|  |  |
| --- | --- |
|  |  |
|  | Cortana Intelligence Suite  Lab CIS011  Support Bot  using Bot Framework |

Contents

[Overview 3](#_Toc469557013)

[Creating your First Bot 5](#_Toc469557014)

[Using a Dialog 9](#_Toc469557015)

[Guided Conversation with FormFlow 13](#_Toc469557016)

[Managing Conversation State 17](#_Toc469557017)

[Building the Final Bot 25](#_Toc469557018)

[Deploying the Bot 37](#_Toc469557019)

[Deleting the Bot 47](#_Toc469557020)

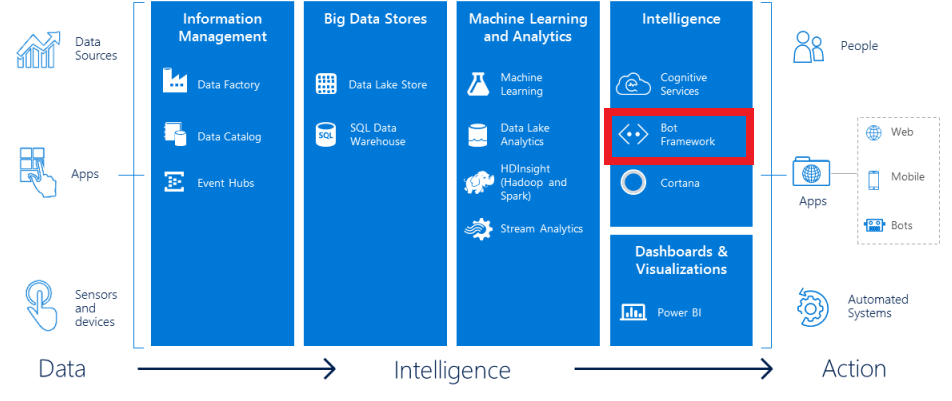
[Appendix A - Azure Bot Service 49](#_Toc469557021)

[Terms of Use 52](#_Toc469557022)

|  |
| --- |
| Overview |

### Summary

This lab introduces the Microsoft Bot Framework. The Bot Framework allows creation of bots that can communicate with people using natural language. It can use a variety of channels from text messages, Skype, Slack, and more.



In this lab, you will create a bot to interact with users. Several bot features will be used including dialogs, form flows, prompt dialogs, and bot state.

### Business Case

Providing support for internal or external users is key to maintaining their satisfaction. Ideally, this support is provided in a timely manner and without using more resources than required.

In this lab, you will create a support bot for a fictitious software application. This bot will be able to gather user information and assist users with some common issues.

### Learning Objectives

Upon completing this lab, you will have hands-on experience with the following functions and concepts related to Microsoft’s Bot Framework:

* Creating a Bot using the Visual Studio template
* Testing a bot using the Bot Framework Channel Emulator
* Using form flow to create a bot form
* Creating a bot dialog
* Interacting with a user through the dialog
* Deploying the bot to Microsoft Azure

### Lab Requirements/Prerequisites

* An Azure subscription is required to deploy the bot once completed. This is not strictly necessary as you can build and test the bot with the emulator and choose not to deploy it if so desired. However, any bots you would like people to use will need to be deployed to Azure.

If you do not have an Azure subscription, you can create one with a free credit at the following link:

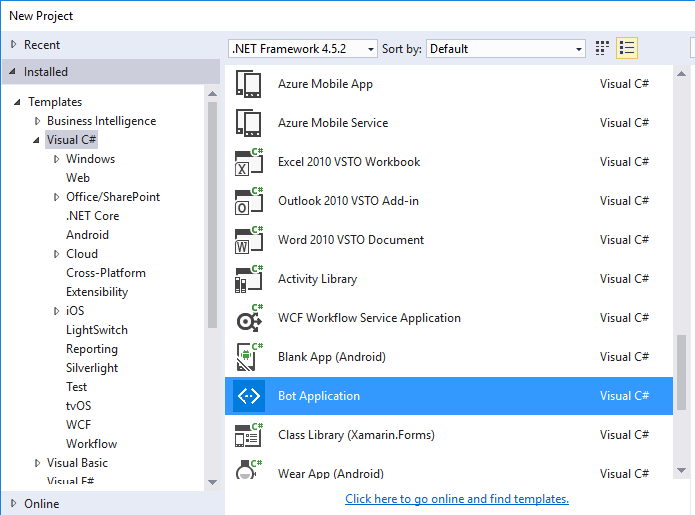
<https://azure.microsoft.com/en-us/free/>

|  |
| --- |
| Creating your First Bot |

### Creating a bot from the template

To begin, we will create a simple bot from the Bot Application template.

1. Open **Visual Studio** and choose the **New** -> **Project** from the **File** menu.
2. Choose the Bot Application template from the **Visual C#** template list.



If you have not yet installed Visual Studio 2015 or the Bot Application template you can obtain them from the following links:

**Visual Studio 2015 Community**

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs>

**Bot Application Template**

<http://aka.ms/bf-bc-vstemplate>

installs to your Visual Studio template folder which is at the following location by default:

%USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C#\

1. Name the project whatever you would like. The code samples are based on a project named SupportBot so give it that name if you would like to be able to copy and paste code directly. Otherwise, you will have to change the namespace whenever copying and pasting code.
2. Select the location you would like for the project and click **OK** to create it.

### Exploring the bot

Now, let’s look at the code for the bot that was created.

1. Expand the **Controllers** folder and open the **MessagesConroller.cs** file.
2. Check out the Post method. It should look like this:

public async Task<HttpResponseMessage> Post([FromBody]Activity activity)

{

if (activity.Type == ActivityTypes.Message)

{

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

// calculate something for us to return

int length = (activity.Text ?? string.Empty).Length;

// return our reply to the user

Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");

await connector.Conversations.ReplyToActivityAsync(reply);

}

else

{

HandleSystemMessage(activity);

}

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

return response;

}

1. This is a simple bot that returns the text back to the user that the user sent and the character length of that text. It performs the following steps:
   1. It receives an Activity object and checks if that type is a Message. If it is not a message, it sends it to the HandleSystemMessage to process it.
   2. If it is a message, it gets handled in the if statement within the Post method. It first gets the length of the message using the Length string attribute.

Then, it uses the CreateReply function to create a reply message back to the user consisting of the original message text and its length.

Finally, it sends this message back to the user with the ReplyToActivityAsync function.

If you have not yet installed the Bot Framework Channel Emulator, you can obtain it from the following link:

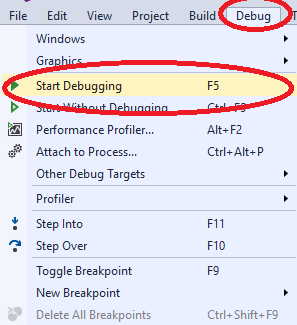
**Bot Framework Channel Emulator**

<https://download.botframework.com/bf-v3/tools/emulator/publish.htm>

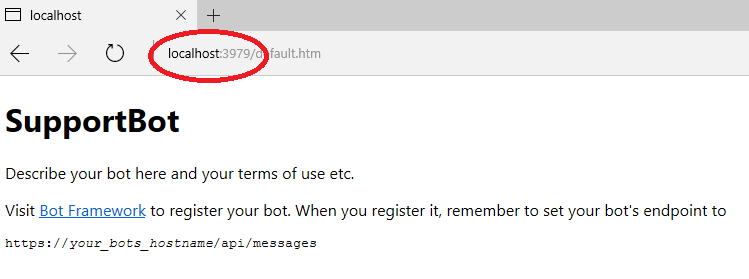
### Test with the Bot Framework Channel Emulator

To begin using the Bot Framework Channel Emulator, we first must verify the project runs and get the port it’s running as.

1. From the **Debug** menu in Visual Studio, select the **Start Debugging** option. (Or, alternatively, just hit **F5**.)

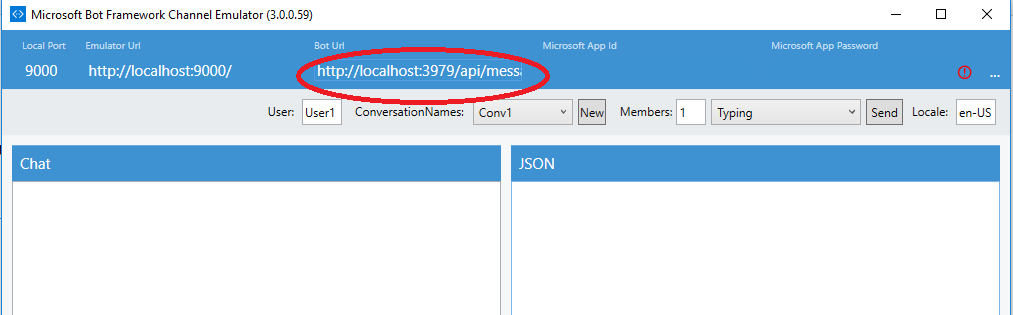


1. Your web browser of choice should execute to the default bot page. Take note of the port it is running on.

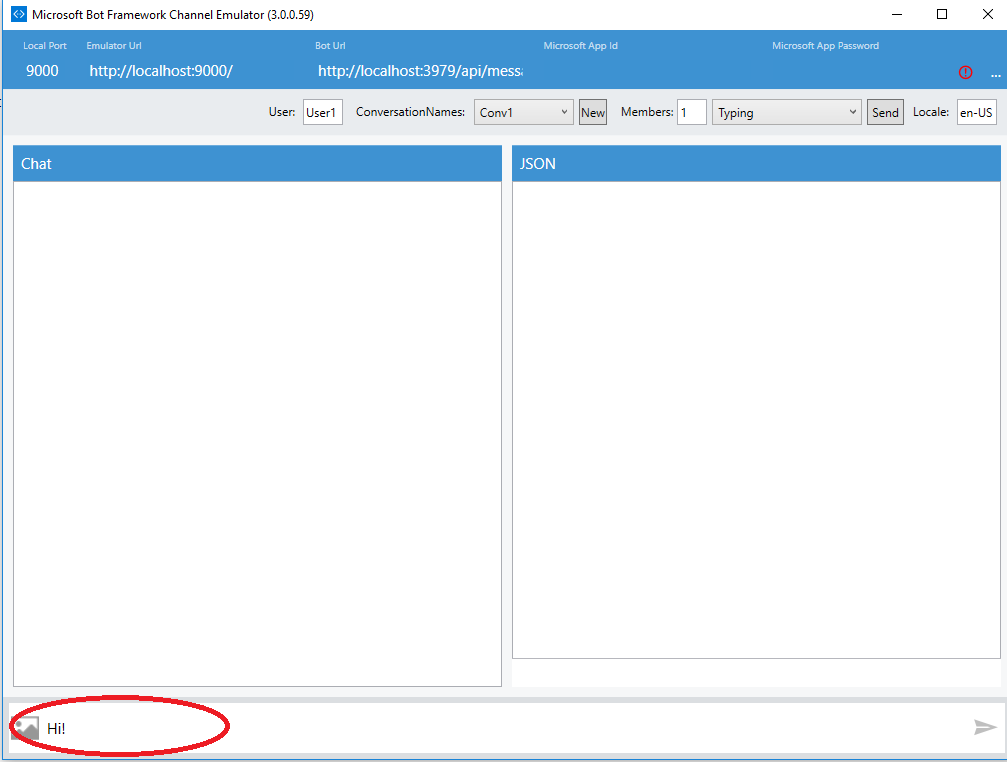


In this case, it’s using port 3979.

