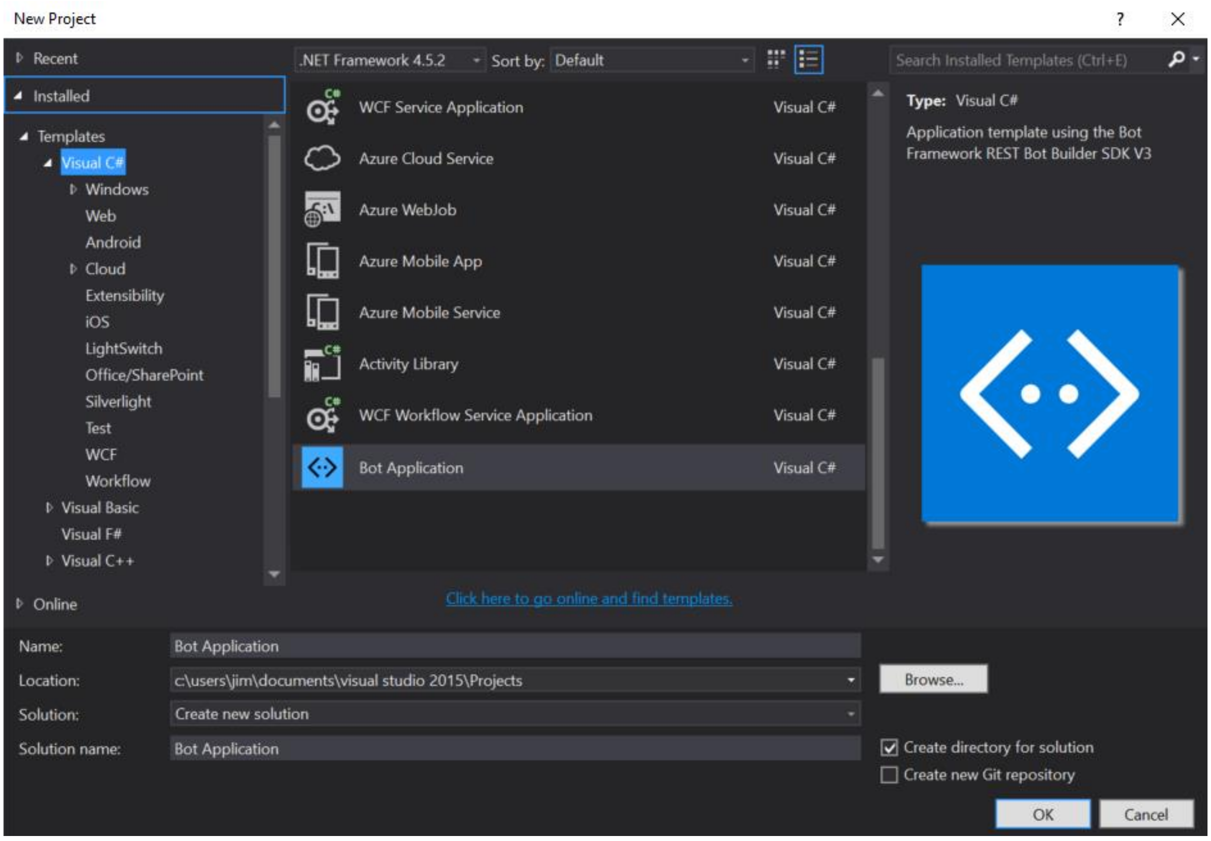
Working Locally

1: Close any open Visual Studio instances. Download Bot Framework template from: **aka.ms/bf-bc-vstemplate**

2: Navigate to: "%USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C#\"

3: Copy the downloaded **Bot Application.zip** zip file from step 1 to the folder from step 2. Do not unzip, just copy the zip file.

4: Open Visual Studio, File > New Project. Under Templates > Visual C# you should see a new template for **Bot Application**:



New Project

.NET Framework 4.5.2 Sort by: Default

Search Installed Templates (Ctrl+E)

- Recent
- Installed
  - Templates
    - Visual C#
      - Windows
      - Web
      - Android
      - Cloud
      - Extensibility
      - iOS
      - LightSwitch
      - Office/SharePoint
      - Silverlight
      - Test
      - WCF
      - Workflow
    - Visual Basic
    - Visual F#
    - Visual C++
- Online

WCF Service Application	Visual C#
Azure Cloud Service	Visual C#
Azure WebJob	Visual C#
Azure Mobile App	Visual C#
Azure Mobile Service	Visual C#
Activity Library	Visual C#
WCF Workflow Service Application	Visual C#
Bot Application	Visual C#

Type: Visual C#

Application template using the Bot Framework REST Bot Builder SDK V3

Click here to go online and find templates.

Name:	Bot Application
Location:	c:\users\jim\documents\visual studio 2015\Projects
Solution:	Create new solution
Solution name:	Bot Application

Browse...

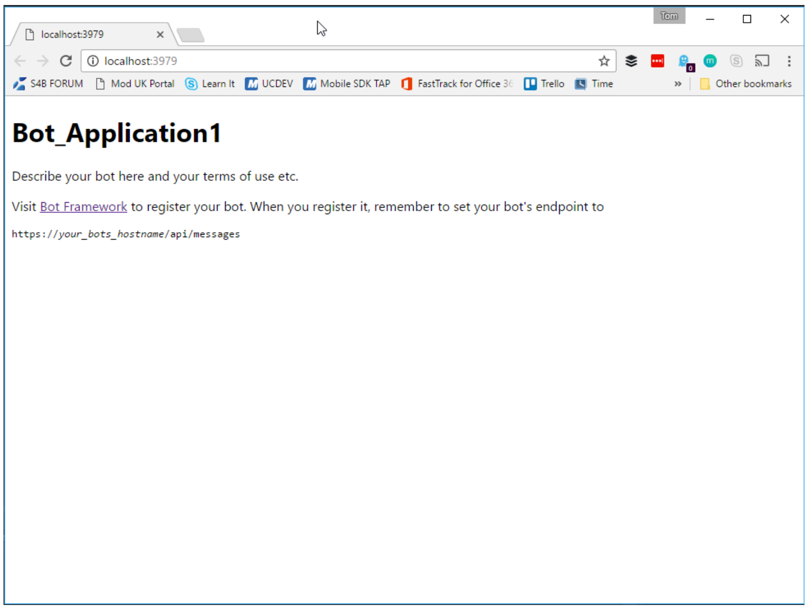
☑ Create directory for solution

☐ Create new Git repository

OK Cancel

5: Create a new project with this template

6: Start Debugging (F5 / green play button icon) to make sure the project builds OK. You should see this web page:



Bot_Application1

Describe your bot here and your terms of use etc.

Visit Bot Framework to register your bot. When you register it, remember to set your bot's endpoint to

`https://your_bots_hostname/api/messages`

Now that we have the base template we can make some changes to it. We're going to start with small changes and gradually add different things throughout the workshop.

First, we're going to get the user's name and ID. We could use this information in a real-world solution to identify the user.

1: open the Controllers/MessagesController.cs file. There is a single public Post method – this is where messages sent from the Bot Framework "arrive". Within the Post method look for the following lines:

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

We're going to change this a little. First let's add a welcome message:

```
var welcomeMessage = $"Hello {activity.From.Name} ({activity.From.Id}).";
```

Then, let's add that text into the reply that's already there:

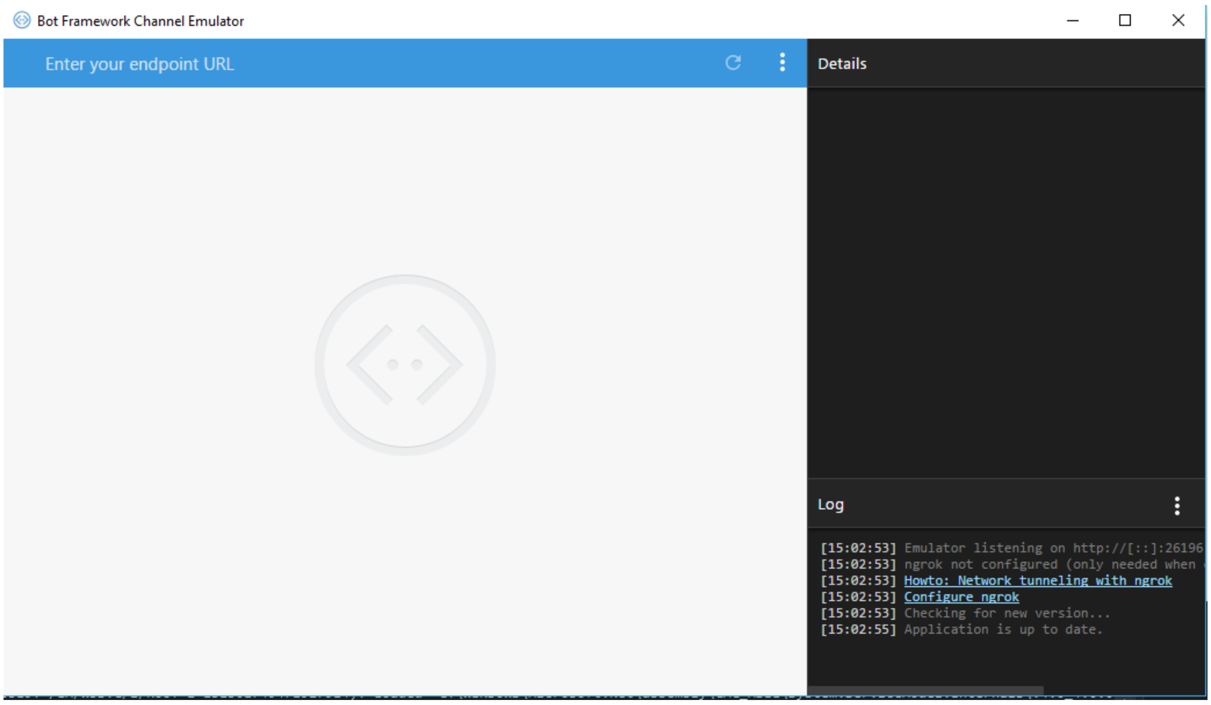
```
Activity reply = activity.CreateReply($"{welcomeMessage} You sent {activity.Text}
which was {length} characters");
```

The new code now looks like:

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

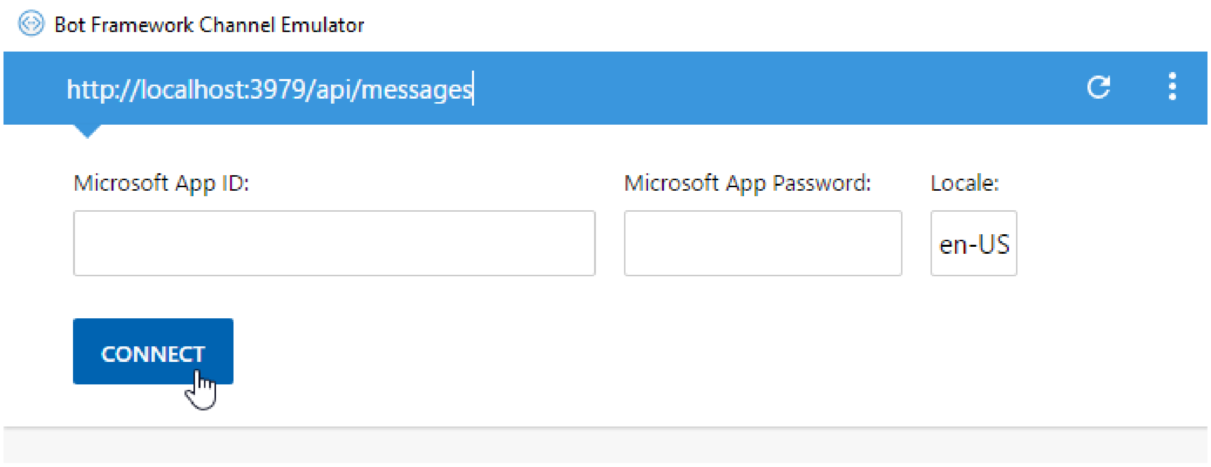
2: Rebuild the solution and make sure there are no errors. You should see the same web page as before. Keep the project running in debugging mode.

