AZURE BOT SERVICE

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Azure Bot Service Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-227-0

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
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 is a consultant on business process automation and a software developer focused on customer success.

He likes technology and enjoys hacking, learning, playing soccer, running, traveling, and being around his family.

Ed is available at https://edfreitas.me.

Acknowledgments

A huge thank you to the fantastic Syncfusion team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Graham High from Syncfusion and James McCaffrey from Microsoft Research. Thank you all.

I dedicate this book to *Chelin* and *Puntico*—may both your journeys be blessed.

Introduction

Microsoft describes Azure Bot Service as a comprehensive development environment that runs on Microsoft Azure, created for designing and building enterprise-grade, conversational AI.

Azure Bot Service allows you to keep control of your data and simultaneously build multilingual conversational bots for different business scenarios, such as customer support, employee productivity, and sales.

Azure Bot Service delivers a comprehensive development experience by providing a visual canvas called Bot Framework Composer, based on an extensible open-source tool set, providing world-class conversational AI with high-quality natural language, speech, and other Cognitive Services capabilities.

When using Azure Bot Service, you keep control of your data while securely connecting to popular channels such as Skype, Microsoft Teams, Messenger, telephony, and many others.

One of the critical characteristics of Azure Bot Service is that it is easy to get started. Furthermore, it comes with many prebuilt dialogs, components, and language models that empower you to create sophisticated conversational designs that include interruption handling, context switching, and cancellations in different languages and formats.

Azure Bot Service includes enterprise-grade security, high availability, compliance, and manageability backed by Azure's core services by being part of Azure.

Another aspect of Azure Bot Service is that after you create a conversational bot, you will deploy it to multiple channels with minimal or no changes, enabling your organization to have a real-world, everyday platform experience.

Azure Bot Service is an exciting technology that allows you to create a bot with little or no code. At the same time, you could create a bot using code with some of the most popular programming languages, such as JavaScript, Python, and C#.

Throughout this book, we will take a low-code/no-code approach to develop bots with the Azure Bot Service, explore some of the critical features of Bot Framework Composer, and see how to deploy to Azure and some channels.

So, without further ado, let's explore what this promising technology has to offer.

Chapter 1 Getting Started

Overview

As is the case with other frameworks, services, and technologies, before we can use them, we need to go through a process where we sign up for the services and install the required tools. That's what we are going to cover throughout this chapter.

Bot Framework Composer intro

The Bot Framework Composer is an open-source integrated development environment (IDE) built on top of the Bot Framework SDK, which provides an extensible SDK and tools to build, test, deploy, and manage intelligent bots.

In contrast, the Bot Framework Composer provides a robust visual authoring canvas enabling dialogs, language-understanding models, QnAMaker knowledge bases, and language generation responses used to create conversational bots.

Composer (the term I'll be using going forward) is a desktop application for Windows, macOS, and Linux.

Installing Node.js

Before installing Composer, you need to have Node.js and npm installed—this is a mandatory requirement for the use of Composer.

I'm using a Windows machine, so the following steps are specific to Windows. Let's get Node.js installed.

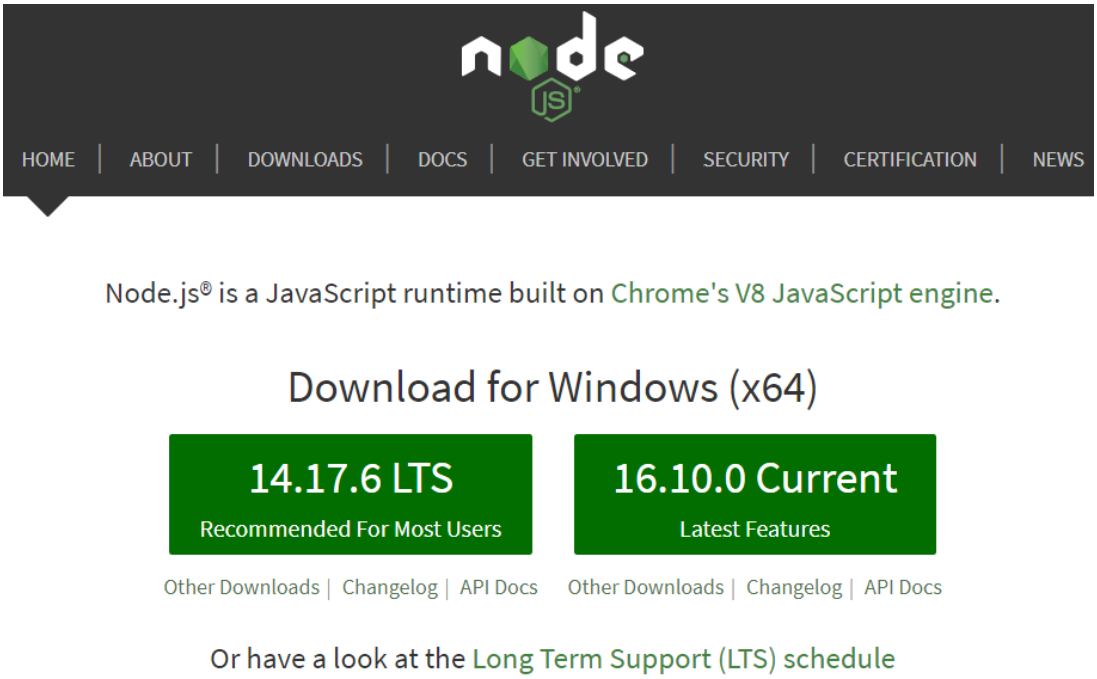

Figure 1-a: Node.js Website

Click the button on the left to install the the Long Term Support (LTS) version of Node.js, which is recommended for most users.

If you already have a version of Node.js installed on your system that's older than the suggested LTS version downloadable from the website, download the current and most up-to-date version.

Once the Node.js installer has downloaded, execute it, and you should see a screen similar to the following one.

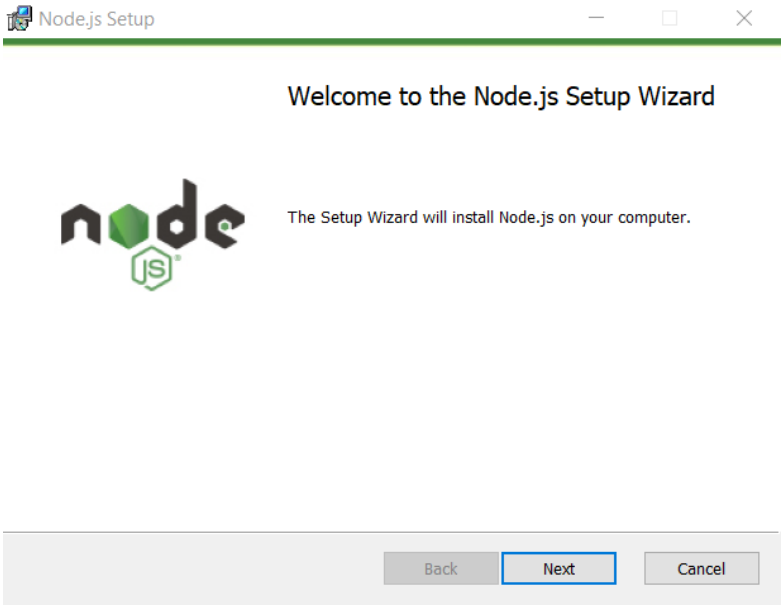

Figure 2-b: Node.js Installer – Welcome Screen

To continue with the process, click **Next**. You should see a screen similar to the following one.

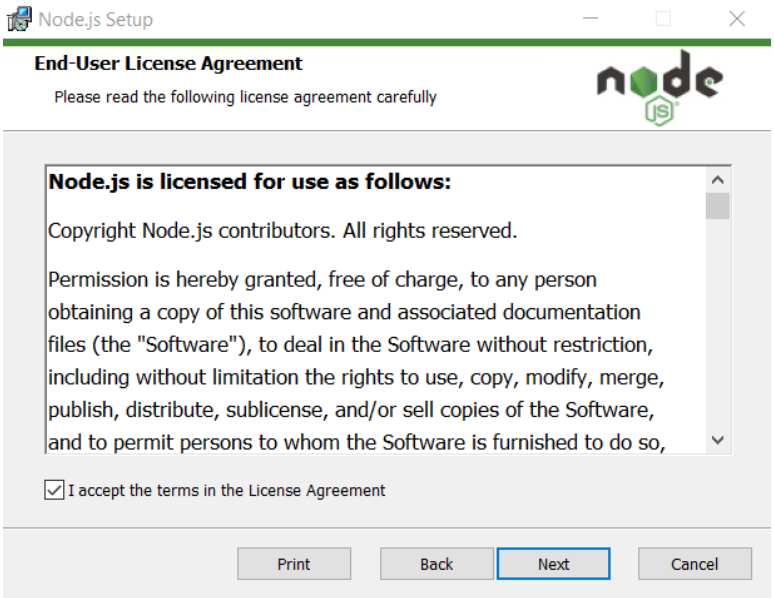

Figure 3-c: Node.js Installer – End-User License Agreement Screen

To continue, select the **I accept the terms in the License Agreement** option and then click **Next**. After that, you should see the following.

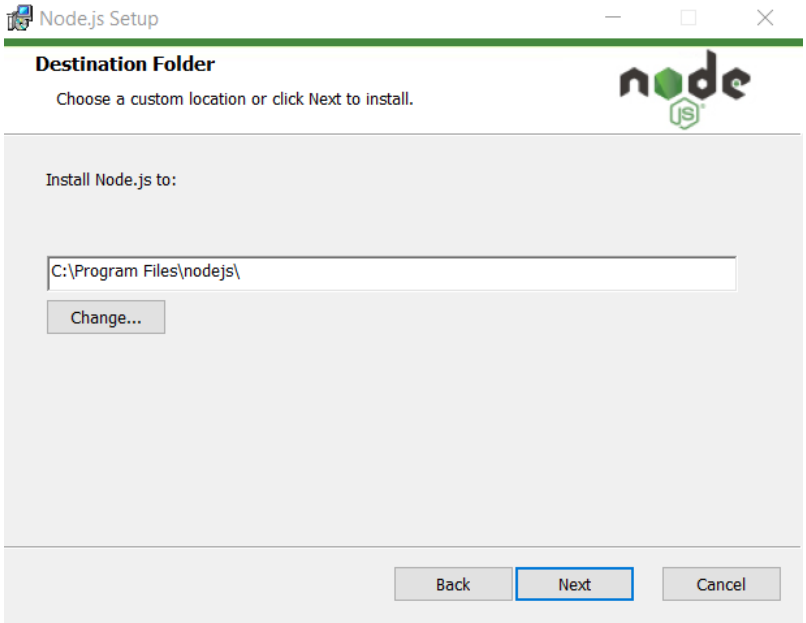

Figure 4-d: Node.js Installer – Destination Folder Screen

I will leave the default installation folder, but you are free to change it to another location. Click **Next** to continue.

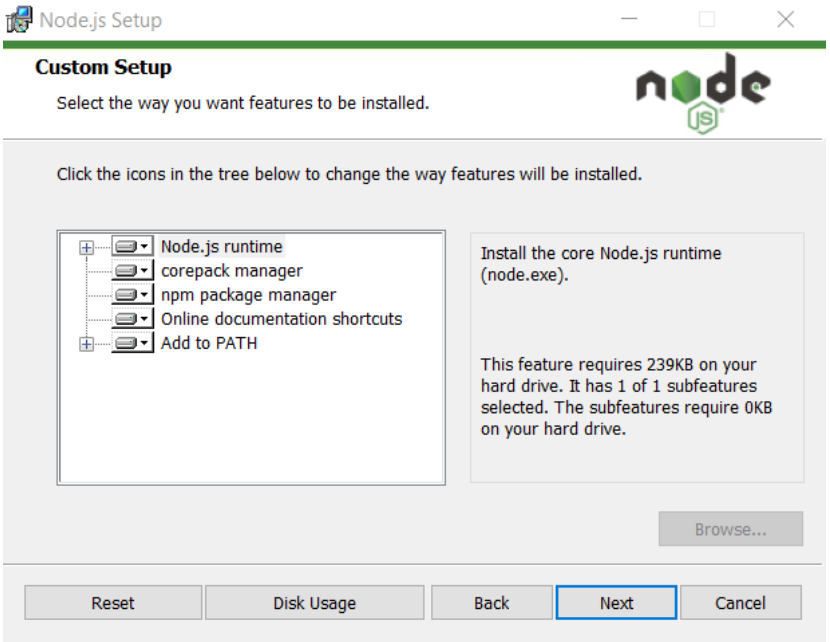

Figure 5-e: Node.js Installer – Custom Setup Screen

After that, you should see the following.

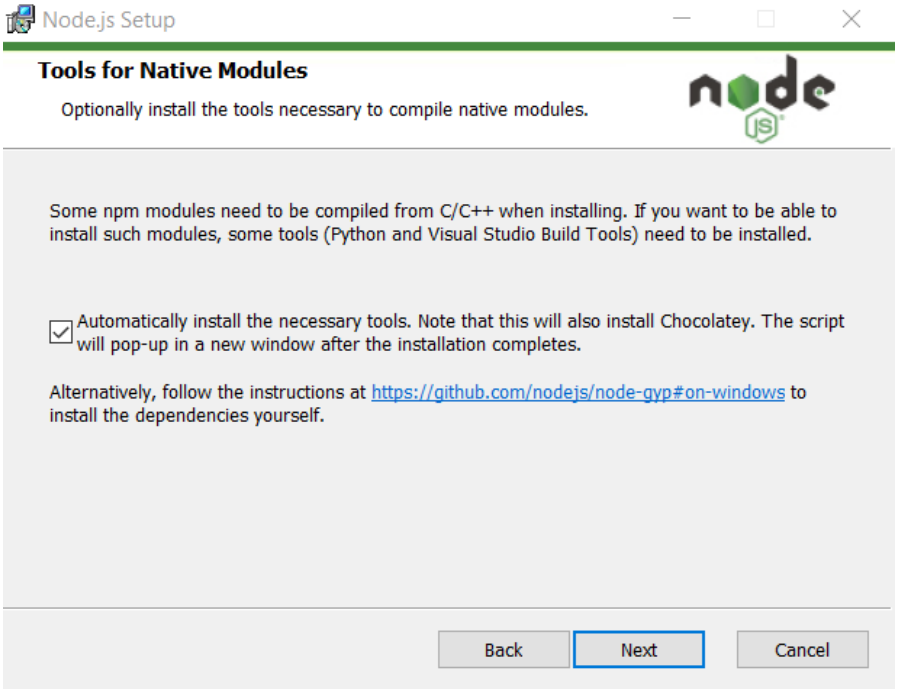

Figure 6-f: Node.js Installer – Tools for Native Modules

Here we are asked to automatically install the necessary tools that some npm modules might require for compilation. There's no harm in having these tools installed, so I recommend clicking the option to install them. Then, click **Next** to continue.

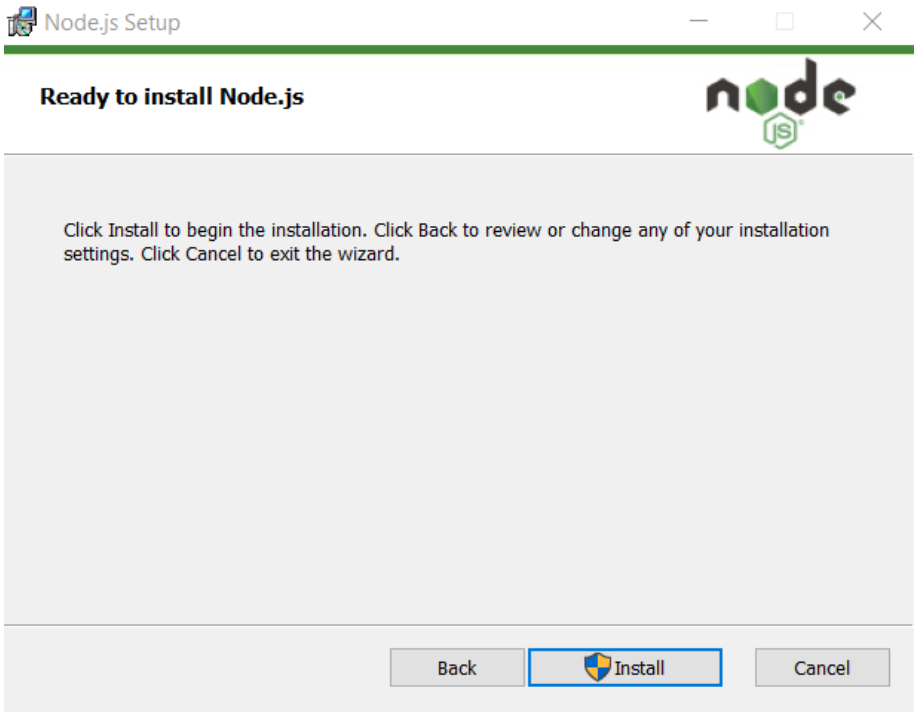

Figure 7-g: Node.js Installer – Ready to install Node.js Screen

Finally, we are ready to install Node.js—click the **Install** button to begin the installation process. You'll see the installation taking place, and once it's complete, you'll see a screen similar to the following one.

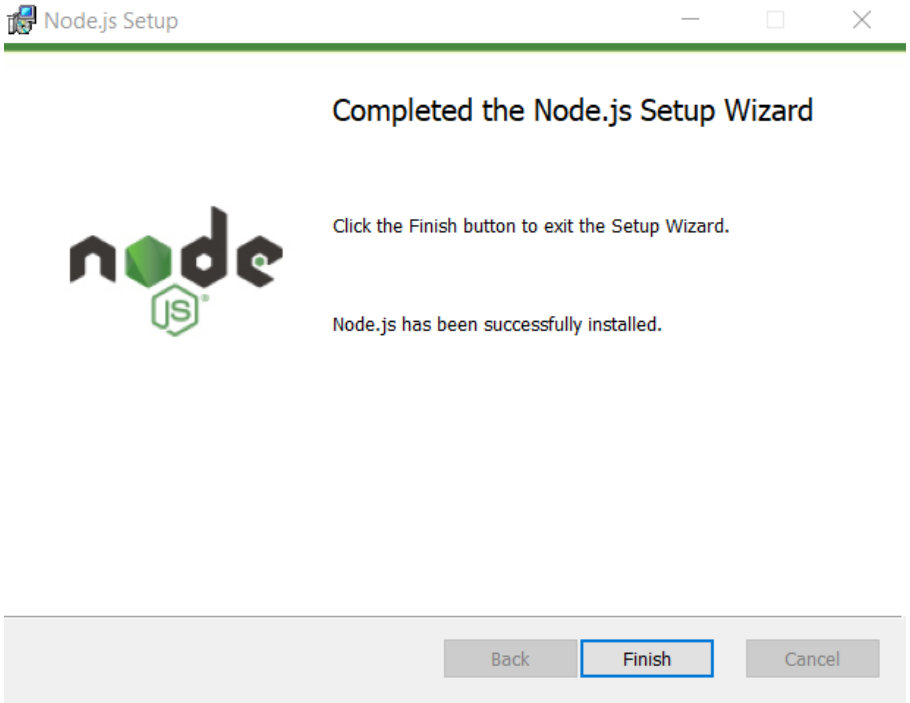

Figure 8-h: Node.js Installer – Completed the Node.js Setup Wizard Screen

Awesome—click **Finish** to close the installer, and Node.js should be ready and good to go.

Installing .NET Core SDK

For C# template support, Composer requires having .NET Core SDK installed. In that case, you'll need to install .NET Core SDK 3.1 or later if you plan to build bots with C#.

Installing Composer

I'm using Windows, so I'm going to use the Composer installer for Windows. Here are the installers for macOS and Linux. Composer is an open-source tool hosted on GitHub.

Once you've downloaded Composer, execute the installer—this will display a screen similar to the following one.

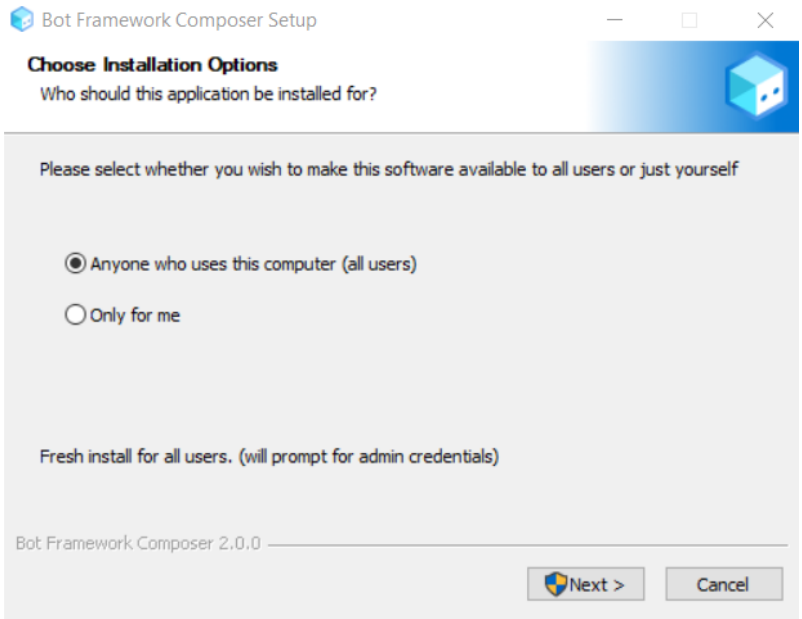

Figure 9-i: Composer Installer – Choose Installation Options Screen

You may choose to install the software by selecting **Anyone who uses this computer (all users)** or selecting **Only for me**. I usually pick and recommend the first option; however, feel free to choose the second option. Then, click **Next** to continue.

Next, we need to choose the **Destination folder**. I usually suggest leaving the default installation path.

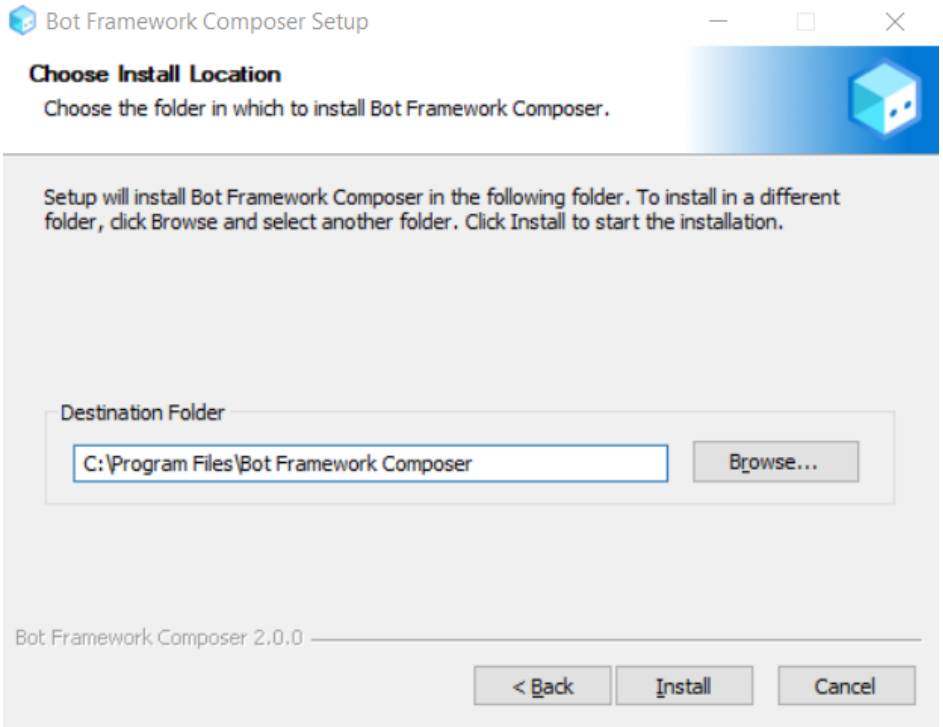

Figure 10-j: Composer Installer – Choose Install Location Screen

You may specify a different installation path if you wish. Once you're done, click **Install**—the installation process will start, and in most cases, will take less than a minute to complete. Before finalizing the setup process, you can choose to run Composer.

In some cases, there are available updates that you can choose to install immediately or later.

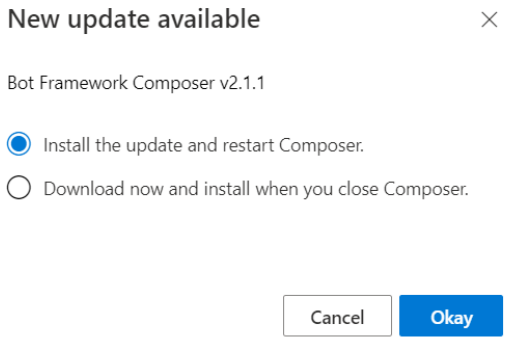

Figure 11-k: Composer – New update available

I recommend installing the latest updates, which I will do by selecting the first option and then clicking **Okay**. You can choose the second option or click **Cancel**.

Composer will restart after installing any updates, and the installer with the latest updates will execute again.

After installing the updates, you can choose to run Composer before closing the installer. The main Composer screen looks as follows.

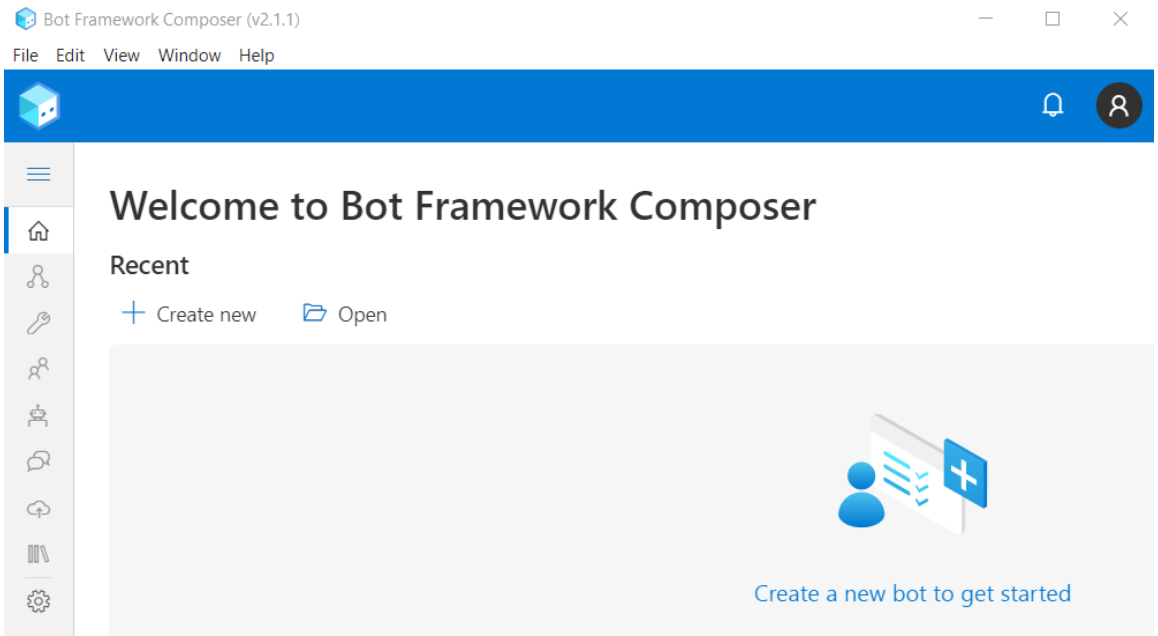

Figure 12-I: Composer – Main Screen

Excellent—we are now ready to start developing bots with Composer, which we will do in the next chapter.

Chapter 2 Composer Bot Basics

Overview

In the preceding chapter, we explored how to install Node.js and Composer and saw how straightforward that process was.

In this chapter, we will use Composer to create a zip code bot using a third-party API, which will give us some insights about zip code locations.

You might find some similarities between the steps and scenarios involved and those available in the Composer documentation quickstart and tutorials (to maintain consistency with the official docs); however, we'll dive deeper into details than those resources do.

This will be an exciting and fun project—so without further ado, let's get started.

Composer UI

First, let's get acquainted with the Composer UI to understand which components make up this product's user interface.

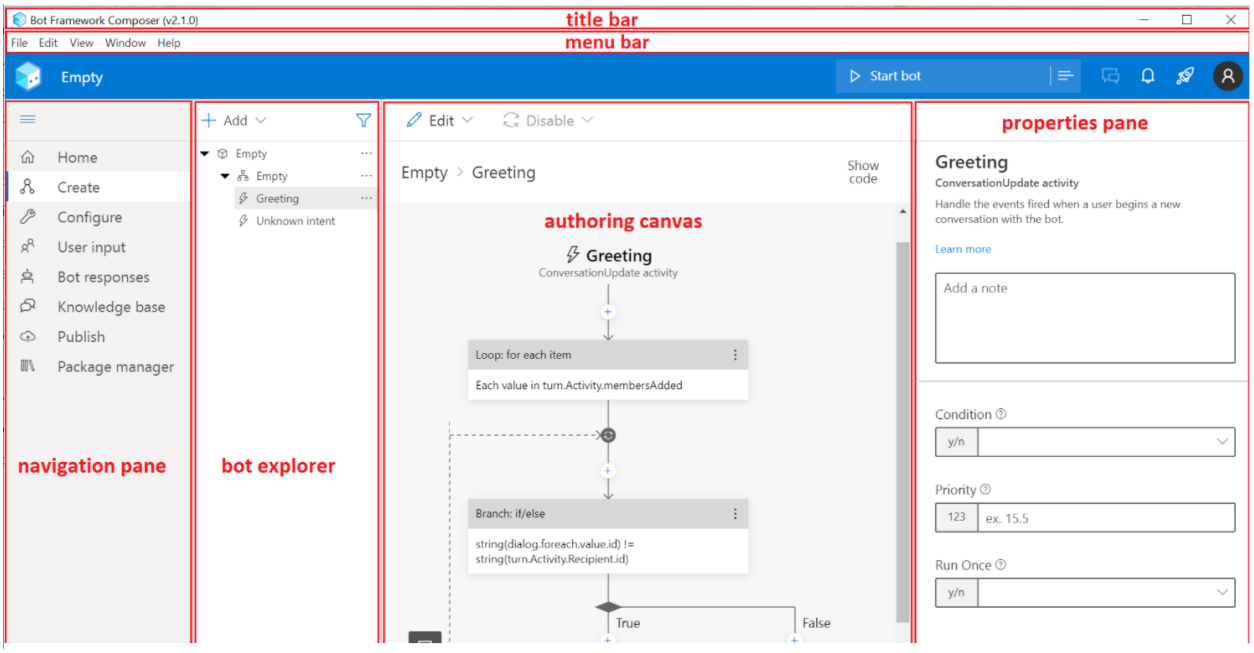

Figure 2-a: Composer's Main Screen (Source: Microsoft)

The Composer UI components highlighted in the preceding figure are the ones that we will be primarily using throughout this book. These elements make up the following four main sections, most commonly known as *panes*:

- The **navigation pane**: This is the main section that contains Composer's main options and features.
- The **bot explorer**: This section displays the elements that make up your bot projects, such as dialogs, triggers, intents, and other bot-specific items.
- The **authoring canvas**: This section is where the bot's logic resides, and it contains all the actions associated with a selected trigger.
- The **properties pane**: This area is where you can set the properties for specific actions, such as sending or receiving an external request, or requesting input from the user.