1. Now, if it’s not already running, execute the Bot Framework Channel Emulator. It should be in your Start menu after having been installed.
2. Set the port in the emulator’s **Bot Url** box to match the port the new bot is running as.



1. Type in a message to send to the bot. It will respond telling you the text you sent and how many characters long it was.



1. When finished executing, you can stop the Visual Studio project from debugging so that further changes can be made to the code. You can do this via the **Debug** menu and then selecting **Stop Debugging** or by hitting **Shift-F5**.

|  |
| --- |
| Using a Dialog |

The default template builds the bot as part of the controller. However, using dialogs allows us to easily manage different conversational processes.

As an example, we would have one root dialog that can divert to other dialogs depending upon what the user wants to do. The user may want to change their stored information, place orders, contact support, etc. Each of these could then be a separate dialog.

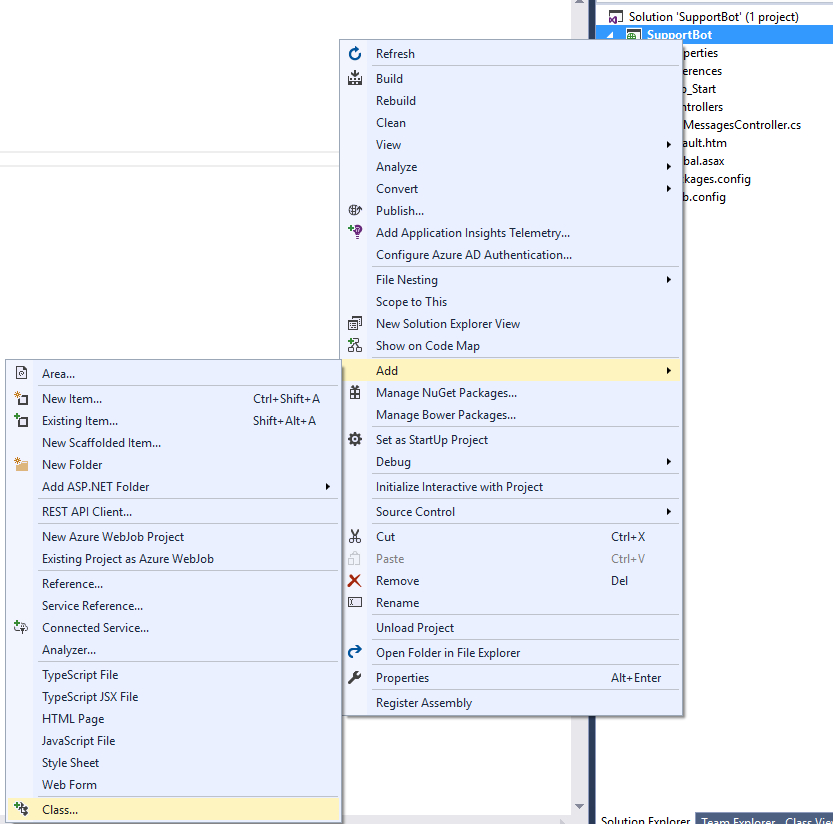
For more information on dialogs please see the documentation at:

<https://docs.botframework.com/en-us/csharp/builder/sdkreference/dialogs.html>

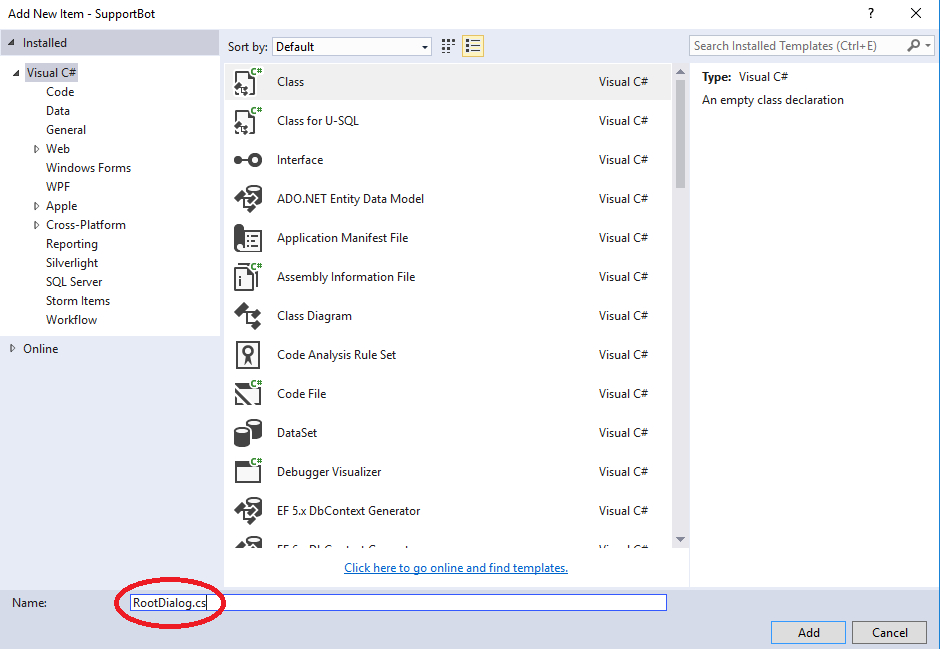
We will also be expanding on dialogs as the lab progresses.

In this example, we will modify the bot to work the same way as the template created bot, but by using a dialog instead.

1. Right click on the project name in Visual Studio and select **Add** and then **Class**. Note that we’re adding new classes to the root of the project, but in a real example you should put each class logically in folders (for instance, a Dialogs folder for dialogs).



1. Name the new class file **RootDialog.cs** and click **Add**.



1. Visual Studio will create a new class. Copy and paste the following code into it, replacing what is already there:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

IMessageActivity result = await argument;

await context.PostAsync($"You sent {result.Text} which was {result.Text.Length} characters");

context.Wait(MessageReceivedAsync);

}

}

}

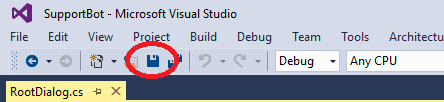
A dialog requires two methods: **StartAsync** and **MessageReceivedAsync**.

The **StartAsync** method is called when a conversation begins. Here, we are greeting the user with a message that says, “**Welcome to the support bot!**”. This will only display with the first message of the conversation.

The **MessageReceivedAsync** method is called every time a message is received from the user. In this case, we are sending back the same text as we had originally sent, but are now retrieving the text value from the argument this time.

Once we return the message, we call **context.Wait(MessageReceivedAsync)**. This tells the bot to wait for another response from the user and gives it the method to use upon receiving that response.

1. Save the changes to the RootDialog.cs file by clicking the disk icon. Alternatively, you could select Save RootDialog.cs from the File menu.



1. Now, we need to modify the MessagesController class to use the new dialog. To start, add the highlighted using command below to the top of the MessagesController.cs file:

using System;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Threading.Tasks;

using System.Web.Http;

using System.Web.Http.Description;

using Microsoft.Bot.Connector;

using Newtonsoft.Json;

using Microsoft.Bot.Builder.Dialogs;

1. Then, replace the Post method with the following highlighted code:

using System;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Threading.Tasks;

using System.Web.Http;

using System.Web.Http.Description;

using Microsoft.Bot.Connector;

using Newtonsoft.Json;

using Microsoft.Bot.Builder.Dialogs;

namespace SupportBot

{

[BotAuthentication]

public class MessagesController : ApiController

{

/// <summary>

/// POST: api/Messages

/// Receive a message from a user and reply to it

/// </summary>

public async Task<HttpResponseMessage> Post([FromBody]Activity activity)

{

if (activity.Type == ActivityTypes.Message)

{

await Conversation.SendAsync(activity, () => new RootDialog());

}

else

{

HandleSystemMessage(activity);

}

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

return response;

}

This simply tells the Bot Framework to create a new **RootDialog** and use it for the conversation.

1. Execute the Visual Studio project as before (via Start Debugging or F5) and test it out with the Bot Framework Channel Emulator. You should see that it functions in the same way as before, except now it greets you with the first message now.
2. When finished testing, stop the Visual Studio project from debugging via the Stop Debugging command or Shift-F5.

|  |
| --- |
| Guided Conversation with FormFlow |

A FormFlow allows us to define a set of fields which the Bot Framework then handles building a dialog for us to collect data for each field. You can view the documentation (and a very good sandwich builder example) at the below link:

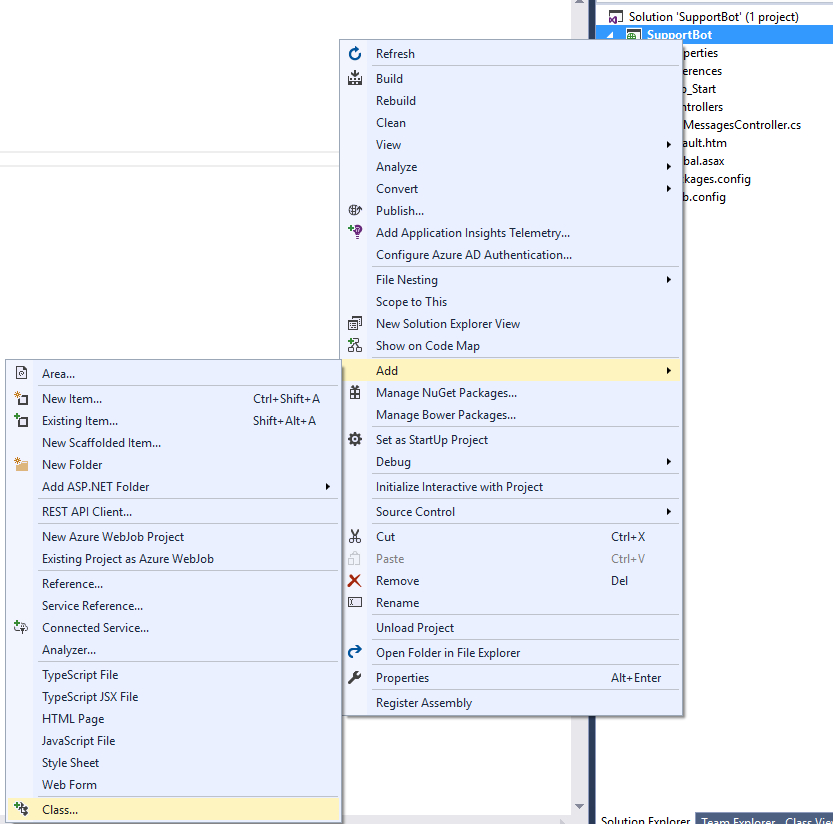
<https://docs.botframework.com/en-us/csharp/builder/sdkreference/forms.html>