3: Now let's test this code locally. (If you haven't already download the Microsoft Bot Framework Channel Emulator from https://emulator.botframework.com/). Install the Emulator by running the file **botframework-emulator-Setup-3.5.25.exe** and walking through the install process. Once running, it should look like this:



```
[15:02:53] Emulator listening on http://[::]:26196
[15:02:53] ngrok not configured (only needed when
[15:02:53] Howto: Network tunneling with ngrok
[15:02:53] Configure ngrok
[15:02:53] Checking for new version...
[15:02:55] Application is up to date.
```

Copy the URL of the webpage from your project (for instance http://localhost:3979), plus the path /api/messages and click Connect (leave the Microsoft App ID and password blank):



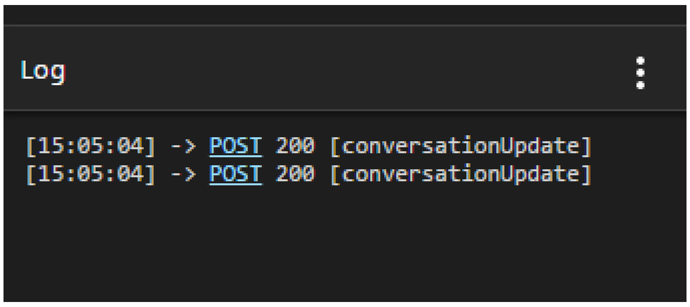
http://localhost:3979/api/messages

Microsoft App ID:

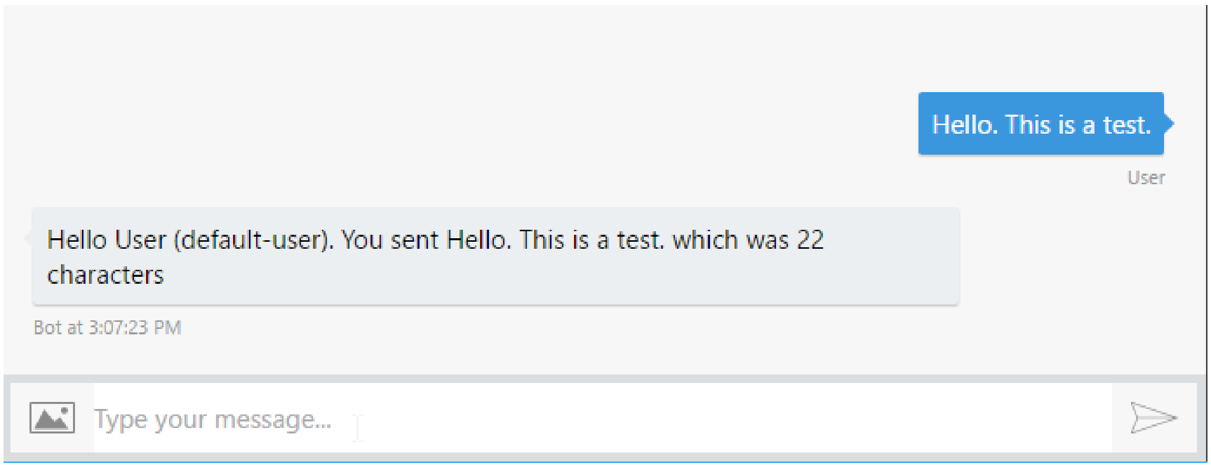
Microsoft App Password:

Locale: en-US

Check the Log (bottom right) for a POST 200 message indicating that the emulator is talking to the API correctly. If so, try typing a message using the text box at the bottom of the window.



```
[15:05:04] -> POST 200 [conversationUpdate]
[15:05:04] -> POST 200 [conversationUpdate]
```

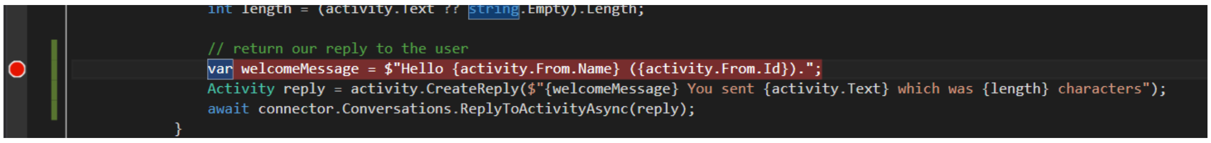

Hello. This is a test.

User

Hello User (default-user). You sent Hello. This is a test. which was 22 characters

Bot at 3:07:23 PM

4: Now go back to the code and add a breakpoint somewhere in the Post method call such as on our welcome message:


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

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

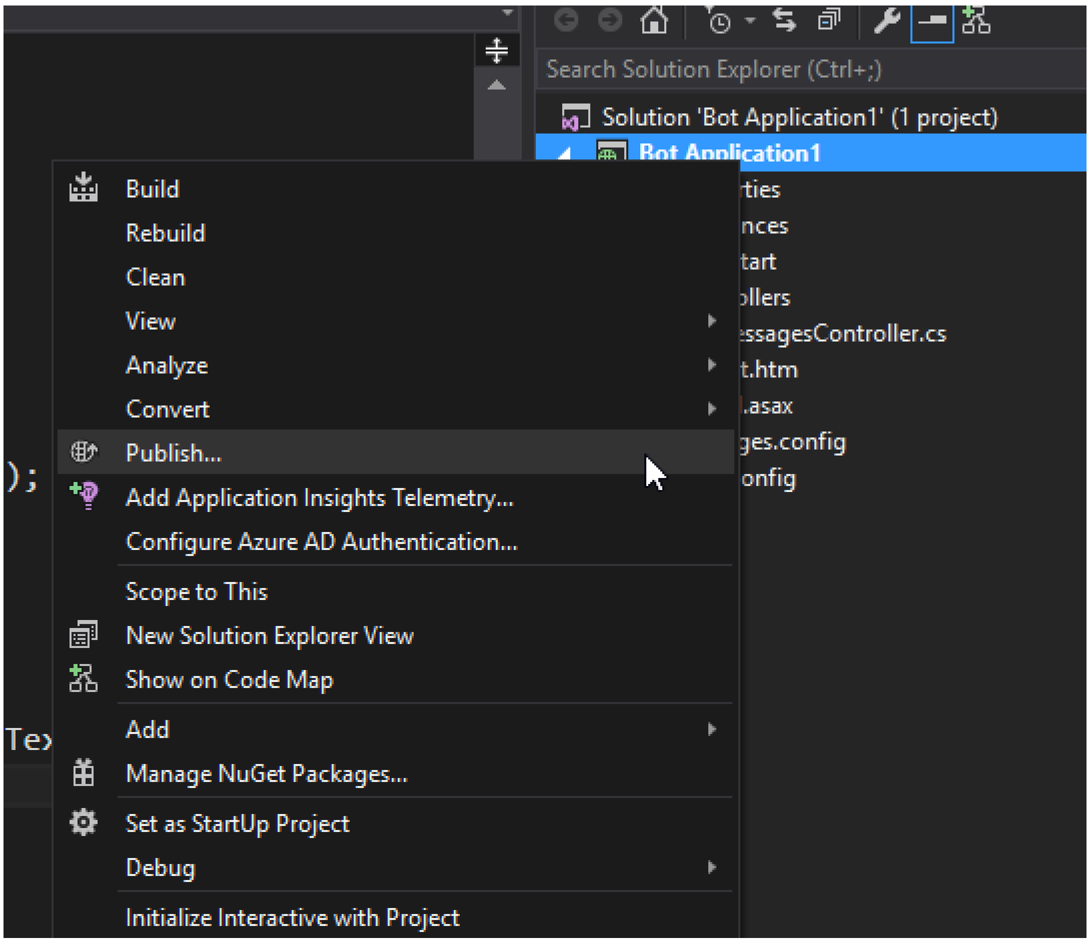
Go back to the emulator and send another message. The breakpoint should be hit. Use Visual Studio to explore the properties of the activity object. This is sent with every call to the API and contains information about the message, the user which sent it, the conversation which the message is a part of and the delivery mechanism being used (in this case, the emulator).

Using this workflow you can build out a fully functional bot without ever needing to publish anywhere and easily test different interactions. The emulator allows you to simulate users joining and leaving, typing events, and conversation termination events. You can also create new conversations quickly.

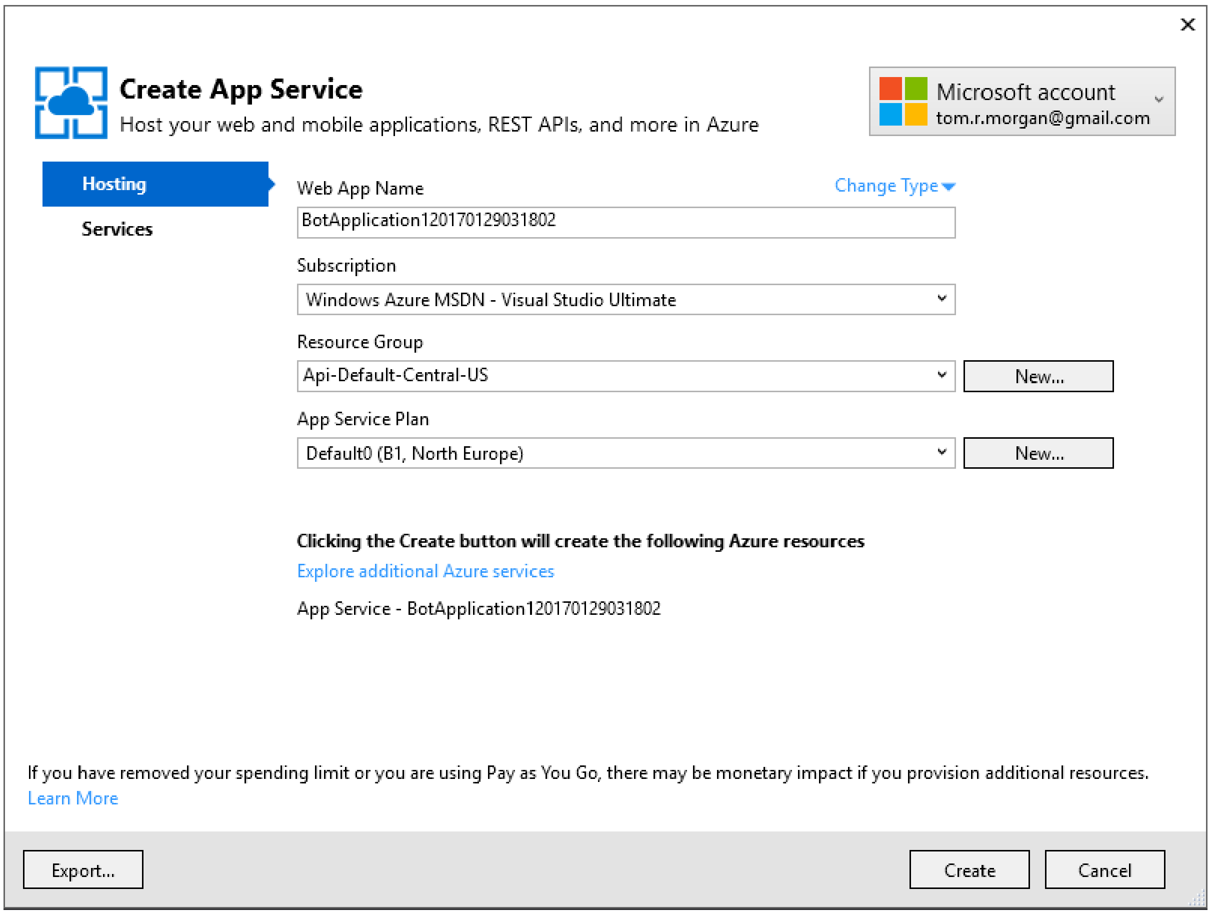
Stop! You're at the end of this section so please don't go any further until asked to.

