{ Lab 7 }

Building Advanced Dialog in Watson Assistant Service

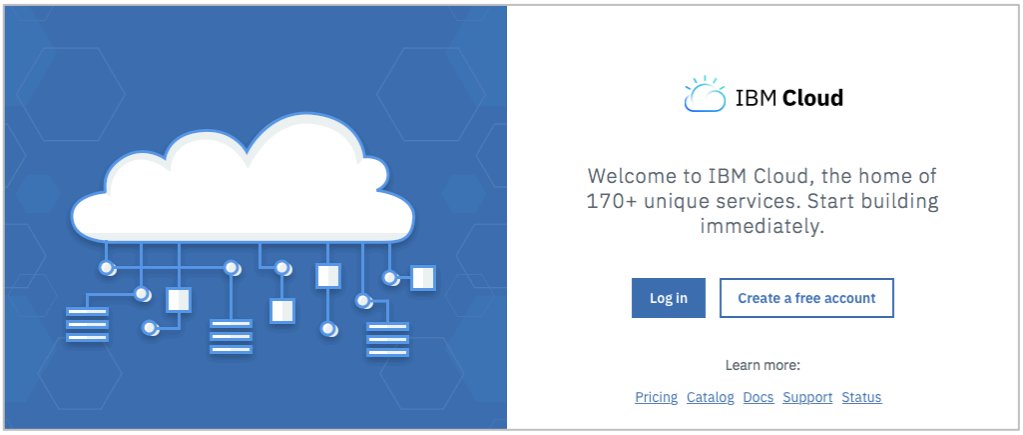
Agenda

Before Starting

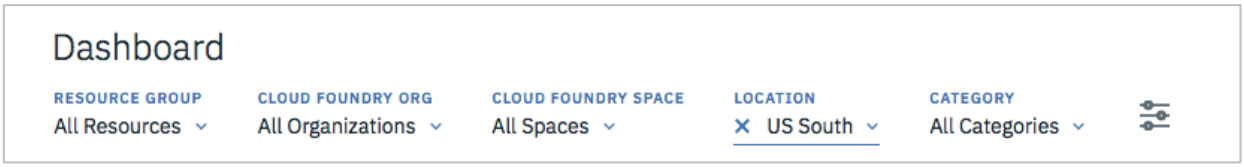
Information

This hands-on required to have an IBM Cloud account. If you don't, you can create one here: http://bluemix.net/.

__ 1. Open a browser and access to IBM Cloud: https://console.ng.bluemix.net.
Log in.



__ 2. Select organization and space to use during this lab



__ 3. If needed, free resources (GB / #Services) in your IBM Cloud Organization / Spaces to run the lab exercises.
If you encounter a resource contention (Error Message saying you are out of resources), clean up your Spaces by deleting existing Apps or Services.

1. Hands-on presentation

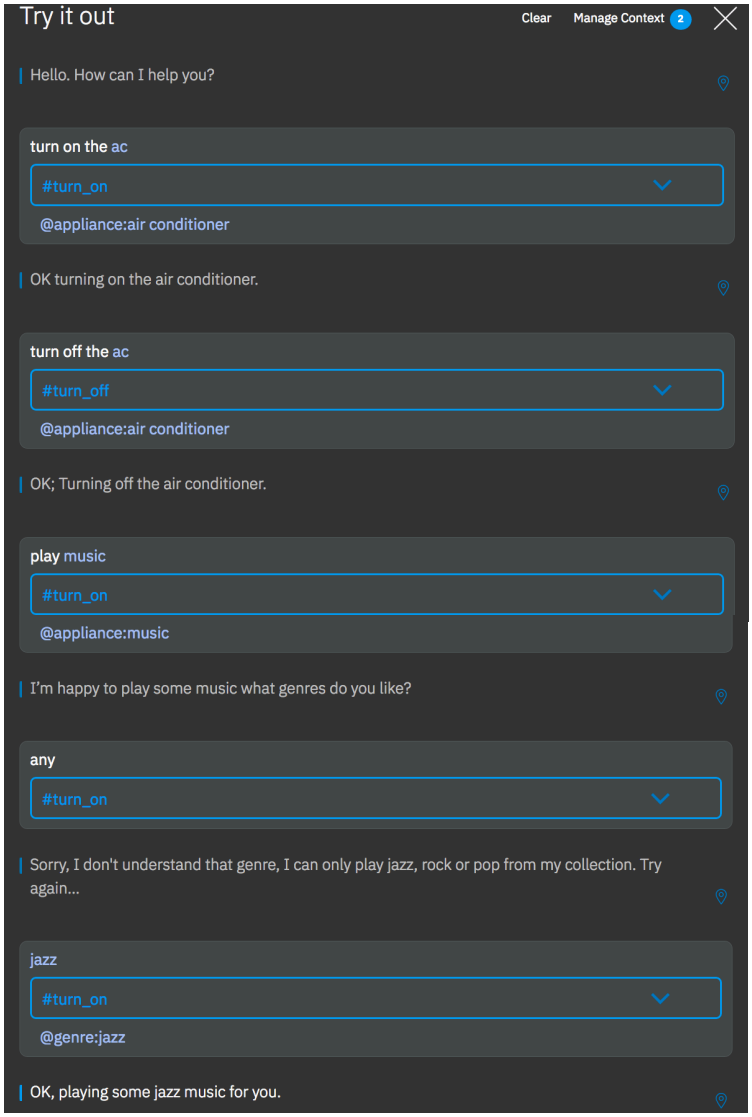
Section 1. Objectives

The purpose of this guide is an instructional approach to working with the IBM Watson Assistant service where you can create virtual agents and bots that combine machine learning, natural language understanding, and integrated dialog tools to provide automated customer engagements. Watson Assistant provides an easy-to-use graphical environment to create natural conversation flows between your apps and your users. Creating your first conversation using the IBM Watson Assistant service entails the following steps:

- Train Watson to understand your users' input with example utterances: Intents and Examples
- Identify the terms that may vary in your users' input: Entities
- Create the responses to your users' questions: Dialog Builder
- Test and Improve

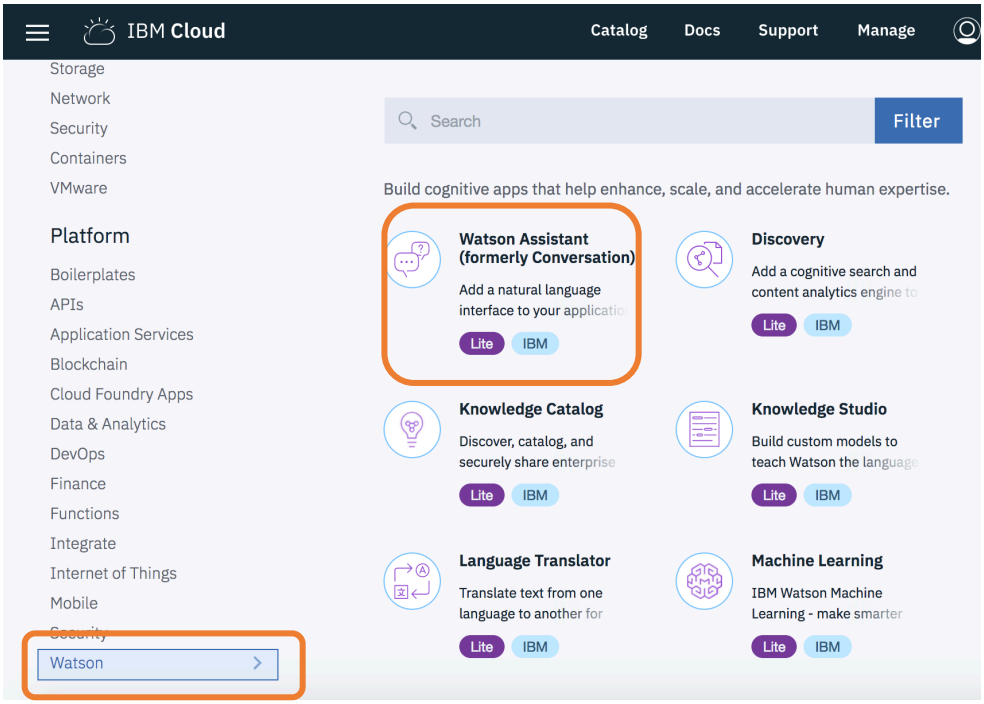
Section 2. Expected Results

- You created an advanced dialog with IBM Watson Assistant

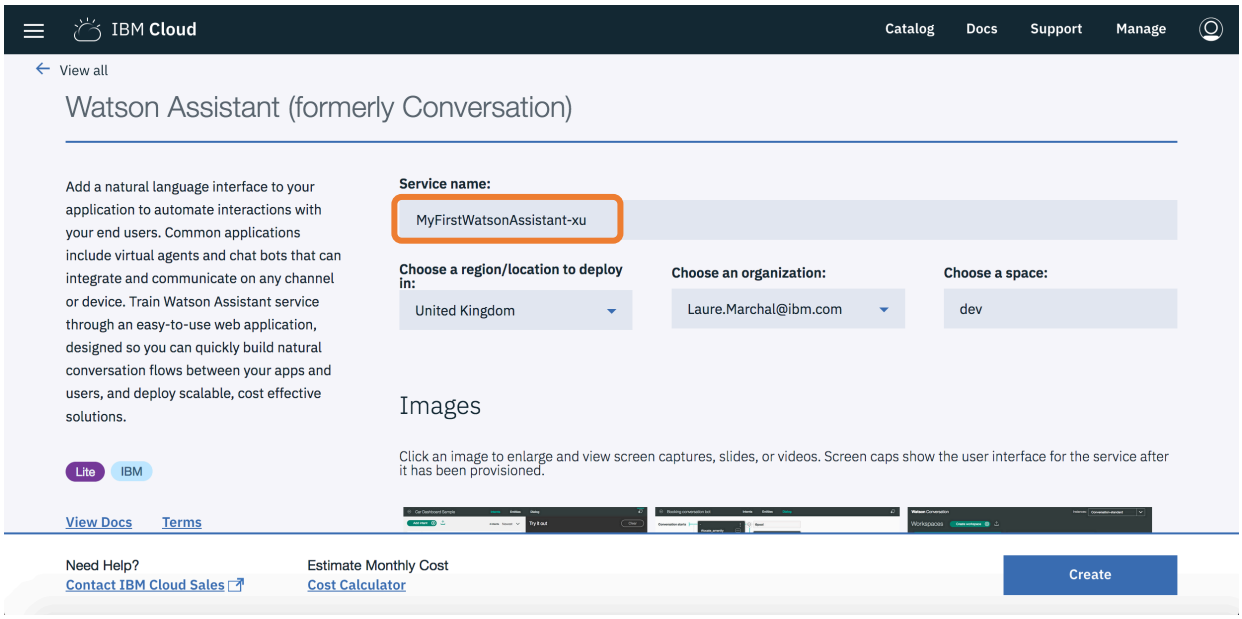


2. Create and Launch

__ 1. In IBM Cloud Catalog, choose “Watson” category

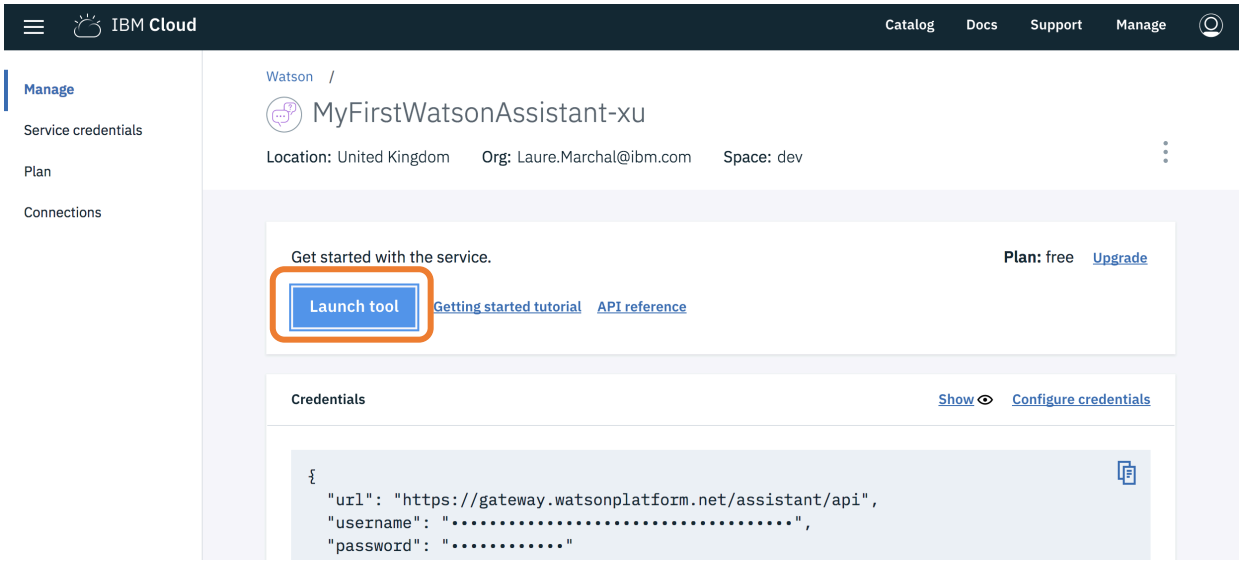


__ **2.** Click on **Watson Assistant (formerly Conversation)** & create an instance: Fill in the App Name & select the *Lite Plan*.