For our example, we will use a FormFlow to collect information regarding the user which will be requesting support from our bot. We would like to collect the following pieces of information:

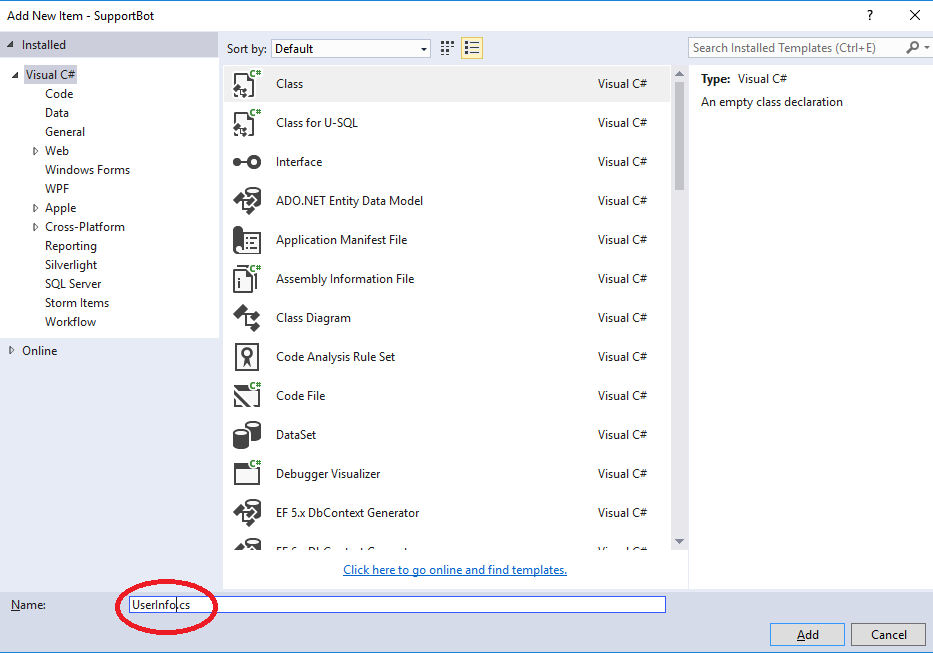
* Name
* Phone Number
* Email Address
* Department

To use a FormFlow, we first need to create a class with the fields defined we would like to collect.

1. Right click on the project name in Visual Studio and select **Add** and then **Class**.



1. Name the new class file **UserInfo.cs** and then click **Add**.



1. Replace *all* the contents of the newly created UserInfo.cs file with the following code:

using System;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

namespace SupportBot

{

public enum DepartmentOptions

{

Accounting,

AdministrativeSupport,

IT

}

[Serializable]

public class UserInfo

{

[Prompt("Please enter your {&}.")]

public string Name;

[Prompt("Please enter your {&}.")]

public string PhoneNumber;

[Prompt("Please enter your {&}.")]

[Pattern(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}")]

public string EmailAddress;

[Prompt("What {&} is this request? {||}")]

public DepartmentOptions? Department;

public static IForm<UserInfo> BuildForm()

{

return new FormBuilder<UserInfo>().Build();

}

}

}

This code is defining a **UserInfo** class that contains the data elements that the Bot Framework will collect. It asks for the fields in the order that they are defined in the class by default so it will start with **Name**, then **Phone Number,** then **Email Address**, and finally **Department**.

C# attributes can be used to provide guidance to the FormFlow regarding how to gather the data. We use two attributes here:

* Prompt: The Prompt attribute tells the FormFlow what text to use to request the data from the user. The {&} pattern in the Prompt string tells it to fill in the name of the field in that location. So, where it says, “Please enter your {&}.” for the Name field it will ask the user to “Please enter your name.”.
* Pattern: The Pattern attribute tells the FormFlow to use a regular expression to validate the data entered. In this case, we are using a regular expression to validate that the value entered for EmailAddress is truly a valid email address.

Enums can be defined to provide a multiple-choice list of options. In this example, we have a **DepartmentOptions** enum defined which contains three possible values:

* Accounting
* Administrative Support
* IT

This enum is used for the **Department** field by declaring its data type to be **DepartmentOptions**.

The final piece for this class is to add a method (in this case called **BuildForm**) that uses the **FormBuilder** class to return a form built from the class as it was defined above.

1. The next step is to modify the RootDialog class to use a FormFlow based on our UserInfo class instead of returning the user’s text and length of that text. To do this, replace *all* the contents of the RootDialog.cs file with the following:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

context.Wait(MessageReceivedAsync);

}

}

}

This code has several changes although the **StartAsync** method stayed the same.

* **MessageReceivedAsync** method: This method uses **context.Call** to call the **BuildForm** method created in the prior step. The last parameter of the **context.Call** method tells it what to call after the current call (in this case, the **UserInfo** form) finishes: the **AfterUserInfoForm** method.
* **AfterUserInfoForm** method: This method is called after the form has been completed. It is currently set to just wait for another conversation to begin.

1. The final step is to execute the Visual Studio project as before (via **Start Debugging** or **F5**) and the Bot Framework Channel Emulator to test it out. Send a message (such as **Hi**) to get it started and follow its prompts through. Try entering an invalid email address to see how it handles validation. You can also try entering a department that’s not in the list and see what happens. Finally, trying saying “**No**” when it asks “**Is this your final selection?**”.

|  |
| --- |
| Managing Conversation State |

You can use **context.ConversationData**, **context.PrivateConversationData**, and **context.UserData** to store state. These are used in the following ways:

* **ConversationData**: This is data specific to the conversation
* **UserData**: This is data specific to the user
* **PrivateConversationData**: This is data specific to the user in the conversation

For the purposes of this lab, we’ll use **ConversationData** to keep things simple.

Our bot is already collecting user data. Now, let’s store this data as part of the conversation upon retrieval. In addition, let’s expand on our requirements to get a response method (either email or phone) and a text description of the issue the user is experiencing.

1. Currently, our **AfterUserInfoForm** method in the **RootDialog.cs** file is waiting for the next task. Let’s modify it to store data and move onto the next step (getting the issue). Modify the **AfterUserInfoForm** method to look like the following highlighted text:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForIssue(context);

}

}

}

The first parameter is the **IDialogContext** representing our conversation context and the second parameter is the **UserInfo** which is passed from the previous step in the conversation (the FormFlow).

We then add several items to the **ConversationData** using **SetValue**: Name, PhoneNumber, EmailAddress, and Department. We populate these with the corresponding values from the **UserInfo** object which was passed into the **AfterUserInfoForm** method.

Finally, we call the **PromptForIssue** method and pass the **context** of the conversation. However, this method does not exist yet, so we will create it now.

1. Add the highlighted **PromptForIssue** method after the **AfterUserInfoForm** method using the following code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForIssue(context);

}

private void PromptForIssue(IDialogContext context)

{

PromptDialog.Text(context, AfterGetIssue, "Please explain in detail the issue you are having.");

}

}

}

This code is very simple. It uses **PromptDialog** to generate a prompt to retrieve a text value from the user. We are using three parameters here for the **Text** method of **PromptDialog**:

* The **context** of the conversation
* The method to call after it has finished prompting
* The text of the prompt itself

We could have put this one line of code in the **AfterUserInfoForm** in place of the call to the **PromptForIssue** method call and everything would still work in the same way.

So, why use a separate function call here? This way, as we proceed further into the lab we can prompt for the issue from multiple points in the conversation without duplicating the code. In a full project, we may use localization features instead of a method such as this to accomplish a similar thing.

The second parameter of the **PromptDialog.Text** method is attempting to call an **AfterGetIssue** method. However, that method does not yet exist. Let’s create it now.

1. Add an **AfterGetIssue** method using the following code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForIssue(context);

}

private void PromptForIssue(IDialogContext context)

{

PromptDialog.Text(context, AfterGetIssue, "Please explain in detail the issue you are having.");

}

private async Task AfterGetIssue(IDialogContext context, IAwaitable<string> result)

{

string issue = await result;

context.ConversationData.SetValue("Issue", $"{issue}");

PromptForContactMethod(context);

}

}

}

The first parameter of the **AfterGetIssue** method is again an **IDialogContext** but this time the second parameter is a **string**. This is because using **PromptDialog.Text** returns a string from the user. We then take this string and add it to the **ConversationData** as an item called **Issue**.

Once again, we are calling another method which does not yet exist. This time, the **PromptForContactMethod**. Let’s go ahead and create it, as well as its **After** method as well now by adding the following highlighted code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForIssue(context);

}

private void PromptForIssue(IDialogContext context)

{

PromptDialog.Text(context, AfterGetIssue, "Please explain in detail the issue you are having.");

}

private async Task AfterGetIssue(IDialogContext context, IAwaitable<string> result)

{

string issue = await result;

context.ConversationData.SetValue("Issue", $"{issue}");

PromptForContactMethod(context);

}

private void PromptForContactMethod(IDialogContext context)

{

PromptDialog.Choice(context, AfterGetContactMethod, new List<string>() { "Email", "Phone" }, $"How would you prefer for support to contact you?");

}

private async Task AfterGetContactMethod(IDialogContext context, IAwaitable<string> result)

{

string contactMethod = await result;

string name;

string email;

string phoneNumber;

string department;

string issue;

context.ConversationData.TryGetValue("Name", out name);

context.ConversationData.TryGetValue("EmailAddress", out email);

context.ConversationData.TryGetValue("PhoneNumber", out phoneNumber);

context.ConversationData.TryGetValue("Department", out department);

context.ConversationData.TryGetValue("Issue", out issue);

if (contactMethod.ToLower() == "phone")

{

//Have support contact {name} from the {department} department at {phoneNumber} regarding {issue} via phone

}

else

{

//Have support contact {name} from the {department} department at {email} regarding {issue} via email

}

await context.PostAsync($"You will be contacted via {contactMethod} within the next hour. Thank you for using the support bot.");

context.Wait(MessageReceivedAsync);

}

}

}

The **PromptForContactMethod** is once again using **PromptDialog**. However, this time it’s using **Choice** instead of **Text**. This presents a list of choices for the user to choose from. In this case, the possible values are **Email** and **Phone**.

The **AfterGetContactMethod** is then called after getting the preferred contact method. Here, we again await a result which is a **string**. This time it is the result of the **PromptForContactMethod**’s **PromptDialog.Choice** call.

Once we have this result, we get the information out of the conversation that we will need: Name, Email Address, Phone Number, Department, and the Issue itself. This is accomplished by using the **ConversationData.TryGetValue** method. The first parameter is the data element to receive and the second is an output variable in which to store the value.

We then check the contact method and handle it appropriately. Note that this portion is not implemented above as it would be specific to different locations how their support ticketing process was handled. The above code is just to give you an example of how to go about handling it.