Publishing & Registering

1: To publish our API, right click on the Solution in the Solution Explorer and click Publish:


Search Solution Explorer (Ctrl+;)
Solution 'Bot Application1' (1 project)
Bot Application1
Build
Rebuild
Clean
View
Analyze
Convert
Publish...
Add Application Insights Telemetry...
Configure Azure AD Authentication...
Scope to This
New Solution Explorer View
Show on Code Map
Add
Manage NuGet Packages...
Set as StartUp Project
Debug
Initialize Interactive with Project

Choose Microsoft Azure App Service. Log into Azure, then click the New button to add a new Resource. Accept the defaults (or change them if you know what you're doing) and click Create:



Web App Name: BotApplication120170129031802

Subscription: Windows Azure MSDN - Visual Studio Ultimate

Resource Group: Api-Default-Central-US

App Service Plan: Default0 (B1, North Europe)

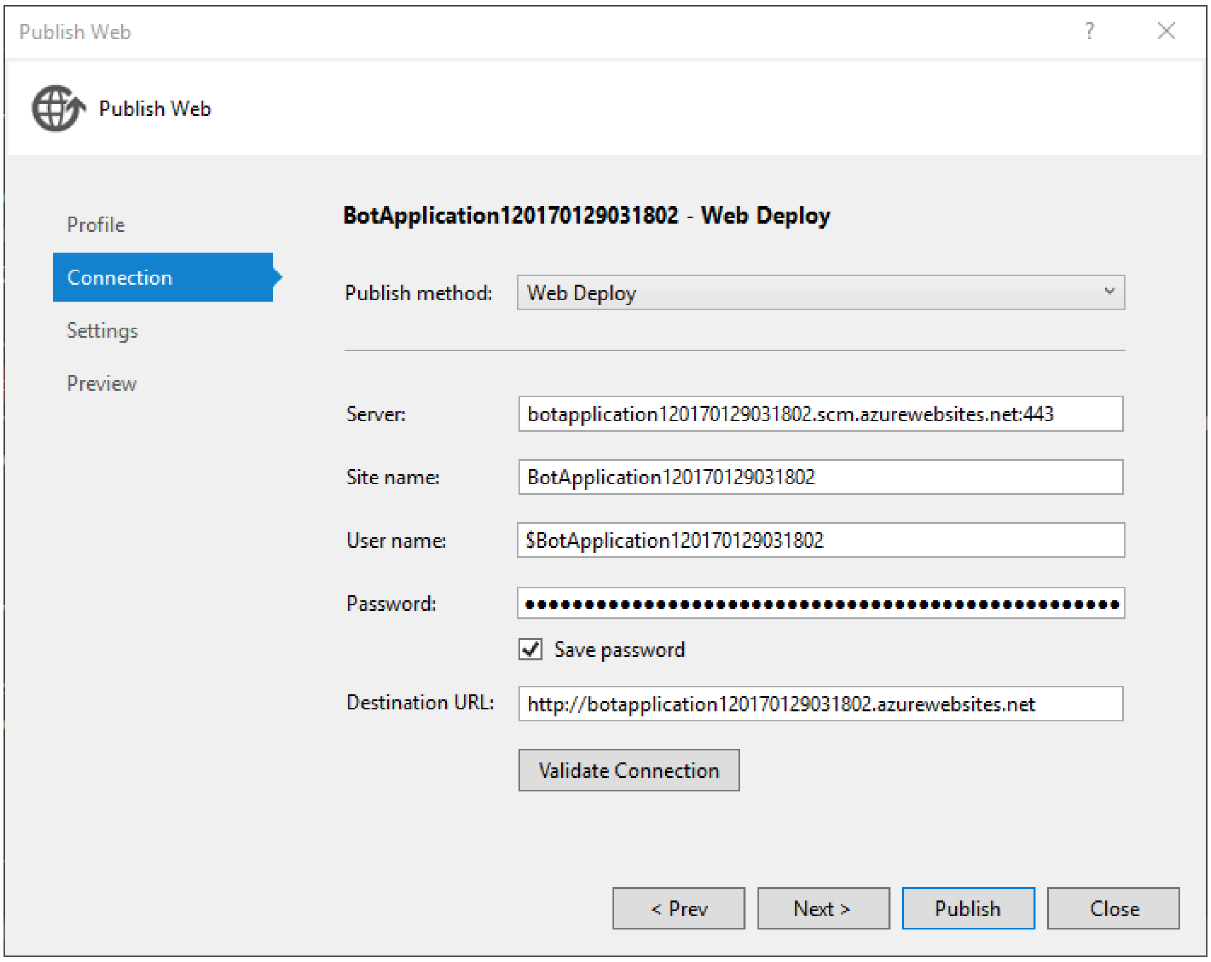
Clicking the Create button will create the following Azure resources

Explore additional Azure services

App Service - BotApplication120170129031802

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

Once the Web App has been created, click Publish:



BotApplication120170129031802 - Web Deploy

Publish method: Web Deploy

Server: botapplication120170129031802.scm.azurewebsites.net:443

Site name: BotApplication120170129031802

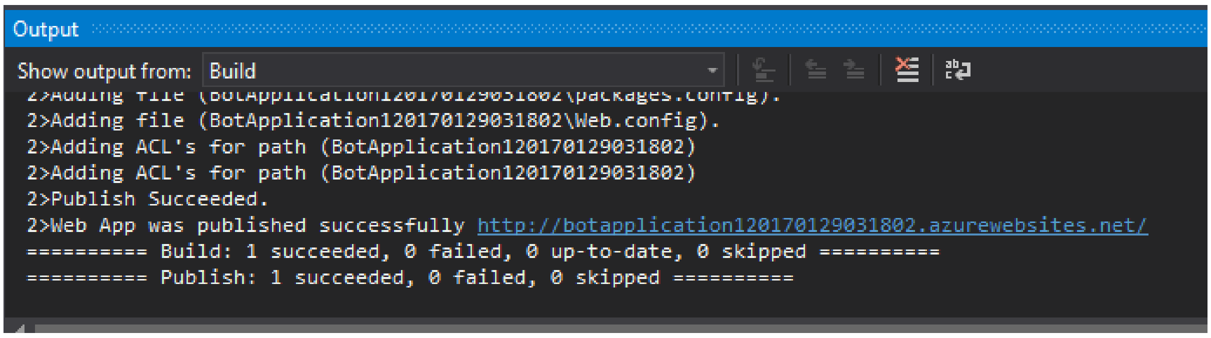
User name: $BotApplication120170129031802

Password: ••

[✓] Save password

Destination URL: http://botapplication120170129031802.azurewebsites.net

Watch the Output window for notification that publishing has succeeded:

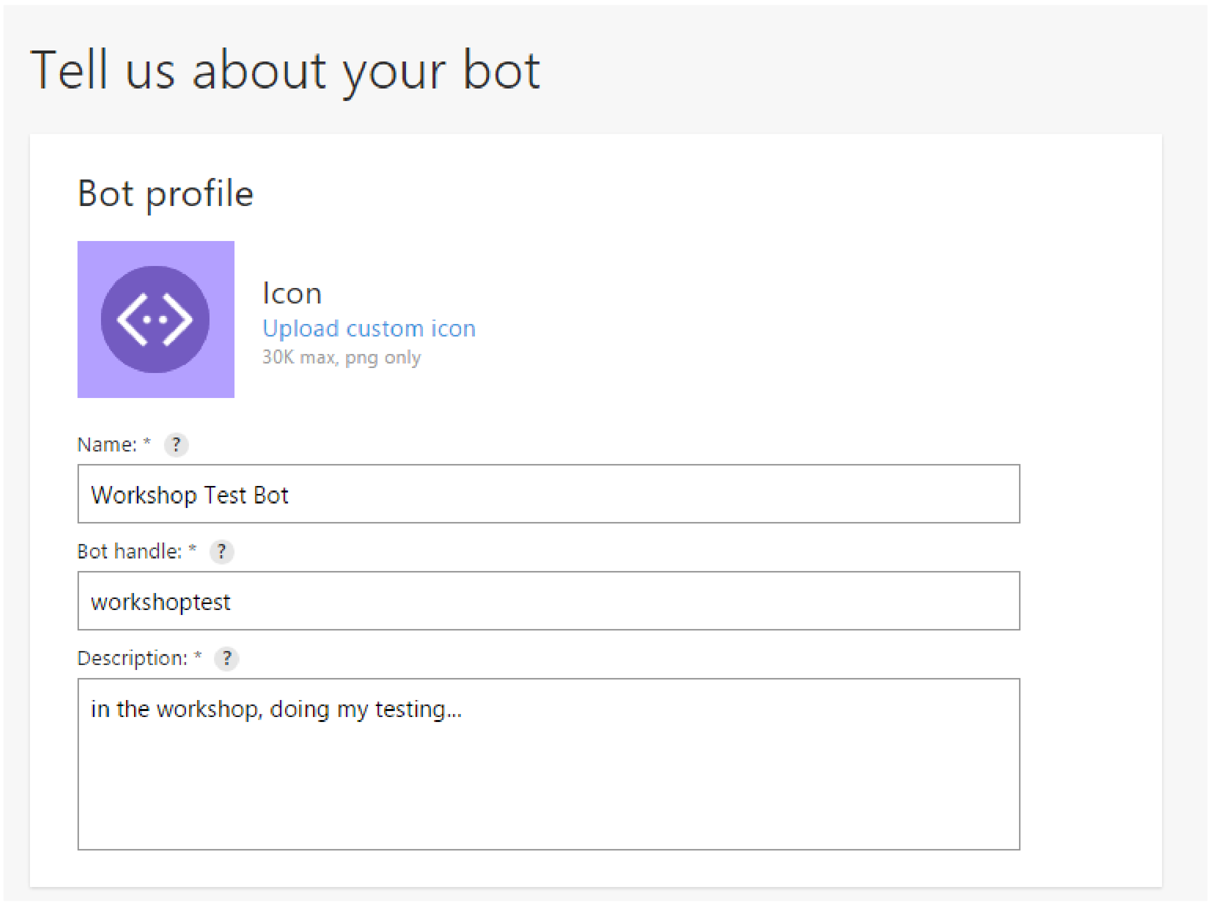


```
2>Adding file (BotApplication120170129031802\Web.config).
2>Adding ACL's for path (BotApplication120170129031802)
2>Adding ACL's for path (BotApplication120170129031802)
2>Publish Succeeded.
2>Web App was published successfully http://botapplication120170129031802.azurewebsites.net/
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
```

The API page should also automatically open, but if it does not use the link in the Output window to open it. Verify that it’s the same page you saw locally. Make a note of the URL.

