Lab 1: Create an instance of Watson Assistant

After completing this lab, you will be able to:

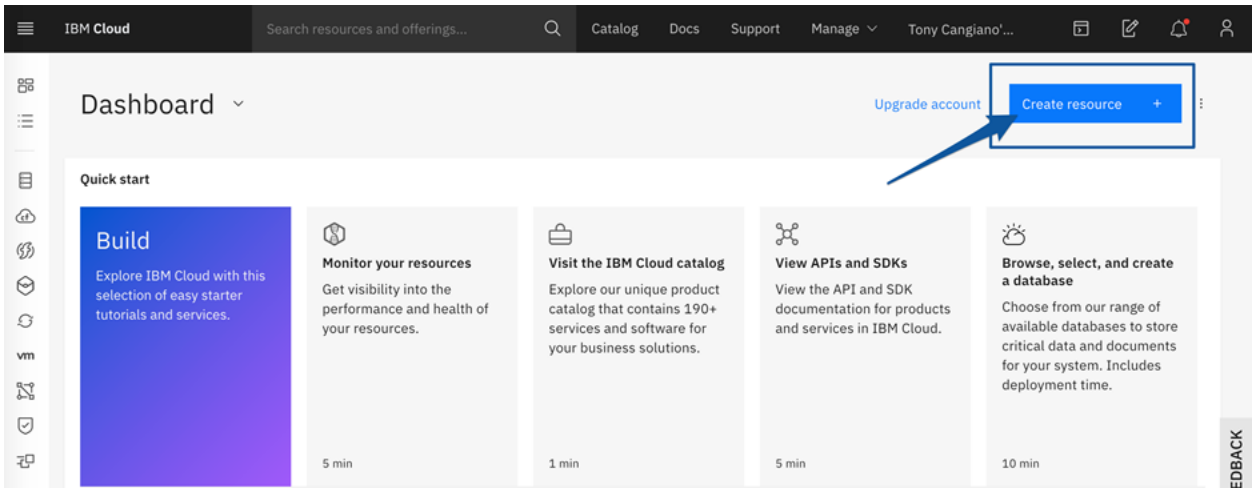
- Create Watson Assistant service.
- Use the Watson Assistant service hosted on the IBM Cloud platform.
- Create a dialog skill for the Watson Assistant

Pre-requisites

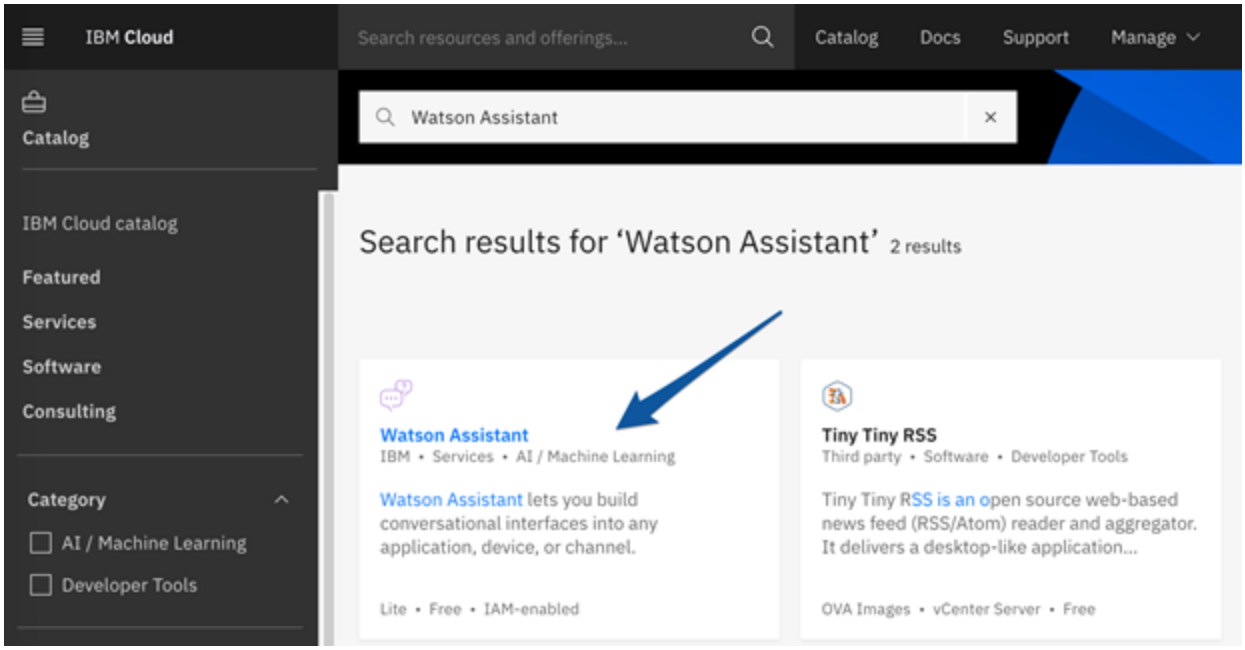
You will need an IBM Cloud account to do this lab. If you have not created one already, click on this link and follow the instructions to create an IBM Cloud account.

Exercise 1: Create a Watson Assistant service

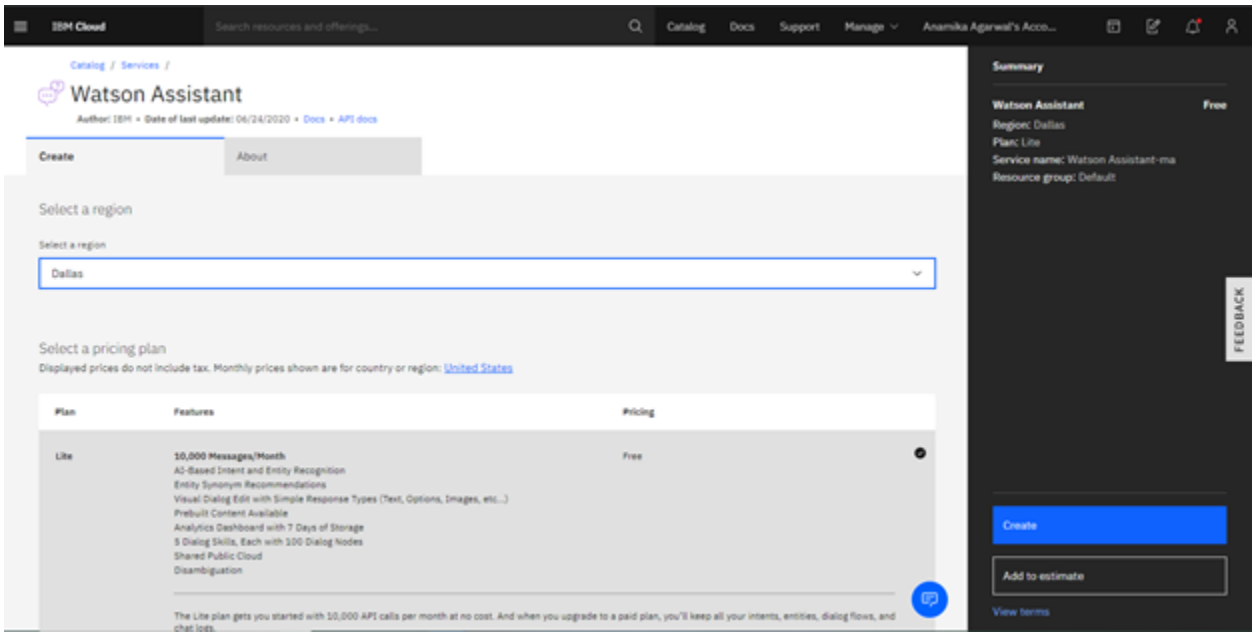
1. Visit https://cloud.ibm.com and log in with your IBM Cloud email and password.
2. You'll find yourself in your IBM Cloud dashboard. Click on the Create resource button on your dashboard, as shown in the picture below.



3. Enter Watson Assistant in the search field and press enter.
4. Now click on the Watson Assistant tile that appears, as shown in the image below.

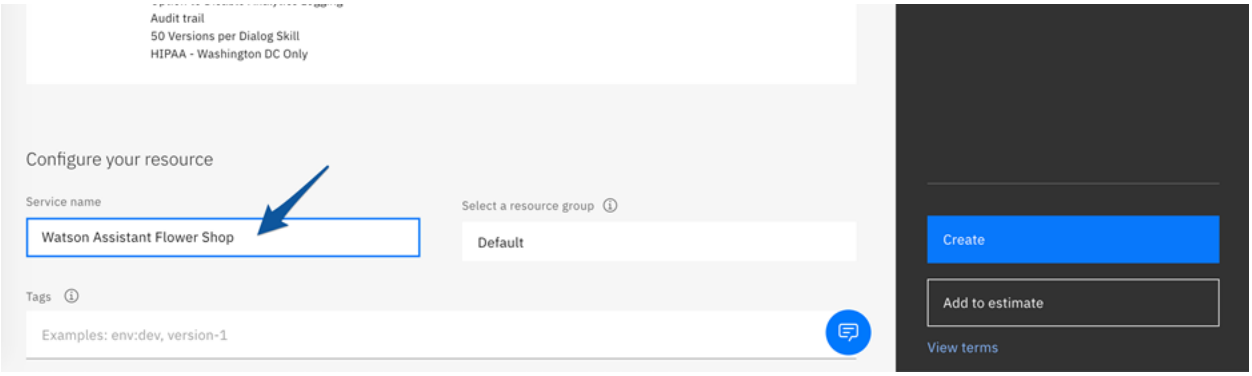


5. You should see a Watson Assistant creation page similar to the image below.

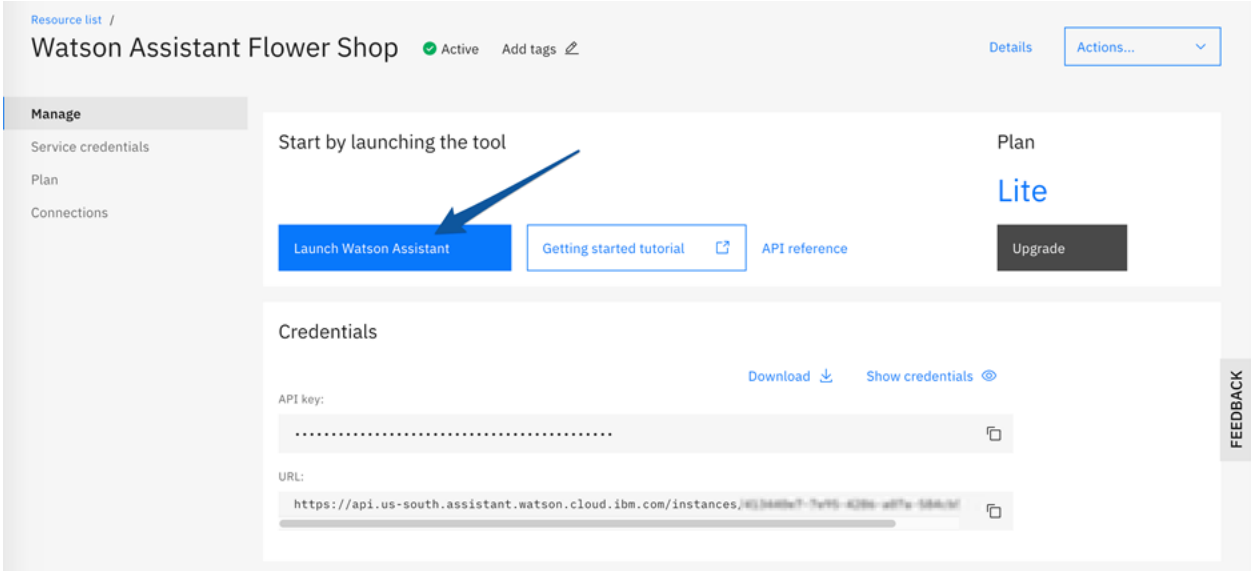


Click on the region drop down to select a data center closer to you. For example, you might select Frankfurt if you live in Europe. This will reduce latency and improve performance as you use Watson Assistant.

6. Scroll down the page and change the instance name to your liking (e.g., Watson Assistant Flower Shop), as shown in the image below.



7. Click on the Create button to create your instance.
8. You'll be redirected to the launch page for the service you just created. Click on the Launch Watson Assistant button to access the web application that will allow you to create chatbots.

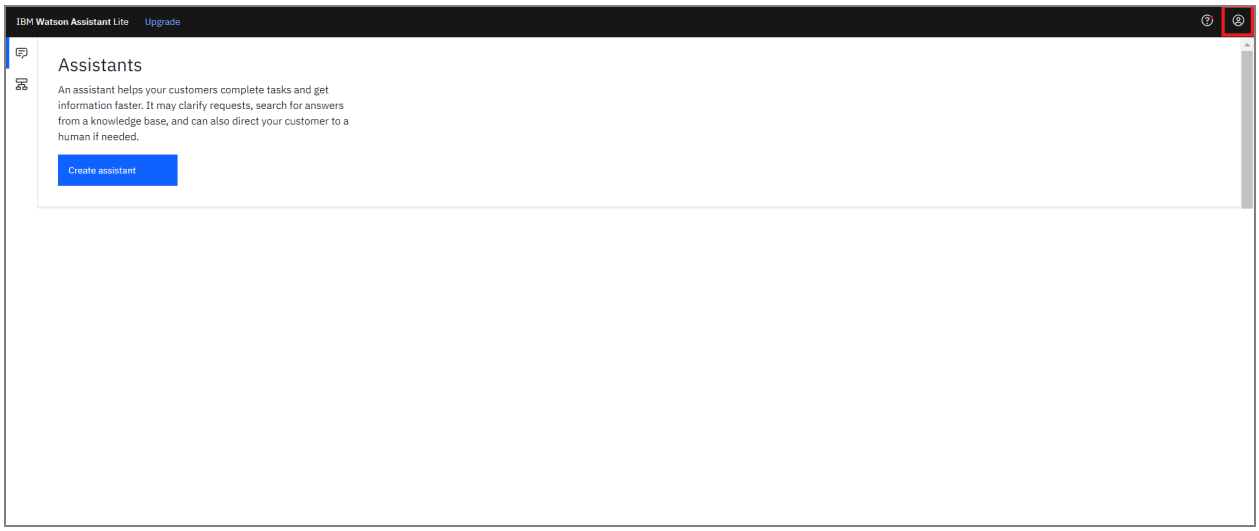


Congratulations on creating your instance of Watson Assistant!

