D&D 5e Character Suggestion Bot

Preface:

This bot was built following the [Azure Bot Service Documentation](https://docs.microsoft.com/en-us/azure/bot-service/?view=azure-bot-service-4.0).

This documentation will also guide you through the steps of setting up and getting started to create your own bot to suggest classes. The prerequisites for this guide are the following:

* Visual Studio 2017
* Bot Builder SDK v4 template for C#
* Bot Framework Emulator

Overview:

This bot was designed and built to help others who might be new to Dungeons and Dragons 5 Edition to give them suggestions on a character to create. Dungeons and Dragons is a large system with many options available for the user to choose from and can be quite daunting if unaided. This bot will help focus the user onto a character suggestion that will help them pick out a Class, Race, and what Role they should be going for.

Design:

The bot was designed to utilize external data structures held with XML files in order to allow developers to easily expand the data with more entries. The bot is able to open these files and parse them in the provided formats. Utilizing XML Serialization it makes it so there are no hard coded entries within the bots code.

Goals:

Have the bot do the following:

* Ask if the user wants a character suggestion
* Step through a few questions about what they like while providing feedback every step of the way.
* Print out a final suggestion with the given answers.

Getting Started:

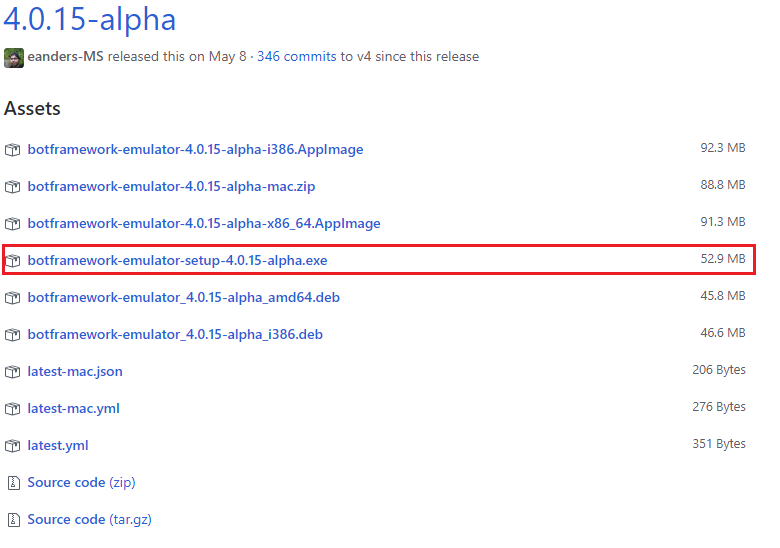
A bot can be created in several ways but we will be going through the route of building it with the [Bot Builder SDK v4 template for C#](https://marketplace.visualstudio.com/items?itemName=BotBuilder.BotBuilderV4).

Additionally make sure you have the following NuGet packages installed:

* [Microsoft.Bot.Builder](https://www.nuget.org/packages/Microsoft.Bot.Builder/4.0.1-preview)
* [Microsoft.Bot.Builder.Dialogs](https://www.nuget.org/packages/Microsoft.Bot.Builder.Dialogs/4.0.1-preview)