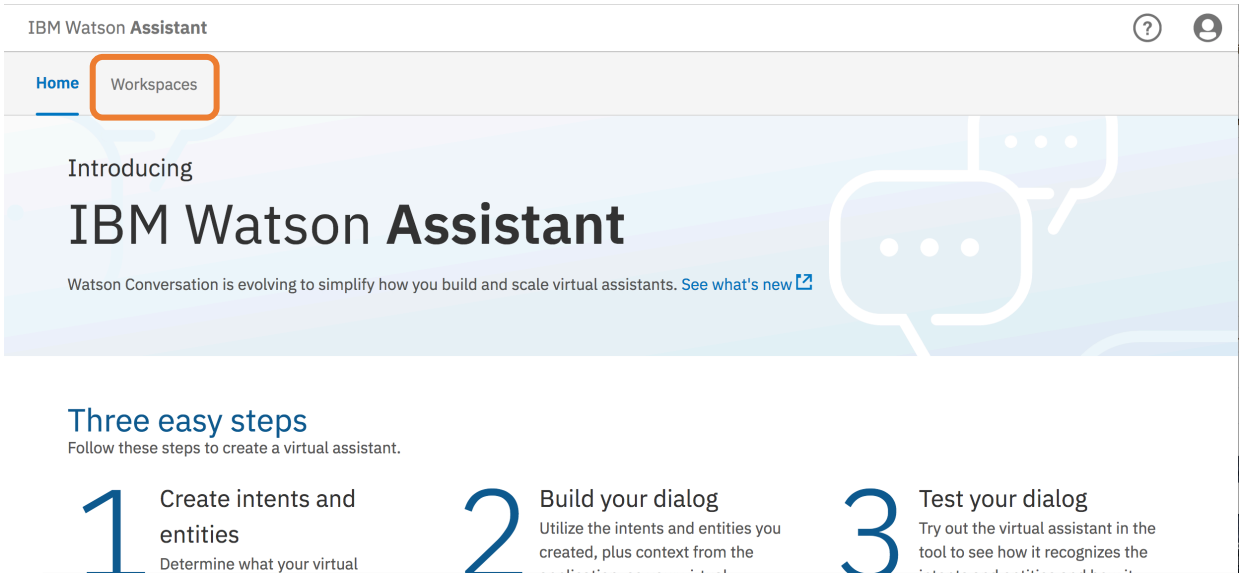


__ **3.** Click **Create**.
Wait for the environment to be created.

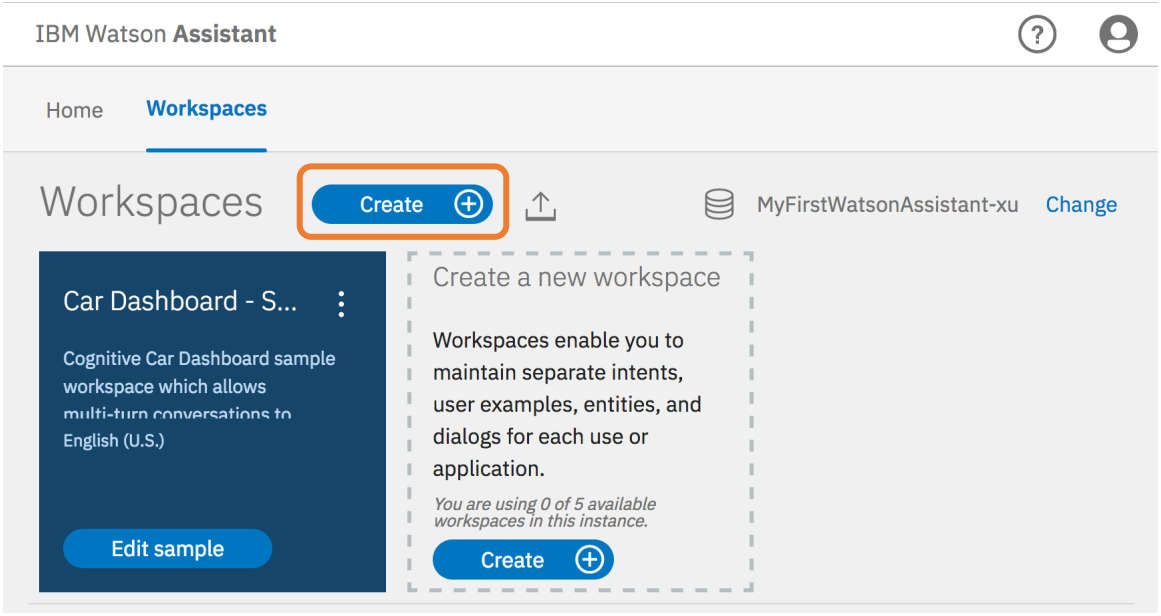
__ 4. Click the **Launch Tool** button.

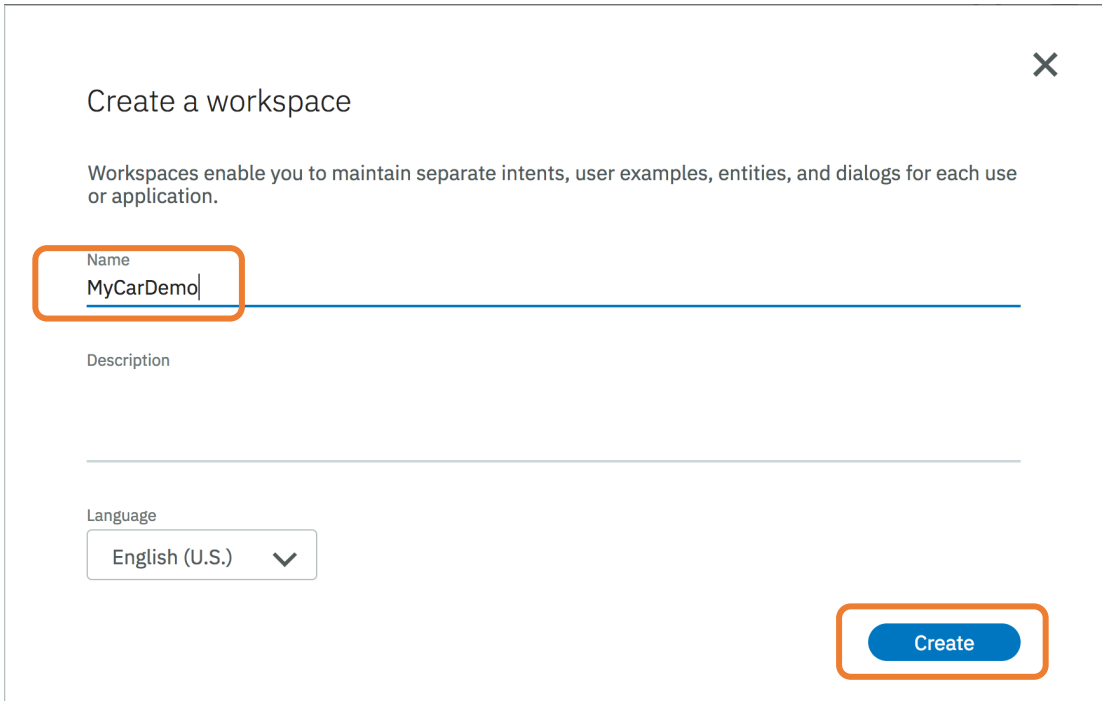


__ 5. Click the **Workspaces** tab.



__ 6. Click **Create** a new workspace, name it: “MyCarDemo” and click **Create**.

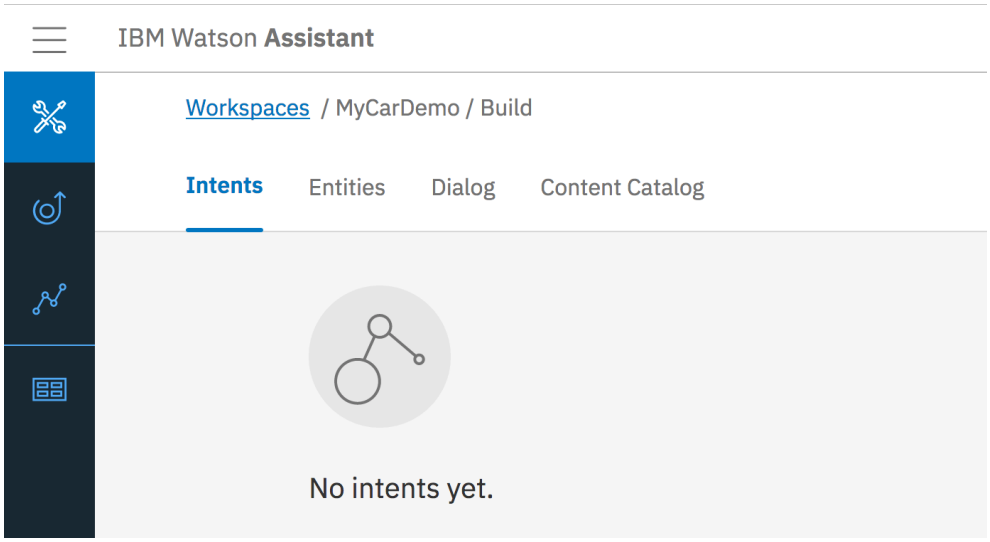




3. Build a chatbot

Section 1. Add Intents and Entities

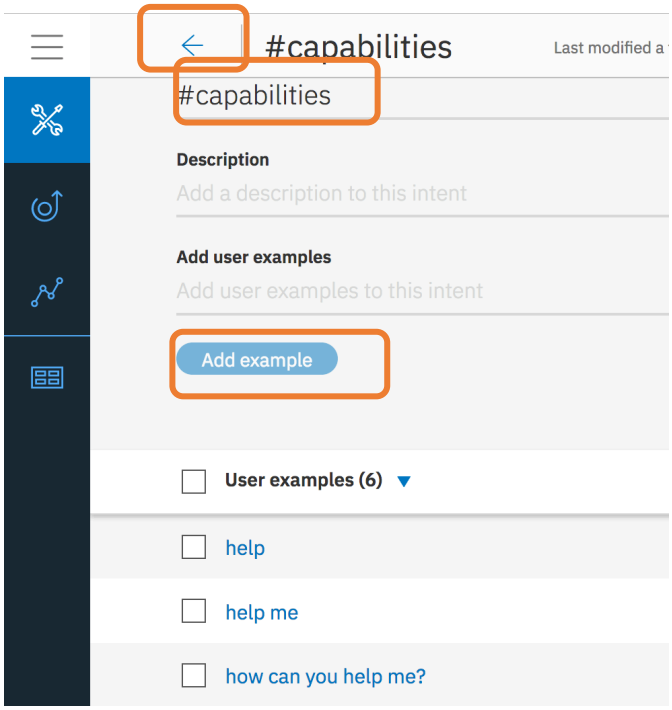
__ 1. Wait for the workspace to be created, then you start building your chatbot and click **Add Intent**.



__ 2. Add a new **Intent**, name it **capabilities** and click **Create Intent**.

__ 3. Add a few examples of how you might ask your car what are its capabilities; for example, *help, help me, how can you help me?, what can you do? What are your capabilities?* And so forth. Minimum of 5. The more the better. After all, machine learning is about building patterns.

__ 4. Click the **blue arrow** to go back to the workspace.



Next you will create two new intents of just what it is that the car can do; for example, turning things on and off is a good place to start.

__ 5. Add a **new Intent**, name it **turn_off** and click **create intent**.

__ 6. Add a few examples of how you might ask your car to turn off certain things; for example: *off, power off*, *stop, turn off, quit* and so forth. Minimum of 5. The more the better.

__ 7. Click the **blue arrow** to go back to the workspace.

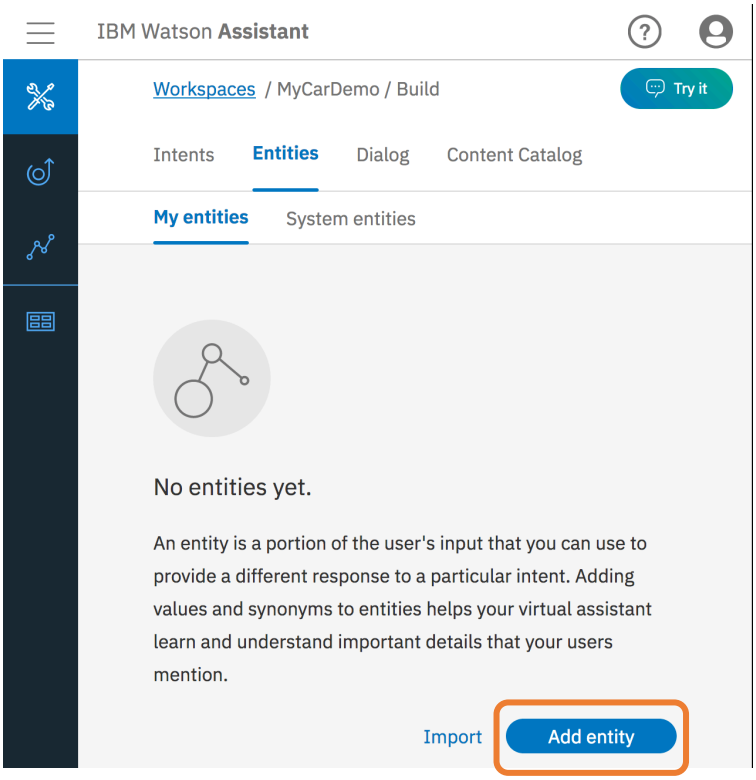
__ 8. Add another **Intent**, name it **turn_on** and click **create intent**.

__ 9. Add a few examples of how you might ask your car to turn on certain things; for example: *begin, on, play, power on*, *turn on* and so forth. Minimum of 5. The more the better.

__ 10. Click the **blue arrow** to go back to the workspace.

You are now ready to create Entities. In this example, Intents are about “what can the system do”, how can you classify and group together in one bucket similar intended things that the car can do. Entities are about what “specific things” can be acted upon now that the system knows what to do. If the system, knows that your intention is to turn on something, then the entity specifies what are some of the things that you can turn on in a car: heater, engine, air conditioner, music—all sorts of things that you can turn on or off in a car.