The **AfterGetContactMethod** method ends with a confirmation to the user that they will be contacted via their selected contact method within the next hour. It then finishes by calling **context.Wait** to wait for any further messages on the conversation. If any further messages came in, they would start at the **MessageReceivedAsync** method.

And, speaking of the **MessageReceivedAsync** method, let’s modify it so that it has a little intelligence built in. If this is a new conversation, we want to collect user information. However, if it is a continuation of an existing conversation we do not need to go through those extra steps again. We can skip to the next part.

1. To do this, modify the **MessageReceivedAsync** method with the following highlighted code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForIssue(context);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

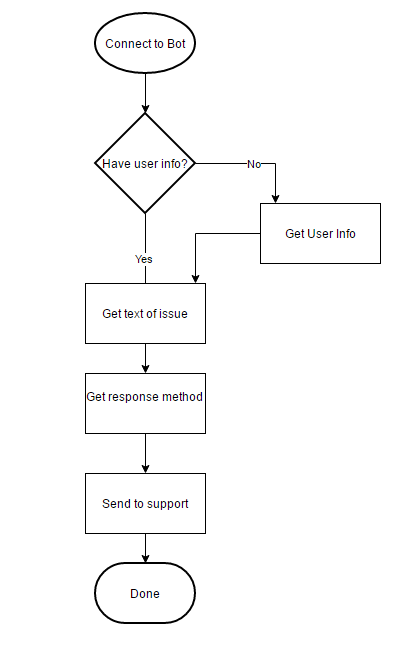
PromptForIssue(context);

}

*…the rest of the code*

Here, we check if there is a name in the **ConversationData** already stored. Now you see that the **TryGetValue** method returns a Boolean value: true if the value was retrieved and false if not. This allows to immediately call the **PromptForIssue** method if we already have the name. If not, then we build the form for the **UserInfo** as before.

As the dialogs start to get more complicated, it can sometimes help to build a flow chart of the process. Here is what our conversation looks like so far:



1. Execute the Visual Studio project as before (via **Start Debugging** or **F5**) and the Bot Framework Channel Emulator again and test out the bot for yourself. After following through to the Done step it will start at the beginning again after sending another message (such as **Hi**). This time, you should see that we have user info so it will skip the Get User Info step above and go right to the issue.

Note that if you do not restart the Bot Framework Channel Emulator it will continue the conversation where it left off.

|  |
| --- |
| Building the Final Bot |

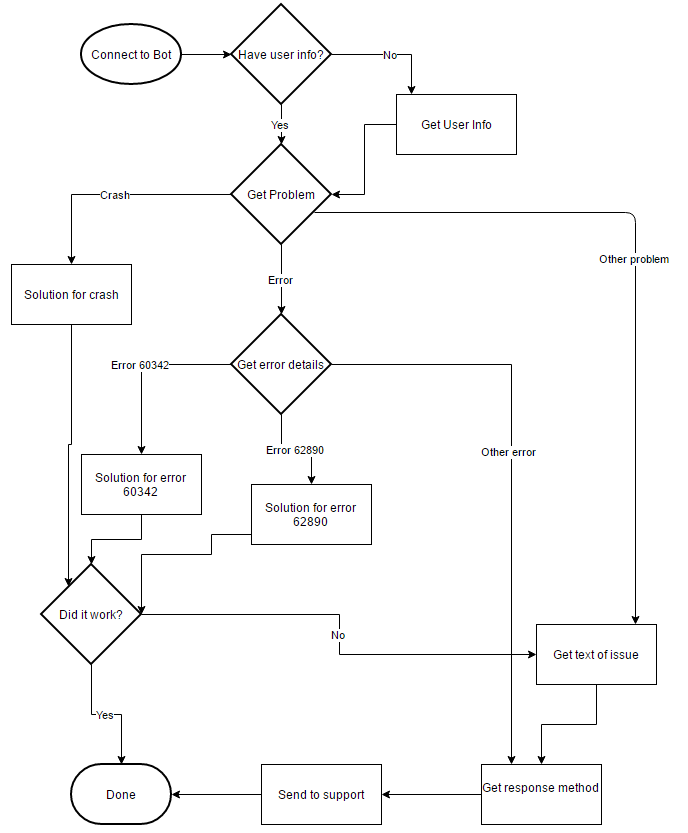
We have a nice bot that collects user information and forwards it on to support. However, we can expand on that. What if we added some common issues that users have and the solutions for them to try and help the user fix issues just using the bot?

Let’s imagine we have a few issues we know about:

* Crashes: These were fixed with the latest version of our software but not everyone has upgraded yet.
* Error codes: for simplicity, in this lab we will just check for two error codes with simple, made up solutions. One code will mean bad data that will require running a repair tool. The other code requires deleting temporary files.
* Other: we do not know how to handle these so they will go right to support

In addition, we should check if each of the presented solutions worked and, if not, transfer to support.

Our final conversation flow chart, then, will look like this:



To implement this logic in full:

1. Modify the **AfterUserInfoForm** function in the **RootDialog.cs** file to the following highlighted text:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForIssue(context);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForProblem(context, userInfo.Name);

}

*…the rest of the code*

The first change here is that instead of calling **PromptForIssue** we will instead call **PromptForProblem**. This method does not exist yet, but it will. Note that it accepts a second parameter of the user’s name. This is to greet the user by name once their **UserInfo** is setup.

In addition, we need to change the **MessageReceivedAsync** method to call **PromptForProblem** instead of **PromptForIssue** via the highlighted text below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForProblem(context, name);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

Here, we are using the output variable **name** from the **ConversationData** and sending it to the **PromptForProblem**. This is another example of retrieving state data for the conversation.

1. Next, we need to create the **PromptForProblem** method and its **After** method. The location in the **RootDialog.cs** file does not matter, but it is convenient to place the methods in the order the conversation would progress. This way, when reviewing the code, you do not have to jump around the file from method to method.

Add the highlighted code below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForProblem(context, name);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForProblem(context, userInfo.Name);

}

private void PromptForProblem(IDialogContext context, string name)

{

PromptDialog.Choice(context, AfterGetProblem, new List<string>() { "Crash", "Error", "Other" }, $"Welcome, {name}! What problem are you having?");

}

private async Task AfterGetProblem(IDialogContext context, IAwaitable<string> result)

{

string problem = await result;

switch (problem.ToLower())

{

case "crash":

await context.PostAsync("To solve this error, please install the latest version of the software. You can find it at http://newestversionsite");

context.ConversationData.SetValue("Issue", "Crash");

PromptForSuccess(context);

break;

case "error":

PromptForError(context);

break;

default:

PromptForIssue(context);

break;

}

}

*…the rest of the code*

In the **PromptForProblem** method, we welcome the user by **name** and then give them a choice of issue categories: Crash, Error, or Other. We outlined solutions for each of these options in the flow chart.

To handle the solutions in our dialog, we use a **switch** and check for the user’s response. If it was a crash, we suggest installing the latest version of software and store a crash as the issue in the **ConversationData**. We then call **PromptForSuccess** (which does not yet exist) to determine if it worked or not.

If the issue was an error message, we call **PromptForError** (another method which does not yet exist) to get further information.

If the issue was something else, we go straight to our **PromptForIssue** method from the previous section of the lab which just gets a text description from the user. From here it would then go on to get the user’s requested response method and to send it on to support.

1. Now, we’ll implement the **PromptForSuccess** method (and its **After** method) in the **RootDialog.cs** file. Add the highlighted code below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForProblem(context, name);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForProblem(context, userInfo.Name);

}

private void PromptForProblem(IDialogContext context, string name)

{

PromptDialog.Choice(context, AfterGetProblem, new List<string>() { "Crash", "Error", "Other" }, $"Welcome, {name}! What problem are you having?");

}

private async Task AfterGetProblem(IDialogContext context, IAwaitable<string> result)

{

string problem = await result;

switch (problem.ToLower())

{

case "crash":

await context.PostAsync("To solve this error, please install the latest version of the software. You can find it at http://newestversionsite");

context.ConversationData.SetValue("Issue", "Crash");

PromptForSuccess(context);

break;

case "error":

PromptForError(context);

break;

default:

PromptForIssue(context);

break;

}

}

private void PromptForSuccess(IDialogContext context)

{

PromptDialog.Choice(context, AfterGetSuccess, new List<string>() { "Yes", "No" }, "Did this solve the issue?");

}

private async Task AfterGetSuccess(IDialogContext context, IAwaitable<string> result)

{

string success = await result;

if (success.ToLower() == "yes")

{

await context.PostAsync("Great! Thank you for using the support bot.");

context.Wait(MessageReceivedAsync);

}

else

{

PromptForIssue(context);

}

}

*…the rest of the code*

Here, we again use **PromptDialog.Choice** to ask if the issue was solved or not. The **AfterGetSuccess** method then checks the result of this prompt. If successful, then, great! We end the conversation and start waiting again with **context.Wait**.

Otherwise, we have an issue we cannot help with so we call **PromptForIssue** from this point to get a text description to send on to support.

1. Finally, we will implement the **PromptForError** method and its **After** method in **RootDialog.cs** to handle checking on specific errors. The code for them is highlighted below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForProblem(context, name);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForProblem(context, userInfo.Name);

}

private void PromptForProblem(IDialogContext context, string name)

{

PromptDialog.Choice(context, AfterGetProblem, new List<string>() { "Crash", "Error", "Other" }, $"Welcome, {name}! What problem are you having?");

}

private async Task AfterGetProblem(IDialogContext context, IAwaitable<string> result)

{

string problem = await result;

switch (problem.ToLower())

{

case "crash":

await context.PostAsync("To solve this error, please install the latest version of the software. You can find it at http://newestversionsite");

context.ConversationData.SetValue("Issue", "Crash");

PromptForSuccess(context);

break;

case "error":

PromptForError(context);

break;

default:

PromptForIssue(context);

break;

}

}

private void PromptForSuccess(IDialogContext context)

{

PromptDialog.Choice(context, AfterGetSuccess, new List<string>() { "Yes", "No" }, "Did this solve the issue?");

}

private async Task AfterGetSuccess(IDialogContext context, IAwaitable<string> result)

{

string success = await result;

if (success.ToLower() == "yes")

{

await context.PostAsync("Great! Thank you for using the support bot.");

context.Wait(MessageReceivedAsync);

}

else

{

PromptForIssue(context);

}

}

private void PromptForError(IDialogContext context)

{

PromptDialog.Text(context, AfterGetError, "What error are you receiving?");

}

private async Task AfterGetError(IDialogContext context, IAwaitable<string> result)

{

string error = await result;

if (error.Contains("60342"))

{

await context.PostAsync("Please try running the repair tool found at http://repairtoolsite");

context.ConversationData.SetValue("Issue", "Error #60342");

PromptForSuccess(context);

}

else if (error.Contains("62890"))

{

await context.PostAsync("To solve this error, please delete the temporary files located in C:\\tempfilelocation");

context.ConversationData.SetValue("Issue", "Error #62890");

PromptForSuccess(context);

}

else

{

context.ConversationData.SetValue("Issue", error);

PromptForContactMethod(context);

}

}

*…the rest of the code*

This time, we use **PromptDialog.Text** to get a text description of the error. We check if either of the errors we know about are mentioned in the text (code 60342 or 62890). If so, we send the user a suggestion regarding how to handle each. We then call **PromptForSuccess** to see if this suggestion was successful or not.