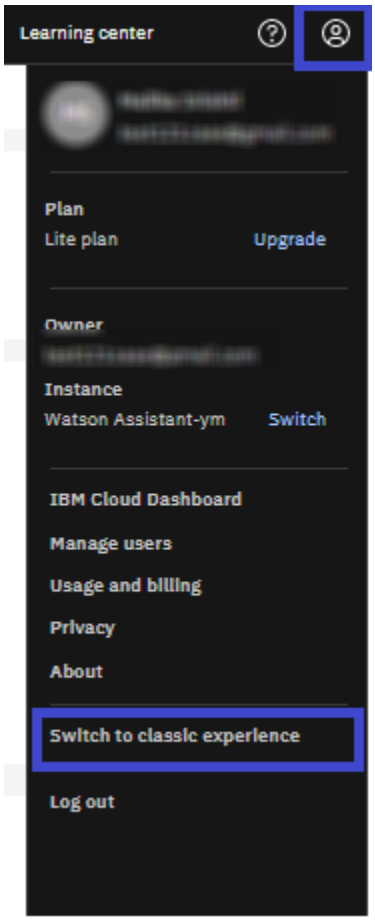
(If you see a survey asking you questions about whether you are a professional or a student, feel free to complete the survey or simply close that pop up.)

If your instance is using the New Watson Assistant, you can switch to the Classic experience by following these steps:

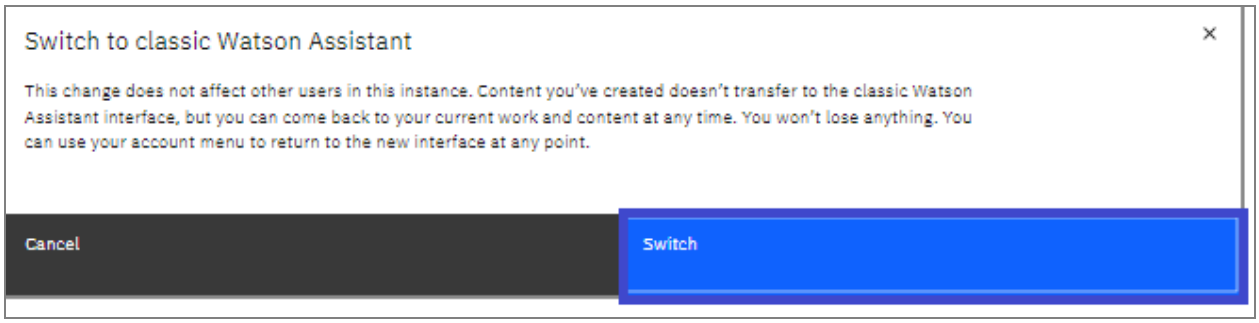
1. From the Watson Assistant interface, click the User Panel icon to open your account menu.



2. Select Switch to Classic experience from the account menu.



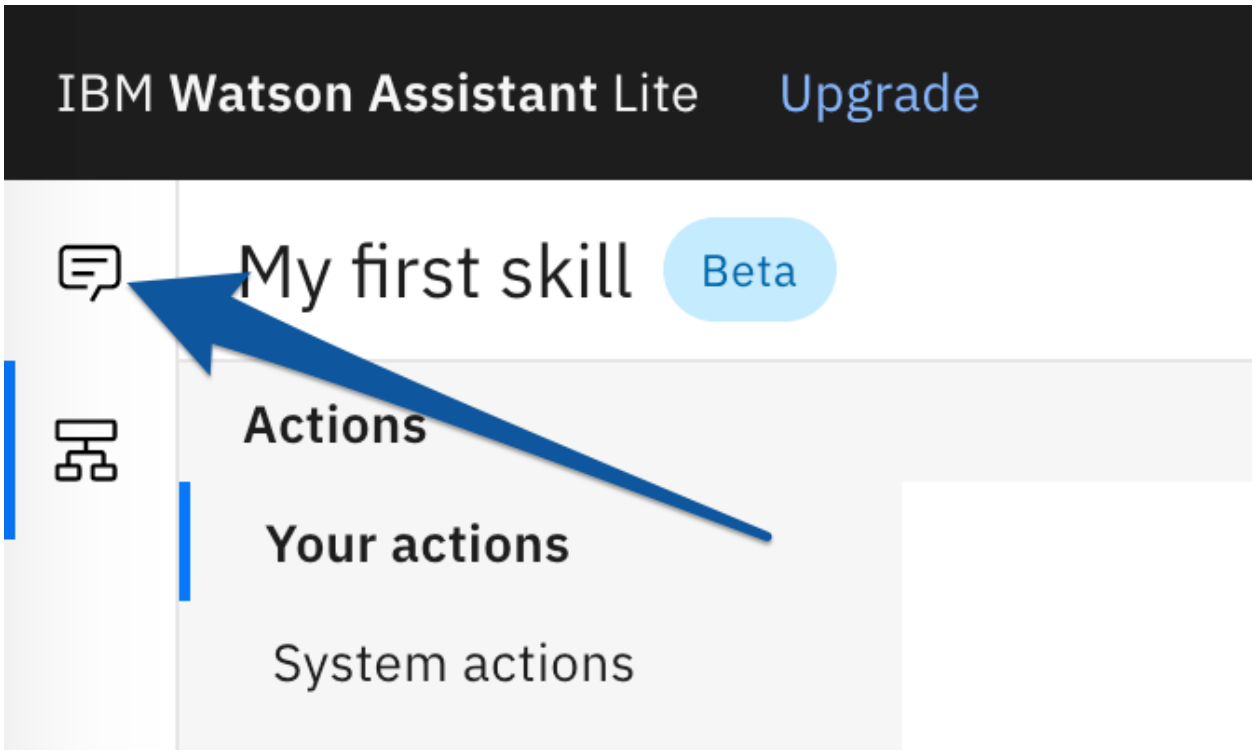
3. Click Switch to confirm that you want to start using the Classic experience.



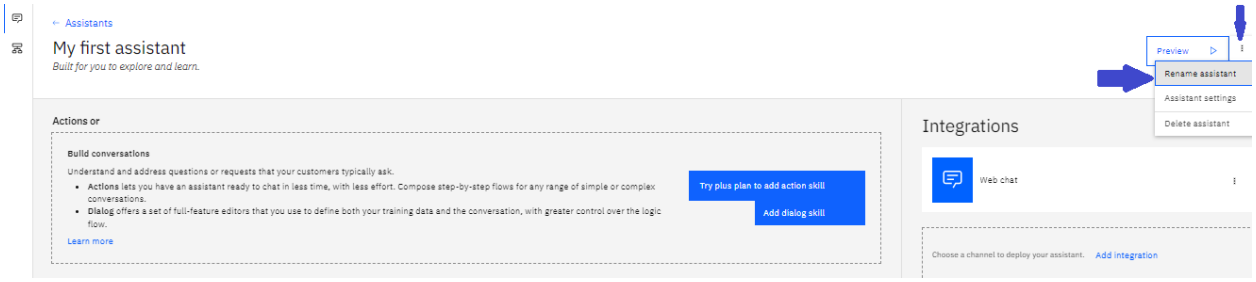
Note: You won't lose any work if you switch to the classic experience, and you can switch back to the new experience at any time by clicking Switch to new experience from the account menu.

Exercise 2: Creating a dialog skill

1. Click on the Assistants icon in the top left corner as shown in figure.



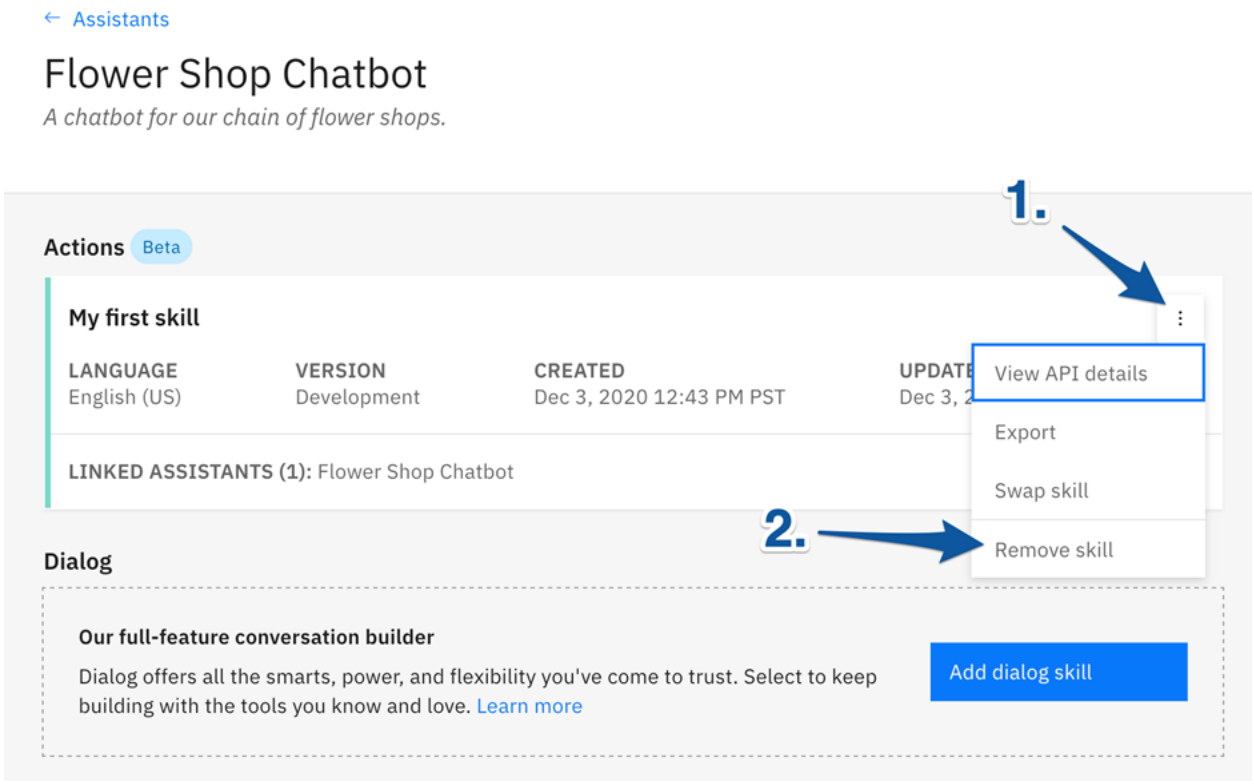
2. A default assistant has been created for you. Click on My first assistant .
3. Click on the three vertical dot icon to the right of the page, above *Integrations* and then click on Settings .



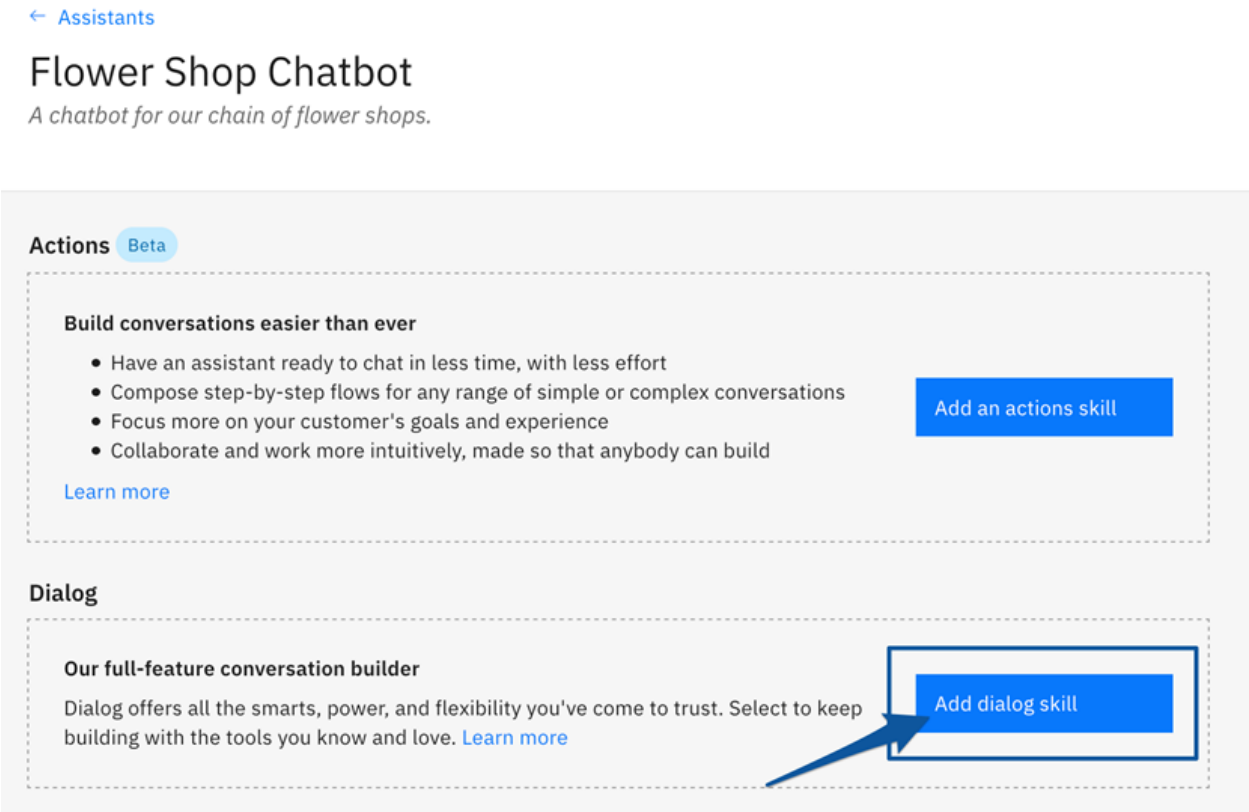
4. Here, change the name of our assistant to Flower Shop Chatbot and optionally change the description too. Once you're done, click on the X at the top to return to our assistant.

5. You can think of the assistant as the actual chatbot. And a chatbot will have one or more skills. Typically, a chatbot will have at least one dialog skill. We'll create one and link it to our assistant, but first, let's remove *My first skill* under *Actions*. (This is a type of skill that is still in beta and we can safely ignore it for now. Once this feature becomes generally available, I will update this course to take full advantage of it.)