There is also a **title bar** containing the application's name and a **menu bar** that you can use to start the execution of a bot, access your account settings, or view application alerts.

As you can see, the UI is self-descriptive and easy to navigate.

Zipcodebase

To create our zip code bot, we first need to sign up for an external third-party API called Zipcodebase that will give us access to a database of zip code information worldwide.

Note: We'll be using the Zipcodebase service; however, this is independent of Composer, and you could use any other zip code API or service provider.

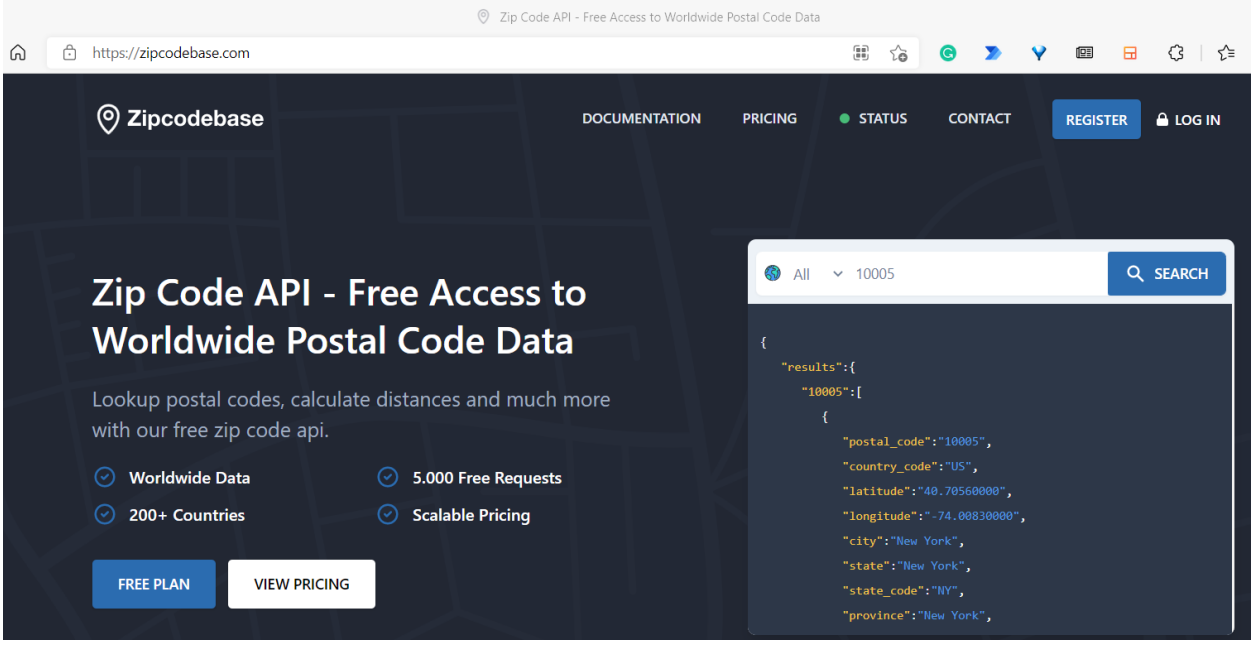

Figure 2-b: Zipcodebase Website

To get started, click either the **FREE PLAN** or the **REGISTER** button—this will take you to a screen similar to the following one.

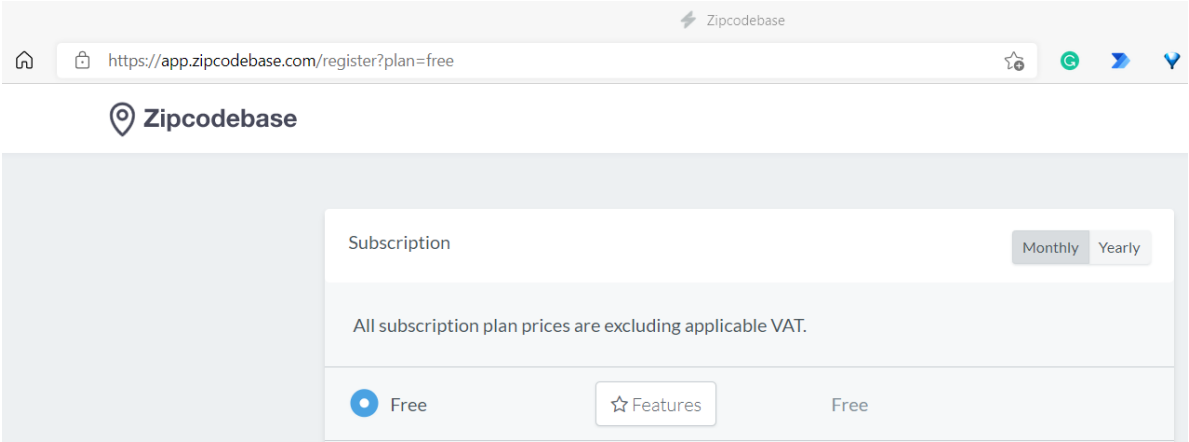

Figure 2-c: Zipcodebase Website – Choosing a Subscription

Make sure the free plan is selected and then scroll to the bottom of the page to fill in the details requested.

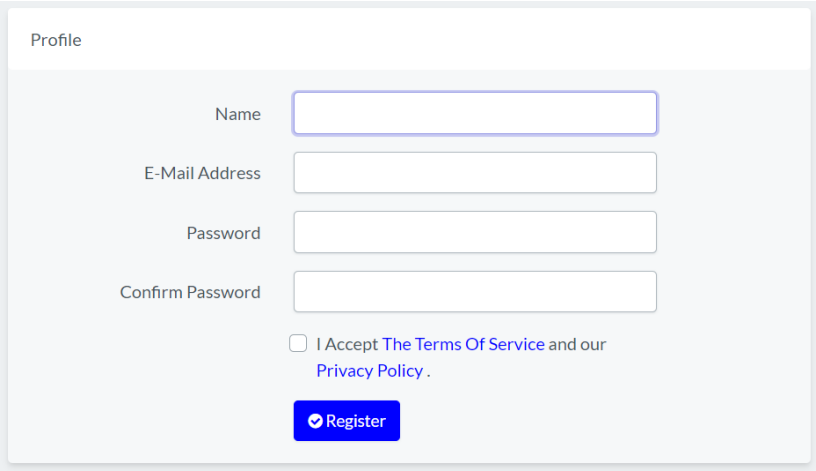

Figure 2-d: Zipcodebase Website – Creating a Profile

After you have registered for the service, you might be prompted to verify your email. Check your email and verify it so you can start to use the service.

After verifying your email address, log in to Zipcodebase to get your API key. You should see the following.

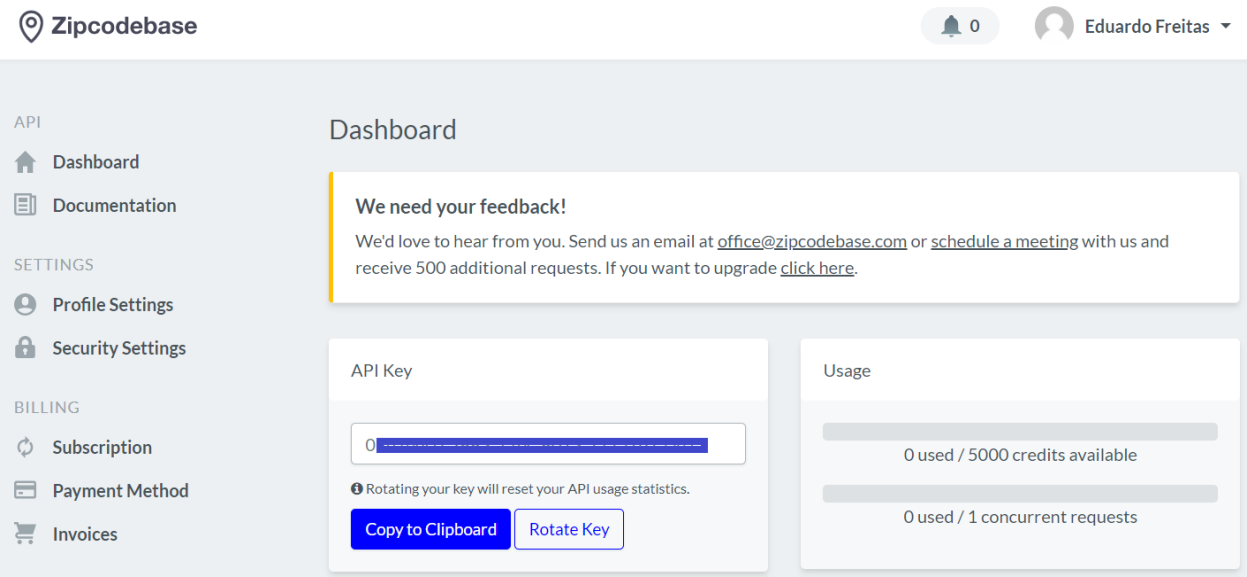

Figure 2-e: Zipcodebase Dashboard – API Key

Make sure you copy the API key to the clipboard.

We are now ready to create an empty bot using Composer.

Creating an empty bot

We are going to develop our zip code bot from scratch. Open Composer and click **+ Create new**.

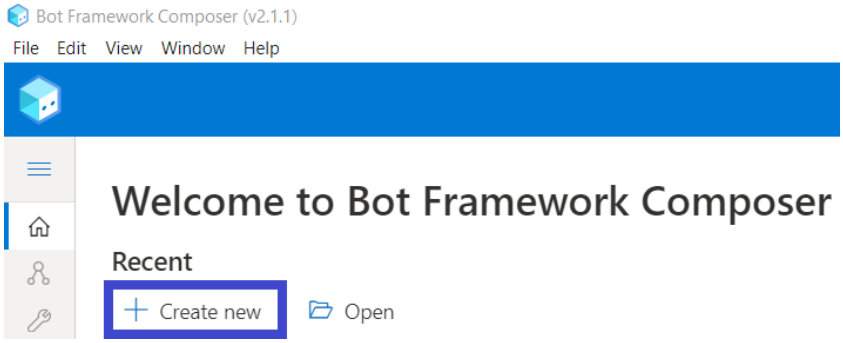

Figure 2-f: Composer Main Screen – Create new Button

A screen similar to the following one will appear. Here you can select the bot template to use.

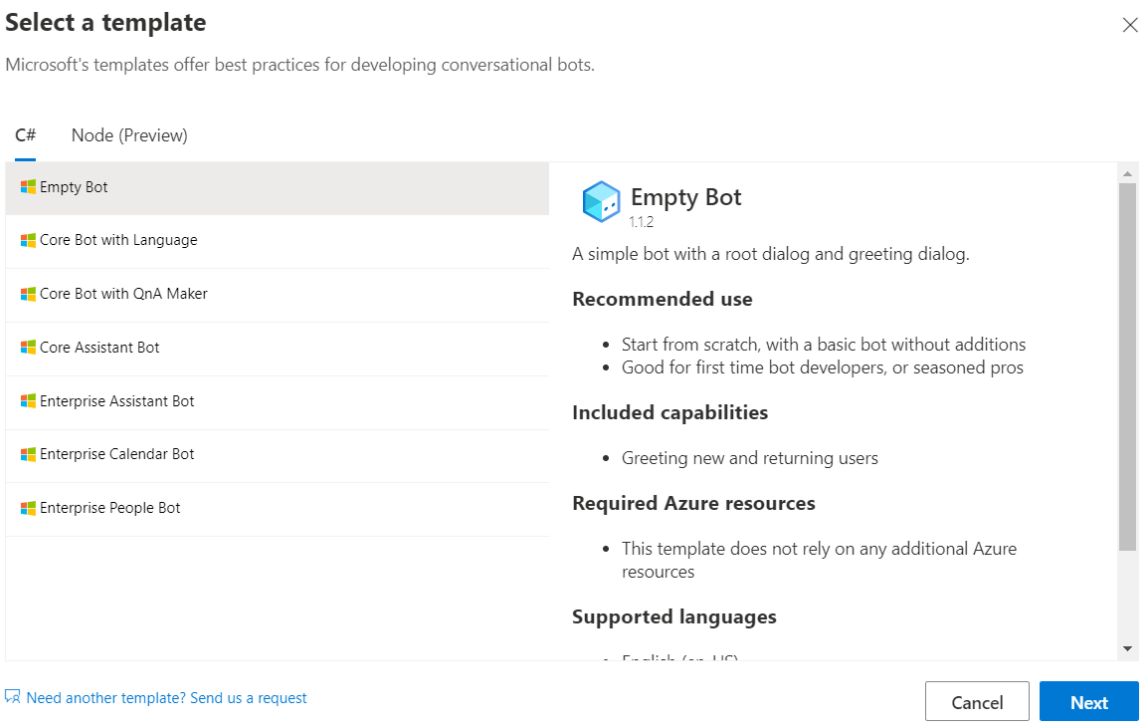

Figure 2-g: Composer – Select a template Screen

Let's select the **Empty Bot** template for **C#** and then click **Next**. After doing that, you will see a screen similar to the following one.

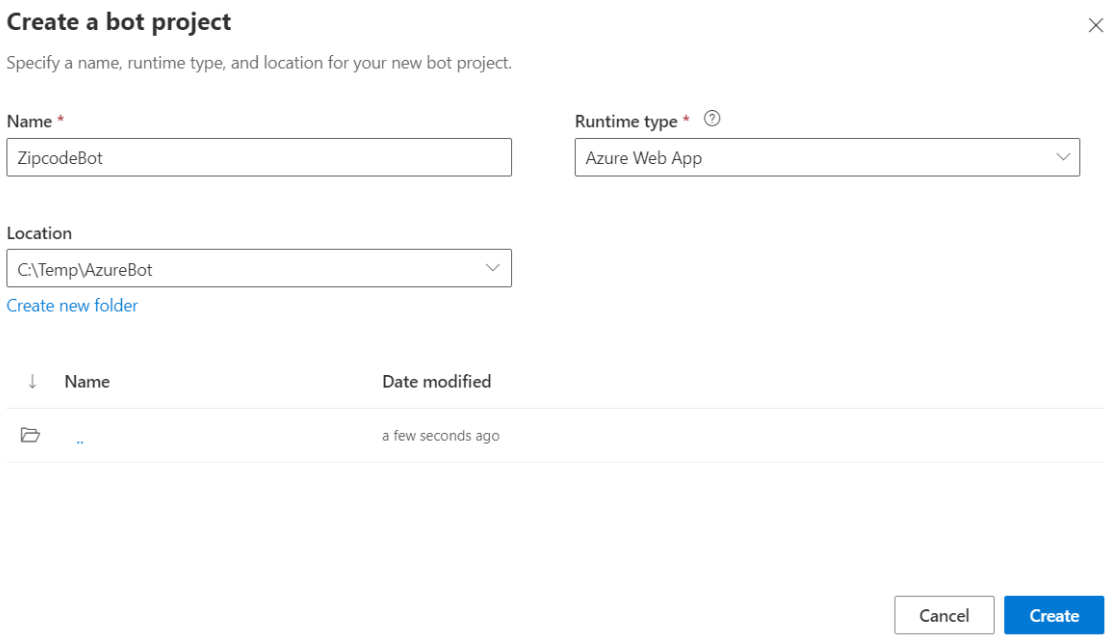

Figure 2-h: Composer – Create a bot project Screen

I've called the project **ZipcodeBot**, set the **Runtime type** to **Azure Web App** (you can also choose **Azure Functions**), and set the **location** to a local folder on my machine (feel free to select a different folder). After you've specified these fields, click **Create**.

Composer will download the bot template, build the runtime, and merge packages—this might take a few seconds or up to a couple of minutes. Afterward, you will see the created bot.

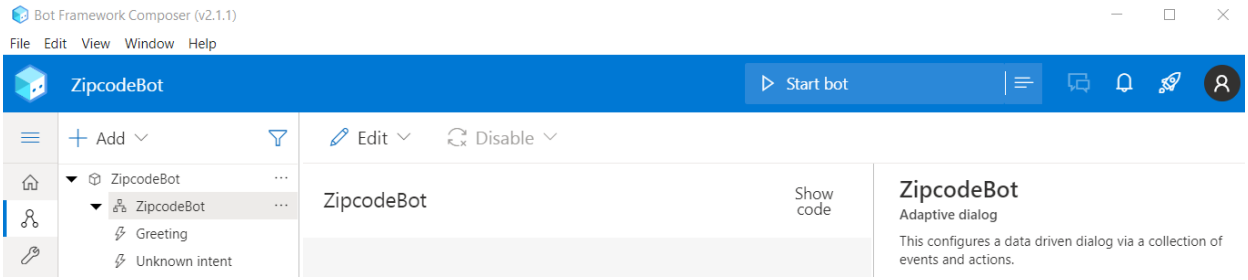

Figure 2-i: Composer – ZipcodeBot Created

Here we can see that we have an empty bot with a dialog called **ZipcodeBot**, and under that, there are two triggers (indicated by lightning icons): one called **Greeting** and another called **Unknown intent**.

The **Greeting** intent executes when the user connects to the bot, sending the user a greeting. On the other hand, the **Unknown intent** runs when the user sends a message, or the bot cannot recognize the user's request. In that case, the bot responds to the user, indicating that it cannot understand the user request.

To start giving our bot some personality, the first thing we need to do is change the **Greeting intent**. To do that, let's click the **Greeting intent** under **ZipcodeBot**, and then click the **Send a response** action.

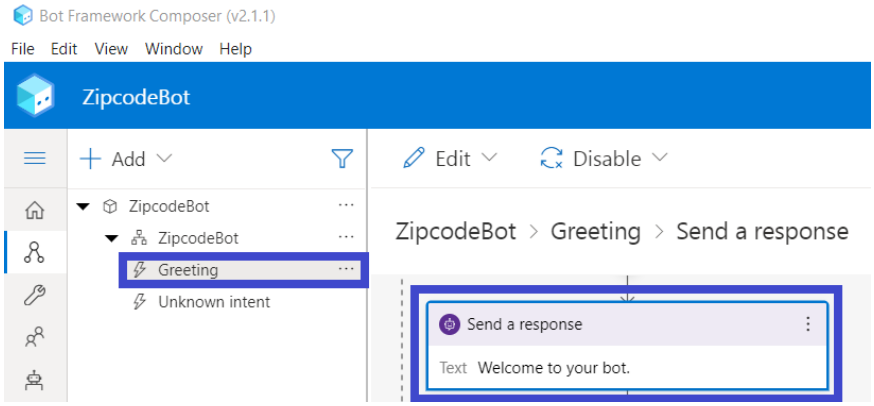

Figure 2-j: Composer – Send a response Action Selected

Then, in the properties pane, find the **Bot responses** section and choose **Welcome to your bot**. After doing that, you'll be able to edit the intended response.

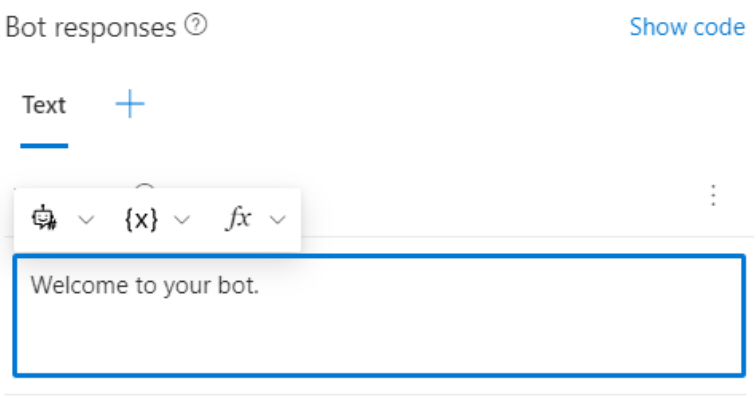

Figure 2-k: Composer – Editing a Bot Response

I'll type the following message: **Welcome to ZipcodeBot. Please type the word 'zip' to start**. However, you can customize this to your taste.

First-time bot execution

Let's run our bot for the first time. To do that, click the **Start bot** button found just below the menu bar.

Figure 2-l: Composer (Start bot Button)

Once the bot has successfully executed, you will see a dialog dropdown similar to the following.

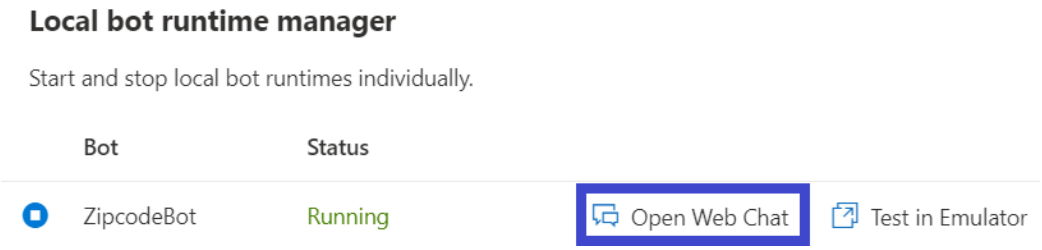

Figure 2-m: Composer – Local bot runtime manager

Next, let's click the **Open Web Chat** option highlighted in Figure 2-m. After doing that, you will see the web chat to interact with the bot.

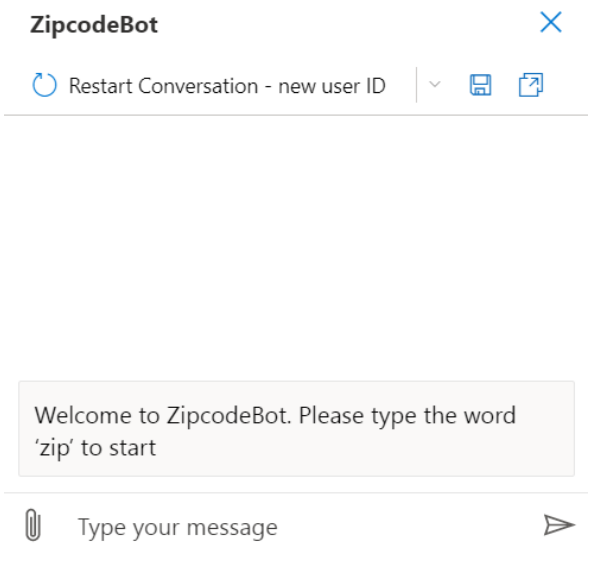

Figure 2-n: Composer – ZipcodeBot Web Chat

To test the bot, type the word **zip**, which will cause the bot to respond with the following message.

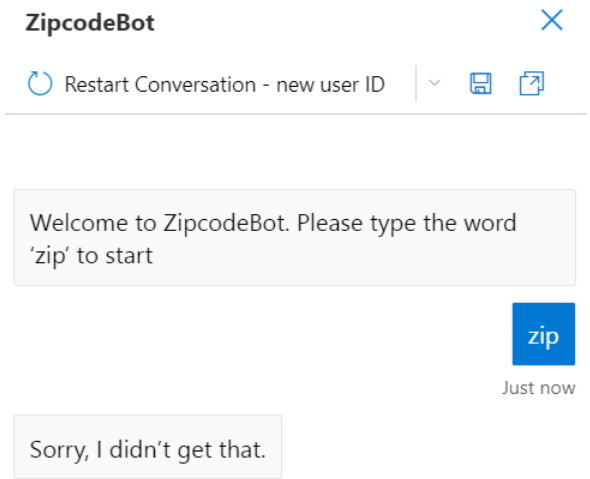

Figure 2-o: Composer – ZipcodeBot Web Chat – Response

As expected, the bot responds that it doesn't understand the intent provided because we haven't programmed this logic yet.

Now that we have executed the bot for the first time, let's stop its execution, which you can do by clicking on the stop icon highlighted in the following figure.

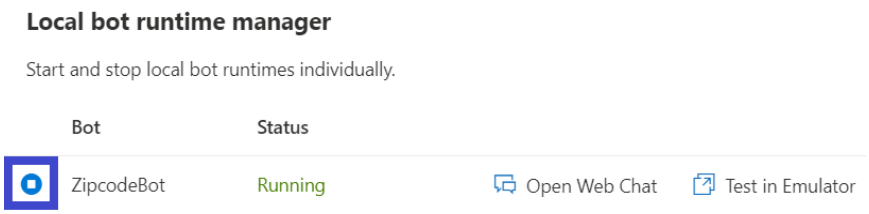

Figure 2-p: Composer – Stop Bot Icon

Clicking that will stop the execution of the bot and make it inactive until it executes again.

Adding a dialog

Bots consist of various components, and some of the most important are dialogs. In other words, most bots are structured as a sequence of dialogs.

A dialog includes specific bot functionality, such as asking the user for a response, sending a reply, or making a request to an API.

Let's create a dialog that can get a zip code from the user. Within Composer's bot explorer pane, select the **ZipcodeBot** top-level element. Next, click the ellipsis (**…**), and then the **+ Add a dialog** menu item.

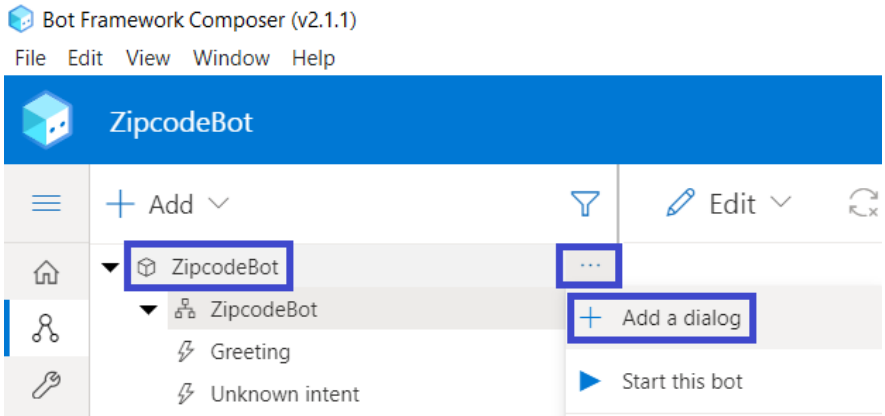

Figure 2-q: Composer – Add a dialog Menu Item

The **Create a dialog** window will appear. Here we can enter a name and description. I'll call this dialog **get_zip**, but you can call it something else.

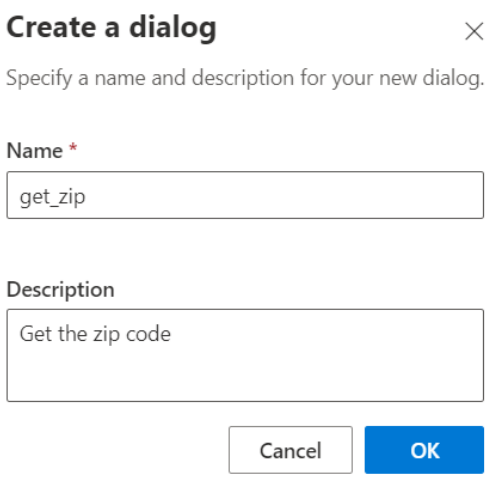

Figure 2-r: Composer – Create a dialog Window

After entering those values, click **OK**. After that, you will see the **get_zip** dialog within the bot explorer pane.

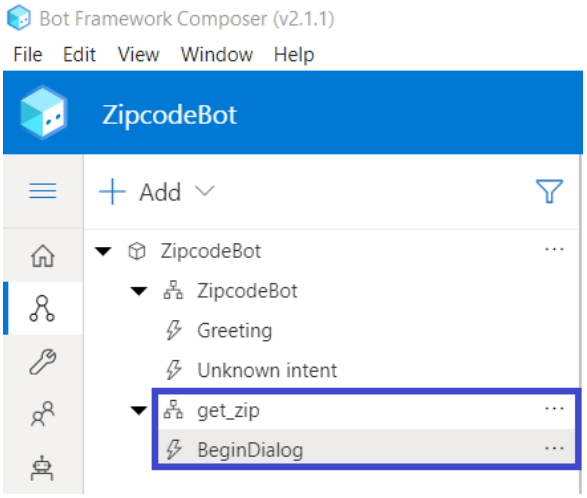

Figure 2-s: Composer – Bot Explorer – get_zip Dialog

To continue, click **BeginDialog**, and then go to the authoring canvas to the right of the bot explorer.

