SKYPE BOTS

SUCCINCTLY

BY ED FREITAS

Syncfusion®

Skype Bots Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

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

Proofreader: John Elderkin

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

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

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

Information is plentiful but harder to digest

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

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

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

The *Succinctly* series

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

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

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

The best authors, the best content

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

Free forever

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

Free? What is the catch?

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

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

Let us know what you think

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

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

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

About the Author

Ed Freitas currently works as a consultant on Software Development applied to Customer Success, mostly related to Financial Process Automation, Accounts Payable Processing, Invoice Data Extraction, and SAP Integration.

He has provided consultancy services and engineered, advised, and supported various projects for global names such as Agfa, Coca Cola, Domestic & General, EY, Enel, Mango, and the Social Security Agency, among many others.

He's also been invited to various companies such as Shell, Capgemini, Cognizant, and the European Space Agency.

He was recently involved in analyzing 1.6 billion rows of data using Redshift (Amazon Web Services) in order to gather valuable insights on client patterns. He holds an M.S. in Computer Science.

He enjoys soccer, running, traveling, life hacking, learning, and spending time with his family. You can reach him at http://edfreitas.me.

Acknowledgements

Introduction

[Skype for Developers](#) is a set of SDKs (software development kits) that allow developers to become part of three billion minutes of daily conversations, as pronounced by Microsoft, and develop on top of an open and very extensible platform, which is Skype. Can you imagine a world without Skype?

There are several SDKs within this vast ecosystem that allow developers to interact with Skype as a platform.

One of the SDKs is Bots, which is powered by the [Microsoft Bot Framework,](#) and I've written a complete [book](#) on the subject for the Succinctly series. Bots provide a way to interact with users in personal conversations. This SDK is by far the most important one when developing for Skype.

Another SDK is Web Control, which provides a way to easily embed a Skype Bot or Skype chat canvas into a website or web application.

Payments is another SDK that is currently in developer preview mode. It allows developers to easily integrate payment capabilities directly into Skype chats, enabling users to purchase products and services.

Finally, add-ins—also in developer preview mode at the time of writing of this book—allow developers to integrate apps directly into Skype conversations. These add-ins help users find and share content while using tools directly within a conversation without needing to switch to another app.

It is also possible to leverage many of these features on Skype for Business and on Microsoft Teams.

The most exciting and cutting edge of these SDKs is Bots. Throughout this book, we'll explore in-depth features of the Bot Framework for Skype that weren't covered in the *Microsoft Bot Framework Succinctly* book, such as Language Understanding Intelligence Service (LUIS) and QnA Maker Service. These will be the subjects we will focus on throughout this book.

In order to make the most of this book, it's good to have some knowledge of C#. We'll be using [Visual Studio](#) 2017 Professional.

By the end of this book, you should have a good foundation on how to develop bots specifically for Skype using LUIS, QnA Maker Service, and Scorables—and hopefully use them to add value to your customers or business processes.

The examples should be fun and easy to follow, and give you a good sense of what is possible with these amazing technologies. I hope you have fun!

Chapter 1 Regex Bots

Quick intro

Bots are a great way to interact with users in Skype or Skype for Business. Bots are conversational agents that users can interact with—to some extent—as though conversing with a person.

In my e-book *Microsoft Bot Framework Succinctly*, I cover most of the aspects of creating and working with bots along with publishing a bot to Skype.

Even though you should be able to fully understand this chapter fairly easily, you might want to take a quick look at that e-book as a way to gain some familiarity with the Bot Framework and understand the basic foundational blocks for building bots, which won't be completely covered in this chapter.

Instead, we'll focus our attention on aspects of bots that were not discussed in *Microsoft Bot Framework Succinctly*. So, let's get started.

The full source code for this chapter can be found here.

Setting up our bot project

In order to get started creating our Cake Bot, we need to first install the Bot Template for Visual Studio.

Once you have the template downloaded, unzip it and place it under this folder: **%USERPROFILE%\Documents\Visual Studio 2017\Templates\ProjectTemplates\Visual C#**. This will display a Bot Application template, which we'll use to create our bot.

First, let's fire-up Visual Studio 2017, and let's create our new bot by using the Bot Application template. You can do this by opening the New Project window as follows: **File** > **New** > **Project**.
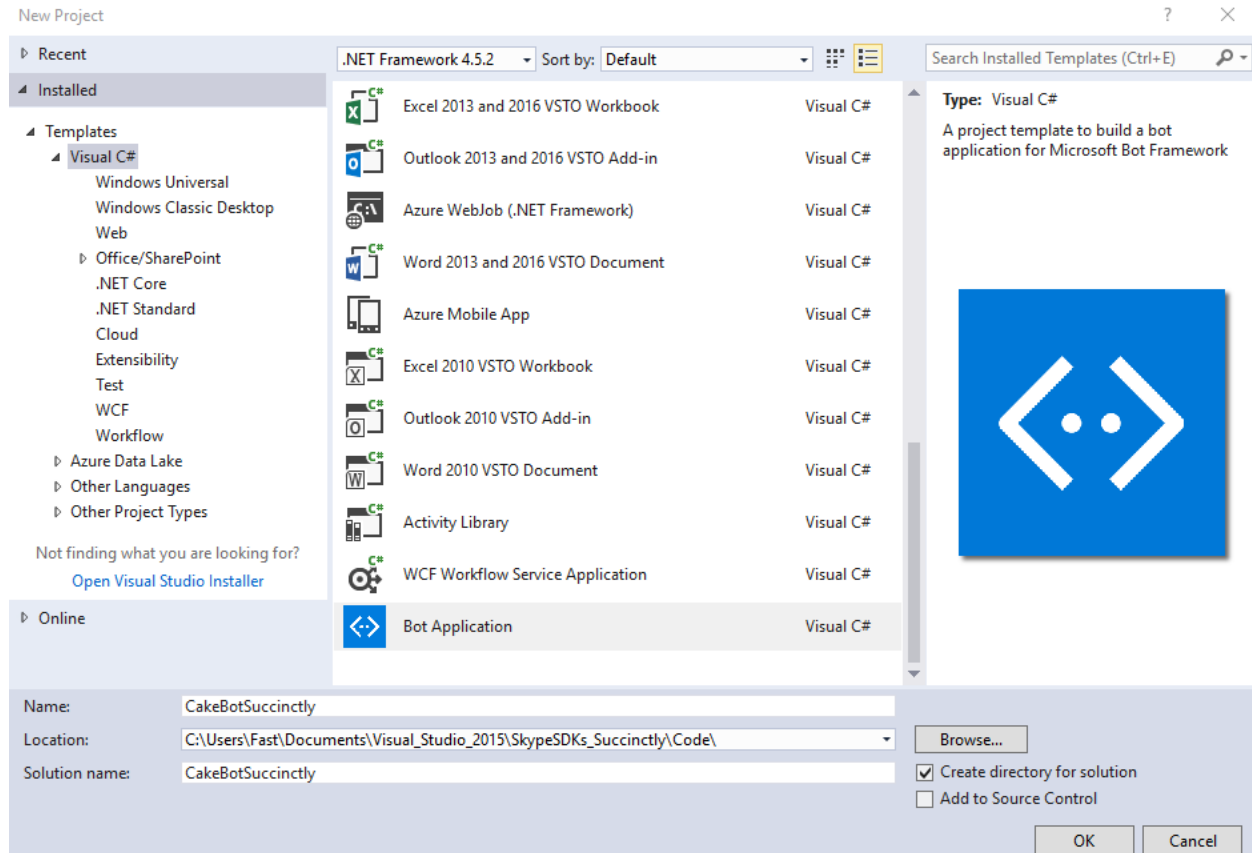
Figure 1-a: Creating a New Bot Application Project

I've decided to call the new project CakeBotSuccinctly, but you may choose any other name you wish.

With the project created, let's define the structure by creating some files we'll need.

But first things first—let's resolve any missing dependency issues. Open **Solution Explorer**, right-click **References**, and then select **Manage NuGet Packages**. We do this because it might be possible that not all required references are up-to-date.

In my case, several references are in this situation (not up-to-date), as can be seen in the following figure.
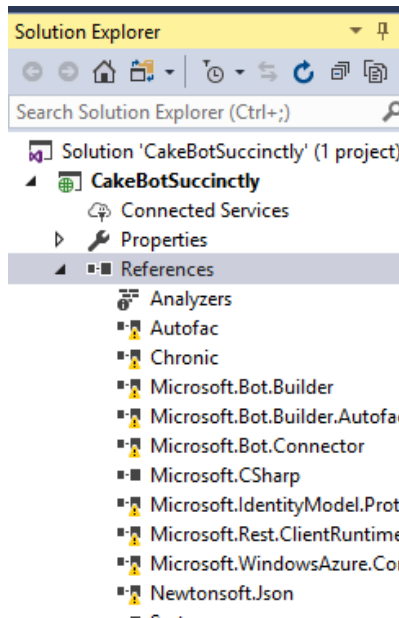
Figure 1-b: Project References Needing an Update

So, let's update them using the NuGet Package Manager. Click **Restore** at the upper right side of the screen, as can be seen in the following figure.
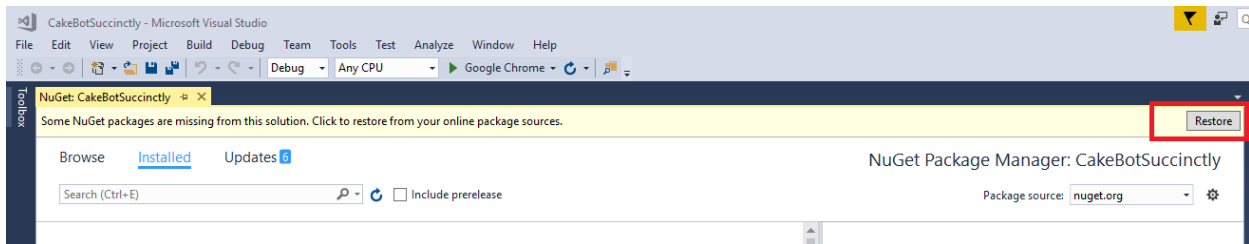
Figure 1-c: Updating Project References

Once you've clicked **Restore**, all references that are not up-to-date will be updated. This can be confirmed by clicking on any of the references seen in the **Solution Explorer** that open up the **Object Browser**. Notice how, after clicking on one of them—shortly after the **Object Browser** opens up—all the references now appear as correct.
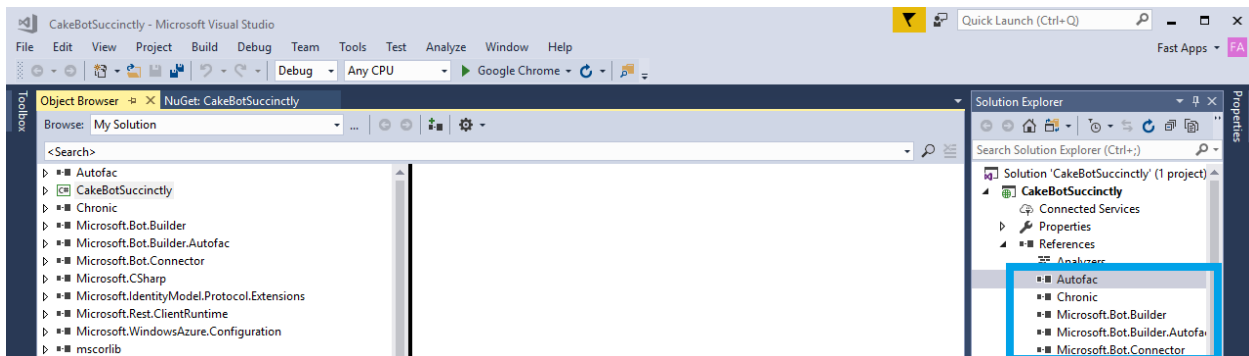
Figure 1-d: Updated Project References

Awesome! We now have all our project references updated. With this out of the way, let's create the basic project structure and add the files we'll need for creating our Skype bot application.

Bot project structure

When creating a Skype bot—or, in general, any bot using the Microsoft Bot Framework, there is no specific rule on how to organize the project files that will be used. However, some recommendations can be made on how to define an easy-to-follow project structure that is simple to remember and implement. This is what we'll focus our attention on now.

The Bot Template for Visual Studio does a pretty good job of defining a basic project structure for our bot; however, there are still a few things missing, and that's what we will add now.

The most important elements that the Bot Template defines are the **Controllers** and **Dialogs** folders.
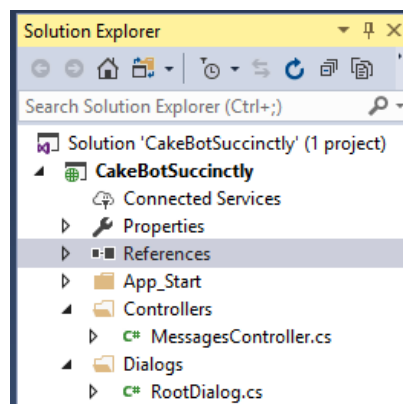
Figure 1-e: Basic Bot Project Structure

The **Dialogs** folder is where we'll create the dialog classes that our bot will be using. We'll also create a **Models** folder, where we'll add our bot's data model and any [FormFlow](#) logic needed.

First, let's create a *WelcomeDialog.cs* file under the **Dialogs** folder. To do so, right-click on the **Dialogs** folder, click **Add**, and then click **Class**. We'll use the **WelcomeDialog.cs** file to hold all the logic that our bot will use to greet users.

Next, let's create a *CakeBotDialog.cs* file following the same steps we've just used for creating *WelcomeDialog.cs*. Our *CakeBotDialog.cs* file will contain the dialog logic for our bot.

With that out of the way, let's now create a **Models** folder, which will contain the C# class that will contain the bot's FormFlow logic. We can create the **Models** folder by right-clicking on the bot's name—in my case, *CakeBotSuccinctly*—and then **Add** > **New Folder**.

Under **Models**, create a new C# class file—I'll call mine *Cakes.cs*—which we'll use for most of the bot's conversational FormFlow logic, data-model, and data-validation. You can do this by clicking on the **Dialogs** folder, then click **Add**, and click **Class**.

Great—having followed these steps, our bot's project structure should now look like Figure 1-f.
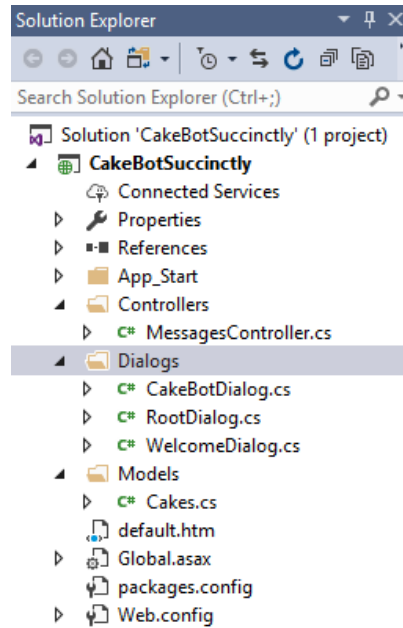
Figure 1-f: Updated Bot Project Structure

RootDialog.cs contains dialog logic that was added automatically by the Bot Template, and for now we won't use it; instead, we'll use *CakeBotDialog.cs*.

With this structure in place, we can now start adding logic to our bot in an organized way.

The bot's Welcome dialog

The purpose of the *WelcomeDialog.cs* file is to include the necessary bot logic in order to start the bot's conversation with the user. Let's jump straight into the code. This is what the complete *WelcomeDialog.cs* file looks like:

Code Listing 1-a: WelcomeDialog.cs

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using CakeBotSuccinctly.Models;

namespace CakeBotSuccinctly.Dialogs
{
    [Serializable]
    public class WelcomeDialog : IDialog
    {
        public async Task StartAsync(IDialogContext context)
        {
            // Str class defined in Models.Cakes.cs
```

```csharp
            await context.PostAsync(Str.cStrHi);
            await Respond(context);

            context.Wait(MessageReceivedAsync);
        }

        private static async Task Respond(IDialogContext context)
        {
            var userName = string.Empty;
            context.UserData.TryGetValue(Str.cStrName, out userName);

            if (string.IsNullOrEmpty(userName))
            {
                await context.PostAsync(Str.cStrNameQ);
                context.UserData.SetValue(Str.cStrGetName, true);
            }
            else
            {
                await context.PostAsync(
                  string.Format(
                  "Hi {0}, how may I help you today? You may order...",
                   userName));
                await context.PostAsync(string.Join(", ",
                    Str.CakeTypes));
            }
        }

        public async Task MessageReceivedAsync(IDialogContext context,
          IAwaitable<IMessageActivity> argument)
        {
            var msg = await argument;
            string userName = string.Empty;
            bool getName = false;

            context.UserData.TryGetValue(Str.cStrName, out userName);
            context.UserData.TryGetValue(Str.cStrGetName, out getName);

            if (getName)
            {
                userName = msg.Text;
                context.UserData.SetValue(Str.cStrName, userName);
                context.UserData.SetValue(Str.cStrGetName, false);
            }
            await Respond(context);
            context.Done(msg);
        }
    }
}
```

Let's now dissect this code into smaller chunks in order to understand what is happening. The first thing we need to do is to add **using** statements that reference the namespaces we'll need.

Code Listing 1-b: The WelcomeDialog Class References

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using CakeBotSuccinctly.Models;
```

The most important ones we'll need are **Microsoft.Bot.Builder.Dialogs** and **Microsoft.Bot.Connector**.

Because *WelcomeDialog.cs* resides under the **Dialogs** folder within our Visual Studio project structure, our **WelcomeDialog** class belongs to the **CakeBotSuccinctly.Dialogs** namespace by default.

It's very important that we set the **Serializable** attribute to the **WelcomeDialog** class, in order for the Bot Framework to treat it as a bot dialog—if not, a runtime exception will be produced when the bot is executed.

It is also essential that **WelcomeDialog** inherits from the **IDialog** interface. This requires the class to implement the **StartAsync** method, which triggers the start of this dialog.

Code Listing 1-c: The StartAsync Method

```csharp
public async Task StartAsync(IDialogContext context)
{
    await context.PostAsync(Str.cStrHi);
    await Respond(context);

    context.Wait(MessageReceivedAsync);
}
```

Within **StartAsync**, **context.PostAsync** is invoked, which is responsible for sending a **hi** string message to the user. This string is represented by the constant **cStrHi**, which belongs to the **Str** class of *Cakes.cs* (which we'll look at later). The **Str** class is simply used to store string constants that our bot will use. Another approach is to use the .NET resources storage technique.

Then, a call to the **Respond** method is invoked (which we'll look at next), and, finally, **context.Wait** is called, passing the **MessageReceivedAsync** method as a callback function.

Let's have a look at what happens within the **Respond** method.

Code Listing 1-d: The Respond Method

```csharp
private static async Task Respond(IDialogContext context)
{
```

```
    var userName = string.Empty;
    context.UserData.TryGetValue(Str.cStrName, out userName);

    if (string.IsNullOrEmpty(userName))
    {
        await context.PostAsync(Str.cStrNameQ);
        context.UserData.SetValue(Str.cStrGetName, true);
    }
    else
    {
        await context.PostAsync(
            string.Format(
            "Hi {0}, how may I help you today? You may order...",
            userName));
        await context.PostAsync(string.Join(", ", Str.CakeTypes));
    }
}
```

The idea behind the **WelcomeDialog** class logic is that the bot will greet the user and then ask for the user's name. This way, an amicable conversation can be initiated by prompting for the next question with the user's name included.

In order to make this happen, the **Respond** method will call **context.UserData.TryGetValue** to check whether the user's name has been stored in the bot's state—which would indicate that the user's name has already been provided (entered by the user). The value stored in the bot's state is then assigned to the **userName** variable.

If there's no value assigned to the **userName** variable, then the bot will respond by requesting the user's name—this is done by invoking **context.PostAsync**—and then a flag called **GetName** (represented by the constant **Str.cStrGetName**) is set to **true** within the bot's state. This is done so that the bot knows that it has requested the name to the user.

On the contrary, if **userName** does have a value assigned to it, this indicates that the bot knows who the user is, and replies back by suggesting what orders can be placed. This is also done by calling **context.PostAsync** twice within the **else** clause.

Let's now look at the implementation of the **MessageReceivedAsync** method.

Code Listing 1-e: The MessageReceivedAsync Method

```
public async Task MessageReceivedAsync(IDialogContext context,
    IAwaitable<IMessageActivity> argument)
{
    var msg = await argument;
    string userName = string.Empty;
    bool getName = false;

    context.UserData.TryGetValue(Str.cStrName, out userName);
    context.UserData.TryGetValue(Str.cStrGetName, out getName);
```

```
    if (getName)
    {
        userName = msg.Text;
        context.UserData.SetValue(Str.cStrName, userName);
        context.UserData.SetValue(Str.cStrGetName, false);
    }

    await Respond(context);
    context.Done(msg);
}
```

Basically, this method checks the bot's state in order to know if the **Name** and **GetName** variables have been set.