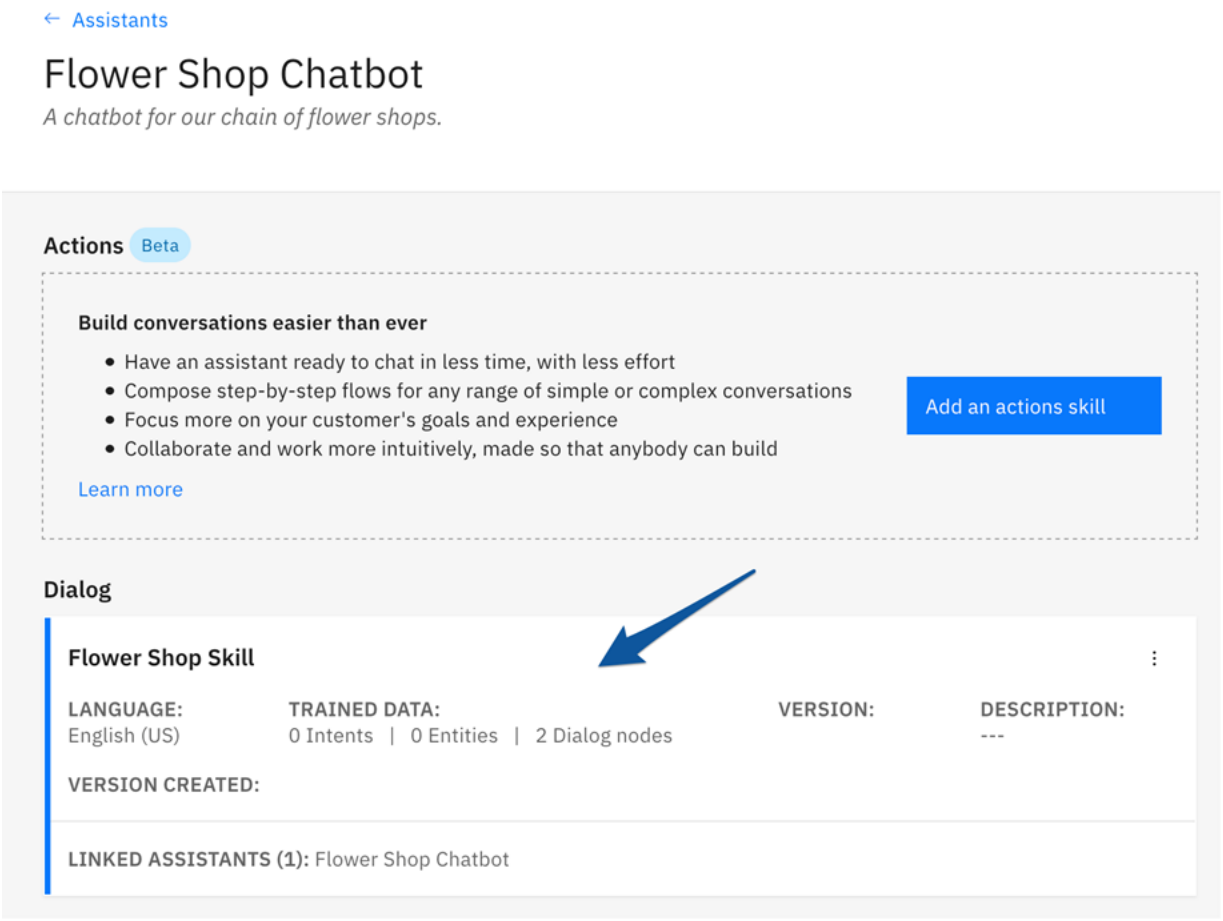
To do so, click on the three vertical dots icon on the right of My first skill , and then click Remove skill. A pop up will appear asking you to confirm. Go ahead and click Remove, confirming your intention to remove the skill from the assistant.



6. Now, click Add dialog skill.



7. Enter Flower Shop Skill and click Create dialog skill. Optionally add a description if you wish.



Watson doesn't really care about these labels and descriptions, but they help humans working on the chatbot better understand how things are organized and why.

8. You'll now be redirected back to your Flower Shop Chatbot. Bookmark this page in your browser so as to quickly access it later on.
9. Finally, click on the Flower Shop Skill tile that was created within your assistant.

(If you see any survey or tour pop ups, feel free to dismiss them.)

Congratulations on completing this lab! 😊

Lab 2: Create Intents

Objective for Exercise:

- Adding intents to your chatbot

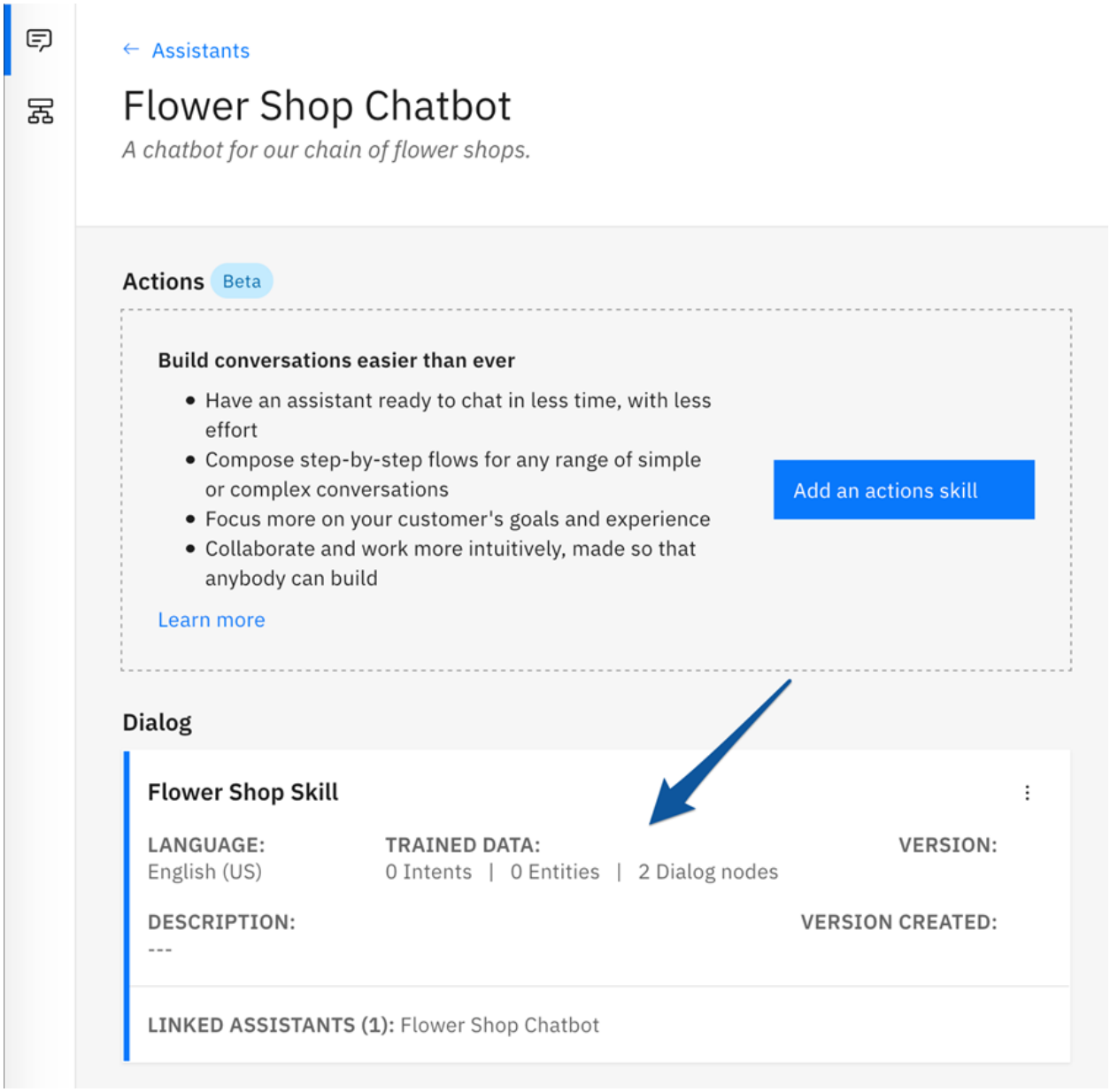
Lab 2: Create Intents

Now that we have an Assistant and its dialog skill, we are ready to start creating our chatbot. We'll start off with intents.

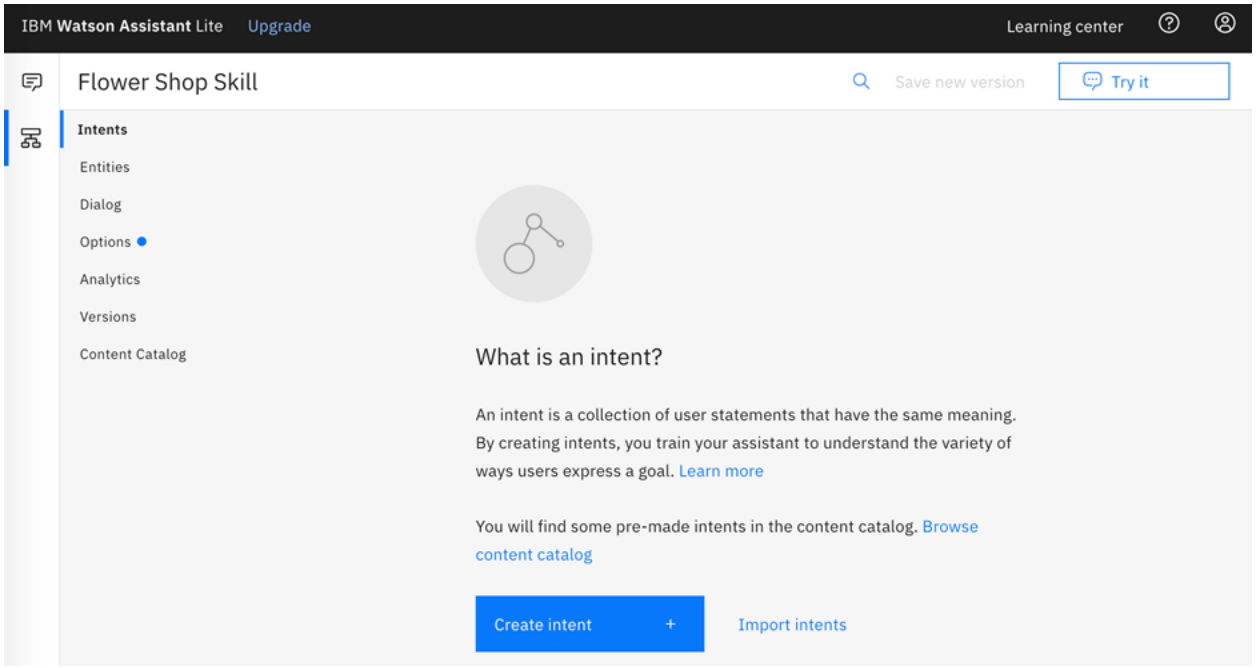
Exercise 1: Enter into your Flower Shop Skill

In order to add intents, we'll need to access our *Flower Shop Skill*.

1. Visit the _ Flower Shop Chatbot _ assistant page that you bookmarked in Lab 1. (If you skipped that step, you can always find your assistant by clicking on the top-left icon within Watson Assistant, which will list all of your assistants.)
2. Click on the _ Flower Shop Skill _ tile within the chatbot, as shown below.



This will get you inside of your skill (as shown in the image below), where you'll be able to create intents, entities, and much more.



By the way, if you don't know how to get back to the Watson Assistant interface altogether, you simply have to visit your dashboard (available at cloud.ibm.com), click on *Services*, and launch your Watson Assistant instance again.

Exercise 2: Create, train, and test intents

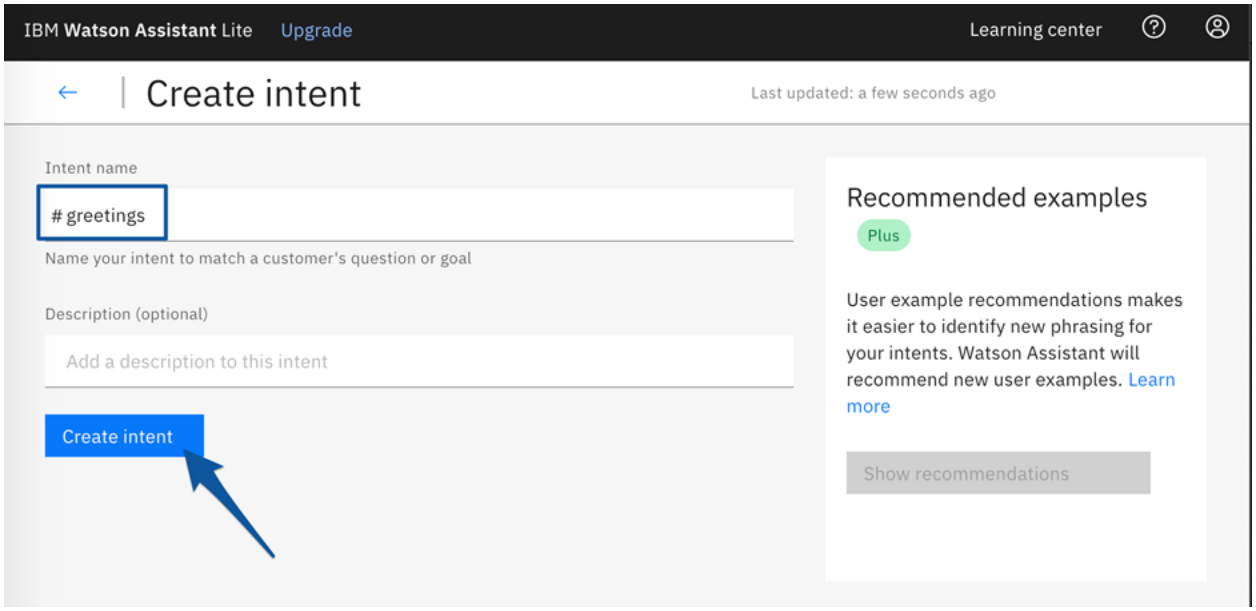
Now that we are inside of the *Intents* section of our skill, we can add intents in a few ways. Namely, we can add them manually, import them from a CSV file, or import them from a pre-made content catalog which includes industry-specific intents.

In this exercise, we'll focus on the most common way. That is, manually adding intents.

