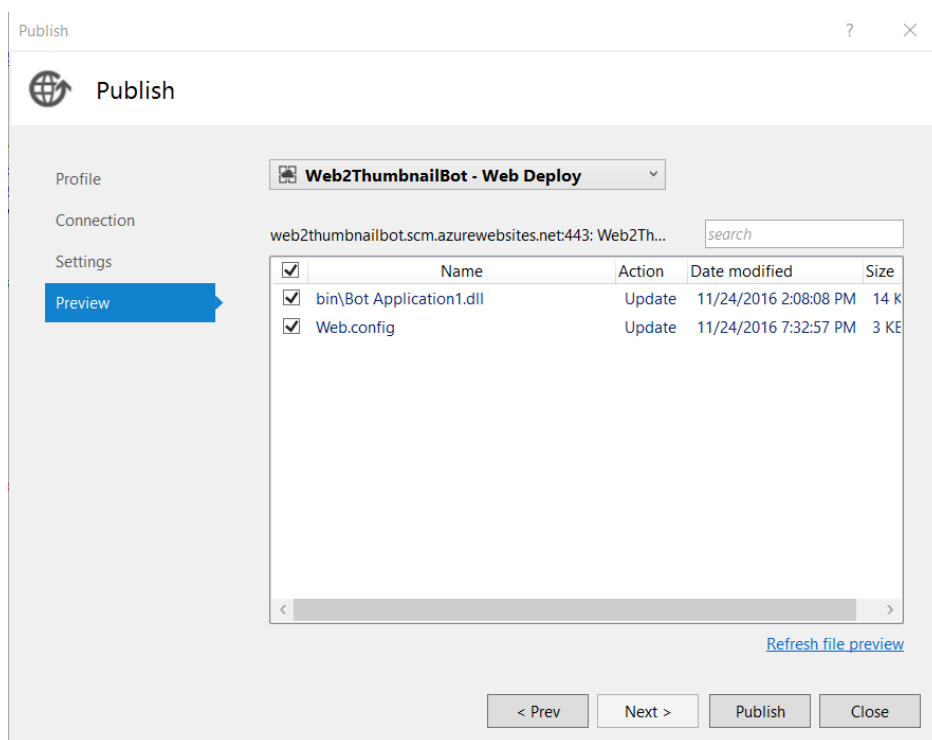


Figure 3.15: The Re-Publishing Process

The publish wizard knows which changes have taken place with our Visual Studio solution, and therefore it knows which changes it needs to commit to the Azure App Service host.

In this case, we changed only the Web.config file, so it will only commit the resulting DLL that is the actual bot itself (once compiled) and, obviously, the Web.config file.

Our bot is now live—hosted on Azure and capable of communicating with the Bot Service and any of the channels available (once these are enabled for our bot).

Let's go back to the developer portal page of our bot, run a small test, then add Skype as a channel.

Adding Skype as a channel

Now that our bot is live, you'll see a button for testing the connection to the bot on the developer portal. There's also a chat window just next to it where you can interact with the bot. This interaction is with the live bot hosted on Azure App Service. Let's try it out.

First, click **Test** and check that the authorization is successful. Next, send the bot a message in the **Chat** window on the right and see if it works.

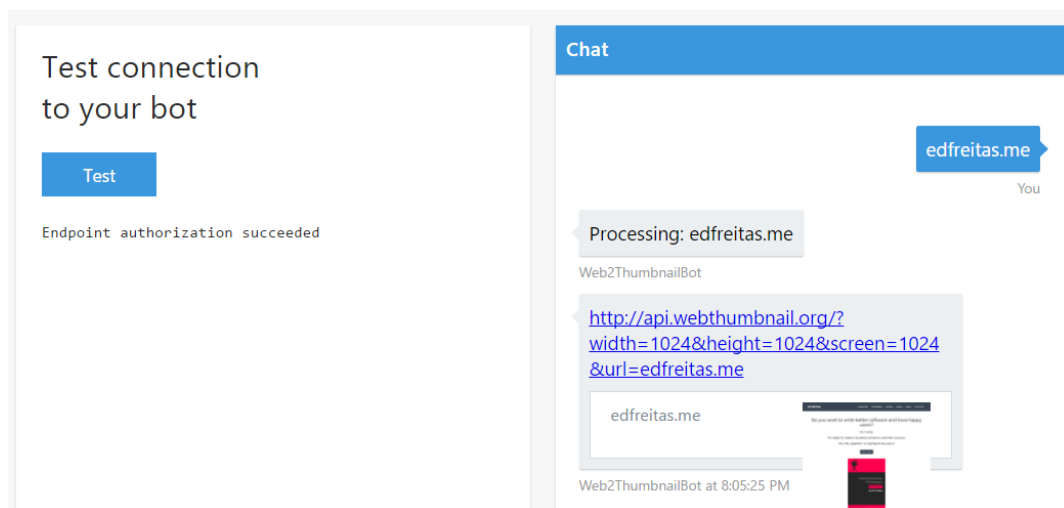




Figure 3.16: The Developer Portal Bot Testing Options

Awesome! It's all working as expected. The authorization succeeded, and the bot is able to return the requested screenshot and thumbnail of the site.

The two problems you're likely to run into are authentication and authorization. If for some reason the authentication process does not work, refer to this [documentation](#).

We can now add our bot to Skype, which is enabled as a default channel. To do this, scroll a bit down the same page and there you will find the Skype channel.

Channels

		Test link	Issues	Enabled	Published	
	Skype	Add to Skype	0	Yes (Preview)	<input type="checkbox"/> Off	Edit
	Web Chat		0	Yes	<input type="checkbox"/> Off	Edit

[Get bot embed codes](#)

Add another channel



	Direct Line	Add
	Email	Add

Figure 3.17: The Default Enabled Channels

A quick way to add the bot to Skype is by clicking **Add to Skype**. By doing this, you'll be able to add the bot to your contact list. Another way is to make it Published (by default this is set to **Off**), which means it will be publicly available in the Skype directory for all users.

For now, we simply want to interact with it through our contact list and send some requests. Let's add it the quick way, by clicking **Add to Skype**. When we do that, we get the screen shown in Figure 3.18.

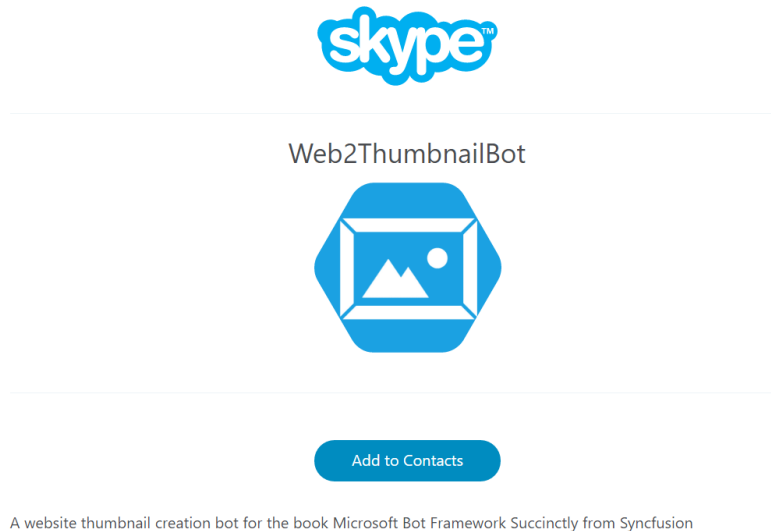


Figure 3.18: Adding the Bot to Our Skype Contacts

There, simply click **Add to Contacts**. The browser will then ask if you want to launch the Skype application. If you accept, the bot will be added to your contact list.

We've come a long way. We now have the bot on our Skype contact list, so now let's try it out for real.

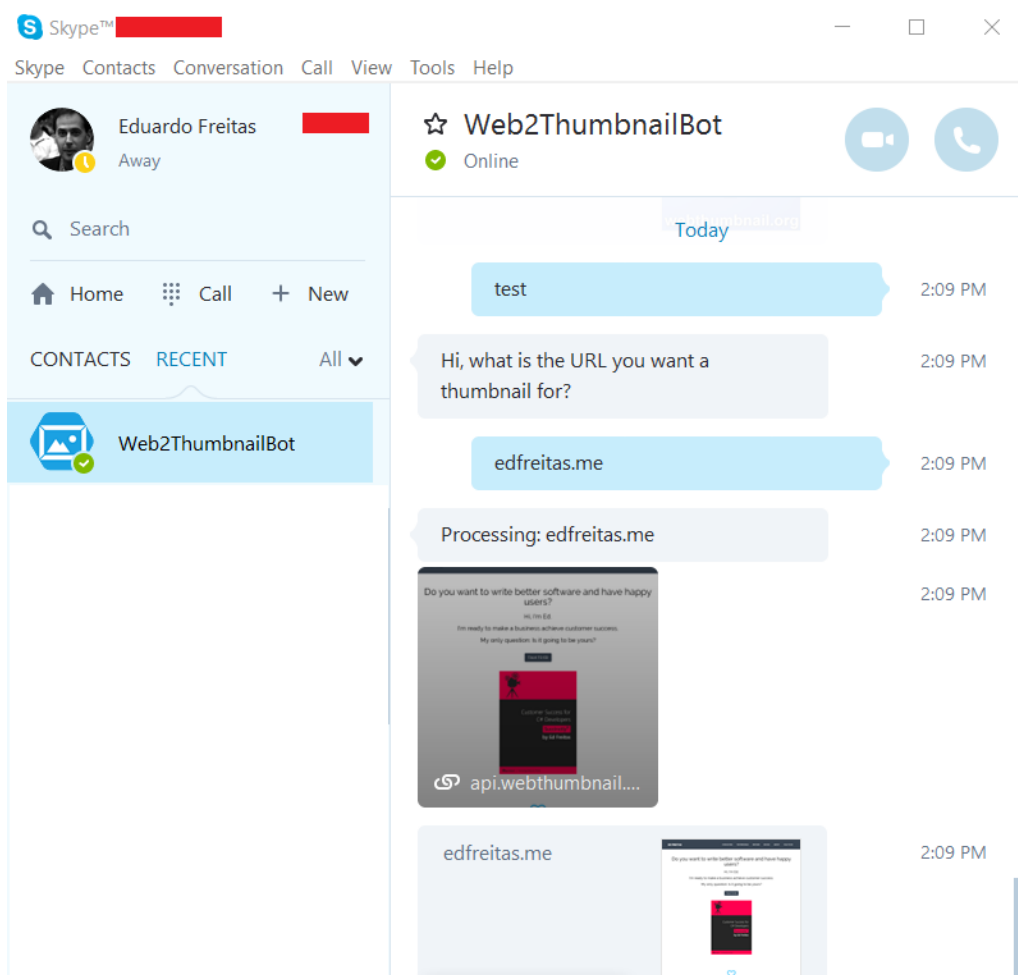


Figure 3.19: Interacting with Our Bot on Skype

It's worth noticing that any bot will always show up on Skype as active (green color).

Fantastic. We are able to interact with our bot, and it works like a charm. And because a bot is a bot, it will not go “away” or go into “don’t disturb” mode. That’s a relief!

Summary

We’ve accomplished our objective of writing, registering, and publishing a simple bot. In this chapter, we’ve gone from interacting with our bot in the Bot Emulator to registering it on the Bot Developer Portal, then publishing it on Azure and adding it to Skype.

We’ve explored in detail the steps required to make this happen, and although there were indeed various steps required, the process was logical and relatively straightforward.

These steps will be necessary when registering and publishing another bot, but we won't cover them again—it will be assumed that we've gone through this explanation and therefore we can focus more on the bot itself and other complementary APIs that will make our next bot more aware and smarter.

We'll make use of FormFlow and an interesting external API. The examples that follow should be interesting, challenging, and fun.

Chapter 4 The QPX Express API

Introduction

Following our awesome screenshot bot, I have a great idea. Why don't we write a bot that provides alerts when flight prices change? What do you think? Now, that's cool!

I don't know about you, but I always find nailing down a good price for my flights is tricky. When I book early, I like to think I'm getting the best deal, but that's not always the case. For example, I've had instances when I had to book just a few days before my trip, and the prices were actually better than when I looked a couple weeks earlier.

Airfares fluctuate based on offer and demand, just as with any other market. When there's a lot of demand for a route during a specific period, prices go up. And when there's not much demand, the airlines will push fares down to fill the planes. Remember, the airlines only make money when planes are up in the air with their seats full.

The bottom line is that there's rarely a time when you get on a half-empty flight and can take the whole row for yourself. The airlines manage to pack their planes as much as they can, especially in the cases of low-cost carriers such as Southwest or Ryanair.

The idea behind this next bot is that we do not want to go to a travel website like [Expedia](#) or [Skyscanner](#) to check for flight prices. Instead, we want our humble bot to do the work for us. Ultimately, the decision to act upon an airfare and book will of course be ours.

We should be able to provide the bot with the origin, destination, and travel dates, and we should get back the current price for the flights we are interested in, but the bot should be able to keep notifying us whenever the price changes for those flights (until we are no longer interested in receiving these updates).

Sounds like a lot of fun, and it will be an incredibly useful tool to have at our disposal.

In order to make this happen, we'll use Azure Table storage to keep airfare information and related flight details. We'll also use a specialized flight-tracking API, and we'll write our code with Visual Studio 2015 and C#.

I can't wait to get started using the bot that will let me know when flight prices change. Now that's putting computing power to work for us, so let's rock and roll.

The QPX Express API

In 2011, our good friends at [Google](#) acquired an amazing company that had been set up in the mid-1990s by MIT computer scientists to tackle the problem of accurately finding airfares without relying on expensive mainframe computers. This company is [ITA Software](#), which powers [Google Flights](#) and many other travel sites.

One of the results of their work is the state-of-art travel engine called QPX, which was first used by travel site [Orbitz](#).

The [QPX Express API](#) is a version of the QPX engine that simplifies flight and airfare searches. It is based on a self-service and pay-as-you-go model and makes it incredibly easy for any developer to get near real-time data about flights and prices.

This is the API our bot will use to gather flight information details and respective prices.

In order to use this API, we'll sign up with a Google account, which can be any [Gmail](#) or [G Suite](#) (previously known as Google Apps) account. If you don't have a Google account, please sign up for Gmail or G Suite.

Super! Let's get ourselves set up with the QPX Express API using our Google account.