If **GetName** has been set to **true**—which is done by the **Respond** method—then it is assumed that the user's response contained within **msg.Text** is the user's name (entered by the user following the bot's request).

In this case, the user's name is stored within the bot's state using the **UserName** variable, and **GetName** is set to **false**—meaning that the user's name is known and the bot doesn't need to ask for it.

Finally, the **Respond** method is invoked, followed by a call to **context.Done**. So, this concludes the bot's *WelcomeDialog* logic. It was actually very straightforward and not complicated at all.

We'll now move our attention to *CakeBotDialog.cs*, where we'll learn some new and exciting techniques.

The Cake bot dialog

In this first iteration of our bot, we won't be using any AI features yet—such as [LUIS](#), which we'll explore and look at later.

Because our bot has no sophisticated (AI-powered) way of knowing the user's intent (the action the user wants to perform), we must somehow instruct the bot which dialog it should use in order to respond to the user's request. This is the main objective of the logic that will be created within *CakeBotDialog.cs*.

So, how can the bot then anticipate where to route user intents—which dialog to use in order to respond to a given request? It's a good question, to which there's not a single answer. One way is by using **Switch** statements. So, let's see how we can implement this dialog routing logic. This approach is sometime called rule-based logic.

The first thing we need to do is to go to the **Solution Explorer** within Visual Studio and double-click on the **CakeBotDialog.cs** file in order to open it. Within the **CakeBotDialog** class, add the following read-only variable, called **dialog**, as follows.

Code Listing 1-f: Starting the CakeBotDialog Class

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

namespace CakeBotSuccinctly.Dialogs
{
    public class CakeBotDialog
    {
        public static readonly IDialog<string> dialog;
    }
}
```

This **dialog** is going to create a chain, which is simply a series of dialogs that get called, one after the other.

The first thing we do after creating the snippet shown in Code Listing 1-f is to write **Chain.PostToChain**, which basically says: take a message that we get back from the user and start running it through a chain. Then we are going to select what we need from that message and use a **select** statement.

As we'll see shortly in the code listing that will follow, all we really care about is the value of the **msg.Text** property, so that's what we'll base our **select** statement logic on.

We'll need to choose from two dialogs: one is the **WelcomeDialog**, and the other is **CakeBotDialog**.

The objective is that when a user types the word "hi," we are able to instantiate the **WelcomeDialog** class, and when the user types anything else, we want to fall into the FormFlow logic of **CakeBotDialog**—this happens after the conversation has started and the bot knows the user's name.

In order to achieve this, we'll need to use a **Switch** statement and use a regular expression to search for the word **hi**. We'll first put in the **Switch** statement, and then put in a regular expression, **RegexCase**.

RegexCase is basically a callback function that passes **context** and **text** to the next dialog in the chain, which will be the **WelcomeDialog**.

RegexCase is going to look for a regular expression, checking to see if it can find the word "hi" (ignoring the case), and then calling the next dialog in the chain.

In order to continue with the **WelcomeDialog**, we need to use the **Chain.ContinueWith** method, which will take two variables: one is the dialog that we want to chain and use, and the other is what will get invoked after the dialog has been finished.

The first variable will be a new instance of **WelcomeDialog**, and the second one will be a new method called **AfterWelcomeContinuation**, which we'll create shortly.

This takes care of one routing. Code Listing 1-g shows what the code now looks like.

```
using Microsoft.Bot.Builder.Dialogs;
using System.Text.RegularExpressions;

namespace CakeBotSuccinctly.Dialogs
{
    public class CakeBotDialog
    {
        public static readonly IDialog<string> dialog =
            Chain.PostToChain()
            .Select(msg => msg.Text)
            .Switch(
            new RegexCase<IDialog<string>>(new Regex("^hi",
                RegexOptions.IgnoreCase), (context, text) =>
            {
                return Chain.ContinueWith(new WelcomeDialog(),
                    AfterWelcomeContinuation);
            }
            )
    }
}
```

Notice that this code is not yet syntactically correct—we are still missing part of the **Switch** statement details, and also the implementation of the **AfterWelcomeContinuation** method.

The idea is to show you step-by-step what is being done and how, rather than to show the finished **CakeBotDialog** class in one go.

Let's explore the other routing path—what happens when a user types anything other than "hi"—after the conversation has started.

This other path is going to use the default case, which is simply going to take in a callback to the next part of the dialog chain, for which we'll use the **ContinueWith** method.

Within **ContinueWith**, we'll be calling **FormDialog.FromForm**, passing a **BuildForm** method— which we'll create later within our *Cakes.cs* file. The **BuildForm** method is going to return a **FormDialog** object.

To **FormDialog.FromForm**, we want to also pass in the option **PromptInStart**, which will automatically kick off our FormFlow when we fall into this section of the code. This is done because otherwise we'll have to send an additional message to the bot in order to start the dialog.

The second **ContinueWith** takes in a call to the **AfterFormFlowContinuation** callback method. Finally, after **Switch**, there's a call to the **Unwrap** and **PostToUser** methods.

Code Listing 1-h shows what the code looks like now, without yet implementing the **AfterWelcomeContinuation** and **AfterFormFlowContinuation** methods.

```csharp
using CakeBotSuccinctly.Models;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace CakeBotSuccinctly.Dialogs
{
    public class CakeBotDialog
    {
        public static readonly IDialog<string> dialog =
            Chain.PostToChain()
            .Select(msg => msg.Text)
            .Switch(
            new RegexCase<IDialog<string>>(new Regex("^hi",
            RegexOptions.IgnoreCase), (context, text) =>
            {
                return Chain.ContinueWith(new WelcomeDialog(),
                    AfterWelcomeContinuation);
            }),
            new DefaultCase<string, IDialog<string>>((context, text) =>
            {
                return
                  Chain.ContinueWith(FormDialog.FromForm(Cakes.BuildForm,
                  FormOptions.PromptInStart), AfterFormFlowContinuation);
            }))
            .Unwrap()
            .PostToUser();

        private async static Task<IDialog<string>>
            AfterWelcomeContinuation(IBotContext context,
            IAwaitable<object> item)
        {
        }

        private async static Task<IDialog<string>>
            AfterFormFlowContinuation(IBotContext context,
            IAwaitable<object> item)
        {
        }
    }
}
```

With this done, let's implement the **AfterWelcomeContinuation** and **AfterFormFlowContinuation** methods.

```csharp
private async static Task<IDialog<string>>
    AfterWelcomeContinuation(IBotContext context,
    IAwaitable<object> item)
{
    var tk = await item;

    string name = "User";
    context.UserData.TryGetValue("Name", out name);
    return Chain.Return("Type one..." + name);
}

private async static Task<IDialog<string>>
    AfterFormFlowContinuation(IBotContext context,
    IAwaitable<object> item)
{
    var tk = await item;
    return Chain.Return("Thanks, you're awesome");
}
```

The **AfterWelcomeContinuation** method returns a message to the current user asking for input—so that the user can indicate the desired **CakeTypes** value: Cup Cake, Triple Layer Cake, or Cream Cake. These values will be defined later in *Cakes.cs*.

There is no actual validation of the **CakeTypes** value entered by the user.

The **AfterFormFlowContinuation** method, on the other hand, simply sends back a thank you message to the user, after the FormFlow process has been completed.

The complete source code of the **CakeBotDialog** class should now look like Code Listing 1-j.

Code Listing 1-j: The CakeBotDialog Class

```csharp
using CakeBotSuccinctly.Models;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace CakeBotSuccinctly.Dialogs
{
    public class CakeBotDialog
    {
        private const string cStrUser = "User";
        private const string cStrName = "Name";
        private const string cStrTypeOne =
            "Type one...";
```

```csharp
        public static readonly IDialog<string> dialog =
            Chain.PostToChain()
            .Select(msg => msg.Text)
            .Switch(
            new RegexCase<IDialog<string>>(new Regex("^hi",
            RegexOptions.IgnoreCase), (context, text) =>
            {
                return Chain.ContinueWith(new WelcomeDialog(),
                    AfterWelcomeContinuation);
            }),
            new DefaultCase<string, IDialog<string>>((context, text) =>
            {
                return
                Chain.ContinueWith(FormDialog.FromForm(Cakes.BuildForm,
                FormOptions.PromptInStart), AfterFormFlowContinuation);
            }))
            .Unwrap()
            .PostToUser();

        private async static Task<IDialog<string>>
            AfterWelcomeContinuation(IBotContext context,
            IAwaitable<object> item)
        {
            var tk = await item;
            string name = cStrUser;
            context.UserData.TryGetValue(cStrName, out name);
            return Chain.Return($"{cStrTypeOne} {name}");
        }

        private async static Task<IDialog<string>>
            AfterFormFlowContinuation(IBotContext context,
            IAwaitable<object> item)
        {
            var tk = await item;
            return Chain.Return("Thanks, you're awesome");
        }
    }
}
```

As you can see, the only difference is that we've created some string constants at class level, instead of using those strings directly inside the **AfterWelcomeContinuation** method; the rest remains the same.

So, we are now done with our **CakeBotDialog** class. Let's wire this up in our *MessagesController.cs* file.

Wiring things up in MessagesController

Go to the **Solution Explorer** and open the *MessagesController.cs* file under the **Controllers** folder. The **Post** method of the **MessagesController** class looks like Code Listing 1-k.

Code Listing 1-k: The "Out-of-the-Box" Post Method of the MessagesController Class

```csharp
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new
            Dialogs.RootDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }

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

The **Post** method is where our bot will receive any incoming requests.

We won't be using the **RootDialog** class for now, and because we want to wire up our **WelcomeDialog** and **CakeBotDialog** classes, what need to do is to change **new Dialogs.RootDialog()** to **Dialogs.CakeBotDialog.dialog**.

So let's go ahead and do this. The **Post** method should now look as follows.

Code Listing 1-l: The "Wired Up" Post Method of the MessagesController Class

```csharp
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () =>
            Dialogs.CakeBotDialog.dialog);
    }
    else
    {
        HandleSystemMessage(activity);
    }

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

With things wired up, we can now focus our attention on the FormFlow logic. The next thing we'll do is implement *Cakes.cs*, which will contain our bot's model as well.

The bot's FormFlow and model

The *Cakes.cs* file under the **Models** folder is where we'll create the bot's FormFlow logic, and also its model.

Let's start off by creating the FormFlow that our bot conversation will follow. In order to do that, let's create a **Cakes** *serializable* class, which will contain the following logic.

Code Listing 1-m: The Cakes FormFlow Class

```csharp
[Serializable]
public class Cakes
{
    [Prompt(Str.cStrQuantity)]
    public string Quantity;

    [Prompt(Str.cStrOptions)]
    public string When;

    public static IForm<Cakes> BuildForm()
    {
        OnCompletionAsyncDelegate<Cakes>
            processOrder = async (context, state) =>
        {
            context.UserData.SetValue(Str.cStrGetName, true);
            context.UserData.SetValue(Str.cStrName, string.Empty);

            await context.PostAsync(
                $"{Str.cStrProcessingReq} {Validate.DeliverType}");
        };

        return new FormBuilder<Cakes>()
            .Field(nameof(Quantity))

            .Message(Str.cStrWhen)
            .Field(nameof(When),
            validate: async (state, value) =>
            {
                return await Task.Run(() =>
                {
                    string v = value.ToString();

                    return Validate.ValidateType
                        (state, value.ToString(), Str.DeliverTypes);
                });
```

```
            })

            .OnCompletion(processOrder)
            .Build();
    }
}
```

Let's look at what is going on here. The **Cakes** class contains two questions that our bot will ask users in a sequential way.

The first question, represented by the variable **Quantity**, prompts the user to indicate how many cakes he or she would like to order.

The second question, represented by the variable **When**, prompts the user to specify when he or she would like the order delivered—the preferred delivery option.

Both are questions because they have a prompt attribute that indicates the question being asked, which is specified by two constants: **Str.cStrQuantity** and **Str.cStrOptions**, which we'll declare shortly.

The third and most important part of the **Cakes** class is the **BuildForm** method. In order to understand what it does, let's divide it into two parts.

The first part of the **BuildForm** method is **OnCompletionAsyncDelegate**. This gets executed at the end of the FormFlow, when both questions have been answered by the user. You can think of it as the event that gets executed when a cake order has been submitted.

The second part of the **BuildForm** method is the **FormBuilder** chain. This raises the first question (for which no validation takes place). When the user has answered it, a second question is raised and then validated—in order to check if the response provided by the user for that second question is valid. If so, **OnCompletionAsyncDelegate** is executed when **OnCompletion(processOrder)** is invoked.

It is important to notice that the validation process for the second question is executed through an anonymous **async** function that receives the FormFlow's current **state** and the value entered by the user for that second question.

There's no technical reason why the full validation of the question's input cannot be done inside the anonymous **async** function itself. However, I think it's nicer and cleaner to have the specific validation logic inside a separate class dedicated specifically for this purpose—which we'll call **Validate**.

Because the **Validate** class is tied up in the bot's FormFlow logic—in other words, to the **Cakes** class logic—we'll declare the **Validate** class inside the same *Cakes.cs* file, so both go hand-in-hand.

Tuples

Before we write the code for the **Validate** class, I'll be making use of an improved feature in C# 7 called [tuples](#)that will come in handy now. In order to use tuples, we'll first need to add a NuGet package to our bot project, which is external from Visual Studio.

Adding an external dependency for a barely useful construction is not extremely exciting. However, for our bot example, it is useful to have certain validations that return more than one value—thus in this case, it is a good chance to showcase this C# feature—even though it is not essential for the Bot Framework.

Open Solution Explorer, right-click on **References**, and select **Manage NuGet Packages**.

Once the NuGet Package Manager window is open, with the **Browse** item selected, in the search, type in **System.ValueTuple**. Then, select this package and click **Install**.
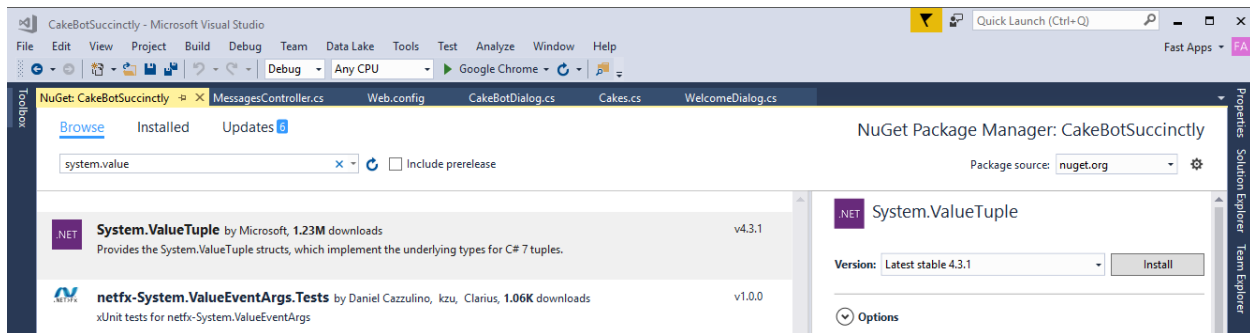
Figure 1-g: The System.ValueTuple NuGet Package

When the package has been installed, we can write the **Validate** class code, which is what we'll do next.

The Validate class

A common good practice is to have separation of concerns within our code. A way to achieve this is by having the bot's FormFlow validations managed by its own class—this is what we'll focus now on. Here's the full source code for the **Validate** class, which belongs to the **CakeBotSuccinctly.Models** namespace.

Code Listing 1-n: The Validate Class

```csharp
public class Validate
{
    public static string DeliverType = string.Empty;

    public static (bool v, string t) TypeExists(string c, string[] types)
    {
        bool v = false;
        string t = string.Empty;
```

```csharp
        foreach (string ct in types)
        {
            if (ct.ToLower().Contains(c.ToLower()))
            {
                v = true;
                t = ct;

                break;
            }
        }

        return (v, t);
    }

    public static ValidateResult ValidateType(Cakes state, string value,
        string[] types)
    {
        ValidateResult result = new ValidateResult { IsValid = false,
            Value = string.Empty };

        var res = TypeExists(value, types);

        string r = $"{Str.cStrDeliverBy} {res.t}";
        DeliverType = res.t;

        return (res.v) ?
            new ValidateResult { IsValid = true, Value = res.t, Feedback
                = res.t } : result;
    }
}
```

Let's try to understand what is going on. The **ValidateType** method returns a **ValidateResult** object; by default, its **IsValid** property is set to **false**, and its **Value** property is set to an empty **string**—which means that the user's input could not be found (validated).

All the **ValidateType** method does is to call the **TypeExists** method, which returns a tuple that includes a Boolean variable—set to **true** (if a **DeliverTypes** value is found), and the actual **DeliverTypes** value that corresponds to the user's input—which is assigned to the static **DeliverType** variable.

The **DeliverType** value is then used within **OnCompletionAsyncDelegate** of the **Cakes** class.

The **TypeExists** method basically loops through the existing **DeliverTypes** (represented by the **string** array **types** variable) and checks whether the user's input (represented by the **string** variable **c**) matches any of the **DeliverTypes**.

If a match is found, the tuple's **v** variable is set to **true**, and its **t** variable is set to the matching **DeliverTypes** value.

Notice that the **ValidateType** method is only invoked when the second question of the FormFlow—"When would you like the cake delivered?"—is executed.

So that concludes the **Validate** class. In order to finalize our *Cakes.cs* file, let's implement the remaining part that corresponds to the **Str** class—which contains the **DeliverTypes** values and other useful constants used throughout the bot.

The Str class

All of the bot's constants are implemented in the **Str** class, which is all that this class does. The code looks as follows.

Listing 1-o: The Str Class

```csharp
public class Str
{
    public const string cStrHi = "Hi, I'm CakeBot";
    public const string cStrNameQ = "What is your name?";
    public const string cStrName = "Name";
    public const string cStrGetName = "GetName";

    public const string cStrWhen =
        "When would you like the cake delivered?";
    public const string cStrProcessingReq =
        "Thanks for using our service. Delivery has been scheduled for: ";

    public const string cStrDontUnderstand =
        "I'm sorry I don't understand what you mean.";

    public const string cStrQuantity = "How many?";
    public const string cStrOptions = "Now or Tomorrow";
    public const string cStrDeliverBy = "Deliver by: ";

    public static string[] DeliverTypes =
        new string[] { "Now", "Tomorrow" };
    public static string[] CakeTypes =
        new string[] { "Cup Cake", "Triple Layer Cake", "Cream Cake" };

