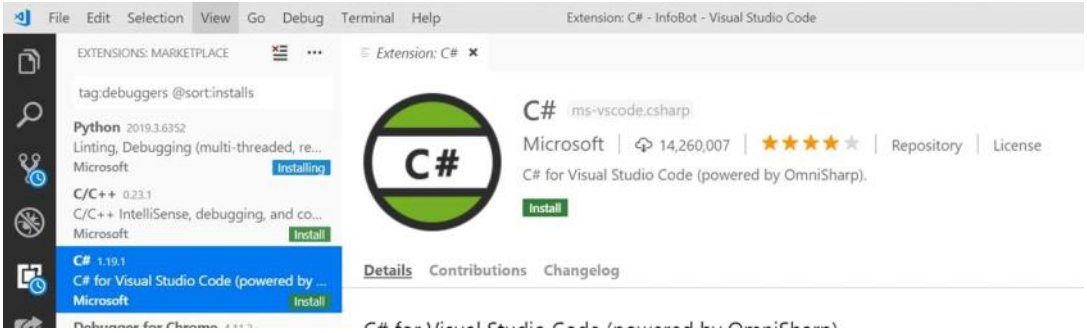
Bots intro workshop

jueves, 6 de junio de 2019 11:56

Requirements

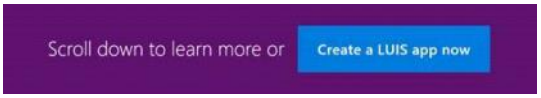
1. Visual Studio, you may use VS 2017 or 2019.
   If you were to use Visual Studio Code, you will need to download the templates instead of performing step 2. In this case, install this extension for C# for Visual Studio Code:



2. (Only VS 2017 or 2019) Intall the Bot Framework SDK templates extension for VS: https://marketplace.visualstudio.com/items?itemName=BotBuilder.botbuilderv4

3. Install the bot emulator: https://github.com/Microsoft/BotFramework-Emulator/releases.

4. Go to https://eu.luis.ai and create a new account. You can register with a Microsoft ID (@outlook, @hotmail), select country and click on "Create a LUIS app now":



   The next steps will be covered during the workshop.

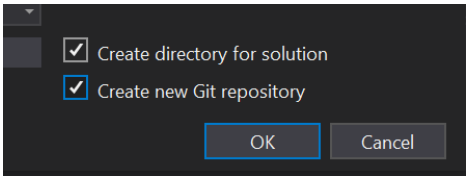
Optional requirements

5. (optional) Azure subscription with permissions to create resources. This will be required to publish a new bot on the cloud in Azure. If not all attendees have access to this, they can either create a new free Azure trial or can just skip the continuous deployment and Azure portal specific steps and focus on testing the bot on their local machine using the emulator.

6. (optional) Github extension for Visual Studio: https://visualstudio.github.com

Part 1 - Create a simple bot locally

1. a. (if using Visual Studio Code). Download the bot template and open its folder in Visual Studio.
   b. (if using Visual Studio 2017 / 2019) Open Visual Studio > File > New Project. If you installed the Bot SDK templates (requirement n#2) you should be able to select "Empty Bot" under Visual C# > Bot Framework > Empty Bot (Bot Framework v4). If you intend to enable source control with a GitHub repo and continuous deployment, be sure to enable this checkbox to create a repo for this project in GitHub:



   Call it for example "HotelBot".

2. Let's take a look at the code, open the HotelBot.cs file. We can see that we have a class EmptyBot derived from ActivityHandler. ActivityHandler methods can be overriden to handle activities.

The base handler is the turn handler (OnTurnAsync method), all incoming activities will be routed through the turn handler and it will then call the individual activity handler according to the activity type. For example, if a message is received, the turn handler (OnTurnAsync) will first process it and will then call the message handler (OnMessageActivityAsyncHandler).
Depending on the types of activities we want our bot to be able to process, we will want to override one or more of these methods:

Event	Handler	Description
Any activity type received	OnTurnAsync	Calls one of the other handlers, based on the type of activity received.
Message activity received	OnMessageActivityAsync	Override this to handle a Message activity.
Conversation update activity received	OnConversationUpdateActivityAsync	On a ConversationUpdate activity, calls a handler if members other than the bot joined or left the conversation.
Non-bot members joined the conversation	OnMembersAddedAsync	Override this to handle members joining a conversation.
Non-bot members left the conversation	OnMembersRemovedAsync	Override this to handle members leaving a conversation.
Event activity received	OnEventActivityAsync	On an Event activity, calls a handler specific to the event type.
Token-response event activity received	OnTokenResponseEventAsync	Override this to handle token response events.
Non-token-response event activity received	OnEventAsync	Override this to handle other types of events.
Other activity type received	OnUnrecognizedActivityTypeAsync	Override this to handle any activity type otherwise unhandled.