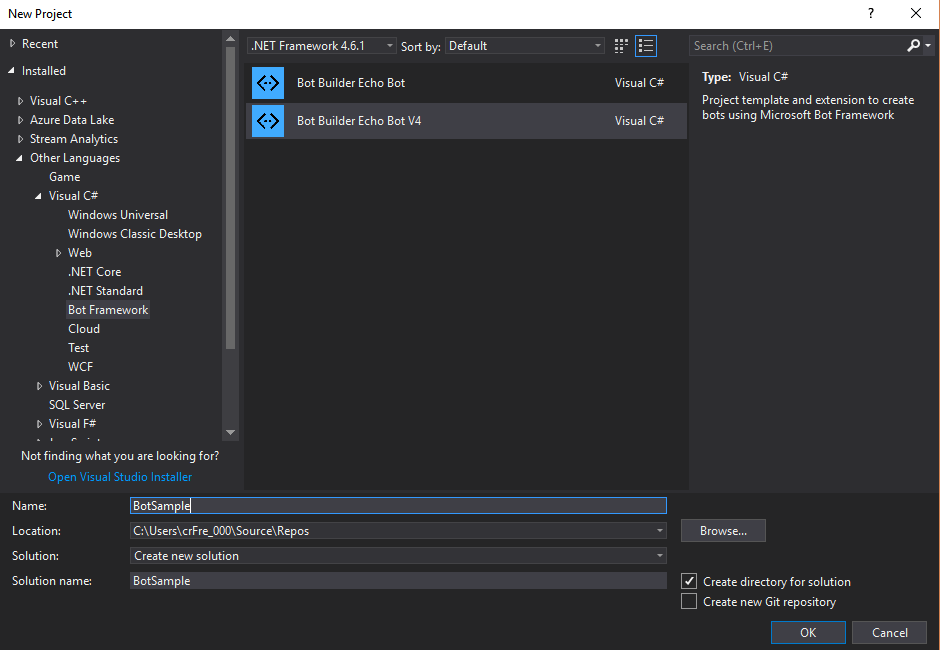
We will be testing out bot with the [Bot Framework Emulator](https://github.com/Microsoft/BotFramework-Emulator/releases). Let’s install it now as it will be incredibly useful for debugging purposes. It is better to do incremental testing when you make changes versus building the entire bot in one go and spending hours implementing fixes.

When you’re at the [Bot Frame Emulator](https://github.com/Microsoft/BotFramework-Emulator/releases) github page, proceed to download the appropriate installer *(Figure 01)*.



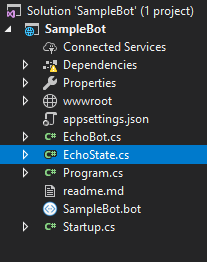
*Figure 01: Download for Bot Emulator*

Once it is downloaded, install it. When that is finished let’s create a new project *(Figure 02)*.



*Figure 02: Creating the new Bot*

Once the bot is created, you should have a fresh clean project. We will be repurposing much of it to fit out needs. The template provided gives us a simple EchoBot. We will be changing this to fit a Waterfall discussion bot so we can pass new information on to each new Async call. Go ahead and rename the EchoBot and EchoState to something else if you wish, ex: *DnDBot* and *DnDState* respectively. *(Figure 03)*.



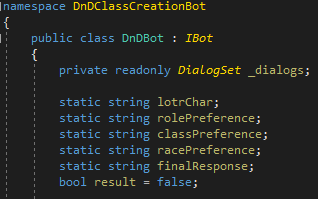
*Figure 03: Solution Explorer of new Project*

The Waterfall method will allow us persist an entire dialogue conversation, and if there is an issue or an error, we will simply start the discussion over from the beginning. Now, let us go to the *‘EchoBot.cs’* and make sure we include the namespaces we need *(Figure 04)*.



*Figure 04: Namespace includes*

We will be using the Dialogs to have access to the nuget functionality we downloaded. We will also use Regular Expressions to help parse information that the user will send us. Now that we are speaking of Dialogue, we should establish our variables we need to preserve the information the user will provide us. We will want a variable for every prompt we require from the user *(Figure 05)*.

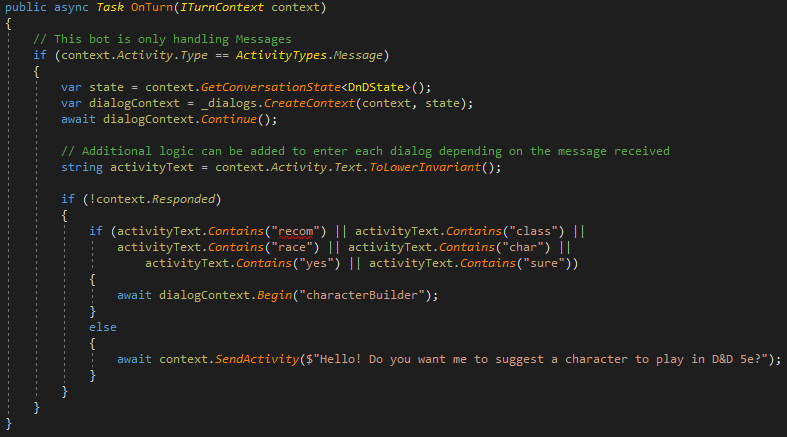


*Figure 05: Variables to hold user responses*

The following variables are used:

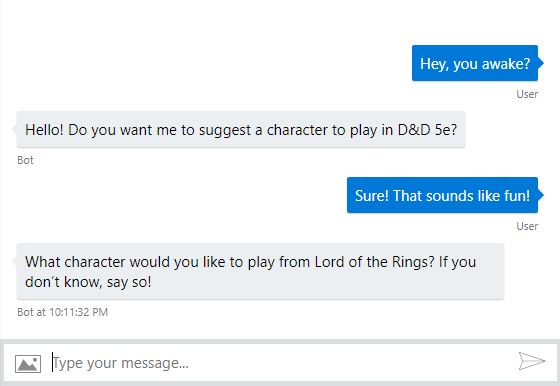
* **lotrChar**: When we prompt the user to select their favorite Lord of the Rings character to help guide them receiving a suggestion.
* **rolePreference**: We will then prompt them about the role they might want to play based on their previous response.
* **classPreference**: We will ask them what class of the provided suggestions they wish to play.
* **racePreference**: We will then provide them with suggestions on which races might be good for their class.
* **finalResponse**: This will handle providing the user the final response that will provide them their suggested character.
* **bool result**: This variable will be our reference for if our bot can’t find what they typed. This will be incredibly helpful later.

We will not worry about the Bot constructor just yet, as we want to set up a couple of other things first. We mainly are focused on the *‘****public async Task OnTurn(ITurnContext context) {}****’* function *(Figure 06)*.



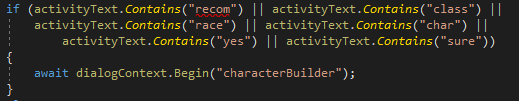
*Figure 06: Task OnTurn function*

When out bot first initiates, we want to make sure we can handle the users first response. This is done in the second if block: ‘***if(!context.Responded)***’. Here we parse the activityText and check for keywords that they may use. If they send anything else that doesn’t get caught by the check, then we prompt them with question which will help them provide a prompt to initiate our bot *(Figure 07)*.



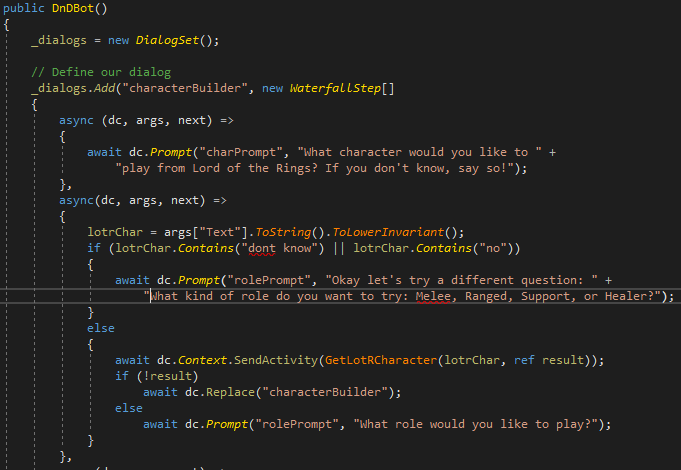
*Figure 07: Bot response example*

We do the check for keywords, so we can handle a large array of user input, from a simple yes, to short phrases. Now, let’s see how we enter the waterfall dialog prompt.



*Figure 08: Starting the Dialogue*

The ***‘await dialogContext.Begin("characterBuilder");’*** code will begin out dialogue named ‘characterBuilder’ *(Figure 09)*.

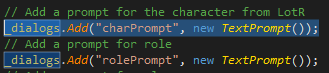


*Figure 09: Beginning of the Waterfall Dialogue Prompts*

There are a few things to take note of here, the code ***‘\_dialogs = new DialogSet();’*** initializing our Dialog system, then we need to add the dialog with ***‘\_dialogs.Add("characterBuilder", new WaterfallStep[]{}’*** . This enables our previous code ***‘await dialogContext.Begin("characterBuilder");’***  to work.