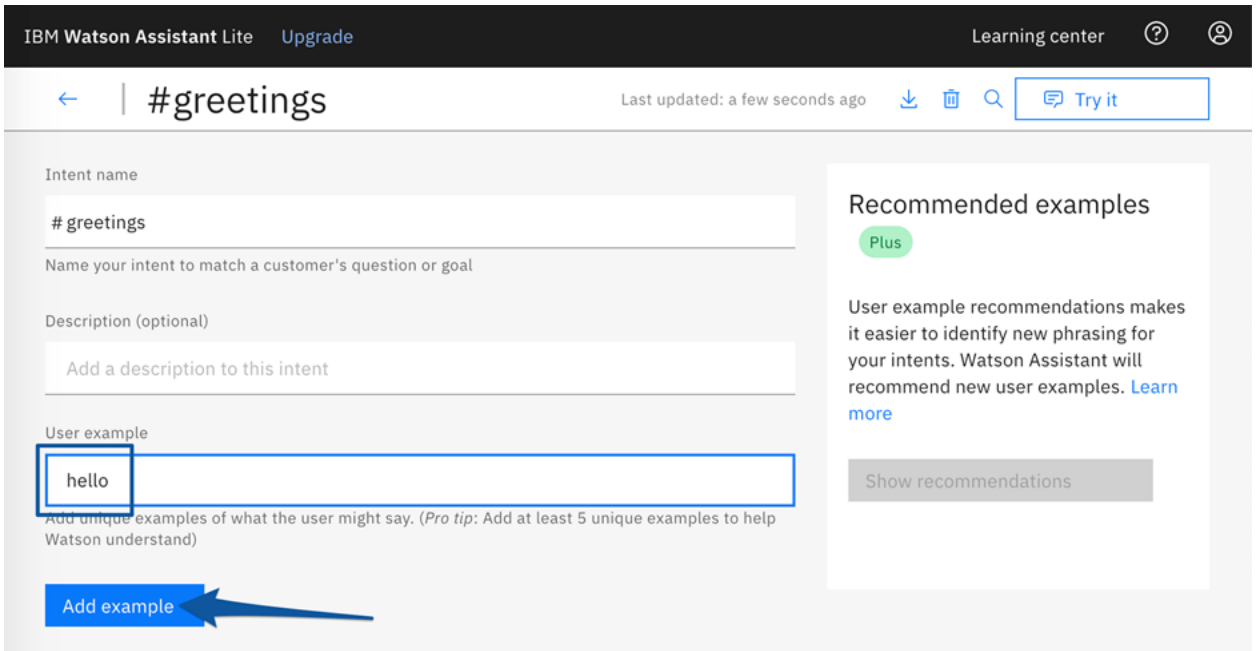
1. From the Intents section of your dialog skill, click on the Create *intent* button. Here you'll be able to define an Intent name and a Description. What happens if you try to call the intent #greeting us with a space in the name?

That's right! We can't use spaces in intent names. So, we either need to use a single word for our intent or add underscore characters to separate multiple words when these are necessary to express what our intent is all about. In our case, we can get away with just calling the intent #greetings. Let's do that.

2. Define a #greetings intent. You can leave the description blank, as the name is descriptive enough, and then click the Create intent button. (Note: The # symbol is automatically prefixed for you, enter greetings in the field, not #greetings.)



3. You'll be prompted to create some user examples to train Watson on the concept of greetings. Enter hello then click *Add example*.



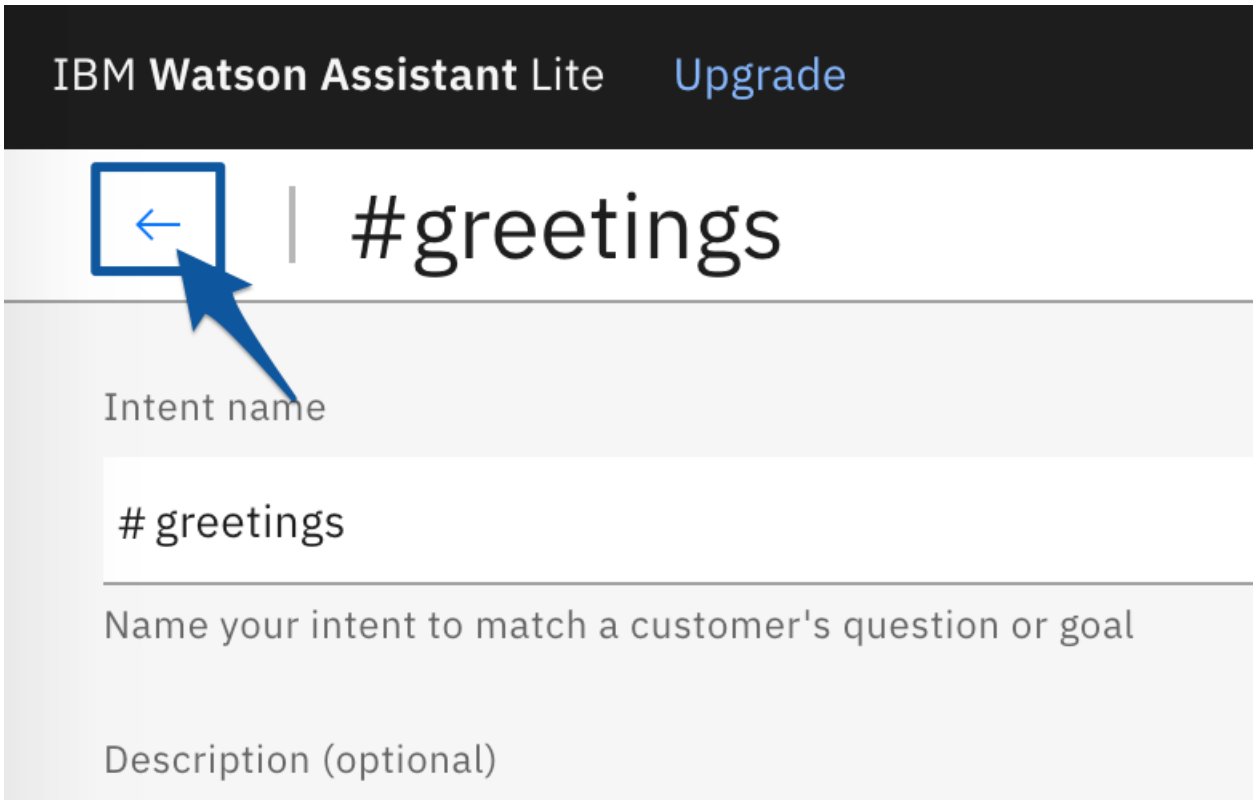
4. Repeat the process for other greeting examples. Add:

- Hi

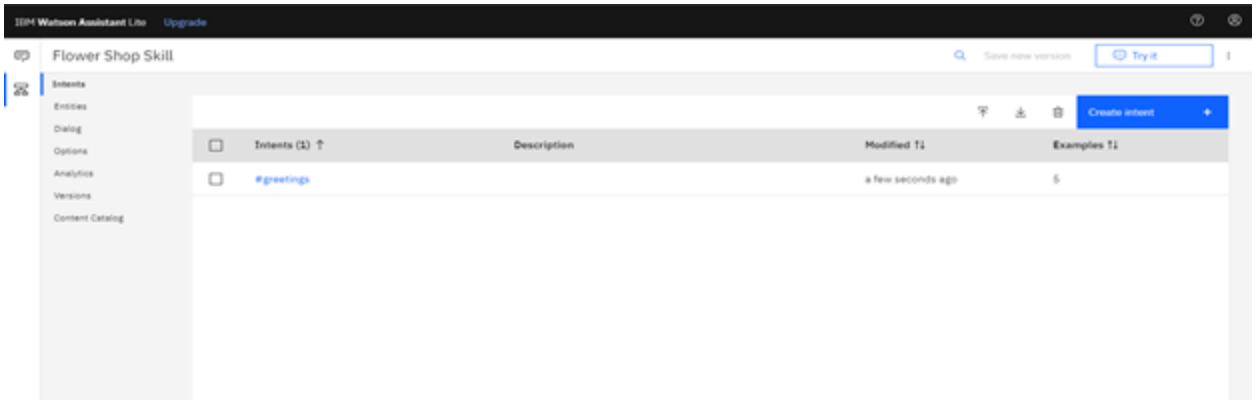
- Hey
- good morning
- good afternoon

and so on. Make sure you add one example at the time.

You don't need to go overboard, especially on such a simple intent, but you should always include at least 5 examples. Ideally more, particularly for more complex intents. If you make a typo in one or two of your examples, don't worry. Keep the typos, as your users are likely to do the same mistakes, so this ends up training Watson on a more realistic input set. When you are done, you can click the back-arrow icon at the top to go back to your list of intents.



5. Once you click on the back-arrow icon, you'll be sent back to the *Intents* section where a list of your intents is shown. Right now, it's just #greetings so let's add more.



By clicking on the Create intent button, repeat the process (step 2 to 4) to add the #thank_you and #goodbyes intents with at least 5 appropriate examples each

For #thank_you, you might use examples such as:

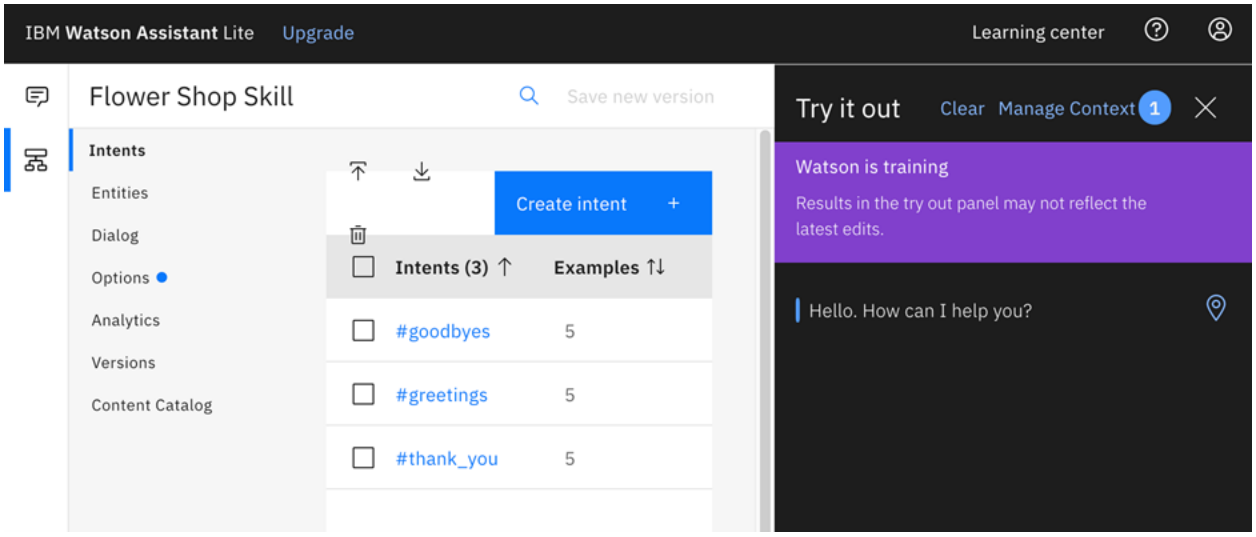
- thank you
- thanks
- thx
- cheers
- appreciate it

For #goodbyes, you might want to use:

- goodbye
- bye
- see you
- c ya
- talk to you soon

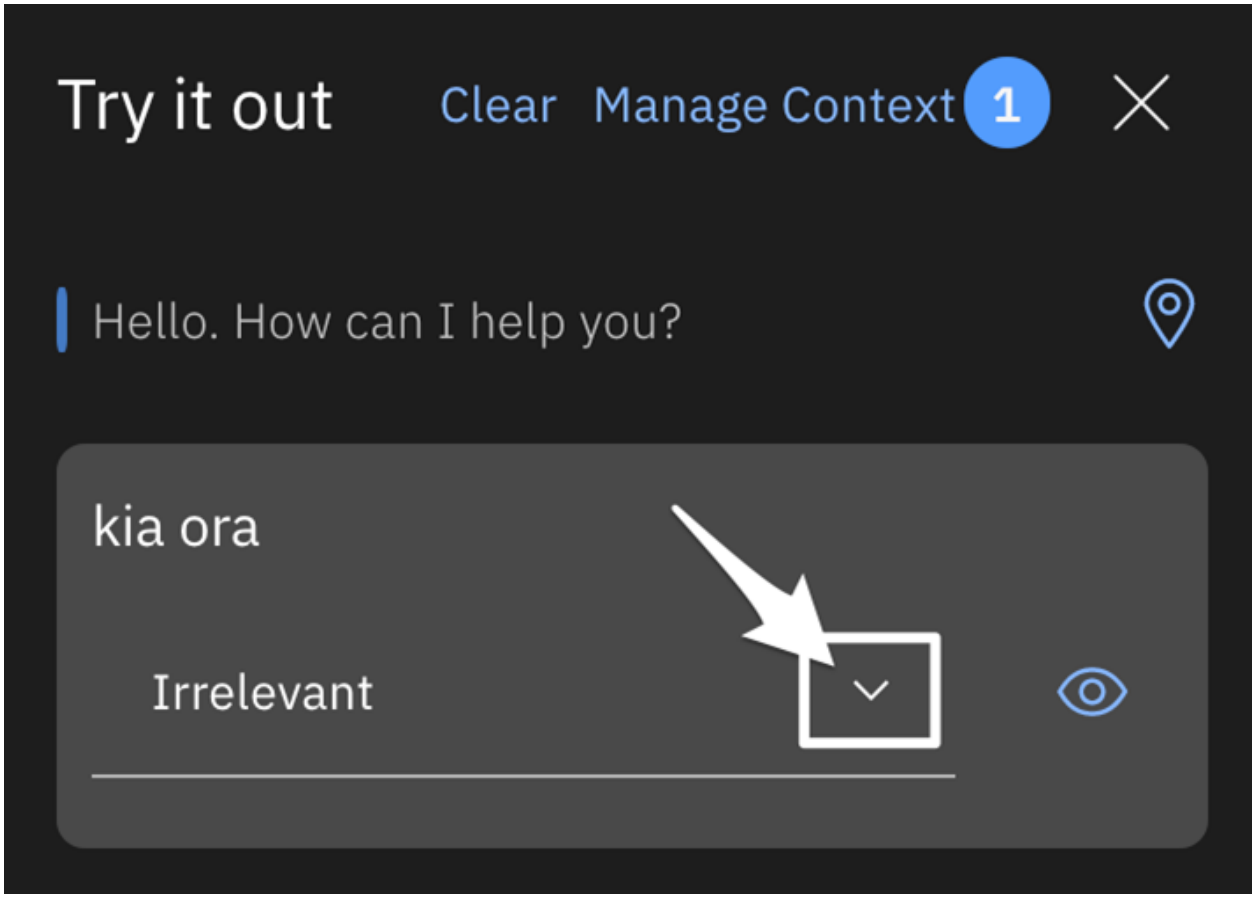
At this point, you'll have the most basic chit chat intents a chatbot needs to have. The more the merrier, of course, but this will do for now.