3. In our code we can see the OnMembersAddedAsync method. As you can see in the table above, this method is called when new members join the conversation. This method receives several parameters:

membersAdded	list of members added to the conversation
turnContext	when the adapter receives an activity, which corresponds to the inbound HTTP request, it generates a turn context object, which contains information about the incoming activity, the sender and receiver, the channel, the conversation, and other data needed to process the activity.
cancellationToken	propagates notification that operations should be canceled.

Right now what the method does, is go through the new members (when we first run the bot, the only new member to appear here will be the user) and if it is not equal to the receiver (in this case the bot) then it will return a welcome message.
Change the code to customize the welcome message:

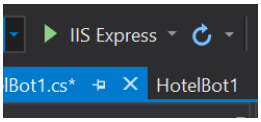
```
await turnContext.SendActivityAsync(MessageFactory.Text($"Hello, I can help you to book an appointment for the SPA, dinner or give you information about the city. How can I help you?"), cancellationToken);
```

Remember that the turnContext parameter contains all information about the activity, and is also used to send back an activity to the user, like we are doing in the above code.

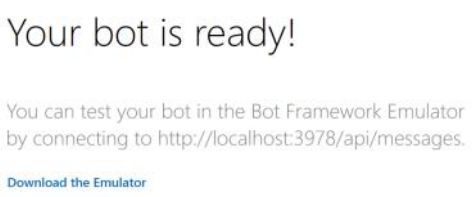
4. Set a breakpoint on that line of code, just double-right click to add it:

Part 2 - Run your bot using the emulator and debug

1. Click F5 (IIS Express)

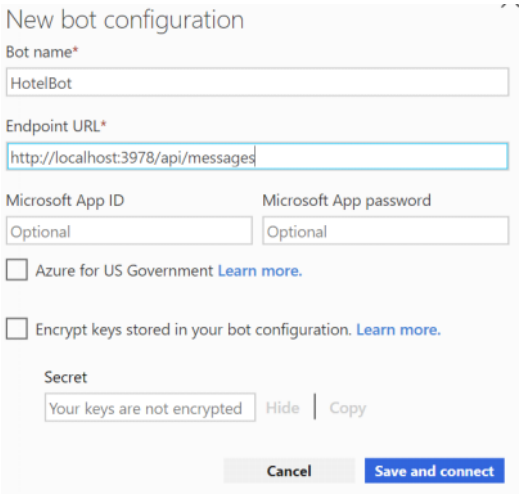


you will see something like this:

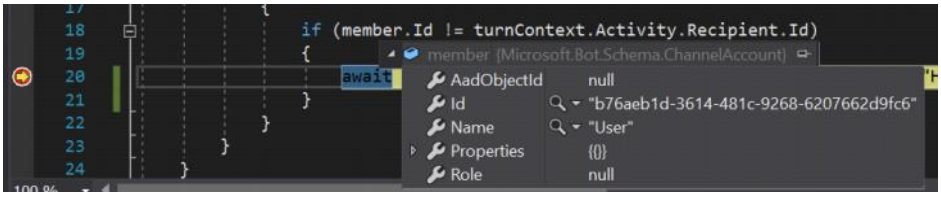


Copy the endpoint URL.

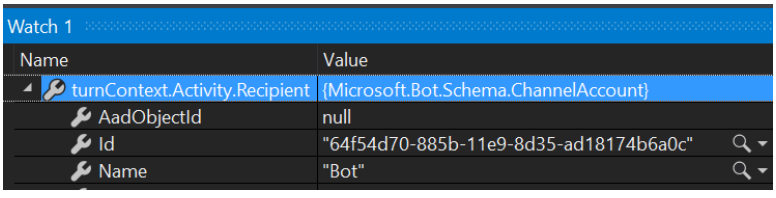
2. Open the Bot Emulator > File > New Configuration, call it "HotelBot" and enter the previous endpoint URL. For this we will not use encryption for secrets or specify API key, but remember that normally you would do so. Click Save and Connect:



3. Now in the emulator you will receive the message we added when new users join the conversation, and the breakpoint will be hit, so go back to Visual Studio to inspect the code.
4. If you inspect the member variable for example, you will see that it corresponds to the user:

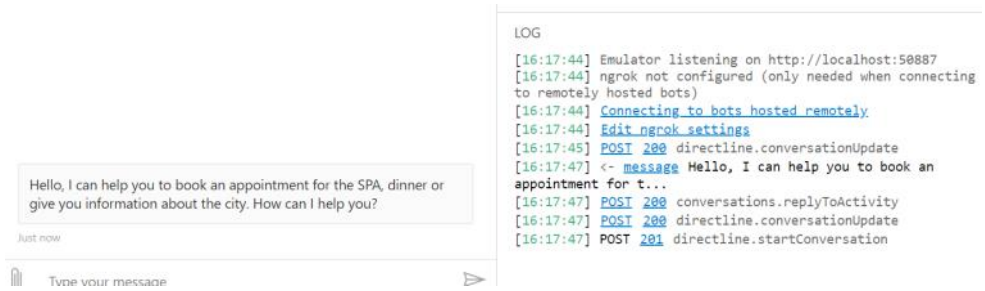


5. And the receiver member of the turncontext parameter is the bot:



6. From now on if you want to debug your bot you can just set breakpoints in your code to inspect what you need.
7. Stop your bot in Visual Studio.