__ 11. Click the **Entities** tab and click **Add entity**.



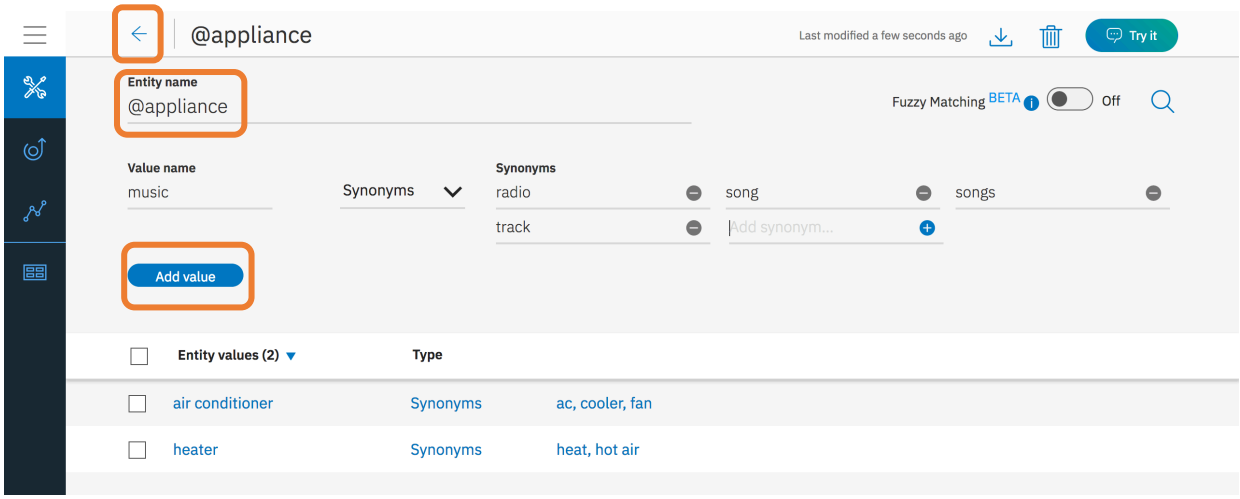
__ 12. Type **appliance** and click **Create entity**.

__ 13. Add new value of **air conditioner**, plus synonyms such as *ac, cooler, fan* and click **Add value**.

__ 14. Add another value of **heater** and add synonyms such as *heat, hot air* and so forth. Click **Add value.**

__ 15. Repeat the above steps and add another value of **music** with synonyms such as *radio, song, songs, track* and so forth. Click **Add value.**

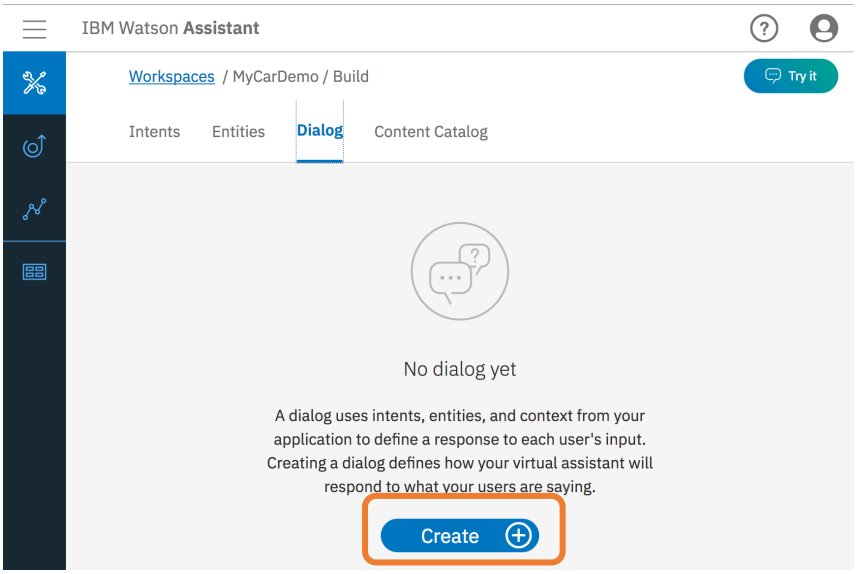
__ 16. Click the **blue arrow** to go back to the workspace.



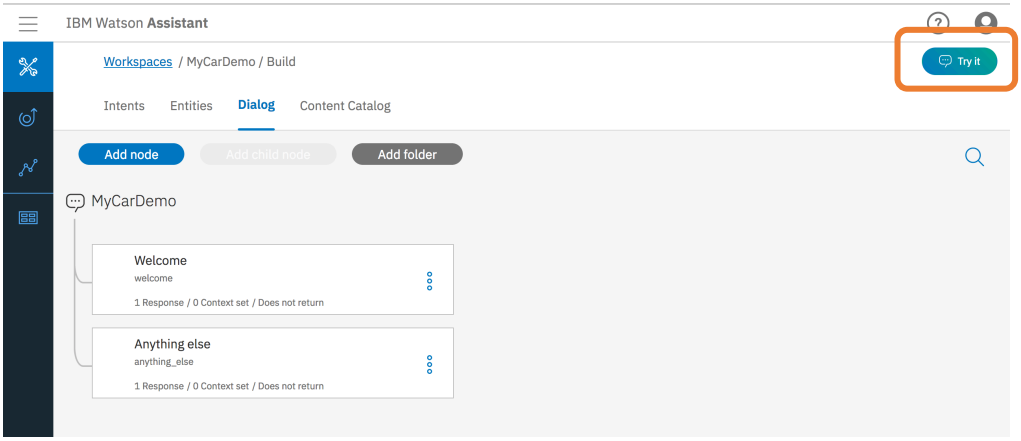
Section 2. Create the dialog tree

__ 17. You are now ready to create the Dialog tree. Click the **Dialog** tab.

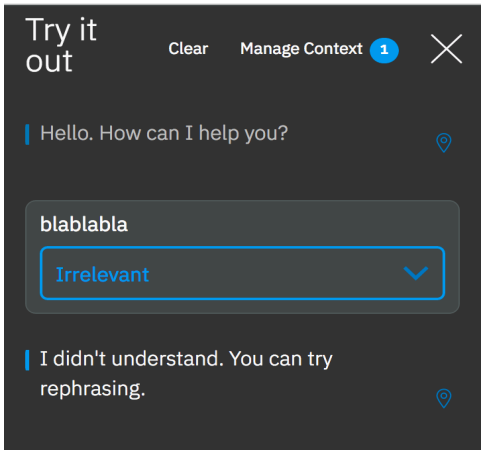
__ 18. To create a new dialog tree, click the **Create button.**



__ 19. You can now see two dialog nodes. They are the two default ones. Click on **Try it**, a new panel will appear on the right side. This is the dialog window that allows you to test your dialog tree.

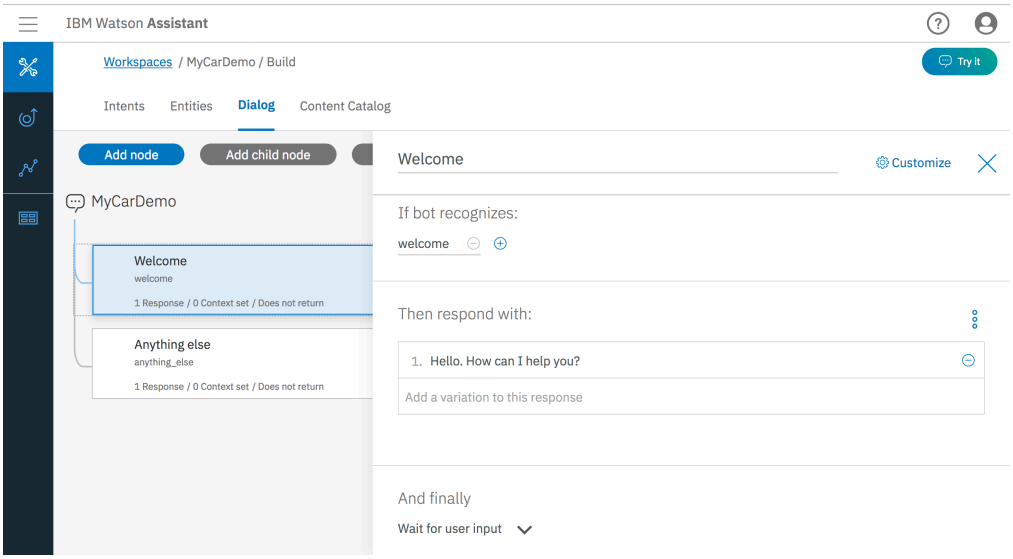


__ 20. The welcome node is executed and the chatbot says “Hello. How can I help you?”. Just type a bunch of letters to see what happens.



So not much happens, because you do not have any other dialog nodes. But the Anything else box opens. That catches all nodes and is always set to true in case the input is not understood.

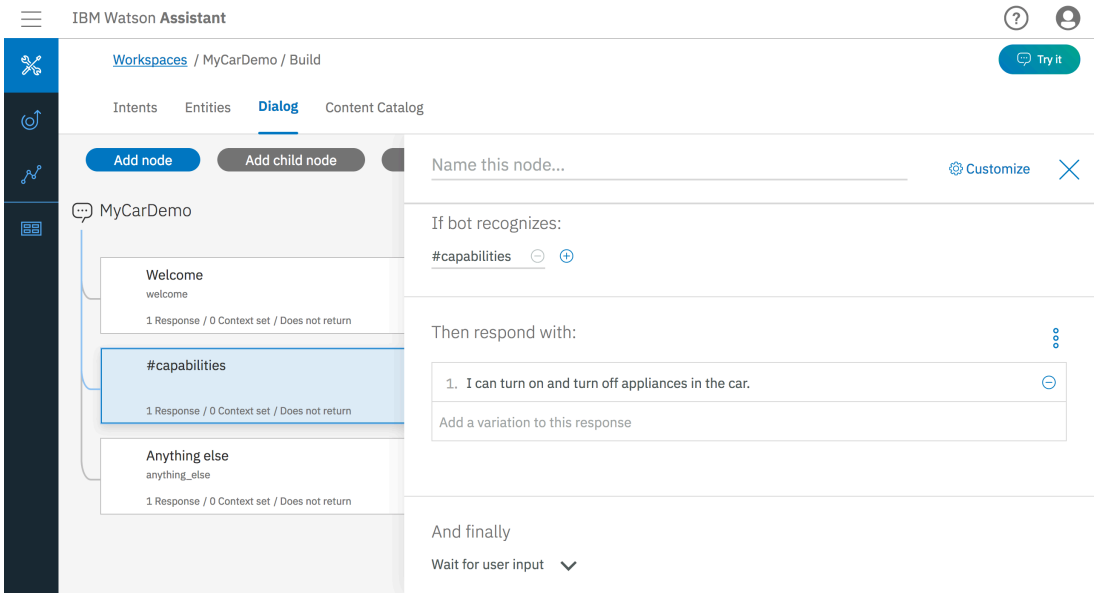
The nodes have a specific structure, which you can see by clicking on them. The top third is merely a Boolean condition: true or false. The next is the response that has a simple and an advanced view (where you can include JSON objects to add dynamic output and contextual variables for multi-turn conversations). And the last is what the chatbot must do after its response.



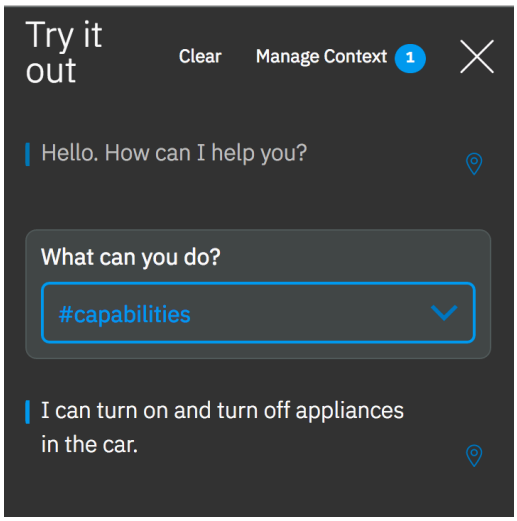
Ok, so let's add more dialog nodes. You will now add sibling nodes (sits underneath, different conversation) and children nodes (sits underneath the parent but with indentation).

__ 21. Click the **Add node** button to create a sibling node after the Welcome. Type **#capabilities** (# to specify an intent). If you just type *#cap…* the rest appears in the drop down and select the option corresponding.

__ 22. In the response section, type: **I can turn on and turn off appliances in the car.**



__ 23. Click the **Try it button** to test your dialog and type **What can you do?**

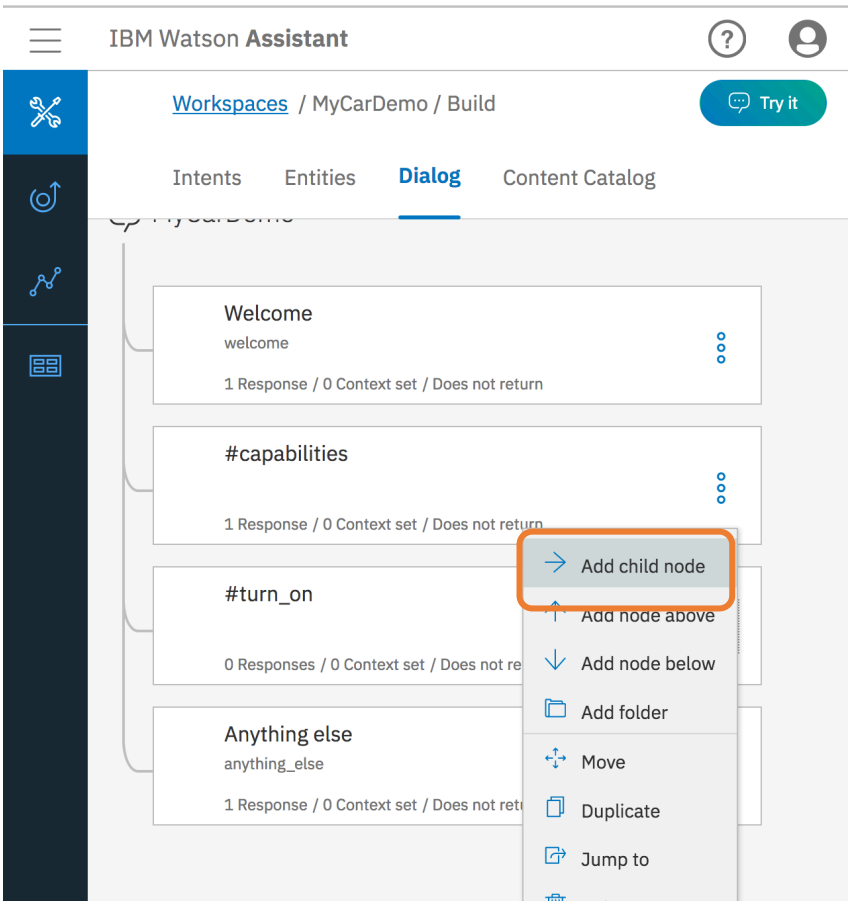


So, the chatbot recognized the intent, the capabilities node had a condition true, so it executed that condition and hence the response that you got.