    public static string cStrNoPush = "NO_PUSH";
    public static string cStrTemplateType = "template";
    public static string cStrPayloadTypeGeneric = "generic";
}
```

The **Str**, **Validate**, and **Cakes** classes all belong to the **CakeBotSuccinctly.Models** namespace.

That concludes our bot's code. But before we can run it, make sure you install the Bot Framework Emulator, which can be found here. You may get a firewall warning.

Remember that the full source code is available at the beginning of this chapter for download.

Running our bot

With the Bot Framework Emulator installed, open it up and run your Visual Studio solution. Click **Run**, which in my case is set by default to use Google Chrome.

On the first run, you will receive instructions on how to register your bot with Microsoft, which requires a web server or endpoint that has HTTPS protocol capability—a relatively new requirement, as of June 23, 2016 (see this link).
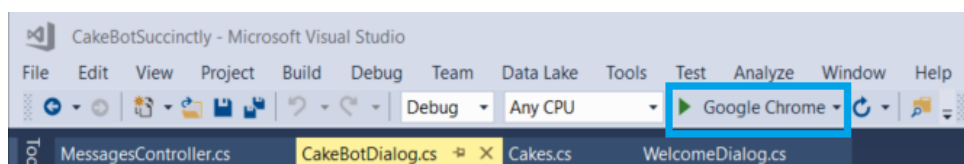

Figure 1-h: The Run Button in Visual Studio

Once our application and the emulator are running, let's start a very basic conversation by greeting the bot with a **hi** message—which triggers the bot's **WelcomeDialog** class logic.
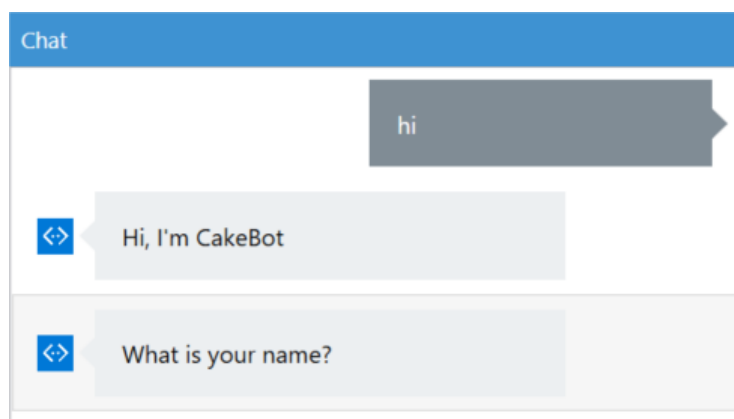
Figure 1-i: Starting a conversation with our Bot using the Emulator

Depending on the version of the Bot Emulator that you installed, the screen might slightly defer from the screen shot in Figure 1-i. In my case, I'm using the Bot Emulator version 3.0.0.59.

Notice that because we have entered the word **hi**, the **RegexCase** in *CakeBotDialog.cs* has triggered the execution of the **WelcomeDialog** object, which is why the bot is asking us for our name.

Although this is not highly sophisticated, because we need to properly start the conversation by typing the word **hi**, it's a good example of how we can use the **Chain.PostToChain** and **Chain.ContinueWith** methods in order to kick off a conversation, and how we can combine dialogs with FormFlow.

We'll later look at improving this, by being able to use other words to start off a conversation, when we explore LUIS.

Let's carry on the conversation with our bot by providing our name and answering any other questions.
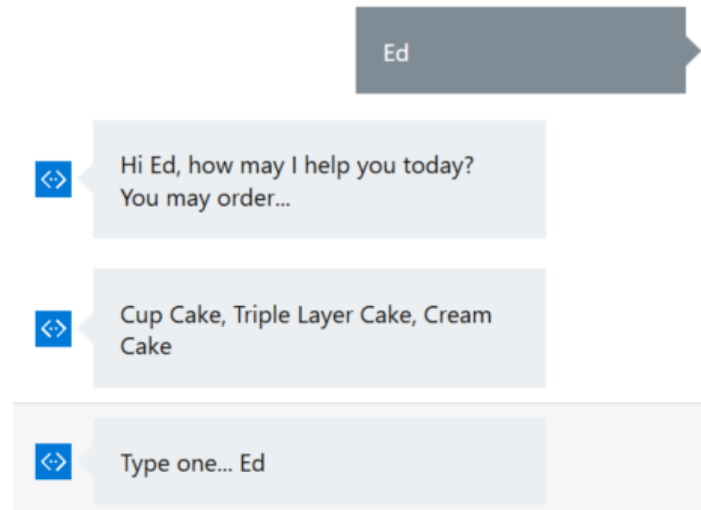

Figure 1-j: Continuing the Conversation with Our Bot Using the Emulator

We can see in Figure 1-j that the first two responses from the bot correspond to logic contained within the **Respond** method of *WelcomeDialog.cs*. The third response corresponds to the execution of the **AfterWelcomeContinuation** method contained within *CakeBotDialog.cs*.

Let's indicate a response to the bot. In my case, I'll type in **Triple Layer**. Notice that this value is not being validated at all—we could potentially just type anything, but let's not do that.
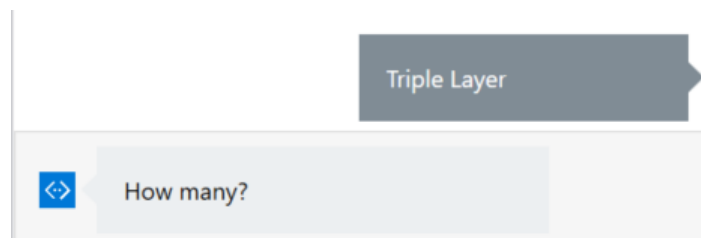
Figure 1-k: Continuing the Conversation with Our Bot Using the Emulator

After indicating my answer to the bot's previous question, notice how the FormFlow dialog is triggered, and we are asked how many cakes we want.

So at this stage, the *WelcomeDialog.cs* logic execution has finalized, and control has been passed to the FormFlow logic—which has been triggered by the execution of the logic contained within the **DefaultCase** section of *CakeBotDialog.cs*.

Next, I'll provide an answer to the FormFlow's first question. I'll indicate **2** as my answer.
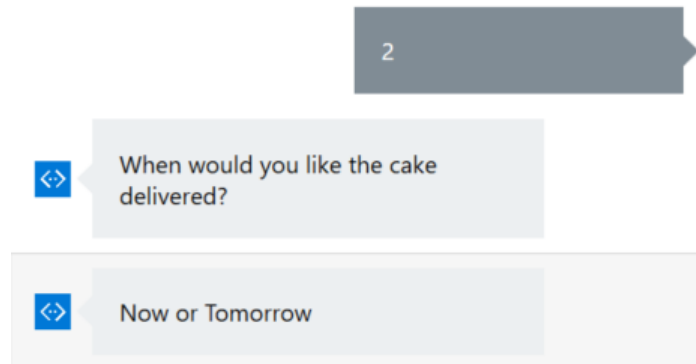
Figure 1-l: Continuing the Conversation with Our Bot Using the Emulator

Notice how the bot requests an input to the second question of the FormFlow and provides two possible answer alternatives—the available **DeliverTypes**: **Now** or **Tomorrow**—which will be validated by the **Validate** class within *Cakes.cs*.

I'll indicate an answer to the question, which in my case will be **Now**. Once I do that, **OnCompletionAsyncDelegate** of the **Cakes** class is executed, and I get the following output from the bot.
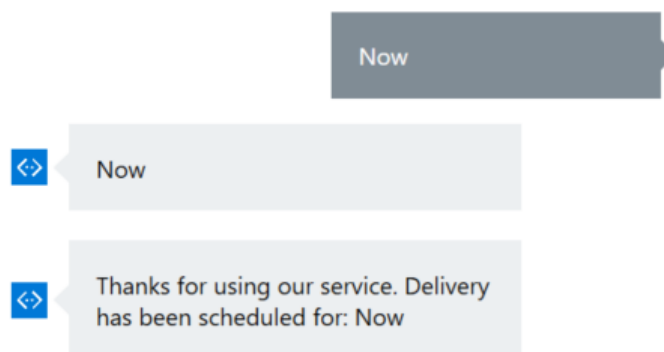
Figure 1-m: Continuing the Conversation with Our Bot Using the Emulator

Finally, there's one last bit that will get executed: the **AfterFormFlowContinuation** method of the **CakeBotDialog** class.

The final output looks like this:
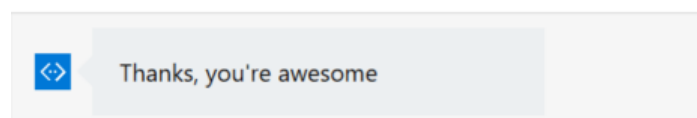
Figure 1-n: Final Response from Our Bot

By following this small but interesting conversation, we've seen how all parts of the various classes we've written so far tie nicely together.

The conversation flows between the *WelcomeDialog.cs* and FormFlow logic contained within the `Cakes` class.

Summary

In this chapter, we've looked at how to build a basic bot that goes beyond FormFlow—which was mostly the scope of *Microsoft Bot Framework Succinctly*. We've also managed to combine dialogs and FormFlow into one single codebase.

We've also seen how to use chaining in order to trigger the correct flow: either a simple dialog or FormFlow dialog—depending on the input provided by the user—and seen how this all ties nicely together by running our bot.

Separation of concerns has been taken into account, in order to keep a concise codebase that is easy to maintain and update.

Despite this implementation, we've noticed that we're a bit limited in how we can initiate a conversation, as we necessarily have to start off by typing "hi."

In the next chapter, we'll start exploring LUIS, which will give our application superpowers so that we can start off the conversation with additional words and use other variations throughout the conversation—without the need to use any `RegexCase` or `Switch` statements.

Don't forget that the full source code for this chapter can be found here. More awesome stuff lies ahead!

Chapter 2 LUIS Bots

Quick intro

In the previous chapter, we created a Skype bot that was able to combine standard dialogs and a FormFlow dialog, which is a very powerful feature. However, the disadvantage we ran into is that triggering the conversation was limited to using one word.

In the real world, we'd like our bot to be a bit smarter and behave more like a human, and also be capable of responding to various user intents, in order to start a conversation.

LUIS is a service from Microsoft that allows us to add conversational intelligence to our apps in a relatively easy way—it provides an API that allows us to make out bots more "self-aware."

LUIS provides an endpoint that will take the sentences you send it and interpret them according to the intention they convey and the key entities that are present. It easily integrates with the Bot Framework and is the next step in making our bot a bit smarter.

What we'll do in this chapter is to modify and adapt the bot we previously wrote so that it is capable of working with LUIS and being able to process various user intents—this will allow our bot to process multiple inputs instead of having one single entry point into conversation.

Sounds exciting, right? Let's get started. The full source code for this chapter can be found here.

Key aspects of LUIS

LUIS is designed to enhance your bot's understanding of natural language through the use of interactive machine learning.

What LUIS really does behind the scenes is play the role of a translator for our bot, but instead of translating one language into another, it's literally translating the user's requests into actions for our bot. So, in a world without LUIS, and based on the previous example bot we wrote, a typical workflow would be like this.

Figure 2-a: A World without LUIS

In this workflow, the bot itself is doing the heavy lifting—based on conditions as to which dialog or sequence of dialogs it must start whenever a user sends an intent.

The problem with this method is that it is extremely limited. What happens when the user enters "hello" instead of "hi"?

I think you know the answer to that one, but in case you don't, let's quickly run our bot again using this scenario. Let's have a look.
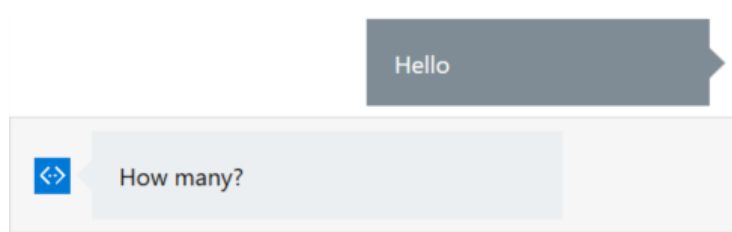

Figure 2-b: Oops… What Happened to Our WelcomeDialog?

Wait a second… isn't "hello" supposed to have the same meaning as "hi"? Well, for us humans, yes—and for LUIS too, once trained. However, for our bot, because we are using a `RegexCase`, only when we enter "hi" is the `WelcomeDialog` class logic triggered.

Otherwise, the FormFlow (`Cakes`) logic is triggered, which is what just happened by typing "hello." That's what LUIS solves.

I suppose we could always write a better regular expression in the `CakeBotDialog` class to take care of recognizing "hi" and "hello," but that's cumbersome, and there could be other ways to initiate a conversation. So, making the regular expression range wider (with more words) is not really an option if you want to scale and have a smarter bot.

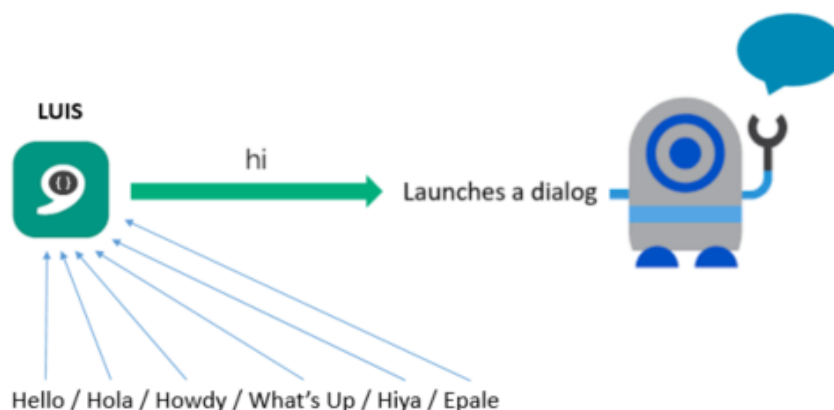In a world with LUIS, our greeting workflow would look like this instead.

Figure 2-c: A World with LUIS

In this scenario, we have the ability to filter all of our requests through the LUIS framework—taking the user's intents (requests) and running them through its natural language-processing engine.

This will in turn indicate to our bot that *hi*, *hola*, and *howdy* all mean the same thing, and all these requests should get funneled to the `WelcomeDialog` class logic.

This also means we don't need to use any more any `RegexCase` or `Switch` statements to trigger dialogs or FormFlow.

Of course, I'm vastly oversimplifying the process here—and what happens internally—for LUIS to make such routing decisions based on the user's intent. But it's important that you understand as a general principle that LUIS does a lot of the heavy lifting of figuring out the subtle nuances of natural language understanding.

Intents

You can think of intents as the verbs of the LUIS world. They are defined as any action that you want your bot to take.

Figure 2-d: An Intent Triggers an Action—It's the Spark in the Bulb

With our Cake bot, we can easily identify two intents: one is the welcome sequence, and the other is the actual cake order itself (the FormFlow part) in which we are asked for the quantity and delivery. These two intents are two actions that our bot can take based upon what we have coded so far.

Once we've created these two intents, LUIS can then use machine learning in order to make a best guess as to what the user wants when a request is sent.

A great aspect of LUIS is that it has the ability to learn over time using active learning. With active learning, LUIS is capable of identifying which user requests it's unsure about—and how to respond to it—and will ask the developer to clarify which intent should go with that request.

As more of these tricky situations are clarified, LUIS will get better at making the right best guess for our bot.

Entities

If intents are the verbs of the LUIS world, then entities can be thought of as the nouns. Entities can be defined as the things that your bot is taking action on.

In our Cake bot, we have the intent to order a cake—the FormFlow process. This intent could be enhanced by adding entities to it. An example would be a phrase like: *ordering a cheese cake every Saturday in Paris near the Eiffel Tower*.

In order to make things easier, I've highlighted the intent in *yellow* and the entities in *blue*.

In this case, we're taking the basic intent of ordering a cake and adding various entities to it, such as *Eiffel Tower*, which indicates the location; *Paris*, which indicates the city; and *Saturday*, which indicates the day. The word *ordering* would be the actual intent.

By adding entities, it is possible to use the same intent for many different actions. Entities help LUIS do a better job of figuring out what the user wants.

Entities can pass information over to the bot, and you can then do something with it in the code—think of them as parameters that are passed to an action, which help you decide in your code what the bot should do.

Utterances

Given that intents are the verbs and entities the nouns of the LUIS world, then utterances are the sentences—they use both intents and entities to form ideas that LUIS can draw from in order to make conclusions.

So: "ordering a cheese cake every Saturday in Paris near the Eiffel Tower" is actually an utterance.

To understand this better, imagine that instead of the words *order* or *ordering*, someone uses the word *buy*. As humans, we know that in such a context, *buy* means the same as *order* or *ordering*.

However, LUIS must try to interpret which intent the phrase with the word *buy* is trying to target—in other words, which action to trigger. So, you can start to see that it's not a simple one-to-one match as if the user had used the word *order* or *ordering*.

Instead, LUIS has to infer that the word *buy* really means the same as *order* or *ordering*. In some instances, LUIS is able to make this leap with the power of active learning. However, sometimes we have to give LUIS a little help—and this is why we have utterances.

By specifying an utterance, we can tell LUIS that when a user uses the word *buy*, they are looking to place an *order*. Utterances help LUIS to have a bit more flexibility in understanding what exactly the user wants.

So, given our example phrase, an utterance would be: *buy a cheesecake*.

The LUIS UI

Now that we've explored the fundamental concepts behind LUIS, let's have a go at it. We'll now explore its UI and set things up.

To get started, you can go to the LUIS [website](#) and use your Microsoft account ([Outlook.com](#), MSN, or Live) to sign in.

If you don't have any of these accounts, you'll need to sign up for one. Once you've logged in, the UI should look similar to the following screenshot.
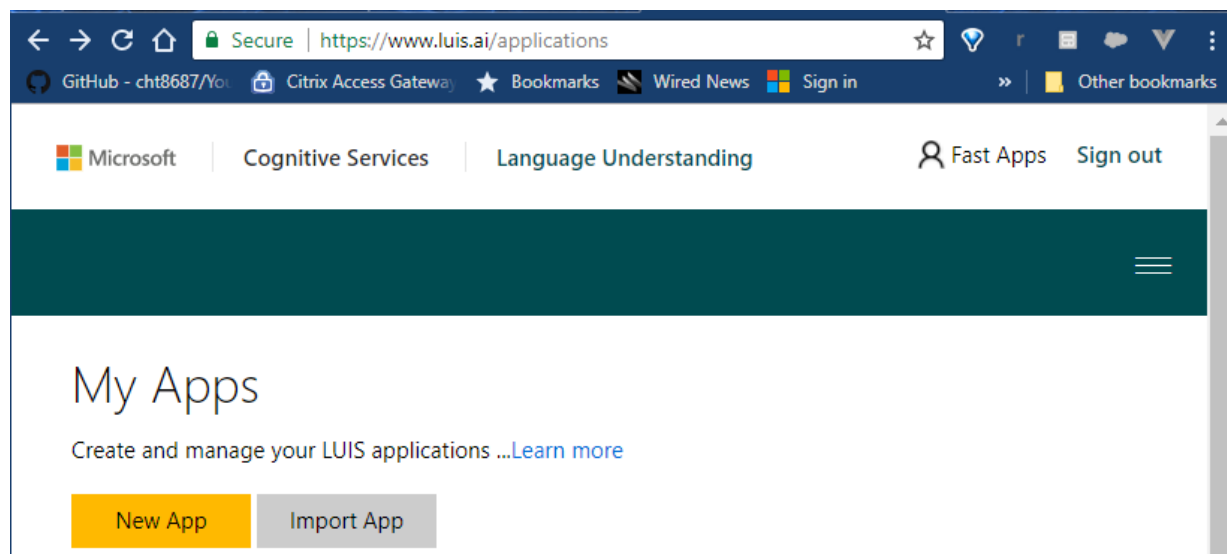

Figure 2-e: The LUIS UI

As Microsoft sometimes tends to update the look and feel of its sites, by the time you are reading this e-book, the UI might look slightly different. However, you should be able to find your way around by following the steps described here.

First, click **New App**. You'll see the following pop-up window on the screen.

Figure 2-f: A New LUIS App

Let's give the app a name—in my case, I'm going to call it **CakeBot**, and select **English** as the app's culture. The other two fields are optional, and for now, let's leave them as is.

If you have an Azure LUIS service active, it's best to use it as the endpoint. This will allow you to consume Azure resources and credits rather than leaving the endpoint key field blank, which limits the number of requests that you'll be able to consume from LUIS (which can run out pretty quickly). But for now, let's leave it blank so we'll be consuming what LUIS gives us out of the box.

Click **Create**, and you'll see an overview of the app created, as follows.
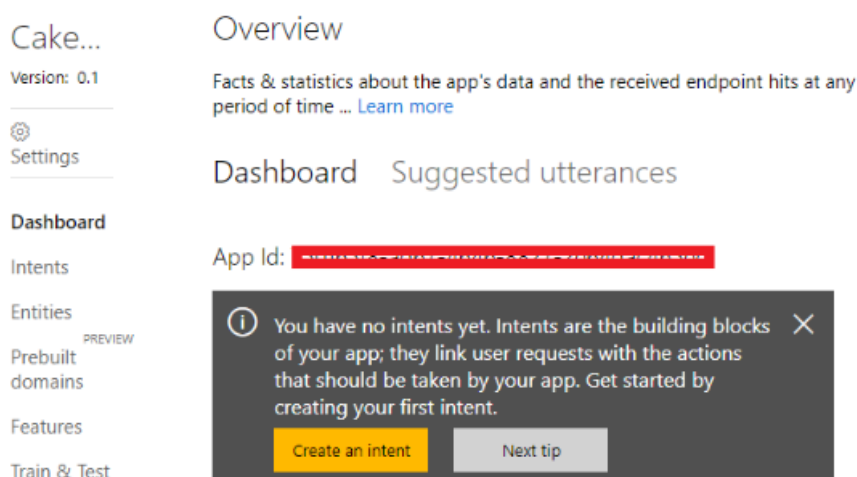
Figure 2-g: Overview of the LUIS App

As you can see in Figure 2-g, LUIS reminds us that this app has no intents. Let's go ahead and create one—click **Create an intent**. You'll see a screen similar to Figure 2-h. Here's where we'll enter our bot's intents.
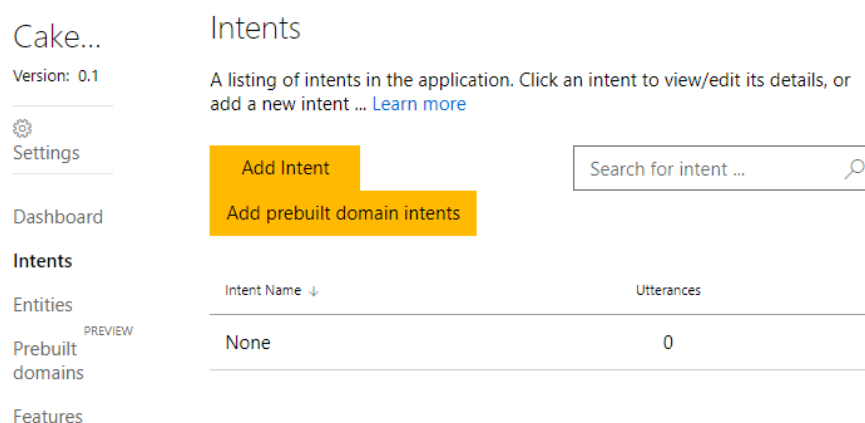

Figure 2-h: The LUIS Intents Screen

Click **Add Intent**, which will show the following pop-up window asking for the intent's name.

Figure 2-i: Adding a New Intent

Our bot has two intents: one is to start the conversation, which corresponds to the **WelcomeDialog** class logic, and the other is the actual cake order process. Let's name our first intent **Order** and click **Save**.

Now we can add any utterances (sentences) that will help the bot determine that the user's intent is to order a cake.

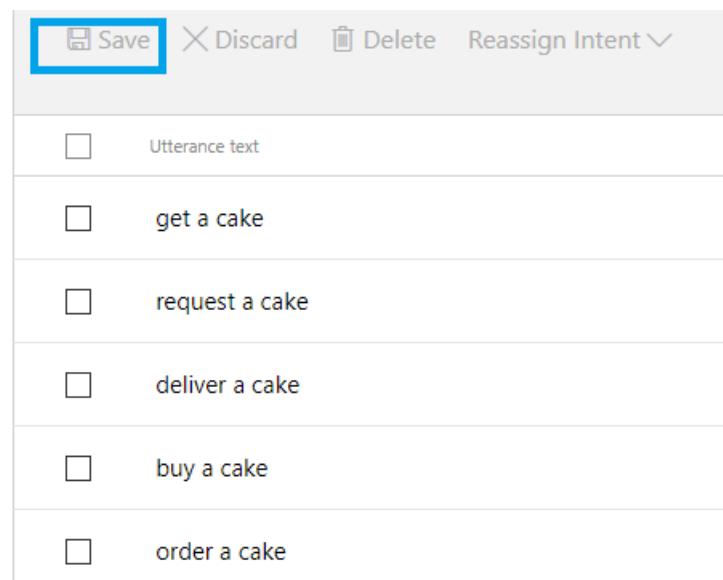Entering an utterance is very easy—all we have to do is to type it in and press **Enter**. Let's add a few that represent the cake order intent.

Figure 2-j: Order Cake Intent Utterances

As you can see, I've added a few short sentences that could represent the user's intent to order a cake. Once added, I also clicked **Save** at the top of the list.

We'll need to train LUIS to recognize these utterances and associate them with the intent to order a cake—but before we do that, let's create an intent for our **WelcomeDialog** class and add some utterances to it.
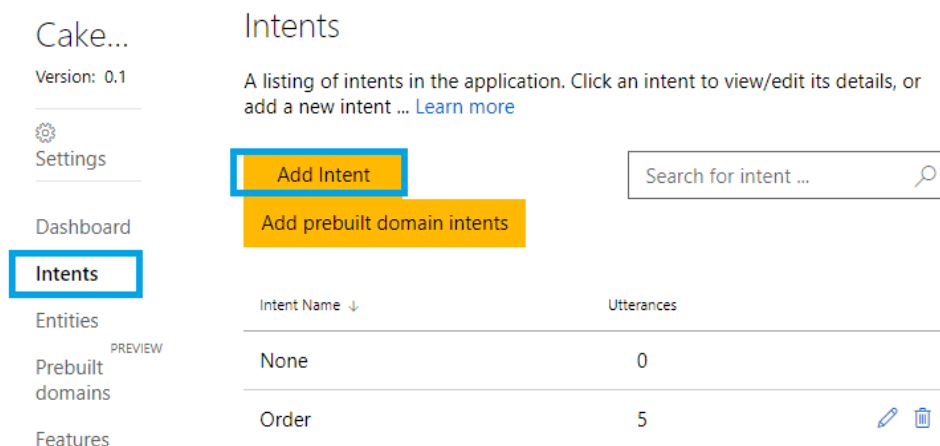
Figure 2-k: Adding Another Intent

In order to do that, click on the **Intent** link on the right side of the screen, which will take you to the intents list. You'll see that our **Order** intent is there. Click the **Add intent** button. When prompted, give it a name—in my case, I'll call it **Welcome**. Once created, let's add some utterances to it.

Figure 2-l shows the utterances I've added following the same steps I took for adding the ones related to the **Order** intent. Pretty straightforward.
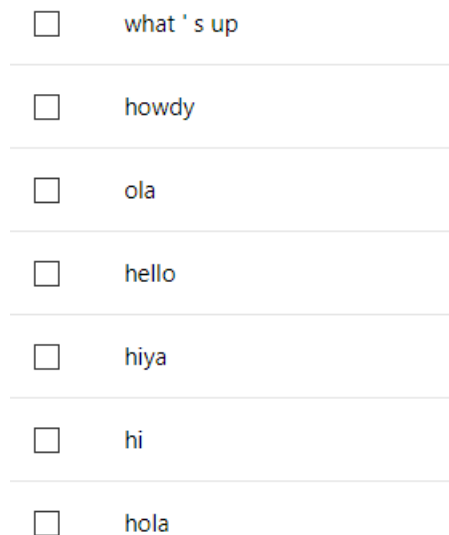

Figure 2-l: Welcome Intent Utterances

With these intents and utterances in place, the next step is to train our bot to use them.

Click on the **Train & Test** link on the right side of the screen.
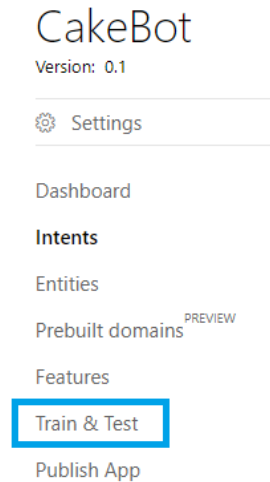
Figure 2-m: The "Train & Test" Option

This will take you to the following screen, where we can perform the training required for LUIS to do its magic.
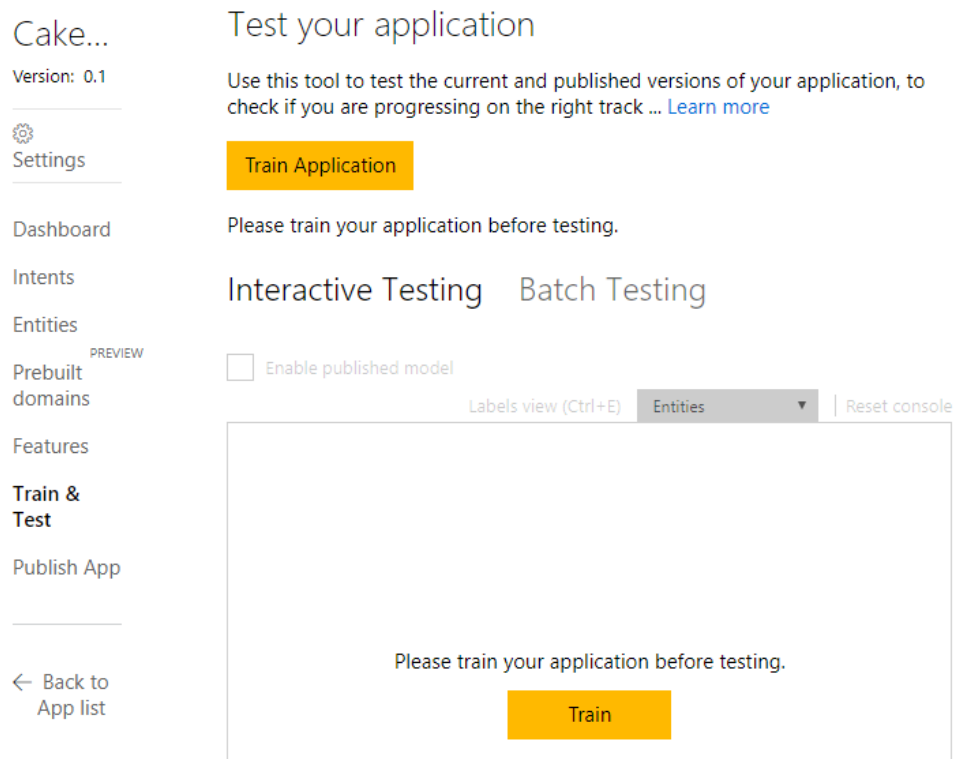

Figure 2-n: The "Train & Test" Screen

All we need to do is to click on either the **Train Application** or **Train** button. Let's go ahead and do this to see what happens.

After clicking on the **Train** button, you'll see how LUIS queues the application for training, and just a couple of seconds later, the application has been trained and is ready. At this point, you should see a screen like Figure 2-o.

Figure 2-o: The Interactive Testing Screen

What we can do here is quickly test our LUIS application by entering a sentence in the text box. I've entered **order cake** and pressed **Enter**. Immediately, LUIS gave me the result that the top scoring intent was **Order** with a score of 1 (100% certainty).

Let's do another quick test—what happens if I type in the word **hallo** (German for "hello")? Interestingly enough, LUIS is able to determine the correct intent for this utterance, as we can see in Figure 2-p, with a confidence of 0.83 (83% certainty).

Figure 2-p: The Interactive Testing Screen

How cool is that? The **Welcome** intent has been correctly matched to the user's input.

Obviously, these are still small inputs, but without LUIS, it would have been very difficult to reach these results without a lot of programming logic—extensive use of regular expressions and conditions in our bot's code.

Now that we've seen what LUIS can do for us, how do we actually integrate this into our bot application and make use of it? This is what we'll explore next.

Integrating LUIS into our bot

Isn't it cool that we've managed to create two LUIS intents in a few minutes and can now add some natural language processing capabilities to our bot? I think it's awesome and mind-blowing. So, what do we need to do now?

After we have taken care of configuring the LUIS side of things, we can now go back to Visual Studio and our bot code to make the appropriate changes so we can use this framework.

The first thing we're going to do is to create a new dialog, which is going to be called `LuisDialog`. In Visual Studio, open the **Solution Explorer**, and under the **Dialogs** folder, create a new class with this name.

Our bot's project structure should now look as follows.
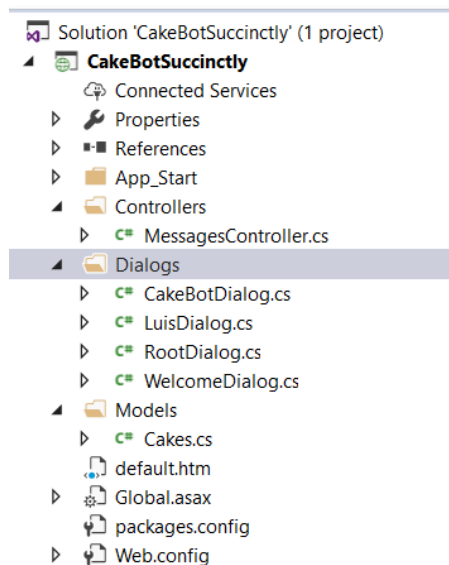
Figure 2-q: The Bot's Project Files after Adding LuisDialog

Once added, open *LuisDialog.cs*. The first thing we are going to do is to decorate this class with the necessary attributes.

In order to do that, we need to define it as a LUIS model—so we're going to put in `LuisModel` at the top of the class as a decorator.

This attribute will take in two parameters—one is the LUIS `App ID`, and the other is the LUIS programmatic `API Key`. These values can be found on the LUIS website—we'll look at this shortly, but first let's focus on the code.

Code Listing 2-a: The LuisDialog Class