8. Another place where you can check what is happening when the bot receives or send messages is in the emulator itself. As activities happen on the bot, you will see on the right a log of events:



Just click on one of them to inspect:

Part 3 - Adding buttons

1. Let's add some buttons so the user can select what they want to do. For this, create a method called SendSuggestedActionsAsync to the EmptyBot class:

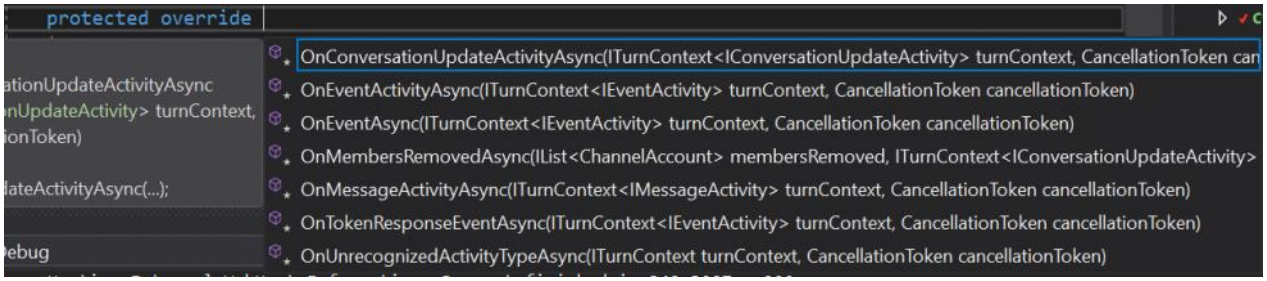
```
private static async Task SendSuggestedActionsAsync(ITurnContext turnContext, CancellationToken cancellationToken)
{
    var reply = turnContext.Activity.CreateReply("Hello, what can I help you with today?");
    reply.SuggestedActions = new SuggestedActions()
    {
        Actions = new List<CardAction>()
        {
            new CardAction() { Title = "SPA", Type = ActionTypes.ImBack, Value = "SPA" },
            new CardAction() { Title = "Restaurant", Type = ActionTypes.ImBack, Value = "Restaurant" },
            new CardAction() { Title = "Tourist information", Type = ActionTypes.ImBack, Value = "Tourist
information" }
        }
    };

    await turnContext.SendActivityAsync(reply, cancellationToken);
}
```

2. Change the OnMessageActivityAsync method to call your buttons method instead:

```
if (member.Id != turnContext.Activity.Recipient.Id)
{
        await SendSuggestedActionsAsync(turnContext, cancellationToken);
}
```

3. When the user selects a button, we will receive a new activity of type Message. If we want to process the user's answer, we will need to overrise the message handler (OnMessageActivityAsync). Type protected override and you will see the list of methods from ActivityHandler which we can override, including the OnMessageActivityAsync one:

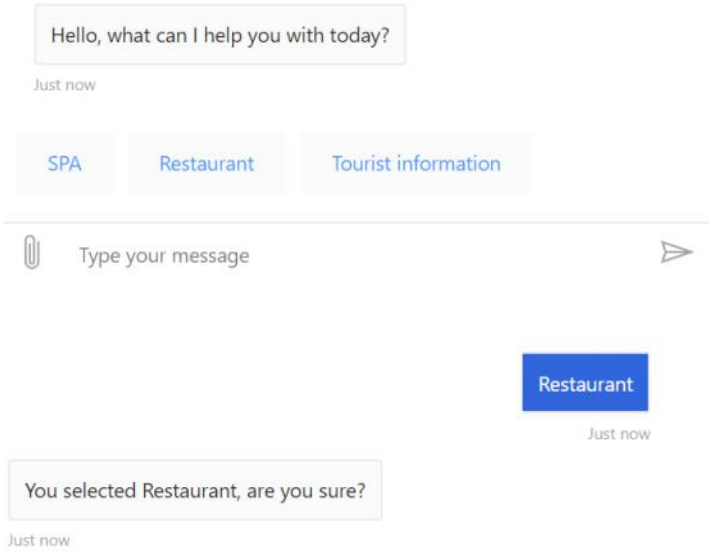


Select it, add the async modifier to the method and enter this code:

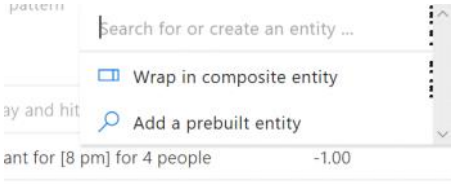
```
protected override async Task OnMessageActivityAsync(ITurnContext<IMessageActivity> turnContext, CancellationToken
cancellationToken)
{
    await turnContext.SendActivityAsync(MessageFactory.Text($"You selected {turnContext.Activity.Text}, are you
sure?"));
}
```

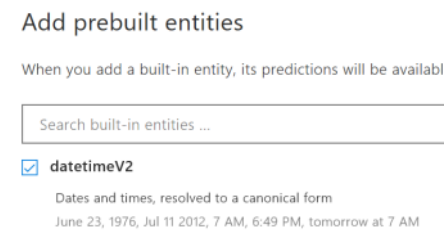
Here what we do is take the value which the user selected (turnContext.Activity.Text) and inform the user about their selection by sending back an activity with that information.

4. Now run (F5), go back to the emulator and test the buttons:

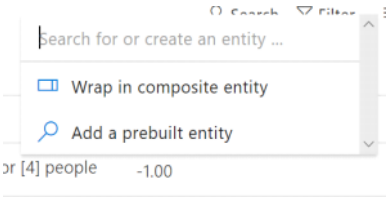


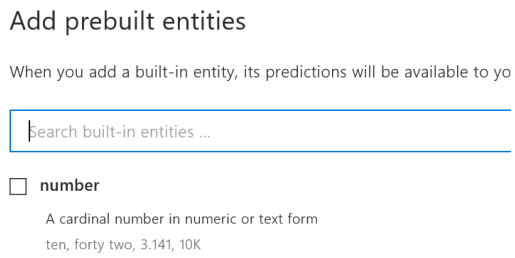
Part 4 - Set up Language understanding for your bot

1. It is fine asking the user what they want to do using buttons as before, but perhaps we would prefer instead to let the user ask freely and detect what their intention is. For this we will be using LUIS.
2. Go to https://eu.luis.ai and sign in.
3. Create new app, call it "HotelLUIS" and give it a description.
4. Click on Build > Intents > Create New Intent > "BookRestaurant"
5. Enter utterance "I would like to book a table at the restaurant for 8 PM for 4 people" and click Enter.
6. Left click over "8" and left click over "PM", it shoud now appear as "[8 PM]". Left click on that and select "Add a prebuilt entity" > select "datetimeV2":
7. 

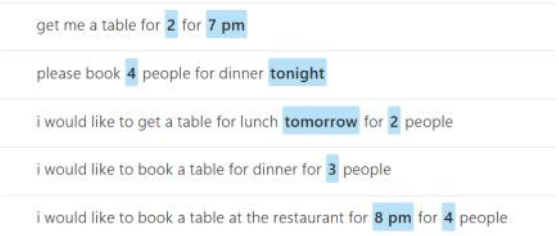


8. Left click over "4" > Add a Prebuilt entity > select Number:

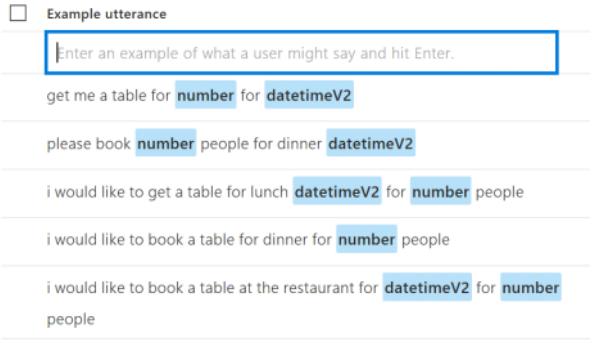




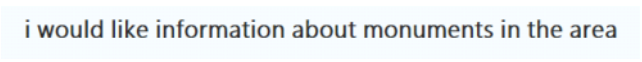
9. Add more utterances as examples of different ways in which a user might ask to book a table at the restaurant. For example:



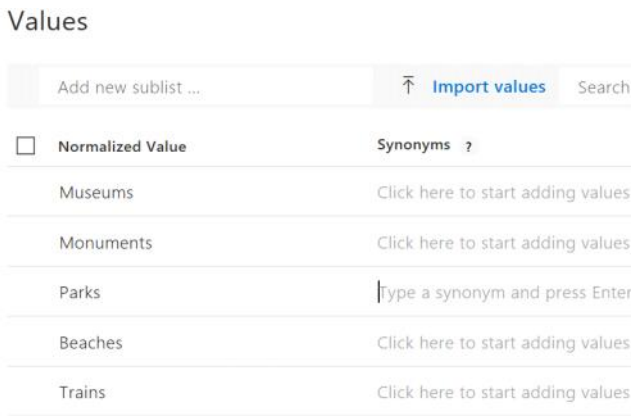
It should now automatically detect the date and number entities, otherwise you can retag any that are incorrect:



10. Click on Create New Intent to add another one called "TouristInfo". Add a utterance such as:

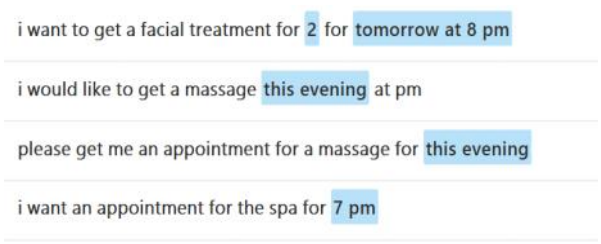


11. Go to Entities > Create New Entity > call it "TouristAttractions" of type List and add the following values:

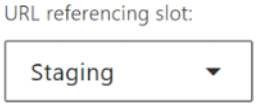


12. If you want, you can add synonims for each value, so that if users say for example "gardens" the bot will know they are asking for information about Parks.
13. Add more utterances, for example:

14. Create one last intent to book appointments at the spa. Click Create New Intent > "BookSPA", and add utterances in the same way:



15. If you are building a bot for a SPA you would probably want to detect exactly what treatment (massage, facial treatment, sauna, etc) the customer wants. For this, you could a new custom entity to detect this, but for now we won't do this just to keep things simple.
16. When done, click on Train.
17. Once training is finished, click on Test and start testing. For example say "I want a table for 7 for July 7th" > Inspect > see with what percentage it detected the BookRestaurant intent and if it detected any entities.
18. If tests are not returning the expected results, add more utterances for each intent, train and test again.
19. Explore the different things we can do in LUIS, for example:
    1. under intents check out the "Add prebuilt domain intent". We will not use them in this workshop, but it is good to know that they are available for a variaty of common domains were you to need them.
    2. under Manage > Publish settings > you can enable sentiment analysis to detect if the user's utterance is positive, neutral or negative.
    3. under Manage > Versions > you can see that you can import a different version of the LUIS app, so you can for example import a LUIS app which already includes intents and entities created by another person.
20. Go to Manage > Application ID, write it down.
21. Click on Publish > Staging. When done publishing, go to Manage > Keys & Endpoints and write down the Authoring Key and endpoint URL, making sure you selected the staging one:



Part 5 - Connect your bot to your LUIS app

1. Right click on your project in Visual Studio > Manage Nuget packages > Browse tab.
2. Look for "Microsoft.Bot.Builder.AI.Luis" > select it and click Install > OK and Accept license terms.
3. Open the appsettings.json and add the keys you wrote down earlier for LUIS:

```
{
    "MicrosoftAppId": "",
    "MicrosoftAppPassword": "",
    "LuisAppId": "Application ID",
    "LuisAPIKey": "Authoring Key",
    "LuisAPIHostName": "Endpoint"
}
```

4. Open the Startup.cs and add this to the ConfigureServices method:

```
services.AddSingleton(sp =>
{

     var luisApp = new LuisApplication(
          Configuration["LuisAppId"],
          Configuration["LuisAPIKey"],
          Configuration["LuisAPIHostName"]);
    var recognizer = new LuisRecognizer(luisApp);
    return recognizer;
});
```

So we create an object for the Luis App we created earlier, and a recognizer we can use to detect the intent and extract the entities of our user input (utterance) using the model from our LUIS app.

5. In HotelBot.cs, add this using statement:

```
using Microsoft.Bot.Builder.AI.Luis;
using System.Linq;
```

6. And in the EmptyBot class add:

private LuisRecognizer _recognizer { get; } = null;

initializing the recognizer in the constructor:

```
public EmptyBot(LuisRecognizer recognizer) : base()
{
        _recognizer = recognizer ?? throw new System.ArgumentNullException(nameof(recognizer));
}
```

7. In the same class, add a constant for each intent:

```
public const string RestaurantIntent = "BookRestaurant";
public const string SpaIntent = "BookSPA";
public const string TouristIntent = "TouristInfo";
```

8. Let's ask the recognizer to analyze the input (utterance) from our user:

```
var recognizerResult = await _recognizer.RecognizeAsync(turnContext, cancellationToken);
```

9. And extract the intent and entities:

```
var (intent, score) = recognizerResult.GetTopScoringIntent();
var entities = recognizerResult.Entities;
```

10. Now let's extract the values of the entites and decide what to say back to the user depending on what they wish to do (depending on the intent):

```
string message = "";
var people = entities["number"]?.FirstOrDefault()?.ToString()?? "1";
var date = entities["datetime"]?.FirstOrDefault()["timex"]?.FirstOrDefault()?.ToString()?? "today";
var topic = entities["TouristAttraction"]?.FirstOrDefault()?.FirstOrDefault()?.ToString()?? "the city";

switch (intent)
{
    case RestaurantIntent:
            message = $"We will book a table for {people} people for {date}.";
                break;
    case SpaIntent:
            message = $"We will book an appointment for the Spa for {people} people for {date}.";
                break;
    case TouristIntent:
            message = $"We will provide you with information about {topic}";
        break;
```