Signing up for QPX Express

To use the QPX Express API, we'll need to create public key credentials on the Google Developers Console and register our project there.

The Google Developers Console is a site used by developers for managing and viewing traffic data, authentication, and billing information for the Google APIs that their projects use.

Within the Google Developers Console, you can find all the settings, credentials, and metadata about the applications you're working on that make use of Google APIs.

To get started, let's point our browser to this [URL](#). When you do that, and if you are signed in with your Google account, you'll see the welcome page shown in Figure 4.0.

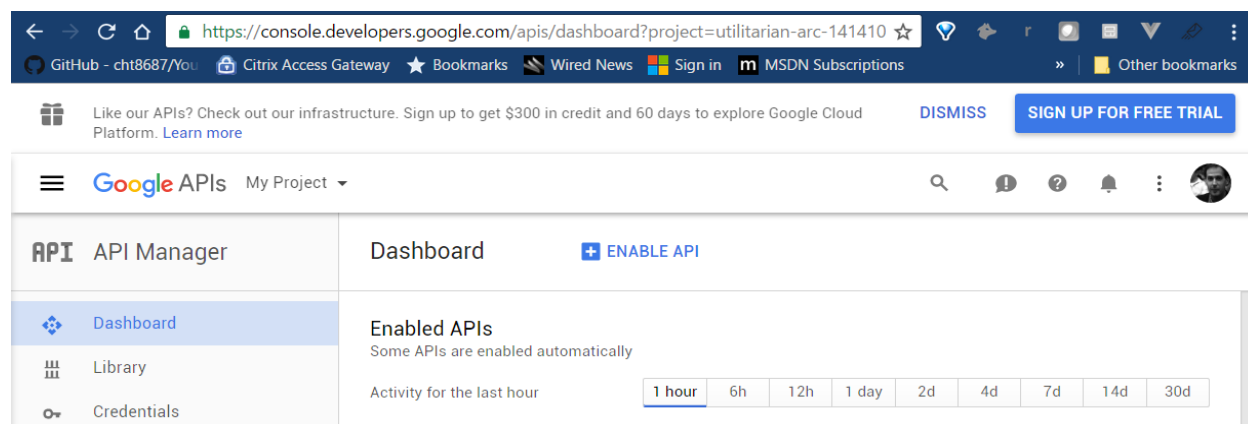


Figure 4.0: The Google Developers Console Welcome Page

You might find a **SIGN UP FOR FREE TRIAL** button at the top. You can sign up for the service, which will give you access to free credits for a certain period (normally two months), but you'll be asked to enter your credit card details—Google does this to verify that you are a real person, not a robot.

In this case, I won't sign up for a free trial—instead I'll explicitly search for the QPX Express API. I can do this by clicking **ENABLE API**. When you do this, you will see the screen in Figure 4.1.

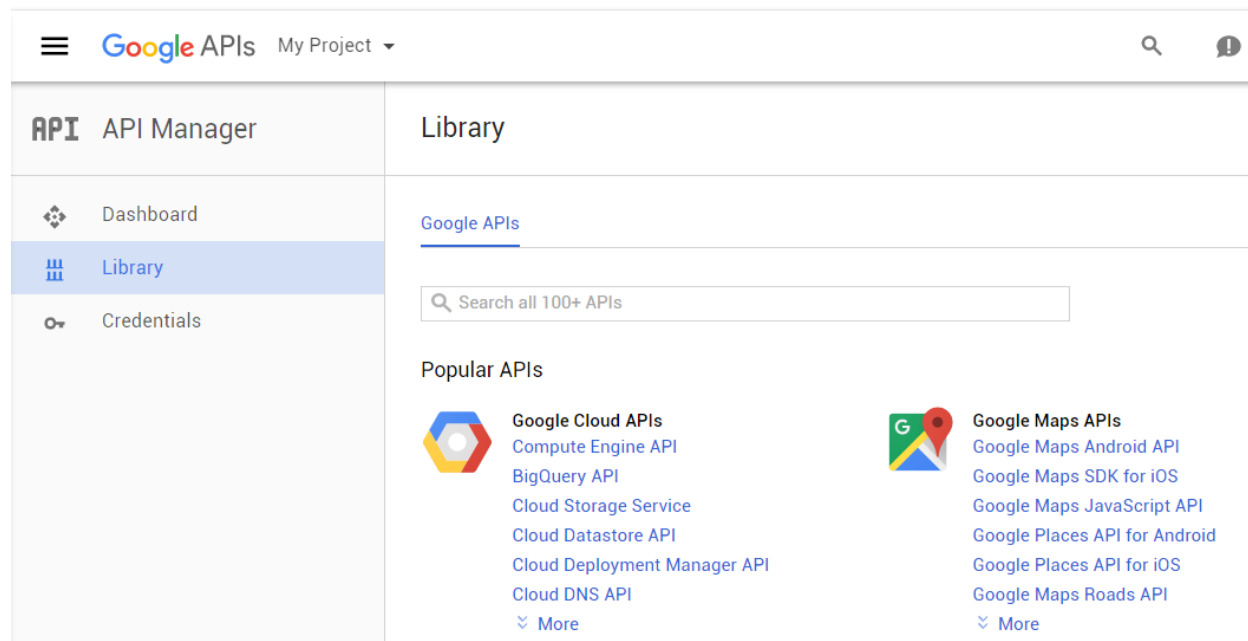


Figure 4.1: The Google Developers Console API Search

In the search bar, type the word **QPX**, and as you type, the API will appear on the list. Figure 4.2 shows how it will look.

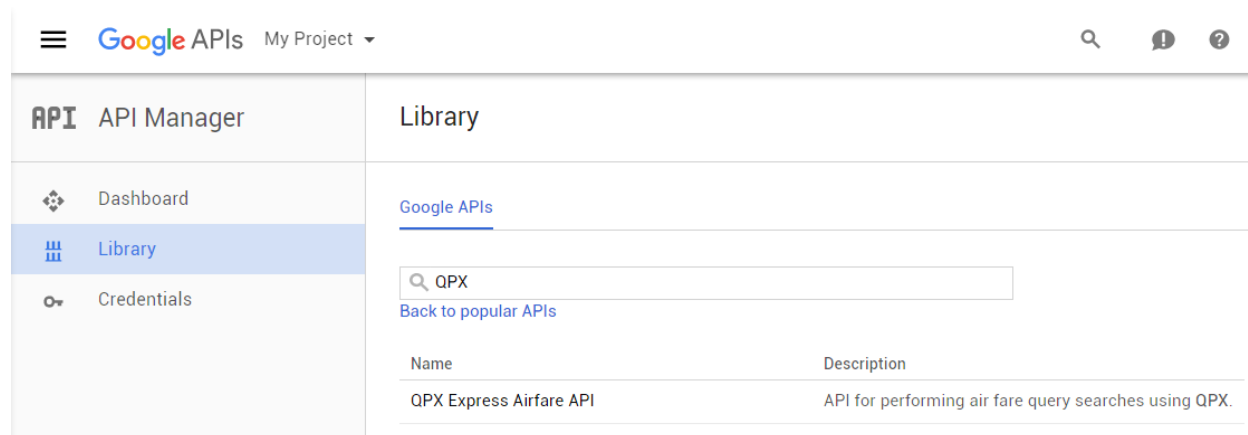


Figure 4.2: The QPX Express API in the Search Results

Let's click on the **QPX Express Airfare API** link (the result shown) to enable the API. After doing this, you'll see the screen in Figure 4.3.

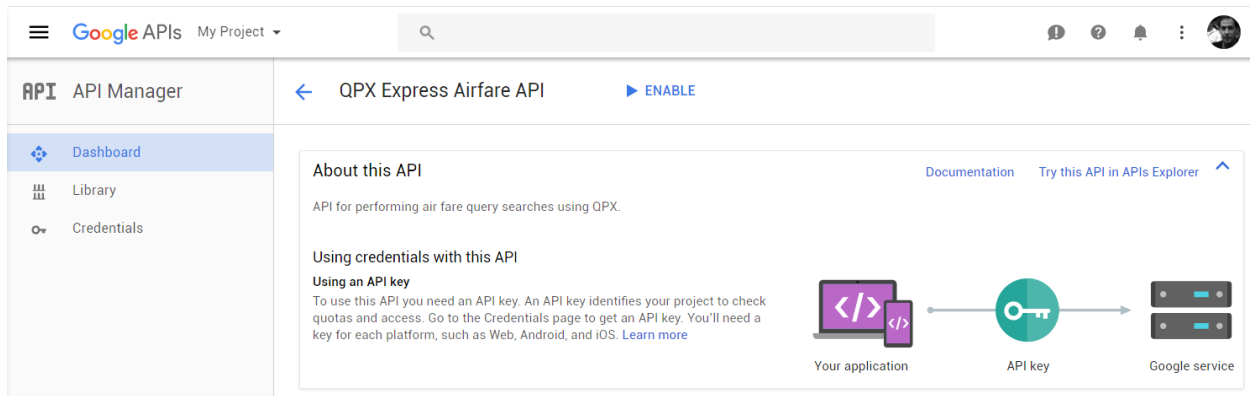


Figure 4.3: The QPX Express API Main Page in the Google Developer Console

Click **ENABLE** to enable the API. When we do this, we will see the screen in Figure 4.4.

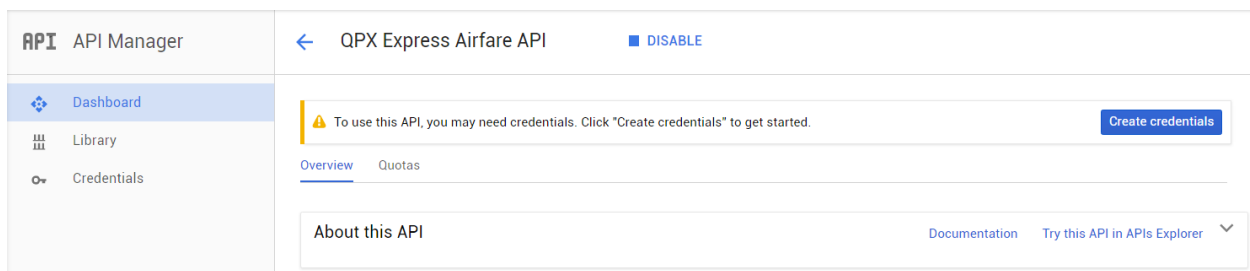


Figure 4.4: The QPX Express API Enabled

Awesome—we now have this API enabled! Next, we need to create our API credentials, so let's click **Create credentials**. When we do this, we'll see the screen in Figure 4.5.

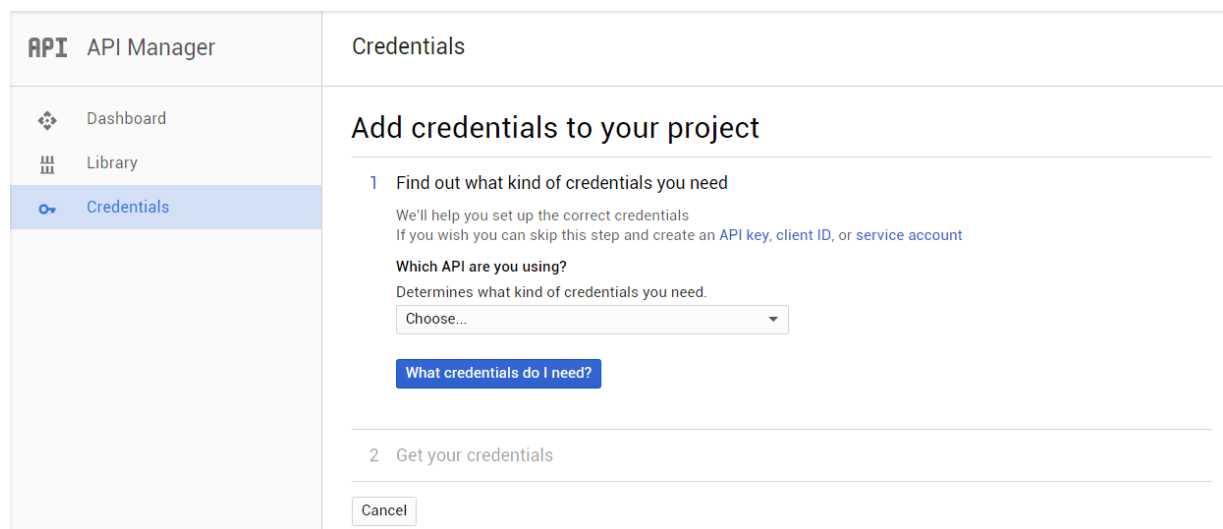


Figure 4.5: Adding QPX Express API Credentials

From the drop-down list, choose **QPX Express Airfare API**, then click **What credentials do I need?**

Which API are you using?
Determines what kind of credentials you need.

QPX Express Airfare API ▼

What credentials do I need?

Figure 4.6: Choosing the Credentials Needed

When you've clicked that button, you'll get the API key, which can be seen in Figure 4.7.

Add credentials to your project

✓ Find out what kind of credentials you need
Calling QPX Express Airfare API

2 Get your credentials

Here is your API key

[Redacted API key] [Copy]

⚠ We recommend restricting this key before using it in production.
Restrictions limit which web sites, IP addresses, or apps can call APIs with this key.
[Restrict key](#)

Done Cancel

Figure 4.7: The Credentials Automatically Created

Make sure you copy the API key, then click **Done**. We can then see our generated API key.

API keys

<input type="checkbox"/>	Name	Creation date ▼	Restriction	Key	
<input type="checkbox"/>	⚠ API key 1	Dec 2, 2016	None	[Redacted API key]	[Edit] [Delete]

Figure 4.8: The API Keys List

Later, you can edit the API key settings to limit how it can be accessed. This is not urgent right now.

Let's next create our VS project, then come back to the Google Developers Console and make any necessary adjustments.

Setting up our VS project

Now that we have covered the basics with the Google Developers Console, we can create our VS project. Let's use the same template and call our project **AirfareAlertBot**.