6. To test our intents, click on the *Try it* button in the top right. A chat panel will appear where you can try user input and see how Watson analyses it and how our chatbot responds. We haven't provided responses yet (we'll do so in the module on *Dialog*) but we can still use it to test the classification of our intents. If you see a *Watson is training* message, please wait for Watson to finish training on your intent examples.



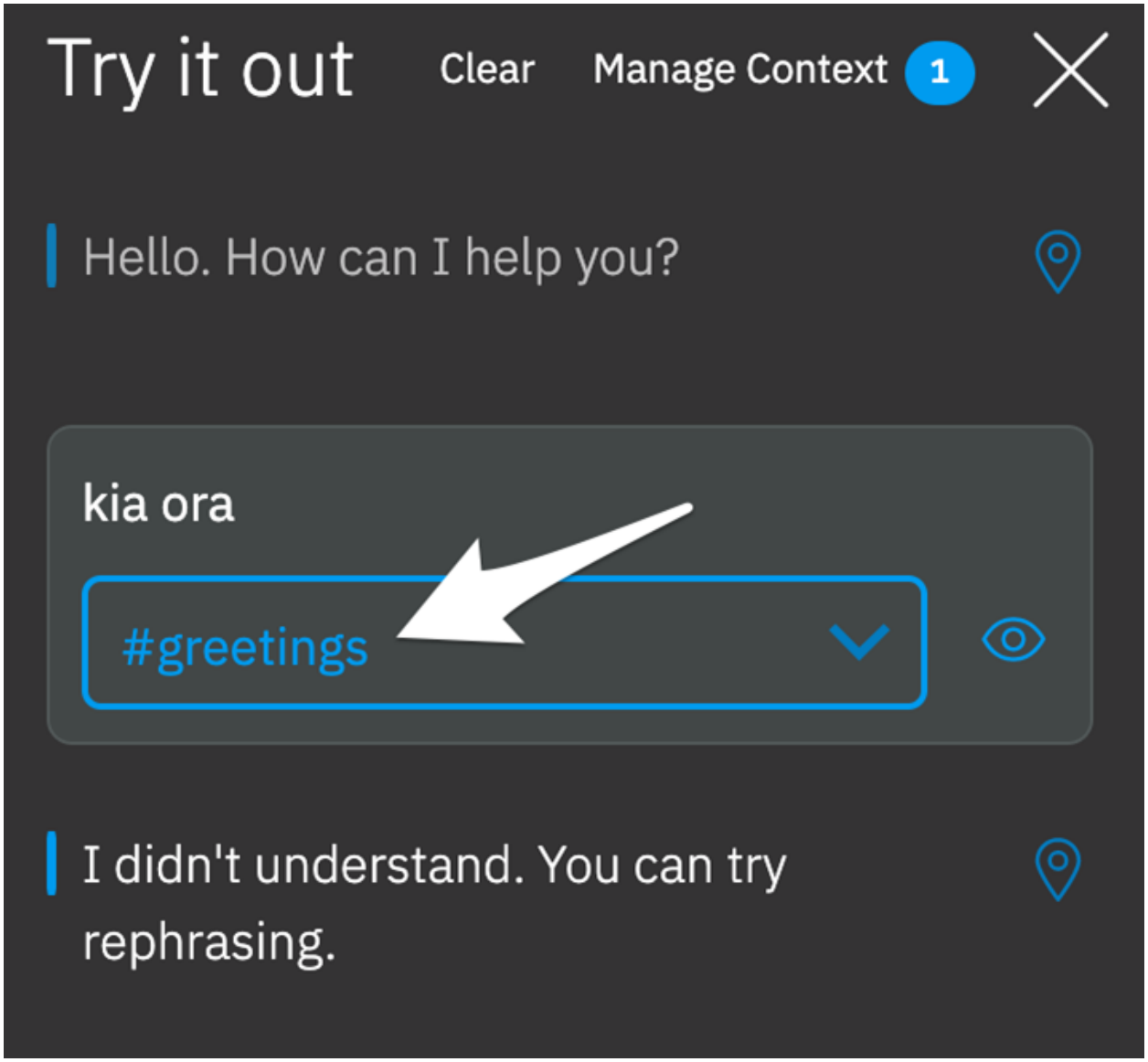
7. Go ahead and try some greetings, thank you, and goodbye messages in the panel. Feel free to try both examples you provided (e.g., thank you) and expressions that you haven't provided as examples. For instance, try howdy and yo.

Now, **try it with kia ora **, a common greeting in New Zealand, but not a globally adopted one. Chances are Watson will categorize it as *Irrelevant*. If Watson miscategorized one example, as it arguably did in this example, feel free to click on the V symbol next to the detected intent to assign a different intent to it. Go ahead and select #greetings for this example.



This will add your utterance (e.g., kia ora) as an example for the intent you picked (e.g., #greetings), further training Watson.

Once Watson is done training, if you test the same utterance again, Watson will correctly recognize the right intent this time.



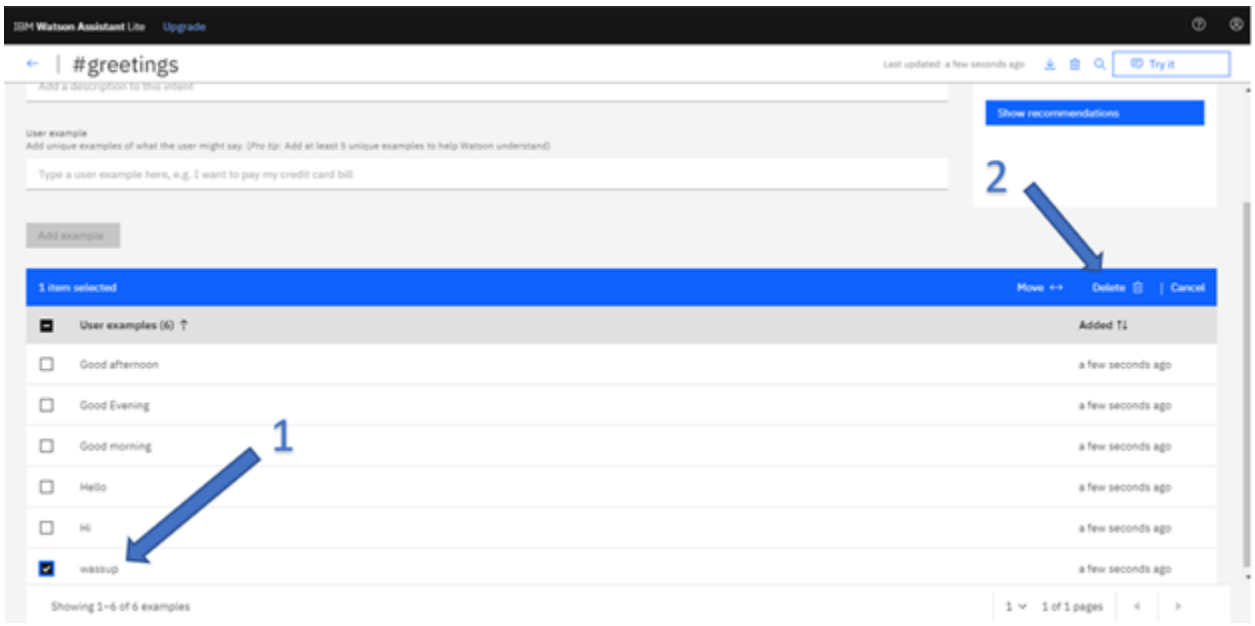
In the *Try it out* panel, what happens if you try a nonsensical input? Randomly smash on the keyboard if you have to. Personally, I produced the beautiful, cat-walking-on-the-keyboard string dljkasdlsa dasldj alskdkas ld. Create your own masterpiece. 😛

Watson will always try its hardest to match the user input to an existing intent, even if it's not a perfect match. But if its confidence level in the best matching intent is very low (below 20%), it will treat the input as *Irrelevant*, as it is likely not relevant to any of our intents.

In the module on the *Dialog* (i.e., Module 4), we'll find out how to deal with situations in which the user enters a question that is irrelevant or outside the scope of our chatbot.

To conclude this exercise, click on an existing intent of your choice in *Intents*, and add one more example to it. Then, select the checkmark next to it, and you'll be given the option to delete it (or even move it to a different intent).

Go ahead and delete that example as shown in the figure below.



In the future, for more complex intents, you'll be able to add examples that originate from your real customers' conversations that they had with your virtual assistant (or your human customer care team) to better train Watson on your business-specific intents (also known as domain-specific intents).

Lab 3: Import Intents

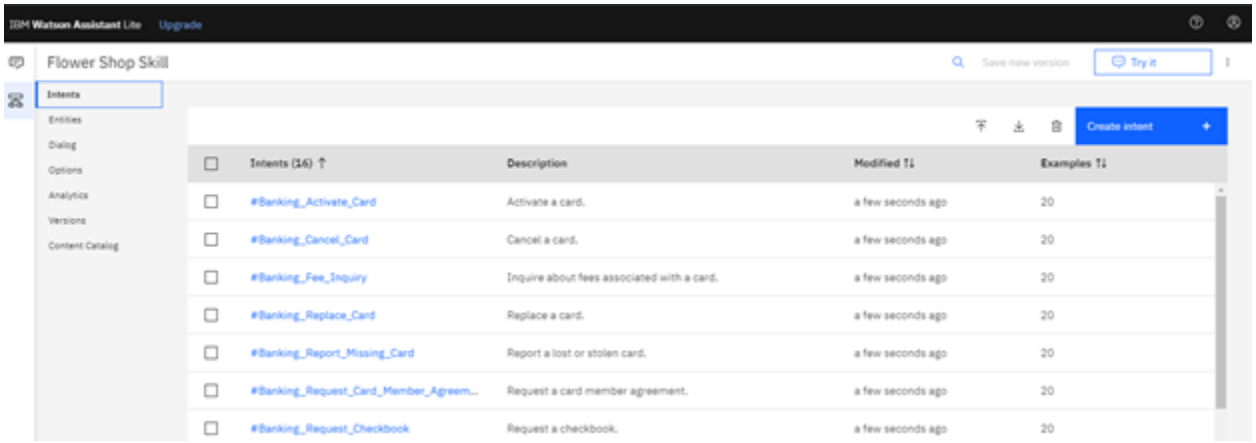
Objective for Exercise:

- Importing intents to your chatbot

Exercise 1: Add intents from the Content Catalog

IBM provides you with some ready-made intents that might be relevant to the scope of your chatbot. To see what's available, click on Content Catalog within your dialog skill.

Choose one category of your choice (e.g., *Banking*) and then click on the Add to skill button next to it. Switch back to the *Intents* section (by clicking on Intents in top left of the page) and you should see a series of new intents relevant to common queries customers may have for the category of your choice.



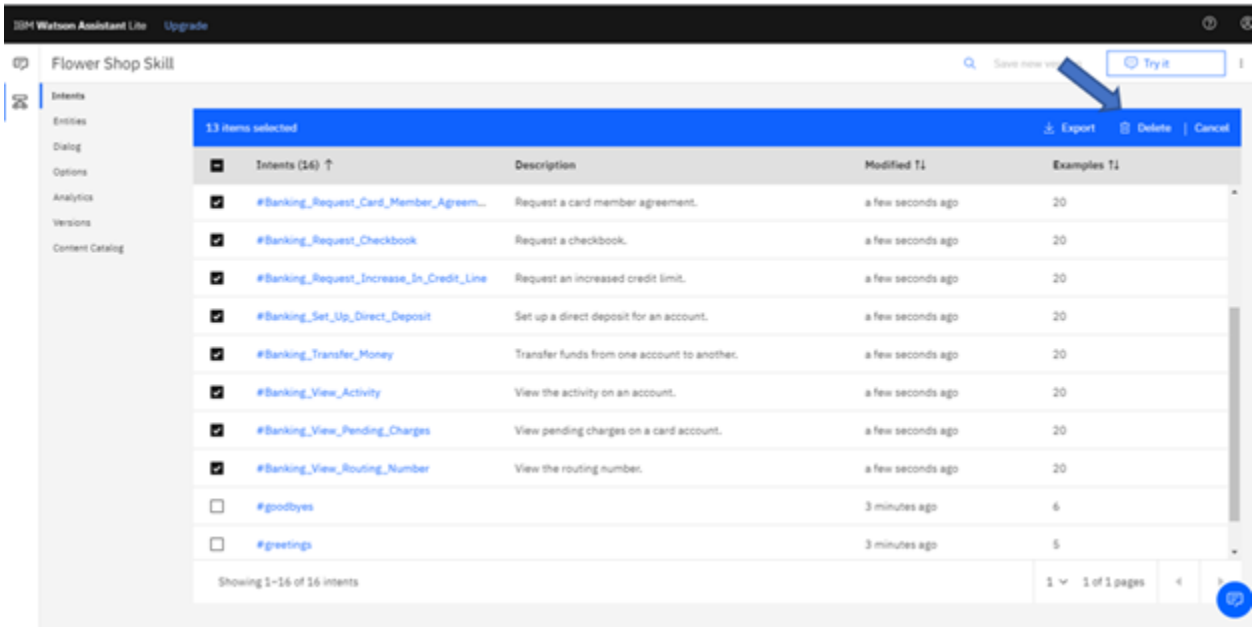
This isn't quite a pre-made chatbot but it's a nice starting point, that you can edit and adapt as needed. Feel free to try them out in the *Try it out* panel.

For example, if you added banking intents, try I lost my credit card in the *Try it out* panel. What intent is detected by Watson?