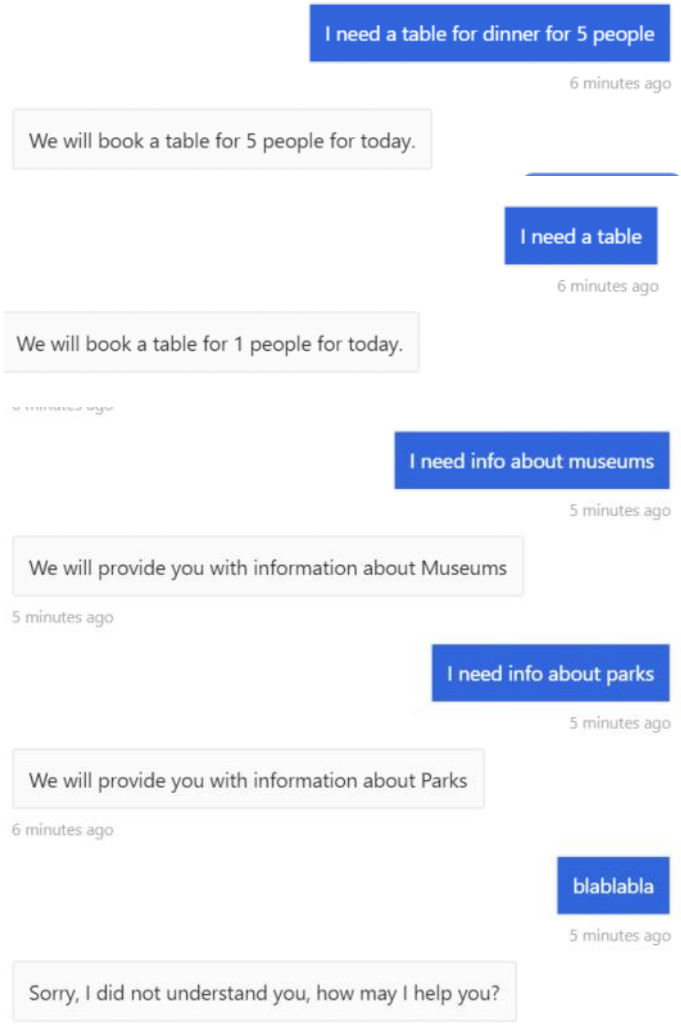
```
    default:
            message = "Sorry, I did not understand you, how may I help you?";
        break;
}

await turnContext.SendActivityAsync(message);
```

11. Now you can change the code of the OnMembesAddedAsync method back to the one we had initially to greet the user:

```
protected override async Task OnMembersAddedAsync(IList<ChannelAccount> membersAdded,
ITurnContext<IConversationUpdateActivity> turnContext, CancellationToken cancellationToken)
{
     foreach (var member in membersAdded)
    {
            if (member.Id != turnContext.Activity.Recipient.Id)
            {
                await turnContext.SendActivityAsync(MessageFactory.Text("Hello, how can I help you today?"));
            }
    }
}
```

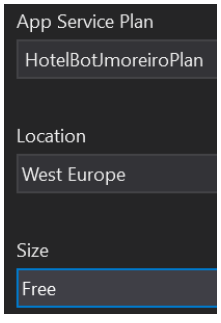
12. Run it and test it in the emulator:



13. You can set breakpoints to see in more details the information that is returned from your LUIS app regarding intents and entities. You can also use the emulator to inspect the requests and responses by clicking on the LUIS trace event, and even correct the LUIS intent detection if you add your LUIS service to the emulator:

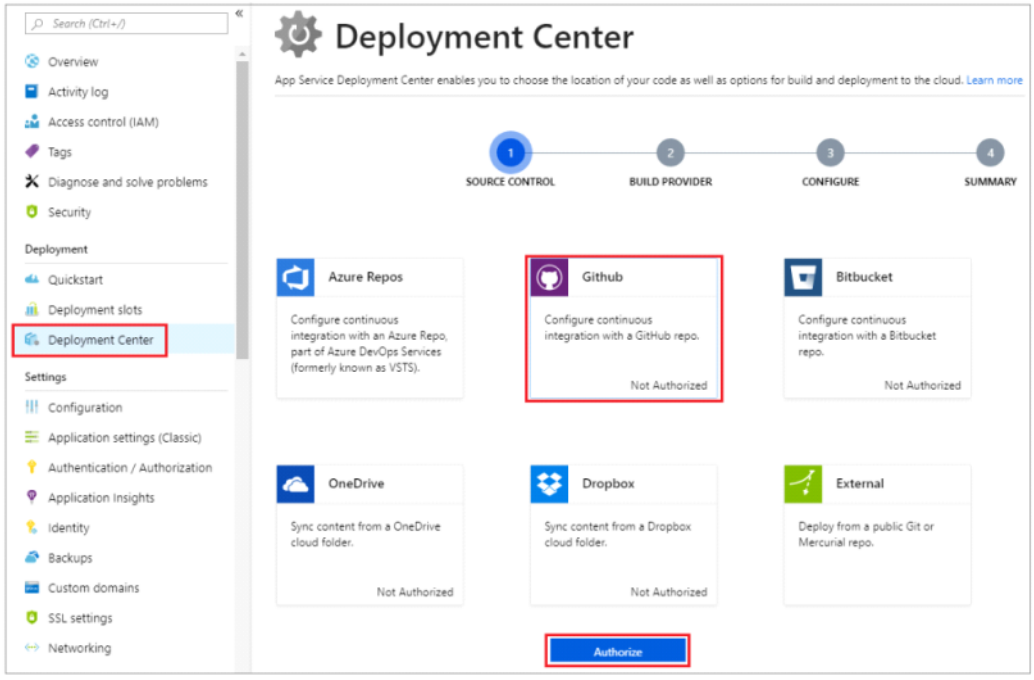
Part 5 - Source control and continuous deployment

1. If not already done, install the GitHub extension for Visual Studio and login to your GitHub account from within Visual Studio.
2. From your project, got o Team Explorer > Changes. Review the changes it has detected. Also check if any files which you don't want to sync to repo appear, if so you may add them to the git ignore. (you might have to run git rm --cached [file] if the file is already tracked).
3. Enter a name for the commit and click Commit.
4. Now click Sync > Publish to GitHub > enter the information for your repo and click Publish.
5. Sync.
6. Go to your GitHub account and confirm that the repo was created succesfully and that your code changes were pushed.
7. Now go ahead and publish yout bot on Azure. To do so right click on your project > Publish.
8. Select App Service > Create New > Publish.
9. On the top right corner make sure you select the right Azure account which you want to use to publish your bot.
   Make sure you select the correct subscription as well.
   Under resource group, select New and call the new resource group HotelBotRG.
   Under Hosting Plan click New, give it any name, region = West Europe and size = Free:

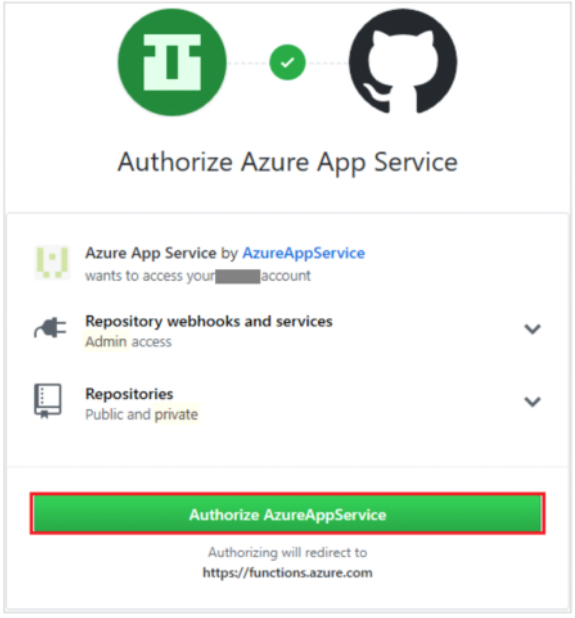


   Click Create. It can take some minutes to deploy, once deployment is finished continue with the next steps.

10. Now go to the Azure Portal (portal.azure.com) and sign into the subscription where you published your bot. Look under Resources to find your bot deployed.
11. Navigate to the App Service page for your bot in the Azure portal and click Deployment Center > GitHub > Authorize:



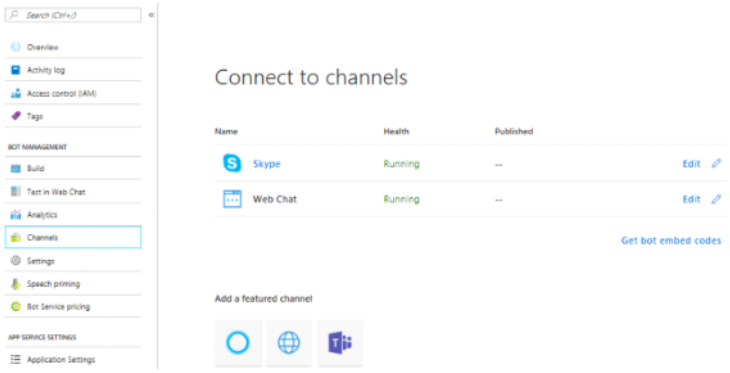
12. In the browser window that opens up, click Authorize AzureAppService:



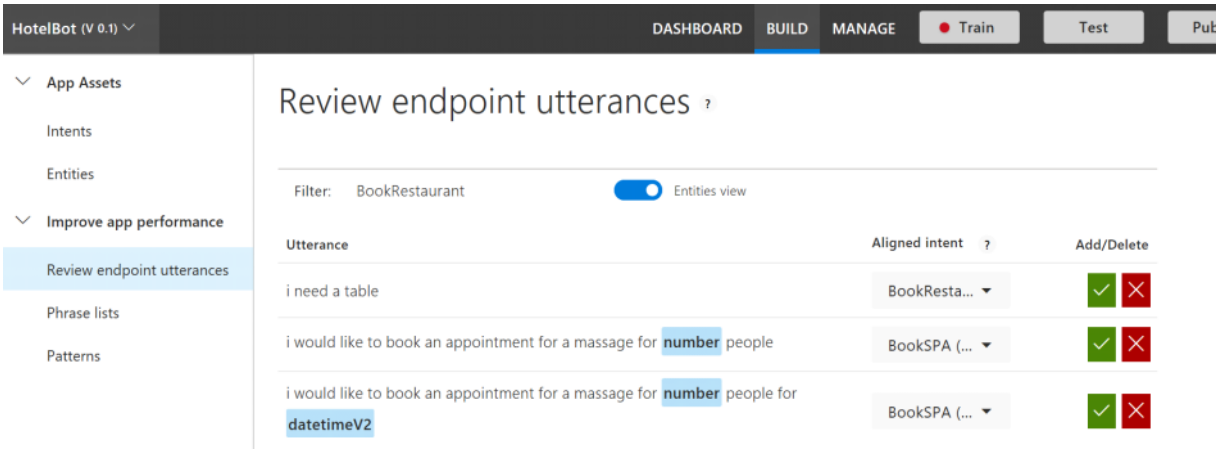
13. After authorizing the AzureAppService, go back to Deployment Center in the Azure portal.
    Click Continue.
    Select App Service build service.
    Click Continue.
    Select Organization, Repository, and Branch.
    Click Continue, and then Finish to complete the setup.
14. Now continuous deployment with GitHub is set up. Whenever you commit to the source code repository, your changes will automatically be deployed to the Azure Bot Service. Go ahead and test it.

Part 6 - Channels and other bot and LUIS features

1. Go to your bot in the Azure portal and click on Bot Management > Channels:

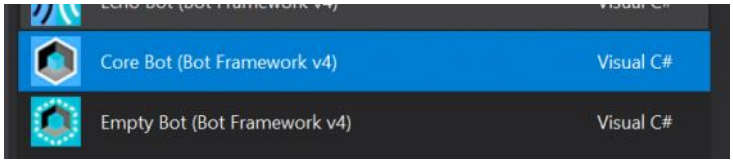


2. Select the channel you wish to connect your bot to.
3. Configure it and start using it.
4. Go to your LUIS app, add an intent and a custom entity. Test the different custom entities that you can create. Test creating a role for an existing entity (for example create roles "origin" and "destination" for a geography entity to use on a BookFlight intent. Add utterances and see how to tag not only the entity but also the role for each.
5. Test your bot with several different utterances and see if any are assigned to an incorrect intent. If so go to your LUIS app > BUILD > Review endpoint utterances > correct the Intent and Entities which are incorrect and click Add. Then click Train.

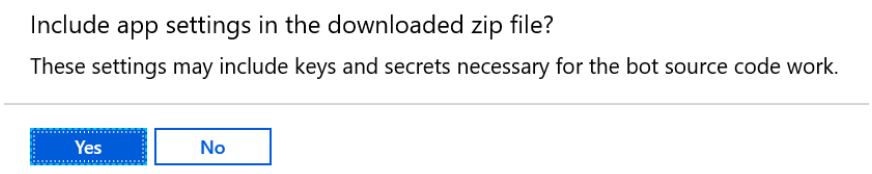


Part 7 - Further steps

1. Now check some samples of more complete bots. For instance, create a new project selecting the Core Bot template:



2. This will include features which we were not able to cover during this workshop such as Dialogs and more. Take a look at the code of this bot to understand how further features (state storage, waterfall dialogs, etc) can be implemented.
3. Also test creating a bot directly in Azure. From your Azure subcription > New Resource > Webapp bot > Basic bot. This basic bot template will create a very simple bot and a LUIS project app associated with it with a simple intent and entities.
4. Give it any name and properties, but select West Europe for both the bot and the LUIS region.
5. Go to Bot management > Build > Download Source Code. Make sure you answer yes to download the app settings:



This will download a zip file with our bot code so we can run locally. Extract the zip onto a folder, then open in Visual Studio.

6. If you want to edit the bot's code directly from within the portal, you can click here:

Make quick changes to your bot code online, run build.cmd in the editor console, and see your changes instantly. Ope
online code editor