__ 24. Clear the chat box and close it.

__ 25. Let's build on this dialog by clicking the **Add node** button and specify the **turn_on** condition.

__ 26. Click the **three vertical dots** of the **turn_on** node to **add a child node**, because you want to specify turn on which appliance. Because it might have an Intent but not an Entity, so let's condition it on appliance.

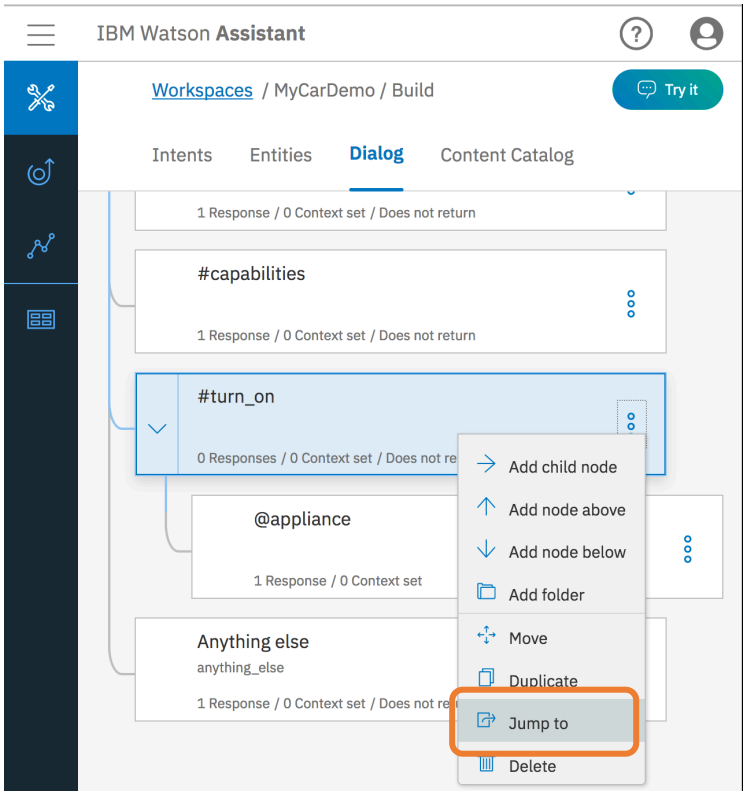


__ 27. So, if there is an intent on an appliance, you asked it to turn on something. Type **@appliance** in the condition section, click **any** in the drop-down list, then type in the response section **OK turning on the @appliance** and close the node window**.**

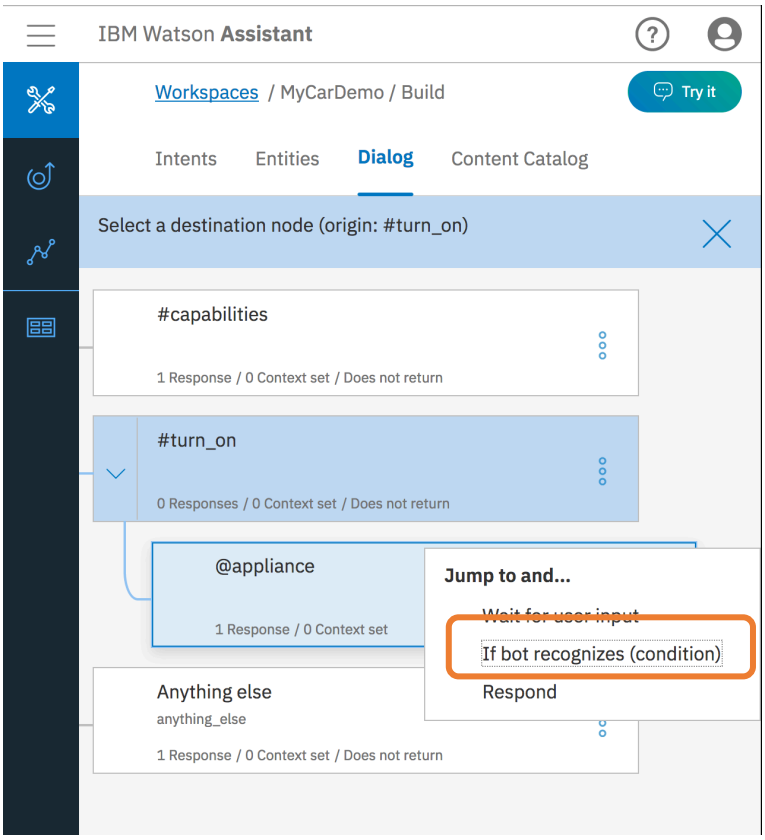
__ 28. Test the dialog and type **turn on the ac.** Notice that it gets the intent and the entity right, but no response.

The reason is that there is a default *wait for user input*. This means, once the *turn_on* node is evaluated for true, then it is going to wait for the user input before moving to the child node *appliance*. You overcome this problem by adding a *Jump to…* This is very useful when you need to move different location in the dialog tree.

__ 29. Click the three vertical dots of the *turn_on* node, click **Jump to** and then click on the *appliance* node**.**



__ 30. So, you either jump to and *wait for the user input* or *respond* directly or *a condition*; click on **If bot recognizes (condition)**.



__ 31. Let's test it. Click the **Try it** button and type: **Turn on the ac**. Notice the response comes back as expected.

So, consider this: What will happen if there is no appliance entity? It will evaluate the *turn_on* node, then be forced to fall back to the root node and if nothing else matches the *turn_on* condition, then the *Anything else* node will capture it.

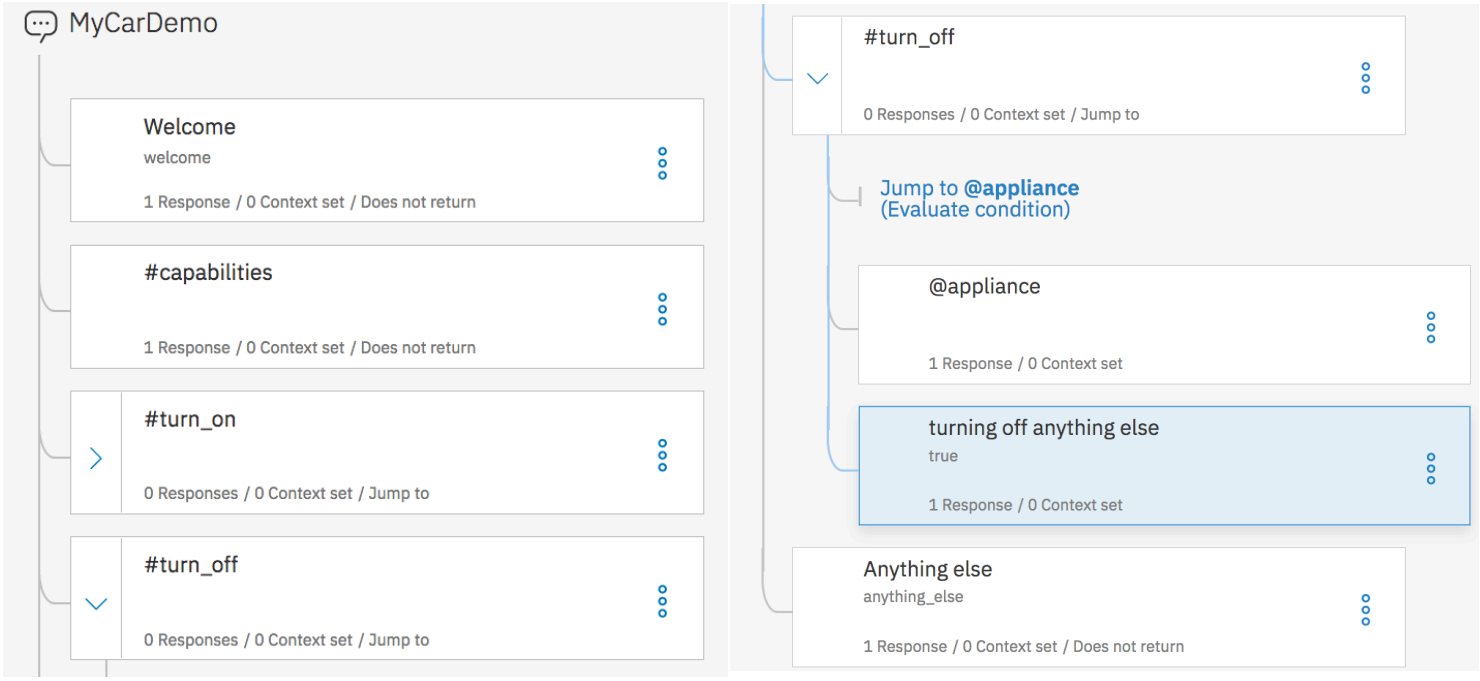
So, let's add a default dialog of our own as a sibling to the *appliance* node, such that if nothing matches, it will capture it with better relevance and not go back to the root.

__ 32. Add a sibling node to the *appliance* node, name it "turning on anything else", set the condition to true (type it in the condition section) and type the following in the response section: **I understood your intent was to turn on something. Try saying turn on the ac.**

This is just some text that the car provides when the user does not provide an appliance to do something with.

__ 33. Create the **turn_off** node underneath the *turn_on* node.

- Create the child node of **@appliance.**
- Ensure it is *jumped from* the *turn_off* node.
- Add a **true** sibling condition node and Name it like the other.
- Copy paste the responses from the *turn_on* node, except have it say turn **off** instead of turn on.

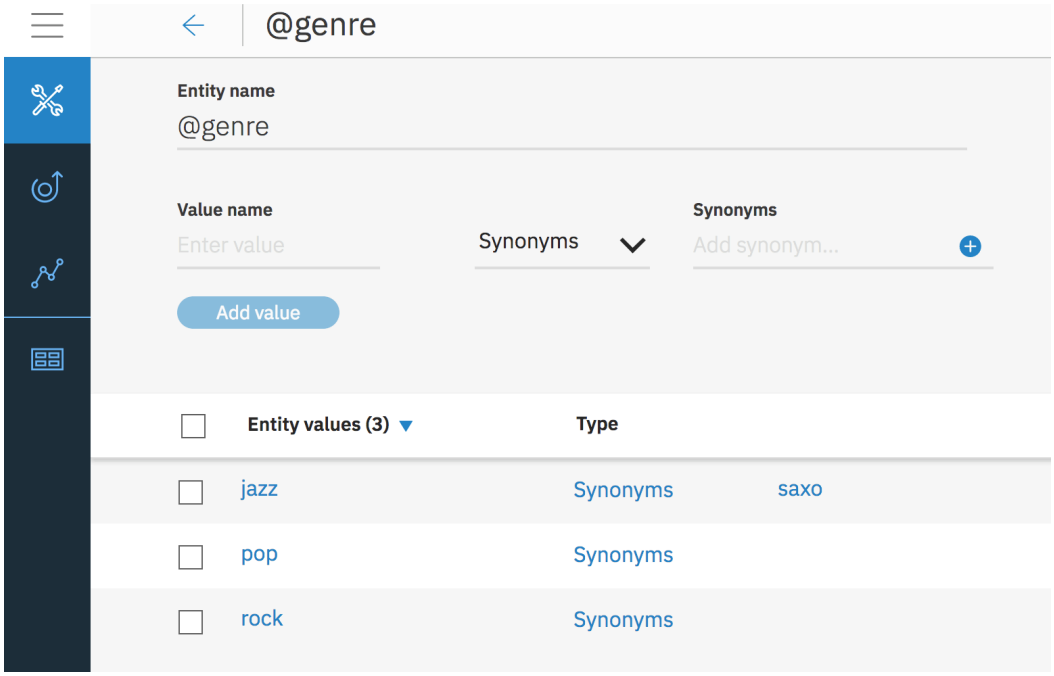


__ 34. Test the dialog and ask to **turn off the ac.** Expect to see *OK, turning off the air conditioner*.

Now let's ask the user for a preferred genre of music. This is a special case and we want to handle it differently. Let's say that we have jazz, rock and pop as music genre. Let's go back to Entities.

__ 35. Click the **Entities** tab.

__ 36. Add a new Entity, name it genre and add values to it: **jazz, pop** and **rock**; specify some synonyms, for example, *saxo* for jazz.