If we do not find a known error code, we use **Conversation.SetValue** to store the error message from the user as the Issue. We then call **PromptForContactMethod** to finish the process of sending the ticket to support.

When all the changes have been implemented, your final **RootDialog.cs** file should look like this:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Threading.Tasks;

using Microsoft.Bot.Builder.Dialogs;

using Microsoft.Bot.Connector;

using Microsoft.Bot.Builder.FormFlow;

using Microsoft.Bot.Builder.FormFlow.Advanced;

#pragma warning disable CS1998

namespace SupportBot

{

[Serializable]

public class RootDialog : IDialog<object>

{

public async Task StartAsync(IDialogContext context)

{

await context.PostAsync("Welcome to the support bot!");

context.Wait(MessageReceivedAsync);

}

public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)

{

string name;

if (context.ConversationData.TryGetValue("Name", out name))

{

PromptForProblem(context, name);

}

else

{

context.Call(FormDialog.FromForm<UserInfo>(UserInfo.BuildForm, FormOptions.PromptInStart), this.AfterUserInfoForm);

}

}

public async Task AfterUserInfoForm(IDialogContext context, IAwaitable<UserInfo> result)

{

UserInfo userInfo = await result;

context.ConversationData.SetValue("Name", userInfo.Name);

context.ConversationData.SetValue("PhoneNumber", userInfo.PhoneNumber);

context.ConversationData.SetValue("EmailAddress", userInfo.EmailAddress);

context.ConversationData.SetValue("Department", userInfo.Department);

PromptForProblem(context, userInfo.Name);

}

private void PromptForProblem(IDialogContext context, string name)

{

PromptDialog.Choice(context, AfterGetProblem, new List<string>() { "Crash", "Error", "Other" }, $"Welcome, {name}! What problem are you having?");

}

private async Task AfterGetProblem(IDialogContext context, IAwaitable<string> result)

{

string problem = await result;

switch (problem.ToLower())

{

case "crash":

await context.PostAsync("To solve this error, please install the latest version of the software. You can find it at http://newestversionsite");

context.ConversationData.SetValue("Issue", "Crash");

PromptForSuccess(context);

break;

case "error":

PromptForError(context);

break;

default:

PromptForIssue(context);

break;

}

}

private void PromptForSuccess(IDialogContext context)

{

PromptDialog.Choice(context, AfterGetSuccess, new List<string>() { "Yes", "No" }, "Did this solve the issue?");

}

private async Task AfterGetSuccess(IDialogContext context, IAwaitable<string> result)

{

string success = await result;

if (success.ToLower() == "yes")

{

await context.PostAsync("Great! Thank you for using the support bot.");

context.Wait(MessageReceivedAsync);

}

else

{

PromptForIssue(context);

}

}

private void PromptForError(IDialogContext context)

{

PromptDialog.Text(context, AfterGetError, "What error are you receiving?");

}

private async Task AfterGetError(IDialogContext context, IAwaitable<string> result)

{

string error = await result;

if (error.Contains("60342"))

{

await context.PostAsync("Please try running the repair tool found at http://repairtoolsite");

context.ConversationData.SetValue("Issue", "Error #60342");

PromptForSuccess(context);

}

else if (error.Contains("62890"))

{

await context.PostAsync("To solve this error, please delete the temporary files located in C:\\tempfilelocation");

context.ConversationData.SetValue("Issue", "Error #62890");

PromptForSuccess(context);

}

else

{

context.ConversationData.SetValue("Issue", error);

PromptForContactMethod(context);

}

}

private void PromptForIssue(IDialogContext context)

{

PromptDialog.Text(context, AfterGetIssue, "Please explain in detail the issue you are having.");

}

private async Task AfterGetIssue(IDialogContext context, IAwaitable<string> result)

{

string issue = await result;

context.ConversationData.SetValue("Issue", $"{issue}");

PromptForContactMethod(context);

}

private void PromptForContactMethod(IDialogContext context)

{

PromptDialog.Choice(context, AfterGetContactMethod, new List<string>() { "Email", "Phone" }, $"How would you prefer for support to contact you?");

}

private async Task AfterGetContactMethod(IDialogContext context, IAwaitable<string> result)

{

string contactMethod = await result;

string name;

string email;

string phoneNumber;

string department;

string issue;

context.ConversationData.TryGetValue("Name", out name);

context.ConversationData.TryGetValue("EmailAddress", out email);

context.ConversationData.TryGetValue("PhoneNumber", out phoneNumber);

context.ConversationData.TryGetValue("Department", out department);

context.ConversationData.TryGetValue("Issue", out issue);

if (contactMethod.ToLower() == "phone")

{

//Have support contact {name} from the {department} department at {phoneNumber} regarding {issue} via phone

}

else

{

//Have support contact {name} from the {department} department at {email} regarding {issue} via email

}

await context.PostAsync($"You will be contacted via {contactMethod} within the next hour. Thank you for using the support bot.");

context.Wait(MessageReceivedAsync);

}

}

}

1. Finally, once again execute the bot as before in Visual Studio (via **Start Debugging** or **F5**) and test it out. Try all the different path options including the different error messages and see the results. Also test the error codes (60342 and 62890) to see those results.

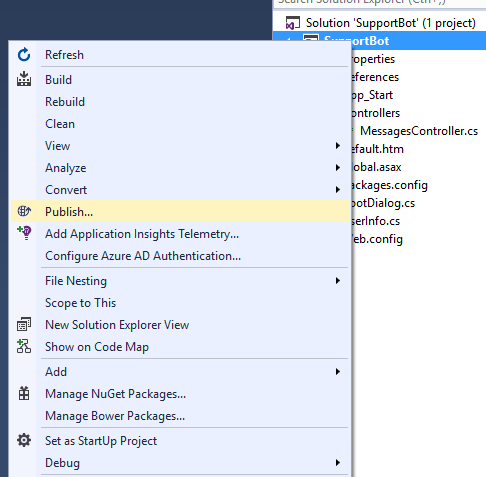
Note that after each time through you can just send a message (such as **Hi**) to restart the conversation and it will pick up where it left off, skipping over the section where it asks for **UserInfo** again. You can also close and reopen the emulator to start over from the beginning again.

|  |
| --- |
| Deploying the Bot |

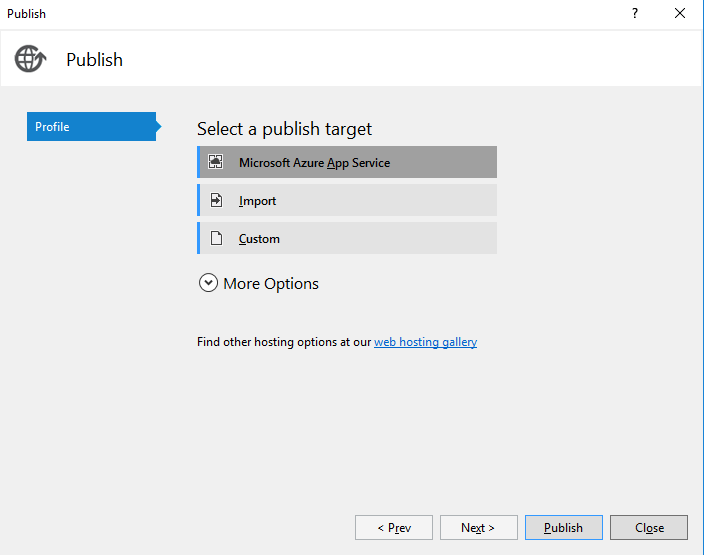
Before deploying the bot, you may wish to first edit the **default.htm** web page that is a part of the project. This is the page your web browser has been opening when you first execute the project in Visual Studio. The **default.htm** page allows you to publish a description, terms of use for your bot, and any other relevant information about your bot.

To deploy the bot, we first need to publish the project to the Azure App Service.

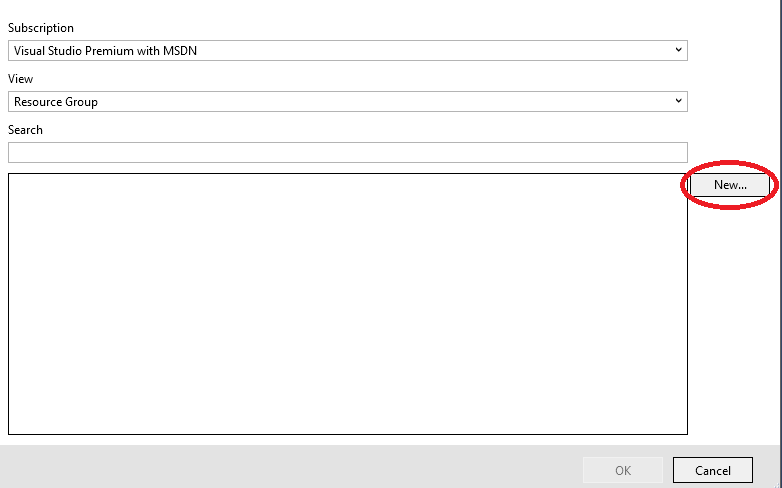
1. Right click on the project and select “Publish”.



1. Click the **Microsoft Azure App Service** option.



1. Make sure your subscription is selected and click **New…**

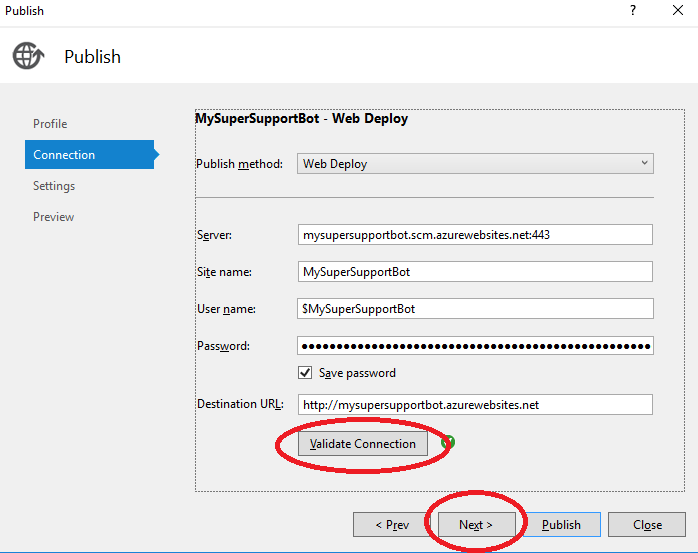


1. Give your API App a name and, if necessary, add a Resource Group and App Service Plan for it using the **New…** buttons for each. The App Service Plan will pop up a new window allowing you to configure it. The Size determines how much it will cost so you can just select Free for now. If this were a production bot we would probably want something more powerful.