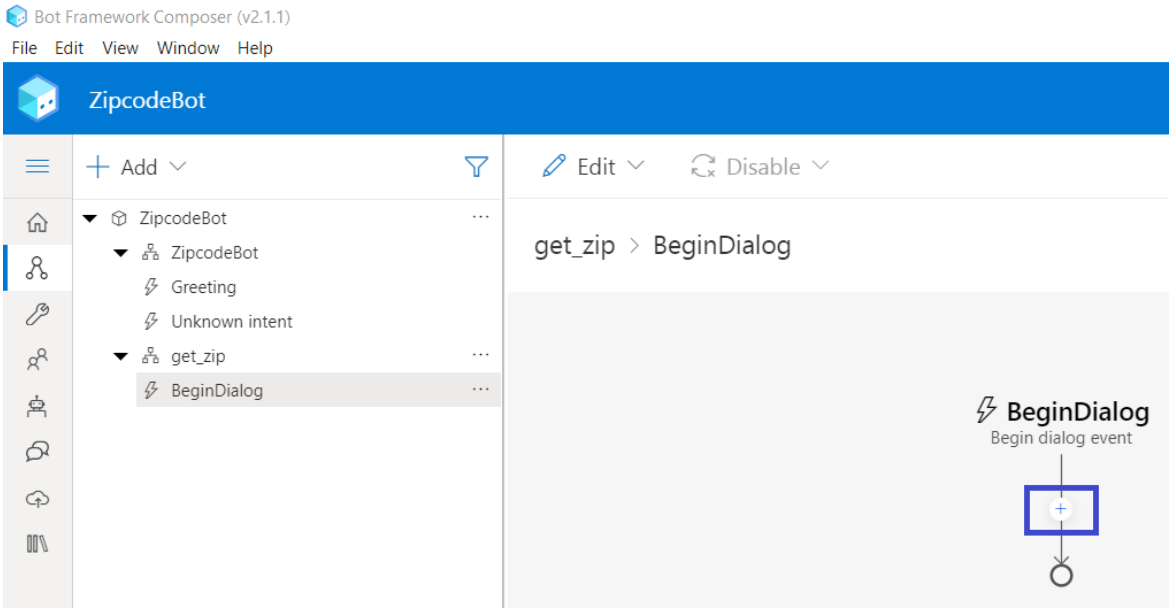

Figure 2-t: Composer – Authoring Canvas – BeginDialog

Under **BeginDialog**, click the **+** button and click on the **Send a response** menu item in the authoring canvas.

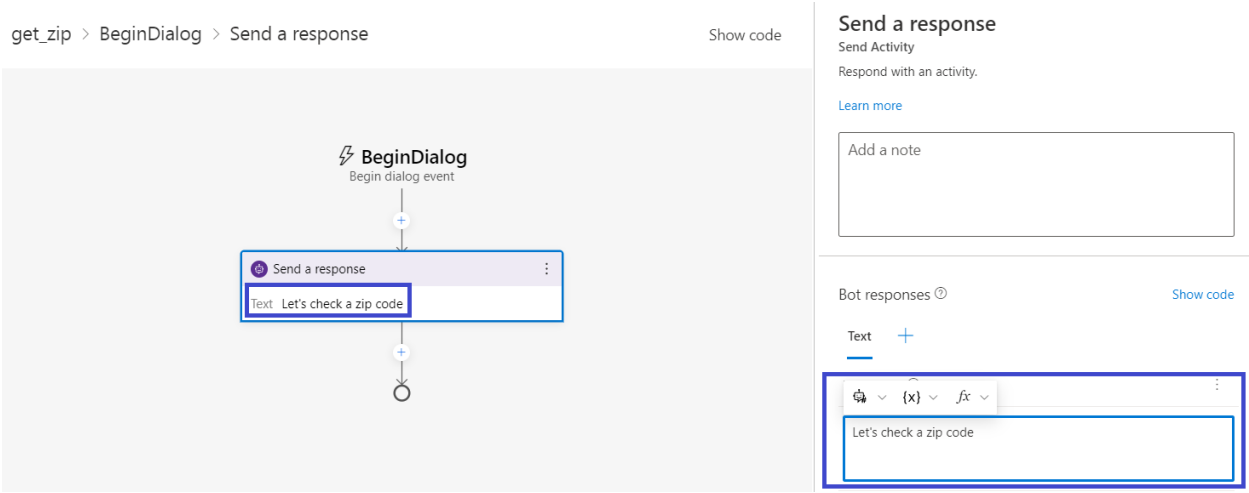

Figure 2-u: Composer – BeginDialog – Send a response

Under **Bot responses**, let's type the following text: **Let's check a zip code**. What we have done is created a dialog called **get_zip**, and this dialog has a trigger called **BeginDialog**, and this trigger has an action called **Send a response**—which we can visualize as follows.

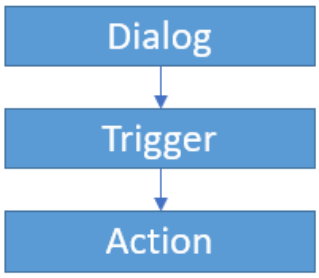

Figure 2-v: Dialog – Trigger – Action

The action, in this case, is the response that is sent to users when the **get_zip** dialog activates.

Tip: To be able to test this new dialog, it is necessary to have a trigger in ZipcodeBot—this will allow you to start the get_zip dialog.

Summary

We have taken the initial steps to create a bot, added some basic functionality, and explored Composer's UI characteristics throughout this chapter.

The cool thing is that nothing we have done and looked at has involved writing any code so far.

In the following chapter, we will continue to expand the bot's functionality by initially executing this dialog from a trigger.

Chapter 3 Expanding the Bot

Overview

Conversational flows within bots are composed of different dialogs, which are connected one to the other.

In the previous chapter, we created the basics of our **ZipcodeBot** and added a new dialog. To be able to use that dialog, we need to invoke it. To do that, we need to start that dialog from a trigger.

So, we need to connect the **get_zip** dialog to the **ZipcodeBot** dialog—the bot's main dialog. To understand this better, let's look at the following figure.

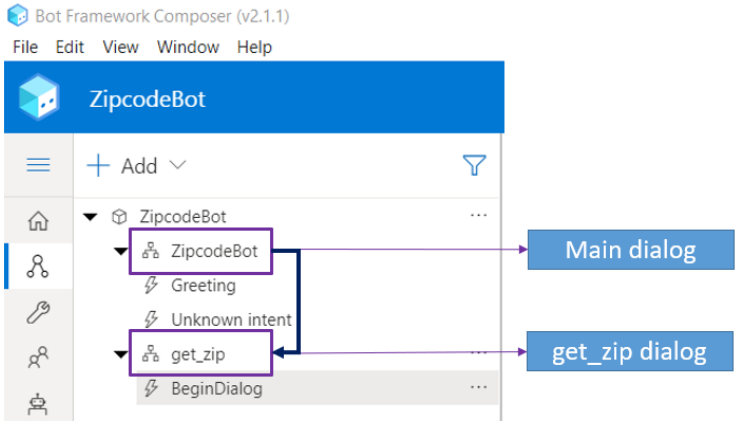

Figure 3-a: ZipcodeBot (Main) Dialog to get_zip Dialog Relationship

Executing the dialog from a trigger

Let's link the **get_zip** dialog to the bot's main dialog—**ZipcodeBot**. To do that, click the main dialog. Under **Recognizer/Dispatch type**, click **Change**—as seen in the properties pane in the following figure.

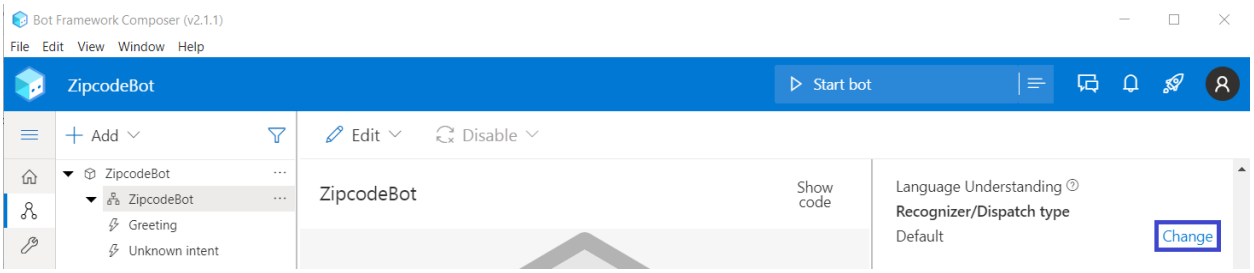

Figure 3-b: Composer – Properties Pane – Recognizer/Dispatch type – Change

A window will appear, which will allow us to choose a recognizer type. Click the **Regular expression** option and then click **Done**.

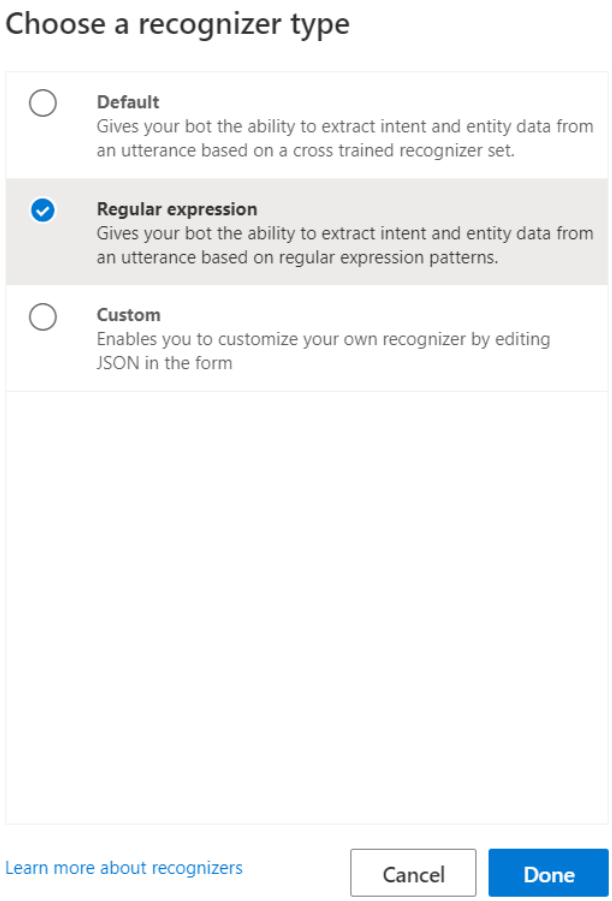

Figure 3-c: Choose a recognizer type Window

Next, click the **ZipcodeBot** dialog, click on the ellipsis (**…**), and click the **+ Add new trigger** menu item.

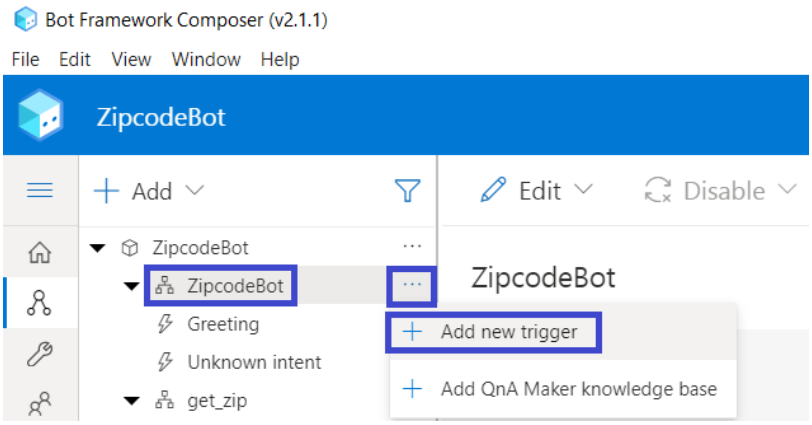

Figure 3-d: Composer – ZipcodeBot – Add new trigger

Once those actions have occurred, an input dialog will appear, requesting the following info: trigger type, trigger name, and regEx pattern.

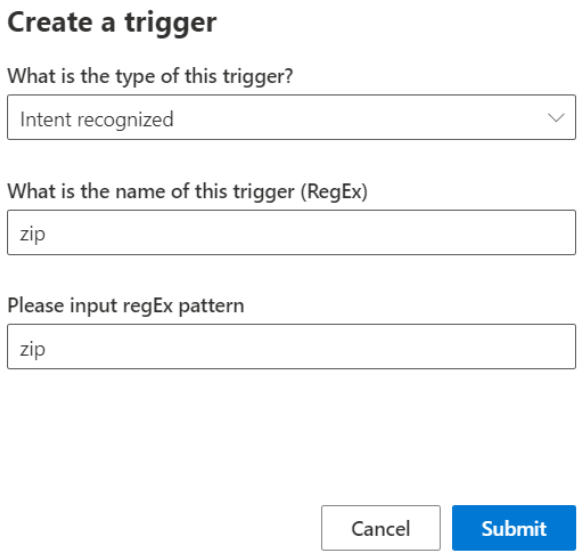

Figure 3-e: Create a trigger Dialog

For the type of trigger, we can leave the default value, which is **Intent recognized**. I will call the name of the trigger **zip** and use **zip** as the value of the regEx pattern. Next, click **Submit**.

The **zip** trigger will appear under **Unknown intent** in bot explorer. We can see that as follows.

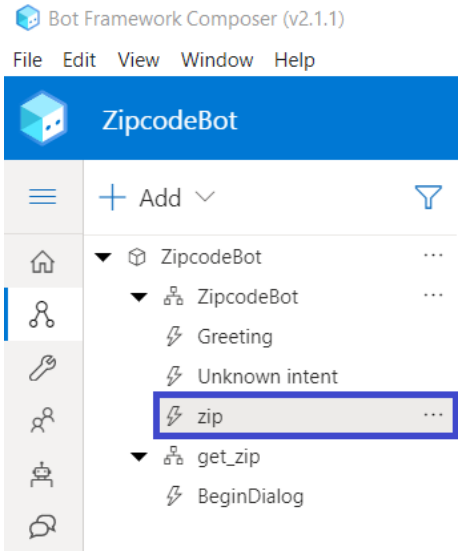

Figure 3-f: Composer – zip Trigger

Let's recap what we've just done. The **zip** trigger instructs the **ZipcodeBot** to look for the word **zip** in any incoming message. To do that, we use regular expressions (also known as regEx).

Next, with the **zip** trigger selected, in the authoring canvas, click **+** > **Dialog management** > **Begin a new dialog**.

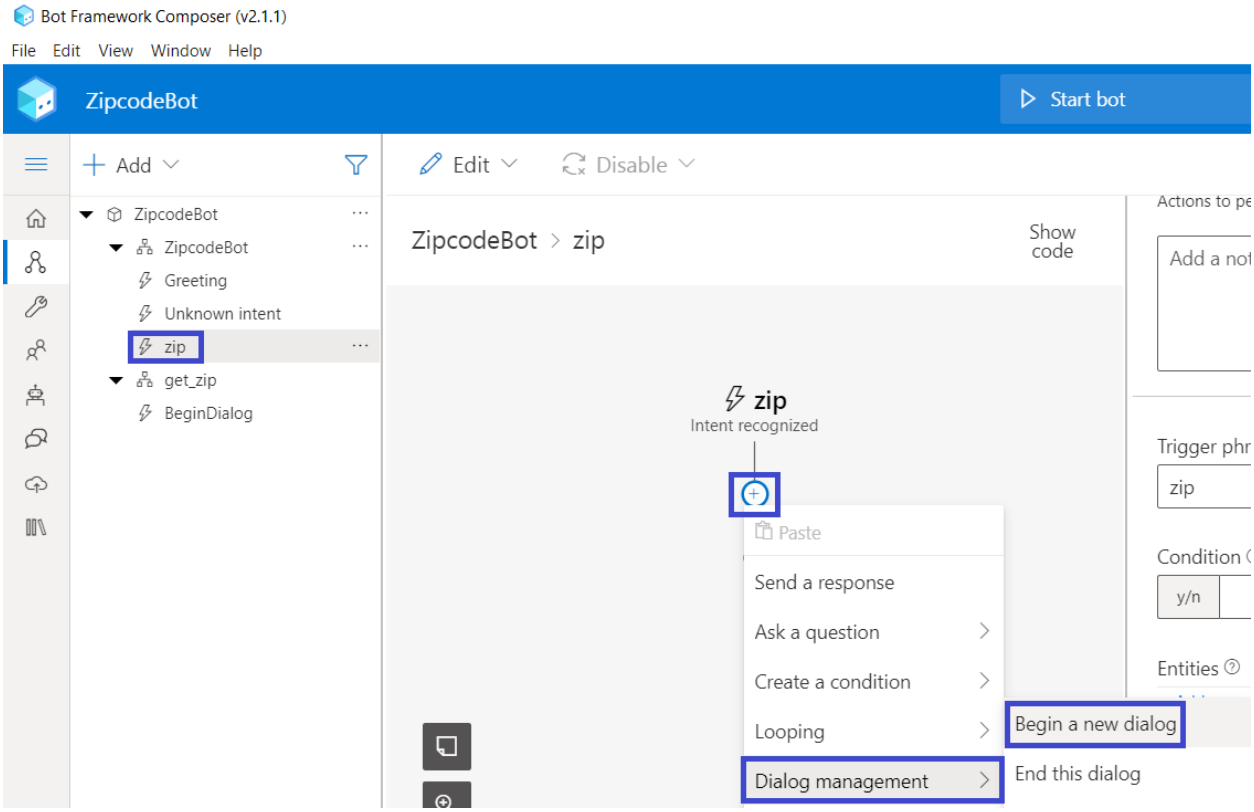

Figure 3-g: Composer – Dialog management – Begin a new dialog

Once the dialog appears, click the **Dialog name** dropdown and choose the **get_zip** option from the **Dialog name** on the properties pane.

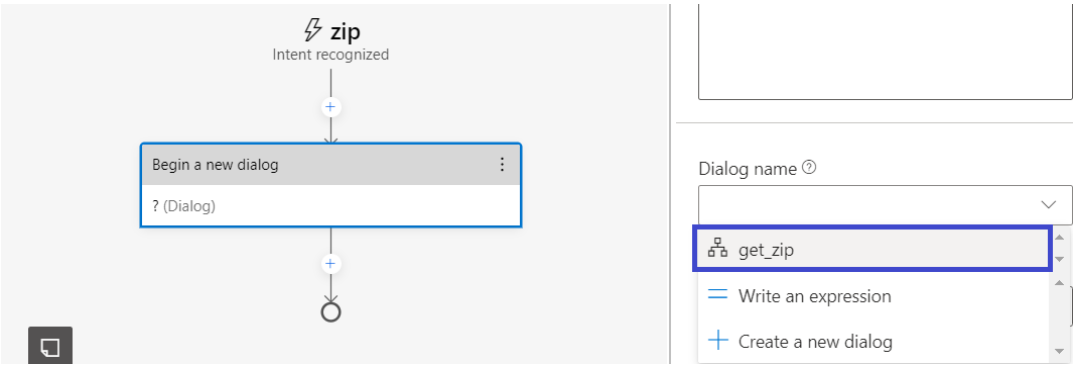

Figure 3-h: Composer – Dialog name – get_zip

We have just created a trigger and indicated that our **ZipcodeBot** could recognize regular expressions, using the word **zip** as the trigger.

Requesting user input

For the **ZipcodeBot** to get the relevant information regarding a zip code, the bot needs to request the user to enter the zip code, and for that, we need to use a **Text input** action.

Under the **get_zip** dialog within bot explorer, select **BeginDialog** and then click **+** under **Begin a new dialog** in the authoring canvas.

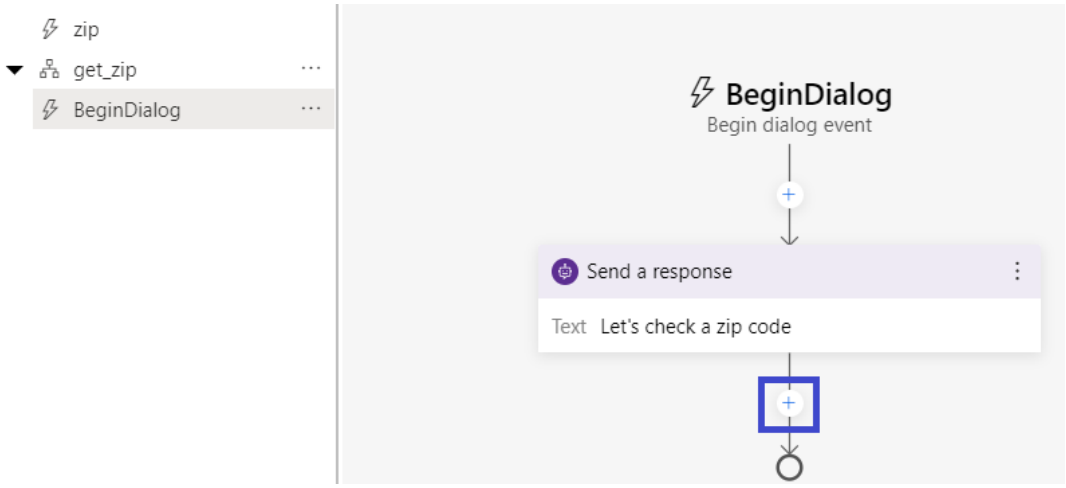

Figure 3-i: Composer – get_zip – BeginDialog

Next, click the **Ask a question** menu item and then **Text**.

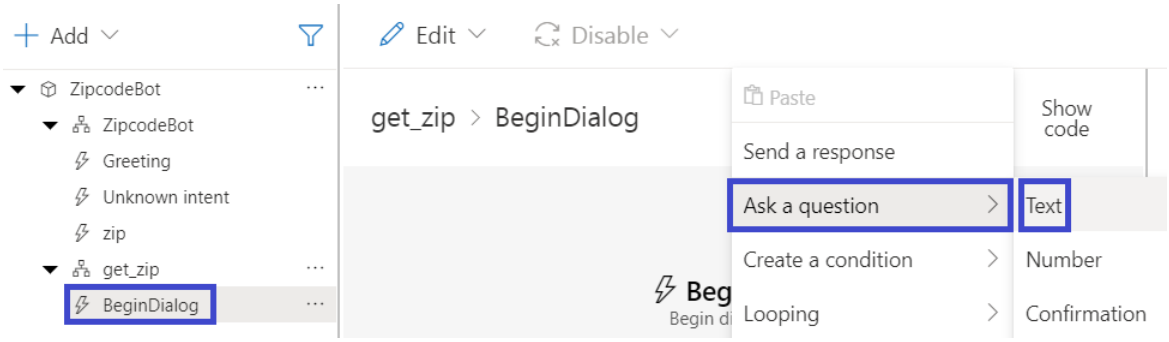

Figure 3-j: Composer – get_zip – BeginDialog – Ask a question – Text

We are prompted to enter the **Prompt for text** followed by the **User input**.

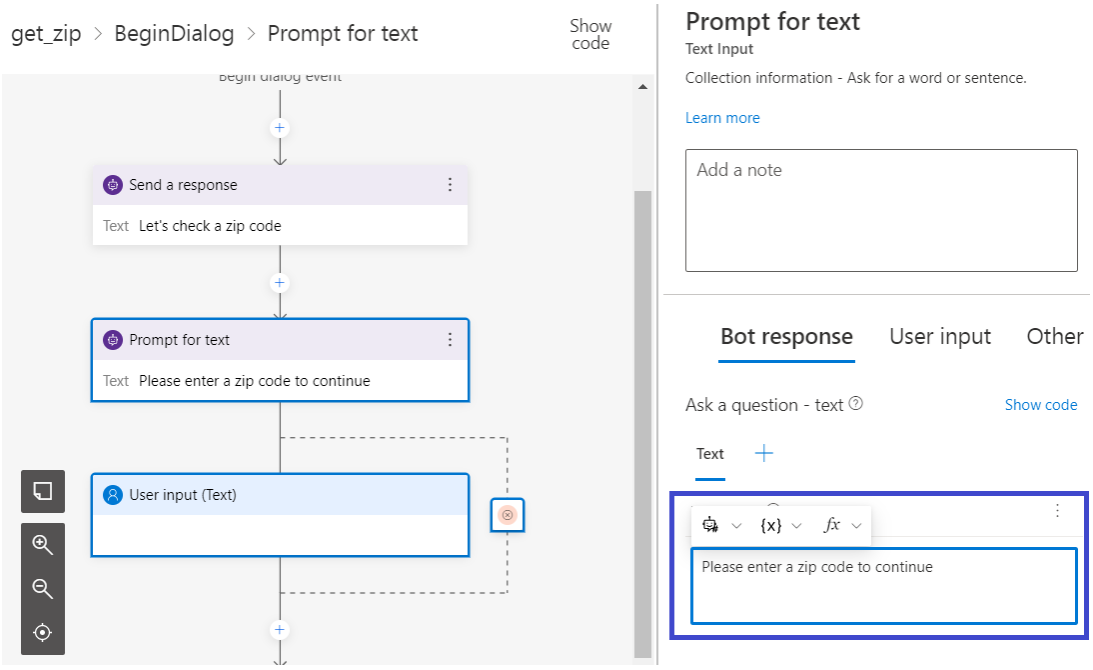

Figure 3-k: Composer – get_zip – BeginDialog – Prompt for text

Under **Bot response**, we can enter the following text: **Please enter a zip code to continue**. With that done, select the **User input (Text)** action. Under **User input**, enter **user.zip** within the **Property** box.

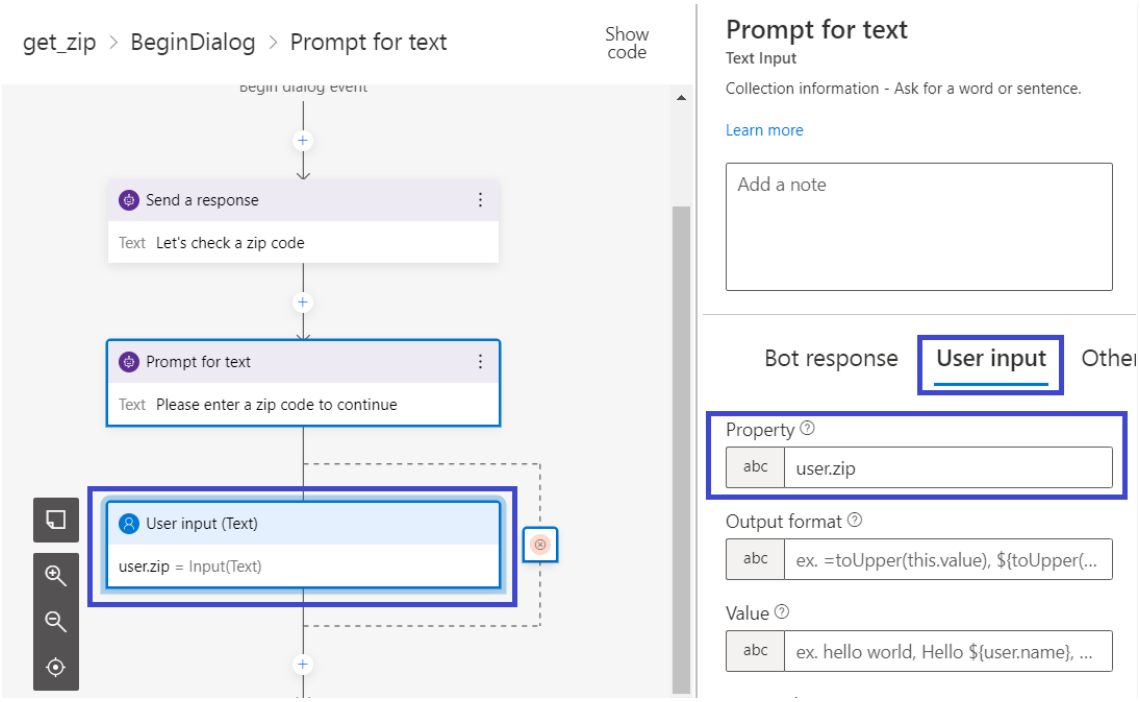

Figure 3-l: Composer – get_zip – BeginDialog – User input

Output format

Now that we have specified the user input, we need to indicate the output format. To do that, click the **Output format** box then enter the value **trim(this.value)** in the field.

The **`trim`** function is a prebuilt expression that removes leading and trailing spaces from a value, and this is useful in case the user enters the zip code with a leading or trailing space.

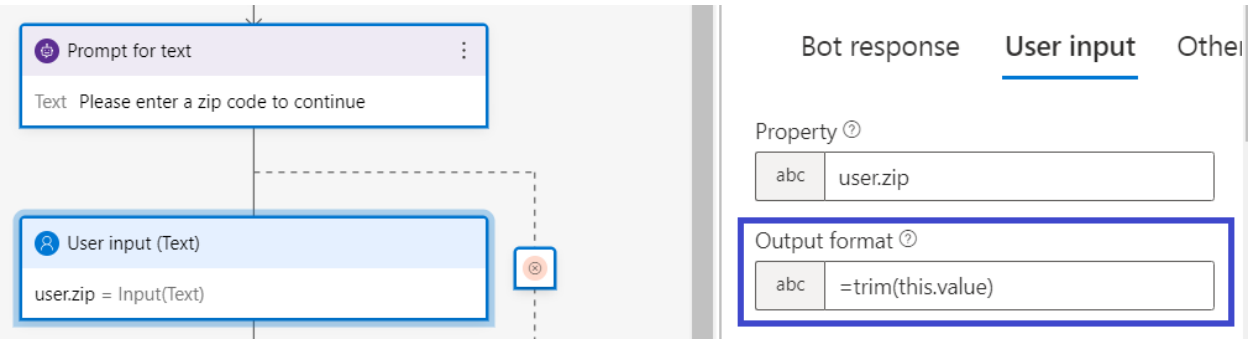

Figure 3-m: Composer – get_zip – BeginDialog – User input – Output format

Input validation

We have managed to gather the user's input and set the output format at this stage. So far, so good. Nevertheless, we need to ensure that the user's data is valid—an action known as input validation.