__ 37. Click on the **blue arrow** to go back to the workspace and go back to the **Dialog** tab.

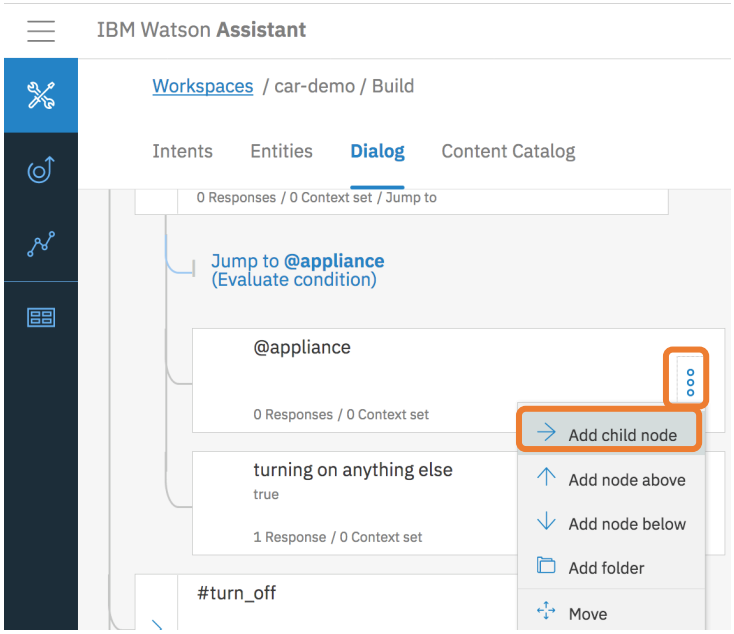
__ 38. Click **Try it** to test with the changes. Notice that the chat is training on the new changes. It will turn green shortly.

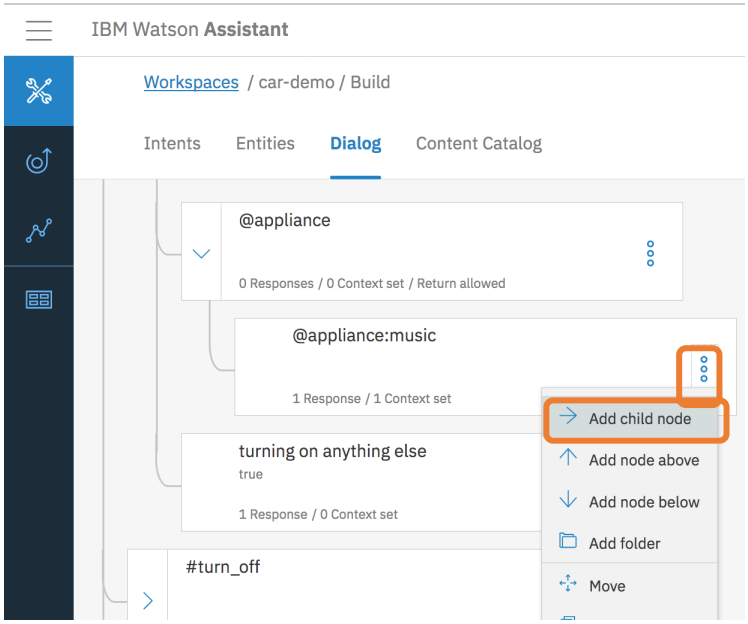
Yeap, this is neural nets, deep learning and the name of the algorithm is back propagation. The fact that it takes a few seconds to build patterns and use models (including Support Vector Machines) on a distributed scale is cutting edge technology.

Since you know that you want to turn something on and that is an appliance (music), then let's begin from the *turn_on->appliance* node.

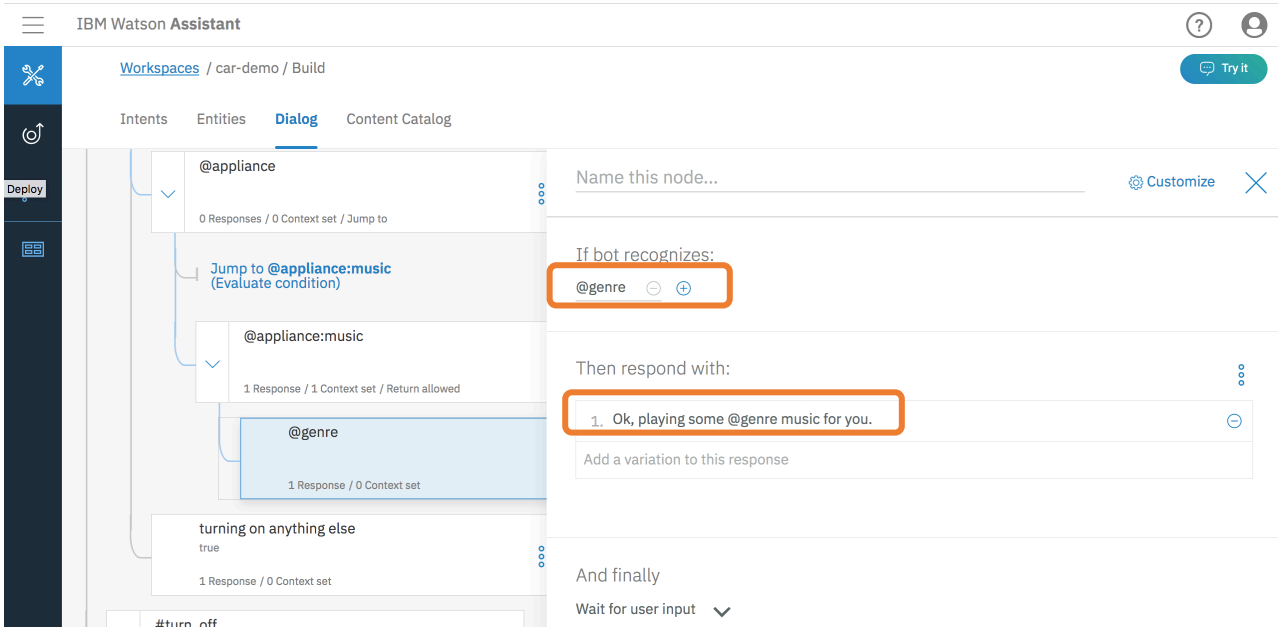
__ 39. From the turn_on -> appliance node, add a child node and specify the condition of **@appliance:music**.

__ 40. And if the condition music is true, then add another child to specify the condition genre by adding a child node the *@appliance:music* node and type **@genre.**





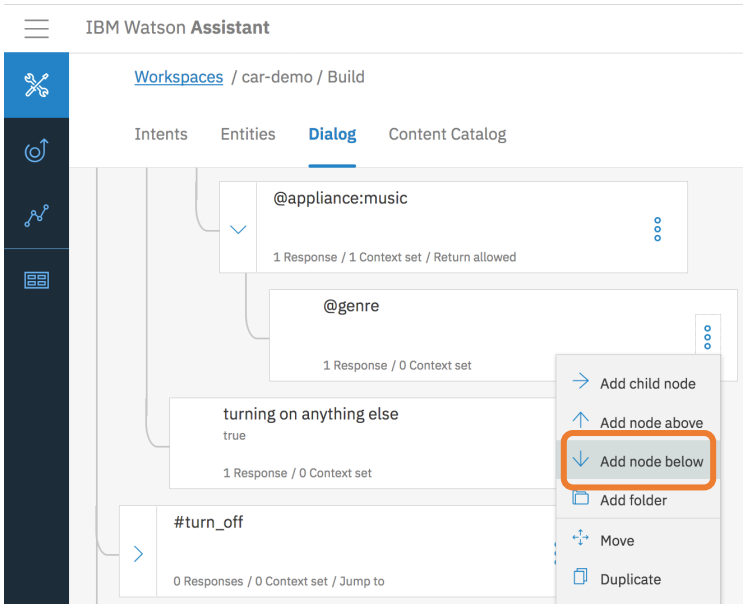
__ 41. Add the following response to the *genre* node: **OK, playing some @genre music for you.**

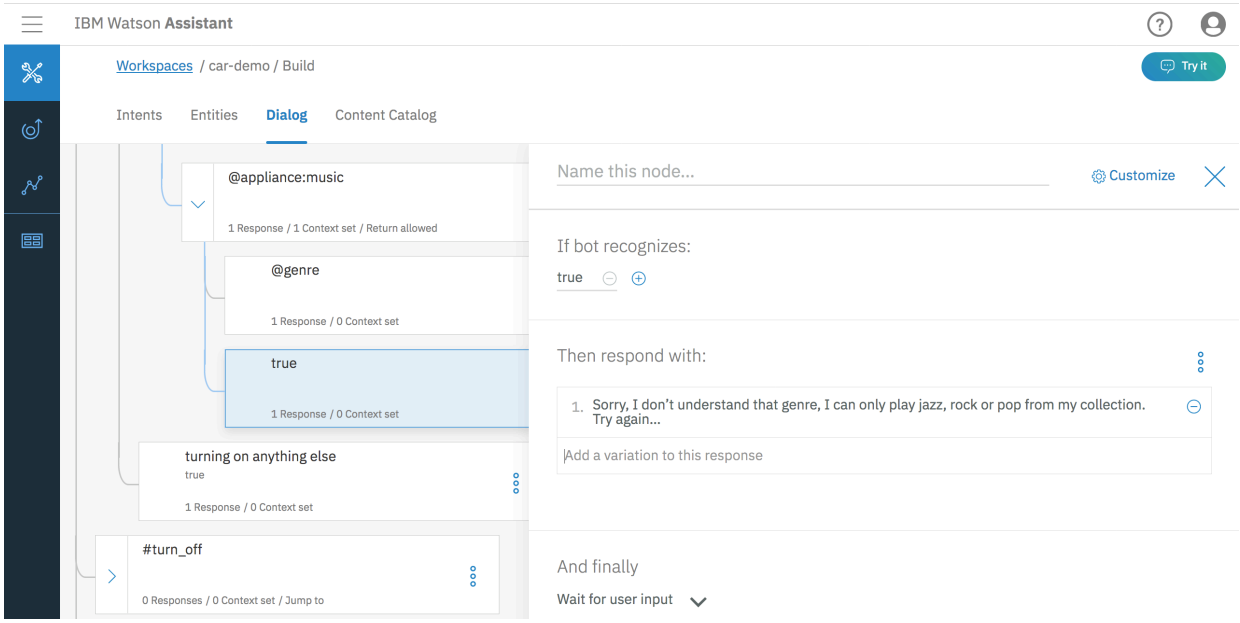


This is just a dummy message, but this where you use some JSON code to interact with the client app and play actual music of a certain genre from Spotify or Amazon Music and so forth.

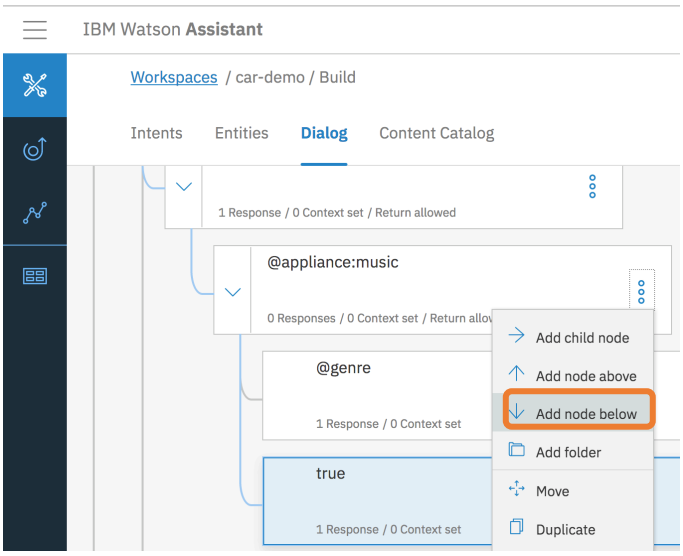
__ 42. Similar to Step 32, if the genre is not understood by the system and the condition is true, then you want to capture a poignant message here.

__ 43. Add a sibling node to the *genre* node, set the condition to true (type it in the condition section) and type the following in the response section: **Sorry, I don't understand that genre, I can only play jazz, rock or pop from my collection. Try again…**



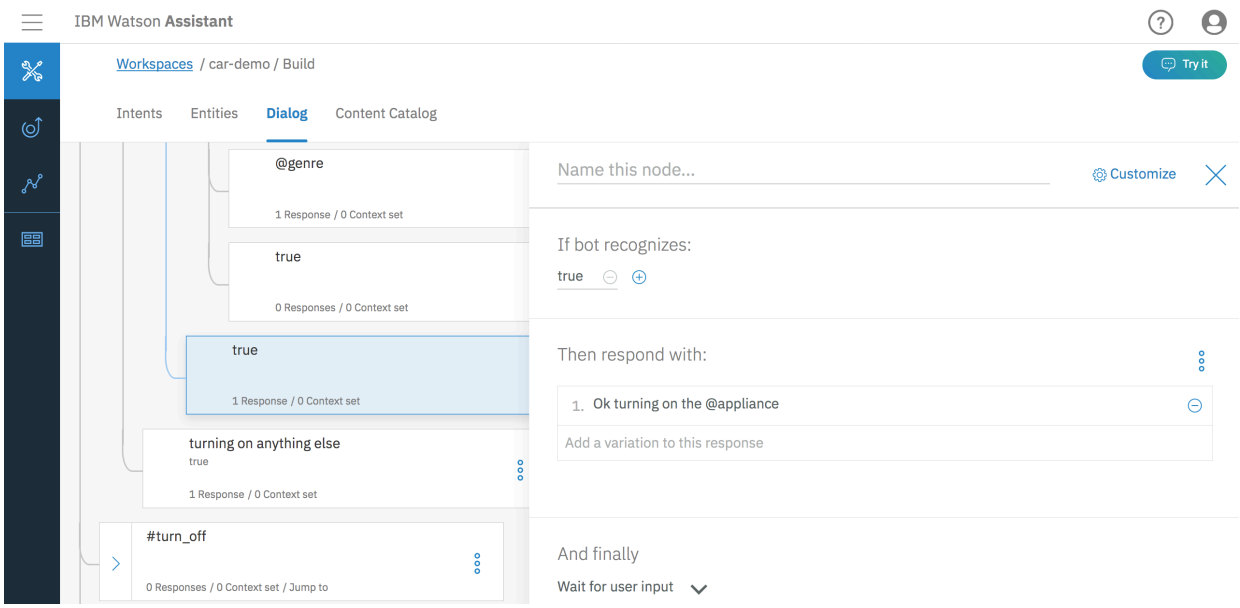


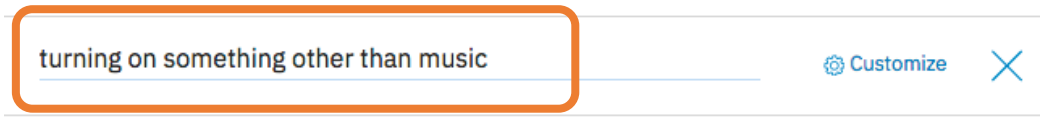
__ 44. Repeat the above step and create another **true** condition for the **appliance:music** node.