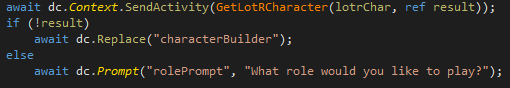
The next thing to note are the ***‘async (dc, args, next) => {}’*** blocks of code. The Waterfall method will step through each async in sequence, every previous async block preserving the dialogcontext and any user input. It is within these blocks where we will do most of our logic.

You will also need to make sure that at the bottom of the constructor of your bot to declare each DialogSet prompt, ***‘\_dialogs.Add("charPrompt", new TextPrompt());’***. It must match the prompt name you use within your ***‘await dc.Prompt’*** calls. If they do not match, you will get a *“Value cannot be null”* error *(Figur 10)*.



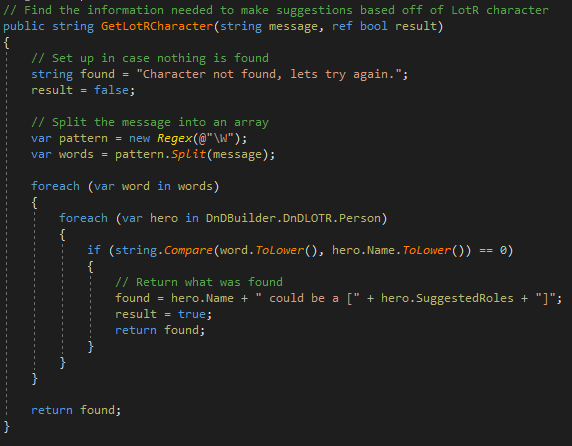
*Figure 10: DialogSet prompts*

This is a good place to exercise Object Oriented Programming but removing any other functionality we want into helper functions *(Figure 11)*.



*Figure 11: Helper Functions and reference returns*

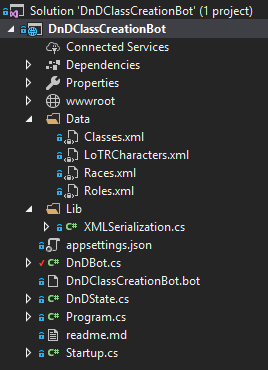
This section of code does several things. I use the ***‘ await dc.Context.SendActivity(GetLotRCharacter(lotrChar, ref result));’*** to cause the bot to wait, call a helper function and return responses via the helper function. I use the ***‘ref result’*** to allow me to know if the function failed or succeeded and then perform the appropriate operations. The ***‘await dc.Replace("characterBuilder");’*** code is something to note, as when our helper function fails by user input or otherwise, it will stop the current dialogue and restart it. This function is also useful if you were trying to create a loop or transition into other dialog trees.



*Figure 12: Helper Function*

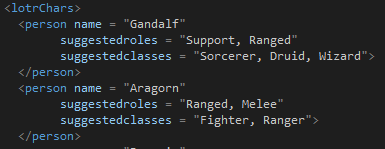
Helper functions are done as per standard C# coding practices. We define the function at the bottom of the file. Here I take in the user input and the ref to the boolean results, then I set up the failure states, ensuring I initially set the results and return value. I then use regular expressions to split the message into an array of words and operate over the input and compare it with my stored data. Depending on the results this function will either return the information requested or say nothing could be found and prompt the bot to restart the dialog. An important note is that this helper function is operated in a separate thread, so any global variables you set or attempt to set will not persist outside of the async call without us defining our own async callbacks *(Figure 12)*.

This brings us to the data structures. For this bot, we are going with modularity and external XML files. This helps us from hardcoded any data into the bot and is a good practice to maintain *(Figure 13)*.



*Figure 13: XML Serialization Setup*

I like to keep my data structures separated in their own folder, and out of practice I have a folder for any library functionality I implement. Now going over XML Serialization is outside of the scope of this documentation, now saying that, you could have something set up similar to mine *(Figure 14)*.