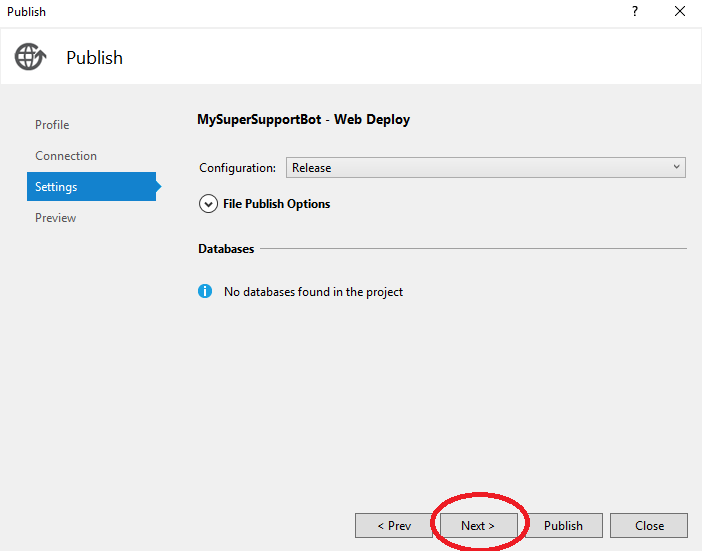
Then, click the **Create** button.



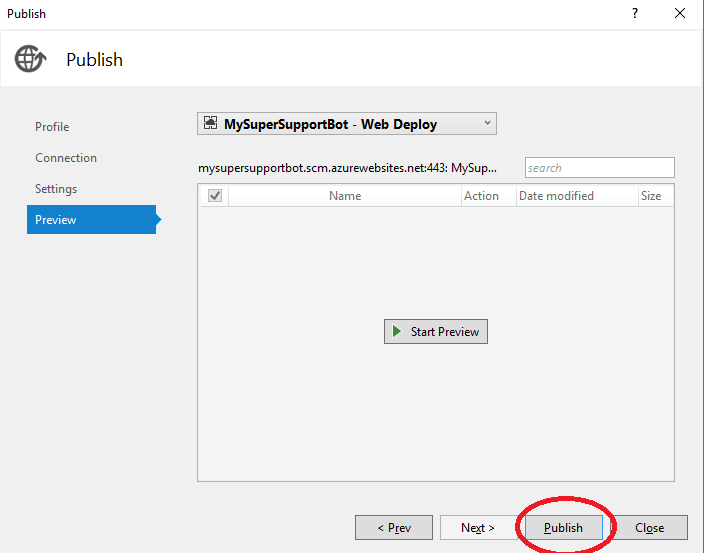
1. Wait while Visual Studio goes through and sets up the Azure environment for the bot. Then, click the **Validate** button to ensure everything was successful. A green check box will appear indicating it was. If so, make a note of the **Destination URL** and then click the **Next** button.



1. You can choose to publish it as a **Release** or **Debug** version depending. If you would like to be able to debug it choose **Debug,** otherwise just leave it on **Release** as the default value and click **Next**.



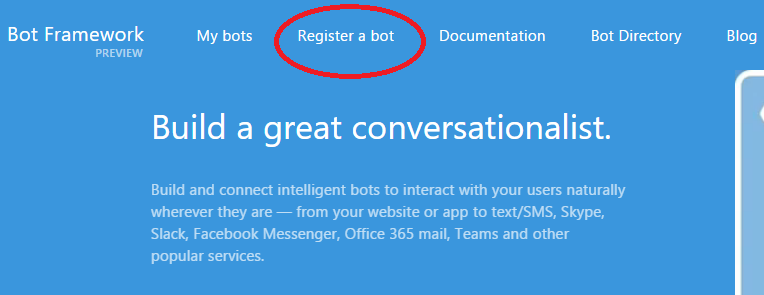
1. On the publish screen, click the **Publish** button to finally publish to Azure.



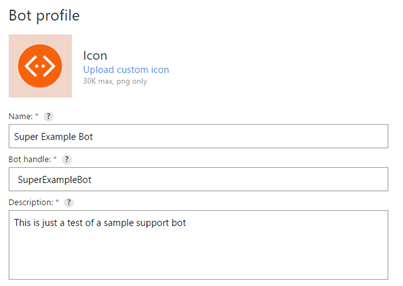
1. Visual Studio will once again run for a while as it publishes and when it finishes will open your web browser and browse to the newly published bot site.
2. The next step is to register the bot. Browse to the following URL:

<http://www.botframework.com>

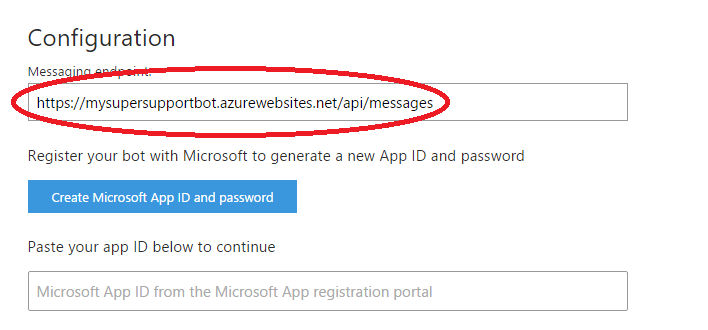
1. Click the **Register a Bot** link. You will need to sign in if you are not already signed in.



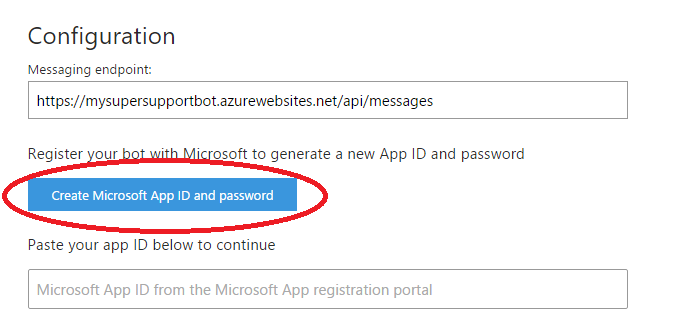
1. Fill in the **Name**, **Bot handle**, and a **Description** for your bot. Make sure to write down your **Bot handle** for later as you can use it as your **BotId** in the configuration file. If you have a custom icon you would like to use you can also upload it here.



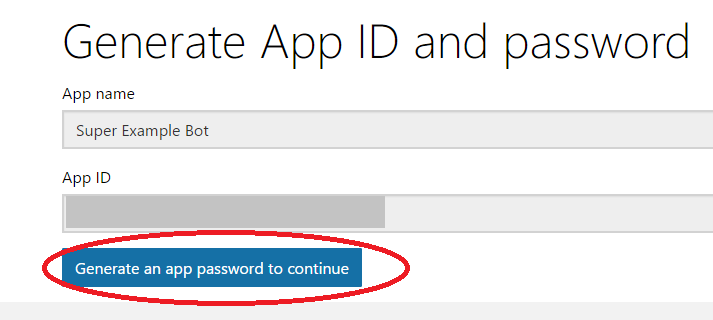
1. Configure the bot to use the deployed endpoint that was published from Visual Studio. This will be the URL plus /api/messages. However, make sure this URL is using https:// instead of http://.



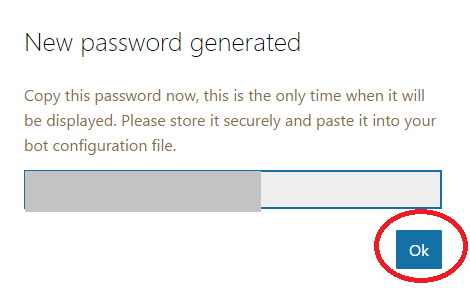
1. Click the **Create Microosft App ID and password** button to generate a new App ID and password.



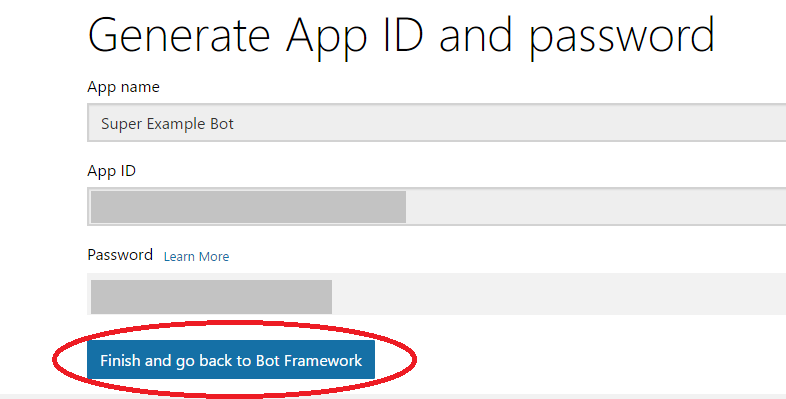
1. A new window or tab will open to generate the App ID and password. Click the **Generate an app password to continue** button.



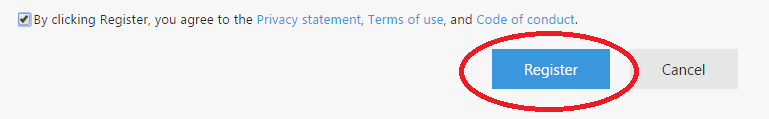
1. Copy the App ID and the password it generates and save them somewhere safe, then click **Ok**. You will need them later for the configuration file.



1. Click the **Finish and go back to Bot Framework** button



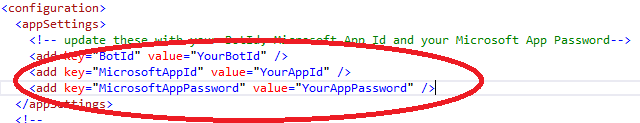
1. It should have filled in your app ID into the configuration box. Scroll down to the bottom of the page and click the **Register** button. Note that you must review and agree to the **Privacy Statement**, **Terms of Use**, and **Code of conduct** first.