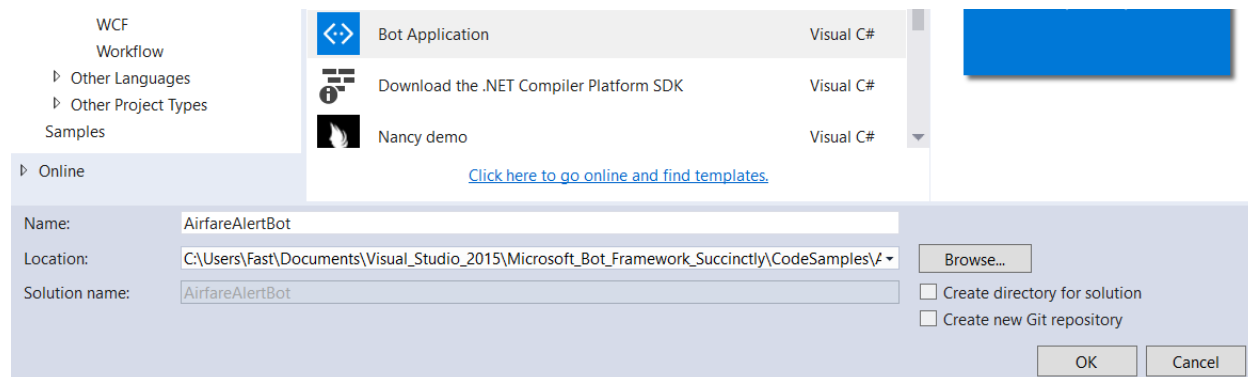


Figure 4.9: Selecting the Bot Template for Our VS Project

Click **OK** to create the project. When the project has been created, open the **Solution Explorer** within Visual Studio, select the **References** section within the project, then right-click and choose **Manage NuGet Packages**.

When you open the NuGet Package Manager, type the word **QPX** in the search bar to find the QPX Express API client library. This is what we'll use in our project. Let's do that now.

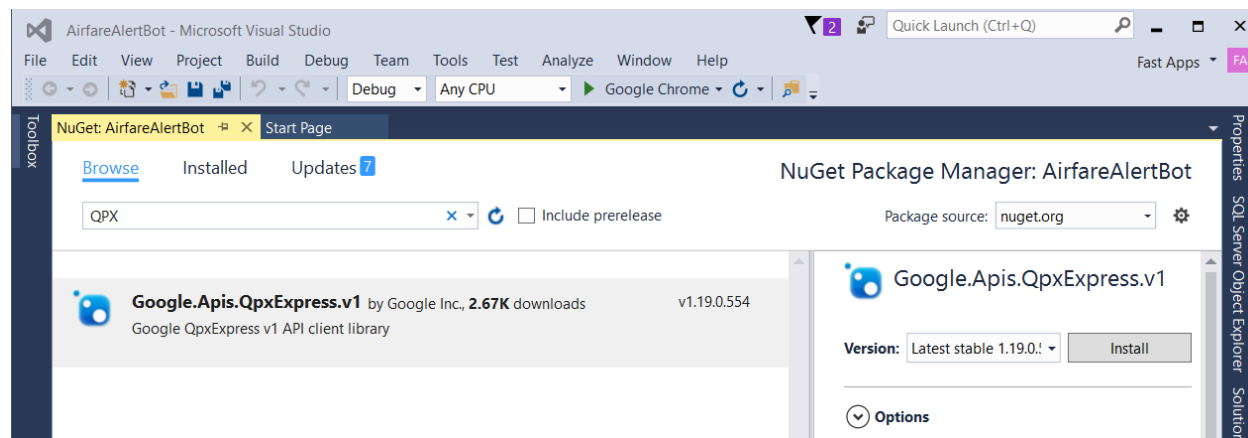


Figure 4.10: The QPX Express API Client Library NuGet Package

Just click **Install** to install this library. Doing so will bring all the required dependencies and references that this library uses into our project.

We can now get started with some code. As you already know, by default the Bot Application template creates the basic structure that the bot will use.

I'll add a new class file to our VS project inside the **Controllers** folder and call it **ExampleFlightData.cs**. Then I'll write some basic code that connects to the QPX Express API and is able to obtain the base fare price for a specific flight, which for now is hard-coded.

Code Listing 4.0: ExampleFlightData.cs

```
using Google.Apis.QPXExpress.v1;
using Google.Apis.QPXExpress.v1.Data;
using Google.Apis.Services;
using System.Linq;

using System.Collections.Generic;

namespace AirfareAlertBot.Controllers
{
    public class ExampleFlightData
    {
        private const string cStrApiKey =
            "Your QPX Express API Key";
        private const string cStrAppName = "AirfareAlertBot";

        public static string[] GetFlightPrice()
        {
            List<string> p = new List<string>();

            using (QPXExpressService service = new QPXExpressService(new
                BaseClientService.Initializer()
            {
                ApiKey = cStrApiKey,
                ApplicationName = cStrAppName
            })))
            {
                TripsSearchRequest x = new TripsSearchRequest();
                x.Request = new TripOptionsRequest();
                x.Request.Passengers = new PassengerCounts
                    { AdultCount = 2 };
                x.Request.Slice = new List<SliceInput>();

                var s = new SliceInput() { Origin = "JFK",
                    Destination = "BOS", Date = "2016-12-09" };

                x.Request.Slice.Add(s);
                x.Request.Solutions = 10;

                var result = service.Trips.Search(x).Execute();

                foreach (var trip in result.Trips.TripOption)
                    p.Add(trip.Pricing.
                        FirstOrDefault().BaseFareTotal.ToString());
            }

            return p.ToArray();
        }
    }
}
```

```
}  
}
```

Let's quickly analyze this code. We create a **QPXExpressService** instance by passing the QPX API key and the name of our bot application, as this allows us to authenticate our application with the QPX Express service.

When we are authenticated, we create a **TripsSearchRequest** instance to retrieve information about the flight we are interested in.

TripsSearchRequest contains a **Request** property to which a **TripOptionsRequest** instance is assigned.

As its name clearly states, **TripOptionsRequest** is an object that allows us to specify the trip details we are interested in fetching from the QPX Express API.

The **Passenger** property from the **TripOptionsRequest** object is assigned to an instance of the **PassengerCounts** class, which is used to indicate how many passengers will be travelling.

The **Slice** property of the **TripOptionsRequest** instance is assigned to a **List<SliceInput>** object. A slice corresponds to a flight. A **SliceInput** object is used to specify the origin, destination, and travel date.

With all of the details of the request defined, we can execute a query to the QPX Express API by invoking the **service.Trips.Search(x).Execute** method, which returns an **ICollection<TripOption>** object that we can then loop in order to retrieve the base fare for the trip.

Notice how concise, clean, and short the QPX Express API is. That's pretty much all we need to know about it, for now.

Make sure you replace the value of the **cStrApiKey** constant with your own QPX Express API key.

Now that we have a basic **GetFlightPrice** method created, we can invoke it from **MessagesController.cs**, as in Code Listing 4.1.

Code Listing 4.1: MessagesController.cs Invoking GetFlightPrice

```
using System;  
using System.Net;  
using System.Net.Http;  
using System.Threading.Tasks;  
using System.Web.Http;  
using Microsoft.Bot.Connector;  
using AirfareAlertBot.Controllers;  
  
namespace AirfareAlertBot  
{  
    [BotAuthentication]
```

```

public class MessagesController : ApiController
{
    public async Task<HttpResponseMessage> Post([FromBody]Activity
        activity)
    {
        if (activity.Type == ActivityTypes.Message)
        {
            ConnectorClient connector = new ConnectorClient(new
                Uri(activity.ServiceUrl));

            string[] p = ExampleFlightData.GetFlightPrice();

            Activity reply = activity.
                CreateReply($"Flight price {p?[0]}");

            await connector.Conversations.
                ReplyToActivityAsync(reply);
        }
        else
        {
            HandleSystemMessage(activity);
        }
        var response = Request.CreateResponse(HttpStatusCode.OK);
        return response;
    }

    private Activity HandleSystemMessage(Activity message)
    {
        if (message.Type == ActivityTypes.DeleteUserData)
        { }
        else if (message.Type == ActivityTypes.ConversationUpdate)
        { }
        else if (message.Type == ActivityTypes.ContactRelationUpdate)
        { }
        else if (message.Type == ActivityTypes.Typing)
        { }
        else if (message.Type == ActivityTypes.Ping)
        { }

        return null;
    }
}

```

If we now run this code with the Bot Emulator, we'll get the result shown in Figure 4.11.

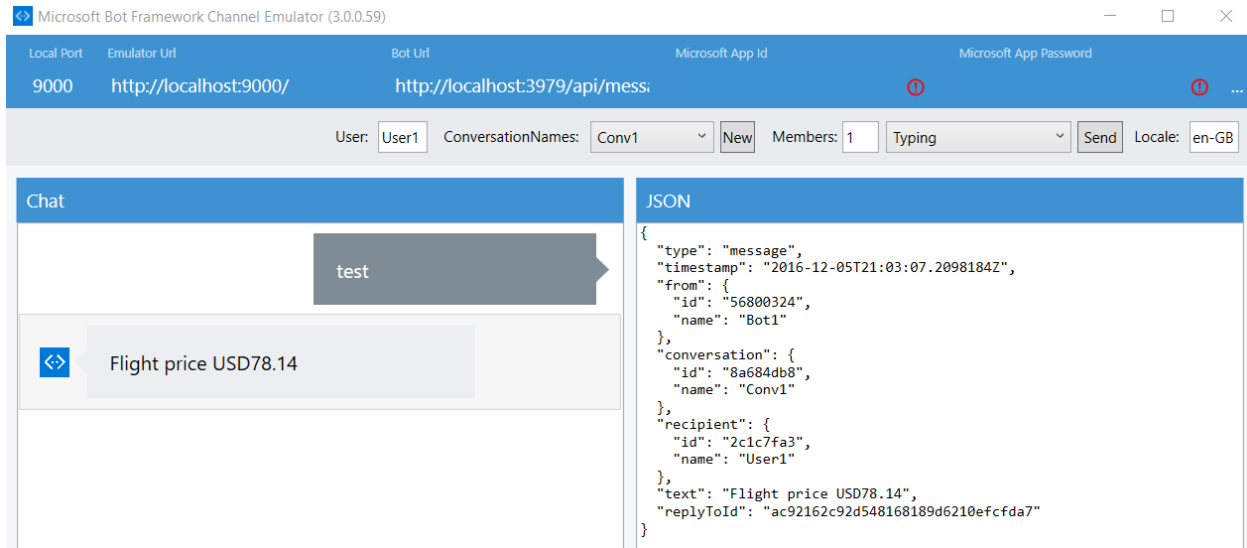


Figure 4.11: The Execution of the Basic QPX Express API Sample Code

Awesome! With just a few lines of code, we were able to retrieve a flight price using the QPX Express API.

Now, let's make our bot a bit more interesting and complex by allowing it to handle a conversation. It will ask questions related to the trip and steer the conversation with the user in such a way that, based on the answers to those questions, it will build a dynamic query to be passed to the QPX Express API to retrieve the cheapest flights for that trip.

Sounds exciting, so let's make the bot smarter. There's still quite a lot of work ahead, and the code base will become increasingly larger and more complex, but it will be nicely structured using FormFlow, which will make it relatively easy for us to navigate across.

AirfareAlertBot core logic

In order to make our bot smarter, let's take a step back and analyze what the bot should do based on the example we've just written.

As with our previous bot, this one should also be able to process multiple requests at the same time without getting stuck in any particular one, so it should process requests asynchronously.

Furthermore, all the information required to process a particular request should be gathered up front (before we attempt to process the request).

The user can also decide to cancel a request by simply typing the word **quit**. The gathered requirements should follow a defined structure with a series of basic questions presented to the user one after the other.

When a request has been submitted, the results should be presented to the user. If the user has chosen to follow a specific flight (for price change notifications), the bot will automatically message the user with the updated flight information if there's a change in price (this will be done in intervals, using a timer).

The user should be able to opt out of receiving any further price change notifications for a flight. We do this by typing the word **unfollow** followed by the request ID of the flight details previously fetched.

In essence, the information required and the execution of the request should be wrapped up into an atomic set of instructions.

With this in mind, the bot will need to request the following information from the user: origin, destination, travel dates (outbound and inbound, if applicable), number of passengers, and whether or not the user wishes to receive price change notifications.

The QPX Express API uses [IATA](#) international airport codes, which means that in order to identify the point of origin and destination, we'll need to use JSON data that maps the common name of any given airport and the city it belongs to with its corresponding IATA code.

If there are multiple airports in a particular origin or destination, the bot should be able to highlight this to users and ask which of those options they prefer.

When the point of origin and destination have been determined, the bot will then gather the remaining details, such as travel dates, number of passengers, and whether the trip is one way or round trip.

Putting this in writing makes it sound like an easy task, but in fact it is not. So, let's break down the problem into parts and do them one at a time. Doing so will make the entire process much easier to achieve and follow.

Because the bot's code base will be quite large, let's wrap up this chapter and dedicate the next one to exploring the most important parts of the AirfareAlertBot code, such as conversational state management, FormFlow code, and expanding MessagesController.cs.

Summary

In this quick chapter, we've examined the concept behind the AirfareAlertBot and we've also become briefly acquainted with the QPX Express API. We are now even able to write some code to interact with it.

We did this to get a feeling for what we will be building in the next chapter, which is the full-blown bot. We'll be exploring key aspects of the Bot Framework and will focus overall on conversational state management, conversational flow using FormFlow, conversational validation, and wiring up the bot's central hub, MessagesController.cs.

Sounds like we have quite a challenge ahead of us, so let's get rolling.

Chapter 5 Airfare Alert Bot

Introduction

Things are about to get really interesting, challenging, and, to some extent, complex.

We're going to get into what bot building is all about, such as handling conversational states and flow, asynchronous requests, and interaction with external services and performing validations.

The bot's code base will be too large to fit into this chapter, so I highly recommend that you download the full Visual Studio solution with all the code and follow along with what will be explained here.

In this chapter, we'll cover the most important parts of the code that are directly related to core bot competencies such as conversational state, flow, validations, and even the bot's brain, `MessagesController.cs`.

The bot also has quite a bit of code that deals with the QPX Express API and how it interacts with Azure Storage (tables and blobs) to store flight requests that can be followed for price changes.

Some code files that are part of the bot won't be covered in this chapter, including `Consts.cs`, `QpxExpressApiHelper.cs`, `QpxExpressCore.cs`, `TableStorage.cs`, and `TableStorageCore.cs`. However, if you download the project and check the code, you'll see comments that will help you navigate and understand what they do. I highly recommend that you do this—it will also present a bit of a challenge for you, as it will force you to see how all the pieces of the puzzle fit together.