```
using CakeBotSuccinctly.Models;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Luis;
```

```csharp
using System;

namespace CakeBotSuccinctly.Dialogs
{
    [LuisModel("<< App ID >>", "<< API Key >>")]
    [Serializable]
    public class LuisDialog : LuisDialog<Cakes>
    {
    }
}
```

As with any other dialog, we need to mark the class with the **Serializable** attribute. It is also important that this class must inherit from **LuisDialog<Cakes>** instead of **IDialog**.

Remember that **Cakes** contains our bot's model and FormFlow process. We now have the basic structure of our **LuisDialog** class, so let's write the methods that correspond to the intents we've defined in our LUIS app.

First, let's define the method that corresponds to this intent. We can do this as follows.

Code Listing 2-b: The Welcome Intent Method

```csharp
[LuisIntent("Welcome")]
public async Task Welcome(IDialogContext context, LuisResult res)
{
    await Task.Run(() => context.Call(new WelcomeDialog(), Callback));
}
```

As we can see, all we need to do is to decorate the **Welcome** method with the **LuisIntent** attribute—this is how the Bot Framework is able to execute this method when LUIS has correctly identified the user's intent as **Welcome**.

The logic inside the method itself is quite straightforward and simple. All it does is invoke **context.Call** and create an instance of **WelcomeDialog**, passing a **Callback** method as a parameter.

That was quite easy, wasn't it? Before we write the method that corresponds to the **Order** intent, let's take care of a few basic class constructs that we need to have in place.

Code Listing 2-c: Basic Class Constructs

```csharp
private readonly BuildFormDelegate<Cakes> OrderCake;

[field: NonSerialized()]
protected Activity _msg;

protected override async Task MessageReceived(IDialogContext context,
    IAwaitable<IMessageActivity> item)
{
```

```
    _msg = (Activity)await item;
    await base.MessageReceived(context, item);
}

public LuisDialog(BuildFormDelegate<Cakes> orderCake)
{
    OrderCake = orderCake;
}

[LuisIntent("")]
public async Task None(IDialogContext context, LuisResult res)
{
    await context.PostAsync(Str.cStrDontUnderstand);
    context.Wait(MessageReceived);
}
```

So, what are these class constructs for? Let's start from the bottom rather than the top. We can see that there's a **None** method that has been decorated with an empty **LuisIntent** attribute.

The **None** method will get executed when LUIS cannot confidently identify any of the user's intents and cannot match either the **Welcome** or **Order** intents.

Next (from the bottom up), we have the **LuisDialog** constructor. The constructor initializes the **OrderCake** delegate, which will be used for the **Order** intent, in order to run the FormFlow process.

There's a **MessageReceived** method that is responsible for awaiting user responses, and it simply invokes the **base** method, with the same name of the **LuisDialog** class.

This **MessageReceived** method uses a **_msg** variable, which is non-serializable, as it is for internal use only. This variable is not exposed by the bot to users.

Finally, we have the **OrderCake** delegate that will be used within the **Order** method to kick off the FormFlow process.

With this logic in place, the only bit we are missing is the method corresponding to the **Order** intent itself, which looks like this:

Code Listing 2-d: The Order Intent Method

```
[LuisIntent("Order")]
public async Task Order(IDialogContext context, LuisResult res)
{
    await Task.Run(async () =>
    {
        bool f = false;
        var r = Validate.ValidateWords(res.Query.ToLower());

        if (r.v) f = true;
```

```
        if (f)
        {
            var cakeOrderForm = new FormDialog<Cakes>(new Cakes(),
                    OrderCake, FormOptions.PromptInStart);
            context.Call(cakeOrderForm, Callback);
        }
        else
         {
            await context.PostAsync(Str.cStrDontUnderstand);
                context.Wait(MessageReceived);
        }
    });
}
```

As we can see, the **Order** method has been decorated with the **Order** attribute, which makes the Bot Framework route to it any user intents that correspond to a cake order.

Inside the method itself, what happens is that we pre-validate and check if the user's input corresponds to a potential cake order. This is done with the **ValidateWords** method, which is part of the **Validate** class of *Cakes.cs*.

Notice that we haven't yet implemented the **ValidateWords** method—we'll do that later.

If the user's input has been successfully pre-validated—which means that the user's input does indeed contain words that correspond to a cake order (this is the purpose of the **ValidateWords** method), then the FormFlow process is initiated. This is done by creating a new instance of **FormDialog** and then invoking **context.Call**.

If the user's input has not been pre-validated—meaning that it doesn't contain words that correspond to a cake order—then the bot returns a message back to the user saying that the input is not understood. This is represented by **Str.cStrDontUnderstand**, which is invoked by **context.PostAsync**.

To wrap up the **LuisDialog** class, the only thing we are missing is the implementation of the **Callback** method invoked by the **Welcome** and **Order** methods. Let's go ahead and implement it as follows.

Code Listing 2-e: The Callback Method

```
private async Task Callback(IDialogContext context, IAwaitable<object>
result)
{
    await Task.Run(() => context.Wait(MessageReceived));
}
```

The **Callback** method executes **context.Wait** by passing the **MessageReceived** method we previously declared as a parameter.

That wraps up all the logic of the **LuisDialog** class. The Code Listing below contains the full source code of this class.

Code Listing 2-f: The LuisDialog Class

```csharp
using System;
using CakeBotSuccinctly.Models;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
using Microsoft.Bot.Connector;
using System.Threading.Tasks;

namespace CakeBotSuccinctly.Dialogs
{
    [LuisModel("<< App ID >>", "<< API Key >>")]
    [Serializable]
    public class LuisDialog : LuisDialog<Cakes>
    {
        private readonly BuildFormDelegate<Cakes> OrderCake;

        [field: NonSerialized()]
        protected Activity _msg;

        protected override async Task MessageReceived(IDialogContext
        context, IAwaitable<IMessageActivity> item)
        {
            _msg = (Activity)await item;
            await base.MessageReceived(context, item);
        }

        public LuisDialog(BuildFormDelegate<Cakes> orderCake)
        {
            OrderCake = orderCake;
        }

        [LuisIntent("")]
        public async Task None(IDialogContext context, LuisResult res)
        {
            await context.PostAsync(Str.cStrDontUnderstand);
            context.Wait(MessageReceived);
        }

        [LuisIntent("Welcome")]
        public async Task Welcome(IDialogContext context, LuisResult res)
        {
            await Task.Run(() => context.Call(
                new WelcomeDialog(), Callback));
        }
```

```csharp
        private async Task Callback(IDialogContext context,
        IAwaitable<object> result)
        {
            await Task.Run(() => context.Wait(MessageReceived));
        }

        [LuisIntent("Order")]
        public async Task Order(IDialogContext context, LuisResult res)
        {
            await Task.Run(async () =>
            {
                bool f = false;
                var r = Validate.ValidateWords(res.Query.ToLower());

                if (r.v) f = true;

                if (f)
                {
                    var cakeOrderForm = new FormDialog<Cakes>(new
                        Cakes(), OrderCake, FormOptions.PromptInStart);
                    context.Call(cakeOrderForm, Callback);
                }
                else
                {
                    await context.PostAsync(Str.cStrDontUnderstand);
                    context.Wait(MessageReceived);
                }
            });
        }
    }
}
```

Now we almost have a working bot using LUIS, but we have a couple of things pending.

One is to implement the **ValidateWords** method within the **Validate** class of *Cakes.cs*, and another is to actually route all requests within the **Post** method of *MessagesController.cs* to the **LuisDialog** class.

Finalizing the Validate class

Our **Validate** class is still missing an important part—the **ValidateWords** method—which will be used in order to pre-validate and check if the input entered by the user is a possible cake order.

Let's go ahead and add it. Within Visual Studio, go to the **Solution Explorer** and open the *Cakes.cs* file. Within the **Validate** class, add the following code.

```
public static (bool v, string t) ValidateWords(string wrds)
{
    var r = (v: false, t: string.Empty);
    string[] wd = wrds.Split(' ');

    foreach (string w in wd)
    {
        r = TypeExists(w, Str.CakeTypes);
        if (r.v) break;
    }

    return r;
}
```

What this method does is quite simple—it checks if the user's input, represented by the **wrds** variable, is one of the valid **CakeTypes**. This is done with the program-defined **TypeExists** method in the **Validate** class.

With this method added, the **Validate** class now looks as follows.

Code Listing 2-h: The Updated Validate Class

```
public class Validate
{
    public static string DeliverType = string.Empty;

    public static (bool v, string t) TypeExists(string c, string[] types)
    {
        bool v = false;
        string t = string.Empty;

        foreach (string ct in types)
        {
            if (ct.ToLower().Contains(c.ToLower()))
            {
                v = true;
                t = ct;

                break;
            }
        }

        return (v, t);
    }

    public static (bool v, string t) ValidateWords(string wrds)
    {
```

```
        var r = (v: false, t: string.Empty);
        string[] wd = wrds.Split(' ');

        foreach (string w in wd)
        {
            r = TypeExists(w, Str.CakeTypes);
            if (r.v) break;
        }

        return r;
    }

    public static ValidateResult ValidateType(Cakes state,
        string value, string[] types)
    {
        ValidateResult result = new ValidateResult { IsValid = false,
            Value = string.Empty };

        var res = TypeExists(value, types);

        string r = $"{Str.cStrDeliverBy} {res.t}";
        DeliverType = res.t;

        return (res.v) ?
            new ValidateResult { IsValid = true, Value = res.t,
            Feedback = res.t } : result;
    }
}
```

Adjusting the Post method

Now we need to adjust the **Post** method of the *MessagesController.cs* file in order to trigger the **LuisDialog** class. The **Post** method currently looks as follows.

Code Listing 2-i: The Current Post Method

```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () =>
            Dialogs.CakeBotDialog.dialog);
    }
    else
    {
        HandleSystemMessage(activity);
    }
```

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

What we must do now is change **Dialogs.CakeBotDialog.dialog** for something else—which will be an entry point to a **LuisDialog** instance. We can achieve this by creating a new method called **MakeLuisDialog**, which we can pass as the second parameter to the **Conversation.SendAsync** method.

We could do this directly on **Conversation.SendAsync**, without creating a separate method—but that would be a bit messy, and more difficult to understand.

Code Listing 2-j: The MakeLuisDialog Method

```
private IDialog<Cakes> MakeLuisDialog()
{
    return Chain.From(() => new LuisDialog(Cakes.BuildForm));
}
```

The **MakeLuisDialog** method creates a chain from a new instance of the **LuisDialog** class by invoking **Cakes.BuildForm**—which returns an **Iform<Cakes>** object, consumed by **Chain.From**—which kicks off the **LuisDialog** logic within our bot.

Let's change our **Post** method to reflect this.

Code Listing 2-k: The "LUIS-aware" Post Method

```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, MakeLuisDialog);
    }
    else
    {
        HandleSystemMessage(activity);
    }

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

We have now finalized everything we need in order to adapt our bot to do some natural language processing using LUIS.