1. Fix any errors if there are any. Otherwise, you will get a message that says **Bot Created**. Click **OK**.

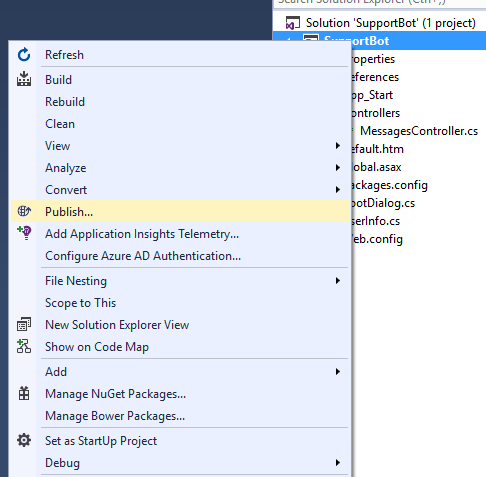


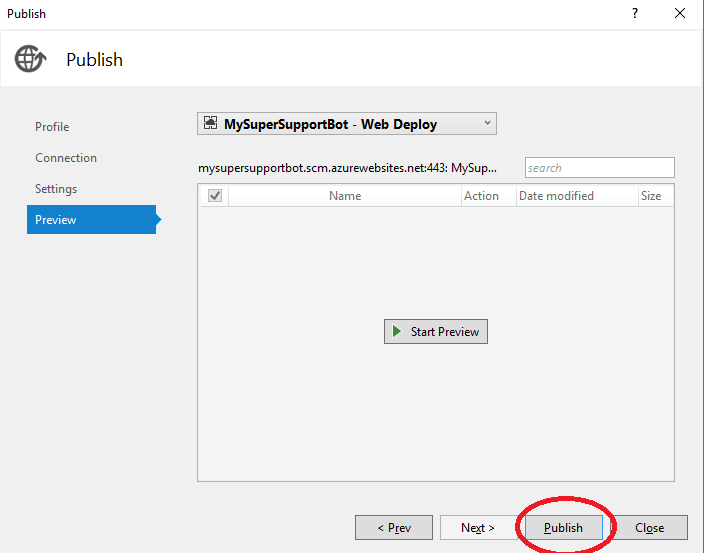
1. Now that the bot has been created, you need to go back to the Visual Studio project. However, leave your web browser open to the bot page as we will be coming back to it.
2. In Visual Studio, open the **Web.config** file. Go to the **appSettings** section and modify it to have your **BotId**, **MicrosoftAppId**, and **MicrosoftAppPassword**. The **BotId** is the **Bot Handle** from step 11. The **MicrosoftAppId** and **MicrosoftAppPassword** are the App ID and password from step 15.



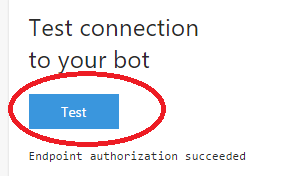
After making these changes, you will receive an error if you try to run the project in Visual Studio and test it using the Microsoft Bot Channel Emulator. You can comment these lines out when testing and then uncomment when ready to deploy.

1. Republish the bot by right clicking on the project and selecting **Publish** and then clicking **Publish** in the window that appears.

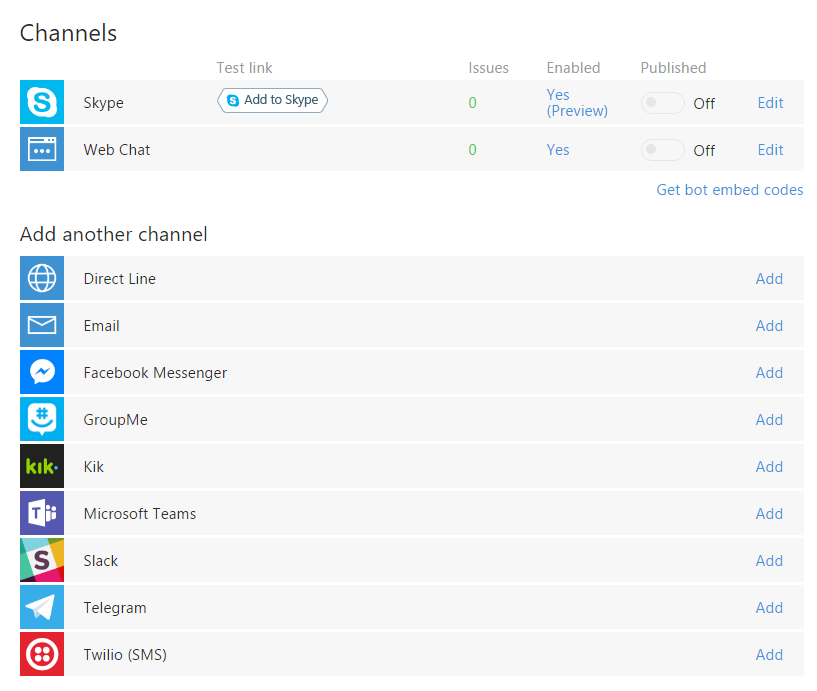




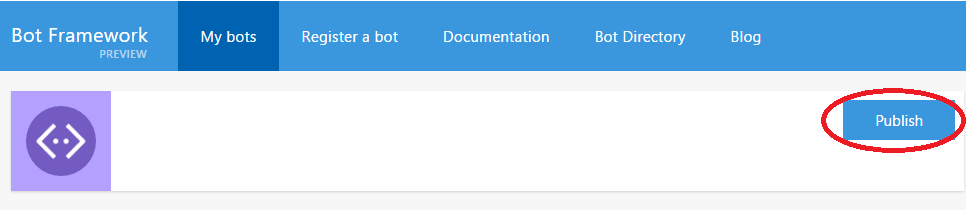
1. Once the bot has been republished, go back to your web browser and click the **Test** button to test authorization. You should get a message that says **Endpoint authorization succeeded**. You can also test it with the **Chat** tool on the same page.



1. Select any channels you wish to add your bot to using the **Channels** section. For this lab bot, you may not wish to add it to any channels but this is where you would configure channels if you were deploying a production bot. Each channel has its own set of instructions for set up.



1. Finally, if you would like to publish your bot to the bot directory, click the **Publish** button and fill out the information required, then click **Submit for Review**.



|  |
| --- |
| Deleting the Bot |

### Delete the bot registration and app service

If you are no longer using a bot, you may wish to delete it.

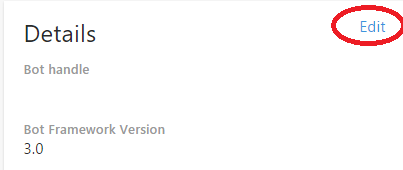
1. Browse to the Bot Framework site (if not already there):

<http://www.botframework.com>

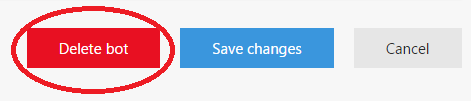
1. Click on **My Bots**. If you have multiple bots you will have to select which one you want to delete.



1. Click the **Edit** button in the **Details** section of the bot page.



1. At the bottom of the page, click Delete bot and then in the confirmation window click Delete bot again if you are sure you wish to delete it.

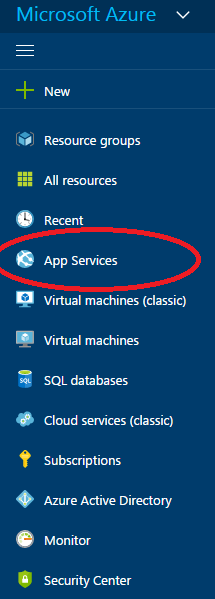


A message should pop up confirming that the bot was deleted.

1. Next, you will want to delete the bot from Azure. First, browse the following URL:

<https://portal.azure.com>

1. Click on App Services.



1. Click on the app service you created earlier when publishing the bot through Visual Studio.
2. Click on the Delete button and click Yes to confirm that you would like to delete it.



For more detail on the Microsoft Bot Framework, see the Bot Framework documentation:

<https://docs.botframework.com/>

### Conclusion

This concludes the *Microsoft Bot Framework* lab. You have successfully created a bot using Visual Studio and .NET and published it to Azure.

To expand on the lab, you could add additional support cases that the bot can handle. You could also add additional fields to the **UserInfo** form to collect and store.

In addition, you can review Appendix A for a high-level overview of the Azure Bot Service and how the differences between using it to build a bot versus how the bot in this lab was built.

|  |
| --- |
| Appendix A - Azure Bot Service |

The Azure Bot Service is a related service for building bots. We won’t go into too much detail in this document regarding building bots for it but it’s worth highlighting how it’s similar and how it’s different to the base Bot Framework.

The Azure Bot Service sets up everything through the web browser. When you first create a bot in Azure Bot Service, it is deployed right away using similar steps to the Bot Framework. These include generating a **App ID** and **App Password**.

The Azure Bot Service then sets up the code within Azure for you based on a selected template. The template include a basic bot and a form bot, among others.

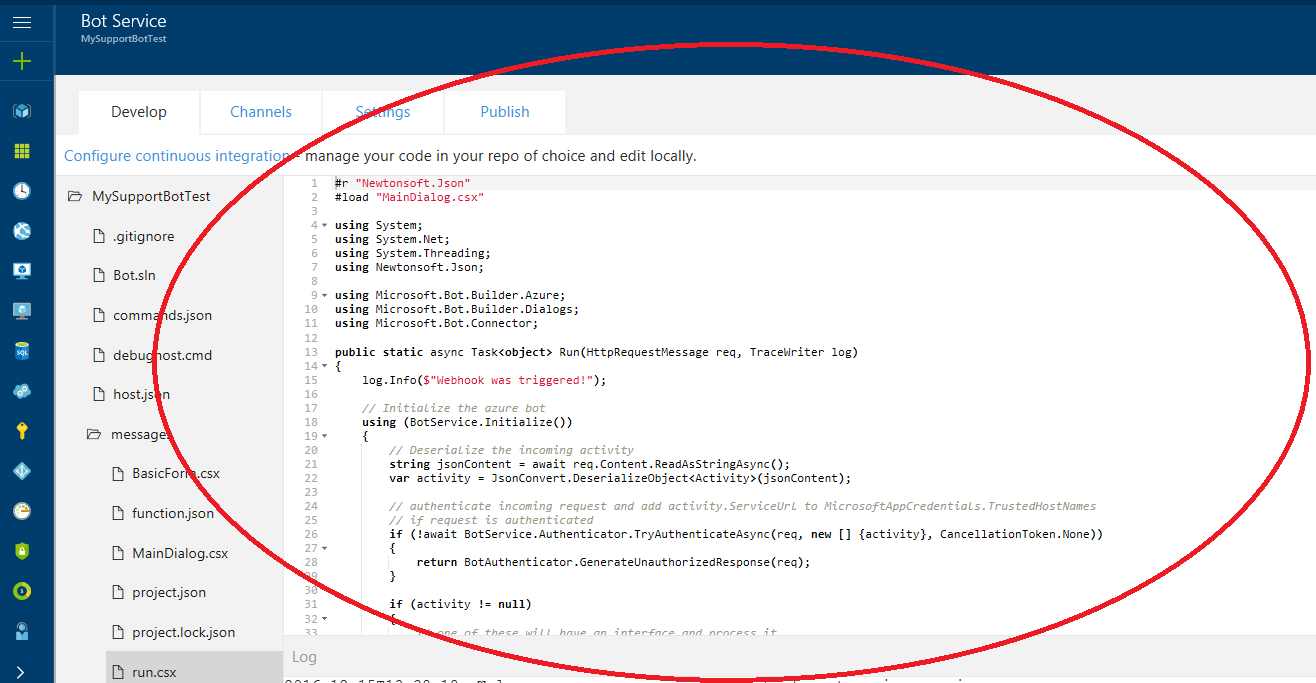
The source code itself is setup differently from how we built the project earlier as well. The Bot Service builds the source control files as .csx (C# Script) files instead of .cs (C#) files. The code is also edited directly in the web browser.