2. Navigate to the Bot Framework site: dev.botframework.com and Register a bot. You may need to sign in first – use your Azure account details.

Fill in the Name, Bot handle and Description. In the Configuration section fill the Messaging endpoint with the **HTTPS version** of the URL you just published, plus the path /api/messages:


Tell us about your bot

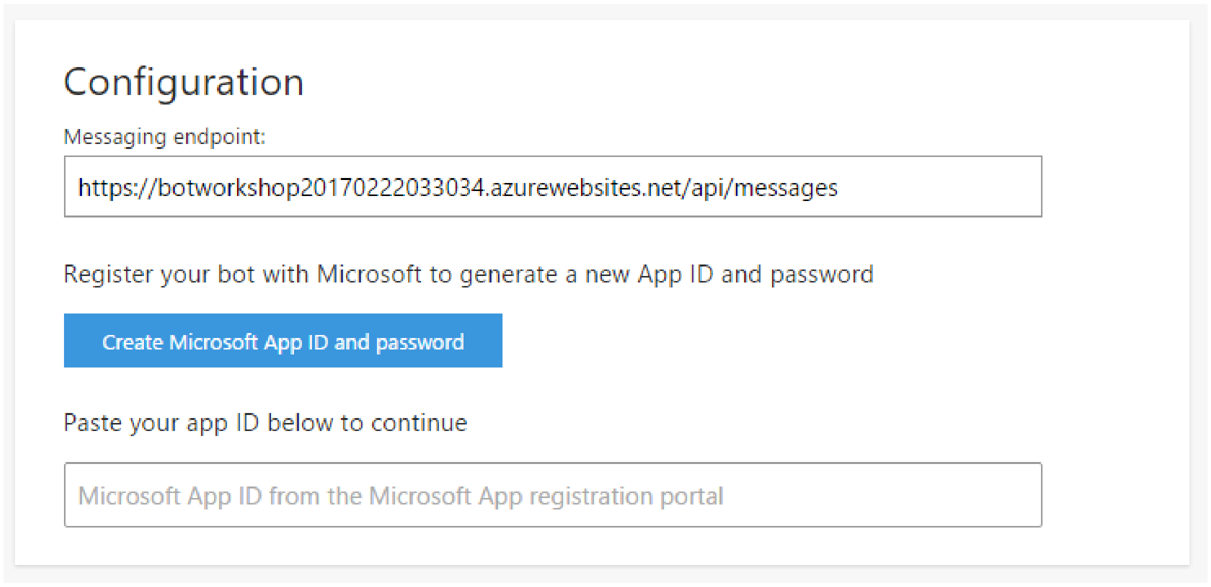
Bot profile

Icon
Upload custom icon
30K max, png only

Name: * ?
Workshop Test Bot

Bot handle: * ?
workshoptest

Description: * ?
in the workshop, doing my testing...


Configuration

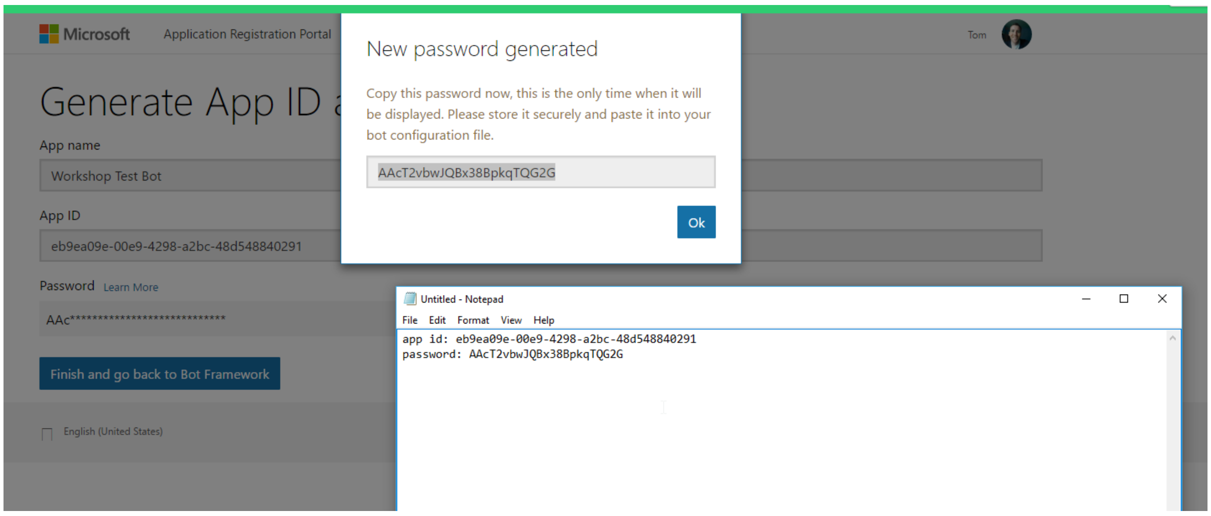
Messaging endpoint:
https://botworkshop20170222033034.azurewebsites.net/api/messages

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

Paste your app ID below to continue

Microsoft App ID from the Microsoft App registration portal

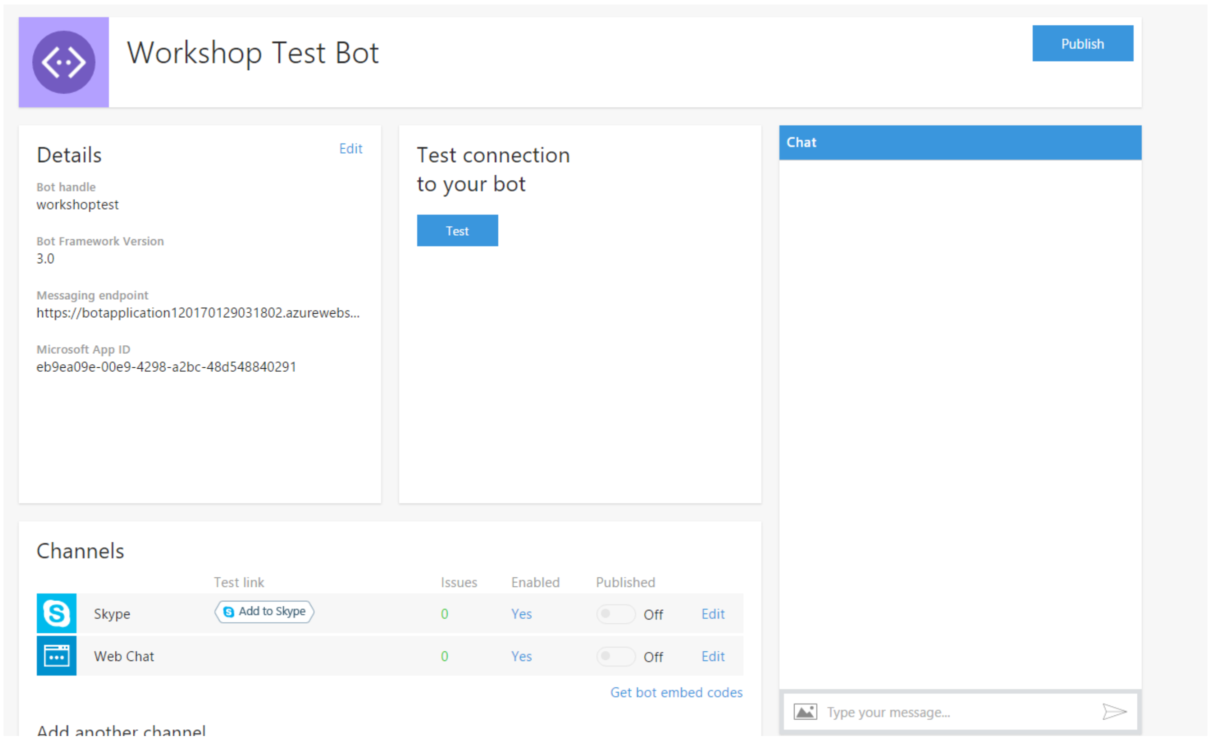
Click the **Create Microsoft App ID and password** button to generate a secure unique key for your API to talk to the Bot Framework and represent your bot. The Application Registration Portal will open in a new window, copy the App ID to a new Notepad window for later use then click the **Generate an app password to continue** button. Copy the password to the Notepad window as well:


Microsoft
Application Registration Portal
Tom
Generate App ID a
App name
Workshop Test Bot
App ID
eb9ea09e-00e9-4298-a2bc-48d548840291
Password Learn More
AAc******************************
Finish and go back to Bot Framework
English (United States)
New password generated
Copy this password now, this is the only time when it will be displayed. Please store it securely and paste it into your bot configuration file.
AAcT2vbwJQBx38BpkqTQG2G
Ok
Untitled - Notepad
File Edit Format View Help

```
app id: eb9ea09e-00e9-4298-a2bc-48d548840291
password: AAcT2vbwJQBx38BpkqTQG2G
```

Once you have a copy of both the App ID and password, click the **Finish and go back to Bot Framework** button. (you don't need to provide an Instrumentation Key)

If you're happy with the Privacy Statement, Terms of User and Code of conduct tick the checkbox at the bottom of the page, then click the **Register** button. You should see a message that your bot has been created, then you will be taken to the bot's dashboard page:



Bot handle
workshoptest

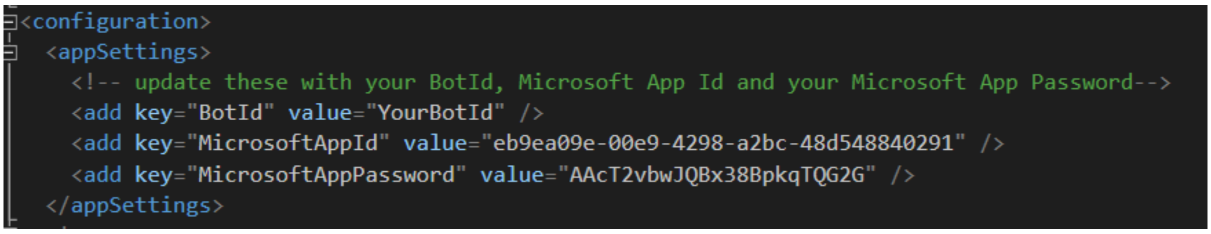
Bot Framework Version
3.0

Messaging endpoint
https://botapplication120170129031802.azurewebs...