The code is nicely structured so that it is readable and easy to follow, and the full VS solution can be found [here](#).

Web.config

By now you have downloaded the VS solution with all of the bot's code. Open the **Solution Explorer** and look for the **Web.config** file. There are some important settings that need to be edited.

Code Listing 5.0: Web.config Settings to be Edited

```
<appSettings>
  <!-- update these with yours -->
  <add key="BotId" value="AirfareAlertBot" />
  <add key="MicrosoftAppId" value="" />
  <add key="MicrosoftAppPassword" value="" />
  <add key="QpxApiKey" value="" />
```



```
<add key="FollowUpInterval" value="60000" />
<add key="StorageConnectionString" value="" />
</appSettings>
```

Let's go over these settings quickly.

The **BotId** is the bot's unique identifier. This is how the bot will be known to the Bot Connector when we register the bot on the developer portal and when we publish it on Azure App Services. I recommend you stay consistent and use the same name on all services.

The **MicrosoftAppId** and **MicrosoftAppPassword** keys are provided by the Bot Developer Portal—refer to [Chapter 3](#) for more details.

The **QpxApiKey** represents the QPX Express API key. The **FollowUpInterval** represents the number of milliseconds that the bot will wait before attempting to check if flight prices have changed.

For testing, you can use a value of 60,000 milliseconds (one minute) for **FollowUpInterval**, which means that the bot will query the QPX Express API every minute, which might be too frequent going forward (as you get only 50 free API calls per day).

I recommend that you use a higher value, such as 3,600,000 milliseconds (one hour), after the testing phase. This will probably be better, as it will lower your QPX Express API consumption rate and make your free tier last longer.

The **StorageConnectionString** represents the connection string to Azure Storage. More details about how to set up and work with Azure Storage can be found in my other e-book, [Customer Success for C# Developers Succinctly](#)—feel free to check this out.

To run the project locally on your machine, you don't need to specify the **MicrosoftAppId** or **MicrosoftAppPassword** keys—these are only required when registering it on the developer portal.

The other values are required to run the bot locally.

WebApiConfig.cs

When we create our VS project using the Bot Application template, some code is created for us behind the scenes. Part of that autogenerated code is contained within the **WebApiConfig.cs** file, which is found under the **App_Start** folder of the VS project.

In essence, a Bot Application template is nothing more than a slightly enhanced WebApi project.

Mapping airports to IATA codes

All airports throughout the world have a corresponding IATA code, which is what airlines and travel companies use when they refer to commercial flights. These codes are printed on suitcase tags and boarding passes. They are also used when making a reservation or booking a flight, so we need to be clear that all commercial flight information is therefore bound to IATA codes for both origin and destination.

By doing a simple search on the Internet for JSON IATA codes, I was able to find this [airports.json](#) file that contains a wealth of information about airports, their codes, and other interesting details such as time zone, longitude, latitude, altitude, city, and country.

By parsing this file, our bot should be able to quickly identify any origin or destination and easily retrieve the corresponding IATA codes that are required when calling the QPX Express API.

Let's create a class that implements **IDisposable** with a method responsible for parsing this JSON data file and populating a **Dictionary** object that will be accessible as a public property containing the list of airports.

Before we write this code, let's install the [RestSharp](#) library from NuGet (see [Figure 2.7](#)) that will be required in order to retrieve the JSON data.

The [Json.NET](#) library will be required to parse and deserialize the JSON response. This library has already been installed as a dependency when we created the project, which means there is no need to install it from NuGet.

Once RestSharp has been installed, let's add a new C# class file to our VS project called **FlightData.cs** under the **Controllers** folder. Here's the code.

Code Listing 5.1: FlightData.cs

```
using System.Collections.Generic;
using RestSharp;
using RestSharp.Deserializers;
using System;
using System.Threading.Tasks;
using Google.Apis.QPXExpress.v1.Data;

namespace AirfareAlertBot.Controllers
{
    // Used in order to store worldwide airport data.
    public class Airport
    {
        public string Name { get; set; }
        public string City { get; set; }
        public string Country { get; set; }
        public string Iata { get; set; }
        public string Icao { get; set; }
        public float Latitude { get; set; }
        public float Longitude { get; set; }
    }
}
```

```

        public float Altitude { get; set; }
        public string Timezone { get; set; }
        public string Dst { get; set; }
    }

    // Used in order to store a flight request.
    public class FlightDetails
    {
        public string OriginIata { get; set; }
        public string DestinationIata { get; set; }
        public string OutboundDate { get; set; }
        public string InboundDate { get; set; }
        public string NumPassengers { get; set; }
        public string NumResults { get; set; }
        public string Follow { get; set; }
        public string Direct { get; set; }
        public string UserId { get; set; }
        public int Posi { get; set; }
    }

    // Responsible for processing a flight request.
    public class ProcessFlight : IDisposable
    {
        protected bool disposed;
        public Dictionary<string, Airport> Airports { get; set; }
        public FlightDetails FlightDetails { get; set; }

        public ProcessFlight()
        {
            FlightDetails = new FlightDetails()
            {
                OriginIata = string.Empty,
                DestinationIata = string.Empty,
                OutboundDate = string.Empty,
                InboundDate = string.Empty,
                NumPassengers = string.Empty,
                NumResults = string.Empty
            };

            Airports = GetAirports();
        }

        // Gets a list of all airports worldwide.
        protected Dictionary<string, Airport> GetAirports()
        {
            string res = string.Empty;

            RestClient client = new
                RestClient(StrConsts.cStrIataCodesBase);

```

```

        RestRequest request = new
            RestRequest(StrConsts.cStrIataCodePath, Method.GET);

        request.RequestFormat = DataFormat.Json;
        IRestResponse response = client.Execute(request);
        JsonSerializer deserial = new JsonSerializer();

        return deserial.
            Deserialize<Dictionary<string, Airport>>(response);
    }

    // Checks if a string is an IATA code.
    public bool IsIataCode(string input, ref List<string> tmpList)
    {
        bool res = false;

        foreach (KeyValuePair<string, Airport> p in Airports)
        {
            if (p.Key.ToLower() == input.ToLower())
            {
                tmpList.Add(p.Key);
                res = true;

                break;
            }
        }

        return res;
    }

    // City that corresponds to an IATA code.
    public string GetAirportCity(string code)
    {
        string res = string.Empty;

        foreach (KeyValuePair<string, Airport> p in Airports)
        {
            if (p.Key.ToLower() == code.ToLower())
            {
                res = p.Value.City;
                break;
            }
        }

        return res;
    }

    // List of IATA codes.
    public List<string> GetCodesList()

```

```

{
    List<string> codes = new List<string>();

    foreach (KeyValuePair<string, Airport> p in Airports)
    {
        codes.Add(p.Key);
    }

    return codes;
}

protected bool HasAirportParamValue(string v, string p)
{
    return (p != string.Empty &&
        v.ToLower() == p.ToLower()) ? true : false;
}

// Searches for IATA codes based on the city or airport name.
public string[] GetIataCodes(string name, string city,
    string country)
{
    List<string> codes = new List<string>();

    foreach (KeyValuePair<string, Airport> p in Airports)
    {
        if (city != string.Empty)
        {
            if ((HasAirportParamValue(p.Value.Name, name) ||
                HasAirportParamValue(p.Value.City, city)) ||
                HasAirportParamValue(p.Value.Country, country))
            {
                codes.Add(p.Key + "|" + p.Value.Name);
            }
        }
        else if (name != string.Empty)
        {
            if ((HasAirportParamValue(p.Value.City, city) ||
                HasAirportParamValue(p.Value.Name, name)) ||
                HasAirportParamValue(p.Value.Country, country))
            {
                codes.Add(p.Key + "|" + p.Value.Name);
            }
        }
    }

    return codes.ToArray();
}

```

```

// Footer of a flight request.
protected static string SetOutputFooter(string guid)
{
    return Environment.NewLine +
        Environment.NewLine +
        ((guid != string.Empty) ?
        GatherQuestions.cStrGatherRequestProcessed +
        GatherQuestions.cStrGatherRequestProcessedPost +
        guid : string.Empty) +
        Environment.NewLine +
        Environment.NewLine + GatherQuestions.cStrNowSayHi;
}

// Creates a flight request output.
public static string OutputResult(string[] lines, string guid)
{
    string r = string.Empty;

    foreach (string l in lines)
        r += l + Environment.NewLine;

    if (lines.Length > 1)
        r += SetOutputFooter(guid);

    return r;
}

// Main method for processing a flight request.
public async Task<string> ProcessRequest()
{
    return await Task.Run(() =>
    {
        string guid = string.Empty;
        TripsSearchResponse result = null;

        string[] res = QpxExpressApiHelper.
            GetFlightPrices(true, Airports,
                FlightDetails, out guid, out result).ToArray();

        return OutputResult(res, guid);
    });
}

// Destructor
~ProcessFlight()
{
    Dispose(false);
}

```

```

public virtual void Dispose(bool disposing)
{
    if (!disposed)
    {
        if (disposing)
        {
            FlightDetails = null;
        }
    }
    disposed = true;
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
}
}

```

To understand this code, let's first examine the JSON data file for a moment. Figure 5.0 shows what a snippet of that data looks like.

```

{"GKA":{"name":"Goroka","city":"Goroka","country":"Papua New Guinea","iata":"GKA","icao":"AYGA","latitude":"-6.081689","longitude":"145.391881","altitude":"5282","timezone":"10","dst":"U"},"MAG":
{"name":"Madang","city":"Madang","country":"Papua New Guinea","iata":"MAG","icao":"AYMD","latitude":"-5.207083","longitude":"145.7887","altitude":"20","timezone":"10","dst":"U"},"HGU":{"name":"Mount
Hagen","city":"Mount Hagen","country":"Papua New Guinea","iata":"HGU","icao":"AYMH","latitude":"-5.826789","longitude":"144.295861","altitude":"5388","timezone":"10","dst":"U"},"LAE":
{"name":"Madzab","city":"Madzab","country":"Papua New Guinea","iata":"LAE","icao":"AYNZ","latitude":"-6.569828","longitude":"146.726242","altitude":"239","timezone":"10","dst":"U"},"POM":{"name":"Port Moresby
Jacksons Intl","city":"Port Moresby","country":"Papua New Guinea","iata":"POM","icao":"AYPY","latitude":"-9.443383","longitude":"147.22005","altitude":"146","timezone":"10","dst":"U"},"MMK":{"name":"Mewak
Intl","city":"Mewak","country":"Papua New Guinea","iata":"MMK","icao":"AYWK","latitude":"-3.583828","longitude":"143.669186","altitude":"19","timezone":"10","dst":"U"},"UAK":

```

Figure 5.0: Snippet of the JSON Data File

Notice that the data contains a key (circled in red) and a value (circled in blue). The key is a **string** and the value is an object with several properties (represented by the **Airport** class in the previous code listing).

The **Airport** class is a C# representation of the data contained within the JSON file hosted on GitHub that contains all the details of the world's airports.

This will be used to retrieve the IATA codes for the origin and destination of a flight—these are required when querying the QPX Express API.

We use the **FlightDetails** class as a placeholder to store the data submitted by a user for a specific flight request. It is also used when retrieving the details of a flight that has already been stored and is being followed for price changes. In other words, it is used to manage the state of a flight request, not the bot's state.

The **ProcessFlight** class implements the **IDisposable** interface, and it is responsible for processing a flight request.

This class contains several helper methods that are used to retrieve the list of airports, get IATA codes, and retrieve the city that corresponds to a particular IATA code.

The **GetAirports** method makes an HTTP request by using a **RestClient** instance. It deserializes the JSON data retrieved, then returns a **Dictionary<string, Airport>** object that populates the **Airports** property.

Two interesting methods of this class are the **IsIataCode** and **GetIataCodes** methods. The **IsIataCode** method loops through the **Airports Dictionary** and checks whether or not the user's input corresponds to an IATA code.

On the other hand, the **GetIataCodes** method is invoked only when the user has not entered an IATA code but instead the name of a city or airport. This method then loops through the **Airports Dictionary** and checks whether the value entered matches either a city or name of an airport, and then returns the corresponding IATA code(s).

The **GetIataCodes** method also contains a couple of **static** methods that are used when the result is displayed to the user. Possible results are **OutputResult** and **SetOutputFooter**. Both are string concatenation methods.

These methods are invoked by the **ProcessRequest async** method, which is used only when the user has submitted all the flight request details.

The **ProcessRequest** method calls the **GetFlightPrices static** method from the **QpxExpressApiHelper** class, which is responsible for invoking the QPX Express API and returning the result of the flight request query.

We'll explore the **QpxExpressApiHelper** class later. Notice that we are also using constants defined in a class named **GatherQuestions** that is defined in the **Consts.cs** file.

That concludes our mapping of IATA codes. With this finished, we can now focus on the main conversational flow of our bot, from which all requests originate, all validation processes are triggered, and all responses returned for final processing.

Handling conversational flow

One of the most difficult aspects of creating a bot is being able to handle a conversational flow with a user. If we had to handle this logic by ourselves, the complexity of creating a bot would skyrocket.

Luckily, the good folks working at Microsoft's Bot Framework team have given us a great hand. They have come up with an API that allows us to rather seamlessly create a sequence of events and provide structure for a conversational flow. This API part is known as [FormFlow](#).

FormFlow works by creating a class that contains a few variables and a **static** method. Each variable represents a question or step in the conversational flow.

The method represents the flow and executes a sequence of subevents (validations), thereby creating the structure and logic for the conversation to follow.

FormFlow handles the validation and conversational state for each step, which means it knows where the conversation is—i.e. at which stage. Validations for each step can be implemented as **async** methods.

For our bot, the conversational flow logic using FormFlow will be kept in a file called **TravelDetails.cs**.

Let's add a new C# class file to our VS project under the **Controllers** folder, and let's put in the code from Code Listing 5.2.