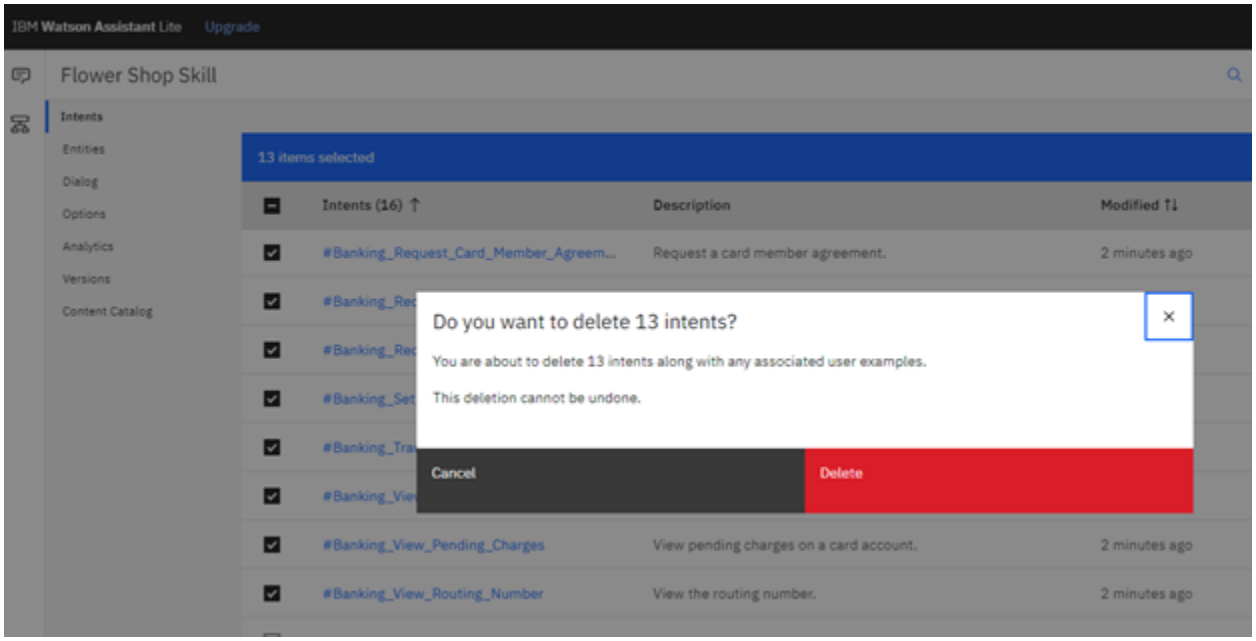
Make sure you let Watson finish training, first. If it's taking a long time, feel free to skip this test, and continue with the deletion below.

As usual, ignore the response that we get from the chatbot. It's simply because we haven't addressed responses yet. What we care about at this stage is that Watson correctly identifies and classifies our input.

We are not going to use banking intents for our flower shop chatbot so select the checkmarks next to them and press the Delete button to remove them. (Make sure you keep the chit chat intents we created.)



Confirm the deletion when prompted.



You'll notice, in the screenshot on the previous page, how you were also given the option to export the selected intents. This is quite useful when reusing intents across different chatbots. Particularly when they are intents you created and not pre-made, as the ones

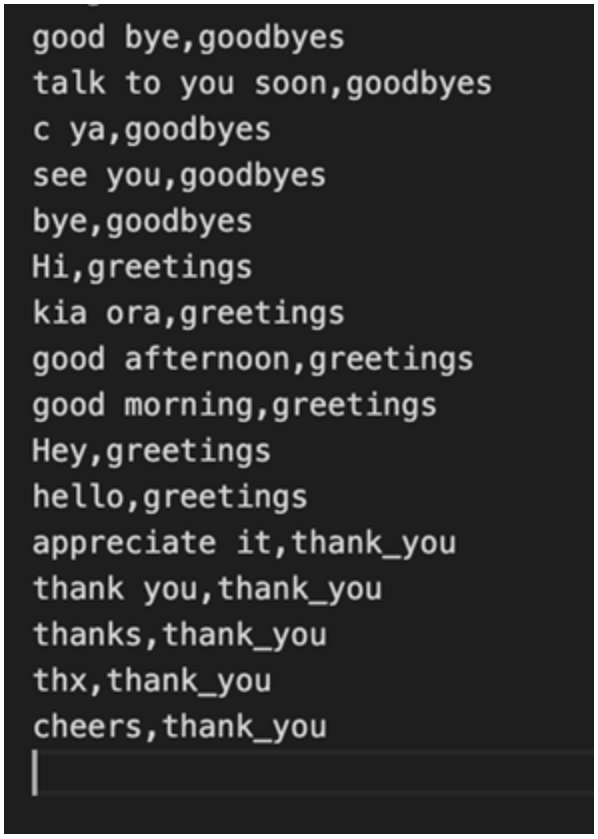
in the Content Catalog will also be available in your new chatbot so there is no need to export and import them.

Go ahead and select our three chitchat intents and click on the Export button to download a CSV file containing our intents and examples.

Open the file to see what it looks like.

	A	B
1	good bye	goodbyes
2	talk to you soon	goodbyes
3	c ya	goodbyes
4	see you	goodbyes
5	bye	goodbyes
6	Hi	greetings
7	kia ora	greetings
8	good afternoon	greetings
9	good morning	greetings
10	Hey	greetings
11	hello	greetings
12	appreciate it	thank_you
13	thank you	thank_you
14	thanks	thank_you
15	thx	thank_you
16	cheers	thank_you
17		
18		

You'll notice that the structure is very simple. The example is in the first column, and the intent it corresponds to, is in the second column. If you open the CSV file in a text editor, you'll see its raw form.



As you can see, it's very easy to create, modify, and delete intents, whether they were manually created or imported from the Content Catalog.

Exercise 2: Import intents from a CSV file

Just like we exported our intents to a CSV file, we can do the opposite and import intents from a CSV file. This format is particularly handy because it allows you to easily import intents (and their examples) from a spreadsheet. Let's see how this works in practice.

1. Download the CSV file I prepared for you or copy and paste the following in a hours_and_location_intents.csv file.

When do you open,hours_info

When are you open,hours_info

What days are you closed on?,hours_info

When do you close,hours_info

Are you open on Christmas' Day?,hours_info

Are you open on Saturdays?,hours_info

What time are you open until?,hours_info

What are your hours of operation?,hours_info

What are your hours?,hours_info

Are you open on Sundays?,hours_info

what are you hours of operation in Toronto,hours_info

what are the hours of operation for your Montreal store,hours_info

list of your locations,location_info

Where are your stores?,location_info

Where are you physically located?,location_info

What are your locations?,location_info

List of location,location_info

Give me a list of locations,location_info

Locations in Canada,location_info

locations in America,location_info

what's the address of your Vancouver store?,location_info

What's the address of your Toronto store?,location_info

do you have a flower shop in Montreal,location_info

where is your Toronto store?,location_info

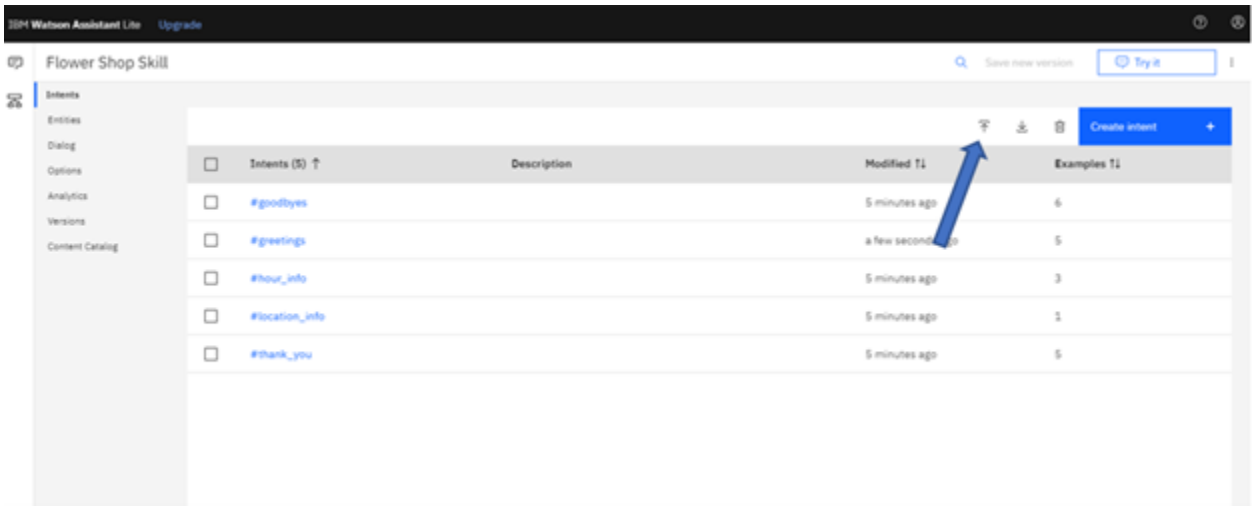
If clicking on the link simply opened the file in a new tab in your browser instead of downloading it, with that tab selected, press CRTL+S on Windows or ⌘+S on Mac to download it.

You'll notice that the structure of the file is very simple. Each line of the file has an example, comma separated by the intent we want to assign to it. Just like the chitchat examples you've seen in the previous page.

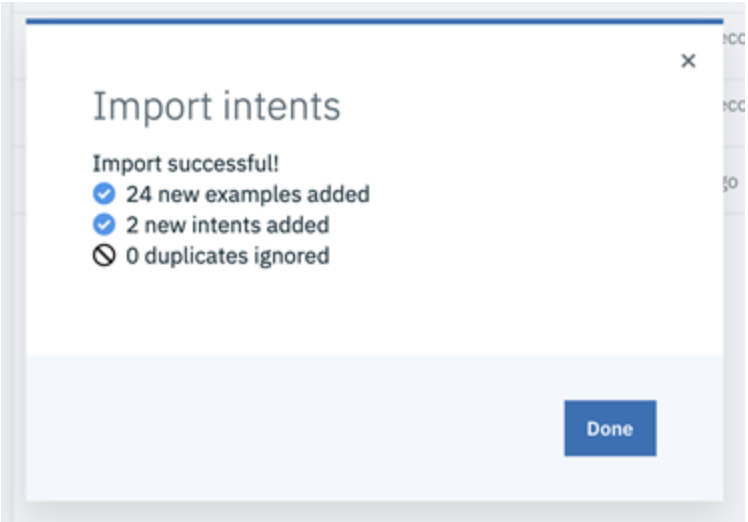
In our *Flower Shop Chatbot* we want to allow people to inquire about hours of operation and addresses of our flower shop stores, so this file includes examples for both #hours_info and #location_info.

Note that the # prefix is not included in the CSV file. It will be automatically added by Watson to the intent names when importing them.

2. From the *Intents* section of your skill, click on the Import intents icon next to the *Add intent* button.



3. Select Choose a file from the window that appears and select the CSV file you downloaded on your local drive.
4. Click on the Import button. A report of what was imported will be shown as seen in the picture below. Click on *Done* to close the window. You now have successfully imported two new intents and their examples to train Watson.



Take a moment to review the intents that were imported and the examples for each of them.

Next, take them for a spin in the *Try it out* panel. Ask questions like you naturally would to inquire about store hours or address information. Does it recognize the intents we imported well enough? Train Watson further by adding your own examples directly from the *Try it* panel when it fails to interpret them correctly.

At this point, our chatbot understands basic chitchat and it detects when a question is about hours of operation vs when it's about location.

Well done completing lab 3!

Lab 4 - Create_Entities

Objective for Exercise:

- How to Create entities.
- Use system entities.

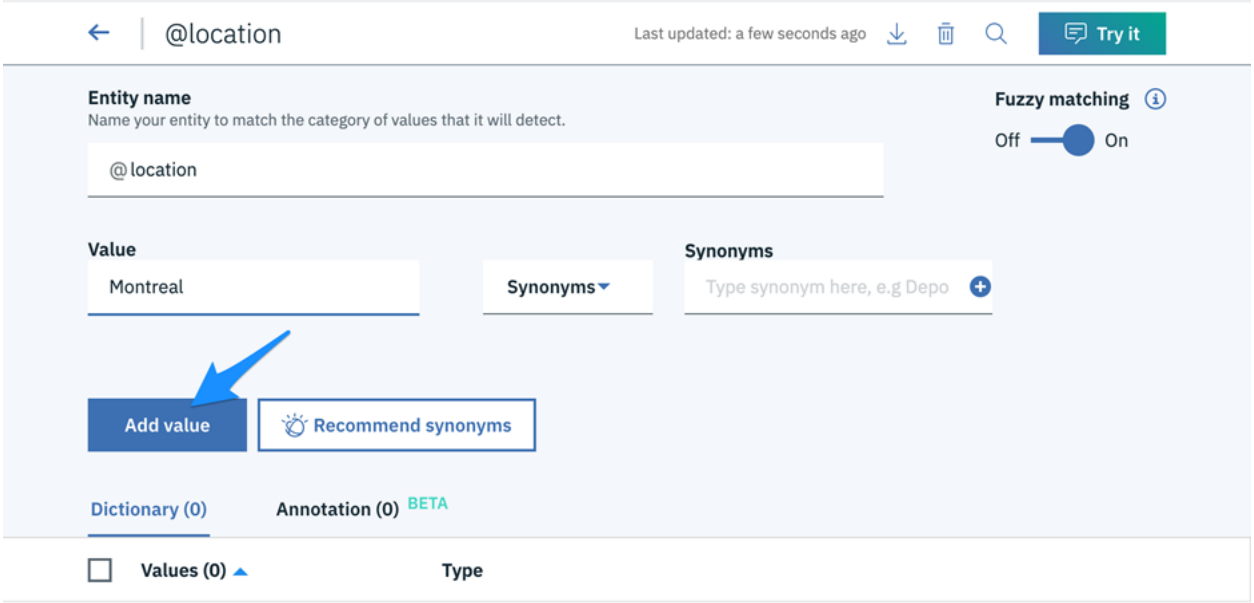
Exercise 1: Creating entities

Entities recognize and capture specific pieces of information in the user input. In our flower shop chain chatbot, people asking us about store hours and locations might provide a specific location.

In our fictitious Flower Shop chain, we have stores in Toronto, Montreal, Calgary, and Vancouver. So, when a user asks, Where is your Toronto store? we shouldn't ignore that extra bit of information so that we can take the location into account when formulating a response.

We can start by creating a @location entity for those cities.

1. In your skill, click on *Entities* to enter the entities section.
2. Here, click the Create *entity* button. Choose @location as the entity name (note that the @ symbol is automatically added for you). Leave *Fuzzy Matching* enabled so that we can still detect the city name even if the user misspells it. Finally, click the *Create entity* button.
3. You'll be prompted to enter entity values and possible synonyms. Enter Montreal and then click *Add Value* to add this entity value to our entity.