Microsoft App ID
eb9ea09e-00e9-4298-a2bc-48d548840291

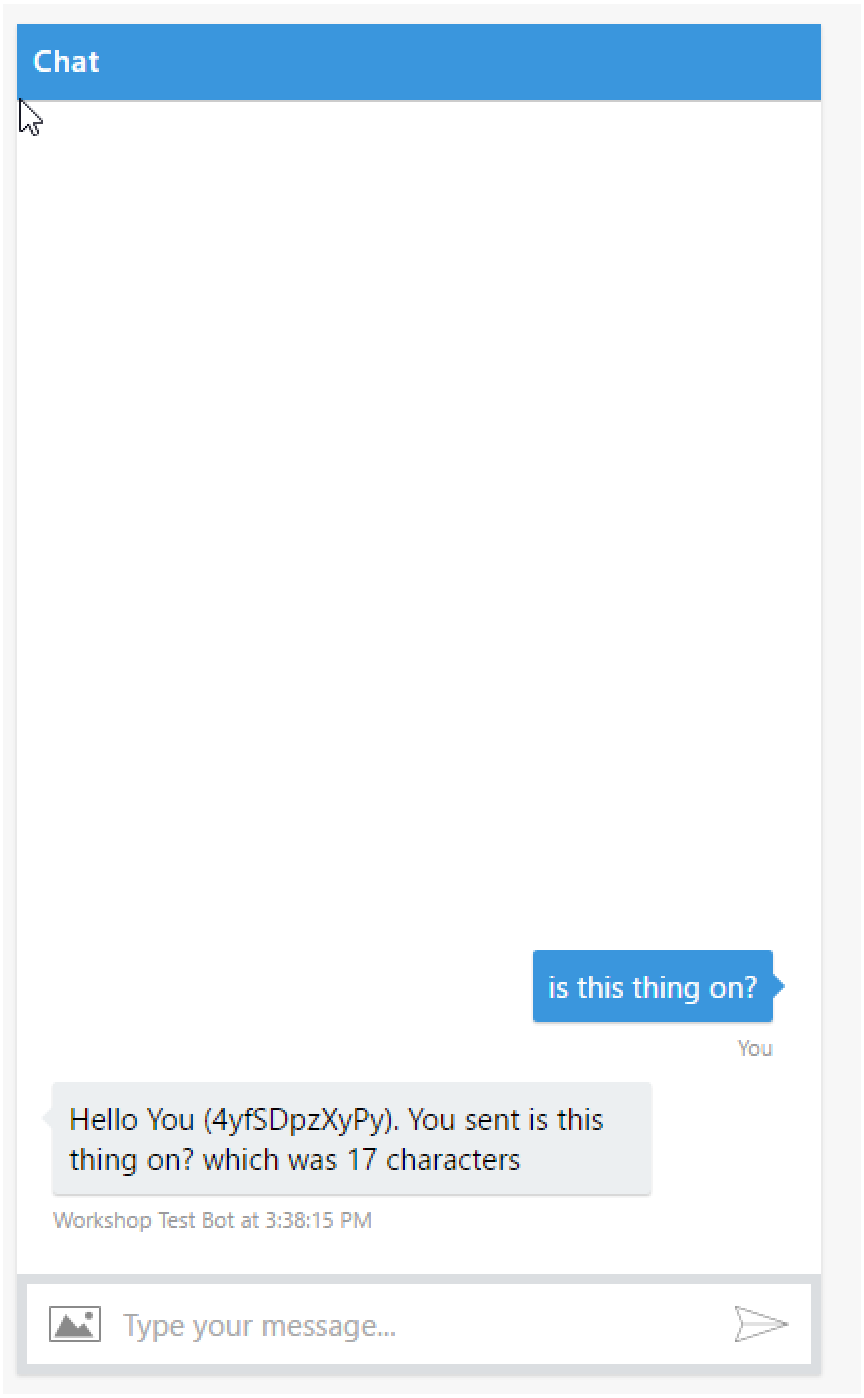
	Test link	Issues	Enabled	Published	
Skype	Add to Skype	0	Yes	Off	Edit
Web Chat		0	Yes	Off	Edit

3. Before the framework will work with our API we need to update our API with the App ID and password generated as part of the registration process. Back in Visual Studio open the web.config file. Near the top of the file are settings for BotId, MicrosoftAppId and MicrosoftAppPassword. Using the Notepad file you created earlier, paste in the App ID and Password into the bottom two keys:


```
<configuration>
  <appSettings>
    <!-- update these with your BotId, Microsoft App Id and your Microsoft App Password-->
    <add key="BotId" value="YourBotId" />
    <add key="MicrosoftAppId" value="eb9ea09e-00e9-4298-a2bc-48d548840291" />
    <add key="MicrosoftAppPassword" value="AAcT2vbwJQBx38BpkqTQG2G" />
  </appSettings>
```

Save the file, then right-click the Solution in Solution Explorer and click Publish again to republish the changes.

4: Once the publish has completed go back to your bot’s dashboard page and click the **Test** button. This tests connectivity to your API and that the API is authorised. You should see the message “Endpoint authorization succeeded”. Try chatting with your bot using the in-built messaging window on the right hand side of the dashboard:



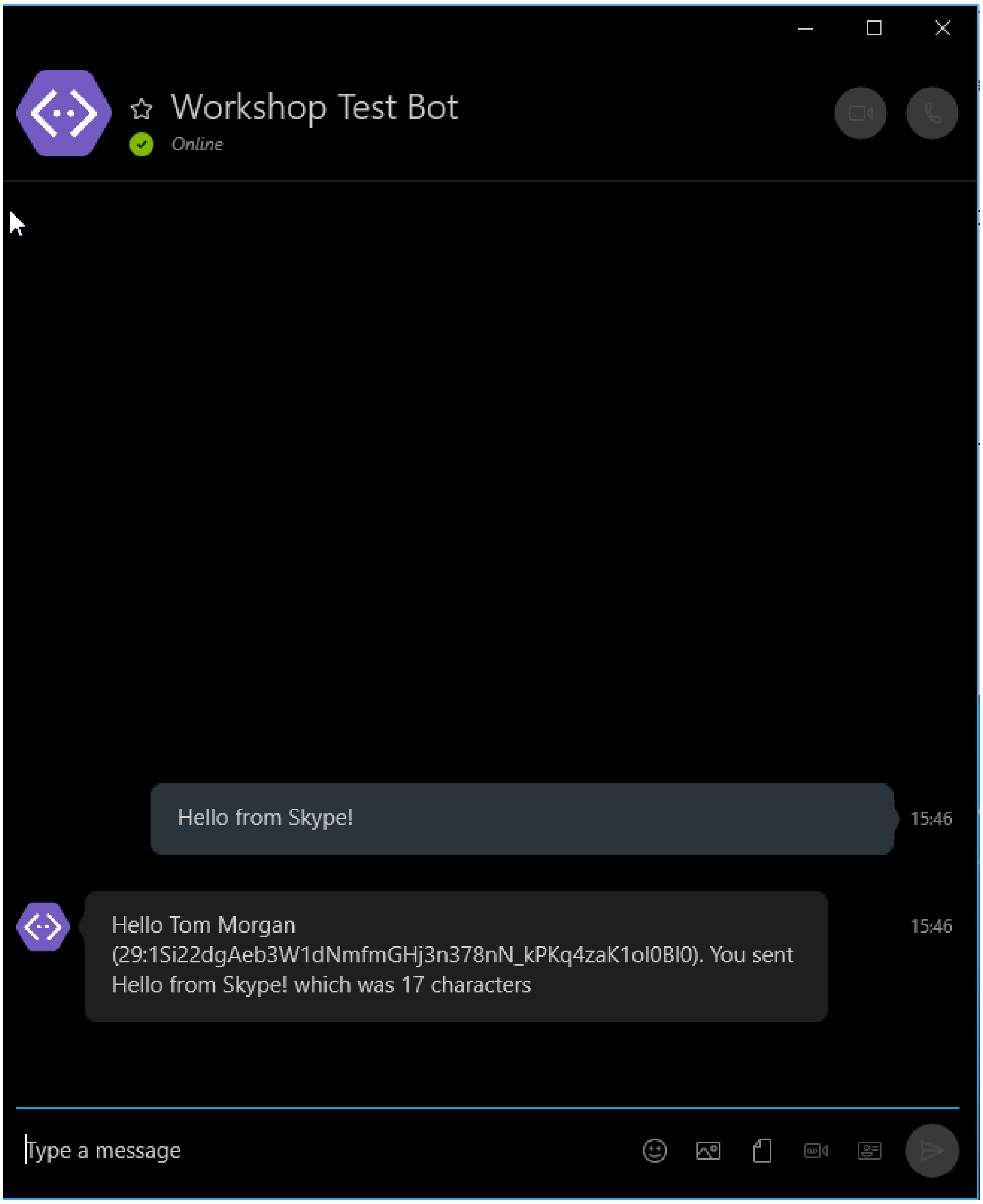
is this thing on?

You

Hello You (4yfSDpzXyPy). You sent is this thing on? which was 17 characters

Workshop Test Bot at 3:38:15 PM

5: Make sure that you have Skype installed, running and signed in. By default your bot is created with the Skype channel already setup, however it will not show up in the public Skype directory until it is published. Until then use the 'Add to Skype' test link shown in the Channels section of the Bot Dashboard to add the bot to your Skype contact list. Once the Skype page opens click the 'Add to Contacts' button. It can take a few moments for the new contact to show in the Skype contact list: if it doesn't show in the Contacts list after 2-3 minutes then sign-out and sign-in to Skype. Once your bot shows in your Contact list try sending it a message to test a fully deployed and running bot running on the Skype network!



Hello from Skype! 15:46

Hello Tom Morgan (29:1Si22dgAeb3W1dNmfmGHj3n378nN_kPKq4zaK1ol0Bl0). You sent Hello from Skype! which was 17 characters 15:46

Stop! You’re at the end of this section so please don’t go any further until asked to.

Cards

We can send more than just text. In this section, we're going to create some examples showing what other formats we can return information in. There are 3 types shown here to get you started, and then if you have time after that there are links to different types of Cards at the end of the section.

Before starting, add the following using statement to the top of the MessagesController file:

```
using System.Collections.Generic;
```

Each of the sections below replaces this line in the Post method:

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

Experiment with them and see how they look in the local emulator, or publish them and see how they look in the Skype client.