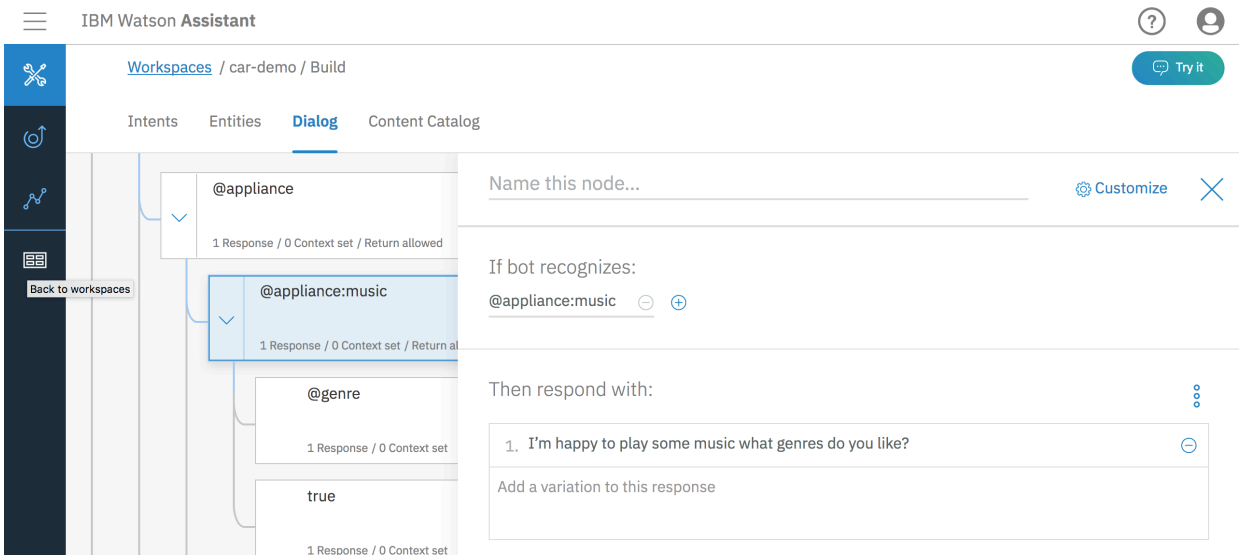
This means, that the appliance:music is true and the appliance is not music it reaches the *true* node you just created. Since the intent was to turn on something, (maybe you want to turn on the ac or the heater) the let's move the response from the appliance node to this **true** condition node.

__ 45. Cut (not copy) and paste the *response* from the appliance node to the **true** node that is a sibling of the **appliance:music** node. And give it a name like this:

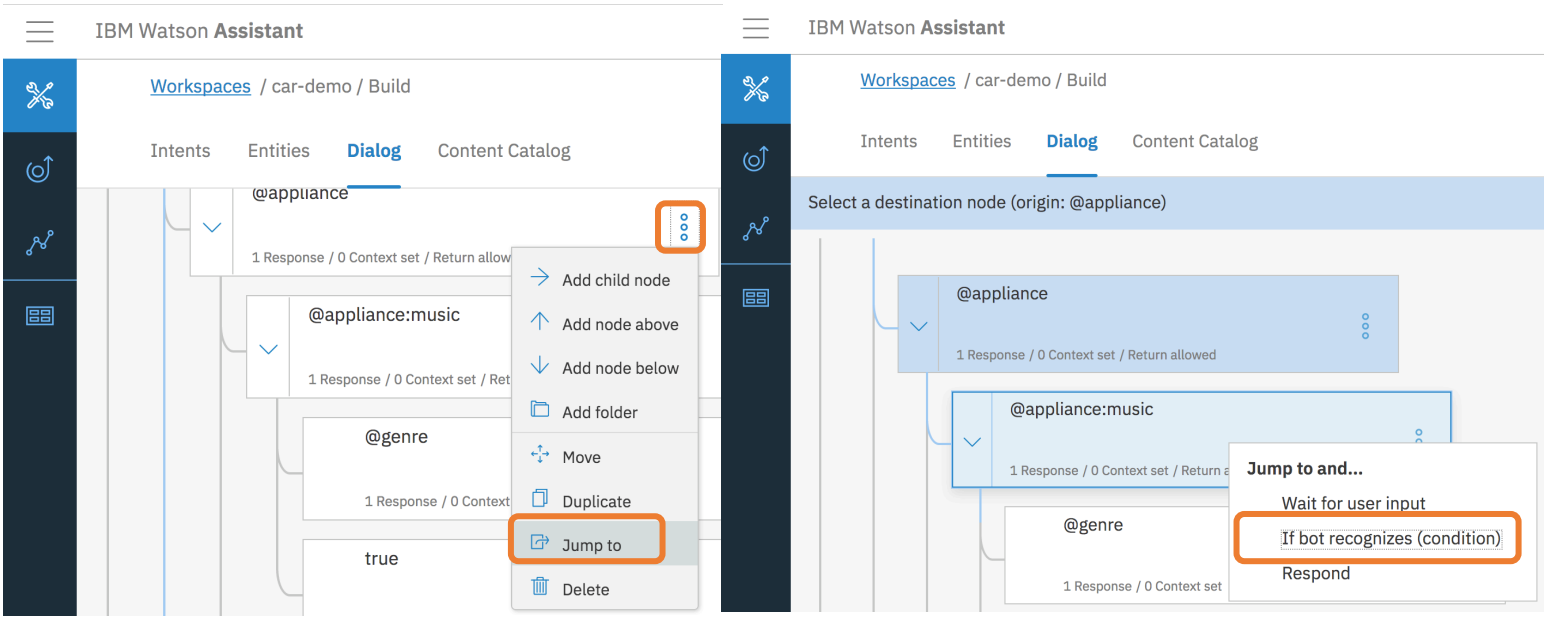




__ 46. And if the appliance:music is true and the system understood (that is, we asked for the correct genre), then the system response for the appliance music node is: **I'm happy to play some music what genres do you like?**

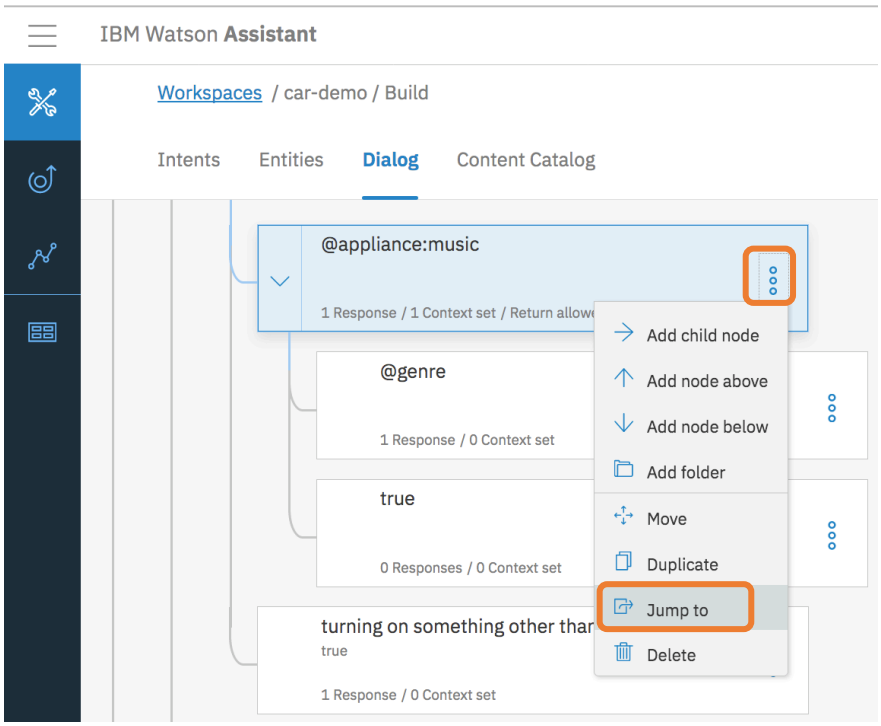


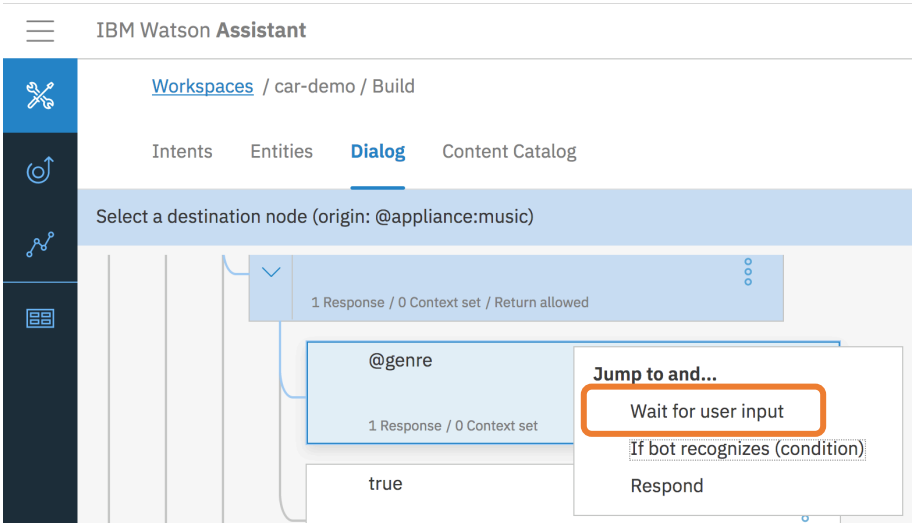
__ 47. From the appliance node, click the three vertical dots and add the **jump to** the *@appliance:music* node and click on **condition**.



This is so once the appliance is captured we want to present a prompt to the user. Jump from where you ask?

__ 48. Click the three vertical dots from the appliance:music node and select the **jump to** the *genre* node but click **wait for user input**.





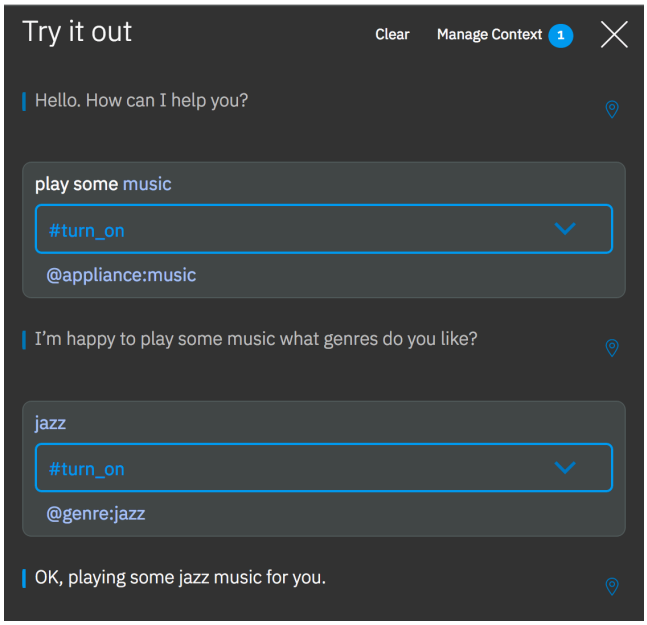
This is because we know there was an appliance and we want to condition on what type of appliance was asked by the user and also capture the answer of the user after the prompt was presented.

Let's test our work.

__ 49. Click **Try it** button and ask the system to: **play some music.**

It understood our intent of wanting to turn on an appliance and that appliance is music. It is going to prompt us for specific genre of music.

__ 50. Type: **jazz**



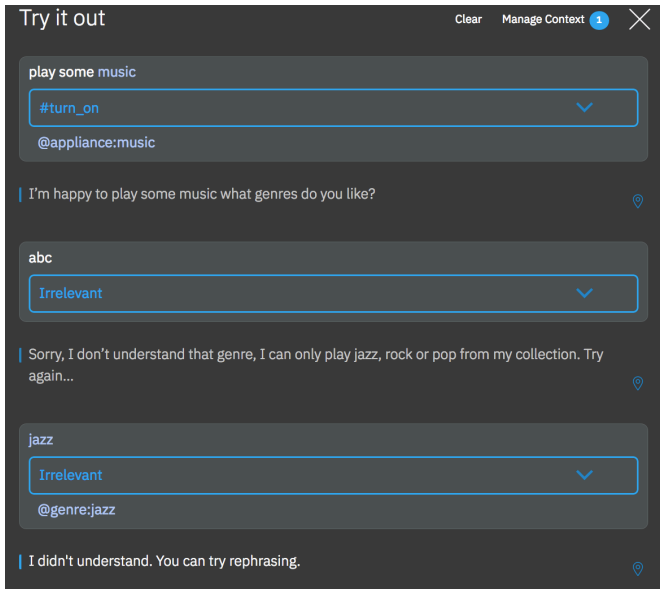
And so it does, well, at least it says it would.

Let's try a slightly different flow.