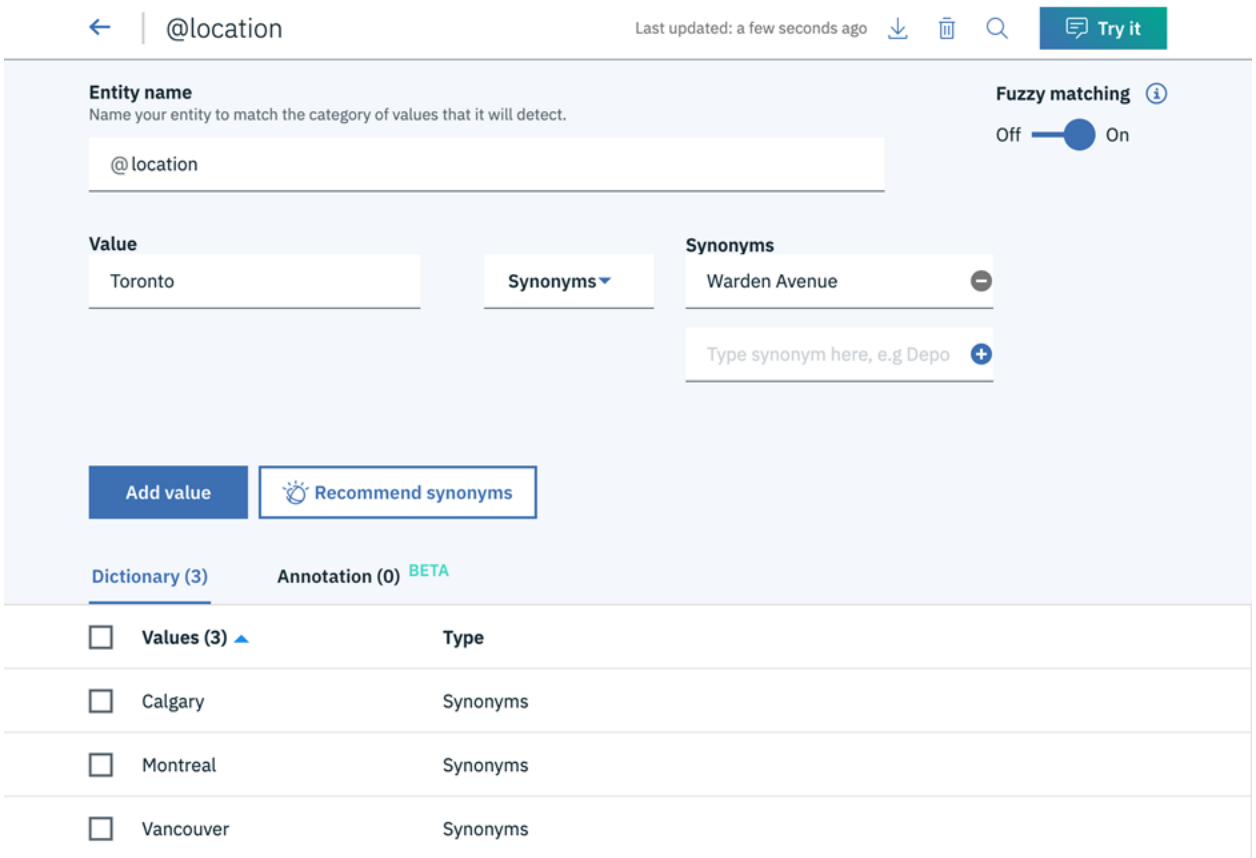


Generally speaking, you won't need a synonym for cities, but you might include some if the city has common nicknames or if people refer to your store location by its street or neighbourhood in the city. Nearby small cities and towns can also act as synonyms. After all, if people are asking about your store in a nearby town, they might be happy with an answer for the nearest city.

Essentially, a synonym is not necessarily the dictionary definition of synonym. Though those are good candidates for synonyms as well when it makes sense. For example, we

could have an entity called @relationship and the entity value @relationship:mother with mom as a synonym for that value. When the user enters a question including the word mom, Watson will detect @relationship:mother (the entity value for that synonym).

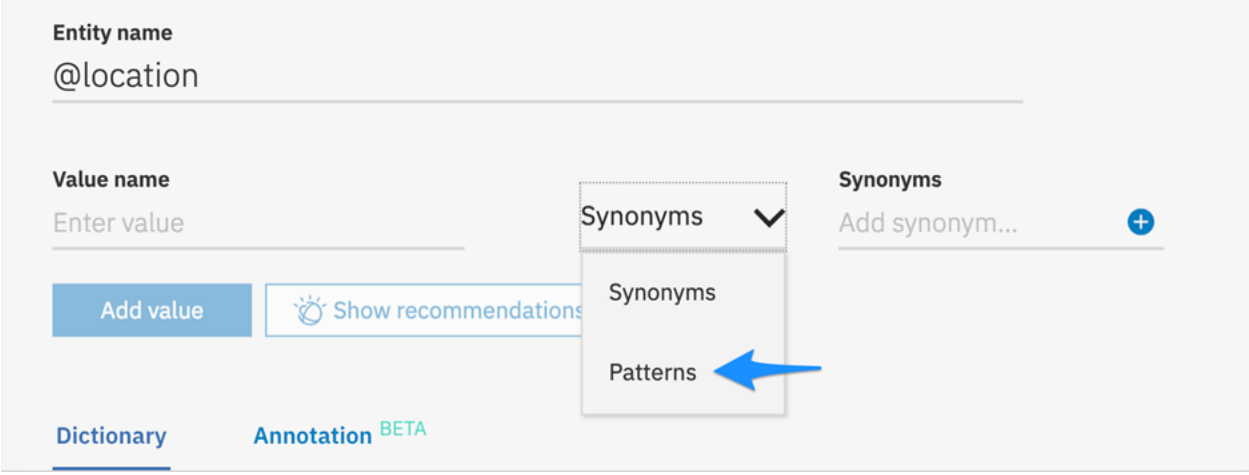
4. Repeat the process for Calgary and Vancouver. Next, add Toronto as well. But for Toronto, add Warden Avenue as a synonym, as shown in the picture below.



Click on the back-arrow in the top-left to go back to your skill.

Open the *Try it out* panel and wait for Watson to finish training. What happens if you try, hours of operation of your warden ave store in the *Try it out* panel?

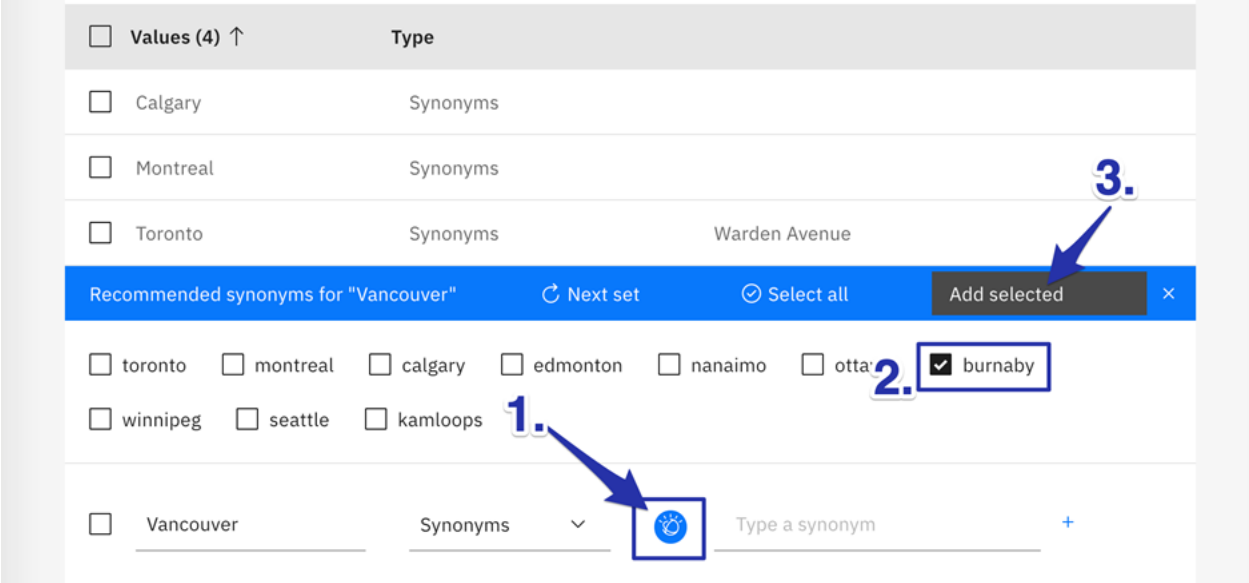
Even though we haven't entered Warden Avenue spelled exactly as defined in the synonyms, fuzzy matching helps our chatbot detect the right entity value. It's worth noting that entity values can also have patterns, accessible from the *Synonyms* drop-down, as shown in the image below.



A pattern is an advanced feature that allows you to detect an entity value based not on a specific string (e.g., its synonym) but rather on a specific pattern like a properly formatted phone number, email address, or website address. If you are a programmer, it's worth noting that you specify your pattern as a Regular Expressions (e.g., ^(?([0-9]{3}))?[-.]?([0-9]{3})[-.]?([0-9]{4})$ to detect that a North American phone number was provided). If you are not a programmer, you can safely ignore this advanced feature. 😊

5. At any time you can click on an entity value to edit its name or synonym. Entities values are allowed to have spaces in them. When you first create an entity value, you're given the option to click on the *Show recommendations* button to select some synonyms from a list provided by Watson.

Try out this feature. Click on *Entities*, click on the @location entity, then select the @location:Vancouver entity value by clicking on it. A Watson icon will appear. Click on it as shown in the picture below.



Watson will make a few suggestions. For example, for Vancouver, it will suggest a few nearby cities as well as other major cities in Canada. **Select Burnaby as a synonym * and then click the Add *selected* button Finally, click on the X icon to close the recommendation section.

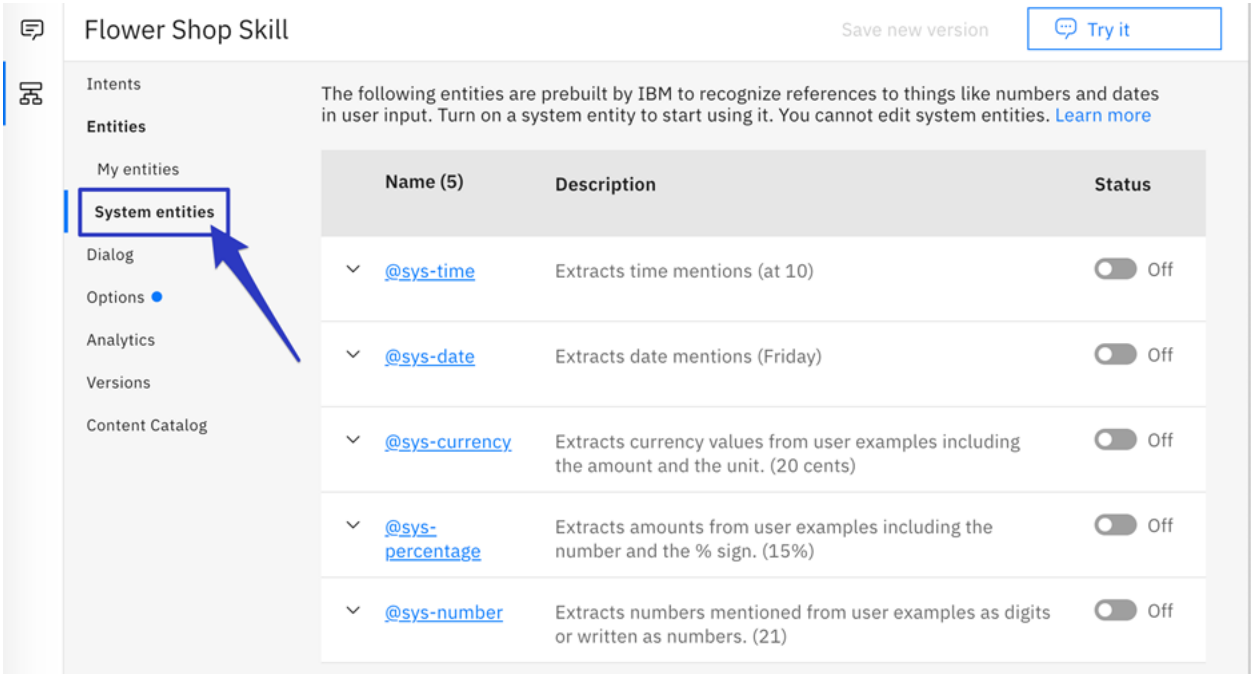
6. Use the *Try it out* panel to test out these entity values. Try entering, What are your hours of operation in Montreal and Where is your Burnaby store located? to see how Watson classifies that user question in terms of intents and entities.

Exercise 2: Take a look at system entities

System entities allow us to easily detect common specific pieces of information like dates, times, numbers, currencies, etc. They are quite convenient when collecting information from the user.

For example, a restaurant reservation booking chatbot might use @sys-date, @sys-time, and @sys-number to detect the date, time, and party size for the reservation.

You'll find the currently available system entities under the *Entities* menu , as shown below.



If you ever need a system entity, you can simply turn it on by clicking on the toggle to the right of it.

In a previous version of this course, we employed the @sys-location and @sys-person entities, but they have since been deprecated.

Generally speaking, it's worth using a system entity if one fits the bill for what you are trying to do. But if it makes your life more difficult due to your specific requirements, you're better off creating your own custom entity as we did in Exercise 1.

Lab 6 - Implement the Dialog

Objective for Exercise:

- How to Create a Dialog and improve the prompt.
- How to Add Chit Chat nodes.

With intents and entities under our belts, we can finally look at the third component: the dialog.

In fact, at this point, our chatbot can understand some intents and detect a few specific pieces of information thanks to entities.

What we are missing is using this information to formulate appropriate responses to the user. We'll do so in this module to create a simple, but useful chatbot.

In this lab, we'll start by defining chit chat responses.

Exercise 1: Create a Dialog and improve the prompt

Let's kick things off by investigating the dialog and adding a good prompt for our chatbot.

1. Click on the *Dialog* section of your skill.
2. Take a moment to investigate the default *Welcome* and *Anything else* nodes that were

generated by default for you, by clicking on them.