Images

```
Activity reply = activity.CreateReply();
reply.Attachments = new List<Attachment>();

reply.Attachments.Add(new Attachment()

  {

    ContentUrl = "http://baconmockup.com/300/200",

    ContentType = "image/png",

    Name = "mmmm, Bacon"

  });
```

Buttons

Buttons are a great way to control the conversation you have with your user. We can add buttons which show up in the conversation, by creating them within a ThumbnailCard or HeroCard object. First create the buttons and add them to a container:

```
CardAction button1 = new CardAction()
                {
                    Value = "I choose Blue",
                    Type = "imBack",
                    Title = "Blue"
                };

                CardAction button2 = new CardAction()
                {
                    Value = "I choose Red",
                    Type = "imBack",
                    Title = "Red"
                };

                List<CardAction> buttons = new List<CardAction>();
                buttons.Add(button1);
                buttons.Add(button2);
```

In this example, we're using the Type "imBack" which means that the value we specify is posted back to the conversation. We could hide this value whilst still receiving it by changing the type to “postBack”. There are other options as well for things like making a call, signing in, showing an image, playing a video etc.

The button collection is then used when creating a new ThumbnailCard:

```
ThumbnailCard thumbnail = new ThumbnailCard()
                {
                    Title = "Pick your favourite colour.",
                    Buttons = buttons
                };
```

This example only specifies the title – you could also specify an image, a subtitle and a specific CardAction when the user taps on the card. (if you want, try specifying these to see how they look).

Finally, the ThumbnailCard is changed to an Attachment object and added to the reply:

```
Attachment attachment = thumbnail.ToAttachment();
Activity reply = activity.CreateReply();
reply.Attachments = new List<Attachment>();
reply.Attachments.Add(attachment);
```

Receipts

There is a special object used to display receipts to users.

Firstly, add the line items:

```
ReceiptItem lineItem1 = new ReceiptItem()
{
  Title = "Bacon 1",
  Image = new CardImage(url: " http://baconmockup.com/300/300"),
  Price = "9.99",
  Quantity = "1"
};

ReceiptItem lineItem2 = new ReceiptItem()
{
  Title = "Bacon 2",
  Image = new CardImage(url: " http://baconmockup.com/400/400"),
  Price = "15.00",
  Quantity = "2"
};

ReceiptItem lineItem3 = new ReceiptItem()
{
  Title = "Bacon 3",
  Image = new CardImage(url: " http://baconmockup.com/500/500"),
  Price = "1.23",
  Quantity = "1"
};
```

Then, create the ReceiptCard object and add the line items to it:

```
var receipt = new ReceiptCard()
                {
                    Title = "Thanks for shopping at the NorDevCon Bacon Shop",
                    Total = "26.22",
                    Tax = "5.24"
                };

receipt.Items = new List<ReceiptItem>();
                receipt.Items.Add(lineItem1);
                receipt.Items.Add(lineItem2);
receipt.Items.Add(lineItem3);
```

Finally, the ReceiptCard object is changes to an Attachment object and added to the reply

```
Attachment attachment = receipt.ToAttachment();
Activity reply = activity.CreateReply();
reply.Attachments = new List<Attachment>();
reply.Attachments.Add(attachment);
```

See how the receipt renders in the Emulator, and in the Skype client. Notice how the images are automatically resized.

You can optionally add additional information to the receipt by adding Facts to the ReceiptCard. These are key value pairs which will display string values. For instance:

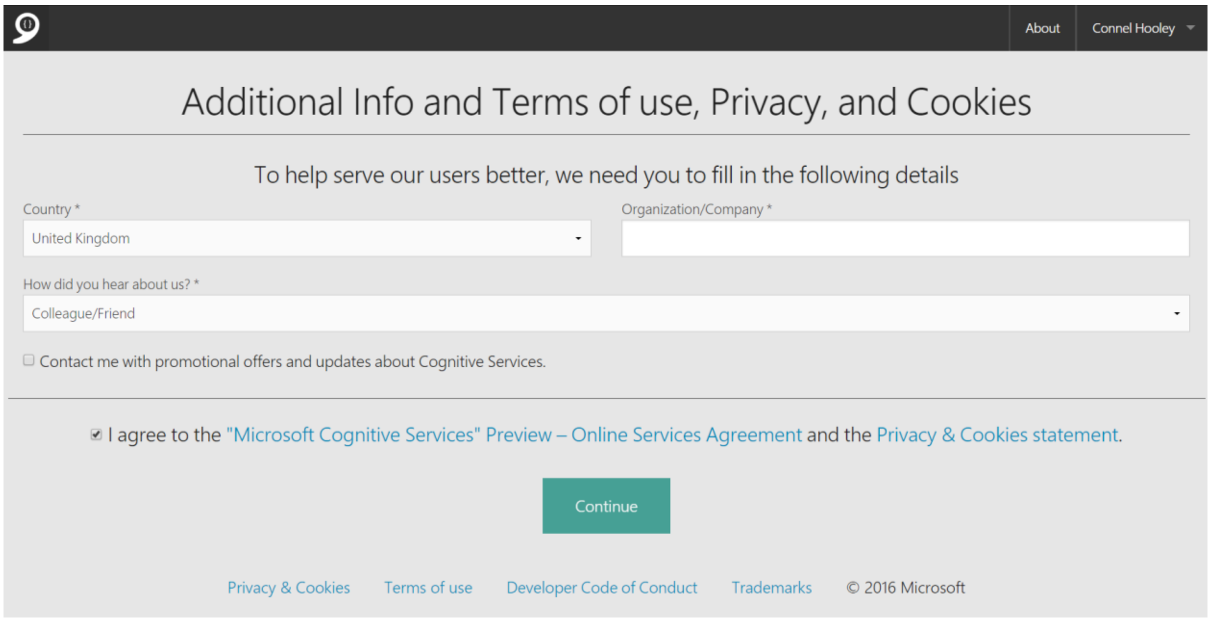
```
var fact = new Fact()
{
  Key = "Order number",
  Value = "12345"
};
receipt.Facts = new List<Fact>();
receipt.Facts.Add(fact);
```

There are numerous other ways to display additional information. Have a look at the Channel Inspector (http://bit.ly/ChInsptr) for a good list of features and how they are rendered in different channels. If you have time, experiment with a different feature (follow the link to Bot Framework Documentation for that Feature).

Stop! You're at the end of this section so please don't go any further until asked to.

Adding AI

1: Navigate to https://www.luis.ai and Sign In with your Azure account. You may need to accept the Terms of Service:



Additional Info and Terms of use, Privacy, and Cookies

To help serve our users better, we need you to fill in the following details

Country * United Kingdom

Organization/Company *

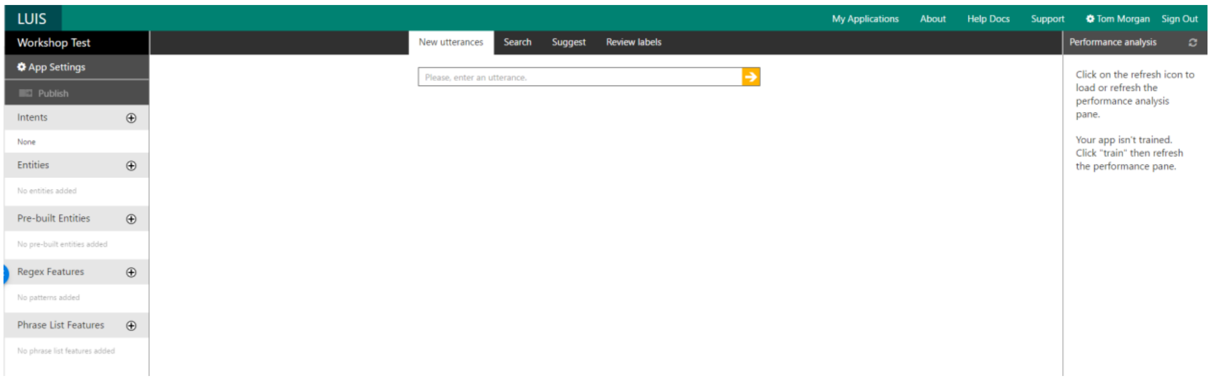
How did you hear about us? * Colleague/Friend

☐ Contact me with promotional offers and updates about Cognitive Services.

☑ I agree to the "Microsoft Cognitive Services" Preview – Online Services Agreement and the Privacy & Cookies statement.

© 2016 Microsoft

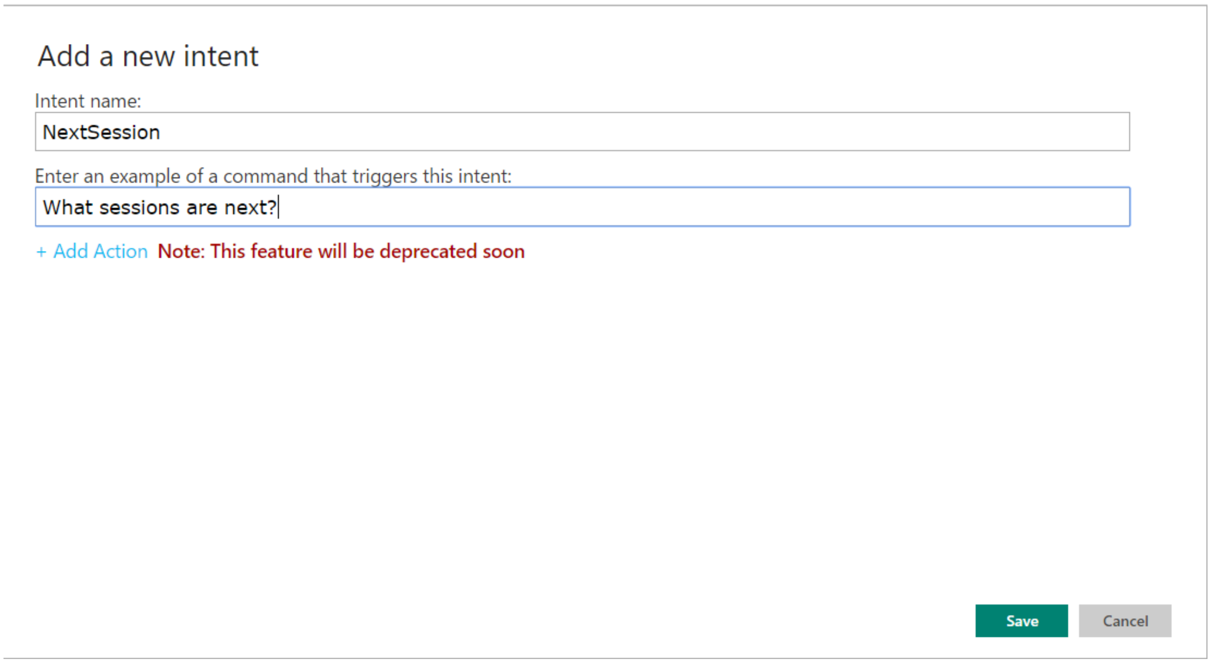
If you're prompted to, cancel the tutorial. Click the **New App** button to create a new Application. Choose an application name, domain and description and click **Add App**. Once the application is created you should see the LUIS dashboard:



Click on the refresh icon to load or refresh the performance analysis pane.

Your app isn't trained. Click "train" then refresh the performance pane.