Code Listing 5.2: TravelDetails.cs

```
using AirfareAlertBot.Controllers;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using System;
using System.Threading.Tasks;

namespace AirfareAlertBot
{
    // This is the FormFlow class that handles
    // the conversation with the user, in order to get
    // the flight details the user is interested in.
    [Serializable]
    public class TravelDetails
    {
        // Ask the user for the point of origin for the trip.
        [Prompt(GatherQuestions.cStrGatherQOrigin)]
        public string OriginIata;

        // Ask the user for the point of destination for the trip.
        [Prompt(GatherQuestions.cStrGatherQDestination)]
        public string DestinationIata;

        // Ask the user for the outbound trip date.
        [Prompt(GatherQuestions.cStrGatherQOutboundDate)]
        public string OutboundDate;

        // Ask the user for the inbound (return) trip date
        // (if applicable - if it is not 'one way').
        [Prompt(GatherQuestions.cStrGatherQInboundDate)]
        [Optional]
        public string InboundDate;

        // Ask the user for the number of passengers.
        [Prompt(GatherQuestions.cStrGatherQNumPassengers)]
        public string NumPassengers;

        // Ask the user if the flight is direct.
        [Prompt(GatherQuestions.cStrGatherProcessDirect)]
    }
}
```

```

public string Direct;

// Ask the user if the flight is to be followed
// (check for price changes).
[Prompt(GatherQuestions.cStrGatherProcessFollow)]
public string Follow;

// FormFlow main method, which is responsible for creating
// the conversation dialog with the user and validating each
// response.
public static IForm<TravelDetails> BuildForm()
{
    // Once all the user responses have been gathered
    // send a response back that the request is being
    // processed.
    OnCompletionAsyncDelegate<TravelDetails>
        processOrder = async (context, state) =>
    {
        await context.PostAsync(
            GatherQuestions.cStrGatherProcessingReq);
    };

    // FormFlow object that gathers and handles user responses.
    var f = new FormBuilder<TravelDetails>()
        .Message(GatherQuestions.cStrGatherValidData)
        .Field(nameof(OriginIata),
            // Validates the point of origin submitted.
            validate: async (state, value) =>
            {
                return await Task.Run(() =>
                {
                    Data.currentText = value.ToString();

                    return FlightFlow.CheckValidateResult(
                        FlightFlow.ValidateAirport(state,
                            Data.currentText, false));
                });
            })
        .Message("{OriginIata} selected")
        .Field(nameof(DestinationIata),
            // Validates the point of destination submitted.
            validate: async (state, value) =>
            {
                return await Task.Run(() =>
                {
                    Data.currentText = value.ToString();

                    return FlightFlow.CheckValidateResult(

```

```

        FlightFlow.ValidateAirport(state,
        Data.currentText, true));
    });
})
.Message("{DestinationIata} selected")

.Field(nameof(OutboundDate),
// Validates the outbound travel date submitted.
validate: async (state, value) =>
{
    return await Task.Run(() =>
    {
        Data.currentText = value.ToString();

        return FlightFlow.CheckValidateResult(
            FlightFlow.ValidateDate(state,
            Data.currentText, false));
    });
})
.Message("{OutboundDate} selected")

.Field(nameof(InboundDate),
// Validates the inbound travel date submitted
// (or if it is a one-way trip).
validate: async (state, value) =>
{
    return await Task.Run(() =>
    {
        Data.currentText = value.ToString();

        return FlightFlow.CheckValidateResult(
            FlightFlow.ValidateDate(state,
            Data.currentText, true));
    });
})
.Message("{InboundDate} selected")

.Field(nameof(NumPassengers),
// Validates the number of passengers
// submitted for the trip.
validate: async (state, value) =>
{
    return await Task.Run(() =>
    {
        Data.currentText = value.ToString();

        return FlightFlow.CheckValidateResult(
            FlightFlow.ValidateNumPassengers(state,
            Data.currentText));
    });
})

```

```

    });
  })
  .Message("{NumPassengers} selected")

  .Field(nameof(Direct),
    // Validates whether the trip is direct or not.
    validate: async (state, value) =>
    {
      return await Task.Run(() =>
      {
        Data.currentText = value.ToString();

        return FlightFlow.CheckValidateResult(
          FlightFlow.ValidateDirect(state,
            Data.currentText));
      });
    })
  .Message("{Direct} selected")

  .Field(nameof(Follow),
    // Validates if the user has submitted
    // the flight to be
    // followed for price changes.
    validate: async (state, value) =>
    {
      return await Task.Run(() =>
      {
        Data.currentText = value.ToString();

        ValidationResult res =
          FlightFlow.CheckValidateResult(
            FlightFlow.ValidateFollow(state,
              Data.currentText));

        FlightFlow.
          AssignStateToFlightData(res, state);

        return res;
      });
    })
  .Message("{Follow} selected")

  // When all the data has been
  // gathered from the user...
  .OnCompletion(processOrder)
  .Build();

  return f;
}

```

```
}  
}
```

If we take a step back and look at this code, we can see that it's actually not a lot, and it's based on method chaining in which each step follows the previous one.

Because **FormFlow** handles the conversational state, it knows where to pick up the conversation and which **validate** method to execute. Not only is this awesome, but it's also convenient and easy to follow. So, let's dive into the details.

First, notice that the **TravelDetails** class is decorated with the **Serializable** attribute.

The **TravelDetails** class contains a series of variables in which each represents a response that the bot will gather from the user. Each variable is marked with the **Prompt** attribute that will indicate the question that the user will be asked to answer.

The only optional variable within all the conversation is the one that corresponds to the **InboundDate**, which is marked with the **Optional** attribute. This happens in response to requests for flight details without a return trip (i.e. a one-way trip).

The other variables are not optional and are thus required during the conversation. The magic happens inside the **BuildForm** method. This method controls the conversational flow and the state of the conversation, and it performs validations during each step for each input required. The **BuildForm** method contains two main parts.

The first part is a delegate called **OnCompletionAsyncDelegate<TravelDetails>** that gets executed when all the details from the user (represented by the variables within the class) have been gathered.

The second part is a **FormBuilder<TravelDetails>** instance that gets created and that contains a series of methods that are chained together and executed one after the other following the sequence that we want the conversation to follow.

Most importantly, for each variable defined within the **TravelDetails** class, **FormFlow** runs a **validate** method that can execute any custom code we want invoked, thus validating the value of that the variable—returning a **ValidateResult** object that determines if **FormFlow** can continue the conversation or requires the user to re-enter the value for the current variable. This is how conversational state management is achieved.

Conversational state, which includes the values of each of the variables defined within the **TravelDetails** class, is available to each validate method within each step through the **state** parameter.

In my opinion, the way **FormFlow** handles the flow within a conversation and makes it incredibly easy is one of the most important and noticeable features of the Bot Framework.

Be sure to note that **FormFlow** doesn't require the validation methods to be asynchronous, but in order to make the bot as responsive as possible, I decided to mark each one as **async** and write each custom validation logic inside the **Run** method of a **Task** object.

The validation logic itself has been abstracted and wrapped up nicely in a separate **static** class called **FlightFlow** that we will explore next.

Custom flow validations

The best part of working with FormFlow is being able to write custom validations for each variable that represents a stage in the conversation.

You could write each validation inside each **Validate** method of **BuildForm**, however that would not look particularly good, and it would not be easy to manage as the code base grows. So, my recommendation is to write this logic in a separate class and then invoke it as needed.

In order to do that, let's create a new C# class file inside our **Controllers** folder of our VS project and call it **FlightFlow.cs**.

Code Listing 5.3: FlightFlow.cs

```
using Microsoft.Bot.Builder.FormFlow;
using System;
using System.Collections.Generic;

namespace AirfareAlertBot.Controllers
{
    // Handles the conversation flow with the users.
    public partial class FlightFlow
    {
        // Interprets an unfollow request command from the user.
        public static bool ProcessUnfollow(string value, ref
            ValidateResult result)
        {
            bool res = false;

            if (value.ToLower().Contains(
                GatherQuestions.cStrUnFollow.ToLower()))
            {
                // Use Azure Table storage.
                using (TableStorage ts = new TableStorage())
                {
                    string guid = value.ToLower().Replace(
                        GatherQuestions.cStrUnFollow.ToLower(),
                        string.Empty);

                    // Remove the flight details for the
                    // guide being followed.
                    if (ts.RemoveEntity(guid))
                    {
                        string msg =
                            GatherQuestions.cStrNoLongerFollowing +

```

```

        guid;
        result = new ValidateResult { IsValid = false,
            Value = msg };
        result.Feedback = msg;

        res = true;
    }
    else
        result = new ValidateResult { IsValid = false,
            Value = GatherErrors.cStrNothingToUnfollow };
    }
}

return res;
}

// Validates that the response to the user is never empty.
public static ValidateResult CheckValidateResult(
    ValidateResult result)
{
    result.Feedback = ((result.Feedback == null ||
        result.Feedback == string.Empty) &&
        result.Value.ToString() == string.Empty) ?
        GatherQuestions.cStrGatherValidData : result.Feedback;

    return result;
}

// Validates the user's response for a direct flight (or not).
public static ValidateResult ValidateDirect(TravelDetails state,
    string value)
{
    ValidateResult result = new ValidateResult
        { IsValid = false, Value = string.Empty };

    string direct = (value != null) ? value.Trim() :
        string.Empty;

    if (ProcessUnfollow(direct, ref result))
        return result;

    if (direct != string.Empty)
    {
        // If direct flight response is correct.
        if (direct.ToLower() == GatherQuestions.cStrYes ||
            direct.ToLower() == GatherQuestions.cStrNo)
            return new ValidateResult
                { IsValid = true, Value = direct.ToUpper() };
        else

```

```

        {
            if (result.Feedback != null)
                result.Feedback =
                    GatherQuestions.cStrGatherProcessTryAgain;
        }
    }
else
{
    if (result.Feedback != null)
        result.Feedback =
            GatherQuestions.cStrGatherProcessTryAgain;
    }

    return result;
}

// Validates the user's response to follow a flight request
// (for price changes).
public static ValidateResult ValidateFollow(
    TravelDetails state, string value)
{
    ValidateResult result = new ValidateResult
        { IsValid = false, Value = string.Empty };

    string follow = (value != null) ? value.Trim() :
        string.Empty;

    if (ProcessUnfollow(follow, ref result))
        return result;

    if (follow != string.Empty)
    {
        // If the response to follow a flight is correct.
        if (follow.ToLower() == GatherQuestions.cStrYes ||
            follow.ToLower() == GatherQuestions.cStrNo)
            return new ValidateResult
                { IsValid = true, Value = follow.ToUpper() };
        else
        {
            if (result.Feedback != null)
                result.Feedback =
                    GatherQuestions.cStrGatherProcessTryAgain;
        }
    }
else
    result.Feedback =
        GatherQuestions.cStrGatherRequestProcessedNoGuid;

    return result;
}

```



```

}

// Validates the user's number of passengers response.
public static ValidateResult ValidateNumPassengers(
    TravelDetails state, string value)
{
    ValidateResult result = new ValidateResult
    { IsValid = false, Value = string.Empty };

    string numPassengers = (value != null) ? value.Trim() :
        string.Empty;

    if (ProcessUnfollow(numPassengers, ref result))
        return result;

    if (numPassengers != string.Empty)
    {
        // Verifies the number of passengers and if correct.
        result = ValidateNumPassengerHelper.
            ValidateNumPassengers(numPassengers);

        if (!result.IsValid && (result.Feedback == null ||
            result.Feedback == string.Empty))
            result.Feedback =
                GatherQuestions.cStrGatherProcessTryAgain;
    }
    else
    {
        if (result.Feedback != null)
            result.Feedback =
                GatherQuestions.cStrGatherProcessTryAgain;
    }

    return result;
}

// Validates the user's travel date
// (outbound or inbound) response.
public static ValidateResult ValidateDate(TravelDetails state,
    string value, bool checkOutInDatesSame)
{
    ValidateResult result = new ValidateResult
    { IsValid = false, Value = string.Empty };
    string date = (value != null) ? value.Trim() : string.Empty;

    if (ProcessUnfollow(date, ref result))
        return result;

    if (checkOutInDatesSame && value.ToLower().Contains(

```

```

        GatherQuestions.cStrGatherProcessOneWay))
    return new ValidateResult { IsValid = true, Value =
        GatherQuestions.cStrGatherProcessOneWay };

if (date != string.Empty)
{
    DateTime res;

    // If it is a proper date.
    if (DateTime.TryParse(value, out res))
    {
        if (checkOutInDatesSame)
        {
            // Performs the actual date validation.
            result = ValidateDateHelper.
                ValidateGoAndReturnDates(ValidateDateHelper.
                    ToDateTime(state.OutboundDate),
                    ValidateDateHelper.ToDateTime(value),
                    ValidateDateHelper.
                    FormatDate(value));
        }
        else
        {
            // If it is a date in the future.
            result = ValidateDateHelper.IsFutureDate(
                ValidateDateHelper.ToDateTime(value),
                ValidateDateHelper.FormatDate(value));
        }
    }
    else
    {
        if (result.Feedback != null)
            result.Feedback =
                GatherQuestions.cStrGatherProcessTryAgain;
    }
}
// If it is not a proper date.
else
{
    if (result.Feedback != null)
        result.Feedback =
            GatherQuestions.cStrGatherProcessTryAgain;
}

return result;
}

// Validates the user's response for origin and destination.
public static ValidateResult ValidateAirport(TravelDetails state,
    string value, bool checkOrigDestSame)
{

```

```

bool isValid = false;

List<string> values = new List<string>();
ValidateResult result = new ValidateResult
    { IsValid = false, Value = string.Empty };

string city = (value != null) ? value.Trim() : string.Empty;

if (ProcessUnfollow(city, ref result))
    return result;

if (city != string.Empty)
{
    // Get the IATA code (if any) corresponding
    // to the user input.
    isValid = Data.fد.IsIataCode(value.ToString(),
        ref values);

    if (isValid)
    {
        // Processes the IATA code response.
        result = ValidateAirportHelpers.
            ProcessAirportIataResponse(state,
                checkOrigDestSame, values.ToArray());
    }
    // When multiple airports are found for a given city.
    else
    {
        // Get all the IATA codes for all the
        // airports in a city.
        string[] codes =
            InternalGetIataCodes(value.ToString().Trim());

        if (codes.Length == 1)
        {
            // When the specific match is found.
            result = ValidateAirportHelpers.
                ProcessAirportResponse(state,
                    checkOrigDestSame, codes);
        }
        else if (codes.Length > 1)
        {
            // When multiple options are found.
            result = new ValidateResult
                { IsValid = isValid, Value = string.Empty };
            result = ValidateAirportHelpers.
                GetOriginOptions(codes, result);
        }
        else
    }
}

```

```

        {
            if (result.Feedback != null)
            {
                result = new ValidateResult
                { IsValid = isValid,
                  Value = string.Empty };
                result.Feedback = GatherQuestions.
                    cStrGatherProcessTryAgain;
            }
        }
    }
    return result;
}
}
}

```

Let's analyze this code to understand what is happening. First, notice that this is a **partial** class, which means there's another part of the class contained within another .cs file that we will examine shortly.