As the user will be entering a zip code, we should at least confirm that the zip code provided is a valid one. To do that, we can check if the zip code supplied is a valid U.S. zip code.

The Zipcodebase API is valid for multiple countries, and that's a lot of zip code country formats to validate, so let's limit the input validation to U.S. zip codes only.

If the user indicates a U.S. zip code with fewer than five characters or more than five characters, the input validation would be invalid. Otherwise, it would be valid.

Let's get that sorted. In the authoring canvas, click **Other** in the properties pane. Expand the **Recognizers** section, click **Add alternative**, and enter the text shown in the following figure.

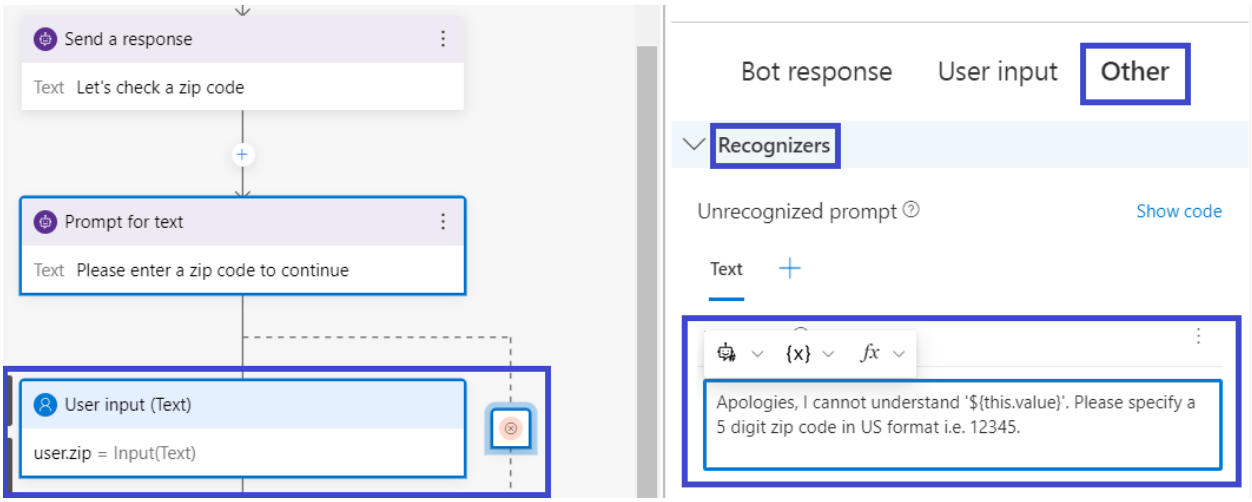

Figure 3-n: Composer – get_zip – BeginDialog – User input (Text) – Other – Unrecognized prompt

The text value is as follows: **Apologies, I cannot understand '${this.value}'. Please specify a 5 digit zip code in US format i.e.12345**.

That's the text response that the bot will return if the user input is not understood.

Next, we need to specify the validation rule to check whether the zip code entered is valid or not. Click the **Validation** section, then under **Validation Rules** click **Add new** > **Write an expression**.

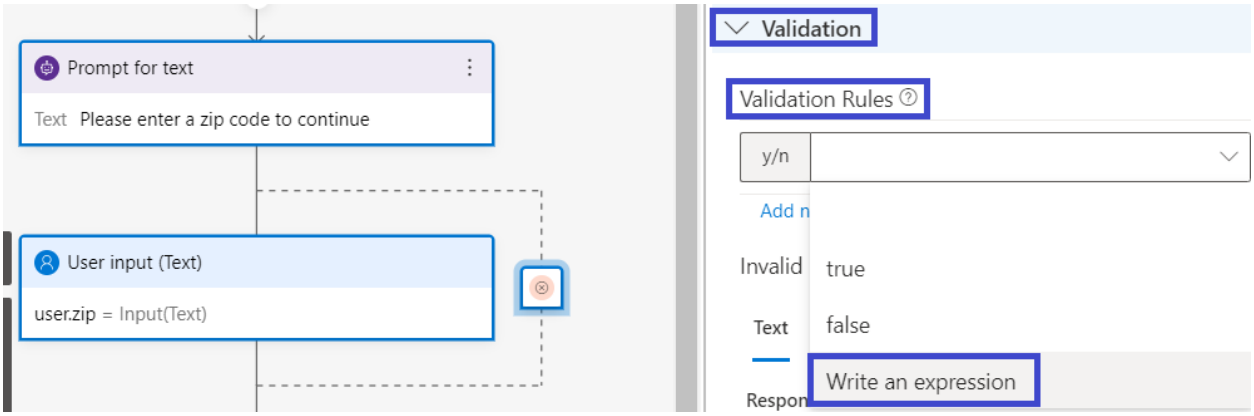

Figure 3-o: Composer – get_zip – BeginDialog – User input (Text) – Other – Validation Rules

Enter the expression **length(this.value) == 5**, which will ensure that the zip code value is five characters long.

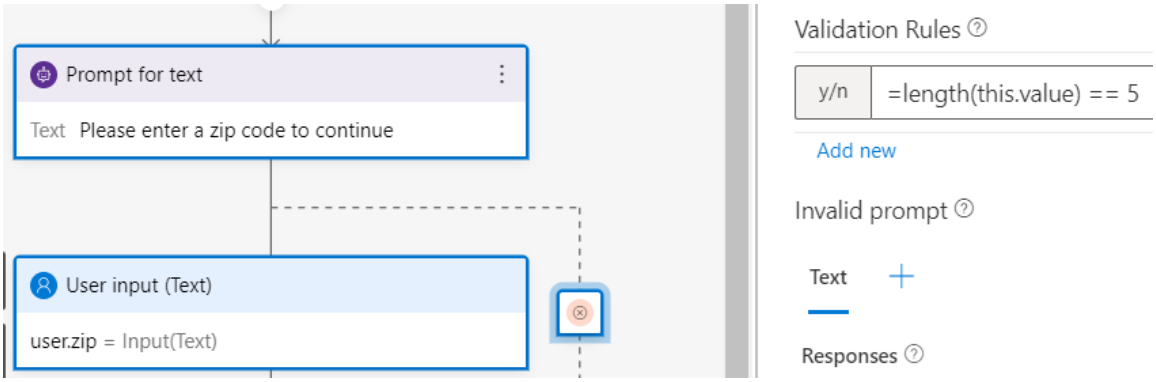

Figure 3-p: Composer – (get_zip – BeginDialog – User input (Text) – Other – Validation Rules

We also want to add a response that the bot can send back to the user if the zip code length is different than five characters. We can do this by clicking **Add alternative** under **Invalid prompt**.

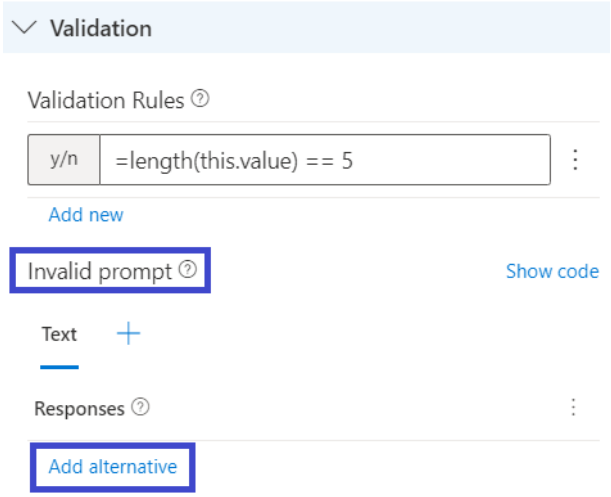

Figure 3-q: Invalid prompt – Add alternative

Enter the following text: **The zip code '${this.value}' is not valid. Please enter a zip code that is 5 digits long**.

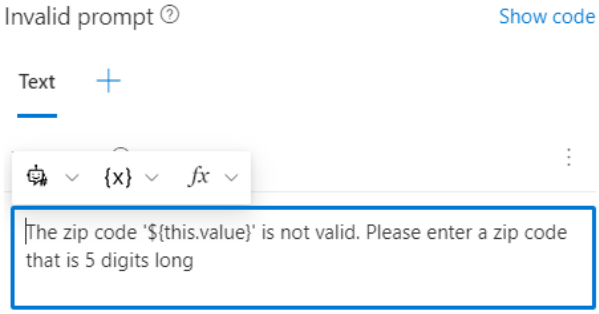

Figure 3-r: Invalid prompt – Text response

Default zip value

Although it is not strictly necessary, it's also possible to add a default value for the zip code that the bot can return.

To do that, click **Prompt configurations**, and under **Default value**, enter a valid U.S. zip code. I'm going to enter **33165**, but you may choose another.

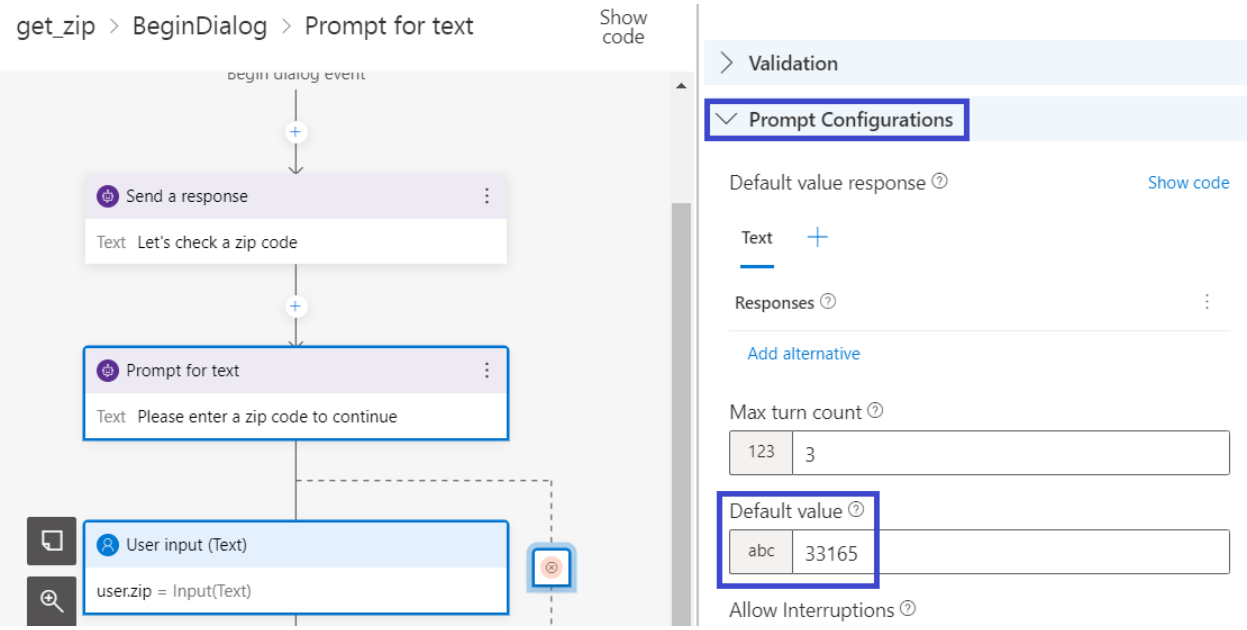

Figure 3-s: Composer – get_zip – BeginDialog – User input (Text) – Other – Prompt Configurations

Summary

We have set the bot up so that whenever a user enters the message **zip**, the bot will respond and request the user to indicate the zip code. If that value is valid, then it will be stored in the **`user.zip`** variable.

If the value is not a valid zip code (not equal to five characters), the bot sends an error message back to the user.

Next, we are going to explore how to make a call to the Zipcodebase API.

Chapter 4 Working with the API

Overview

So far, we have created the bot with enough functionality to ask the user for a zip code and send a reply in case the user's input is not adequate.

However, if the user's feedback is correct and the zip code valid, we cannot process it. That's what we are going to do throughout this chapter.

Getting the API key

To retrieve the data and information related to the zip code, we need to invoke the Zipcodebase API.

To use Zipcodebase, we need to call the API using an API key. Switching back to the Zipcodebase web page, let's copy the value under **API Key** from the **Dashboard**.

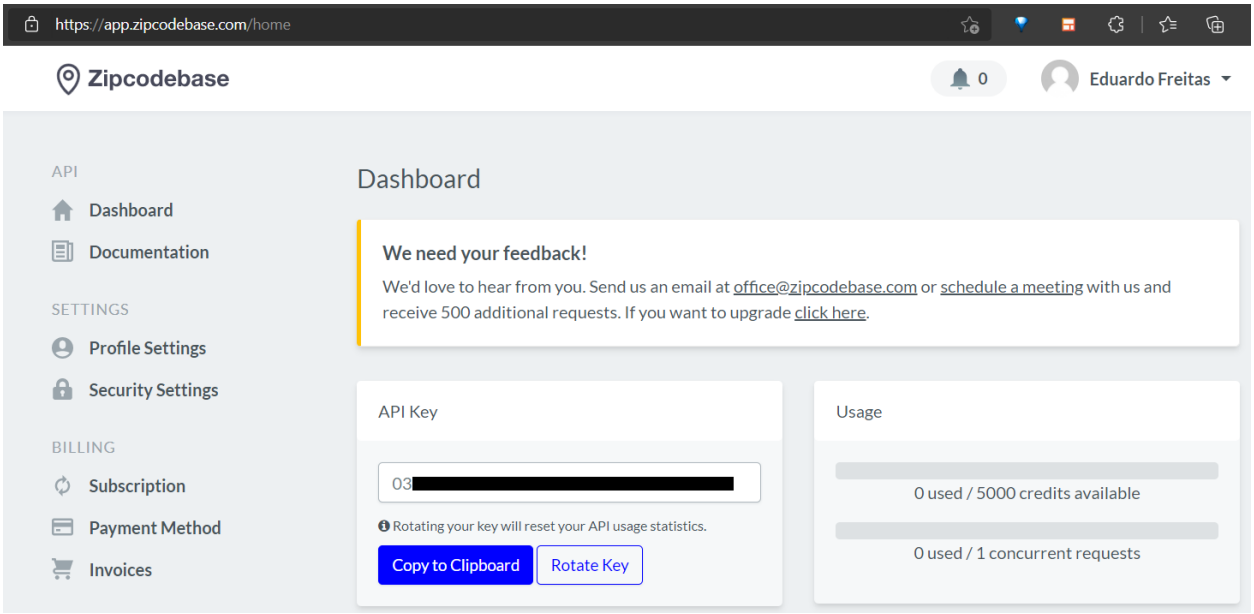

Figure 4-a: Zipcodebase Dashboard with API Key

HTTP request

Going back to Composer, in the bot explorer, make sure that the **BeginDialog** is selected. Then below all the existing actions added, click **+** > **Access external resources** > **Send an HTTP request**.

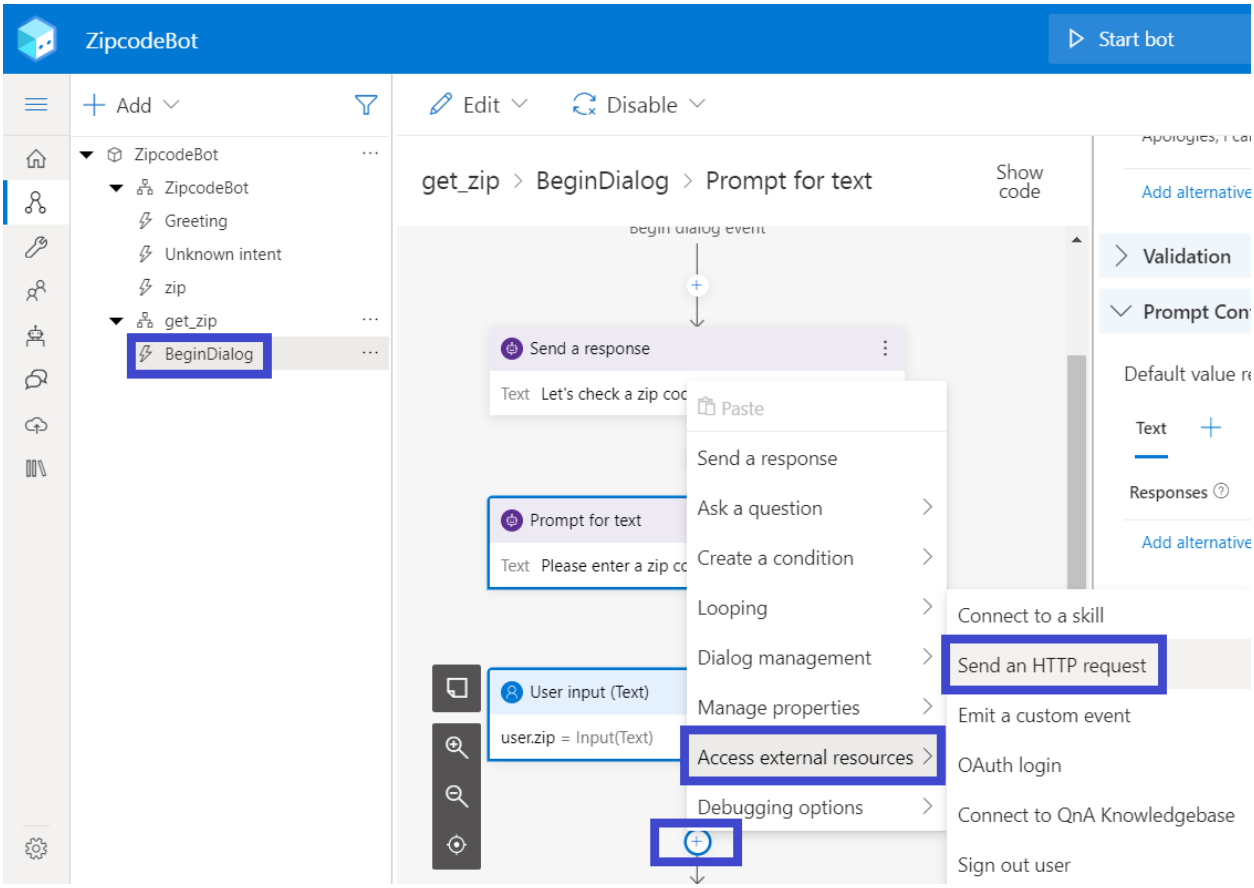

Figure 4-b: Composer – BeginDialog – Access external resources – Send an HTTP request Menu Item

The following details are visible in the properties pane. Select the **GET** option under the **HTTP method**.

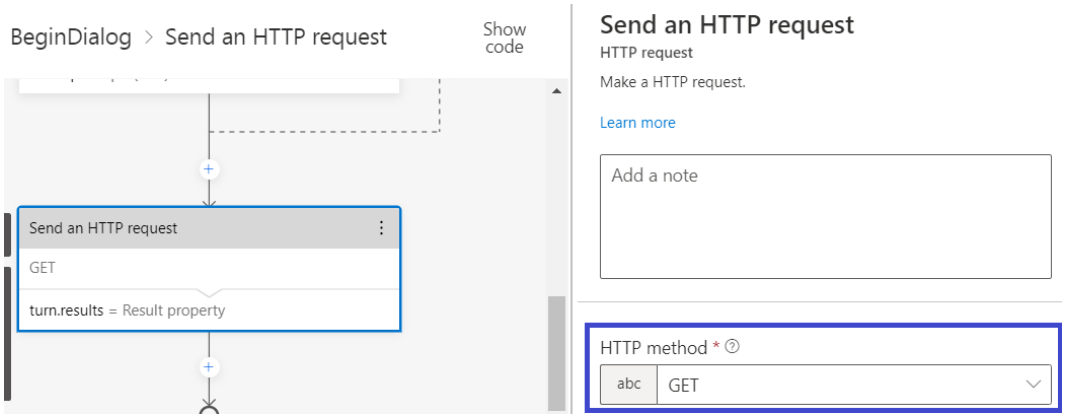

Figure 4-c: Composer – BeginDialog – Send an HTTP request – HTTP method

At this stage, we need to get the API URL, which we can get from the Zipcodebase website as highlighted in the following figure.

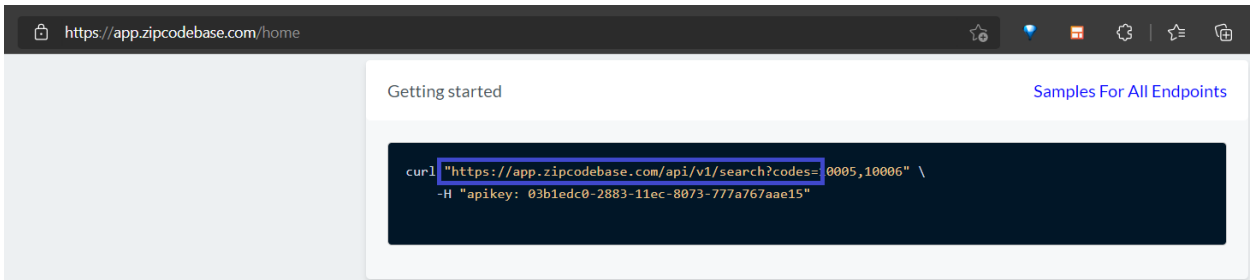

Figure 4-d: Zipcodebase – API URL

The following is the Composer-compatible version of the URL, because the **`user.zip`** variable contains the zip code submitted by the user.

`https://app.zipcodebase.com/api/v1/search?codes=${user.zip}`

Next, we need to add the API key to the URL and the country code. So let's copy the **API Key** value from the Zipcodebase website to add it to the URL.

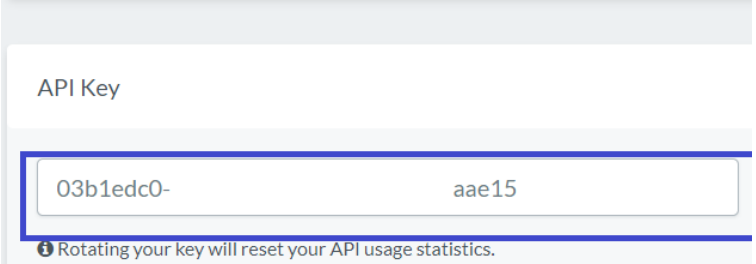

Figure 4-e: Zipcodebase Website API Key Value

In the following URL, replace **`API_KEY_VALUE_GOES_HERE`** with the value of your API key copied from the Zipcodebase website.

`https://app.zipcodebase.com/api/v1/search?apikey=API_KEY_VALUE_GOES_HERE&codes=${user.zip}&country=US`

Enter the URL into the **Url** field within the properties pane of Composer, as shown in the following figure.

Figure 4-f: Composer – BeginDialog – Send an HTTP request – Url

When the bot performs the HTTP request, the response must be stored somewhere—in a variable assigned to the **`Result`** property.

We will store the result within the **dialog.api_response** variable. The dialog is a scope that retains its properties for the duration of a dialog, in this case, **BeginDialog**.

Tip: To understand how properties and variable scopes work, I suggest looking at the official documentation.

Figure 4-g: Composer – BeginDialog – Send an HTTP request – Result property

Next, we need to set the **Response** type value to **json**.

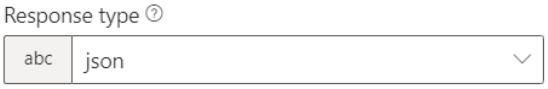

Figure 4-h: Composer – BeginDialog – Send an HTTP request – Response type

HTTP status code

When working with HTTP requests, status codes are essential. Status codes indicate whether the request was successful or not.

Therefore, the bot must determine whether the response was successful before sending a response to the user.

A status code with a value of 200 indicates that the response obtained from the API was successful. A status code with a different value would suggest a problem accessing the API—in a situation like this, we need to create a branch.

Creating a branch

To create a branch—which in programming would be the equivalent of using an if-else condition—we need to go back to the authoring canvas and click **+** under **Send an HTTP request**. Then, click **Create a condition** > **Branch: If/else**.

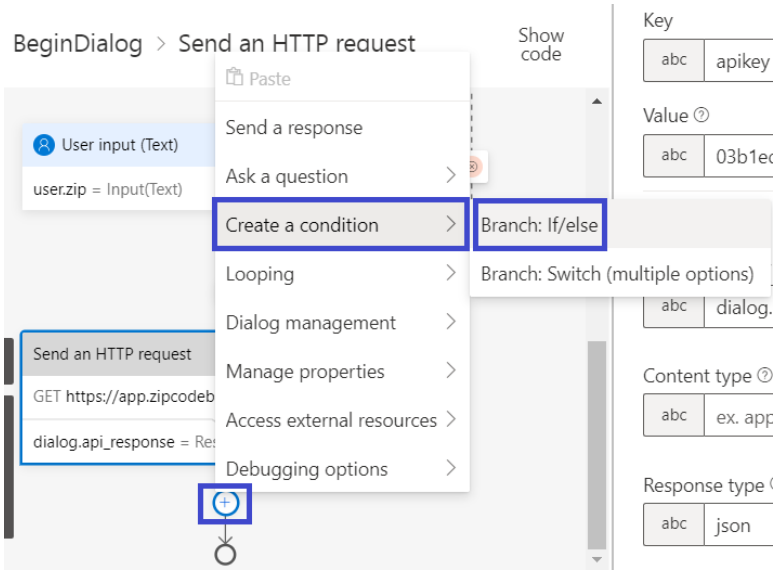

Figure 4-i: Composer – BeginDialog – Create a condition – Branch: If/else

The **Branch: If/else** appears in the authoring canvas. Select the branch, and in the properties pane, under **Condition**, select the **Write an expression** option.

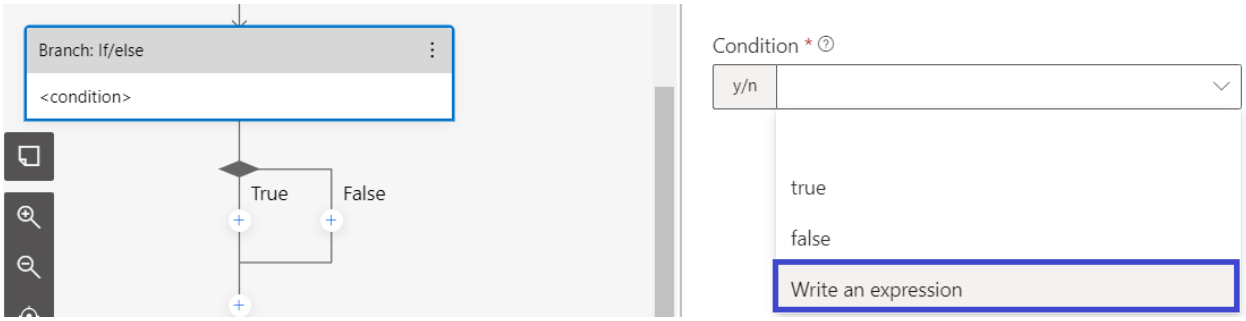

Figure 4-j: Composer – BeginDialog – Branch: If/else – Condition – Write an expression

As the **Condition**, set the value to: **dialog.api_response.statusCode == 200**.

Figure 4-k: Composer – Begin Dialog – Branch: If/else – Condition

Under the **True** branch, click **+**.

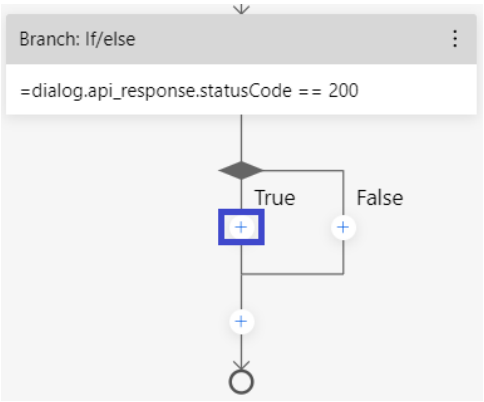

Figure 4-l: Composer – BeginDialog – Branch: If/else – True Branch

Once that is done, click **Manage properties** > **Set properties**.

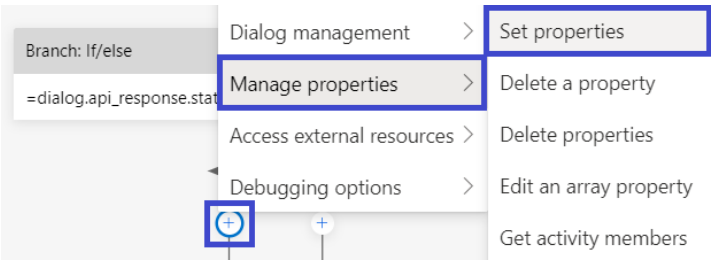

Figure 4-m: Composer – BeginDialog – Branch: If/else – Manage properties – Set properties

You will then see the following.

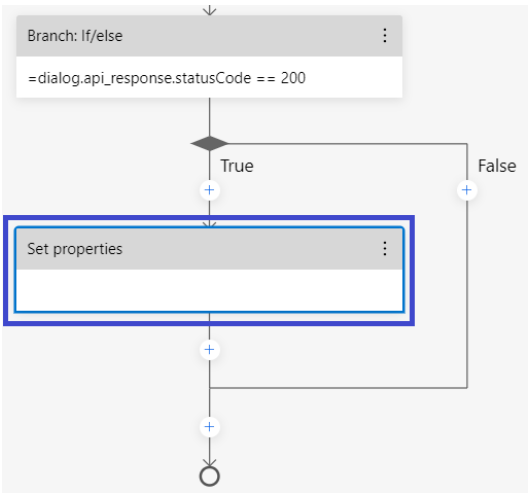

Figure 4-n: Composer – BeginDialog – Branch: If/else – True Branch – Set properties

We will use the **Set properties** action because we want to assign the response values obtained from the API to a few variables. This way, the results are provided to the user with the requested zip code information.

Querying the API

Before proceeding, we need to check which values the API can return. To do that, let's use Hoppscotch, which is an open-source API development web application that allows you to query and interact with any REST API.