All we need to do now is test it using the emulator—this is what we'll do next.

The full source code for this chapter can be found here.

53

Testing our LUIS-enabled bot

We've converted our bot into a more advanced one by giving it LUIS superpowers.

Now, let's test it out using the emulator to see if the correct method—and subsequently, conversational dialog—is triggered whenever the user enters a new intent. We want to see how the conversation flows compared to how it was before.

First, we'll add the LUIS App ID and App Key. Let's go back to the LUIS website and fetch them.

The App ID is easy to see immediately. Once you've logged in to LUIS and clicked on your app, it's just below the Overview header, as seen in the following figure.
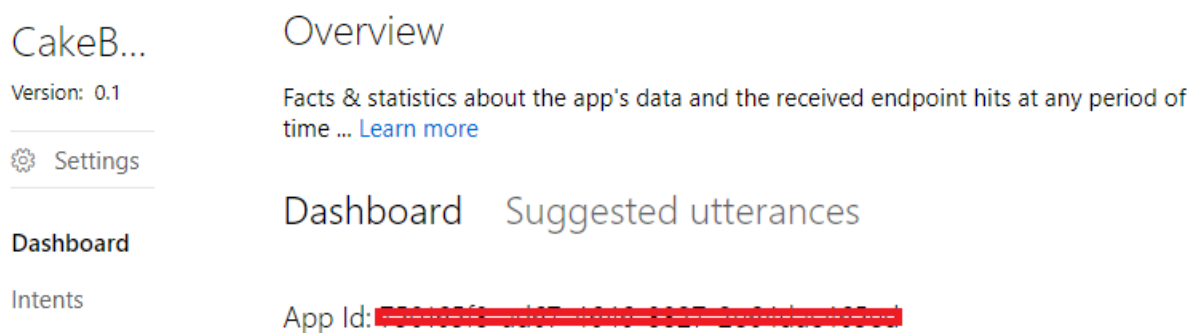
Figure 2-r: The LUIS App ID

You can find the Programmatic API Key or App Key by clicking on the **My keys** option from the top menu.
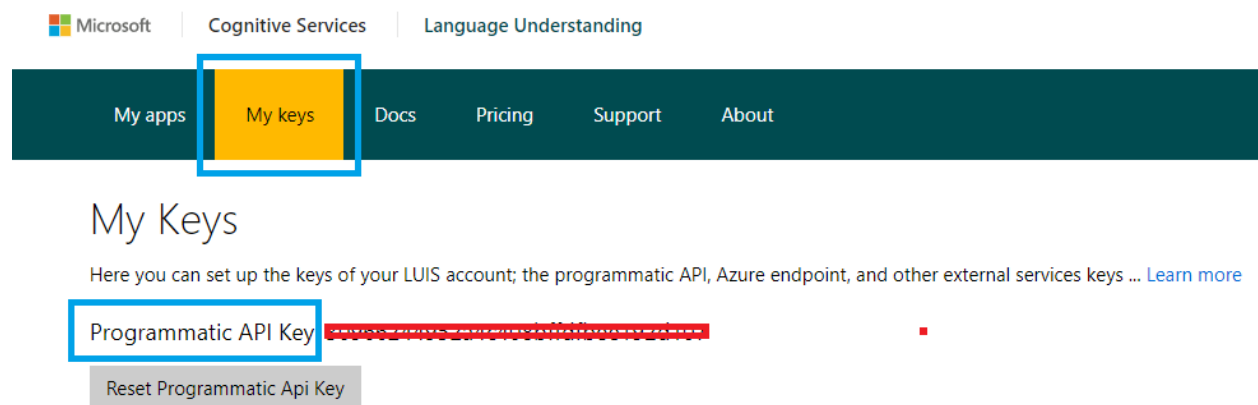

Figure 2-s: The LUIS App Key

Copy and paste them onto the placeholders defined on the `LuisDialog` class decorator, as shown in Code Listings 2-a and 2-f.

Now we are ready to run a quick test using the emulator, so open it up. In Visual Studio, click **Run**.
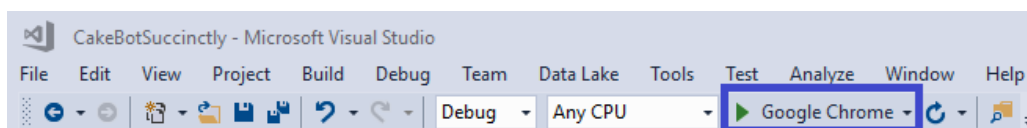

Figure 2-t: Run Button in Visual Studio

Once the application is running, go to the emulator, and let's greet our bot by typing **hi**. We'll see what is shown in Figure 2-u.
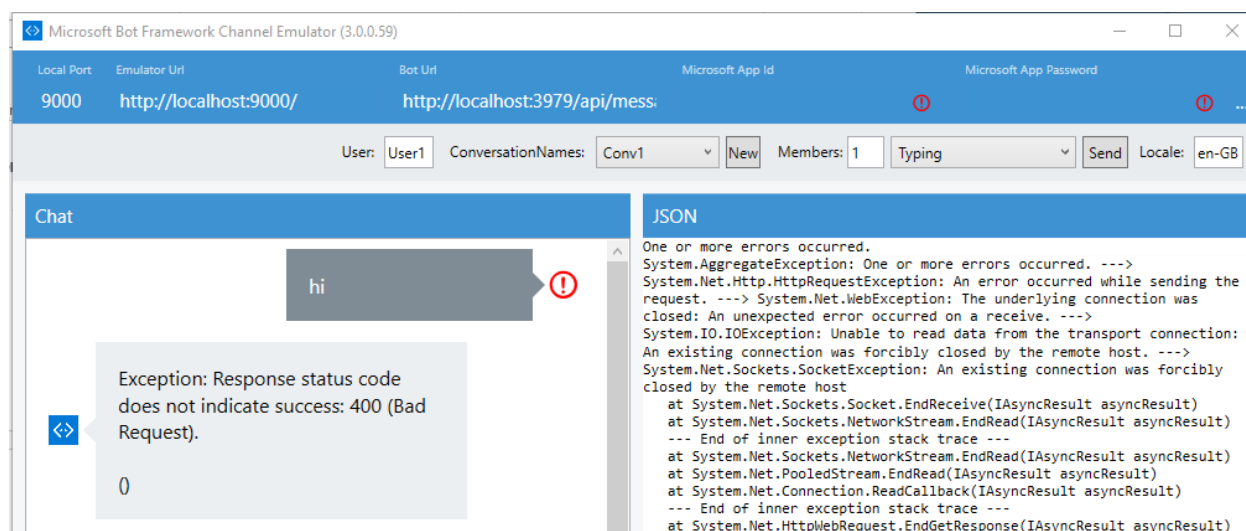

Figure 2-u: Oops—A Bot Exception

What happened here? If our code is correct—which it certainly is—why is a bad request being returned when the bot runs?

The reason for this bad request is that when the bot tries to connect to the web endpoint that represents the LUIS service indicated by the App ID and App Key, it is unable to get any response back—thus resulting in a 404 error.

This is due to the fact that the LUIS endpoint has not been published. To solve this, go back to the LUIS UI, and under the **Publish App** option, click **Publish**. This will make the LUIS endpoint publicly available to our code.
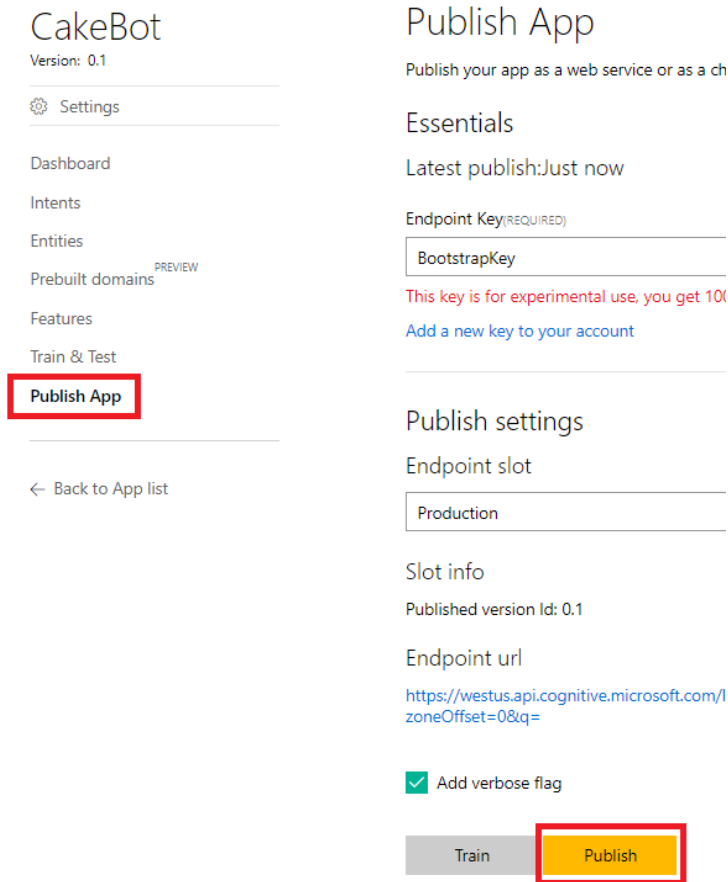
Figure 2-v: Publish Option in LUIS

Once published, we can try again. Let's now type in **hola** (which means "hello" in Spanish) to see if the correct method of the `LuisDialog` class is executed.

I've set a breakpoint on the `Welcome` method of the `LuisDialog` class to see if this LUIS intent is executed when I type in this word.
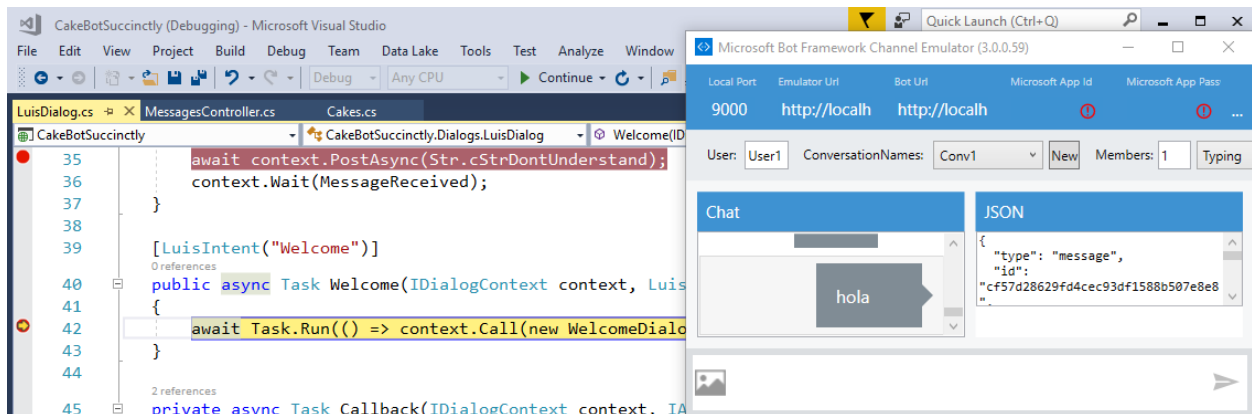

Figure 2-w: Welcome Intent Execution

In Figure 2-w, we can clearly see that the word "hola" has correctly triggered the execution of the `Welcome` method—which means it has been identified by LUIS as the right intent.

That's the magic of LUIS: without breaking our heads, and without having to write complex regex routing conditions—by simply using the `LuisIntent` method decorator—we are able to tell LUIS which method it needs to execute when the user's intent has been correctly identified.

If we now press **F10** in Visual Studio, we should be able to follow the bot's conversational logic. The `WelcomeDialog` class logic will be executed first, and shortly after, the FormFlow logic, where a cake order can be placed.
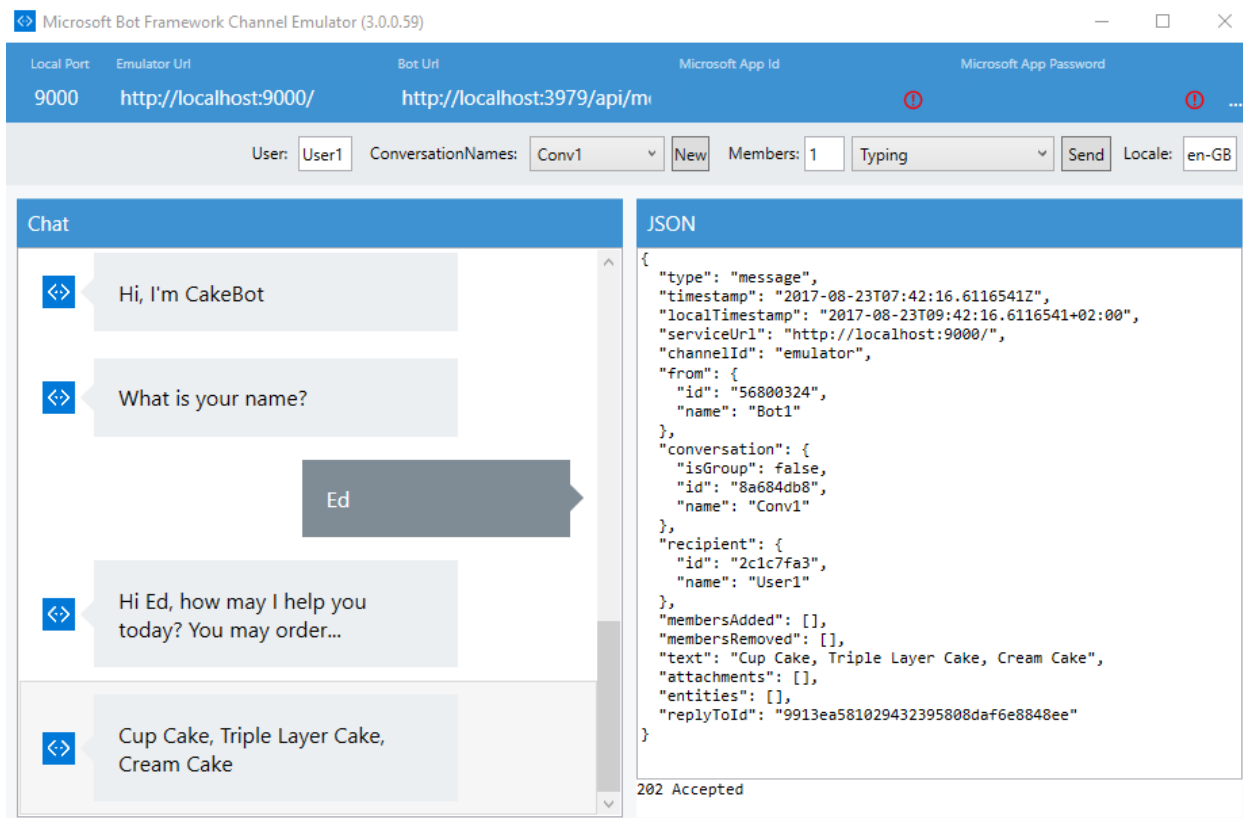

Figure 2-x: WelcomeDialog Execution

In Figure 2-x, we clearly see the full execution of the logic contained within the `WelcomeDialog` class.

At this stage, the bot is waiting for the user's input and expecting to pre-validate the type of cake the user is interested in, which should trigger the execution of the `Order` method of the `LuisDialog` class.

So, let's type in **cup cake** in order to continue the conversation. Once the `Order` method of the `LuisDialog` class has correctly pre-validated *cup cake* as a valid response, the FormFlow process kicks in, as can be seen in the following figure.
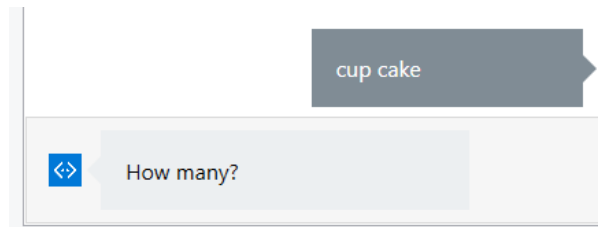
Figure 2-y: FormFlow Started

The bot is now executing the logic of the **BuildForm** method contained within the **Cakes** class of *Cakes.cs*. When we typed in **cup cake**, LUIS was able to determine that the word "cake" matches the **Order** intent.

LUIS was able to infer this because we entered multiple utterances (sentences) for this intent using the word *cake*, such as *deliver a cake*, on the LUIS UI.

What would have happened if instead we had typed in *triple layer*, *cream,* or simply *cup*?

The answer is quite simple—basically the **Order** method would not have been triggered, thus not initiating the FormFlow process. This is because neither *triple layer*, *cream*, or *cup* are utterances of the **Order** intent—they have not been defined yet.

Instead, the **None** or **Welcome** methods of the **LuisDialog** class would have been triggered, depending on the score LUIS might have given to the user's input.

So, in order to overcome this and make our bot a bit more robust, we can add these small utterances: *triple layer*, *cream*, or *cup* to the **Order** intent within the LUIS UI.

By taking this approach and testing various scenarios, using different keywords, it is possible to fine-tune LUIS and make our bot more robust—which, in essence, means that the bot will be able to respond better to user requests—by being able to match better user inputs to intents, thus triggering the right method within the **LuisDialog** class.

Let's continue the conversation by answering the questions from the FormFlow process.
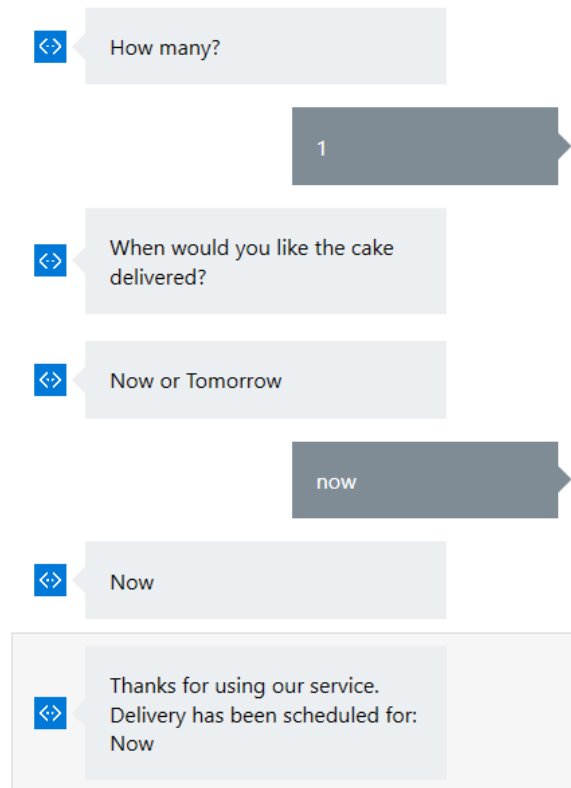
Figure 2-z: FormFlow Finished

As we can see in Figure 2-z, the `BuildForm` method—responsible for running the FormFlow process—of the `Cake` class (*Cakes.cs*) has been fully executed and the conversation has been finalized.

Summary

Throughout this chapter we've seen how integrating LUIS capabilities into our bot will drastically reduce the complexity of routing user requests to the correct conversational dialogs.

By using LUIS and adding a few decorators to some of our bot's methods, we are able to easily make our bot execute the right method in order to trigger the appropriate dialog or FormFlow process.

In the next chapter, we'll explore how we can use the QnA Maker Service in order to build, train, and publish a question and answer bot based on FAQ URLs, or structured lists of questions and answers.

Chapter 3 QnA Bots

Quick intro

So far, we've explored how to create a basic Skype bot based on regular expressions, and then added natural language processing capabilities to it using LUIS—which really gave some nice superpowers to our bot and simplified the process of routing user requests.

But what if we want to build a bot based on a series of questions and answers? The QnA Maker Service from Microsoft—which, like LUIS, is part of the Cognitive Services suite, allows us to quickly build, train, and publish question-and-answer bots based on FAQ URLs or structured lists of questions and answers.

Just like LUIS, the QnA Maker Service exposes endpoints that can be consumed by a bot application built with the Bot Framework, such as our Skype bot.

Even with LUIS, making a fully fluid conversational bot is not easy. Creating a conversationally fluid bot would require entering a lot of intents, entities, and utterances, which would have to be followed up in the code by dialogs and specific FormFlow logic. This gradually builds up into complexity.

Although bot applications could be developed for any particular subject of interest—such as checking for availability of airline tickets, or an assistant for an online shop—many people tend to think of bots as automated answering agents. In essence, you ask a question and expect an answer.