This separation of classes makes the code easier to read and follow, and it makes the code more manageable. However, it is not a must.

This **partial** class is not making any use of the Bot Framework's features, so it is merely acting as a container for the validation logic that the **BuildForm** method invokes when validating each step in the flow of conversation. It's a way to keep our main validation logic all in one place.

The first method within this class is called **ProcessUnfollow**, and it removes a flight request that has been saved on Azure Storage (in order to receive price alert changes), so it is no longer watched.

This method is executed on all validations, as it is possible for a user to issue an unfollow command during each step of the conversation.

Following that, the next method is called **CheckValidateResult**. Its purpose is to ensure that any **ValidateResult** object returned by any of the validation steps does not return an empty **Feedback** property. When the **IsValid** property of **ValidateResult** is **false**, the bot is prevented from sending back an empty **string** response.

The **ValidateFollow** method checks if the user has typed a valid response for the follow question (if a flight request is to be followed)—either a Yes or a No.

The **ValidateDirect** method does the same for the direct question (whether a flight is direct or it allows multiple connections).

The **ValidateNumPassengers** method validates if the number of passengers is a number between 1 and 100.

The **ValidateDate** method is slightly more complex, as it not only verifies that the **OutboundDate** and **InboundDate** (if applicable) are correct, but it also checks that the dates are not in the past and that the **InboundDate** is a future date compared to the **OutboundDate**.

Finally, the most interesting and complex validation method within this **partial** class is the **ValidateAirport** method. This is responsible for validating that the origin and destination airports are both correct—that they exist (are valid IATA codes) and, if the user has indicated multiple airports for the city, that the existing options are returned so that the user can choose any option.

Notice how all these validation methods return a **ValidateResult** object. Even though their logic seems quite straightforward, there's still some code required in order to carry out a proper validation.

In this sense, and to keep the code as clean as possible, you've probably noticed that some validation methods internally invoke their own validation helper methods (which are contained in separate classes). These are responsible for handling very specific validation logic for each of the steps in the conversation flow. We'll see this code later.

Some of these validation helper classes are **ValidateNumPassengerHelper**, **ValidateDateHelper**, and **ValidateAirportHelpers**.

With **FlightFlow.cs** out of the way, let's explore the remaining bits of the **FlightFlow** **partial** class. I've placed this in a C# class file called **FlightFlowData.cs**.

Code Listing 5.4: FlightFlowData.cs

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

namespace AirfareAlertBot.Controllers
{
    // Keeps the state of the conversation.
    public class Data
    {
        public static ProcessFlight fd = null;
        public static StateClient stateClient = null;
        public static string channelId = string.Empty;
        public static string userId = string.Empty;
        public static Activity initialActivity = null;
        public static ConnectorClient initialConnector = null;
        public static string currentText = string.Empty;
    }

    public partial class FlightFlow
    {
        // Gets relevant IATA codes for a user's response.
    }
}
```

```

private static string[] InternalGetIataCodes(object value)
{
    string find = value.ToString().Trim();
    string[] codes = null;

    codes = Data.fd.GetIataCodes(string.Empty,
        find, string.Empty);

    if (codes.Length == 0)
        codes = Data.fd.GetIataCodes(find,
            string.Empty, string.Empty);

    return codes;
}

// Set the bot's state as the internal state.
public static void AssignStateToFlightData(
    ValidationResult result, TravelDetails state)
{
    if (result.IsValid)
    {
        string userId = Data.fd.FlightDetails.UserId;

        Data.fd.FlightDetails = new FlightDetails()
        {
            OriginIata = state.OriginIata,
            DestinationIata = state.DestinationIata,
            OutboundDate = state.OutboundDate,
            InboundDate = state.InboundDate,
            NumPassengers = state.NumPassengers,
            NumResults = "1",
            Direct = state.Direct,
            UserId = userId,
            Follow = result.Value.ToString()
        };
    }
}
}
}
}

```

The remaining bit of the **FlightFlow** class contains just a few methods.

The **InternalGetIataCodes** method is responsible for retrieving the corresponding IATA codes based on the user's response. It is invoked by the **ValidateAirport** method.

AssignStateToFlightData simply assigns the value of the bot's state to an instance of **FlightDetails** that is assigned to the **Data.fd.FlightDetails** property, which will be used mostly by the **CheckForPriceUpdates** method.

This wraps up the main custom validation logic. Imagine for a moment that all this code had been placed inside the validation methods of **BuildForm** within the **TravelDetails** class. That would have been quite a mess. The code would be incredibly difficult to understand and maintain. However, with this approach, we have a clear separation of concerns and code readability.

Internal validation helpers

Before totally closing off validations, let's have a look at each of the validation helper classes described so far, starting with the code for **ValidateAirportHelpers.cs**.

Code Listing 5.5: ValidateAirportHelpers.cs

```
using Microsoft.Bot.Builder.FormFlow;
using System;

namespace AirfareAlertBot.Controllers
{
    // A set of helper methods used to validate
    // origin and destination airports.
    public class ValidateAirportHelpers
    {
        // Checks that the origin and destination are not the same.
        private static ValidateResult CheckOrigDestState(
            TravelDetails state, string field)
        {
            ValidateResult result = new ValidateResult
            { IsValid = false, Value = string.Empty };
            result.Feedback = GatherErrors.cStrGatherSameCities;

            if (state?.OriginIata.ToLower() != field.ToLower())
                result = new ValidateResult
                { IsValid = true, Value = field };

            return result;
        }

        // Checks that the IATA code submitted by the
        // user for origin or destination is a valid code.
        public static ValidateResult ProcessAirportIataResponse(
            TravelDetails state, bool checkOrigDestSame, string[] items)
        {
            string field = string.Empty;
            string[] airport =
                new string[] { items[0] + "|" +
                    Data.fد.GetAirportCity(items[0]) };

            ValidateResult result = ProcessPrefix(state,
```

```

        checkOrigDestSame, airport, out field);
    result.Feedback = (result.IsValid) ? field :
        GatherErrors.cStrGatherSameCities;

    return result;
}

// Part of the validation of the origin and destination checks.
private static ValidateResult ProcessPrefix(TravelDetails state,
    bool checkOrigDestSame, string[] codes, out string field)
{
    string[] code = codes[0].Split('|');

    string prefix = !checkOrigDestSame ? "Origin" :
        "Destination";
    field = $"{prefix}: {code[1]} ({code[0]}";

    ValidateResult result = (checkOrigDestSame) ?
        CheckOrigDestState(state, code[0]) :
        new ValidateResult { IsValid = true, Value = code[0] };

    return result;
}

// Part of the validation of the origin and destination checks.
public static ValidateResult ProcessAirportResponse(
    TravelDetails state, bool checkOrigDestSame, string[] codes)
{
    string field = string.Empty;

    ValidateResult result = ProcessPrefix(state,
        checkOrigDestSame, codes, out field);
    result.Feedback = (result.IsValid) ? field :
        GatherErrors.cStrGatherSameCities;

    return result;
}

// Show the user the various airport options available.
public static ValidateResult GetOriginOptions(string[] values,
    ValidateResult result)
{
    result.Feedback = GatherErrors.cStrGatherMOrigin +
        Environment.NewLine + Environment.NewLine;

    foreach (string o in values)
    {
        string[] parts = o.Split('|');
        string newLine = $"{parts[0]} = {parts[1]}";
    }
}

```



```

        result.Feedback += newLine + Environment.NewLine +
            Environment.NewLine;
    }

    return result;
}
}
}

```

We use these methods to validate the data associated with the origin and destination airports. Let's go over each one of them briefly.

The **CheckOrigDestState** method checks that the origin and destination airports entered by the user are not the same.

The **ProcessAirportIataResponse** method checks that the IATA codes for origin and destination are actually valid. This is done in conjunction with the **ProcessPrefix** method, which is **private**.

The **ProcessAirportResponse** method is similar to **ProcessAirportIataResponse**, with the difference being that **ProcessAirportResponse** is invoked after multiple airports have been found in a specific city.

Finally, the **GetOriginOptions** method is used to show the user which airport options have been found, which allows the user to choose one.

That concludes the **ValidateAirportHelpers** class. Let's now explore **ValidateDateHelper.cs**.

Code Listing 5.6: ValidateDateHelper.cs

```

using Microsoft.Bot.Builder.FormFlow;
using System;
using System.Collections.Generic;

namespace AirfareAlertBot.Controllers
{
    // This helper class is used to validate trip dates.
    public class ValidateDateHelper
    {
        // Parses and converts a string date to a DateTime date.
        public static DateTime ToDateTime(string date)
        {
            return DateTime.Parse(date);
        }

        // Determines the number of days between two dates.
        private static int DaysBetween(DateTime d1, DateTime d2)
        {

```

```

{
    TimeSpan span = d2.Subtract(d1);
    return (int)span.TotalDays;
}

// Formats the date to a more human readable way ;)
public static string FormatDate(string dt)
{
    string res = dt;
    List<string> nParts = new List<string>();

    string tmp = dt.Replace("/", "-").
        Replace(" ", "-").Replace(".", "-");
    string[] parts = tmp.Split('-');

    if (parts?.Length > 0)
    {
        int j = 1;
        foreach (string str in parts)
        {
            if (str != "-")
            {
                string t = string.Empty;

                if (j == 2)
                    t = str.Substring(0, 3);
                else
                    t = str;

                nParts.Add(t);
            }

            j++;
        }

        if (nParts.Count > 0)
        {
            int i = 1;
            List<string> pParts = new List<string>();

            foreach (string p in nParts)
            {
                if (p.Length < 2 && (i == 1))
                    pParts.Add(p.PadLeft(2, '0'));
                else if (p.Length == 2 && (i == 3))
                {
                    DateTime n = ToDateTime(dt);
                    pParts.Add(n.Year.ToString());
                }
            }
        }
    }
}

```

```

        else
            pParts.Add(p);

        i++;
    }

    res = string.Join("-", pParts.ToArray()).ToUpper();
}

return res;
}

// Checks if a date is in the future.
public static ValidateResult IsFutureDate(DateTime dt, string
value)
{
    ValidateResult result = new ValidateResult
        { IsValid = false, Value = string.Empty };

    if (DaysBetween(DateTime.Now, dt) >= 0)
        result = new ValidateResult
            { IsValid = true, Value = value };
    else
        result.Feedback = GatherErrors.cStrGatherStatePastDate;

    return result;
}

// Main method responsible for validating trip dates.
public static ValidateResult ValidateGoAndReturnDates(DateTime
go, DateTime comeback, string value)
{
    ValidateResult result = new ValidateResult
        { IsValid = false, Value = string.Empty };

    result = new ValidateResult
        { IsValid = false, Value = string.Empty };

    if (DaysBetween(go, comeback) >= 0)
        result = new ValidateResult
            { IsValid = true, Value = value };
    else
        result.Feedback = GatherErrors.cStrGatherStateFutureDate;

    return result;
}
}
}
}

```

The main purposes of this helper class are to ensure that date validations are correct and to format the date as an easily readable response. Let's briefly go over the methods of this class.

The first two methods, **ToDateTime** and **DaysBetween**, convert a **string** date into **DateTime** object and calculate the number of days between two dates, respectively.

The **FormatDate** method, as its name implies, formats the date string, which makes it easier for humans to read.

The method **IsFutureDate** checks whether a date is in the future. This is quite an important requirement when looking for a flight.

Finally, **ValidateGoAndReturnDates** is the main method for validating the outbound and inbound trip dates. The main check it performs is verifying that the difference between the inbound and outbound dates is at least on the same day or in the future.

That concludes the **ValidateDateHelper** class. Let's now check **ValidateNumPassengerHelper.cs**.

Code Listing 5.7: ValidateNumPassengerHelper.cs

```
using Microsoft.Bot.Builder.FormFlow;
using System;

namespace AirfareAlertBot.Controllers
{
    public class ValidateNumPassengerHelper
    {
        // Validates the number of passenger responses.
        public static ValidateResult ValidateNumPassengers(string value)
        {
            ValidateResult result = new ValidateResult
            { IsValid = false, Value = string.Empty };

            try
            {
                int res = Convert.ToInt32(value);

                if (res >= 1 && res <= 100)
                {
                    result = new ValidateResult
                    { IsValid = true, Value = value };
                }
                else
                {
                    result.Feedback =
                        GatherErrors.cStrGatherStateInvalidNumPassengers;
                }
            }
            catch { }

            return result;
        }
    }
}
```

```
}
```

This class contains one method called **ValidateNumPassengers** that checks that the number of passengers is a valid integer number between 1 and 100.

With these validations explained, let's now move our attention to `MessagesController.cs`, which is where the bot receives the messages from users and replies to them.

The bot's central hub

The `MessagesController.cs` file is the main entry point for our bot to receive and send messages.

When we selected the Bot Application template to create our VS solution, some very basic code was added by default to the `MessagesController.cs` file (which we explored in [Chapter 2](#)).

However, given that our bot is now quite complex, let's remove that boilerplate code and use this one.

Code Listing 5.8: MessagesController.cs

```
using System;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;
using Microsoft.Bot.Connector;
using AirfareAlertBot.Controllers;

using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using System.Timers;
using Microsoft.Azure;
using System.Web.Http.Controllers;

namespace AirfareAlertBot
{
    // Main class responsible for main user-to-bot interactions.
    [BotAuthentication]
    public class MessagesController : ApiController
    {
        protected bool disposed;

        // Timer used to check for flight price changes.
        private static Timer followUpTimer = null;
        // Set to true when flight price changes are checked.
        private static bool timerBusy = false;
    }
}
```

```

// Set to true when user-to-bot interaction is ongoing.
private static bool msgBusy = false;