2: The first intent we're going to create is for which sessions are next. On the left-hand side click the plus button next to the title Intents. Specify NextSession as the intent and provide an example phrase where it would be appropriate to use this intent:


Add a new intent

Intent name:

NextSession

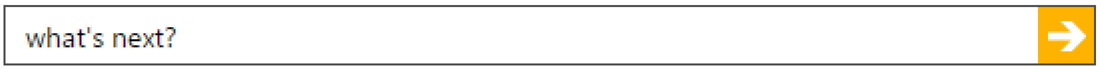
Enter an example of a command that triggers this intent:

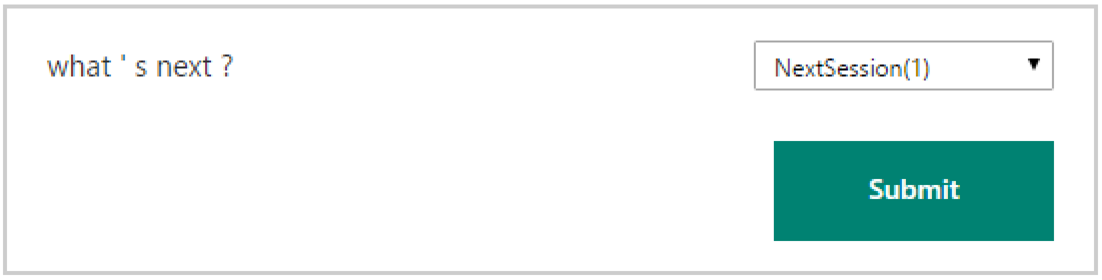
What sessions are next?

Click Save. The phrase is added to the 'New utterances' section and tagged with the NextSession intent. Click Submit to add this linking of the phase to the intent.

Let's add another utterance. In the input box at the top of the page, add "Tell me what sessions are next." and click the yellow arrow. Use the drop down to select NextSession and click Submit.

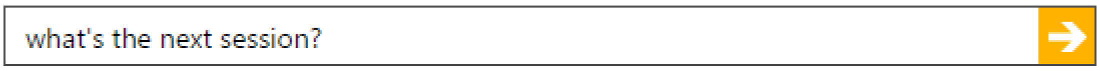
Now that we've added two intents click the 'Train' button at the bottom left of the page. Once the training is completed, try adding the phrase "what's next?" to the search box:


what's next?

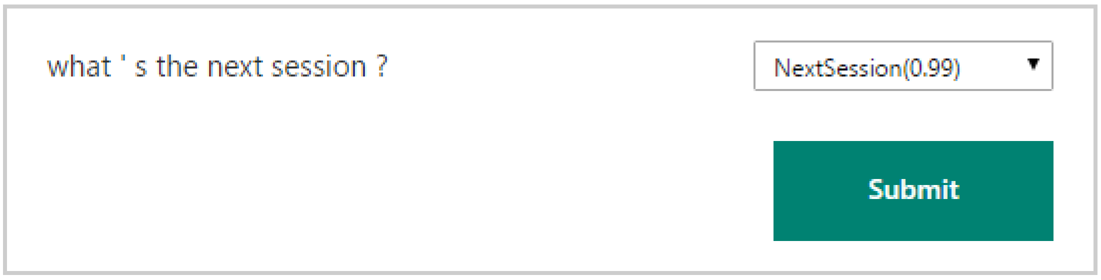

what ' s next ?

NextSession(1)

Notice how LUIS has correctly chosen the NextSession intent for this phrase. The certainty percentage is shown in brackets after the intent (in this case 100%). To confirm this, click the Submit button. Do this again for the phrase "what's the next session?":



what's the next session?



what ' s the next session ?

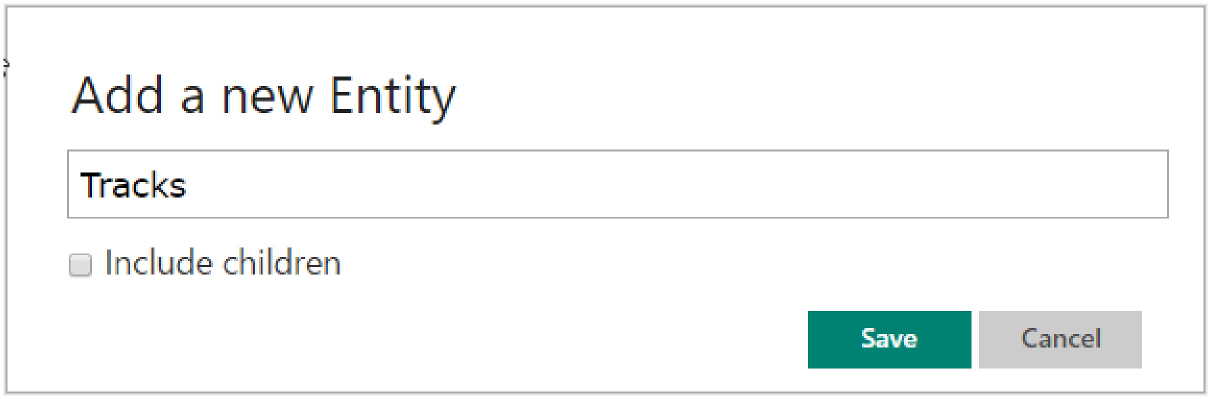
NextSession(0.99)

By adding extra phrases and confirming the correct intent you are reinforcing and improving the phrase matching capabilities of your AI application.

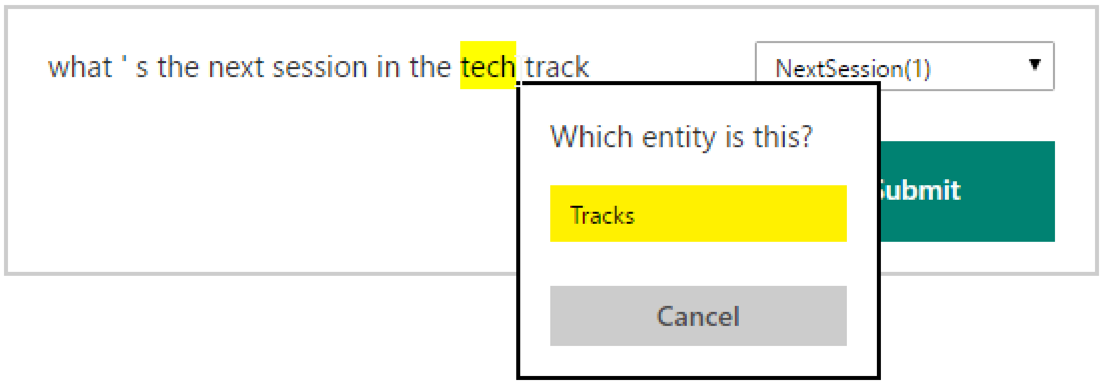
Re-train the model by click the 'Train' button again.

3: Enter the phrase "what's the next session in the tech track" and click the yellow arrow. Although LUIS knows which intent is appropriate and correctly selects it there is additional information in the phrase. The 'tech track' part of the phrase is additional information we can use to refine what information we provide.

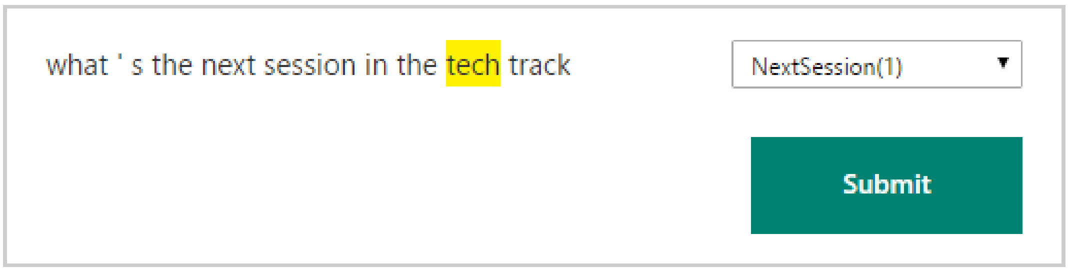
Click the plus sign next to the Entities title. Specify 'Tracks' as the entity name and click Save:



If it's not still on the screen, go back and type in the phrase "what's the next session in the tech track" again. This time, select the word 'tech' in the phrase with the mouse (as if you were highlighting it). As you release the mouse button the 'Which entity is this?' box pops up. Select 'Tracks':



Submit the phrase with the entity selected:



Repeat this for the following tracks: mobile, agile, business, workshop. Remember to choose the Intent each time – if you make a mistake just re-enter the exact phrase and then change it. You can also use the Review Labels link.

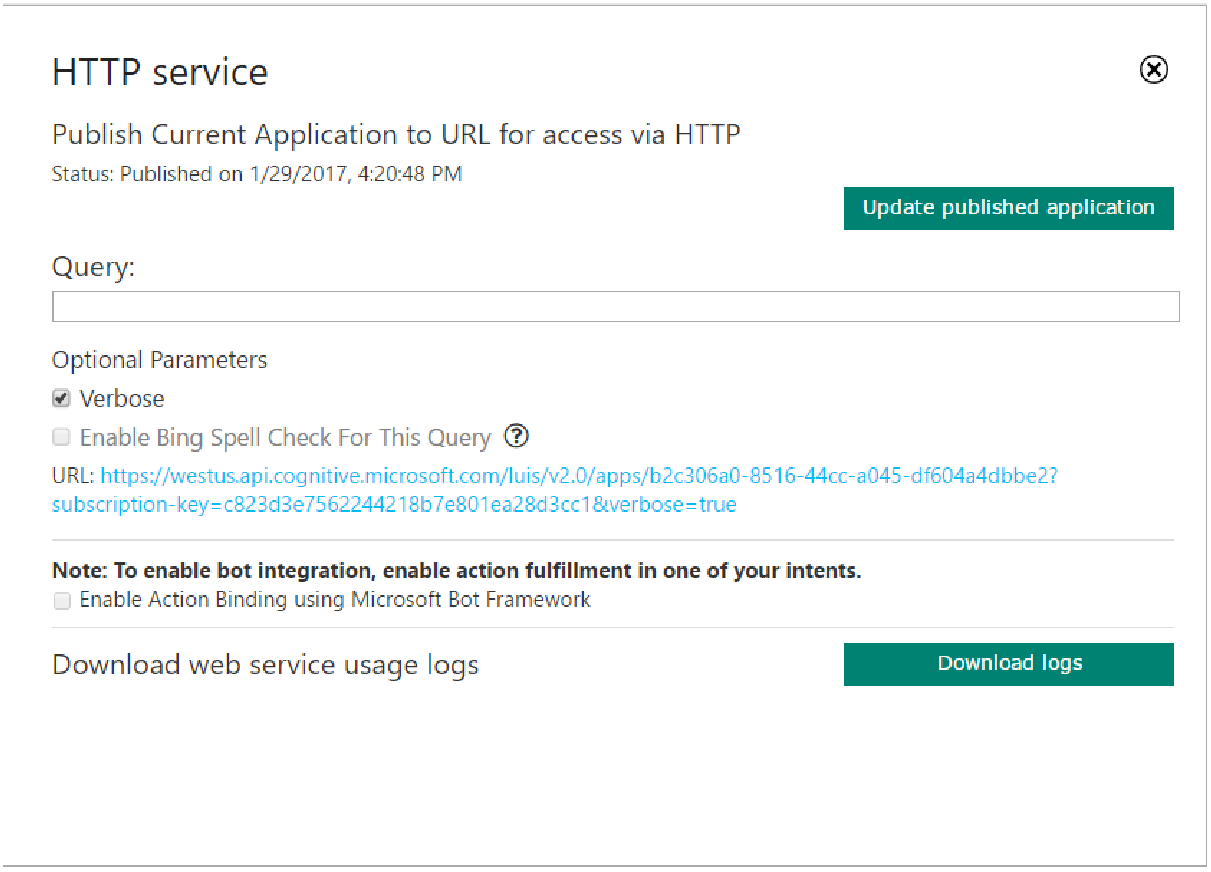
Click ‘Train’ to re-train your model.