The QnA Maker Service takes this process to the next level as it abstracts an extra layer of complexity from us by building on top of LUIS. This allows us to create bots based on the notion of taking a knowledge base and converting it into pairs of questions and answers from FAQ URLs or documents.

By being able to auto-extract question and answer (QnA) pairs and using LUIS to infer various utterances from those pairs, the QnA Maker Service makes it easy to create bots that act as answering agents.

This sounds very exciting—and as you'll see shortly, it's easier than you think.

Creating the knowledge base

The first step to creating a QnA bot is to create a knowledge base, which is nothing more than a list of questions and answers.

Creating a knowledge base is as simple as pointing the tool to an existing source and ingesting the QnA content.

QnA Maker is able to auto-extract QnA pairs from most FAQ URLs and documents. If it is not able to auto-extract, it has an option to manually add and edit QnA pairs. So, let's give it a go.

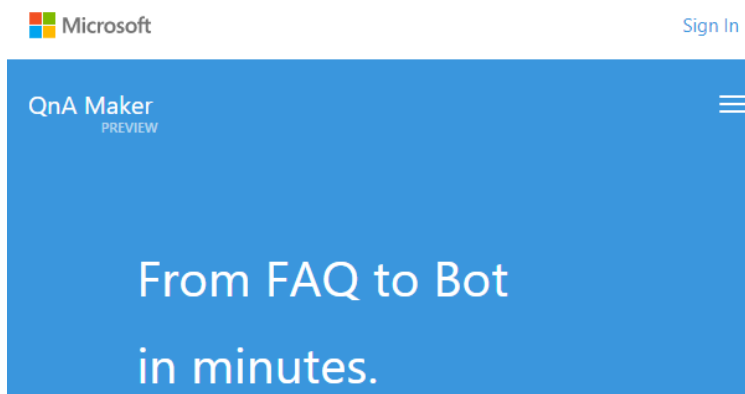Using your Microsoft account, go to the QnA Maker site and click **Sign In**.

Figure 3-a: QnA Maker Website

Once you've provided your credentials, you might be asked to confirm if QnA Maker can access some of your Microsoft account profile details. When prompted, click **Yes** in order to continue.

Given that QnA Maker is provided under Cognitive Services Terms, the free preview provides at the time of writing up to 10 transactions per minute (10K transactions per month), so you might be asked to agree to the these service terms in order to continue.

Next, click the **Create new service** option. We're going to create a bot that will answer questions related to Creative Commons; however, you may choose to use a totally different FAQ site or upload your own content.

When prompted, enter a name in the **SERVICE NAME** field and a URL in the **FAQ URL(S)** field.

Figure 3-b: A New QnA Service

There's also an option to manually upload a document—if no URLs are provided—that must contain QnA pairs. At the time of writing, QnA can consume .tsv, .txt, .docx, and .pdf files.

What the QnA Maker Service will do is to crawl the content—either the provided URLs or document—creating a knowledge base that will serve as the brain for our bot. The information crawled can be edited in the following step.

In order to create the knowledge base, click **Create**. The QnA Maker crawler will index the information provided and almost immediately provide a list of the questions and answers it was able to identify, which should look similar to the following figure.
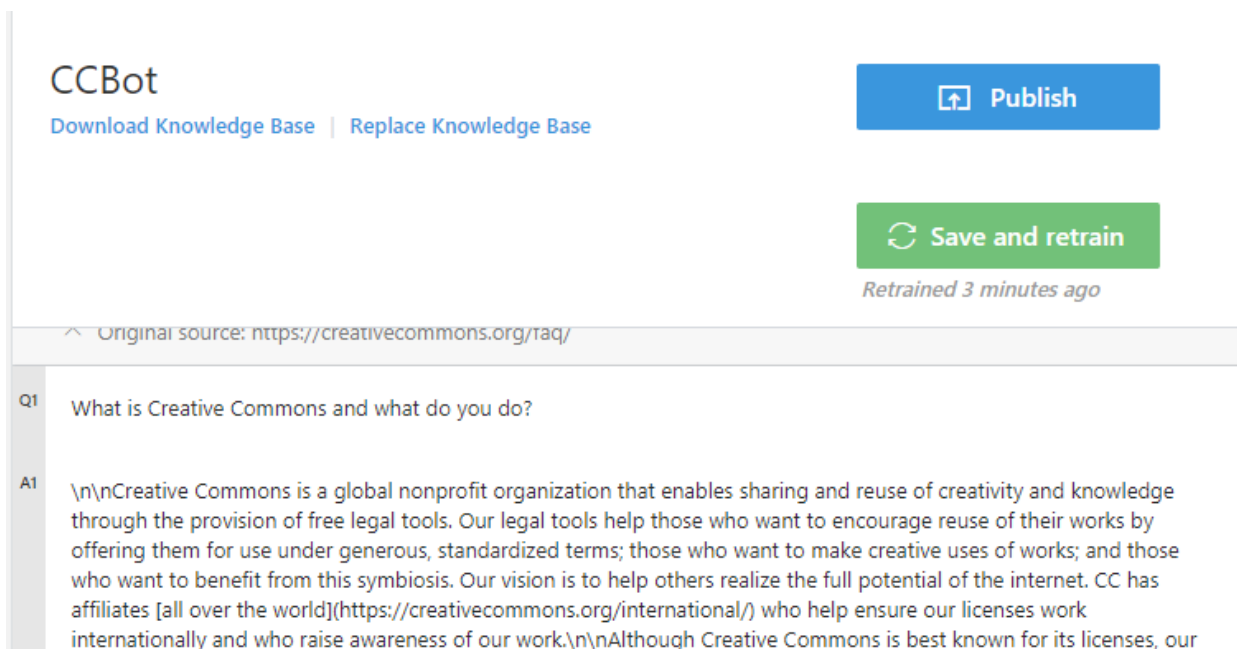

Figure 3-c: The QnA Service Results

If you scroll down the results, you'll see all the QnA pairs that the QnA Maker Service was able crawl, identify, and successfully parse in a relatively fast amount of time. It's quite impressive.

Notice though how, in this example, the answers extracted contain some markup. Before we click **Publish**, it is possible to go through each of the answers and manually edit and remove these markup entries. However, that could be a long, tedious, and time-consuming process—especially if there are a lot of answers, as in this case.

So, what we'll do is leave the content as it is and remove the markup in case we need to via code. Most conversational platforms are able to automatically parse markup, so that shouldn't be an issue.

If there's anything else that needs to be manually added, you can do it at this stage. You need to click on the **Save and retrain** button before publishing. Once you've clicked **Publish**, you'll be presented with the following information.
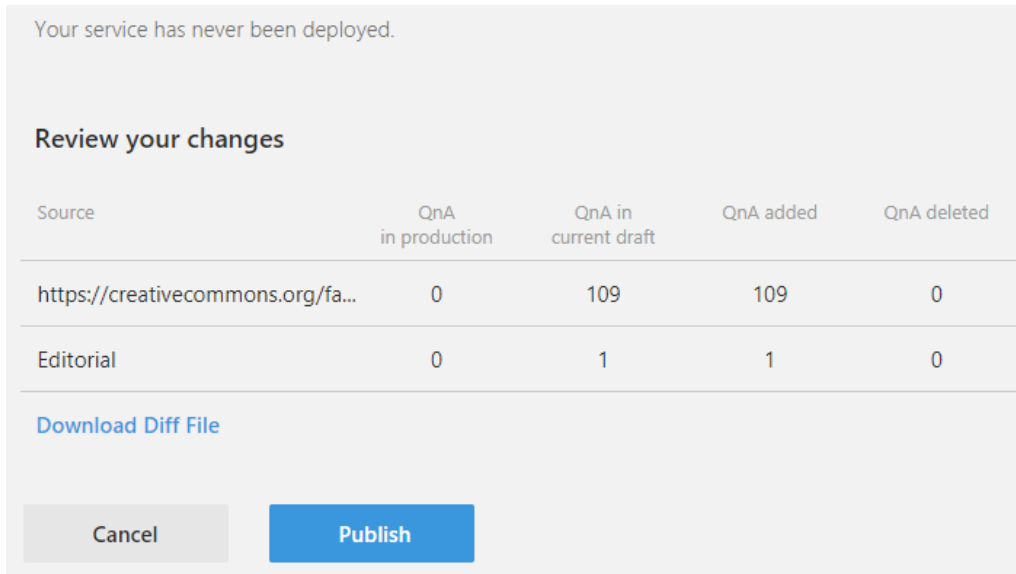
Figure 3-d: The QnA Service Results Recap

In order to finalize the publication of these QnA pairs, click **Publish**. You'll be presented with a screen similar to Figure 3-e.
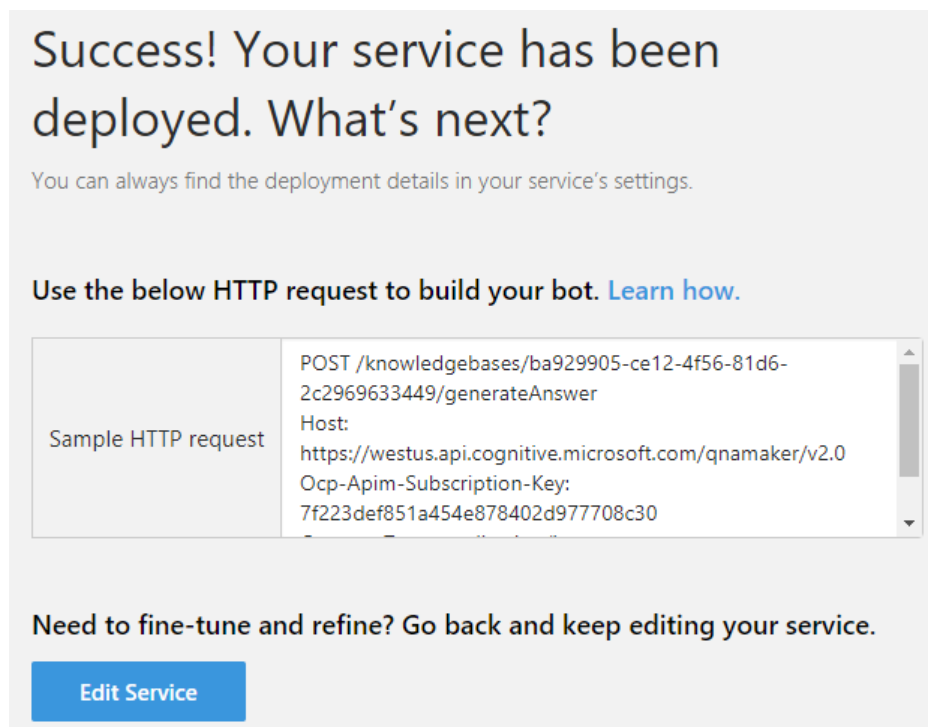

Figure 3-e: The QnA Service Results Deployed

We now have a QnA Service that we can use to create a Creative Commons bot.

We can also find this service under the **My services** option, as shown in the following figure.
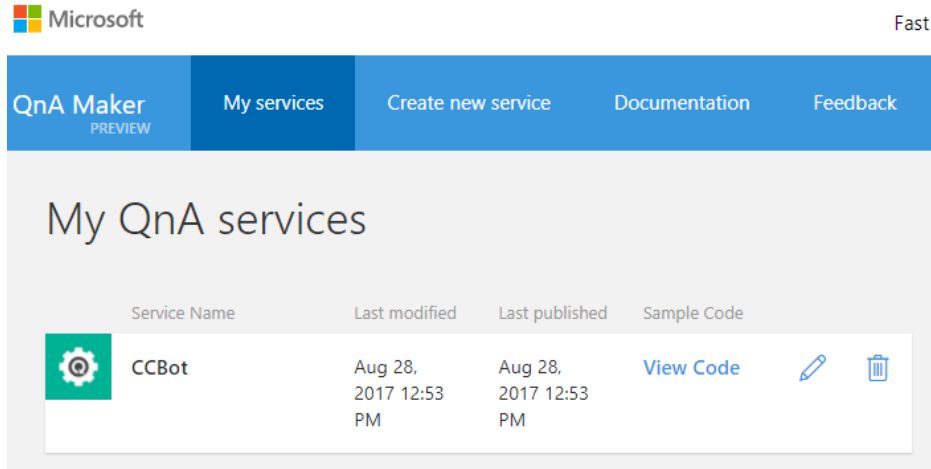
Figure 3-f: List of QnA Services

QnA Maker Dialog

Now that we have a knowledge base ready that we can use to create our bot, we need to shift our attention to writing some code.

The QnA Maker Dialog is a library available on NuGet that acts as a wrapper around a QnA Maker Service endpoint. Instead of directly communicating with our QnA service through the published HTTP endpoint, it's much easier to use the QnA Maker Dialog library.

Open Visual Studio and create a new bot application, using the previously installed Bot Template. Let's name this new bot application **CCBot** (using the same steps described in Figure 1-a).

Once the project has been created, go to **Solution Explorer** and right-click **References**, and then select **Manage NuGet Packages**.

Once the NuGet Package Manager window is open, click **Restore** to restore all NuGet packages that are missing from the Visual Studio project.
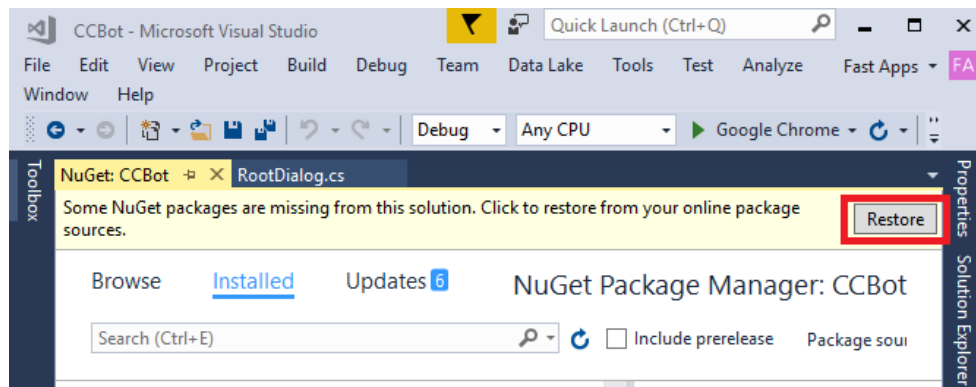

Figure 3-g: The Restore Button in NuGet

Once all the missing packages have been restored, type **QnA** in the search box—you might need to open the NuGet Package Manager window again, as it is possible it might have closed itself when the missing packages were restored. Select **QnAMakerDialog** by Gary Pretty.
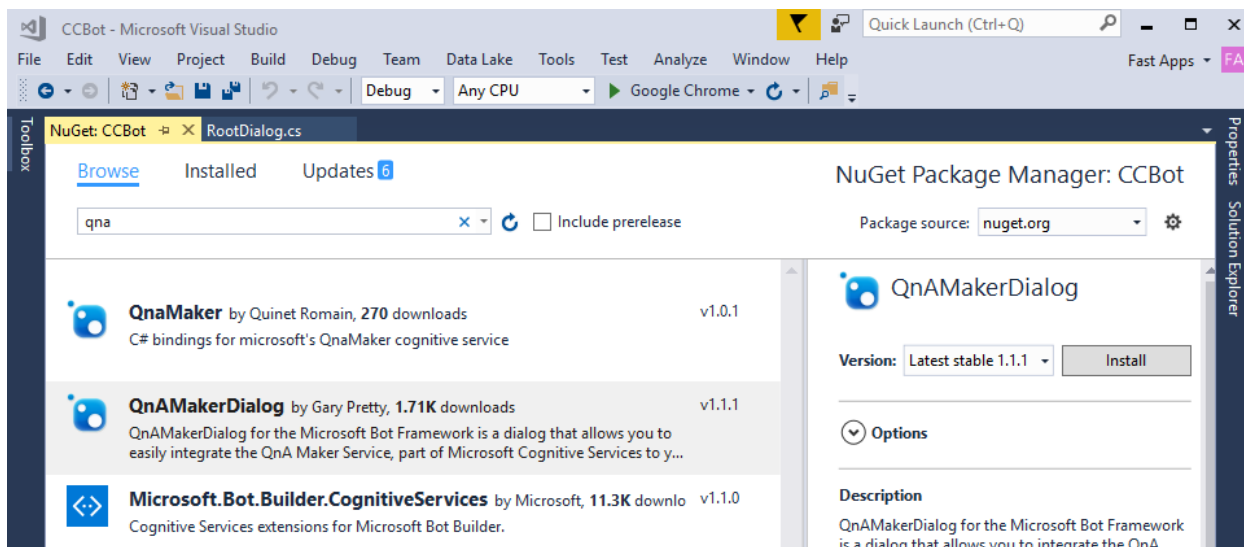

Figure 3-h: The QnAMakerDialog Package

Once the package has been installed, build the project and add any missing references. For this example, we'll be writing our code on the *RootDialog.cs* file.

So, let's open up *RootDialog.cs* in order to make the necessary adjustments to use this library. Remove the code that comes out of the box and replace it with the following code.

Code Listing 3-a: The New RootDialog Class

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using QnAMakerDialog;

namespace CCBot.Dialogs
{
    [Serializable]
    [QnAMakerService("SUBSCRIPTION_KEY", "KNOWLEDGE_BASE_ID")]
    public class RootDialog : QnAMakerDialog<object>
    {
        public override async Task NoMatchHandler(
            IDialogContext context, string originalQueryText)
        {
            await context.PostAsync(
            $"Couldn't find an answer for '{originalQueryText}'.");
            context.Wait(MessageReceived);
        }
```