// Initializes the flight price changes check Timer.
private static void SetFollowUpTimer()
{
    if (followUpTimer == null)
    {
        followUpTimer = new Timer();
        followUpTimer.Elapsed += new
            ElapsedEventHandler(OnTimedEvent);

        double interval = 10000;

        try
        {
            string fupInterval =
                CloudConfigurationManager.GetSetting(
                    StrConsts.cStrFollowUpInterval);
            interval = Convert.ToDouble(fupInterval);
        }
        catch { }

        followUpTimer.Interval = interval;
        followUpTimer.Enabled = true;
        timerBusy = false;
    }
}

// Main method for running the flight price changes check.
private static Task ProcessTimer()
{
    return Task.Run(async () =>
    {
        return await Task.Run(async () =>
        {
            bool changed = false;

            // The bot checks in Azure Table storage...
            using (TableStorage ts = new TableStorage())
            {
                // ...If a stored flight request has had any
                // price changes
                // and if so, send the user this information...
                changed = await
                    ts.CheckForPriceUpdates(
                        Data.fد.Airports,
                        Data.fد.FlightDetails,
                        Data.initialActivity,

```

```

        Data.initialConnector,
        Data.currentText);

        timerBusy = false;
    }

    return changed;
});
});
}

// Triggers the flight price changes check.
private static void OnTimedEvent(object source,
    ElapsedEventArgs e)
{
    if (!timerBusy && !msgBusy)
    {
        timerBusy = true;
        ProcessTimer();
    }
}

// FormFlow end method: Executed once the request
// details have been gathered from the user.
internal static IDialog<TravelDetails> MakeRootDialog()
{
    return Chain.From(() => FormDialog.FromForm(
        TravelDetails.BuildForm, FormOptions.None))
        .Do(async (context, order) =>
        {
            try
            {
                var completed = await order;

                // Request processed.
                string res = await Data.fd.ProcessRequest();

                // Request result sent to the user.
                await context.PostAsync(res);

                Data.fd.FlightDetails = null;
            }
            // This also gets executed when a 'quit' command
            // is issued.
            catch (FormCanceledException<TravelDetails> e)
            {
                Data.fd.FlightDetails = null;

                string reply = string.Empty;
            }
        })
}

```

```

        if (e.InnerException == null)
            reply = GatherQuestions.cStrQuitMsg;
        else
            reply = GatherErrors.cStrShortCircuit;

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

// Inits the bot's conversational internal state.
private void InitState(Activity activity)
{
    if (Data.stateClient == null)
        Data.stateClient = activity.GetStateClient();

    if (Data.fd.FlightDetails == null)
        Data.fd.FlightDetails = new FlightDetails();

    Data.fd.FlightDetails.UserId = activity.From.Id;
}

// Inits and invokes the ProcessFlight constructor.
private void InitFlightData()
{
    if (Data.fd == null)
        Data.fd = new ProcessFlight();
}

// Gets the user and channel IDs of the conversation.
private void GetUserAndChannelId(Activity activity,
    ConnectorClient connector)
{
    Data.channelId = activity.ChannelId;
    Data.userId = activity.From.Id;
    Data.initialActivity = activity;
    Data.initialConnector = connector;
}

~MessagesController()
{
    Data.fd.Dispose();
}

// Send the bot's default welcome message to the user.
private async void SendWelcomeMsg(ConnectorClient connector,
    Activity activity)
{

```



```

        if (Data.fd == null)
        {
            Activity reply = activity.CreateReply(
                GatherQuestions.cStrWelcomeMsg);

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

    // Initializes the bot.
    protected override void Initialize(HttpControllerContext
        controllerContext)
    {
        base.Initialize(controllerContext);
        SetFollowUpTimer();
    }

    // Process an unfollow command outside a conversation.
    private async void ProcessUnfollow(ConnectorClient connector,
        Activity activity)
    {
        if (activity.Text.ToLower().Contains(
            GatherQuestions.cStrUnFollow)) {
            ValidateResult r = new ValidateResult { IsValid = false,
                Value = GatherErrors.cStrNothingToUnfollow };

            if (FlightFlow.ProcessUnfollow(activity.Text, ref r))
            {
                Activity reply = activity.CreateReply(r.Feedback);

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

    // This is the bot's main entry point for all user responses.
    public async Task<HttpResponseMessage> Post([FromBody]Activity
        activity)
    {
        try
        {
            // Inits the flight price change check timer.
            SetFollowUpTimer();

            // Set the user-to-bot conversational status as ongoing.
            msgBusy = true;
        }
    }

```

```

        // Inits the Bot Framework Connector service.
        ConnectorClient connector = new
            ConnectorClient(new Uri(activity.ServiceUrl));
        // Gets the user and channel IDs.
        GetUserAndChannelId(activity, connector);

        // When the user has typed a message
        if (activity.Type == ActivityTypes.Message)
        {
            // Let's greet the user.
            SendWelcomeMsg(connector, activity);

            // Init the state and flight request.
            InitFlightData();
            InitState(activity);

            ProcessUnfollow(connector, activity);

            // Send the FormBuilder conversational dialog.
            await Conversation.SendAsync(activity,
                MakeRootDialog);
        }
        else
            await HandleSystemMessage(connector, activity);
    }
    catch { }

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

    // A response has been sent back to the user.
    msgBusy = false;

    return response;
}

private Task<Activity> HandleSystemMessage(ConnectorClient
    connector, Activity message)
{
    // Not used for now...
    // Here put any logic that gets on any of these.
    // Activity Types
    if (message.Type == ActivityTypes.DeleteUserData)
    {
    }
    else if (message.Type == ActivityTypes.ConversationUpdate)
    {
    }
    else if (message.Type == ActivityTypes.ContactRelationUpdate)
    {
    }
}

```

```

    }
    else if (message.Type == ActivityTypes.Typing)
    {
    }
    else if (message.Type == ActivityTypes.Ping)
    {
    }