So, point your browser to the Hoppscotch website (I usually use either Microsoft Edge or Google Chrome, but feel free to use another modern browser). Add the following URL to the field highlighted in the screenshot below.

`https://app.zipcodebase.com/api/v1/search?apikey=API_KEY_VALUE_GOES_HERE&codes=33165&country=US`

Make sure you replace **`API_KEY_VALUE_GOES_HERE`** with the value of your Zipcodebase **API Key**. For testing purposes, I've replaced **`${user.zip}`** with **`33165`**.

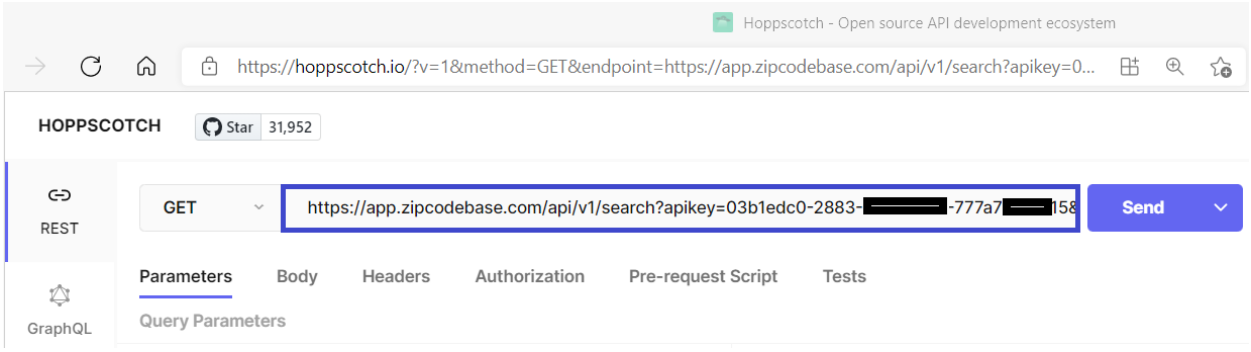

Figure 4-o: Hoppscotch Website Querying the Zipcodebase API

With the URL pasted, you can test the API by clicking **Send**. Let's see what happens.

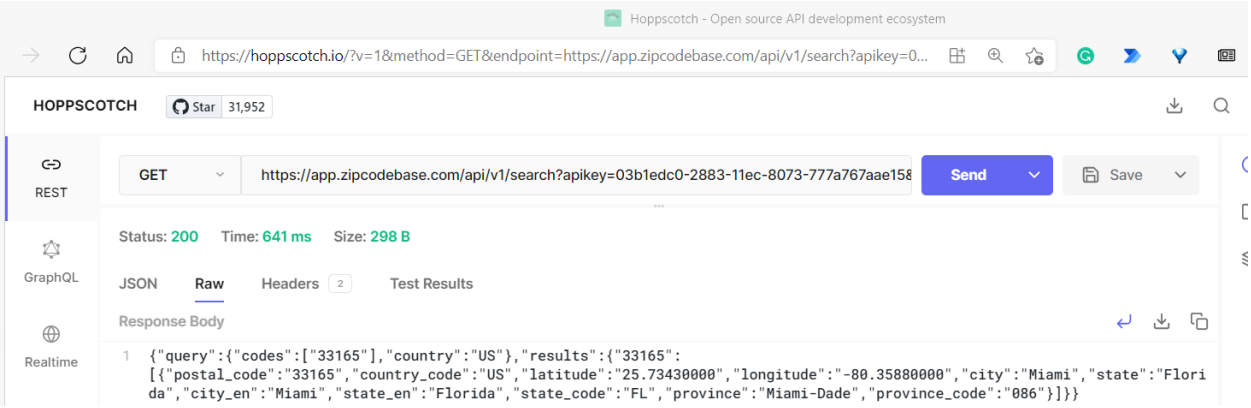

Figure 4-p: Hoppscotch Website – Zipcodebase API Query Raw Results

As seen in the preceding figure, the API returns a result. To see the result in full detail, click the **JSON** tab. In most cases, when using the Hoppscotch website, the **JSON** tab will be the default tab.

JSON Raw Headers 2 Test Results

Response Body


```
1  {
2    "query": {
3      "codes": [
4        "33165"
5      ],
6      "country": "US"
7    },
8    "results": {
9      "33165": [
10       {
11         "postal_code": "33165",
12         "country_code": "US",
13         "latitude": "25.73430000",
14         "longitude": "-80.35880000",
15         "city": "Miami",
16         "state": "Florida",
17         "city_en": "Miami",
18         "state_en": "Florida",
19         "state_code": "FL",
20         "province": "Miami-Dade",
21         "province_code": "086"
22       }
23     ]
24   }
25 }
```


Figure 4-q: Hoppscotch Website – Zipcodebase API Query JSON Results

By inspecting the result returned by the API, we can find two distinct sections: the **`query`** and **`results`** sections. We want to get the data from the **`results`** section.

Notice that the **`results`** JSON object contains another JSON object with the zip code value (**`33165`**) passed as a query parameter to the API call, which is an array of JSON objects with one element in this case.

First assignment

For each of those values contained within that array element, we want to create a property and assign its respective value so that the bot can return them to the user as a response.

To do that, go back to Composer, make sure that the **Set properties** action is selected, and then within the properties pane, click **Add new** under **Assignments**.

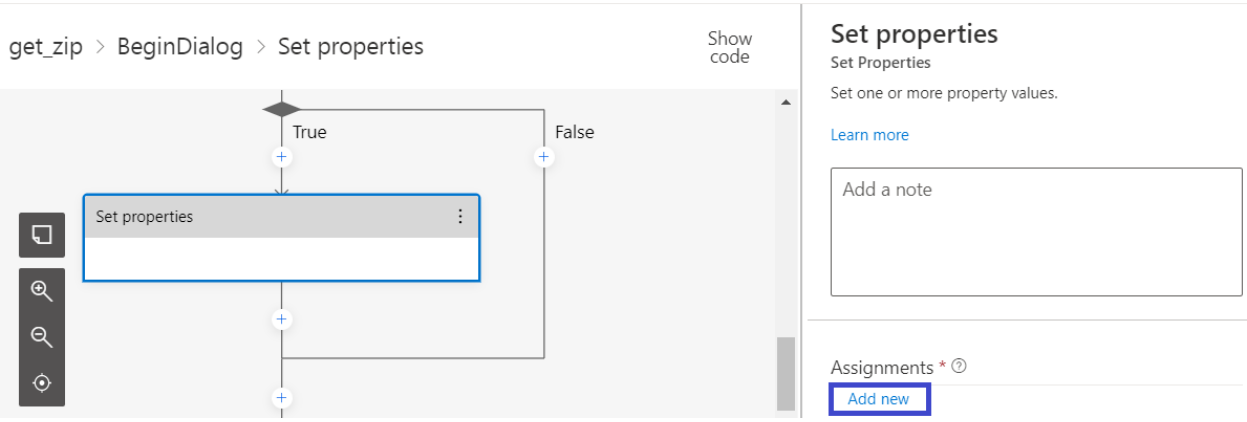

Figure 4-r: Composer – BeginDialog – Set properties – Assignments – Add new

Let's add **`dialog.postal_code`** as the **Property** name.

Let's also add **`dialog.api_response.content.results[user.zip][0]['postal_code']`** as the **Value**, as seen in the following figure.

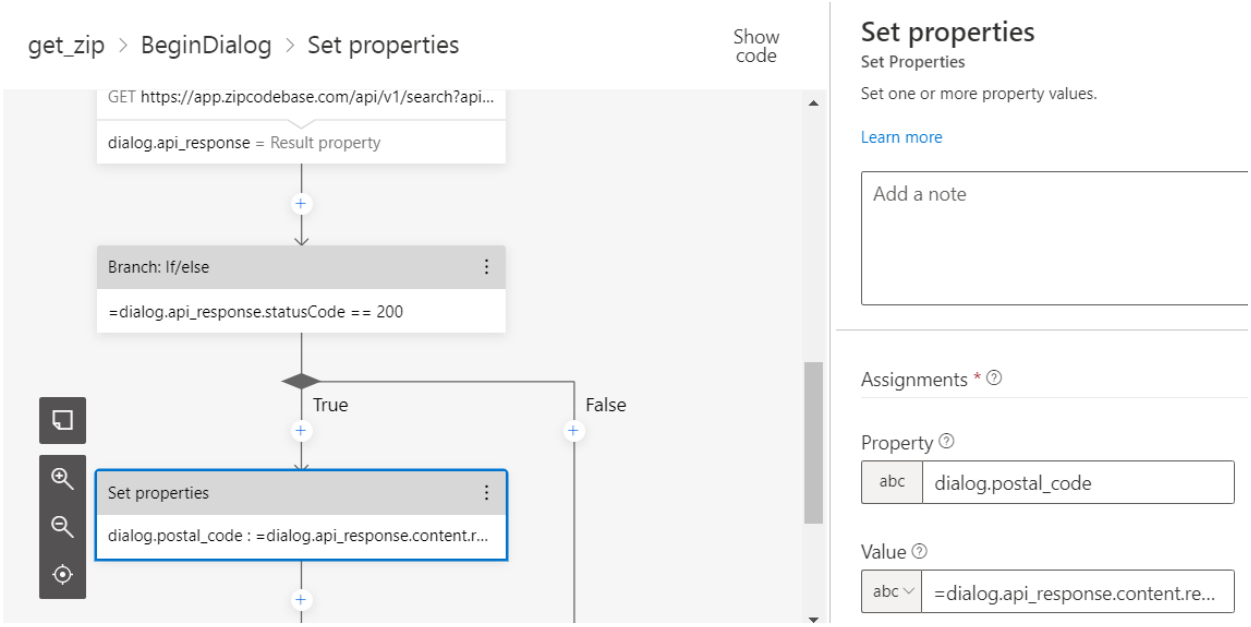

Figure 4-s: Composer – BeginDialog – Set properties – Assignments – Property/Value

So, let's analyze what we have just done here. We are going to store the zip code returned by the API call within the **`dialog.postal_code`** variable. The zip code returned by the API is accessible using **`dialog.api_response.content.results[user.zip][0]['postal_code']`**.

But how did I reach the conclusion that the zip code returned by the API is available through **`dialog.api_response.content.results[user.zip][0]['postal_code']`**? To understand this better, let's have a look at the following diagram.

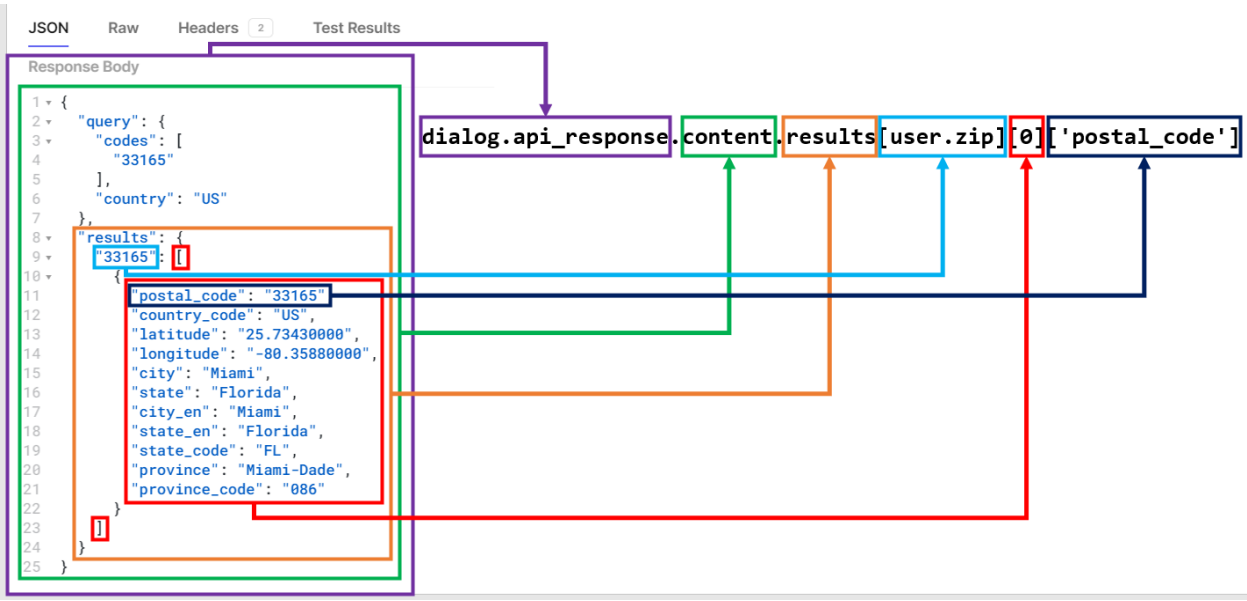

Figure 4-t: Relationship between the API Response and Assignments Value

We can see that **`dialog.api_response`** corresponds to the complete response (highlighted in **purple**) returned by the API, including the header and **Response Body**.

Contained within the **Response Body**, we find the **`content`** highlighted in **green**.The **`content`** includes a **`query`** and a **`results`** object. Within the **`content`** object, highlighted in **orange-yellow**, we see the **`results`** object.

Within the **`results`** object, we find another object (highlighted in **light blue**) that contains an array, and this object has the same value as the zip code queried (which we previously stored in the **`user.zip`** variable).

Finally, within the array's first element (**`[0]`**), highlighted in **red**, we find each of the properties. We want to retrieve the value of the **`postal_code`** property.

Other assignments

Now that we have assigned the value of the first property (**`postal_code`**) returned by the API call, let's do the same for the other properties returned by the API.

All we need to do is click **Add new** for each property we want to add from the **`results`** object.

By using **`dialog.api_response.content.results[user.zip][0]['country_code']`**, we can retrieve the value of **`country_code`**. Store it as **`dialog.country_code`**.

Figure 4-u: Composer – The dialog.country_code Property

Likewise, with **`dialog.api_response.content.results[user.zip][0]['latitude']`**, we can retrieve the value of **`latitude`**. Store it as **`dialog.latitude`**.

Figure 4-v: Composer – The dialog.latitude Property

With **`dialog.api_response.content.results[user.zip][0]['longitude']`**, we can retrieve the value of **`longitude`**. Store it as **`dialog.longitude`**.

Figure 4-w: Composer – The dialog.longitude Property

With **`dialog.api_response.content.results[user.zip][0]['city']`**, we can retrieve the value of **`city`**. Store it as **`dialog.city`**.

Figure 4-x: Composer – The dialog.city Property

With **`dialog.api_response.content.results[user.zip][0]['state']`**, we can retrieve the value of **`state`**. Store it as **`dialog.state`**.

Figure 4-y: Composer – The dialog.state Property

With **`dialog.api_response.content.results[user.zip][0]['city_en']`**, we can retrieve the value of **`city_en`**. Store it as **`dialog.city_en`**.

Figure 4-z: Composer – The dialog.city_en Property

With **`dialog.api_response.content.results[user.zip][0]['state_en']`**, we can retrieve the value of **`state_en`**. Store it as **`dialog.state_en`**.

Figure 4-aa: Composer – The dialog.state_en Property

With **`dialog.api_response.content.results[user.zip][0]['state_code']`**, we can retrieve the value of **`state_code`**. Store it as **`dialog.state_code`**.

Figure 4-ab: Composer – The dialog.state_code Property

With **`dialog.api_response.content.results[user.zip][0]['province']`**, we can retrieve the value of **`province`**. Store it as **`dialog.province`**.

Figure 4-ac: Composer – The dialog.province Property

With **`dialog.api_response.content.results[user.zip][0]['province_code']`**, we can retrieve the value of **`province_code`**. Store it as **`dialog.province_code`**.

Figure 4-ad: Composer – The dialog.province_code Property

Note: Each** Value **field starts with a** = **character, whereas the** Property **fields do not.

Tip: It is not necessary to save each value returned by the API, just the ones that your bot will use. In this case, I've opted to store each value returned by the API call to highlight how to do it.

Summary

Our bot is not ready yet. Nevertheless, the goal of this chapter was to explore how to interact with the API, obtain a response, and store each response value, which we have done.

In the chapter that follows, we will add the remaining steps to finalize the creation of our bot.

Chapter 5 Finalizing the Bot

Overview

Looking at the authoring canvas, we can see that the **BeginDialog** looks as follows.

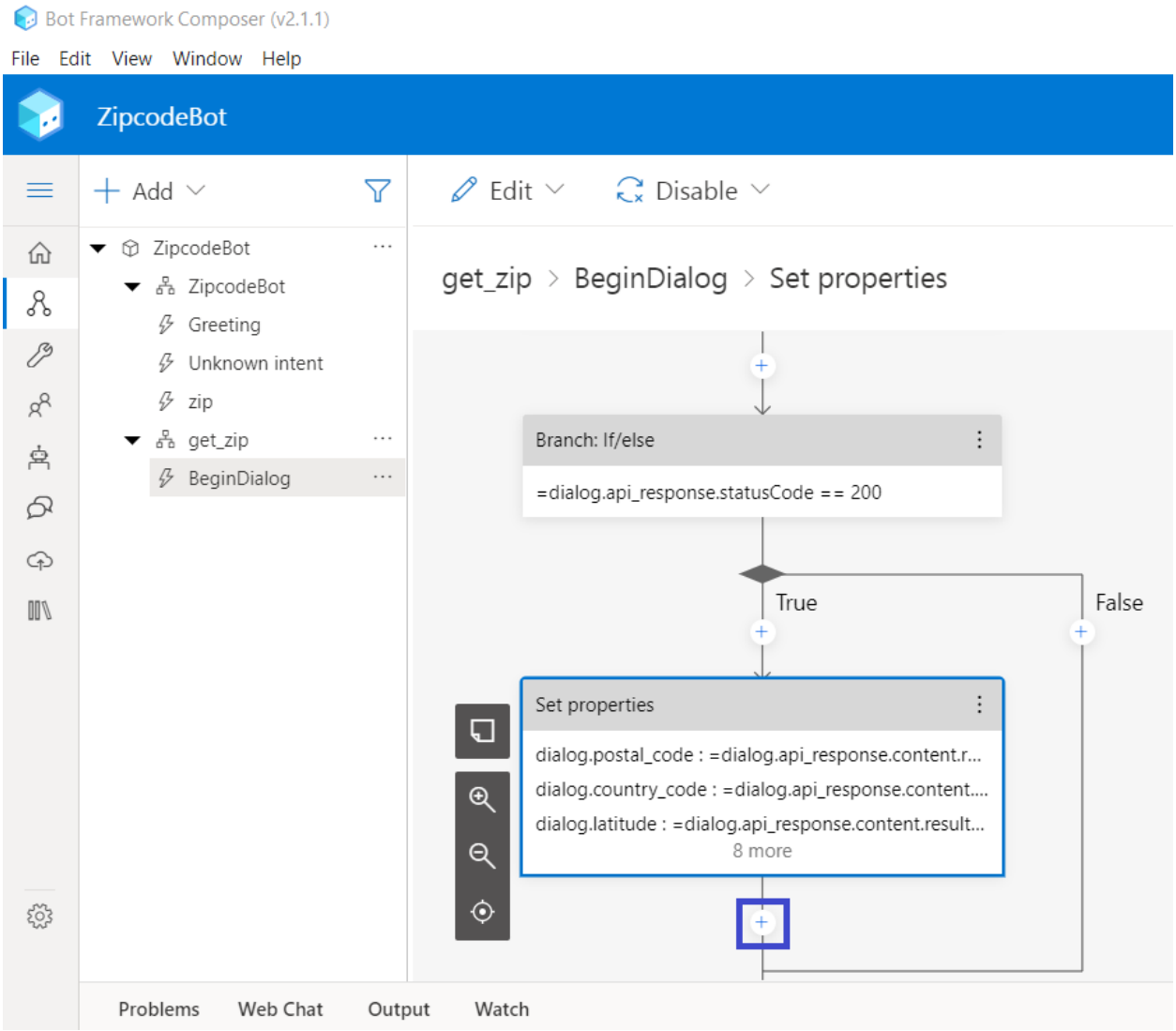

Figure 5-a: Composer – BeginDialog

We can see that the **Set properties** action now contains all the assignments we previously created, which is excellent!

API results as a response

Let's send a response to the user with some of the details obtained using the API. Click **+** just below **Set properties**. Then, click **Send a response**.

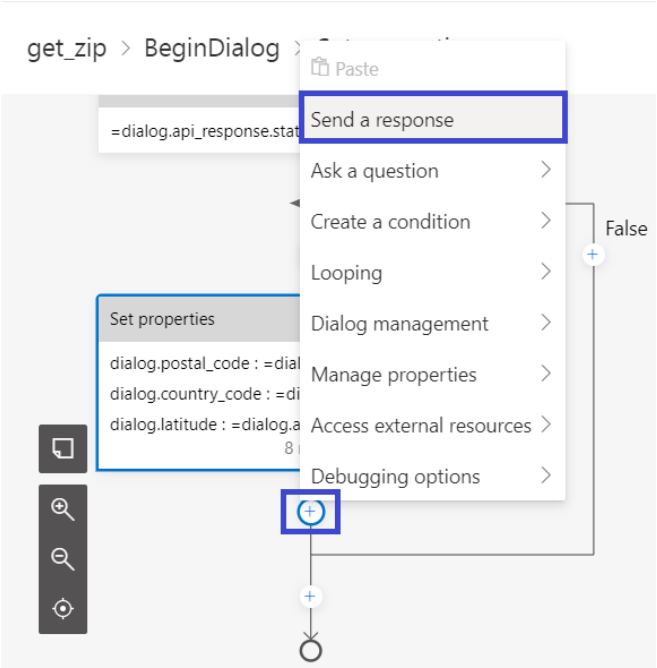

Figure 5-b: Composer – BeginDialog – Send a response

Select **Send a response** > **Add alternative**.

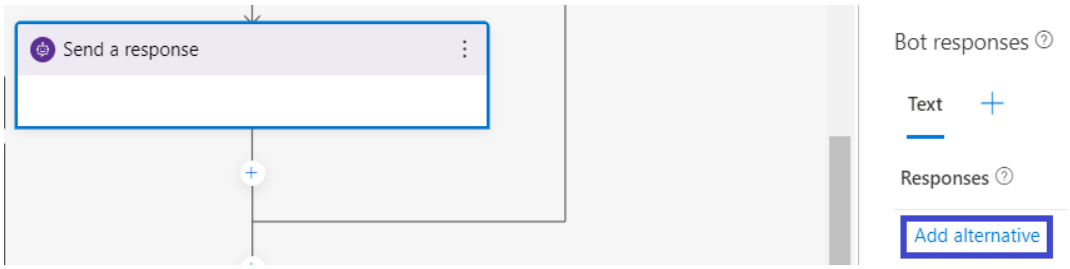

Figure 5-c: Composer – BeginDialog – Send a response – Responses

We can enter the answer that the bot will return to the user (obtained from the API). Let's respond to the user with the city, state, and county (province).

So, let's enter the following text as a response: **`City: ${dialog.city}, State: ${dialog.state}, County: ${dialog.province}`**.

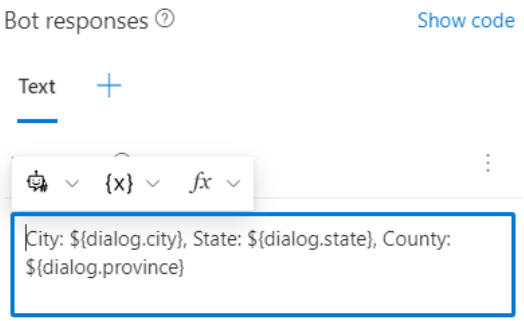

Figure 5-d: Composer – BeginDialog – Send a response – Bot responses

With that done, let's test our bot for the first time.

First execution

To execute the bot for the first time, click **Start bot** as shown in the following figure. Starting a bot takes a few seconds.

Figure 5-e: Composer – Start a bot

Once the bot is running, click **Open Web Chat**.

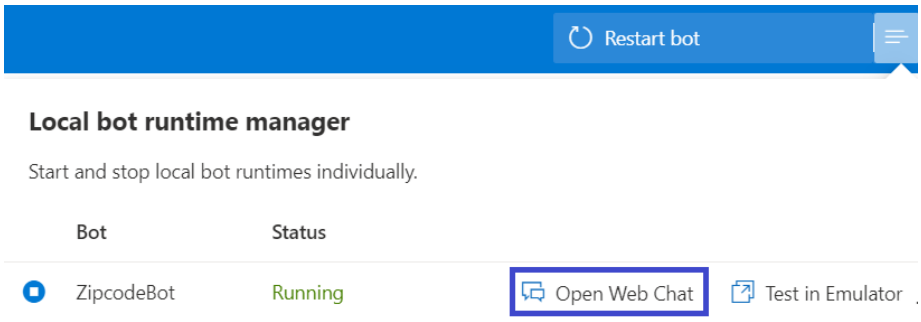

Figure 5-f: Composer – Open Web Chat

The chat window will appear on the right-hand side of Composer.

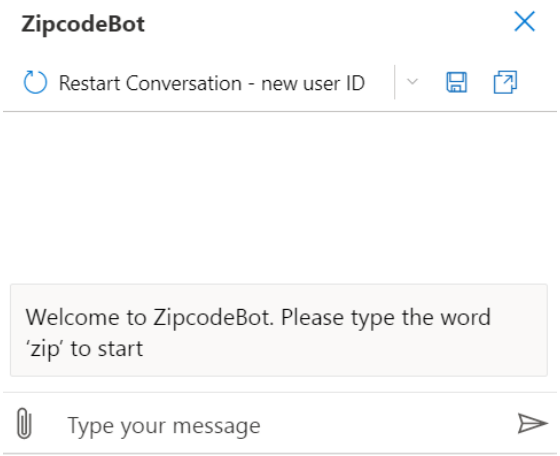

Figure 5-g: Composer – Chat Window (1)

To begin the conversation with the bot, let's type the word **zip**; this triggers the bot to respond and request the user to enter a zip code to continue.

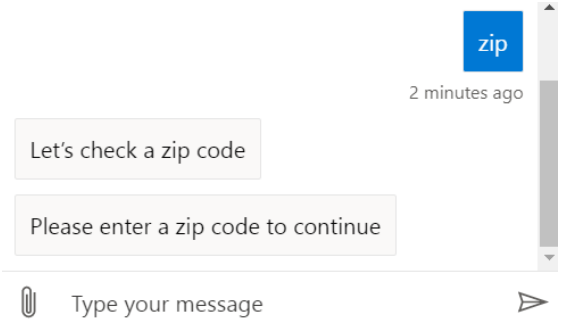

Figure 5-h: Composer – Chat Window (2)

Let's enter **98052** as the zip code and see how the bot responds.

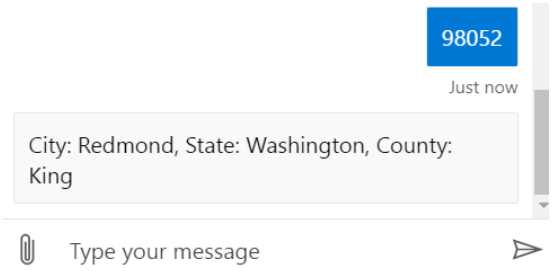

Figure 5-i: Composer – Chat Window (3)

Great—we can see that the bot returned the city, state, and county corresponding to that zip code using the Zipcodebase API.

To stop the execution of the bot, click the buttons highlighted in the following figure (first on 1 and then on 2).

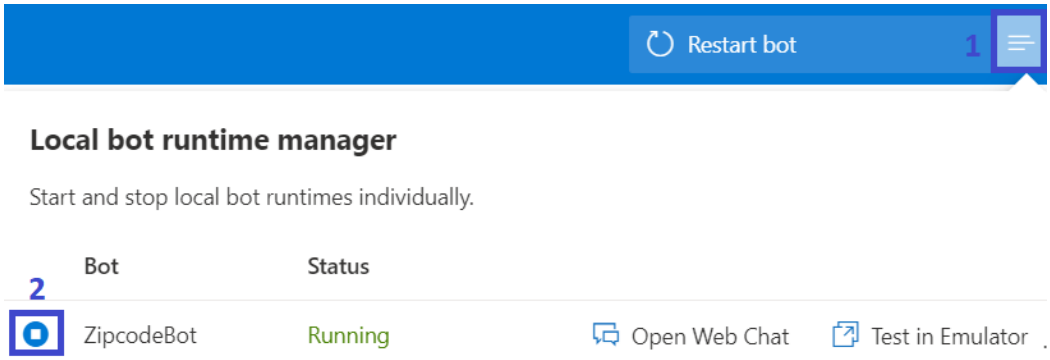

Figure 5-j: Composer – How to Stop the Bot

Different status code branch

When we created the branch to check if the API returns a 200 HTTP status code, we finished the branch that occurs when that condition is true; however, we did not specify what happens if the API returns a status code different than 200.

We can first send a response to the user indicating that an error occurred when calling the API. To do that, click **+** under **False** (**Branch: If/else** action) and then click **Send a response**.

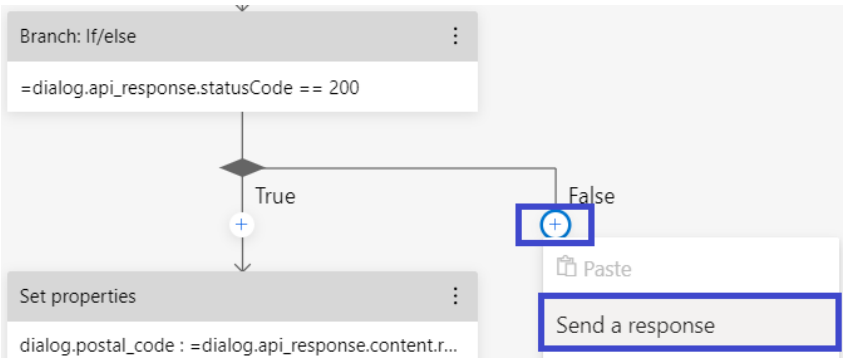

Figure 5-k: Composer – BeginDialog – Branch: If/else – False Branch – Send a response

Then, under **Responses**, enter text that indicates that a problem happened when calling the API in the properties pane.