```
        [QnAMakerResponseHandler(50)]
        public async Task LowScoreHandler(IDialogContext context, string
            originalQueryText, QnAMakerResult result)
        {
            await context.PostAsync(
            $"Found an answer that could help...{result.Answer}.");
            context.Wait(MessageReceived);
        }
    }
}
```

Let's try to understand what is going on here. This new **RootDialog** class now inherits from **QnAMakerDialog<object>**.

The QnA Maker Dialog library allows us to take incoming text messages from the bot, send them to the published QnA Maker Service, and send the answer sent back from the service to the bot as a reply. It basically acts as a wrapper around the QnA service endpoint and a bridge between the bot and the knowledge base.

When no matching answer in the knowledge base can be found, the **NoMatchHandler** method is executed. This can be overridden (as we've done in Code Listing 3-a) in order to send our own customized message.

Although the default implementation is already good enough for most developers, it is also possible to provide a slightly stronger way of doing things by defining a custom handler and decorating it with a **QnAMakerResponseHandler** attribute, indicating the maximum score to which the handler should respond.

Code Listing 3-a contains a custom handler that is executed when the confidence score is below 50—this is what the **LowScoreHandler** method does. We can add as many custom handlers as needed. Any scores above 50 will be handled the default way, and the appropriate response determined by the QnA service will be returned to the user.

That's all there really is to this. We just need to do one thing—which is to add the subscription key and knowledge base identifier.

We can find both by clicking on the **View Code** link, as seen in Figure 3-f. Once you click **View Code**, you'll see something like this.

Figure 3-i: The QnA HTTP Request Example

The first box highlighted corresponds to the knowledge base identifier, and the second one to the subscription key. Copy them over to the previous code listing and replace each respective placeholder.

Let's build our Visual Studio project, run it, and then open the emulator in order to test it out. The second question on the Creative Commons FAQ page is about copyright, as can be seen in the following figure.

Figure 3-j: The Creative Commons FAQ Page

Let's type in a question that uses the word "copyright" (or the word itself) to see what result the QnA Maker Dialog returns as a response through the bot.
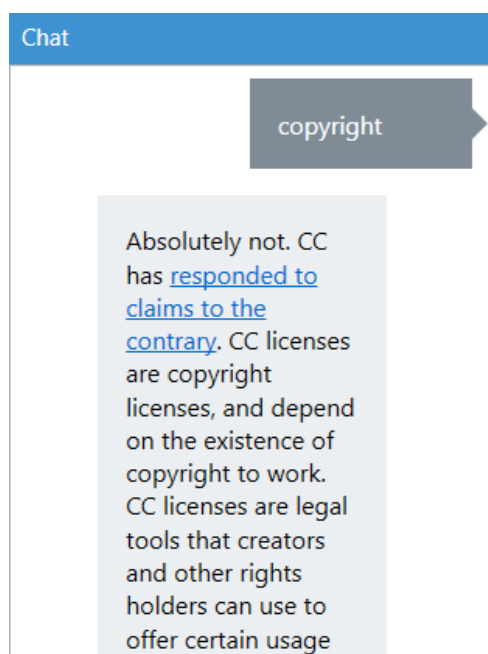

Figure 3-k: Testing the Bot Using the QnA Service

As we can see, the QnA Service was able to return the answer that actually corresponds to the question highlighted in Figure 3-j.

Is Creative Commons against copyright?

Absolutely not. CC has responded to claims to the contrary. CC licenses are copyright licenses, and depend on the existence of copyright to work. CC licenses are legal tools that creators and other rights holders can use to offer certain usage rights to the public, while reserving other rights. Those who want to make their work available to the public for limited kinds of uses while preserving their copyright may want to consider using CC licenses. Others who want to reserve all of their rights under copyright law should not use CC licenses.

Figure 3-l: The Response for the Question Tested with the Bot

How cool is that? With barely any code at all, we managed to create a bot that takes data from the Creative Commons FAQ and create a knowledge base that can be queried through a conversation using a bot.

Say we want to know what Creative Common does with its money. Let's try out this example to see if we get the correct result.

Where does the money go?

Creative Commons is a global nonprofit organization that enables sharing and reuse of creativity and knowledge through free legal tools, with affiliates all over the world who help ensure our licenses work internationally and
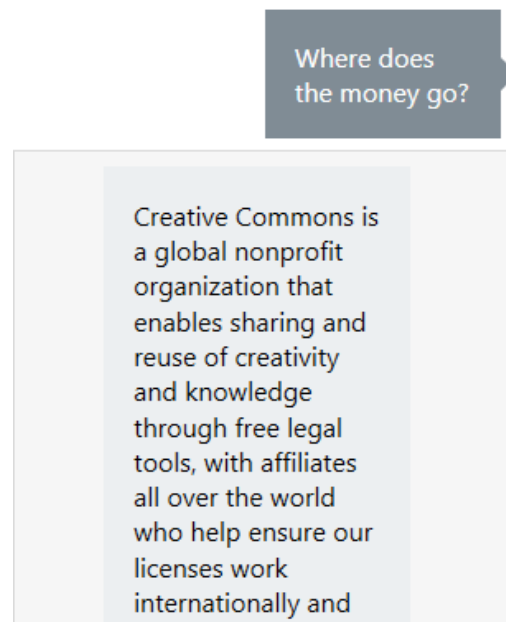
Figure 3-m: Another Bot Test using the QnA Service

Let's now look at the FAQ site and see if the bot's response matches the answer on the website.

Why does Creative Commons run an annual fundraising campaign? What is the money used for and where does it go?

Creative Commons is a global nonprofit organization that enables sharing and reuse of creativity and knowledge through free legal tools, with affiliates all over the world who help ensure our licenses work internationally and raise awareness about our work. Our tools are free, and our reach is wide.

Figure 3-n: The Response for the Question Tested with the Bot

As we can see, the QnA Service has been able to return the correct answer, as it was able to infer that the question *Where does the money go?* is actually the same thing as *What is the money used for and where does it go?*—which itself is part of a bigger question.

Summary

We can start to appreciate that the QnA Service literally took LUIS a few steps forward and abstracted even more complexity for us, making it simple and easy to create a conversational agent based on a knowledge base—and with barely any code. In my view, this is quite an impressive feat.

Like with anything, there's always the possibility to improve things and add extra logic, which can make this bot application even more robust and capable of handling more scenarios. However, this simple example suffices to demonstrate how far this technology has come, and how easy it is to create such a solution.

I have to finish by remarking that indeed Microsoft's marketing slogan for the QnA Service is probably one of the most accurate ones I've ever read: "From FAQ to Bot in minutes."

In the next chapter, we'll explore how we can take this example to the next level by using Scorables for global message handling and interrupting dialogs when required. This is a great technique that can allow us to achieve a lot of flexibility when designing a conversational solution on Skype.

The full source code for this chapter can be found here.

Chapter 4 Scorables

Quick intro

If you've been using the Bot Framework for some time, you might be familiar with dialogs and the dialog stack—where you have a root dialog, and you can pass the control to child dialogs, which will return the control back to the parent dialog.

This can give us a lot of flexibility when designing a conversational flow—for instance, when using LUIS to determine a user's intent and then falling back to standard dialogs if no intent can be recognized, or when a specific action (such as a greeting) needs to be executed—which is what we did in Chapter 2.

This is great, but there are times when we might want to be able to interrupt our current dialog stack in order to handle an incoming request, such as responding to specific user queries that aren't related to the content existing on the bot's knowledge base. This is where Scorables come into play.

You can think of Scorables as a special type of dialog that we can use within a conversational flow using the Bot Framework in order to interrupt the current dialog and handle an incoming request. Scorables act as global messaging handlers.

With these global messaging handlers, it is possible to have bots that decide how to handle a message and whether it needs to be handled at all. By using this feature, we are able to create bots that have better decision-making capabilities based on the score of an incoming message.

In this chapter, we'll expand on the previous QnA example in order to introduce some extra flexibility and processing capabilities to our bot.

The full source code for this chapter can be found [here](#).

The basics of Scorables

As the name implies, scorables are all about rating incoming messages and giving them a score. Basically, scorable dialogs monitor a bot's incoming messages and decide whether to handle each message. If a message should be handled, the Scorable will set a score between 0 and 1 (100%) as to what priority it should be given.

If a Scorable matches against an incoming message (and has the highest score if there are multiple matches), it can then handle the response to the user rather than being picked up by the current dialog in the stack. This is quite powerful and allows for a lot of flexibility.

In our previous example, we scratched the surface of this concept by briefly examining the `QnAMakerResponseHandler` decorator, which abstracts some of the complexity of implementing Scorables when using the QnA Maker Dialog.

Based on this concept and what we've done so far, let's expand our previous QnA example so that we can have the bot respond to FAQs in both English and French. This will be possible by using a Scorable.

Implementing a Scorable

The Bot Builder SDK for .NET uses AutoFac for inversion of control and dependency injection. One of the ways AutoFac is used is through Scorables.

To create a Scorable, we need to create a class that implements the **IScorable** interface by inheriting from the **ScorableBase** abstract class.

In order to have that Scorable applied to every message in the conversation, the bot needs to register the **IScorable** interface as a service with the conversation's container.

When a new message arrives to the conversation, it passes that message to each implementation of **IScorable** in order to get a score. The one with the highest score is then processed.

To extend our QnA example to support Creative Commons French FAQs, we need to create a separate dialog that points to a different QnA Maker Service that will be specific for FAQs written in French.

By default, the Scorable will pass on the conversation to the QnA dialog that corresponds to the QnA Maker Service for English FAQs, and if the Scorable detects the word *French*, it will then pass the control to the QnA dialog that corresponds to the QnA Maker Service for French FAQs.

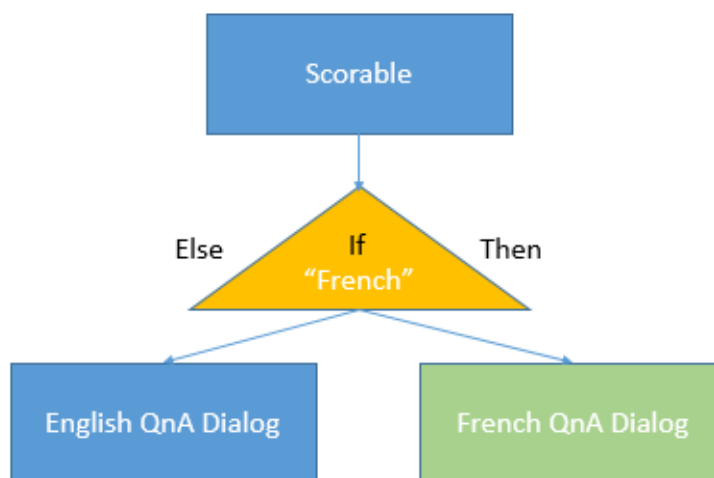Below is a graphical explanation of what we are trying to achieve.

Figure 4-a: The Scorable Logic

The first thing we need to do is create a **FaqSettingsDialog** that we'll add to the stack whenever the user responds with the word *French* in the conversation. Let's code this as follows.

```csharp
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using System;
using System.Threading.Tasks;

public class FaqSettingsDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        await context.PostAsync(
            "This is the FAQ Settings. Reply to go back");

        context.Wait(MessageReceived);
    }

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

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

You can add this code as a separate file under the **Dialogs** folder of your Visual Studio project and name it *FaqSettingsDialog.cs*. Think of the **FaqSettingsDialog** class as the decision part (triangle) in Figure 4-a.

For now, this **FaqSettingsDialog** class doesn't do anything—it's just a placeholder for the logic we'll be implementing, which we'll come back to in a bit.

Let's now focus our attention on the Scorable itself. Create a **FaqScorable** class, which should provide an implementation of the **ScorableBase** abstract class in order to implement the **IScorable** interface.

First things first—create a new C# class file under the **Dialogs** folder in your Visual Studio project called *FaqScorable.cs*, and let's start off by adding a **PrepareAsync** method—which will inspect the incoming message to check if it matches the text we are looking for (*French).*

If there's a match, we'll return the message to be used as state for scoring; otherwise, **null** is returned—which indicates no match. The code should look as follows.

Code Listing 4-b: The PrepareAsync Method of the FaqScorable Class

```
protected override async Task<string> PrepareAsync(IActivity activity,
    CancellationToken token)
{
    var message = activity as IMessageActivity;

    if (message != null && !string.IsNullOrWhiteSpace(message.Text))
        if (message.Text.ToLower().Equals("french",
            StringComparison.InvariantCultureIgnoreCase))
            return message.Text;

    return null;
}
```

We now need to implement a **HasScore** method, which is invoked by the calling component in order to determine if the Scorable has a score—if there's a match. We can implement this method as follows.

Code Listing 4-c: The HasScore Method of the FaqScorable Class

```
protected override bool HasScore(IActivity item, string state)
{
    return state != null;
}
```

Next, we need a **GetScore** method. This will be invoked to get the score for the Scorable, which will be all other Scorables that have a score. We can implement it as follows.

Code Listing 4-d: The GetScore Method of the FaqScorable Class

```
protected override double GetScore(IActivity item, string state)
{
    return 1.0;
}
```

The way this will work is that the Scorable with the highest score will process the message when the **PostAsync** method is invoked.

Within the **PostAsync** method, we'll add **FaqSettingsDialog** to the stack, so it can become the active dialog. The **PostAsync** method looks as follows.

Code Listing 4-e: The PostAsync Method of the FaqScorable Class

```
protected override async Task PostAsync(IActivity item, string state,
    CancellationToken token)
{
```

```
    var message = item as IMessageActivity;

    if (message != null)
    {
        var settingsDialog = new FaqSettingsDialog();
        var interruption = settingsDialog.Void<object,
            IMessageActivity>();

        task.Call(interruption, null);
        await task.PollAsync(token);
    }
}
```

The missing part is that when the scoring process is complete, we must invoke the **DoneAsync** method, where all the resources used in the scoring process are released. This is what we'll implement next.

Code Listing 4-f: The DoneAsync Method of the FaqScorable Class

```
protected override Task DoneAsync(IActivity item, string state,
CancellationToken token)
{
    return Task.CompletedTask;
}
```

This concludes the **FaqScorable** class. The full code of this class is listed in the following Code Listing.

Code Listing 4-g: FaqScorable.cs

```
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Builder.Internals.Fibers;
using Microsoft.Bot.Builder.Scorables.Internals;
using Microsoft.Bot.Connector;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace CCBot.Dialogs
{
    public class FaqScorable : ScorableBase<IActivity, string, double>
    {
        private readonly IDialogTask task;

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

```csharp
protected override async Task<string> PrepareAsync(
    IActivity activity,
    CancellationToken token)
{
    var message = activity as IMessageActivity;

    if (message != null &&
        !string.IsNullOrWhiteSpace(message.Text))
    {
        if (message.Text.ToLower().Equals("french",
            StringComparison.InvariantCultureIgnoreCase))
        {
            return message.Text;
        }
    }

    return null;
}

protected override bool HasScore(IActivity item, string state)
{
    return state != null;
}

protected override double GetScore(IActivity item, string state)
{
    return 1.0;
}

protected override async Task PostAsync(IActivity item,
    string state, CancellationToken token)
{
    var message = item as IMessageActivity;

    if (message != null)
    {
        var settingsDialog = new FaqSettingsDialog();

        var interruption = settingsDialog.Void<object,
            IMessageActivity>();

        task.Call(interruption, null);

        await task.PollAsync(token);
    }
}

protected override Task DoneAsync(IActivity item, string state,
```

```
            CancellationToken token)
        {
            return Task.CompletedTask;
        }
    }
}
```

We've now implemented our Scorable, but we need to register it—this is what we'll do next. But before we do that, let's implement a **CancelScorable** in the same way we've implemented **FaqScorable**, with the only difference being that it resets the dialog stack when the Scorable is called.

So, create a new C# class file called *CancelScorable.cs*, and place it under the **Dialogs** folder of your Visual Studio project.

Code Listing 4-h: The CancelScorable Class

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Builder.Internals.Fibers;
using Microsoft.Bot.Connector;
using Microsoft.Bot.Builder.Scorables.Internals;

namespace CCBot.Dialogs
{
    public class CancelScorable : ScorableBase<IActivity, string, double>
    {
        private readonly IDialogTask task;

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

        protected override async Task<string> PrepareAsync(
            IActivity activity, CancellationToken token)
        {
            var message = activity as IMessageActivity;

            if (message != null &&
                !string.IsNullOrWhiteSpace(message.Text))
            {
                if (message.Text.ToLower().Equals("cancel",
                    StringComparison.InvariantCultureIgnoreCase))
                {
                    return message.Text;
                }
            }
```

```
            }

            return null;
        }

        protected override bool HasScore(IActivity item, string state)
        {
            return state != null;
        }

        protected override double GetScore(IActivity item, string state)
        {
            return 1.0;
        }

        protected override async Task PostAsync(IActivity item,
            string state, CancellationToken token)
        {
            task.Reset();
        }
        protected override Task DoneAsync(IActivity item,
            string state, CancellationToken token)
        {
            return Task.CompletedTask;
        }
    }
}
```

Registering a Scorable

To register the Scorable that we've created, we need to create a
GlobalMessagesHandlerBotModule.cs file where we can define a module that registers the
FaqScorable class as a component that implements the **IScorable** interface.

So, using **Solution Explorer**, under the root of your Visual Studio project, create a new C#
class file and name it *GlobalMessagesHandlerBotModule.cs.* Let's add the following code to this
file.

Code Listing 4-i: GlobalMessagesHandlerBotModule.cs

```
using Autofac;
using CCBot.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Builder.Scorables;
using Microsoft.Bot.Connector;

namespace CCBot
```

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

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

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

We are simply creating the **FaqScorable** class as a module. The next thing we need to do is to register the module with the conversation's container—this is done in *Global.asax.cs*.

Open *Global.asax.cs*—the **FaqScorable** class can be registered to the conversation's container by registering **GlobalMessageHandlersBotModule** as follows.

Code Listing 4-j: The Updated Global.asax.cs File

```
using System.Web.Http;
using Autofac;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Internals.Fibers;

namespace CCBot
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            RegisterBotModules();

            GlobalConfiguration.Configure(WebApiConfig.Register);
        }

        private void RegisterBotModules()
        {
            Conversation.UpdateContainer(builder =>
            {
```

```
            builder.RegisterModule(new ReflectionSurrogateModule());
            builder.RegisterModule<GlobalMessageHandlersBotModule>();
        });
    }
}
}
```

Let's compile and run this to see what happens—notice that we haven't modified *MessagesController.cs*, which still points to the **RootDialog** class that interfaces with the QnA Maker Service—so that hasn't changed.

In principle, we should be able to query the Creative Commons English knowledge base (as we haven't added the French one yet), and the Scorable should kick in whenever we type in the word **French**; otherwise, we should get the standard results from the QnA service. Let's run it and see.

In order to test it, I'll type in the word **copyright**. I would expect the QnA service to come back with the corresponding FAQ.
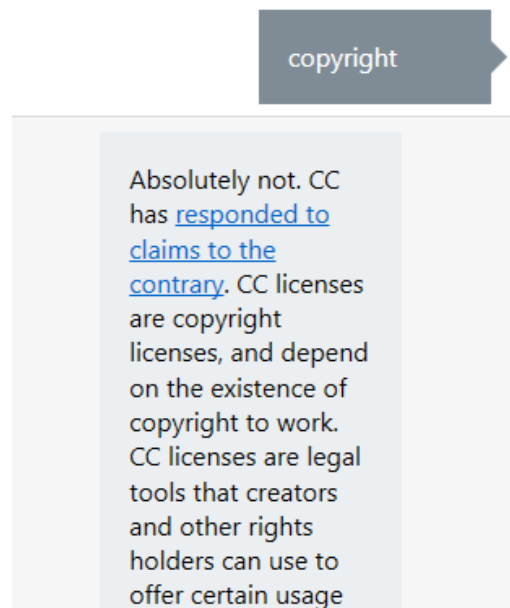
Figure 4-b: The QnA Service Response

Okay, so that worked fine. Let's try now to type in **French** to see if the Scorable kicks in.
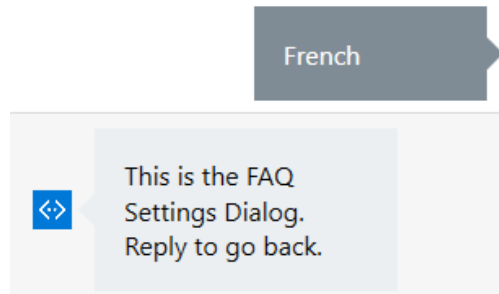
Figure 4-c: The FaqScorable in Action

Awesome—this also worked beautifully. If we now type anything else, like **back**, we should be back to the main dialog and have QnA services responses available again. Let's see if this is true.
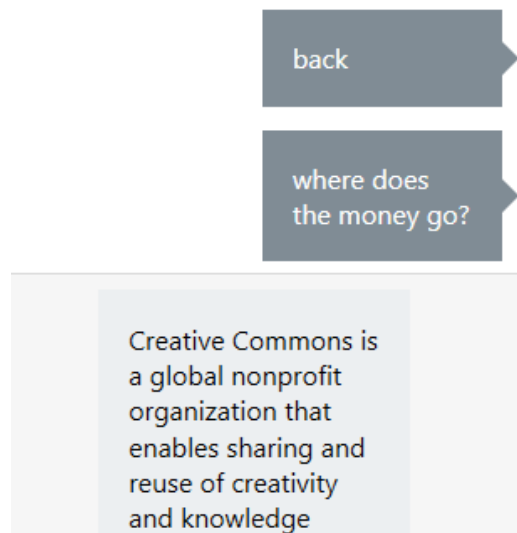

Figure 4-d: The QnA Service Back in Action

This also worked great! We've managed to implement a Scorable within our QnA bot, but we still need to hook up the Creative Commons French FAQs and add the corresponding logic so that our bot can trigger this service. This is what we'll do next.

Implementing another QnA service

Now that we have successfully implemented Scorables in our bot, let's finish the goal we have set for ourselves. We'll need to go back to the code we wrote in the *FaqSettingsDialog.cs* file and make some changes.

Before we make any changes to the code, let's create a new QnA service for the Creative Commons FAQs in French—we can do this by following the same steps we did in the previous chapter, when we created the **CCBot** QnA service.

The only difference is that the URL will be https://creativecommons.org/faq/fr/, and we can name our new service **CCBotFrench**. Once created, don't forget to publish it—this will publicly expose the service for our code.

With this new QnA service published, let's add another class to the *RootDialog.cs* file that will be the entry point to this service.

We can do this by simply cloning the **RootDialog** class, changing its name, and then putting the subscription key and knowledge base ID of the new QnA service. After doing this, our *RootDialog.cs* file should look as follows.

Code Listing 4-k: The Updated RootDialog.cs File

```csharp
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using QnAMakerDialog;

namespace CCBot.Dialogs
{
    [Serializable]
    [QnAMakerService("Eng Subscription ID", "Eng KB ID")]
    public class RootDialog : QnAMakerDialog<object>
    {
        public override async Task NoMatchHandler(IDialogContext context,
        string originalQueryText)
        {
            await context.PostAsync(
                $"No answer for '{originalQueryText}'.");
            context.Wait(MessageReceived);
        }

        [QnAMakerResponseHandler(50)]
        public async Task LowScoreHandler(IDialogContext context, string
            originalQueryText, QnAMakerResult result)
        {
            await context.PostAsync(
                $"Found an answer that could help...{result.Answer}.");
            context.Wait(MessageReceived);
        }
    }

    [Serializable]
    [QnAMakerService("Fr Subscription ID", "Fr KB ID")]
    public class FrenchDialog : QnAMakerDialog<object>
    {
        public override async Task NoMatchHandler(IDialogContext context,
        string originalQueryText)
        {
            await context.PostAsync(
```

```
                $"No answer for '{originalQueryText}'.");
            context.Wait(MessageReceived);
        }

        [QnAMakerResponseHandler(50)]
        public async Task LowScoreHandler(IDialogContext context, string
        originalQueryText, QnAMakerResult result)
        {
            await context.PostAsync(
                $"Found an answer that could help...{result.Answer}.");
            context.Wait(MessageReceived);
        }
    }
}
```

Don't forget to replace the respective subscription and knowledge base identifiers for each service.

As you can see, we literally just copied the **RootDialog** class as is, then renamed it to **FrenchDialog**—which points to the QnA service we just created for the Creative Commons FAQs in French.

It's very important to keep in mind that these classes can only be linked to the correct QnA service if the correct subscription and knowledge base identifiers are used—the **QnAMakerService** attribute decoration is what binds the class to the correct QnA service.