__ 51. Type: **play some music.**

__ 52. For subsequent response, type: **abc**

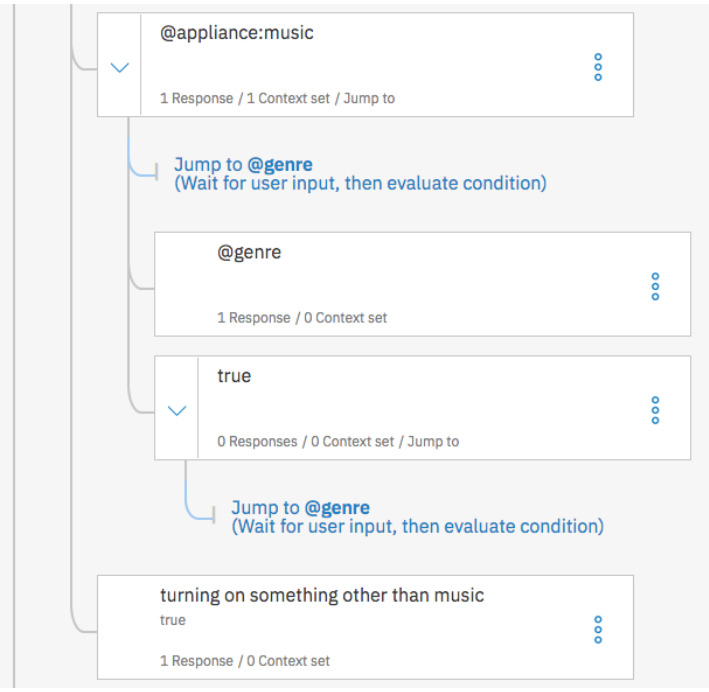
__ 53. Since *abc* is not in its collection, then type: **jazz**



Notice, it got nothing right and it lost the context. It went to the leaf node genre -> *true* and by default it went back to the root. So, you want to include a **Jump to** (from the *true* leaf node back to the *genre* node) and you want it to continue from a *user input*.

__ 54. From the *true* node sibling of *genre* node, click the three vertical dots and select **Jump to,** click on the *genre* node and click **wait for user input.**

You want to continue from a user input. This means that the system has prompted you for user input and the *jump to* user input means that the system wants to come back for additional user input and condition the dialog at this granular level not from the root node.

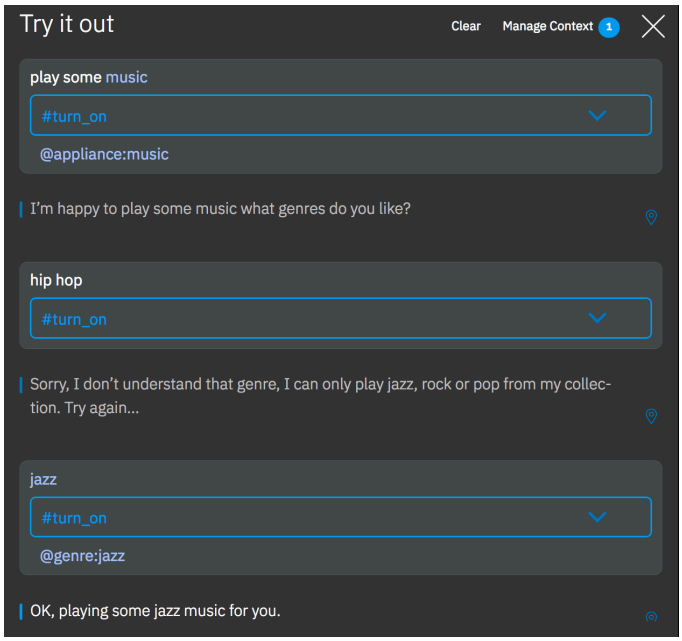


Test the chat flow with similar dialog:

__ 55. Type: **play some music**

__ 56. Type: **hip hop**

__ 57. Type: **jazz**



Notice that it did not lose the context of the conversation and did not go all the way back to the *turn_on* condition (node) and get stuck there.

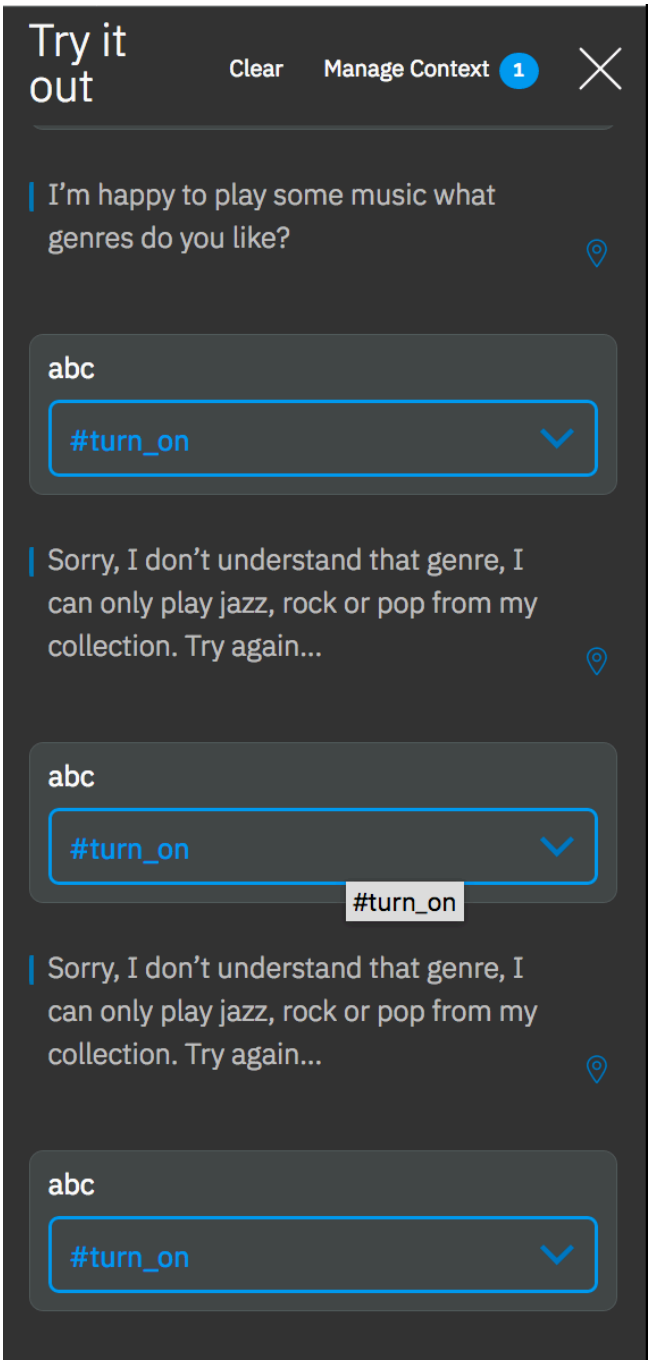
Now let's explore another user input:

__ 58. Clear the chat window

__ 59. Type: **play music**

__ 60. Type: **abc**

__ 61. Again, type: **abc**

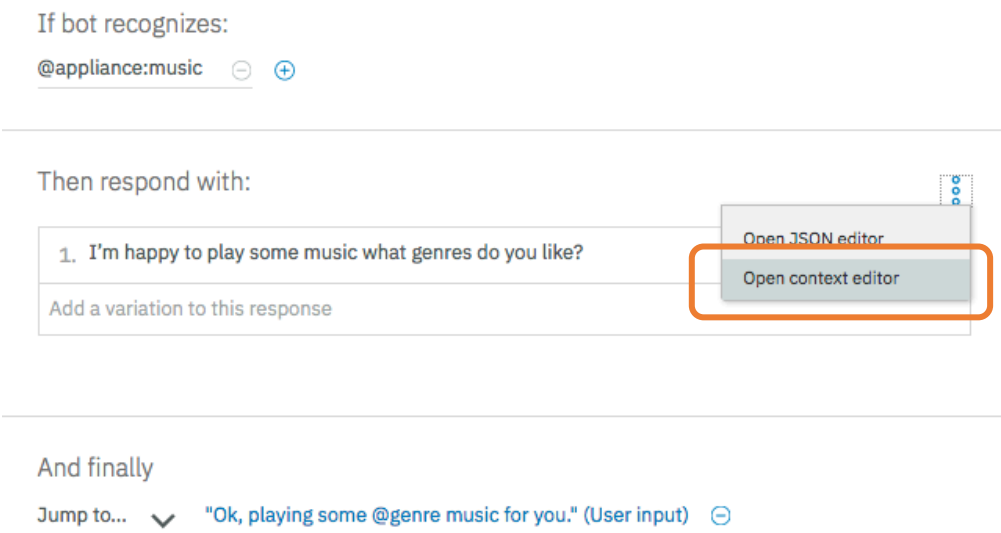


Notice that it is stuck in a loop. Let's fix that. The best practice to remove this loop is to add a context variable.

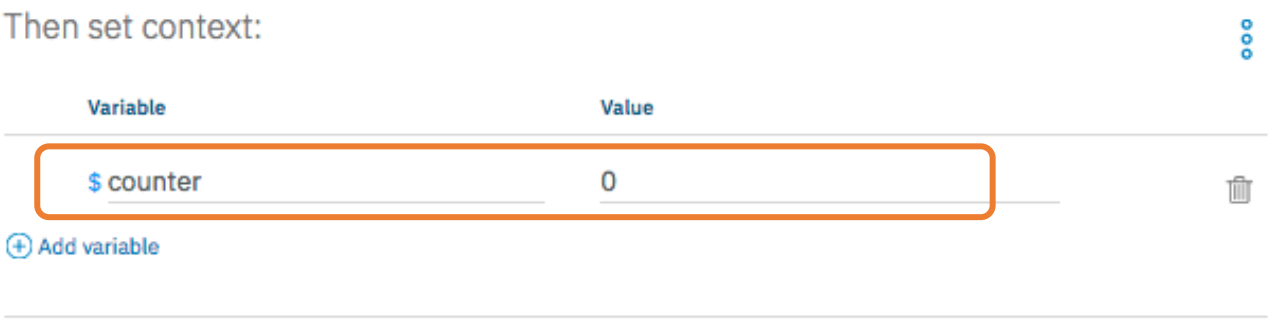
We are going to include a context variable that measures how often does the genre -> **true** node variable gets hit.

We want to initialize the **true** node one level higher before it is called by the condition.

__ 62. Click on the *appliance:music* node (this is the one level up from the *true* node to its lower right) to modify it, click the three vertical dots in the response section and select **Open context editor**.



__ 63. Add a context variable in the context editor and instantiate it to 0 :



And now you must update the context response part in the *true* node.

__ 64. Open the *true* node, click on the three vertical dots in the response section and select **Open context editor**.

__ 65. Change the condition section of this *true* node to $counter<1 and edit the context to look like this:

The “<?$counter+1?>” is the syntax inside a JSON editor where you want to evaluate some expression. This means inside the context variable, set the variable to counter plus one.

This means, the first time the condition hits the appliance:music node it will set the variable to zero and the subsequent times it will increment it by one. We want to have a count of how many times we hit the genre node with same “not true” response so it does not get stuck in a loop… it has a count of our hits; less than one means if I type genre music that is not in the collection over and over again, then do not fall for my trap again and again.

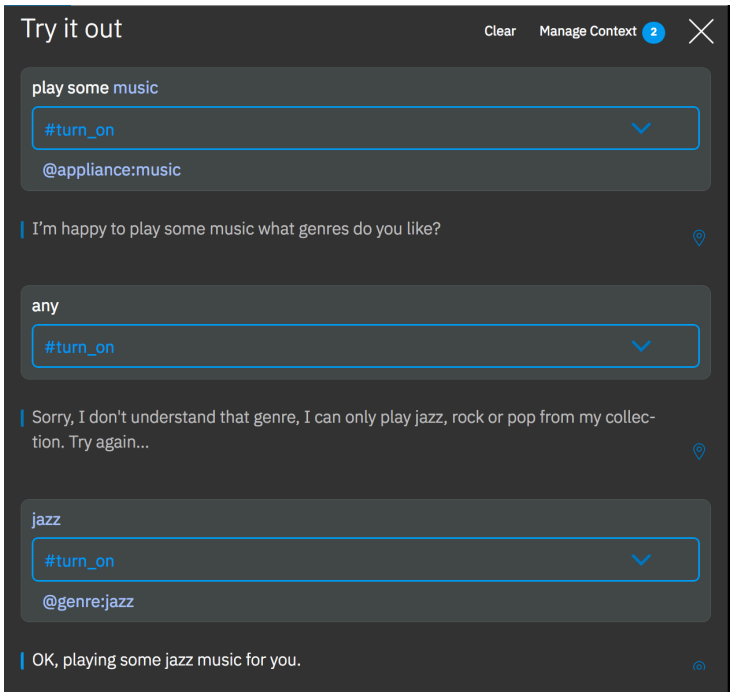
Let's test the dialog flow again:

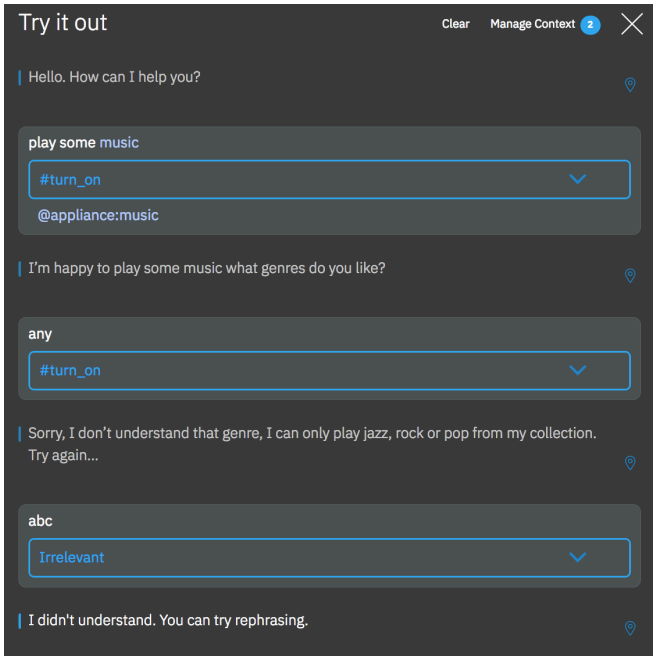
__ 66. Type: **play some music**

__ 67. Type: **any**

__ 68. Type: **jazz**

OK so that was normal operation, now let's make it fall into the loop trap by keep typing irrelevant genre of music.