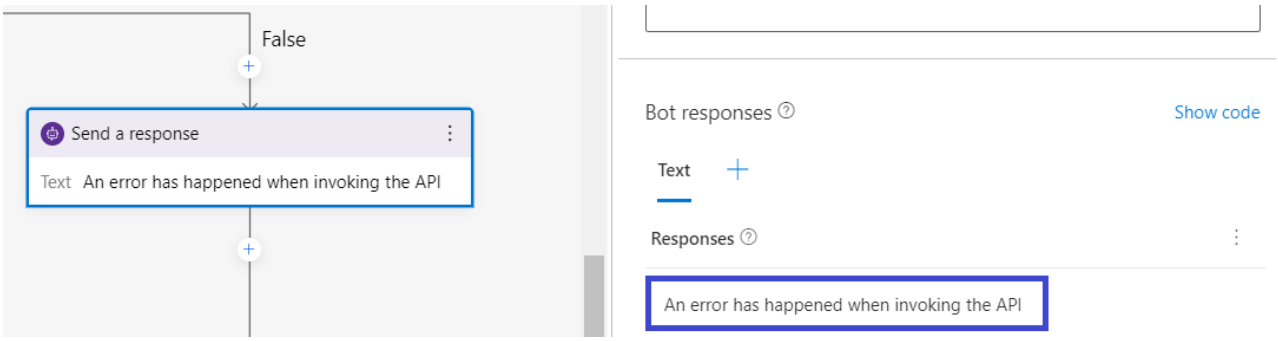

Figure 5-l: Composer – BeginDialog – Branch: If/else – False Branch – Send a response – Responses

In this branch, we need to remove the zip code value entered by the user so that the value doesn't persist on the **`user.zip`** variable when the API returns an error or cannot process the request.

To do that, click **+** under the **Send a response** action just added.

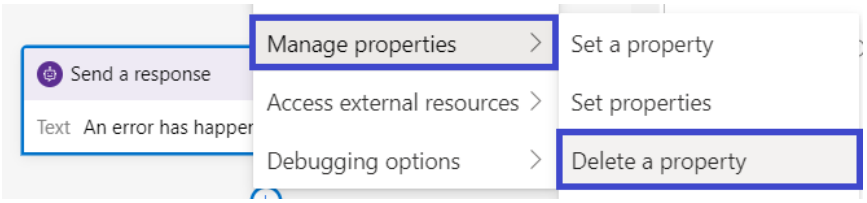

Figure 5-m: Composer – BeginDialog – Manage properties – Delete a property

The **Delete a property** action is shown. In the properties pane, enter **user.zip** for the **Property** field.

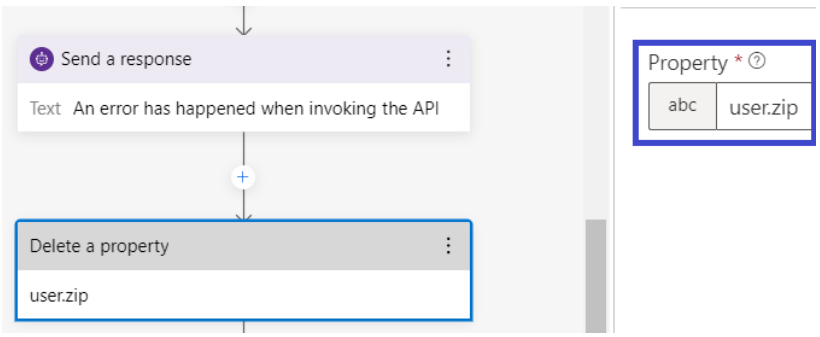

Figure 5-n: Composer – BeginDialog – Delete a property – Property

If the call to the API fails, the bot will not store the zip code value that the user entered.

Adding a package

Another helpful feature that any bot should have is the ability to allow the user to interrupt the conversation, which involves canceling the active dialog. So let's implement this. In Composer, click the **Package manager** icon (which resembles three books).

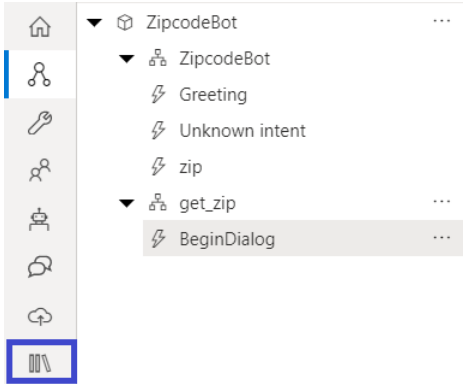

Figure 5-o: Composer – Package Manager Icon

Then, within the search field, type **helpandcancel** and press **Enter**. Click the package **Microsoft.Bot.Components.HelpAndCancel** to select it.

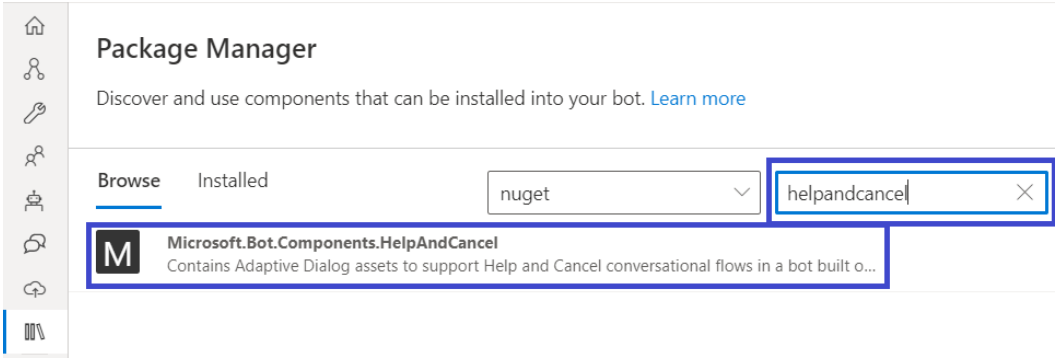

Figure 5-p: Composer – Package Manager – Microsoft.Bot.Components.HelpAndCancel (1)

Then, click the installation button.

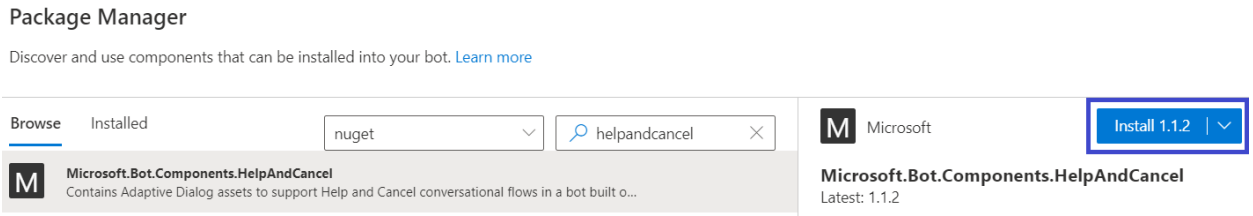

Figure 5-q: Composer – Package Manager – Microsoft.Bot.Components.HelpAndCancel (2)

After that, Composer will install the package. Following the package installation, a **Project Readme** pop-up window might appear. If it does, click **OK**.

Then you will see the package as installed.

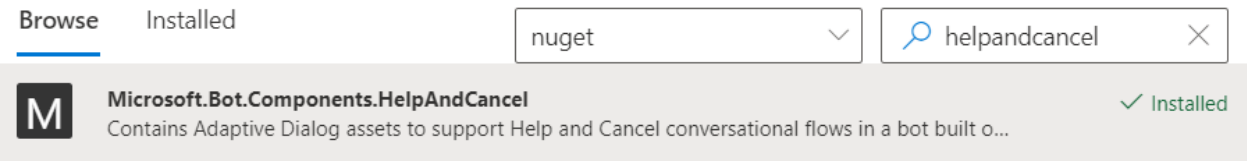

Figure 5-r: Composer – Package Manager – Microsoft.Bot.Components.HelpAndCancel (Installed)

Interrupting the conversation

Now we have the technical ability to interrupt the conversation if the user wishes to, but we still need to implement that functionality.

The installation of this new package added a **CancelDialog** and a **HelpDialog** to the bot. If this is not visible, close Composer and open it again. After opening Composer, you will see the following welcome screen.

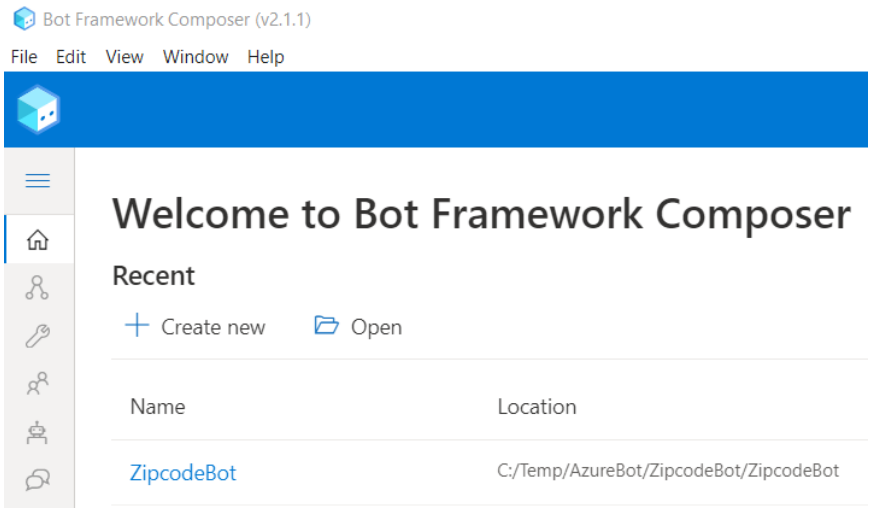

Figure 5-s: Composer – Welcome Screen

Click **ZipcodeBot** to open the bot. This will take you to the authoring canvas, where you can continue to work on the bot. Notice that now, in bot explorer, you can see the **CancelDialog** and **HelpDialog** added to the bot.

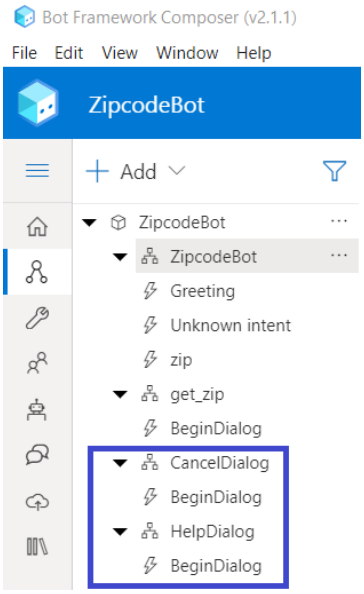

Figure 5-t: Composer – Bot Explorer – CancelDialog and HelpDialog

We will use the **CancelDialog** to give the user the option of canceling the conversation with the bot.

To do that, select the **ZipocodeBot** dialog, and then click **+ Add new trigger**.

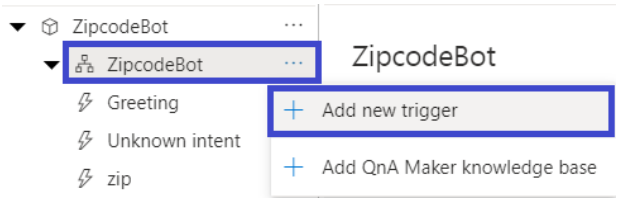

Figure 5-u: Composer – ZipcodeBot – Add new trigger

As the trigger type, leave the default option **Intent recognized**. Let's enter **cancel** as the trigger name, and as for the input pattern, let's enter the value **stop|quit|cancel**. When you're done, click **Submit**.

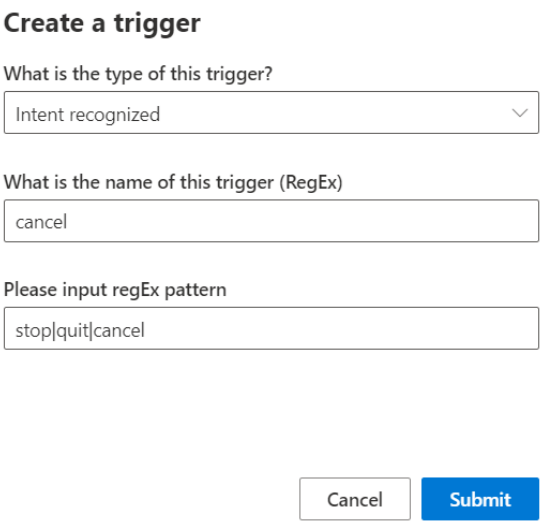

Figure 5-v: Composer – Create a trigger

Now, Composer creates the **cancel** trigger.

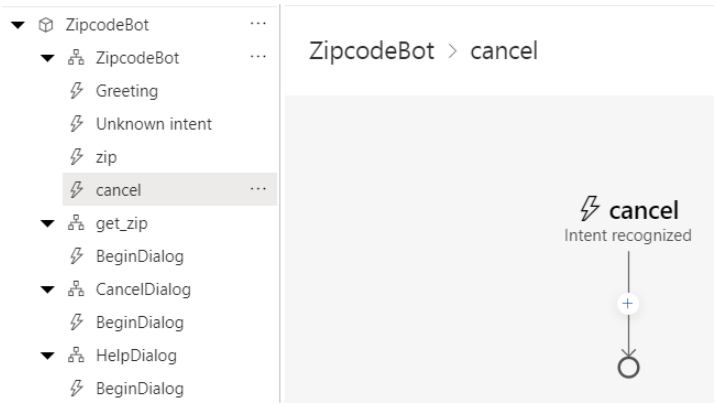

Figure 5-w: Composer – The cancel Trigger

Next, we need to add a **CancelDialog**, so let's do that.

Adding a CancelDialog

In the authoring canvas, under **cancel** (**Intent recognized**), click **+** > **Dialog management** > **Begin a new dialog**.

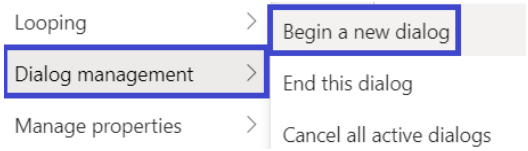

Figure 5-x: Composer – Dialog management – Begin a new dialog

Then in the properties pane, under **Dialog name**, select the option **CancelDialog**.

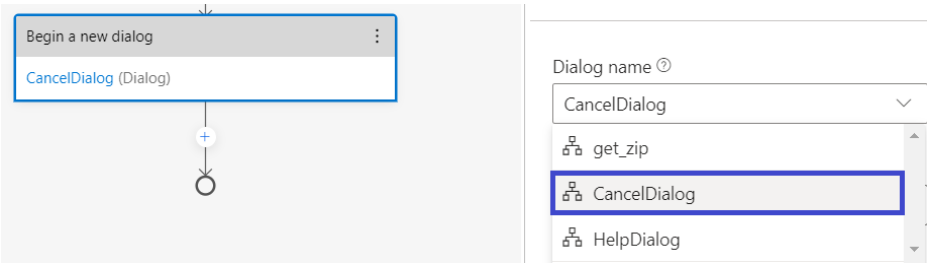

Figure 5-y: Composer – Cancel – Begin a new dialog – Dialog name – CancelDialog

The **cancel** trigger connects to the **CancelDialog**, and we have the basic framework for allowing the user to stop the conversation with the bot.

Enabling interruptions

So far, the bot knows how to retrieve the zip code information using the Zipcodebase API, but it still doesn't know what to do when the user wants to interrupt the conversation. Therefore, we will now enable interruptions so that the conversation can stop when the user requests it.

In the bot explorer, click **get_zip**, and then select the **BeginDialog** trigger. In the authoring canvas, select the **Prompt for text** action. Then, in the properties pane, under **Other**, change the value of **Allow Interruptions** to **true**.

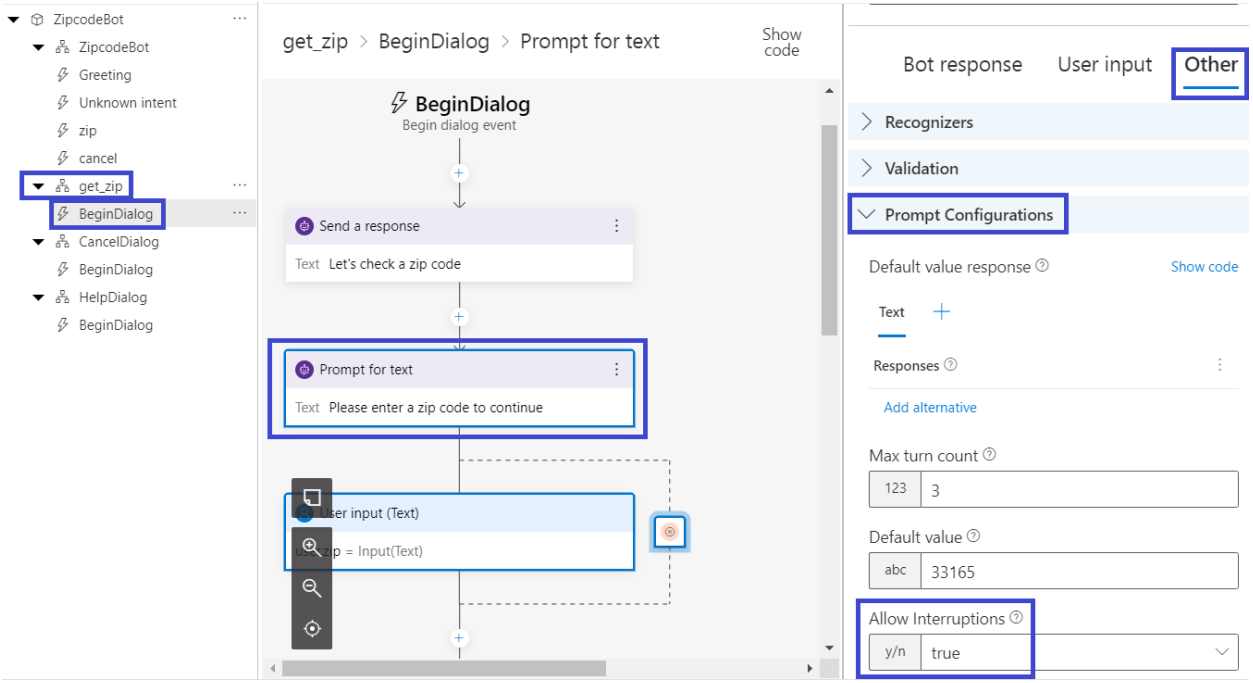

Figure 5-z: Composer – Enabling Interruptions

By enabling interruptions following the steps described within the **get_zip** dialog, the bot can respond to any cancellation requests that the user makes.

Testing interruptions

Let's give this a try. Click the **Start bot** button to execute the bot and start a new conversation. Once the bot starts, click **Open Web Chat**.

Let's begin the conversation by entering the word **zip**. Then, after the bot responds, enter **cancel**, **stop**, or **quit**.

Once you do that, the **CancelDialog** kicks in, and the bot responds by asking if you would like to cancel the conversation. We can see this conversational flow as follows.

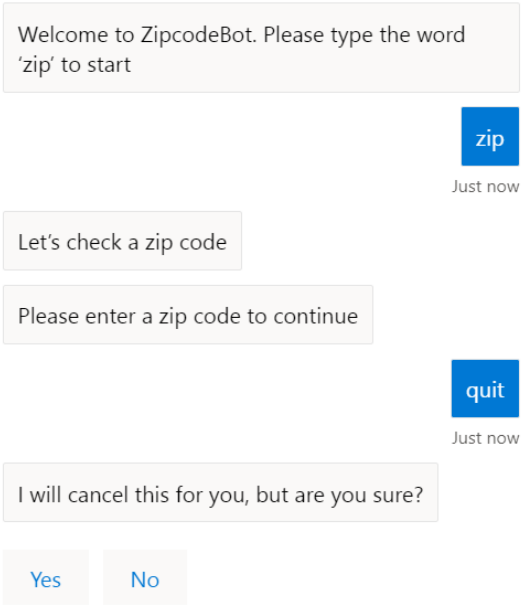

Figure 5-aa: Composer – ZipcodeBot Conversation – Using the CancelDialog (1)

The bot will stop the conversation when the **Yes** button is clicked, and when the **No** button is clicked, the regular conversational flow will continue. I will click the **Yes** button.

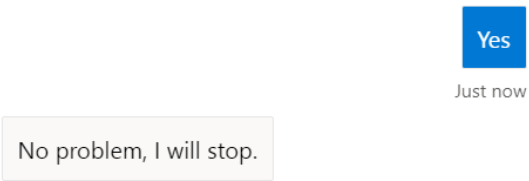

Figure 5-ab: Composer – ZipcodeBot Conversation – Using the CancelDialog (2)

As you can see, the bot responded correctly to the request to stop the conversation. So, by using the **CancelDialog**, we made the bot slightly more intelligent by understanding the intent to abort a conversation.

Nicer output

The last thing I'd like to cover in this chapter is returning the bot's response as a card rather than a single line of text. By doing this, we make the bot's response look better and more professional.

Select the **get_zip** dialog in bot explorer (which is probably already selected). Choose the **Send a response** action on the **True** branch in the authoring canvas.

With that done, go to the properties pane and click **+** > **Attachments**.

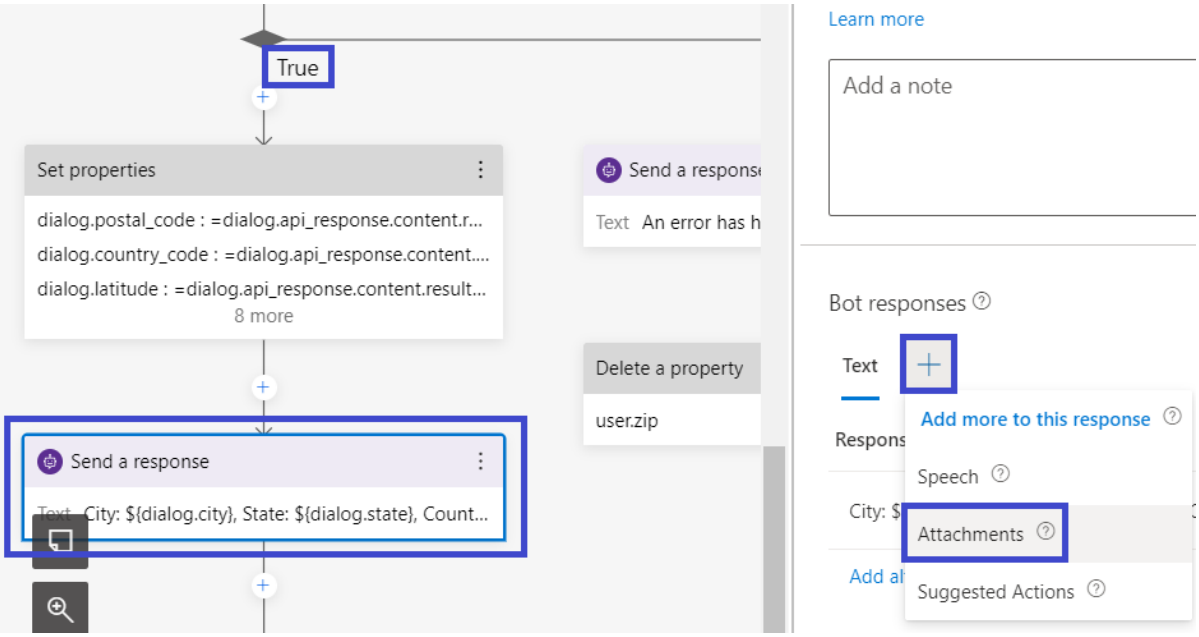

Figure 5-ac: Composer – True Branch – Send a response – Attachments

Next, click **Add new attachment** > **Create from template** > **Adaptive card**.

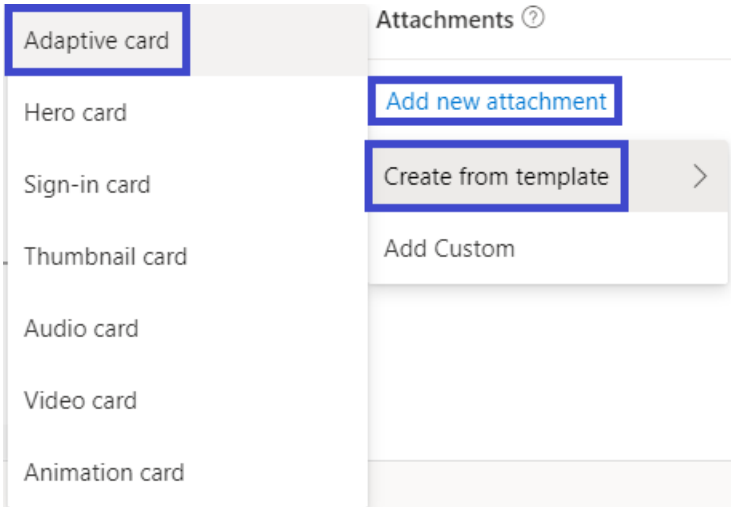

Figure 5-ad: Composer – Add new attachment – Create from template – Adaptive card

In the editor, modify the text field with the following:

```
${user.zip} = ${dialog.city}, ${dialog.state} (${dialog.province} county)
```

Figure 5-ae: Composer – Attachment Editor (Expanded)

With that done, we can restart the bot and test it again. Click the **Restart bot** button, and then **Open Web Chat**.

Figure 5-af: Restart bot Button

Click **Restart conversation – new user ID** to begin fresh. Then, as usual, enter the word **zip**, and enter any valid U.S. zip code. I'll enter **80027**.

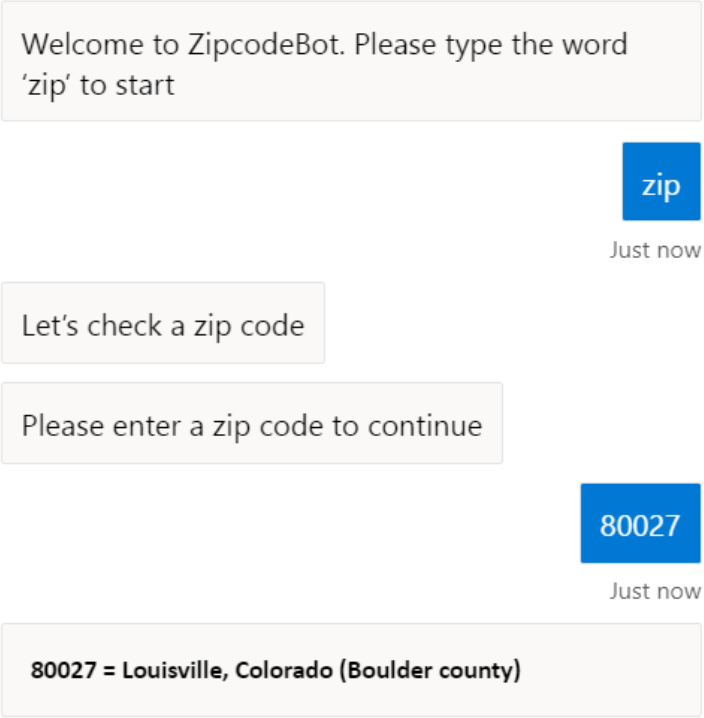

Figure 5-ag: Conversation with an Adaptive Card Response

As you can see, the bot's response is an **adaptive card** rather than a line of text. Although it's not state-of-the-art UI, it's a step forward.

Summary

Throughout this chapter, we went through the necessary steps to finalize our ZipcodeBot by giving it the ability to stop conversations.

Although the bot's functionality is straightforward, the process for creating the bot has been rather effortless. So far, we haven't had to write a single line of code.

That's part of the magic of Composer—to take something as complex as a bot's code and completely abstract it from the person creating the bot.

In the chapter that follows, we'll get to see firsthand all the work (and code) that Composer has created behind the scenes for us, which otherwise we would have written (before Composer).

Furthermore, we'll take that generated code and push it to Azure so we can have a fully functioning bot running in the cloud, using Azure Bot Service.

Chapter 6 Bot Code Structure

Overview

Composer does a lot for us. It acts as an abstraction layer that hides all the underlying code required to create and execute a bot.

Next, we will explore the bot project structure that Composer has bootstrapped and created for us.

Locating the project

If you have Composer running, close and reopen it to locate the folder where Composer has created the bot code. When Composer opens, you will see the name of your bot project and the location on the disk where the project resides.

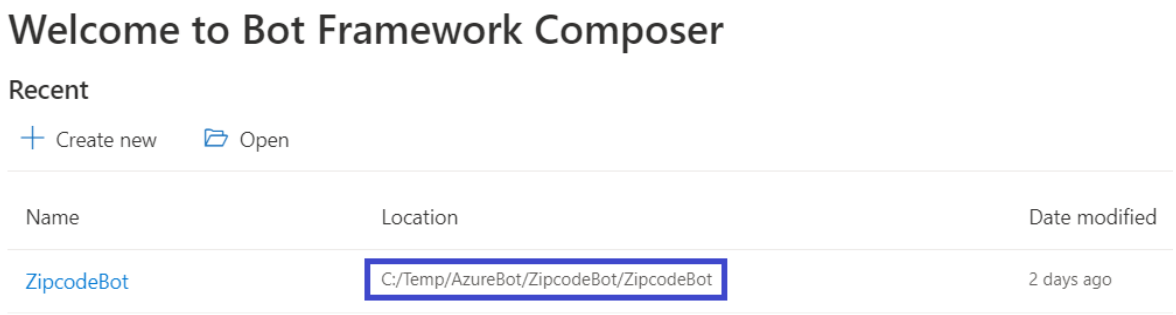

Figure 6-a: Composer – Welcome Screen – Location of the Bot Project

Let's navigate to the project's folder (**Location**) and look at what Composer has created.