    return null;
}
}
}

```

Now, let's understand what is going on here.

First and foremost, the **MessagesController** class implements the **ApiController** interface. As we've seen, this application is nothing more than a slightly modified WebApi project.

Three **private static** variables have been declared—**followUpTimer**, **timerBusy**, and **msgBusy**.

The variable **followUpTimer** is of type **System.Timers.Timer**, which is a timer that gets executed every **FollowUpInterval** (found in the Web.config file) number of milliseconds.

The sole purpose of the **followUpTimer** object is to execute code that checks if price changes have occurred for flights that are being watched and inform the user that is following them.

The **timerBusy** variable is used as a semaphore so that the code being executed when **followUpTimer** goes off doesn't get triggered a second time.

The **msgBusy** variable also acts as a semaphore—it indicates that communication between the user and the bot is taking place. It is also used to prevent execution of any other code during that moment.

The method **SetFollowUpTimer** is used to initialize the **followUpTimer**. This method sets up all the necessary parameters that **followUpTimer** requires.

The **ProcessTimer** method is responsible for running all the code that checks if flights being followed by the user have had any price changes and informing the user in such an event.

The **OnTimedEvent** method, as its name implies, is triggered every **FollowUpInterval** number of milliseconds. It invokes the **ProcessTimer** method.

At this stage, we are mostly done with the code that checks for flight price changes. Here comes the interesting bit—next on the list is the **MakeRootDialog** method. This is a unique and interesting method because it is required when we work with FormFlow. It creates a **FormDialog** instance based on the **IForm<TravelDetails>** object returned by **TravelDetails.BuildForm**. In other words, it literally creates the FormFlow dialog.

The built-in **Do** method contained within **MakeRootDialog** gets executed when all the flight request details have been gathered from the user. There are two parts to this **Do** method.

The first part, contained within the **try** section, gets executed when all the details from the flight request have been gathered from the user. This leads to the execution of the actual request itself, which is accomplished by invoking **Data.fd.ProcessRequest**. The result of **ProcessRequest** is returned to the user when **context.PostAsync** gets called.

The second part, contained within the **catch** section, gets executed either when an exception within the **try** section occurs or when the user cancels the request by typing the word **quit**. If **e.InnerException** is **null**, the user has cancelled the request. In any case, a response is sent back to the user by calling **context.PostAsync**.

The **InitState** method initializes the bot's internal conversational state. This is necessary in order for the bot to start off with a clean slate with the user.

The **InitFlightData** method is responsible for creating an instance of the **ProcessFlight** class that will be used throughout the bot's lifecycle to process flight requests.

The **GetUserAndChannelId** method sets the **channelId**, **userId**, **initialActivity**, and **initialConnector** properties, which are needed to check for flight price changes.

The **SendWelcomeMsg** method is super simple—it creates an **Activity** instance called **reply** that is sent as a response to the user when the conversation starts.

Keep in mind that the user always starts the conversation by sending a message to the bot.

The **Initialize** method invokes the inherited **Initialize** method from **ApiController** and calls the **SetFollowUpTimer** method.

The **ProcessUnfollow** method executes an unfollow command outside of a conversation when the previous conversation has finalized (and before a new conversation has started). This method is invoked within the **Post** method.

The **Post** method is another crucial function for the bot. This is the main entry point for all user input.

Within the **Post** method, several initialization methods are called. We've previously examined these, so we know when a user has sent a message because the type of **activity.Type** is equal to **ActivityTypes.Message**. There are several other **ActivityTypes** that can be handled separately within the **HandleSystemMessage** method, but these are not used by our bot.

The really interesting part of the **Post** method comes when it internally invokes the **Conversation.SendAsync** method. I mean interesting in the sense that an **activity** instance is passed as a parameter (which corresponds to the user's input) along with a callback to the **MakeRootDialog** method.

MakeRootDialog creates the **FormFlow** instance that handles all the conversational flow with the user and keeps track of the state of the conversation. In essence, every time the **Conversation.SendAsync** method is called, the **Post** method passes the user's input (represented by the **activity** instance) to the **FormFlow** instance that is handling the entire conversation (along with its corresponding validations).

This separation of concerns is, in my opinion, the epitome of design excellence from the Bot Framework team. It makes the code readable and easy to manage, and it allows the code base to grow while being nicely organized, which makes the creation of any bot possible.

Running our bot

Those were the bot's main code parts! A lot of things are going on there. Creating a bot is not a trivial task, and there are a few complexities associated with the process, such as handling its very asynchronous nature and also managing state. The Bot Framework makes it easier to handle both aspects by allowing your code to scale.

With our bot ready, it's now time to give it a whirl and see its nice results. I tested the bot quite extensively while writing this e-book, in part because I also wanted to make a tool that I could use myself. Here, I run a couple of requests to show you what output we get.

I'll publish it as a Skype bot (privately, not listed on the Bot Directory) in order to run a couple of example flight requests. However, I encourage you to try it out with the Bot Channel Emulator first.

We've covered the steps required to register a bot on the developer portal and publish it on Azure App Services, so we'll skip that part here.

I've registered the bot on the developer portal, added the **MicrosoftAppId** and **MicrosoftAppPassword**, published the bot to Azure App Services, and added it as a Skype contact.

Let's see some example interaction.

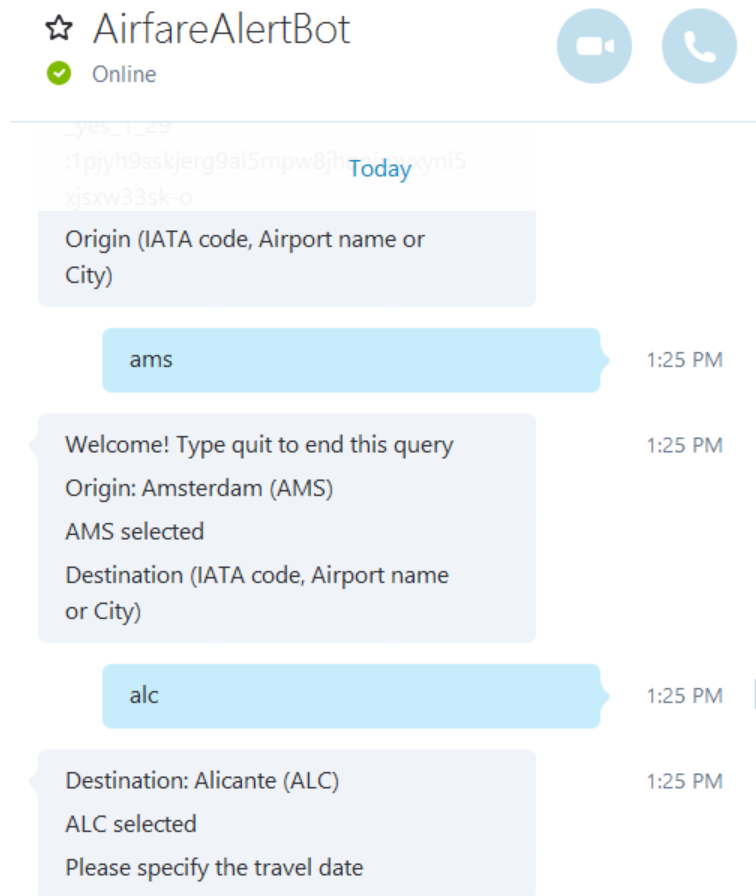


Figure 5.1: Running Our Bot—Gathering Data (Screenshot 1)

The first thing I do is greet the bot by saying “hi” (actually, any word does the trick at this point).

The bot replies with a greeting and asks for some valid data, such as the point of origin. I reply with the name of my departure city—in this case, Amsterdam (IATA code: AMS).

When the bot has the point of origin confirmed, it requests the point of destination. In this case, I type the IATA code for the airport of my destination—Alicante (IATA code: ALC).

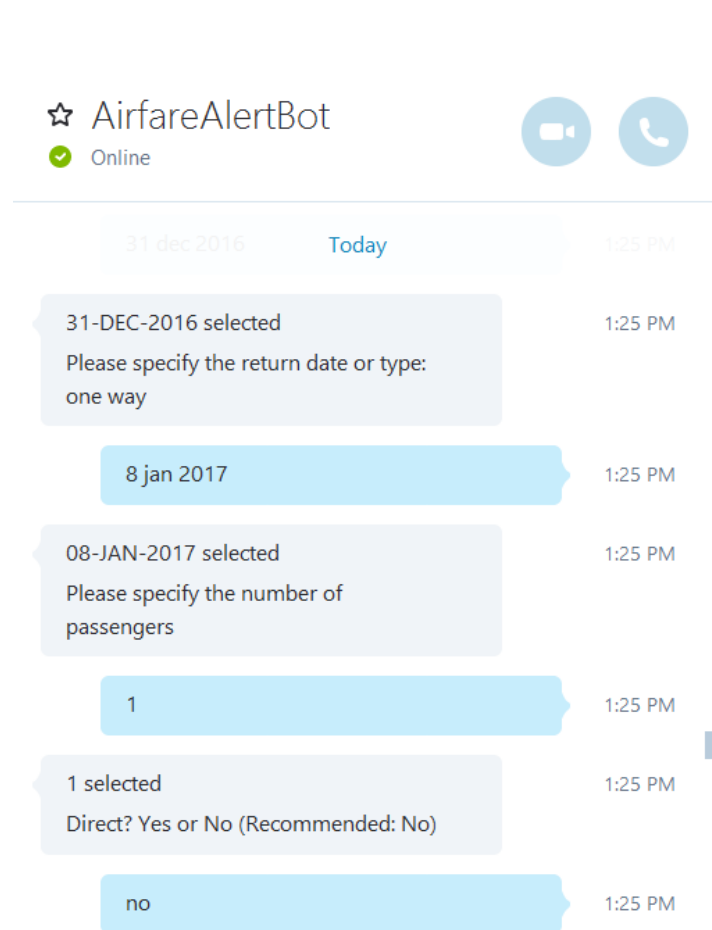


Figure 5.2: Running Our Bot—Gathering Data (Screenshot 2)

With the origin and destination confirmed, the bot will ask for the travel date and a return date (if applicable). In my case, I specify 31-DEC-2016 (as the outbound date) and 08-JAN-2017 (as the inbound date), respectively.

Notice that I entered both travel dates with a slightly different format and the bot formatted them to DD-MMM-YYYY.

When the dates have been confirmed, the bot asks for the number of passengers. In my case, I enter 1.

Following that, the bot asks if the flights are direct. I specify my response as “no.”

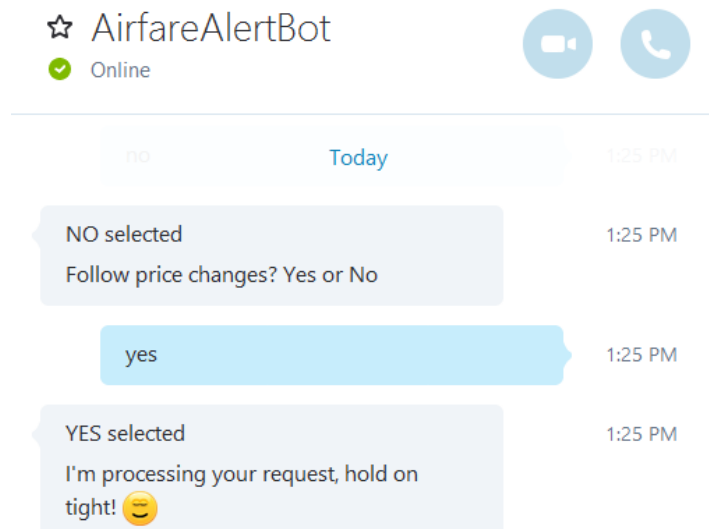


Figure 5.3: Running Our Bot—Gathering Data (Screenshot 3)

Next, the bot asks if I would like to follow this trip for price changes. I reply “yes” to that. With all the details gathered, the bot lets me know the request is being processed.

When the flight request has been processed, I receive the feedback in Figure 5.4.

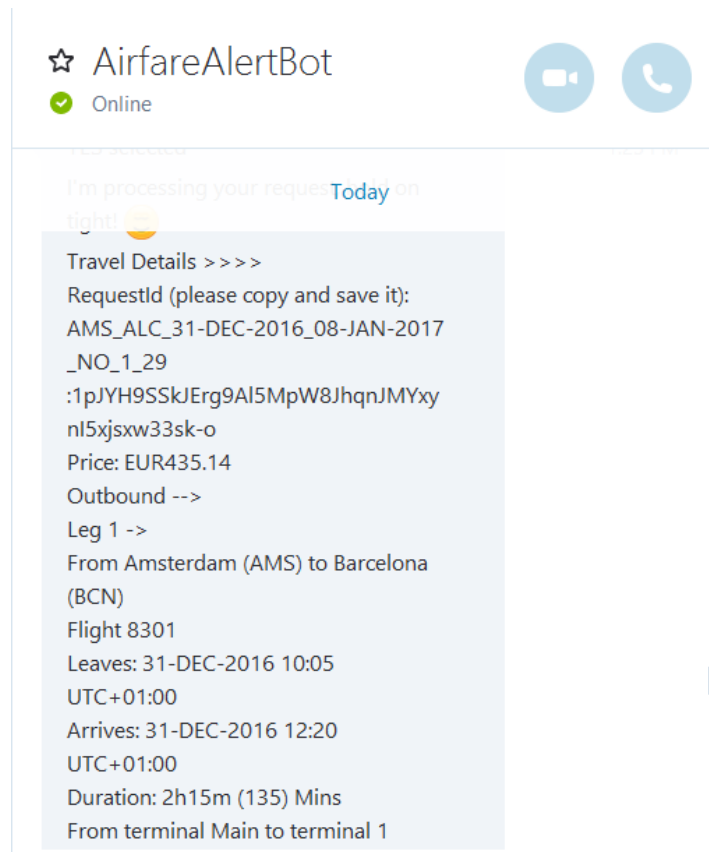


Figure 5.4: Running Our Bot—Results (Screenshot 1)

Because I requested that the flights be followed for price changes, the bot returned a RequestId that I can later use to unfollow it (when I am no longer interested in it).

All of the flight details were provided immediately below the RequestId. We can see further details of the inbound flight in Figure 5.5.

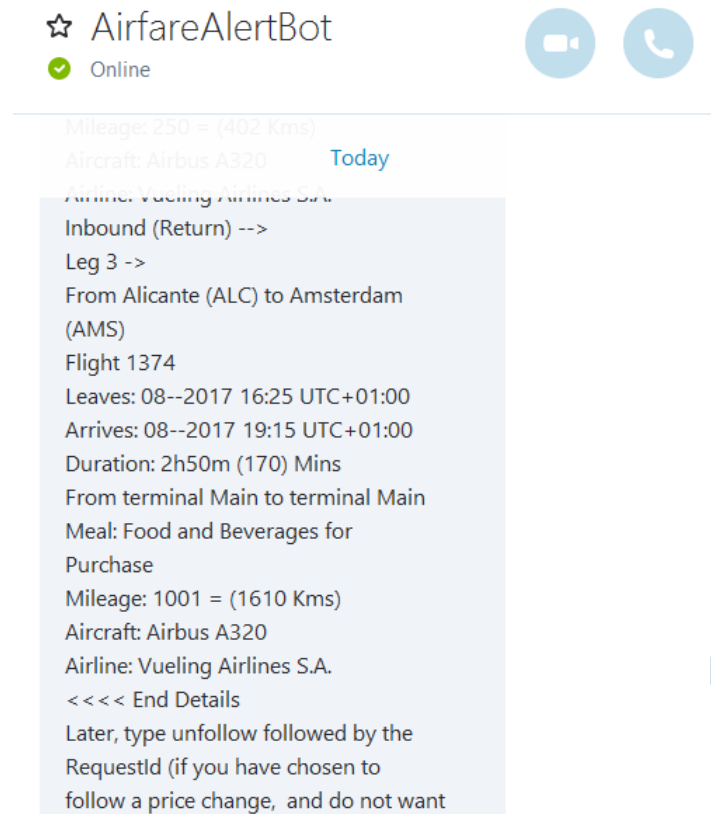


Figure 5.5: Running Our Bot—Results (Screenshot 2)

Awesome! The bot has given me the travel details I wanted. Notice that at the end of a request, if we have chosen to follow a flight, the bot will also add a few extra lines explaining how to later unfollow the request.

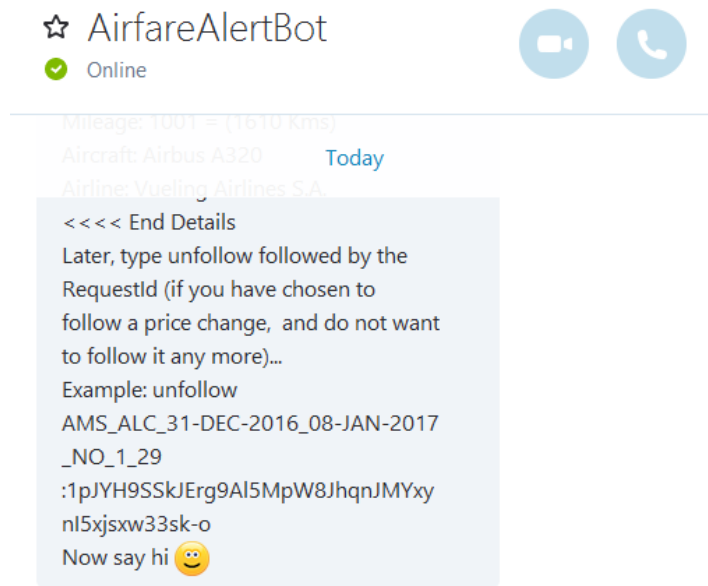


Figure 5.6: Running Our Bot—Results (Screenshot 3)

The request has been processed and the bot expects a greeting (such as “hi” or any other word) to start with a new request (a new conversation). At this point, you can also unfollow a trip (or you can do this later when a new conversation starts).

I decide to unfollow the trip as I am no longer interested in it, and I type the unfollow command followed by the RequestId I received.

The command should follow this convention, with a space between the word “unfollow” and the RequestId:

unfollow RequestId

The request ID is one long string, i.e.:

ams_alc_31-dec-2016_08-jan-2017_yes_1_29:1pjyh9sskjerg9al5mpw8jqnjmyxyni5xjsxw33sk-o

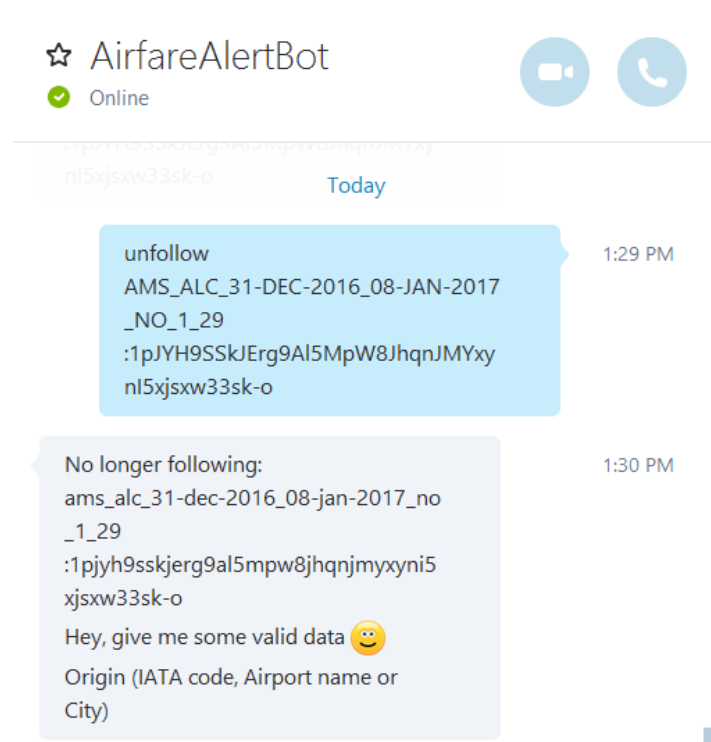


Figure 5.7: Unfollowing a Flight Request

The bot then replies by telling us that the flight has been unfollowed. The original follow request is then removed from Azure Storage.

Because the RequestId is so long, it's not very user-friendly. An enhancement you might add is creating a short RequestId alias such as "ABCDEF."

In order to show you what a price change notification looks like, I manually edited the entry on Azure Storage where this information is stored, and I changed the price. Figure 5.8 shows what the bot returned.

```
<<<<< PRICE CHANGE >>>>>
Travel Details >>>>
Price: EUR435.14
Outbound -->
Leg 1 ->
From Amsterdam (AMS) to Barcelona
(BCN)
Flight 8301
Leaves: 31-DEC-2016 10:05
UTC+01:00
Arrives: 31-DEC-2016 12:20
UTC+01:00
Duration: 2h15m (135) Mins
From terminal Main to terminal 1
Meal: Food and Beverages for
Purchase
Mileage: 770 = (1239 Kms)
Aircraft: Airbus A320
```

Figure 5.8: A Price Change Update Message

Wrapping up

The QPX Express API's results are ordered by prices, so you always get the best possible price for any given route. That's very cool.

However, be sure to keep in mind that sites such as Google Flights use the full-blown QPX API (rather than QPX Express), which means you might get a slightly better price from Google Flights. However, during my tests, I was blown away with the results from the QPX Express API—they almost always matched the Google Flights results. So, it is a very accurate API.

Regarding the AirfareAlertBot source code, you might use it for tracking your own flights, and you are welcome to make any improvements. If you do so, I'd love to hear from you. However, you cannot use this code commercially, and you are not allowed to modify this code and create a modified bot application for profit. Usage of this code is solely permitted for personal and educational purposes.

We've covered a lot of ground in this e-book, and it's been one of the most interesting subjects I've ever had the fortune to explore.

Bots are just getting started, and this is a whole new area of software development that has been getting a lot of media attention lately. Some companies have started to openly embrace it, either by creating their own bots or by providing developers a framework for creating them.

Of the various bot libraries and frameworks I've had the chance to put my hands on, the Microsoft Bot Framework is my favorite. The set of APIs provided out-of-the-box, and the clean code it allows you to write, are both absolutely wonderful advantages that this framework offers.

But that's not all. There are other components within the Microsoft stack that have largely come out of Microsoft Research, such as [Cognitive Services](#), [QnA Maker](#), and [LUIS](#), that can take bot creation to the next level, adding further capabilities such as natural language processing.

If bots get you excited, I definitely recommend that you explore these other services, and why not expand the concepts presented here and improve on them?

For me, learning more about bots has been a lovely experience, even though it was incredibly challenging for me at some points.

I hope you have enjoyed reading this e-book as much as I have loved writing it. With a desire that this information will inspire any further interests in bots you might have, thanks so much for following along. Maybe we can follow up in the future with more bot fun.

Cheers, and happy bot hacking! All the best, Ed.