Experiment with different phrases and see how well (or otherwise) LUIS reacts to them. Submit extra phrases to improve how well the AI correctly identifies other phrases which can be answered with the NextSession intent.

Stop! You’re at the end of this section so please don’t go any further until asked to.

Testing the APIs

1: In the LUIS dashboard make sure your bot has recently been trained then in the top left hand corner, click Publish. Ignore the checkbox about bot integration and click **Publish web service**. If publishing succeeds then you should see the unique link shown:


HTTP service

Publish Current Application to URL for access via HTTP

Status: Published on 1/29/2017, 4:20:48 PM

Query:

Optional Parameters

☑ Verbose

☐ Enable Bing Spell Check For This Query

URL: https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/b2c306a0-8516-44cc-a045-df604a4dbbe2?subscription-key=c823d3e7562244218b7e801ea28d3cc1&verbose=true

Note: To enable bot integration, enable action fulfillment in one of your intents.

☐ Enable Action Binding using Microsoft Bot Framework

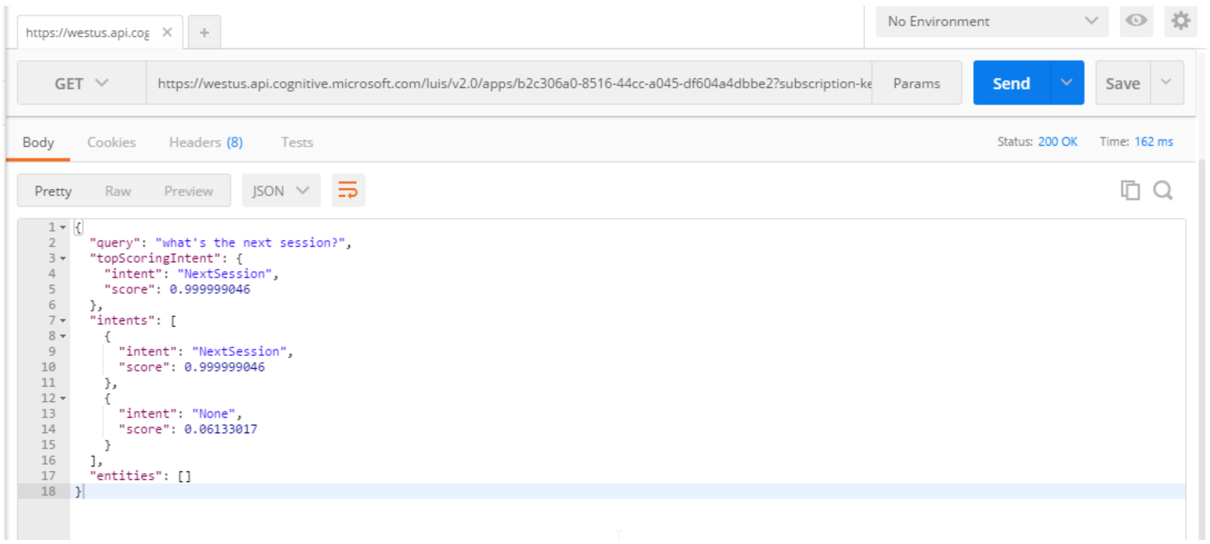
Download web service usage logs

The API works by passing the phrase to check as a parameter along the query line. To see this in action, type in a query in the Query text box and see how the URL changes.

Make a copy of the URL with an example phrase:

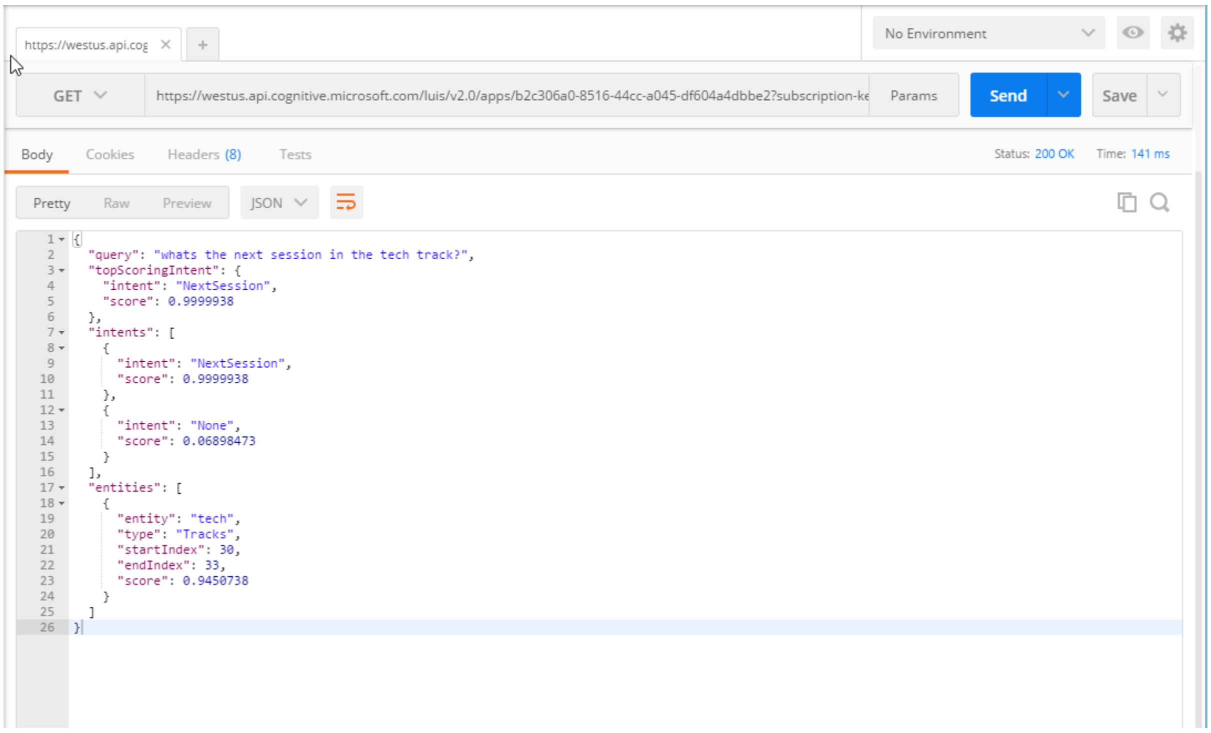
URL: https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/b2c306a0-8516-44cc-a045-df604a4dbbe2?subscription-key=c823d3e7562244218b7e801ea28d3cc1&q=what's%20the%20next%20session%3F&verbose=true

Using Fiddler, Postman or equivalent perform a GET to this URL and look at the JSON returned. You can see the best matched intent and any entities. In this example there aren't any entities:



```
{
  "query": "what's the next session?",
  "topScoringIntent": {
    "intent": "NextSession",
    "score": 0.999999046
  },
  "intents": [
    {
      "intent": "NextSession",
      "score": 0.999999046
    },
    {
      "intent": "None",
      "score": 0.06133017
    }
  ],
  "entities": []
}
```

Try a phrase that contains a known entity, such as 'what's the next session in the tech track' to see how entities are shown.



```
{
  "query": "whats the next session in the tech track?",
  "topScoringIntent": {
    "intent": "NextSession",
    "score": 0.9999938
  },
  "intents": [
    {
      "intent": "NextSession",
      "score": 0.9999938
    },
    {
      "intent": "None",
      "score": 0.06898473
    }
  ],
  "entities": [
    {
      "entity": "tech",
      "type": "Tracks",
      "startIndex": 30,
      "endIndex": 33,
      "score": 0.9450738
    }
  ]
}
```

Once you're happy with these calls you can optionally remove the verbose=true URL parameter – this removes the full list of Intents and their score whilst retaining the topScoringIntent (which is probably the only one you are interested in.

Sessions API

There is also a HTTP API we can call for information about what the next sessions actually are. Because of the timing of this workshop it's actually better to use yesterday's schedule rather than today's as there aren't any more track-specific sessions after this one! You can use the following URLs to retrieve information about either today or yesterday:

http://nordevconwebapi.azurewebsites.net/api/onnext - get next sessions (all tracks) http://nordevconwebapi.azurewebsites.net/api/onnext?day=1&time=0830 – get next sessions for Day 1 (yesterday) at a specific time.

http://nordevconwebapi.azurewebsites.net/api/onnext?track=tech – get next session for a specific track

http://nordevconwebapi.azurewebsites.net/api/onnext?day=1&time=0830&track=tech – get next session for Day 1 (yesterday) at a specific time, for a specific track.

Data is returned in JSON format:

```
[
 {
  "Title": "If You’re Happy And You Know It (Inside the mind of a developer)",
  "Author": "Dom Davis",
  "StartTime": "2017-02-24T10:45:00",
  "LengthMins": 45,
  "Location": "Norfolk County Council Auditorium",
  "Track": "Tech"
 }
]
```

Stop! You’re at the end of this section so please don’t go any further until asked to.

Adding AI to your Bot

This section is much less prescriptive as now you know everything you need to add AI into your bot and have it return useful information. Try and complete this section yourself, or if you are running out of time or want a helping hand, there is a sample solution in the GitHub repo.

We now have a number of useful moving parts. We have a working bot which can read user messages and reply to them. We have a HTTP service we can call to identify if a phrase contains a specific intent. We have another HTTP service we can call to retrieve information about sessions.

Task: Join all these parts together so that if a Skype user uses your bot to ask “what’s the next session in the tech track” then they are provided with information about the track. To do this you’ll need to send the phrase the user typed to LUIS using the API endpoint and read the intents. If the intent is NextSession then you’ll need to call the session API to find out what the sessions are and then format the JSON into a nice text format. Once that’s done, go back and examine the entities to see if there are any specific tracks the user was asking about, and if so tailor the API call and the message back to the user. Remember to handle the use case when the intent returned from LUIS is ‘None’ and return a message to the user that you don’t know what they mean, and consider what happens if a user were to ask “what are the next sessions in the tech and agile tracks”.

Interesting Info: Once you’ve sent a number of different phrases to LUIS you might want to go back and do additional training. LUIS will remember every single phrase that was sent to it via the API and you can review these, see the decisions LUIS made, and change them to improve the AI learning. You can do this either by click ***Suggest*** *from the LUIS Dashboard menu to see phrases which can be better matched, or* ***Review labels*** *to see all phrases every sent.*