Project folder structure

Within the bot project folder, we can find a Visual Studio solution file called **ZipcodeBot.sln**. We can open it with the latest version of Visual Studio (in my case, the 2019 Community Edition).

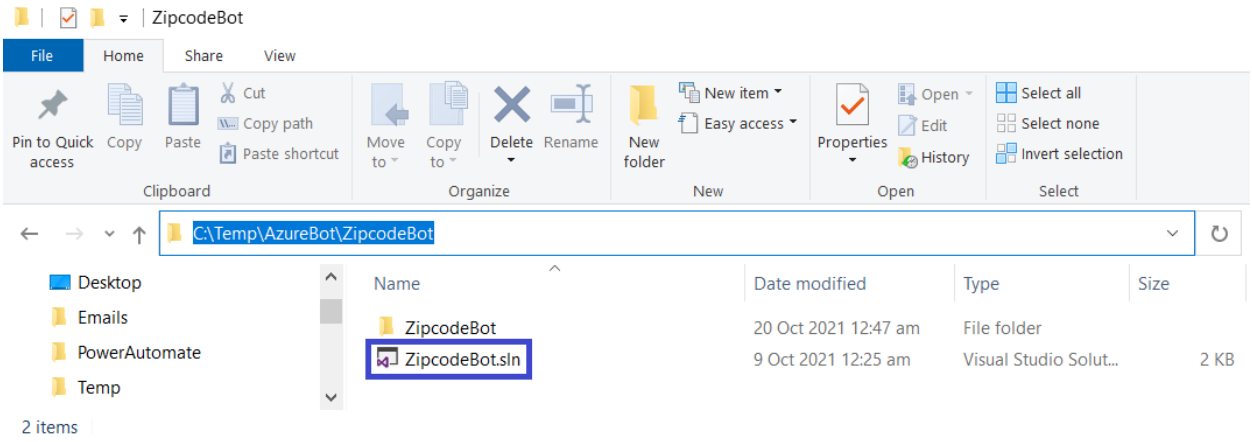

Figure 6-b: Bot Folder Structure and Content

Double-click the **ZipcodeBot.sln** file to open it. Once it is open, go to **Solution Explorer** to look at the project structure and files.

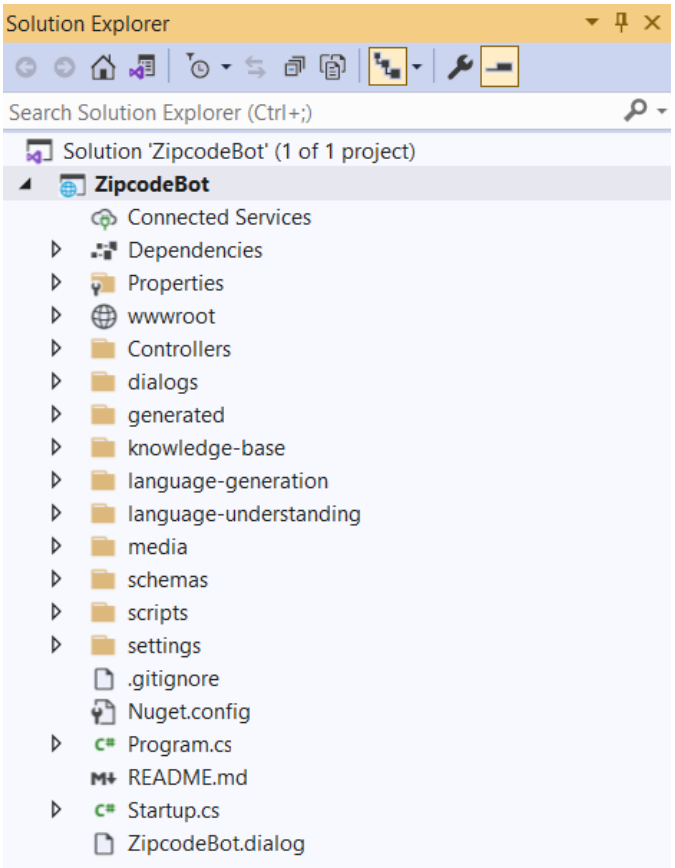

Figure 6-c: Bot Project Structure (Visual Studio)

As we can see, the **ZipcodeBot** project contains many folders, each with many files. If we had to create this project structure manually, this would be a significant task; however, Composer took care of this without us noticing.

We won't go through all the details or look at every folder and file. Let's just look at the most critical folders and files.

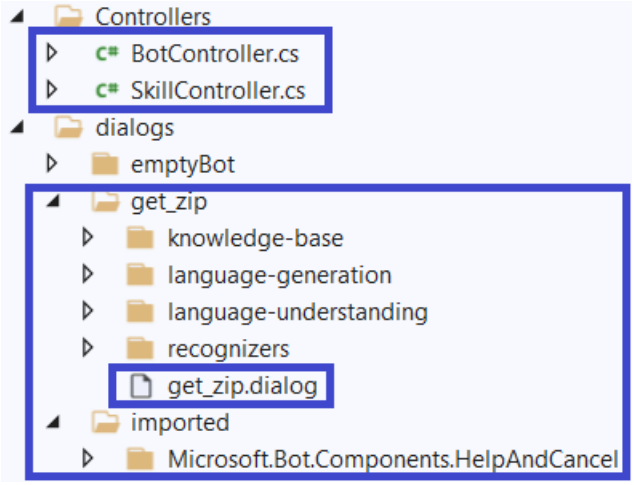

Figure 6-d: Bot Project Structure – Most Important Folders and Files

One of the critical elements of the bot is the controller. The controller—in this case, **BotController.cs**—is the core engine for processing bot requests to the appropriate route. Let's look at this file.

Code Listing 6-a: BotController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Bot.Builder;
using Microsoft.Bot.Builder.Dialogs.Adaptive.Runtime.Settings;
using Microsoft.Bot.Builder.Integration.AspNet.Core;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace ZipcodeBot.Controllers
{
    [ApiController]
    public class BotController : ControllerBase
    {
        private readonly Dictionary<string, IBotFrameworkHttpAdapter>
          _adapters = new Dictionary<string, IBotFrameworkHttpAdapter>();
        private readonly IBot _bot;
        private readonly ILogger<BotController> _logger;

        public BotController(
            IConfiguration configuration,
            IEnumerable<IBotFrameworkHttpAdapter> adapters,
```

```
            IBot bot,
            ILogger<BotController> logger)
        {
            _bot = bot ?? throw new ArgumentNullException(nameof(bot));
            _logger = logger;

            var adapterSettings = configuration.GetSection(
              AdapterSettings.AdapterSettingsKey).
              Get<List<AdapterSettings>>() ??
              new List<AdapterSettings>();
            adapterSettings.Add(AdapterSettings.CoreBotAdapterSettings);

            foreach (var adapter in adapters ??
              throw new ArgumentNullException(nameof(adapters)))
            {
                var settings = adapterSettings.FirstOrDefault(
                s => s.Enabled && s.Type == adapter.GetType().FullName);

                if (settings != null)
                {
                    _adapters.Add(settings.Route, adapter);
                }
            }
        }

        [HttpPost]
        [HttpGet]
        [Route("api/{route}")]
        public async Task PostAsync(string route)
        {
            if (string.IsNullOrEmpty(route))
            {
                _logger.LogError($"PostAsync: No route provided.");
                throw new ArgumentNullException(nameof(route));
            }

            if (_adapters.TryGetValue(route,
              out IBotFrameworkHttpAdapter adapter))
            {
                if (_logger.IsEnabled(LogLevel.Debug))
                {
                    _logger.LogInformation($"PostAsync: routed '{route}'
                    to {adapter.GetType().Name}");
                }

                // Delegating the processing of the HTTP POST to the
                // appropriate adapter.
                // The adapter will invoke the bot.
                await adapter.ProcessAsync(Request,
```

```
                    Response, _bot).ConfigureAwait(false);
            }
            else
            {
                _logger.LogError($"PostAsync: No adapter
                  registered and enabled for route {route}.");
                throw new KeyNotFoundException($"No adapter registered
                  and enabled for route {route}.");
            }
        }
    }
}
```

Without delving into specific details, the **BotController.cs** code has two main functionalities. The first is to initialize BotFrameworkHttpAdapter adapters—this occurs within the **`BotController`** constructor.

The second functionality is to route incoming requests, which the **`Postasync`** method does, by delegating the HTTP request processing to the appropriate adapter.

ZipcodeBot dialog

In **Solution Explorer**, scroll down, and there you'll find the **ZipcodeBot.dialog** file, which contains the bot's main logic and rules.

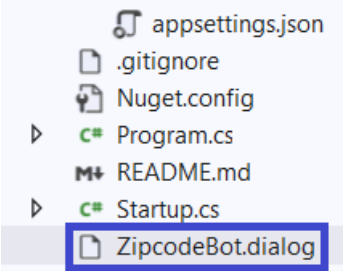

Figure 6-e: The Project Root Files

Behind the scenes, the **ZipcodeBot.dialog** file loads when the bot executes, and this file contains all the main settings that the bot uses to establish the conversation with the user. Let's inspect the contents of this file.

Code Listing 6-b: ZipcodeBot.dialog

```
{
  "$kind": "Microsoft.AdaptiveDialog",
  "$designer": {
```

```
    "name": "ZipcodeBot",
    "description": "",
    "id": "A79tBe"
  },
  "autoEndDialog": true,
  "defaultResultProperty": "dialog.result",
  "triggers": [
    {
      "$kind": "Microsoft.OnConversationUpdateActivity",
      "$designer": {
        "id": "376720"
      },
      "actions": [
        {
          "$kind": "Microsoft.Foreach",
          "$designer": {
            "id": "518944",
            "name": "Loop: for each item"
          },
          "itemsProperty": "turn.Activity.membersAdded",
          "actions": [
            {
              "$kind": "Microsoft.IfCondition",
              "$designer": {
                "id": "641773",
                "name": "Branch: if/else"
              },
              "condition": "string(dialog.foreach.value.id) !=
                            string(turn.Activity.Recipient.id)",
              "actions": [
                {
                  "$kind": "Microsoft.SendActivity",
                  "$designer": {
                    "id": "859266",
                    "name": "Send a response"
                  },
                  "activity": "${SendActivity_Greeting()}"
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "$kind": "Microsoft.OnUnknownIntent",
      "$designer": {
        "id": "mb2n1u"
      },
```

```
      "actions": [
        {
          "$kind": "Microsoft.SendActivity",
          "$designer": {
            "id": "kMjqz1"
          },
          "activity": "${SendActivity_DidNotUnderstand()}"
        }
      ]
    },
    {
      "$kind": "Microsoft.OnIntent",
      "$designer": {
        "id": "d5ER8p",
        "name": "zip"
      },
      "intent": "zip",
      "actions": [
        {
          "$kind": "Microsoft.BeginDialog",
          "$designer": {
            "id": "uvw6RC"
          },
          "activityProcessed": true,
          "dialog": "get_zip"
        }
      ]
    },
    {
      "$kind": "Microsoft.OnIntent",
      "$designer": {
        "id": "RkkXZi",
        "name": "cancel"
      },
      "intent": "cancel",
      "actions": [
        {
          "$kind": "Microsoft.BeginDialog",
          "$designer": {
            "id": "KM4gcW"
          },
          "activityProcessed": true,
          "dialog": "CancelDialog"
        }
      ]
    }
  ],
  "generator": "ZipcodeBot.lg",
  "id": "ZipcodeBot",
```

```
  "recognizer": {
    "$kind": "Microsoft.RegexRecognizer",
    "intents": [
      {
        "intent": "zip",
        "pattern": "zip"
      },
      {
        "intent": "cancel",
        "pattern": "stop|quit|cancel"
      }
    ]
  }
}
```

Essentially, we can see that most of the dialog details we created using Composer are here. Notice that in one of the actions, there's a reference to the **get_zip** dialog. There are references to the **Branch: if/else** and also the **cancel** dialog.

The get_zip dialog

Going back to **Solution Explorer**, double-click the **get_zip** file to open and inspect it.

Code Listing 6-c: get_zip.dialog

```
{
  "$kind": "Microsoft.AdaptiveDialog",
  "$designer": {
    "id": "dBaQjz",
    "name": "get_zip",
    "comment": "Get the zip code"
  },
  "autoEndDialog": true,
  "defaultResultProperty": "dialog.result",
  "triggers": [
    {
      "$kind": "Microsoft.OnBeginDialog",
      "$designer": {
        "name": "BeginDialog",
        "description": "",
        "id": "pCF5nd"
      },
      "actions": [
        {
          "$kind": "Microsoft.SendActivity",
          "$designer": {
            "id": "J7llQd"
```

```
        },
        "activity": "${SendActivity_J7llQd()}"
      },
      {
        "$kind": "Microsoft.TextInput",
        "$designer": {
          "id": "qJM9qX"
        },
        "disabled": false,
        "maxTurnCount": 3,
        "alwaysPrompt": false,
        "allowInterruptions": true,
        "prompt": "${TextInput_Prompt_qJM9qX()}",
        "unrecognizedPrompt":
          "${TextInput_UnrecognizedPrompt_qJM9qX()}",
        "property": "user.zip",
        "outputFormat": "=trim(this.value)",
        "validations": [
          "=length(this.value) == 5"
        ],
        "invalidPrompt": "${TextInput_InvalidPrompt_qJM9qX()}",
        "defaultValue": "33165"
      },
      {
        "$kind": "Microsoft.HttpRequest",
        "$designer": {
          "id": "itxd2e"
        },
        "method": "GET",
        "url": "https://app.zipcodebase.com/api/v1/search?
                apikey=03b1edc0-2883-11ec-8073-777a767aae15
                &codes=${user.zip}&country=US",
        "headers": {},
        "resultProperty": "dialog.api_response",
        "responseType": "json"
      },
      {
        "$kind": "Microsoft.IfCondition",
        "$designer": {
          "id": "kOFBrL"
        },
        "condition": "=dialog.api_response.statusCode == 200",
        "actions": [
          {
            "$kind": "Microsoft.SetProperties",
            "$designer": {
              "id": "KhWRrn"
            },
            "assignments": [
```

```
                {
                  "property": "dialog.postal_code",
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['postal_code']"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['country_code']",
                  "property": "dialog.country_code"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['latitude'] ",
                  "property": "dialog.latitude"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['longitude']",
                  "property": "dialog.longitude"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['city']",
                  "property": "dialog.city"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['state']",
                  "property": "dialog.state"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['city_en']",
                  "property": "dialog.city_en"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['state_en']",
                  "property": "dialog.state_en"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['state_code']",
                  "property": "dialog.state_code"
                },
                {
                  "value": "=dialog.api_response.content.
                            results[user.zip][0]['province']",
                  "property": "dialog.province"
```

```
                },
                {
                  "value": "=dialog.api_response.content.
                             results[user.zip][0]['province_code']",
                  "property": "dialog.province_code"
                }
              ]
            },
            {
              "$kind": "Microsoft.SendActivity",
              "$designer": {
                "id": "Qmqx7A"
              },
              "activity": "${SendActivity_Qmqx7A()}"
            }
          ],
          "elseActions": [
            {
              "$kind": "Microsoft.SendActivity",
              "$designer": {
                "id": "tFVM0T"
              },
              "activity": "${SendActivity_tFVM0T()}"
            },
            {
              "$kind": "Microsoft.DeleteProperty",
              "$designer": {
                "id": "NSO8CP"
              },
              "property": "user.zip"
            }
          ]
        }
      ]
    }
  ],
  "generator": "get_zip.lg",
  "recognizer": "get_zip.lu.qna",
  "id": "get_zip"
}
```

As you might have noticed, the highlights of this file are the HTTP request to the Zipcodebase API and the property assignments from the results obtained from the API.

Although this file is easy to read, its beauty is that Composer has abstracted the creation of this definition file by providing us with a pleasant and easy-to-use UI.

appsettings.json

Another important file for the bot to function correctly is **appsettings.json**. This file contains settings and definitions on how the project executes. Within **Solution Explorer**, under the **settings** folder, double-click the file to open it.

The most critical section is the **runtime** part, and as you will see in the listing that follows, it contains instructions on how to execute the project using the .NET Core **`run`** command.

You can verify this by right-clicking **ZipcodeBot** within **Solution Explorer**, then clicking **Properties**.

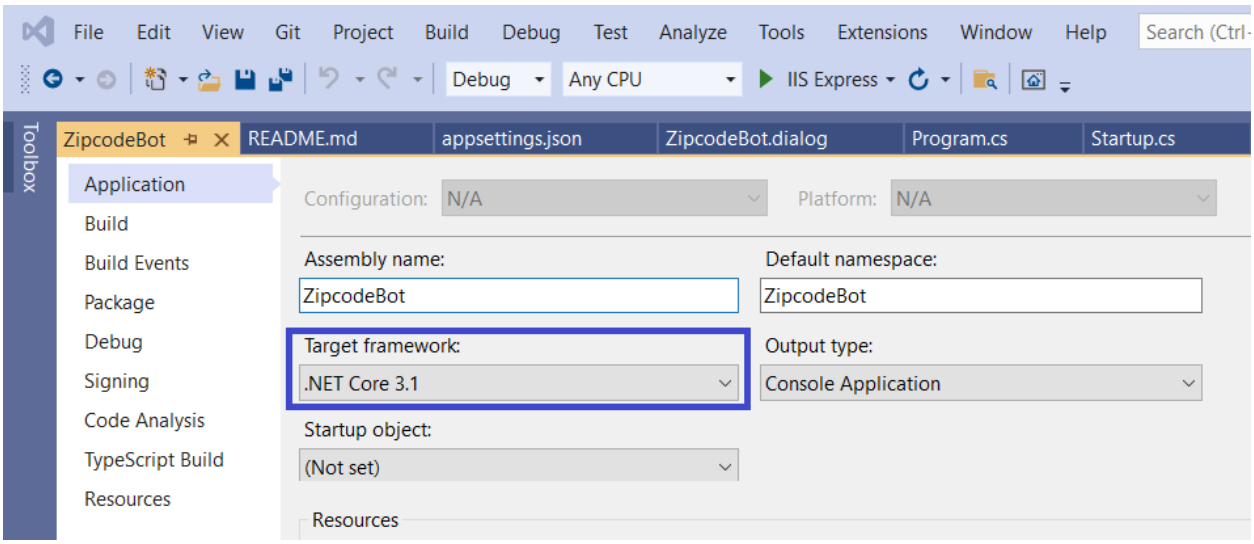

Figure 6-f: ZipcodeBot Properties – Visual Studio

Here is the code for **appSettings.json**.

Code Listing 6-d: appsettings.json

```
{
  "customFunctions": [],
  "defaultLanguage": "en-us",
  "defaultLocale": "en-us",
  "importedLibraries": [],
  "languages": [
    "en-us"
  ],
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "luFeatures": {
```

```
    "enableCompositeEntities": true,
    "enableListEntities": true,
    "enableMLEntities": true,
    "enablePattern": true,
    "enablePhraseLists": true,
    "enablePrebuiltEntities": true,
    "enableRegexEntities": true
  },
  "luis": {
    "authoringEndpoint": "",
    "authoringRegion": "",
    "defaultLanguage": "en-us",
    "endpoint": "",
    "environment": "composer",
    "name": "ZipcodeBot"
  },
  "MicrosoftAppId": "",
  "publishTargets": [],
  "qna": {
    "hostname": "",
    "knowledgebaseid": "",
    "qnaRegion": "westus"
  },
  "runtime": {
    "command": "dotnet run --project ZipcodeBot.csproj",
    "customRuntime": true,
    "key": "adaptive-runtime-dotnet-webapp",
    "path": "../"
  },
  "runtimeSettings": {
    "adapters": [],
    "features": {
      "removeRecipientMentions": false,
      "showTyping": false,
      "traceTranscript": false,
      "useInspection": false,
      "setSpeak": {
        "voiceFontName": "en-US-JennyNeural",
        "fallbackToTextForSpeechIfEmpty": true
      }
    },
    "components": [],
    "skills": {
      "allowedCallers": []
    },
    "storage": "",
    "telemetry": {
      "logActivities": true,
      "logPersonalInformation": false,
```