*Figure 14: Sample xml class*

However you decide to set up your data structures is up to you, but ideally we will still want to have the same information at hand. I recommend this as a challenge opportunity to set up your data structures as necessary to handle the questions. Remember, we need a data structure for the following:

* Lord of the Rings Characters with their name, suggested roles, and suggested classes.
* Roles structure that will contain Melee, Ranged, Support, Healer with each category providing suggested classes that fall under those four categories.
* Classes that will contain all the classes we require and their primary attributes so we can do comparisons later.
* Races that the player can select from, which each race will have their Name, Attribute Bonuses primarily.

Once you have your data structures in place, it’s time to go to our BotState.cs and initialize our data structures *(Figure 15)*.

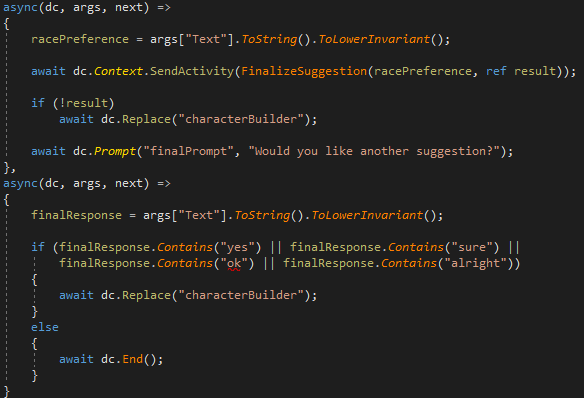


*Figure 15: Initializing our Data Structures*

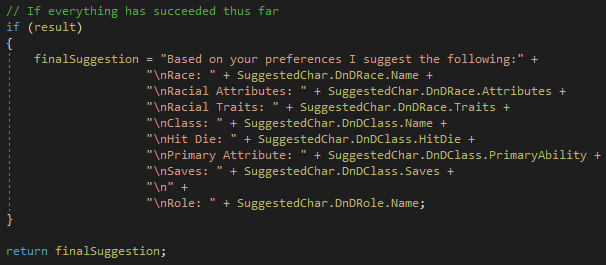
Now we have our primary builder ***‘public class CharacterBuilder {}’***, that contains all the information serialized from our XML documents. We then have a new class named ***‘public class CharacterSuggestion {}’*** that will hold our final suggestions.

You now have the necessary building blocks to challenge yourself to build out the other async calls in the waterfall dialog in the Bot.cs file.

At the end of my waterfall dialog I take the last input needed from the user, in my case it being their choice for race, and then call a helper function that puts the final touches, uses the previous suggestions in the variables and packs it all together in a nice response and sent it off to the user *(Figure 16 & 17)*.

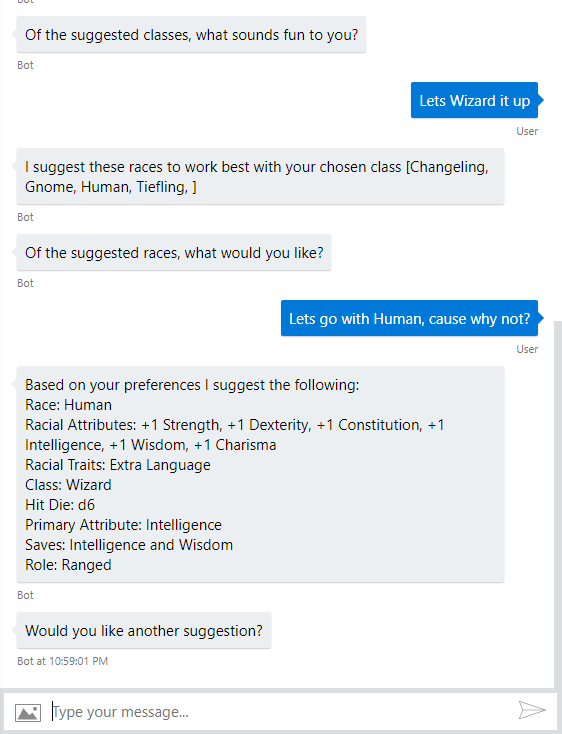


*Figure 16: Final async calls*



*Figure 17: Helper function output*

With that, you should be able to finish the bot and have a bot that will be able to give you recommendations on what character you could create for Dungeons & Dragons 5e! *(Figure 18)*

**

*Figure 18: Bot giving suggestions and recommedations*