__ 69. Clear the chat and type: **play music**

__ 70. Type: **any**

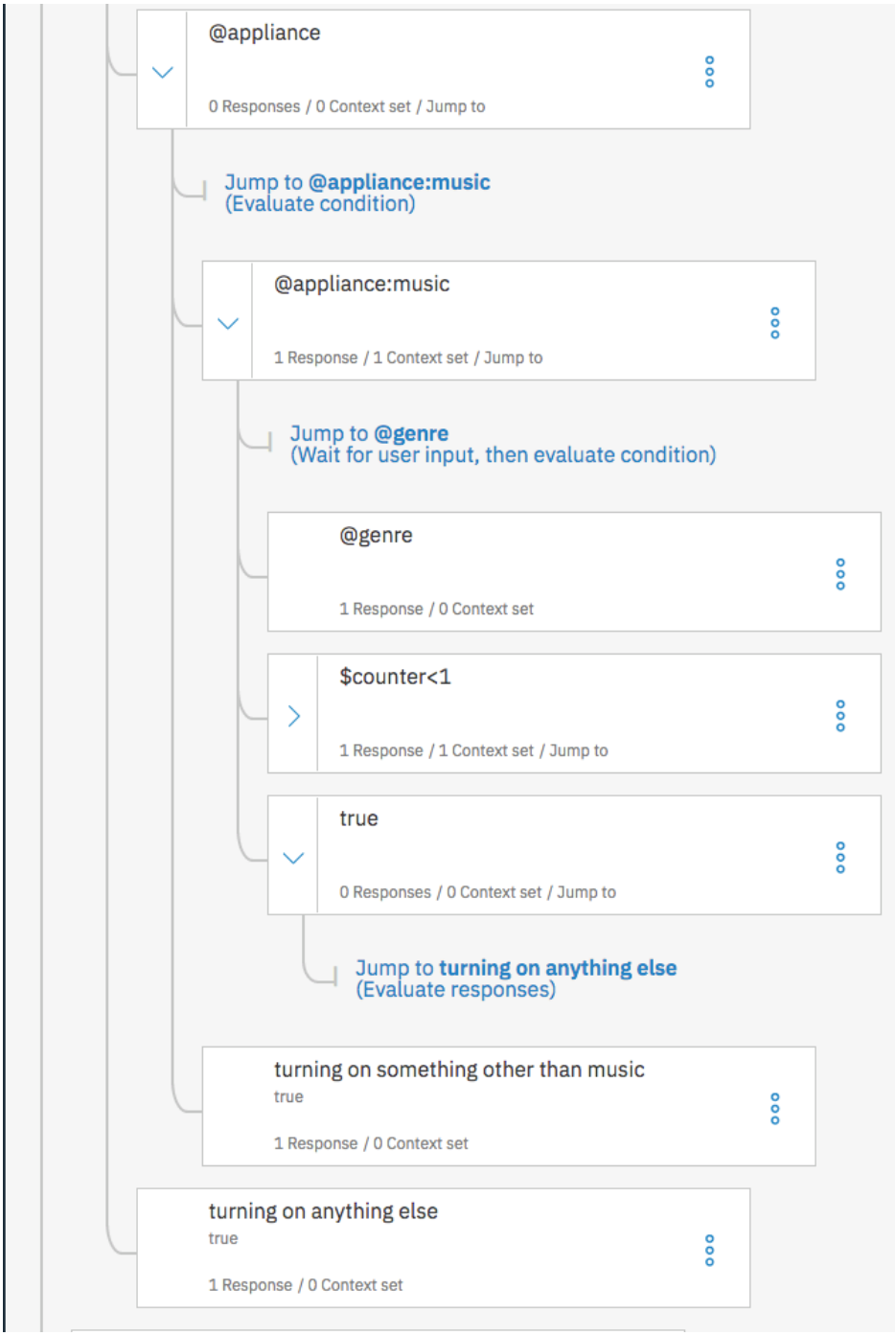
__ 71. Type: **abc**

Notice that it now goes back to root, but it is not what we want to do. We want to say to the user that we understand that he's trying to turn on something but he's not doing it well, so it should go back to the *true* node, sibling the *@appliance* node.

To do that, you must add a new **true child node** to the *@appliance:music* node.

__ 72. Add a **child node** to the *@appliance:music* node and type **true** in the condition section.

__ 73. Click the three vertical dots of this new *true* node, select **Jump to…;** click on the *turning on anything else* node, sibling of the *@appliance* node and select **Respond.**

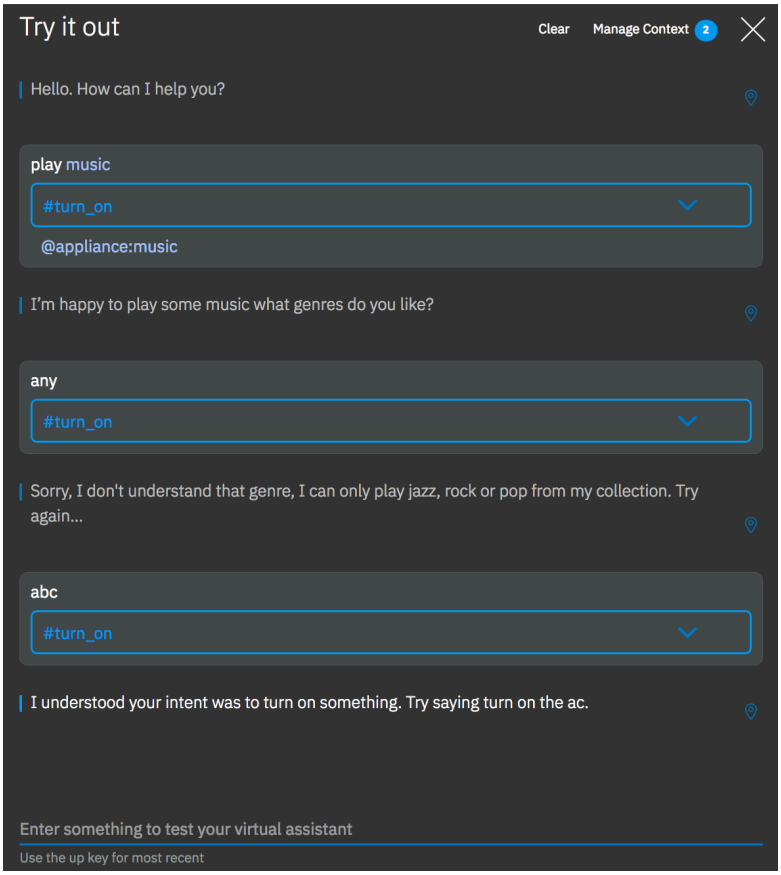


Let's now test the dialog again:

__ 74. Clear the chat and type: **play music**

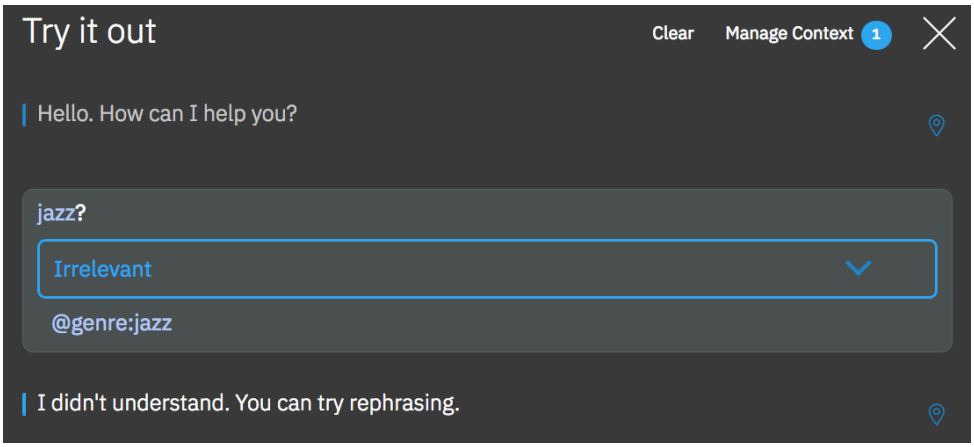
__ 75. Type: **any**

__ 76. Type: **abc**



No more loop, right? Otherwise, check your JSON code inside the response sections of both nodes you edited. Usually the biggest gotchas are the curvy double quotes.

__ 77. Let's try one more scenario. Clear the chat window and at the very start of the conversation type **jazz?**.



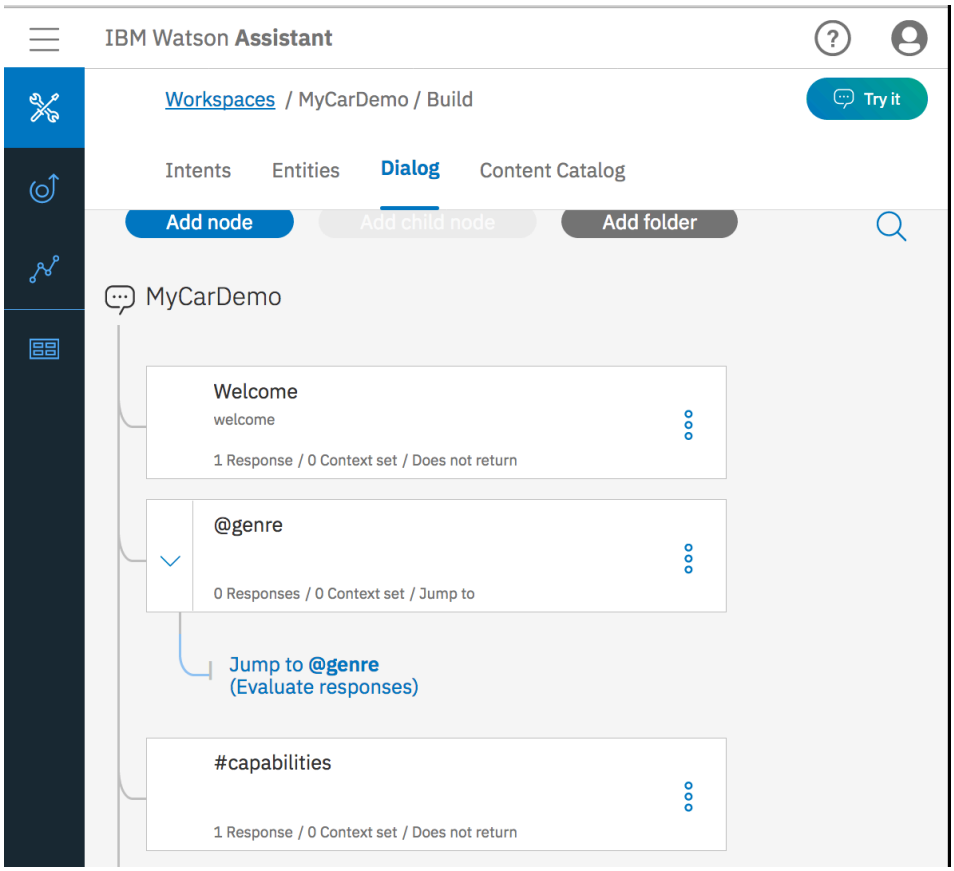
Notice that it did not understand the classification (Intent) but got the correct entity, then it does not know where to go yet and what if you want it to play jazz from your first response.

In other words, can you start the conversation by conditioning on an entity instead of intent first? The answer is YES!

__ 78. Create a sibling node underneath the **Welcome** node and set the condition to the entity by typing **@genre.**

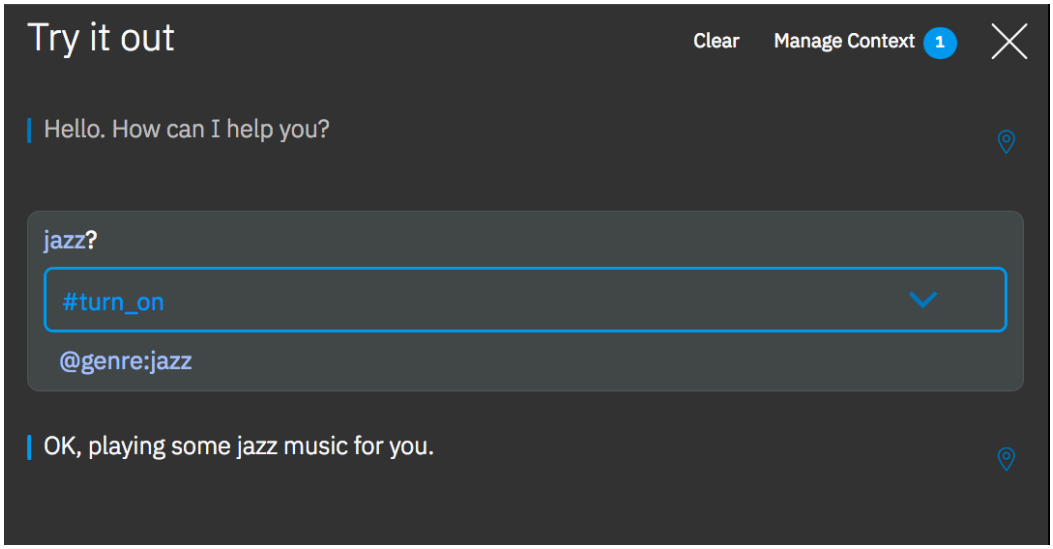
__ 79. Click the three vertical dots of this new node, select **Jump to;** click on the other *@genre* node all the way from the *turn_on* node and select **Respond.**

Because you want to display the "OK, playing some @genre music for you" from the very start.



Test it.

__ 80. Clear the chat window and from the start of the conversation, type: **jazz?**



Notice that it jumps straight to the entity (it is conditioned on entity instead of intent).

Congratulate yourself. In 81 steps you performed some advanced dialog creation techniques.

And here is the answer to a burning question that you may have: How do I handle chitchat: greetings and good byes and off topic conversation?

The best way to do that is to have a separate workspace that just understands chitchat and understands related intents and responds accordingly and then have the client app to talk to both workspaces; if the main workspace is not able to answer the question, then it can point to the adjacent workspace.