With this done, the only thing remaining is to adapt the **FaqSettingsDialog** class so that the correct QnA service can be triggered.

Code Listing 4-l: The Updated FaqSettingsDialog.cs File

```
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using System;
using System.Threading.Tasks;
using CCBot.Dialogs;
using System.Threading;

public class FaqSettingsDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        await context.PostAsync(
            "French FAQ. Type 'english' to go back.");

        context.Wait(MessageReceived);
    }

    private async Task MessageReceived(IDialogContext context,
```

```
    IAwaitable<IMessageActivity> result)
    {
        var message = await result;

        if ((message.Text != null) && (message.Text.Trim().Length > 0))
        {
            if (message.Text.ToLower().Contains("english"))
                context.Done<object>(null);
            else
            {
                var fDialog = new FrenchDialog();
                await context.Forward(
                    fDialog, null, message, CancellationToken.None);
            }
        }
        else
        {
            context.Fail(new Exception(
                "Message was not a string or was an empty string."));
        }
    }
}
```

So, to recap with the modifications we've just made, once the Scorable intercepts the word **French**, it triggers the execution of **FaqSettingsDialog**—which will forward the conversation to **FrenchDialog** if any keyword is provided other than the word **English**—which returns the execution back to **RootDialog**.

Running with two QnA services

We've reached an important milestone—we've managed to implement a bot with a Scorable and two QnA services, so now is the moment of truth. Let's compile and run it, and see what happens.
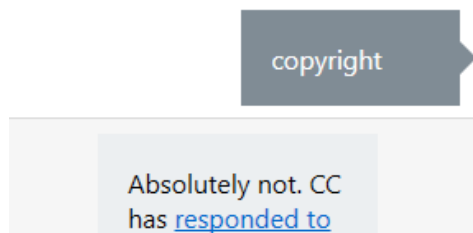
Figure 4-e: The English QnA Service Is Triggered

Here we can see that when we type **copyright**, the correct English FAQ response is returned. Let's see what happens when we type **French**.
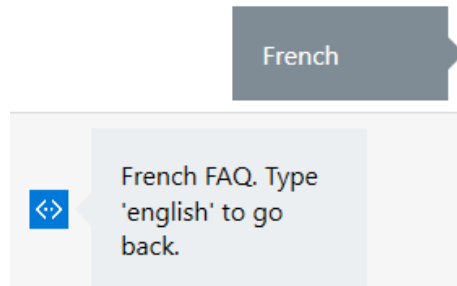
Figure 4-f: The Scorable Is Triggered

We can see that the Scorable has been triggered. Before we test out the QnA service for French FAQs, let's type in the word **English** in order to see if we are able to go back to the QnA service for English FAQs.
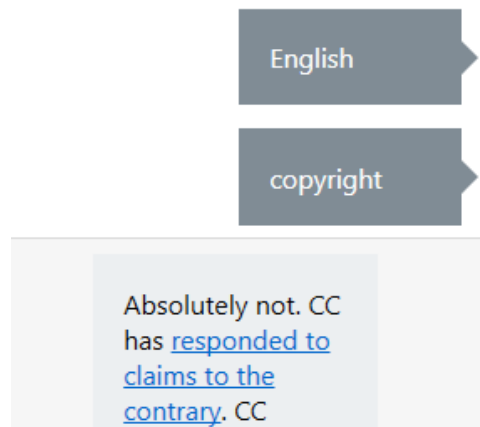

Figure 4-g: The English QnA Service Is Triggered Again

Excellent—everything is working as expected. Now, let's type in **French** again to have the Scorable kick in again.
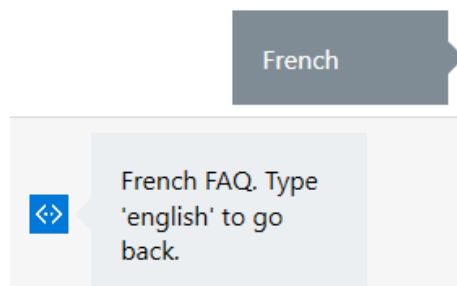

Figure 4-h: The Scorable Is Triggered Again

The Scorable was correctly triggered, as expected. Now let's type in **le logo** to see if we get an FAQ result from the French QnA service.
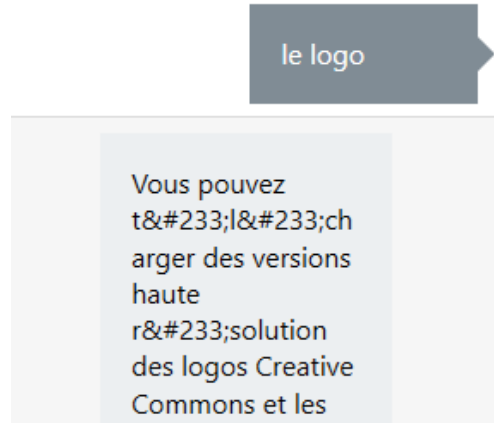
Figure 4-i: The French QnA Service Is Triggered

How wonderful is this!? Our second QnA service that points to the French FAQs was correctly triggered.

But wait, there's one thing we missed—we can't go back to the English QnA service unless we restart the emulator, which would be the equivalent of starting a new conversation.

I actually did this on purpose and left it out to give you a small challenge. As you can also see, there's also a bit of markup removal that needs to be done—so there you go, a nice challenge for you to take forward.

Summary

We've come a long way since we started with our first QnA bot example. The goal of this chapter was to demonstrate the power and possibilities of using Scorables, as a way to intercept user requests and redirect them to other dialogs, depending on your business requirements.

Although the example outlined here is rather simple in terms of the business logic behind it, to be able to execute two QnA services instead of simply one, for now it is enough to demonstrate how Scorables can add an extra processing layer to our bot. This gives us the ability to do more things than we could do by sticking with the normal stack execution of dialogs.

In the next chapter, we'll wrap up by publishing our latest example to Skype and interacting with it there instead of using the emulator.

The full source code for this chapter can be found here.

Chapter 5 Publishing

Quick intro

We've now explored the most interesting features of Skype Bots programming, such as LUIS, the QnA Maker Service, and Scorables—really cool APIs and development paradigms.

All we are missing is to publish our latest example to Skype and test it there—this is what we'll do now. Ready, set, go!

Publishing to Skype

Registering and publishing a bot to Skype is very easy. First, log in to the Bot Framework Developer Portal with your Microsoft account, and click **My bots**.
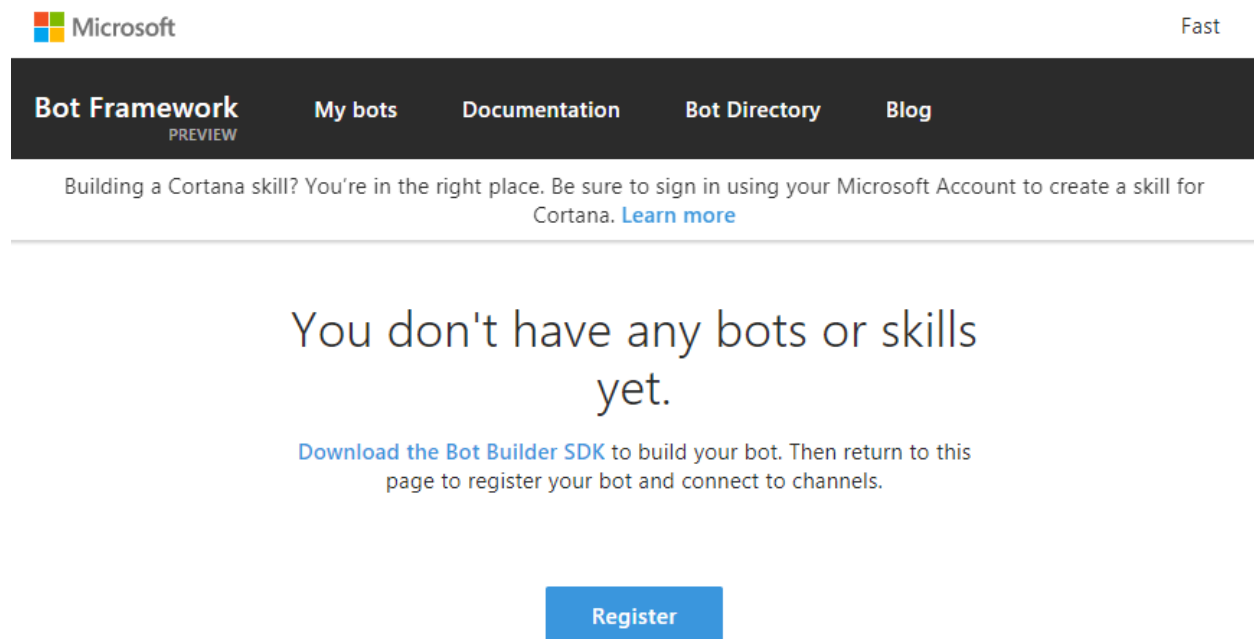
Figure 5-a: The Bot Framework Developer Portal

Once logged in, click **Register** in order to start the process of registering our bot with Skype. The process is pretty straightforward and intuitive (this is also explained in-depth in my other e-book—*Microsoft Bot Framework Succinctly*).

As we can see in Figure 5-b, we are required to enter several fields, but the most important one is the **Messaging endpoint**—which is the public URL where our bot will reside.

Configuration

Messaging endpoint

```
https URL
```

Register your bot with Microsoft to generate a new App ID and password

Create Microsoft App ID and password

* Paste your app ID below to continue

```
Microsoft App ID from the Microsoft App registration portal
```

Figure 5-b: Fields Required for Publishing our Bot to Skype

In order to get a public URL for our bot, we need to open **Solution Explorer** in Visual Studio, right-click on the project name, and then click **Publish**. We'll then be presented with a wizard that is very intuitive to follow. We'll be asked for some information in order for the wizard to be able to create the Azure App Service that will host the bot.

We won't go over the whole process here, you can see in Figure 5-c how the publishing wizard's main screen looks after you have chosen the **Publish** option.
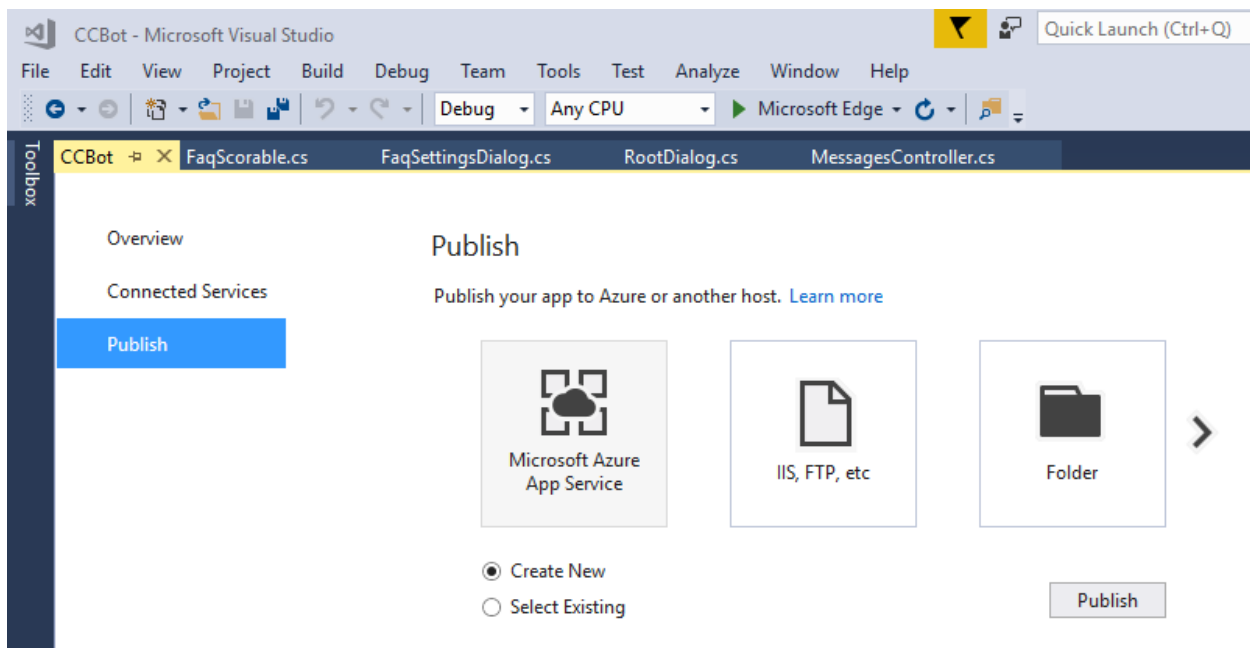
Figure 5-c: The Publishing Wizard in Visual Studio

In Figure 5-d, we can also see the publishing process taking place.

Figure 5-d: The Publishing Process

Once the publishing process is complete, we can get the bot's public URL, as shown in Figure 5-e.

Figure 5-e: Published Succeed

It is very important that when specifying the messaging endpoint, we add **/api/messages**, and make sure that the **https** is used.

Figure 5-f: The Bot's Messaging Endpoint

Once we've filled in all the data required on the Bot Framework Developer Portal, click **Register** on the bottom of the form to complete the process.

By default, the Bot Framework automatically deploys the bot to the Skype and Web Chat channels. To test the bot in Skype, all we need to do is to click on the **Skype** icon on the Bot Framework Developer Portal page, as shown in Figure 5-g.
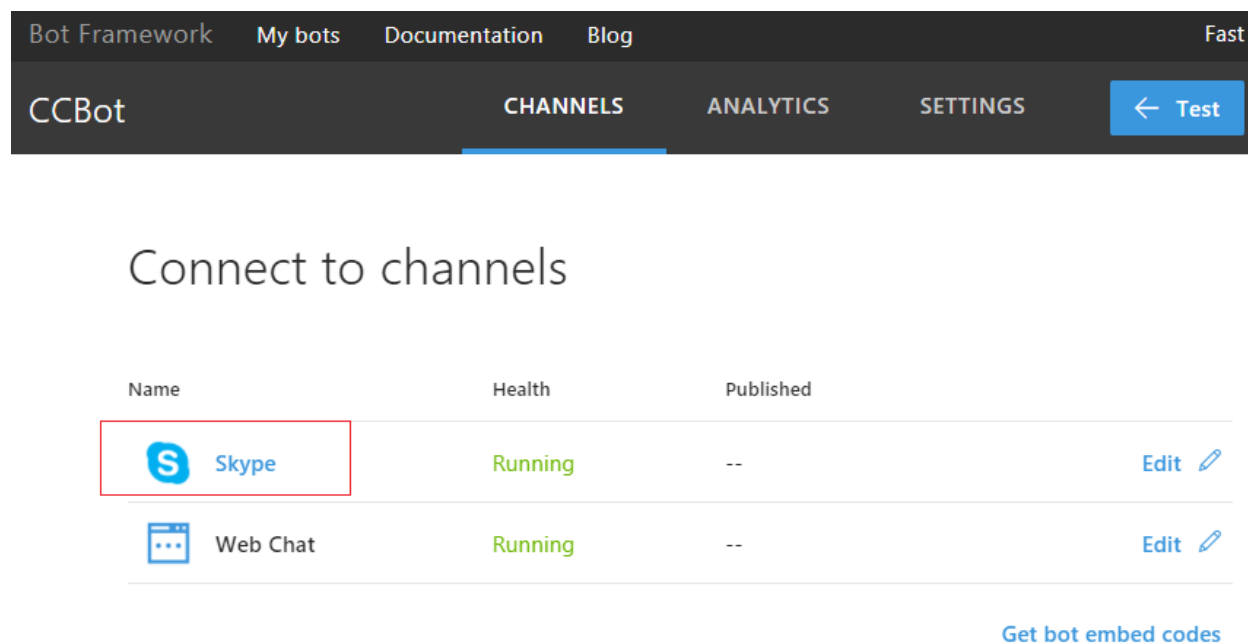

Figure 5-g: The Try on Skype Button on the Bot Developer Portal

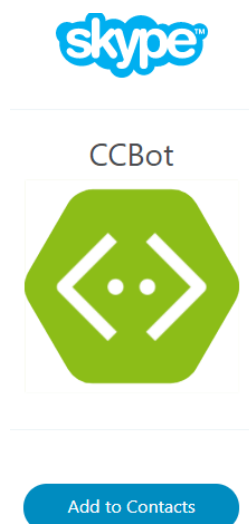Once you click on the **Skype** icon, you'll be asked to add the bot to your contacts.

Figure 5-h: The "Add to Contacts" Option

Next, you'll be presented a browser pop-up window asking you to **Open Skype**.

Finally, the bot will be added to your Skype contacts list. You'll be asked to confirm within the Skype application that you would like the bot added to your contacts list. That's it—let's now do a quick test.

Testing on Skype

Now that we've added the bot to Skype, we can run a quick test to see if it works as expected. As we've already tested it extensively on the emulator and have not changed the code, things should work in the same way as on the emulator.

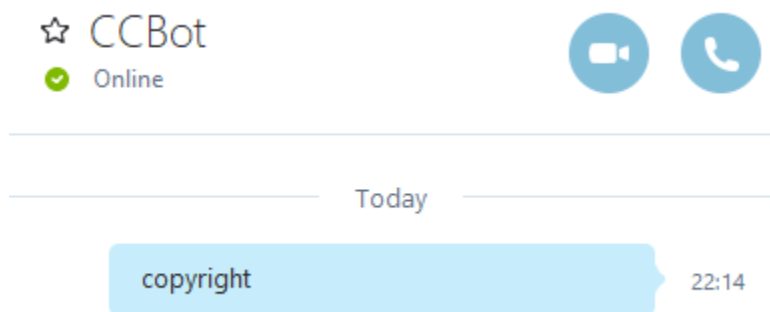So, let's simply enter some keywords and see what the bot responds to.

Figure 5-i: Testing Our Bot on Skype

Looks cool! But wait, it seems that the bot is not giving back any answers. The reason is that we forgot to add the bot's App ID and Password from the Bot Framework Developer Portal to the *Web.config* file within our Visual Studio project.

Code Listing 5-a: The AppSettings Part of the Web.config File

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

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

The **BotId** is the bot handle, the **MicrosoftAppId** is the App ID, and the **MicrosoftAppPassword** is the auto-generated password.

We can get these values from the Developer Portal by clicking on **Settings** and then **Manage Microsoft App ID and password**.

Figure 5-j: The Bot's App ID and Password

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

Once you've entered the values, save the changes in Visual Studio and publish the bot again. Then, go to Skype and test the bot again. Type in **copyright**.
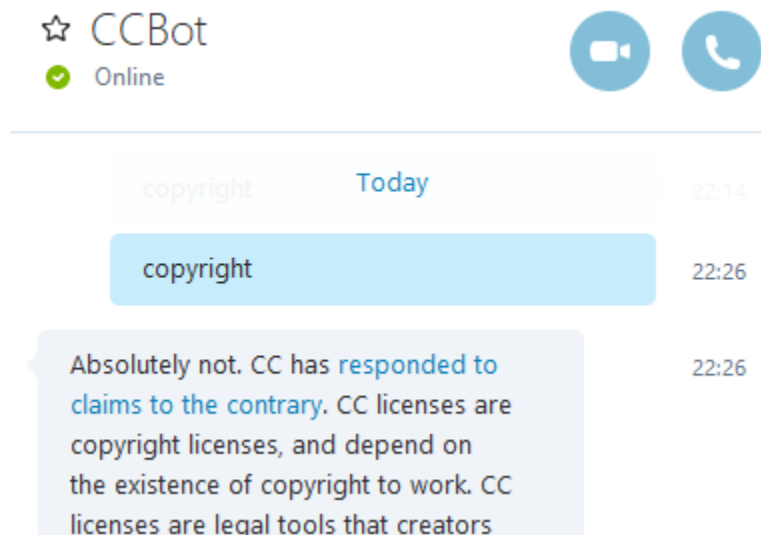

Figure 5-k: Testing Our Bot on Skype Again

Now we're in business! Feel free to try it further, but bear in mind that the results will be the same as when we tested using the emulator.

We've come to the end of our journey—but before wrapping up, I'd like to close with some comments.

Closing comments

It's been a very interesting journey of exploration—we've expanded upon the concepts learned in the *Microsoft Bot Framework Succinctly* book and looked at APIs and technologies that take bots to the next level.

Skype is Microsoft's premier communication tool, and one of the most popular communication platforms in the world.

Bots are already starting to add value to businesses and improve the experience for both users and companies. As the ecosystem continues to evolve, there's a lot of potential for new APIs to further enrich this platform.

There are still plenty of Skype APIs that we didn't cover in this book, and I invite you to explore some of these newest APIs—still in preview mode—such as Payments and Add-Ins, which could make the conversational experience even more enjoyable for users.

This book has laid out a solid foundation for you to keep exploring all the amazing possibilities that this vast ecosystem has to offer.

I hope you have enjoyed this book, and that it has helped you delve further into Skype development—whether for fun or business purposes.

Thanks for reading and supporting the *Succinctly* series. Be awesome!