For more detail on the Azure Bot Service, see the Azure Bot Service site:

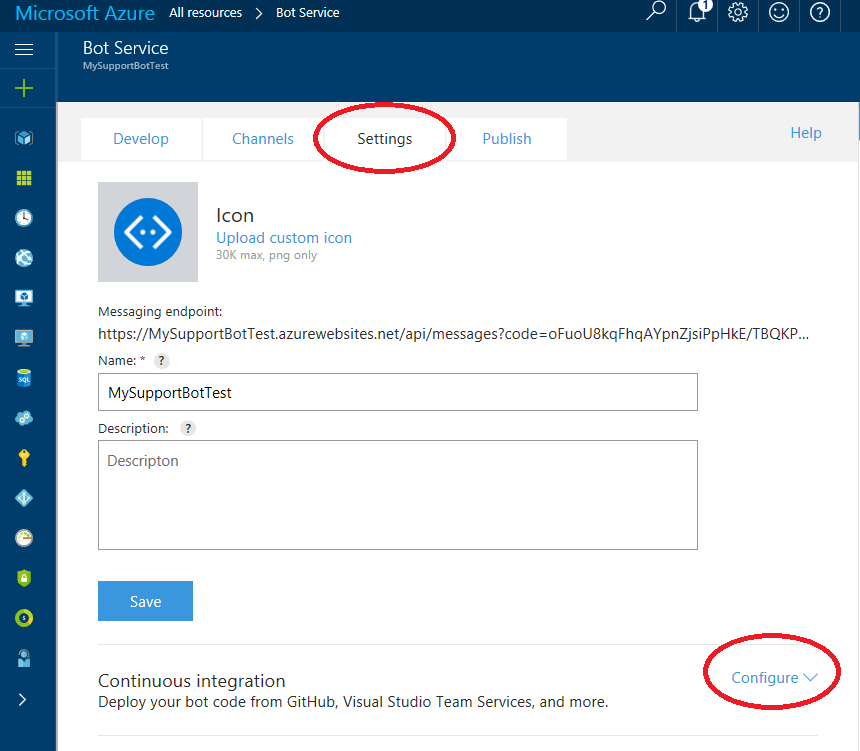
<https://azure.microsoft.com/en-us/services/bot-service/>

For documentation for the Azure Bot Service, see the below link:

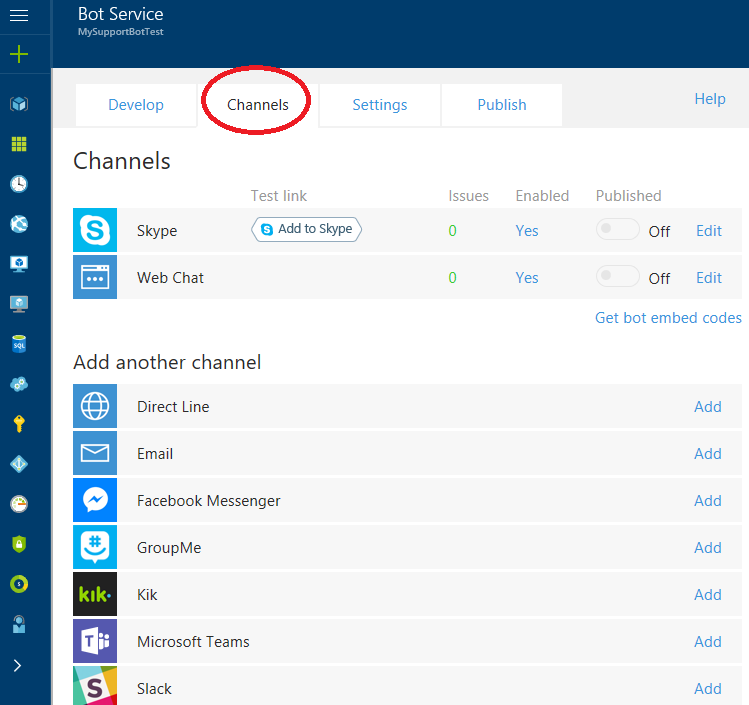
<https://docs.botframework.com/en-us/azure-bot-service/>



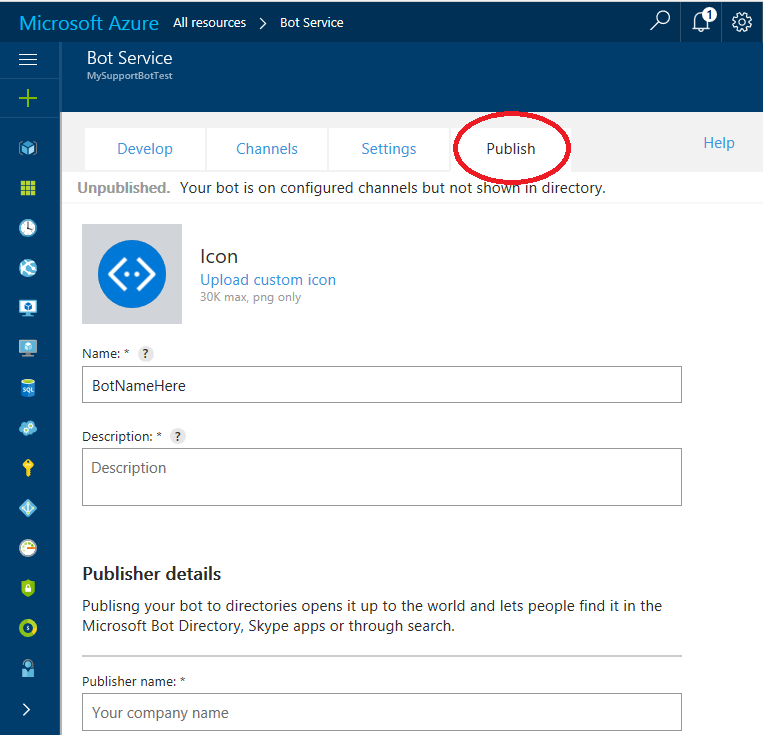
However, while it is very convenient there are some downsides to editing the code in the browser. For instance, it lacks debugging features and Intellisense. If you would like to be able to use these features, you can setup the continuous integration. This allows you to store your code in source control and have it integrate with the bot. Source control options include Visual Studio Team Services and GitHub, among others. It also allows the use of a file sharing platform such as OneDrive or Dropbox. You can enable continuous integration on the **Settings** page.



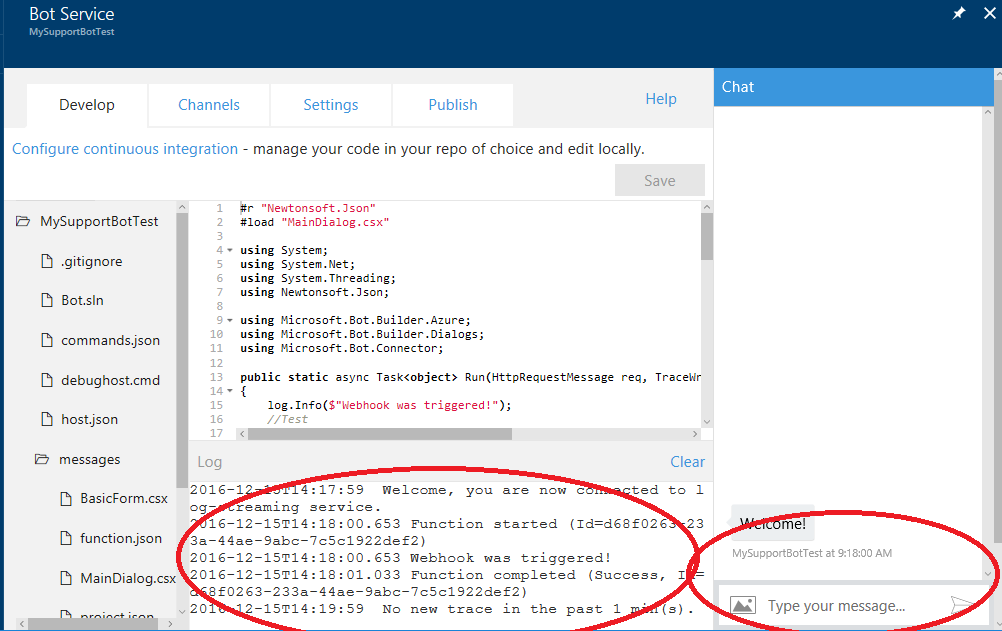
The Azure Bot Service also allows channel configuration in the same way as channel configuration worked for the bot we built above.



The Azure Bot Service allows publishing to the Microsoft Bot Directory as above using the **Publish** page as well.



Finally, there is a chat pane built into the Azure Bot Service that allows you to test the bot as you are developing it as well as a log to display any error messages.



|  |
| --- |
| Terms of Use |

© 2016 Microsoft Corporation. All rights reserved.

By using this Hands-on Lab, you agree to the following terms:

The technology/functionality described in this Hands-on Lab is provided by Microsoft Corporation in a “sandbox” testing environment for purposes of obtaining your feedback and to provide you with a learning experience. You may only use the Hands-on Lab to evaluate such technology features and functionality and provide feedback to Microsoft.  You may not use it for any other purpose. You may not modify, copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell this Hands-on Lab or any portion thereof.

COPYING OR REPRODUCTION OF THE HANDS-ON LAB (OR ANY PORTION OF IT) TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THIS HANDS-ON LAB PROVIDES CERTAIN SOFTWARE TECHNOLOGY/PRODUCT FEATURES AND FUNCTIONALITY, INCLUDING POTENTIAL NEW FEATURES AND CONCEPTS, IN A SIMULATED ENVIRONMENT WITHOUT COMPLEX SET-UP OR INSTALLATION FOR THE PURPOSE DESCRIBED ABOVE.  THE TECHNOLOGY/CONCEPTS REPRESENTED IN THIS HANDS-ON LAB MAY NOT REPRESENT FULL FEATURE FUNCTIONALITY AND MAY NOT WORK THE WAY A FINAL VERSION MAY WORK.  WE ALSO MAY NOT RELEASE A FINAL VERSION OF SUCH FEATURES OR CONCEPTS.  YOUR EXPERIENCE WITH USING SUCH FEATURES AND FUNCITONALITY IN A PHYSICAL ENVIRONMENT MAY ALSO BE DIFFERENT.

**FEEDBACK**.  If you give feedback about the technology features, functionality and/or concepts described in this Hands-on Lab to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose.  You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software or service that includes the feedback.  You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them.  These rights survive this agreement.

MICROSOFT CORPORATION HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE HANDS-ON LAB, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT.  MICROSOFT DOES NOT MAKE ANY ASSURANCES OR REPRESENTATIONS WITH REGARD TO THE ACCURACY OF THE RESULTS, OUTPUT THAT DERIVES FROM USE OF THE VIRTUAL LAB, OR SUITABILITY OF THE INFORMATION CONTAINED IN THE VIRTUAL LAB FOR ANY PURPOSE.

DISCLAIMER

This lab contains only a portion of the features and enhancements in Microsoft Azure SQL Data Warehouse. Some of the features might change in future releases of the product.