```
      "options": {
        "connectionString": ""
      }
    }
  },
  "skillConfiguration": {},
  "skillHostEndpoint": "http://localhost:3980/api/skills"
}
```

Summary

These **.dialog** files are the essence of the bot and contain its core logic. The C# code-behind then parses the content of both files and executes the bot.

Now that we understand what lies behind the scenes, we are ready to publish our bot to the Azure Bot Service, which we'll do in the next and final chapter.

Chapter 7 Publishing the Bot

Overview

By going through all the steps explained throughout the book's previous chapters, we have managed to create a small functional bot and looked at how some of its most critical parts work. Our focus now will be on how to deploy and publish it to Azure Bot Service.

Prerequisites

To deploy and publish a bot, the following prerequisites are required:

- Microsoft Azure subscription.
- Node.js 12.13.0 or later.
- The latest version of the Azure CLI.
- PowerShell 6.0 or later.

Azure Portal

Before we can deploy the bot to Azure, we need to have an account. If you don't have an Azure account, getting one is very easy—you can create a free account here if you don't have one.

Figure 7-a: Microsoft Azure Website

Once you have set up an Azure account, you can log in by going to the Portal. Before deploying and publishing the bot, you must ensure that your Azure subscription is registered to use **Microsoft.BotService**—this is known as resource provider registration.

Resource provider registration

The official Azure documentation provides comprehensive information on how to resolve errors for resource provider registration.

The **Microsoft.BotService** namespace is an essential requirement that must be registered to your Azure subscription for Composer to deploy and publish your bot successfully.

Therefore, you must follow the steps described in the documentation (specifically on **Solution 3 - Azure Portal**). You can do this by checking your Azure subscription properties, then in **Resource providers**, look for **Microsoft.BotService** and ensure that this provider is registered (if it isn't, click **Register**).

The following figure shows how the provider appears on my Azure subscription (called **Visual Studio Dev Essentials**—your subscription might have a different name). You might find that this provider is not registered; if so, you'll have to register it.

Whether the **Microsoft.BotService** provider appears as registered by default might be dependent on the type of Azure subscription you have purchased.

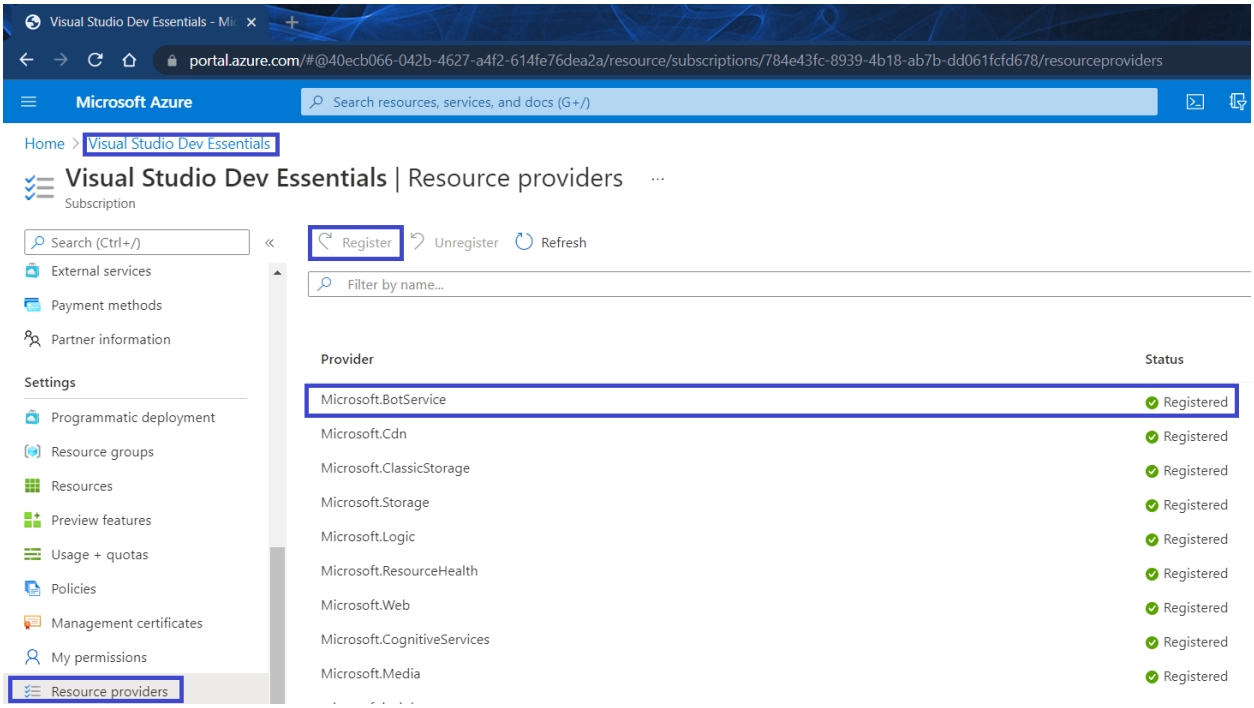

Figure 7-b: Azure Portal – Subscription – Resource providers – Microsoft.BotService

Deploying from Composer

With Composer and your bot open, click the **Publish** icon on the navigation pane as shown in the following figure.

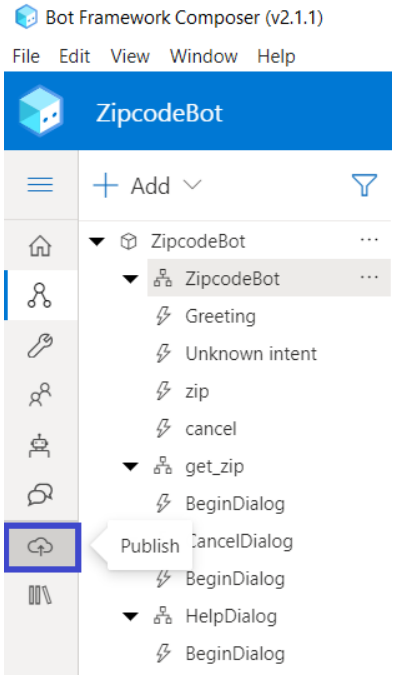

Figure 7-c: Composer – Publish Icon – Navigation Pane

Then, select **ZipcodeBot**.

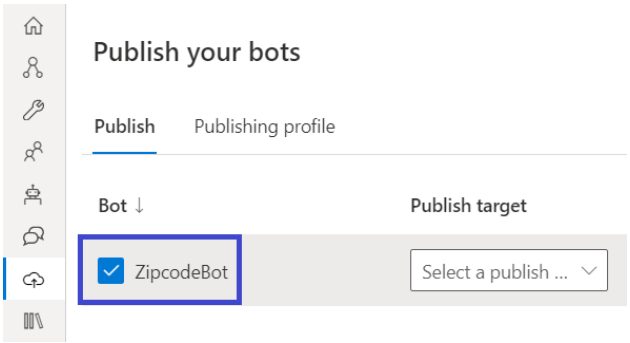

Figure 7-d: Composer – Publish your bots – ZipcodeBot

After that, we need to define a **Publish target**, so click the dropdown option, **Manage profiles**, or the **Publishing profile** tab directly.

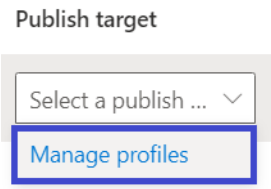

Figure 7-e: Composer – Publish your bots / Publish target / Manage profiles

Next, click **Add new**.

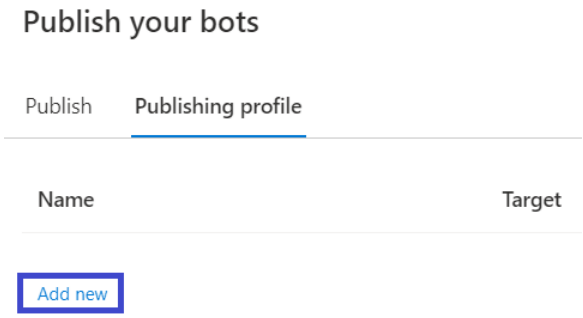

Figure 7-f: Composer – Publishing profile – Add new

The **Create a publishing profile** pop-up window appears. Here we need to enter the profile name—in this case, **ZipcodeBot**, and as the **Publishing target**, choose **Publish bot to Azure**.

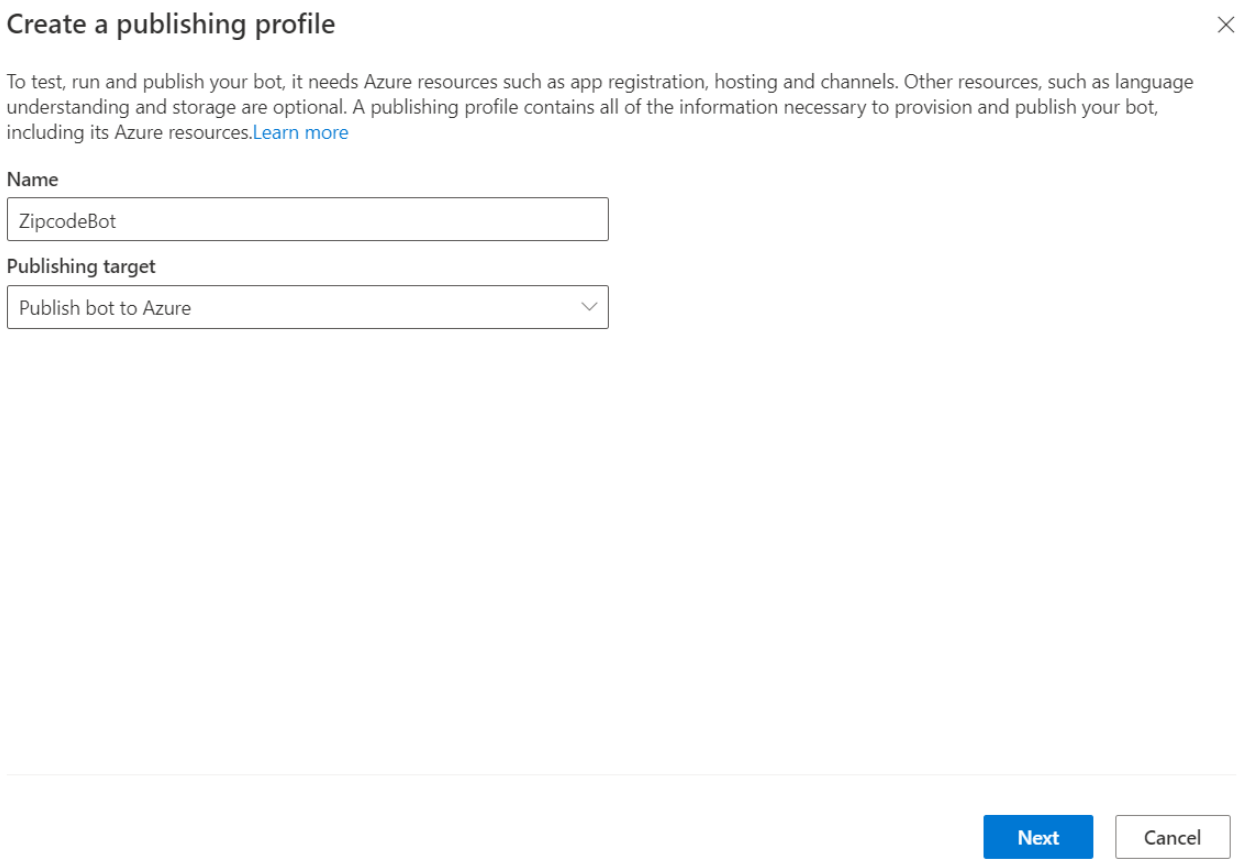

Figure 7-g: Composer – Create a publishing profile (1)

Click **Next**, and you'll see the following options. Let's select **Create new resources** > **Next**.

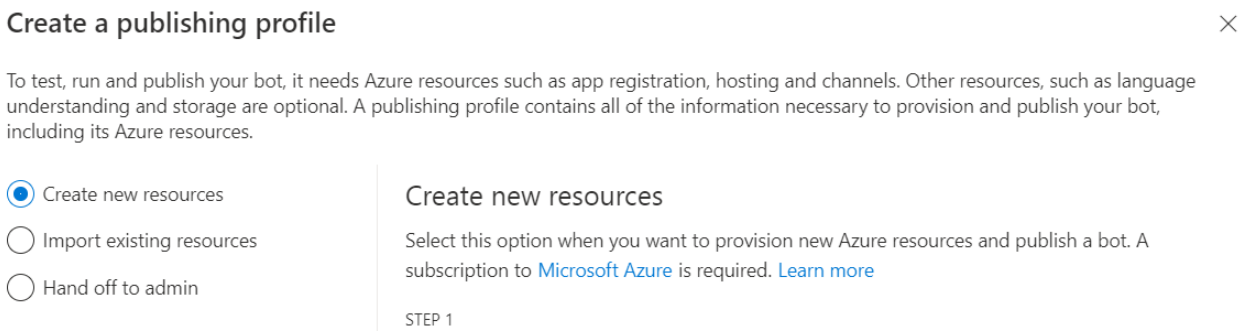

Figure 7-h: Composer – Create a publishing profile (2)

A pop-up window appears, prompting you to sign in to Azure.

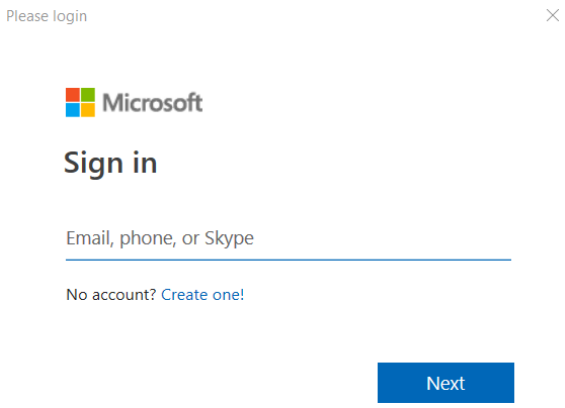

Figure 7-i: Composer – Azure Sign In

There, enter the email address that you used when you signed up for Azure, and click **Next**. After that, enter the password, and click **Sign in**.

We'll have to enter the Azure and resource details. Select your Azure **Subscription**. As for the **Resource group**, select **Create new**, and give it the name **bots**.

For the **Operating System**, leave the default option selected: **Windows (Recommended)**.

For the **Resource details**, set the **Name** field to **thezipcodebot** (feel free to give it another name).

As for the **Region** set to **West US** and the **LUIS region** set to **West US**, feel free to choose any others if you wish; these are not mandatory.

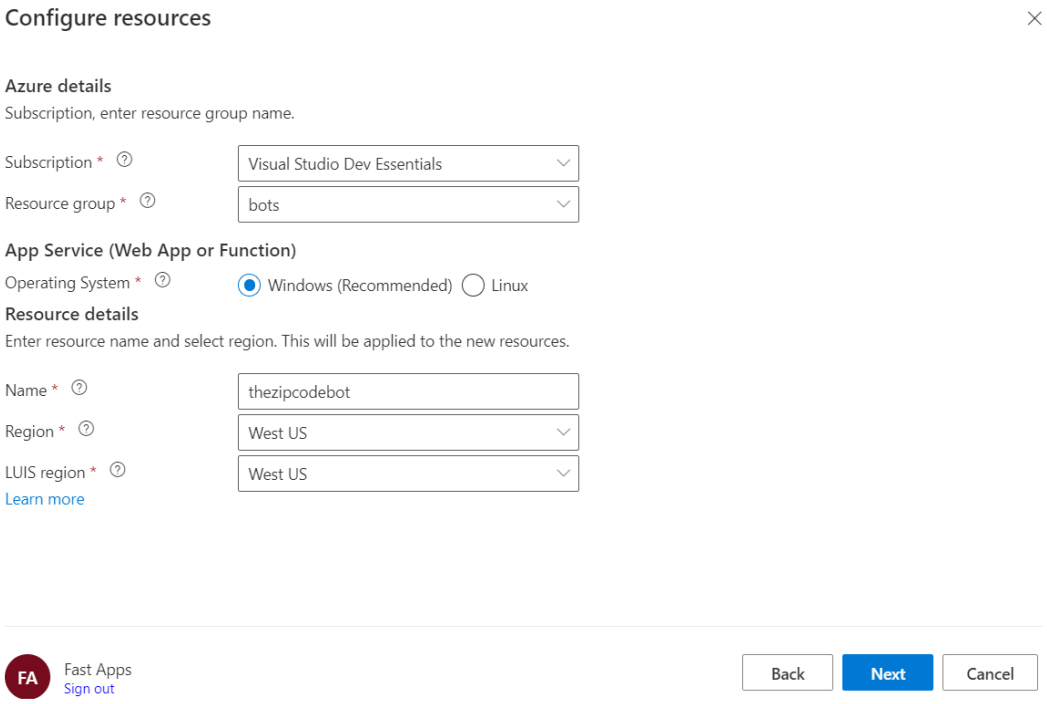

Figure 7-j: Composer – Configure resources

After entering those details, click **Next** to continue. You will see the list of Azure resources required for the bot to run.

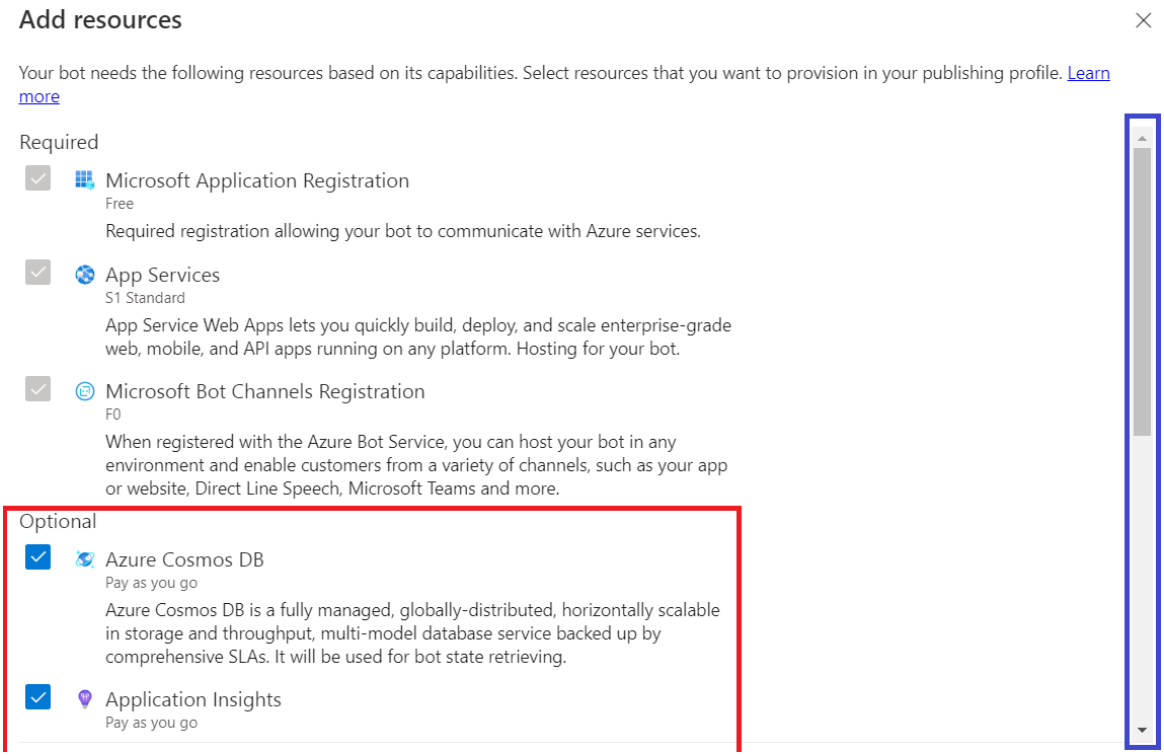

Figure 7-k: Composer – Add resources

As for the resources selected by default as **Optional**, such as **Azure Cosmos DB** or **Application Insights**, I would recommend unselecting them all (except the two LUIS services) to avoid incurring unnecessary Azure costs.

Check the complete list of **Optional** resources (from top to bottom) and remove all the ones selected by default (except the two LUIS services).

If you want to experiment with LUIS, the Azure Language Understanding service, you must leave the **Microsoft Language Understanding Authoring Account** and **Prediction Account** resources selected.

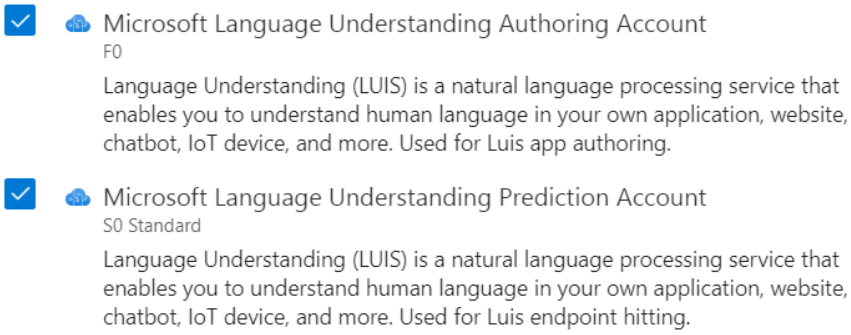

Figure 7-l: Composer – Add resources – LUIS

We will not cover LUIS in this book; however, I encourage you to explore the official LUIS documentation and how you can use and integrate this functionality into your bot.

For my bot deployment, I'm going to leave the **Microsoft Language Understanding Authoring Account** and **Prediction Account** resources selected.

After unselecting the resources that you won't use, click **Next** to continue.

Review resources to be created

The following resources will be created and provisioned for your bot. Once provisioned, they will be available in the Azure portal.

Resource Type	Resource Group	Name	Region
Microsoft Application Registration	bots	thezipcodebot	global
App Services	bots	thezipcodebot	westus
Microsoft Bot Channels Registration	bots	thezipcodebot	global
Microsoft Language Understanding Authoring Account	bots	thezipcodebot-luis-authoring	westus
Microsoft Language Understanding Prediction Account	bots	thezipcodebot-luis	westus

Figure 7-m: Composer – Review resources to be created

Next, you'll see a screen with the list of resources that the bot will require at this stage. Click **Create** to continue (the **Create** button is not visible in Figure 7-m, but it exists in the application).

Composer will provision the required Azure resources to deploy and publish the bot.

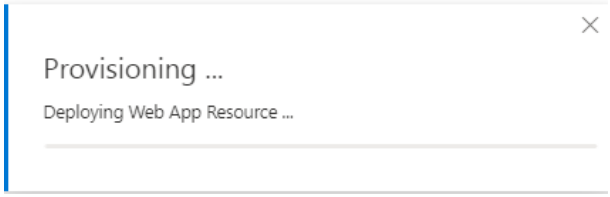

Figure 7-n: Composer – Provisioning Azure Bot Resources

Once the operation finalizes, you will see the following message.

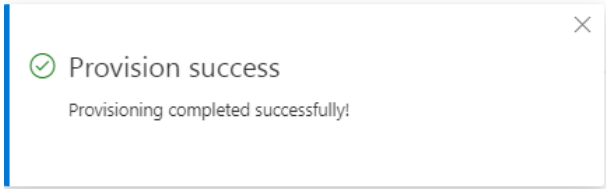

Figure 7-o: Composer – Provision success

Checking Azure resources

Let's open a web browser, go to the Azure Portal, and click **All resources** to view the resources created by Composer.

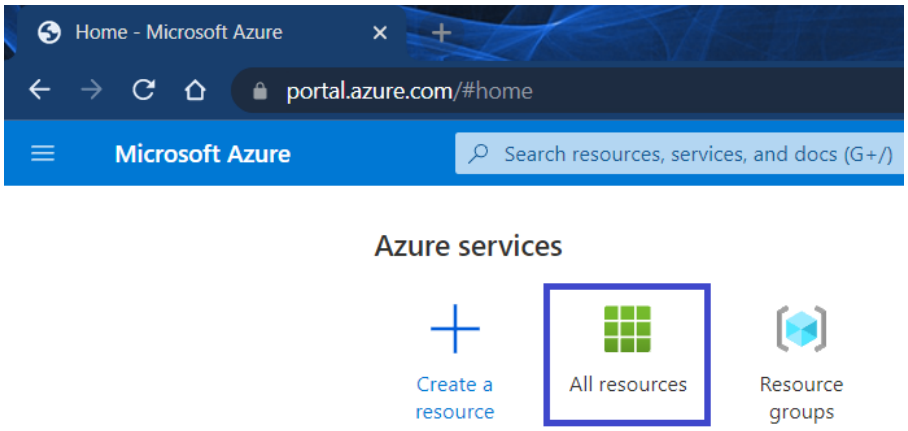

Figure 7-p: Azure Portal – Main Page

After clicking **All resources**, you will see the **App Service Plan** and the **App Service**, among others.

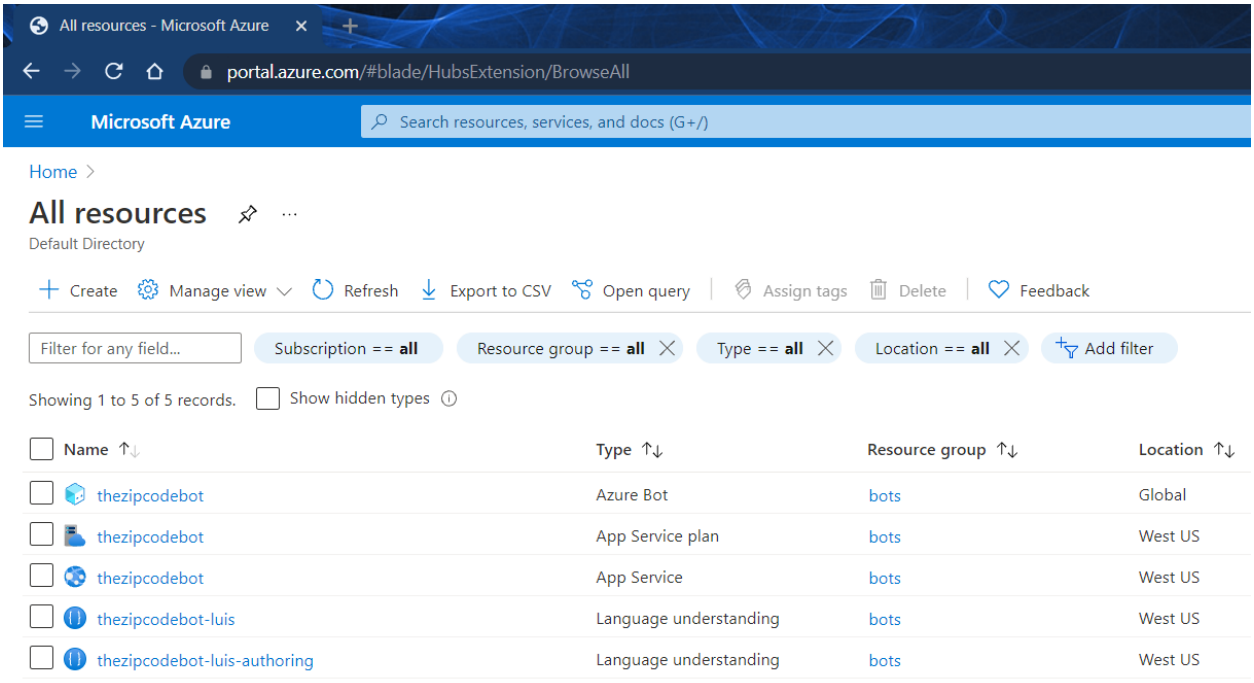

Figure 7-q: Azure Portal – All resources

Publishing

Back in **Composer**, go to the **Publish** tab and set the bot's **Publish target** as **ZipcodeBot**. Then, click **Publish selected bots**.

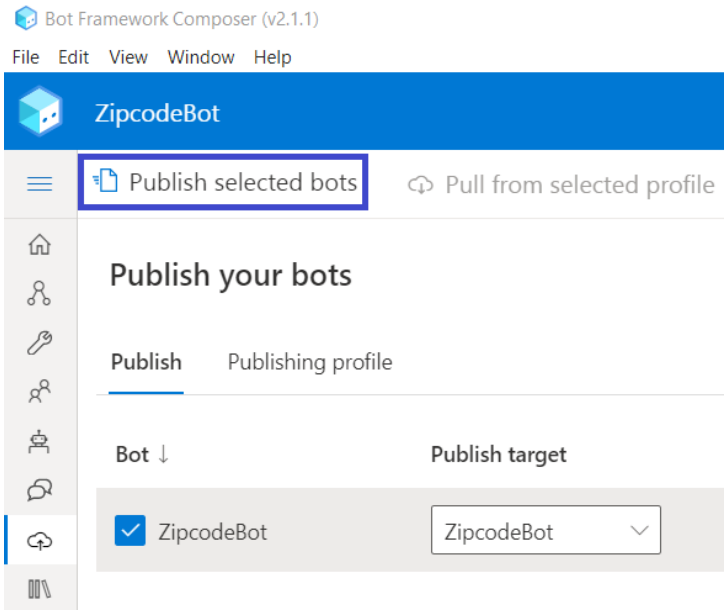

Figure 7-r: Composer – Publish selected bots

After clicking **Published selected bots**, you will see the following pop-up window.

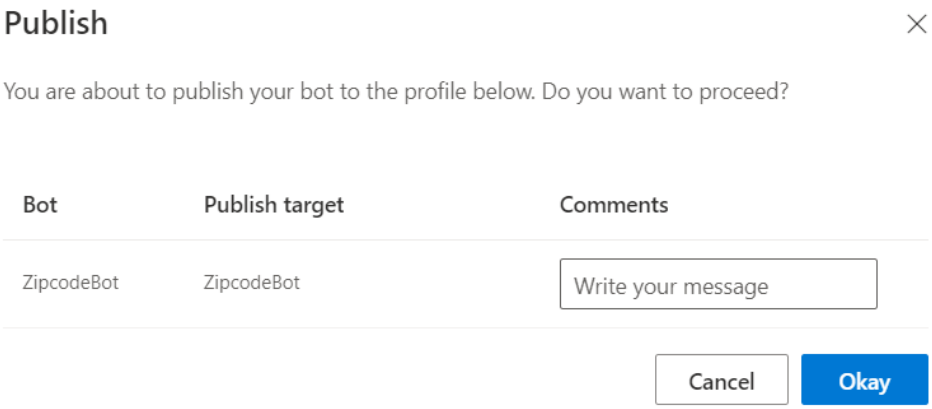

Figure 7-s: Composer – Publish

To publish the bot, click **Okay**—you will see the following message.

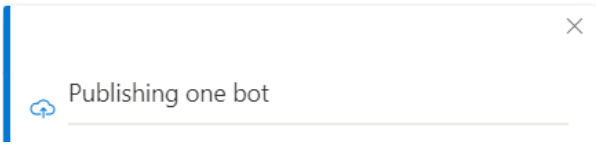

Figure 7-t: Composer – Publishing one bot

The deployment and publishing process might take a few minutes depending on your internet connection speed.

You will also see message updates depending on what stage of the process the deployment and publishing are, such as **Creating build artifact…**.

Bot ↓	Publish target	Date	Status	Message
ZipcodeBot	ZipcodeBot	10-23-2021		Creating build artifact...

Figure 7-u: Composer – Publishing one bot – Creating build artifact

Once you're done, you will see the following **Success** message.

Bot ↓	Publish target	Date	Status	Message
ZipcodeBot	ZipcodeBot	10-23-2021	✓	Success

Figure 7-v: Composer – Publishing one bot – Success

Testing the Azure bot

With the bot successfully deployed and published, the next thing to do is test it. Let's go back to **Azure Portal**, then click **All resources**.

In my case, the Azure Bot Service is the first item on my list of resources. Make sure you click the element that has the column **Type** value set to **Azure Bot**.

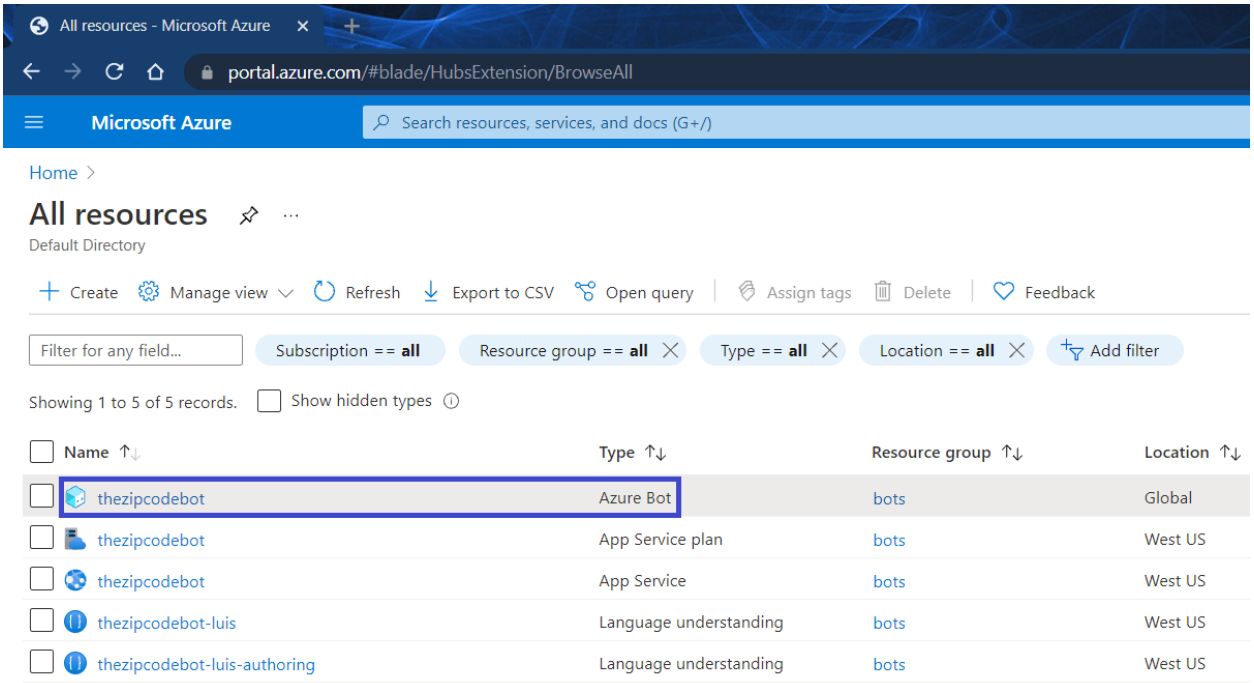

Figure 7-w: Azure Portal – All resources – Azure Bot Highlighted

After you click the Azure Bot item, you'll see the Azure Bot blade. Ensure your browser has cookies enabled (indicated by the small eye icon highlighted in **red** in the upper-right side of Figure 7-x)—which most browsers do by default; however, some cookies might not be allowed.

Click **Test in Web Chat** and start a conversation with the bot. As you can see in the following figure, the bot works just great, the same as it did in Composer.

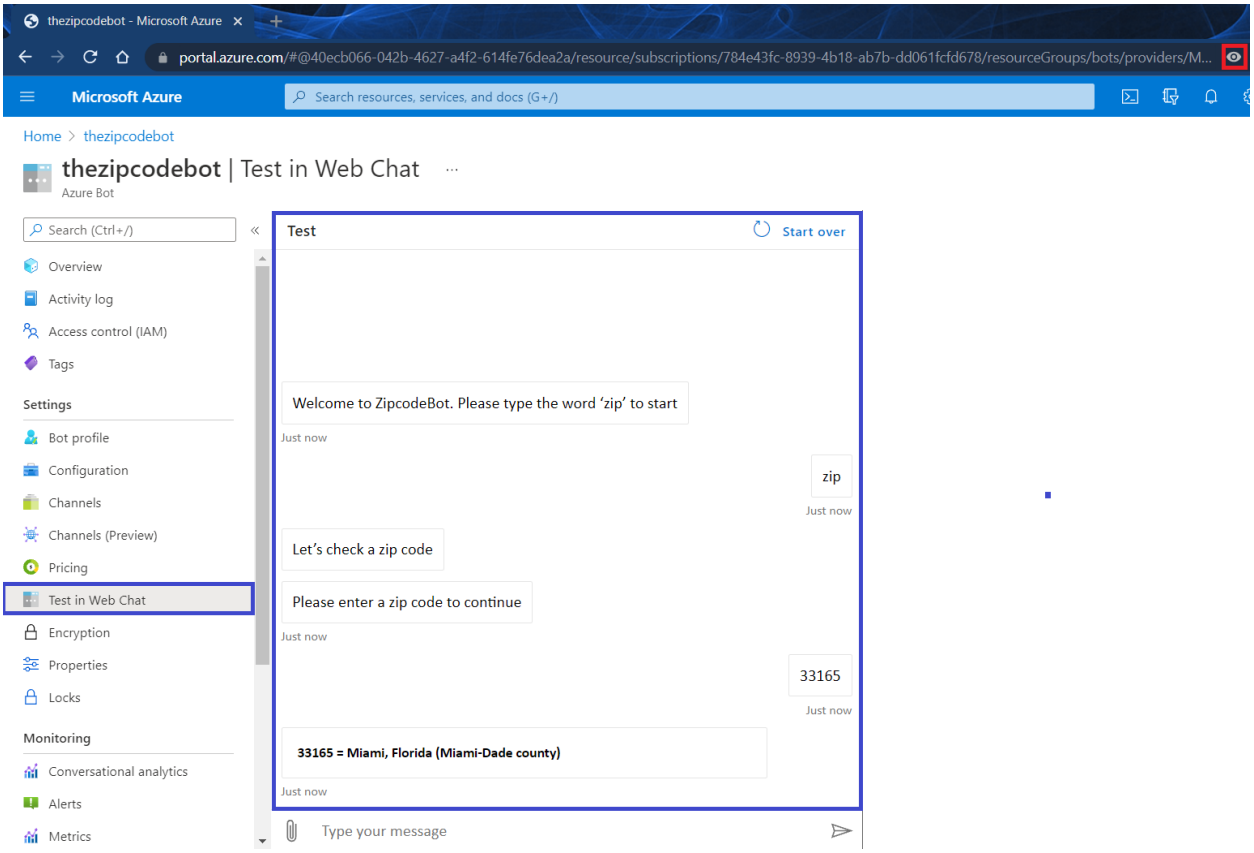

Figure 7-x: Azure Portal – Azure Bot Service – Test in Web Chat

Excellent—we have achieved our goal!

Closing thoughts

Although we have created a basic bot that returns information about zip codes using a third-party API through the course of this book, we got a working and fully functional bot deployed to the Azure cloud without a single line of code. If that's not impressive, I don't know what is!

The other exciting aspect of this tale is that we could have added more sophisticated features to the bot without writing any code, such as by using LUIS.

In my opinion, combining Composer and Azure Bot Service is powerful, yet easy to understand and do.

From the initial days of the Microsoft Bot Framework—reserved mostly for highly qualified C# developers—creating bots with Microsoft technologies has come a long way.

These technologies now empower non-professional developers (also known as citizen developers) to create compelling bots without any coding knowledge, which interests businesses.

I invite you to keep exploring what you can do with Composer and Azure Bot Service—we've barely scratched the surface of what's possible.

Consider adding LUIS-enabled capabilities to your bot so that it can understand natural language intents or use more articulated dialogs. The possibilities are endless.

A great way to continue your journey is to explore the official Microsoft documentation, look at some of the samples available, and add your ideas to the mix. If you come up with something cool, I'd love to hear about it.

I hope this book has given you some valuable pointers that you can take on that journey. Thanks for reading—until next time, take care, keep learning, and have fun!