3. Open the *Try it out* panel and click on the *Clear* link at the top to start testing the chatbot

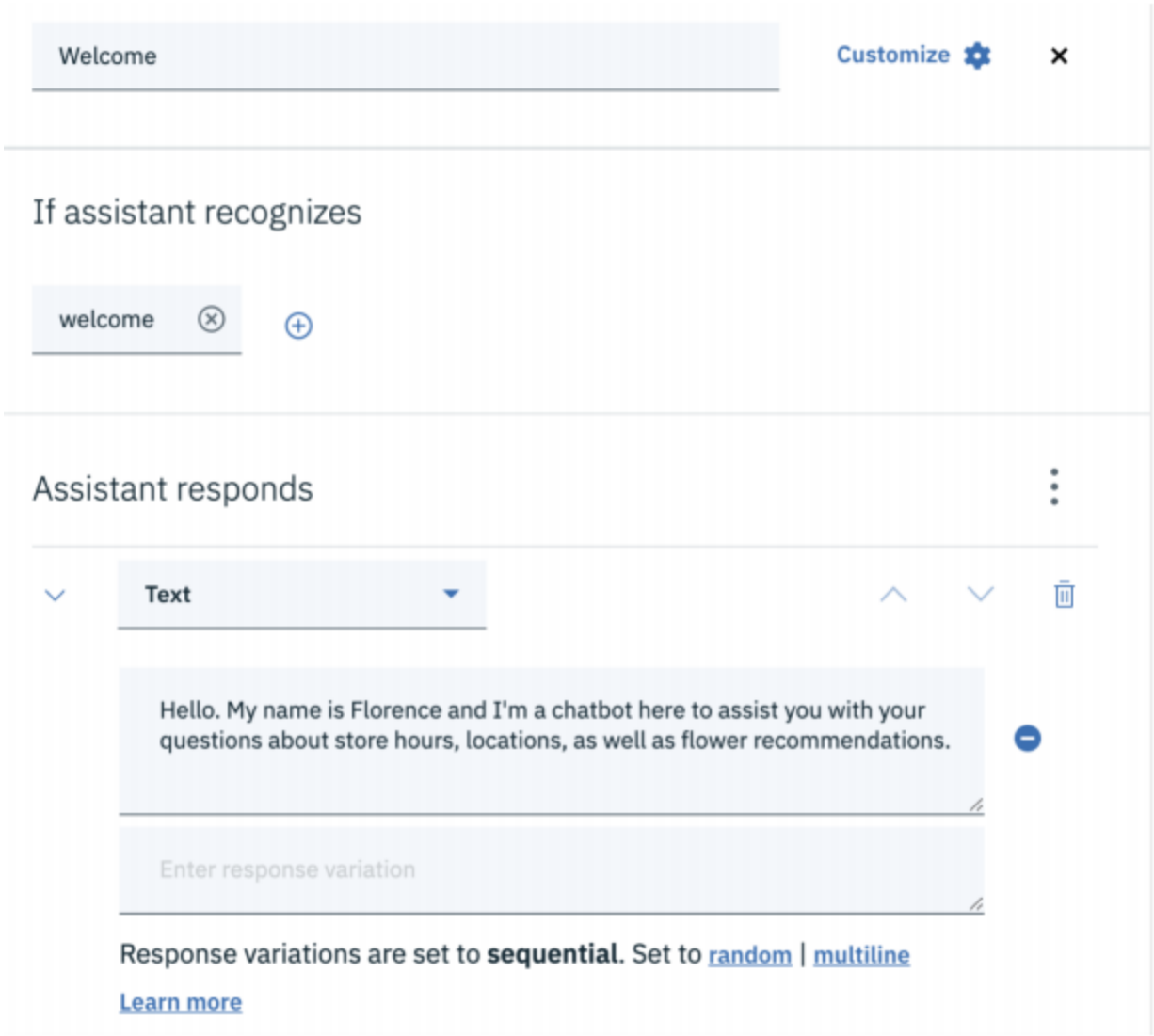
from scratch.

The default prompt, "Hello, How Can I help you?" is actually not very user-friendly. Let's change it.

Close the *Try it out* panel, and then select the *Welcome* node. Here you'll want to edit the response to say:

Hello. My name is Florence and I'm a chatbot here to assist you with your questions about store hours, locations, as well as flower recommendations.

Change the name from Florence, to whatever flower-inspired name you prefer, to make it yours.



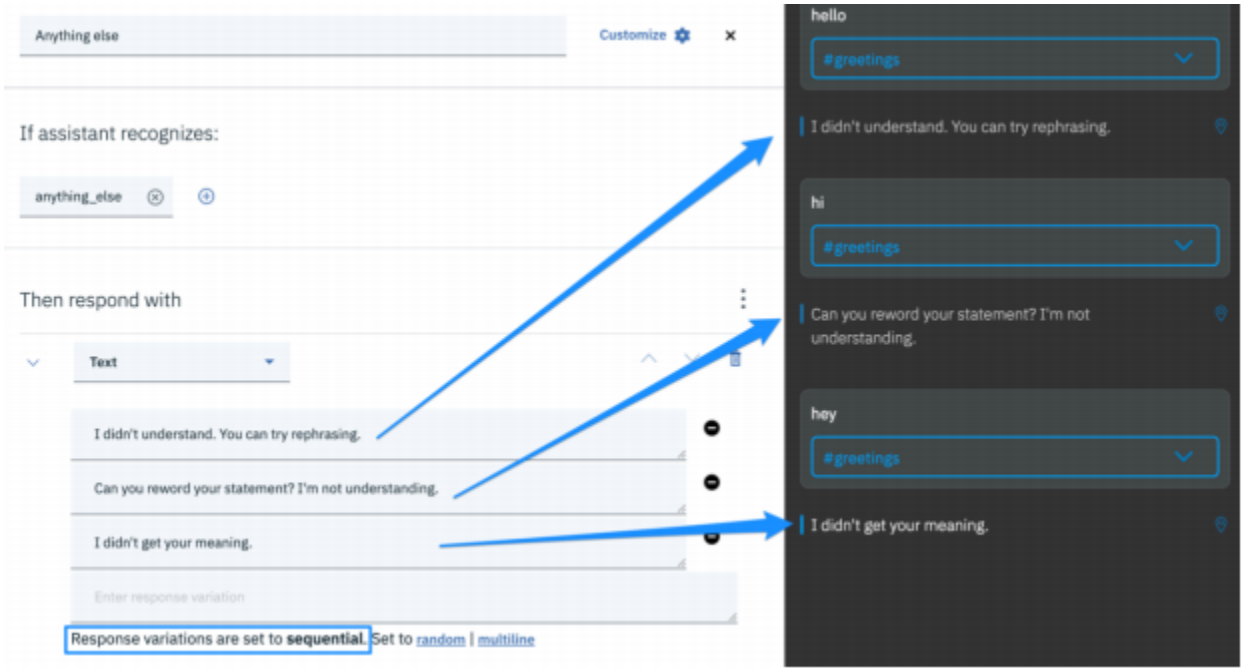
Open the *Try it out* panel and click the *Clear* link at the top once again. This time you should see the much more informative prompt we just specified.

4. Good. Now try replying hello in the *Try it out* panel. What happens? Watson recognized the

right intent (i.e., #greetings) but doesn't have a node to handle greetings, so the fallback node *Anything else* was executed. We'll remedy this in the next exercise.

It's worth noting that if you enter a greeting (or anything at this point) multiple times, you'll get a different response each time. The reason for this is that the *Anything else* node has three response variations by default. Furthermore, these are set in sequential mode. So, every time we hit this node, the next response variation is provided to us, as

shown below.



If we didn't care about the specific order, we could set this to random and a random variation would be provided each time.

The reason why we want variation is that we don't want to robotically say to the user, *I don't understand* every time the chatbot fails to handle the user input with an appropriate node. It gets old fast and makes our chatbot come across as not as smart as it could be.

For nodes that are unlikely to be hit multiple times within a conversation, it's okay to have a single response with no variations. In every other case, variations are good to have.

You might wonder whether you should prefer sequential or random for your variations. Sequential works well when you plan to leverage your knowledge that the node was hit multiple times to provide a better response to the user. Random when the variation that is given to the user doesn't matter.

Go ahead and add a fourth variation to the *Anything else* node, with the following text:

It looks like we are not quite getting each other today. Would you like to talk to a human agent instead? If so, please contact us at 555- 123 - 4567 or email us at support@example.org.

The \ before the @ is needed to display the special character (something that programmers call, "escaping").

Because we have this node's responses set in sequential mode, we ensure that this escalation of the sort is only given as an option after we failed to understand the user four times in the

same conversation. If this was set to random, we'd risk escalating the very first time we don't understand the user, which is typically not what we want.

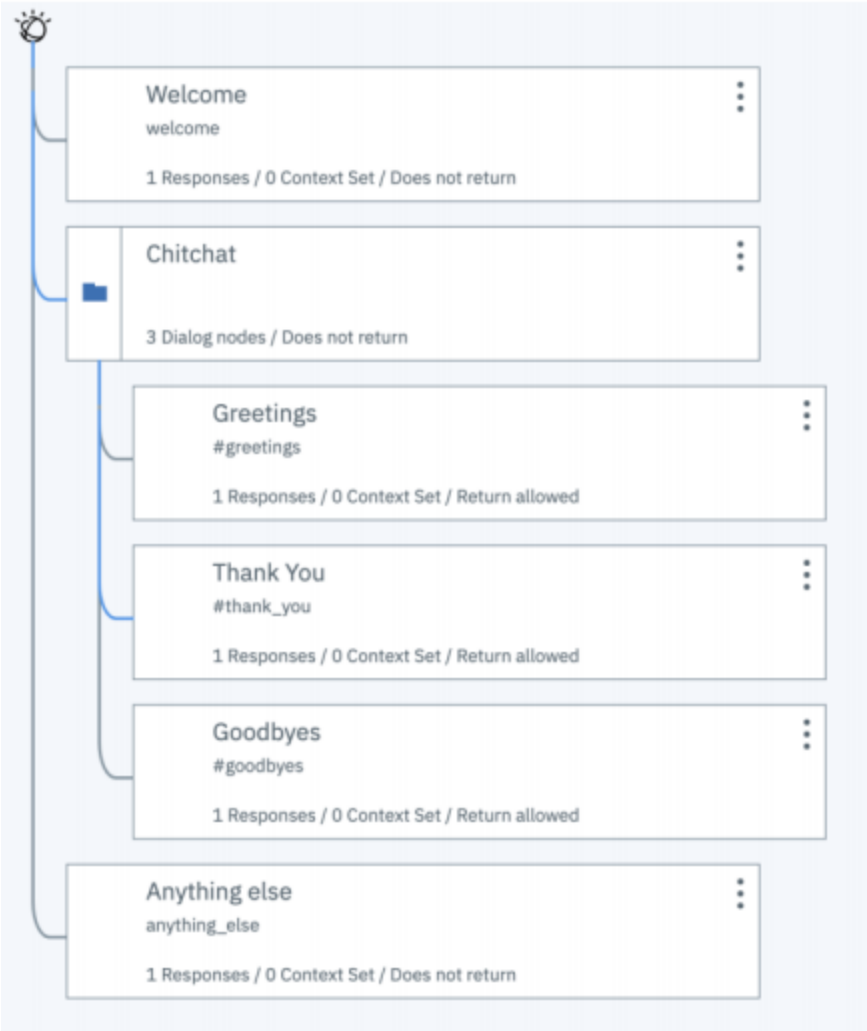
Exercise 2: Add Chit Chat nodes

As you know, we have three chit chat intents: #greetings, #thank_you, and #goodbyes. We now need to have nodes that specify what response we want to give the user when such intents are detected.

We have a couple of strategies possible here. We could create three nodes, one for each of these intents. This is the most common and simple approach. The other option would be to create a single node for chit chat that uses multiple conditional responses attaching a condition to each response.

I would recommend that you stick to the traditional way as it's more flexible. It allows us to add more chit chat nodes down the line, as well as making the chit chat logic more complex if needed.

We do still want to keep things organized, separating small talk from domain-specific nodes. So, we'll create a folder for chit chat, and we'll create three nodes inside for now. The picture below shows the structure of the lab's end result.

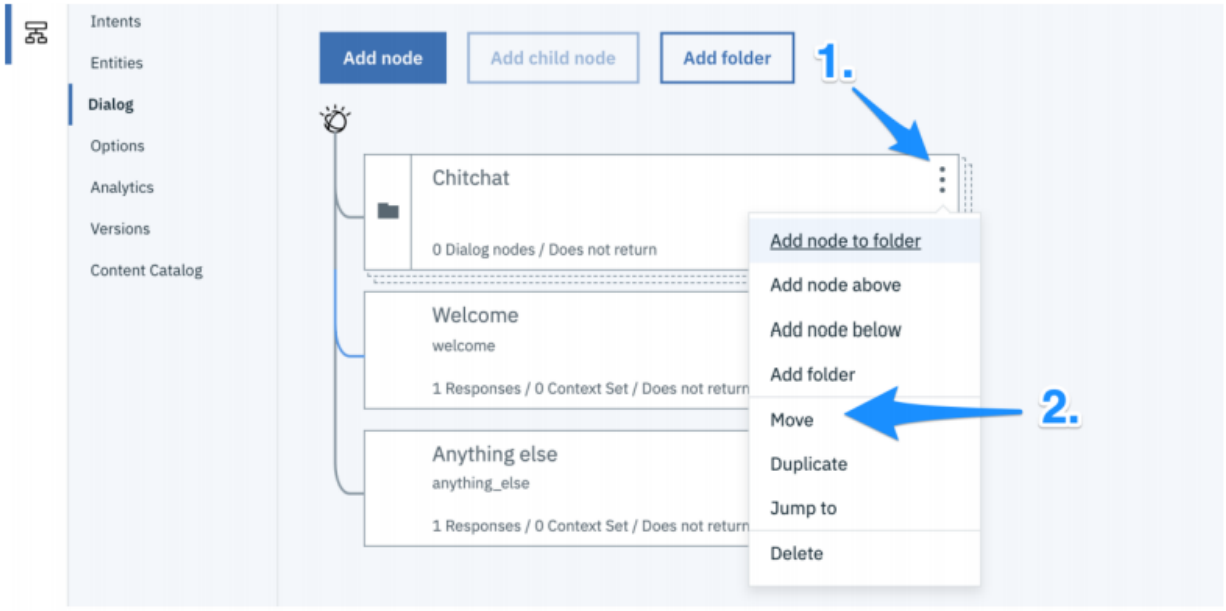


At any time, you'll be able to collapse or expand the folder by clicking on the folder icon next in the *Chitchat* folder.

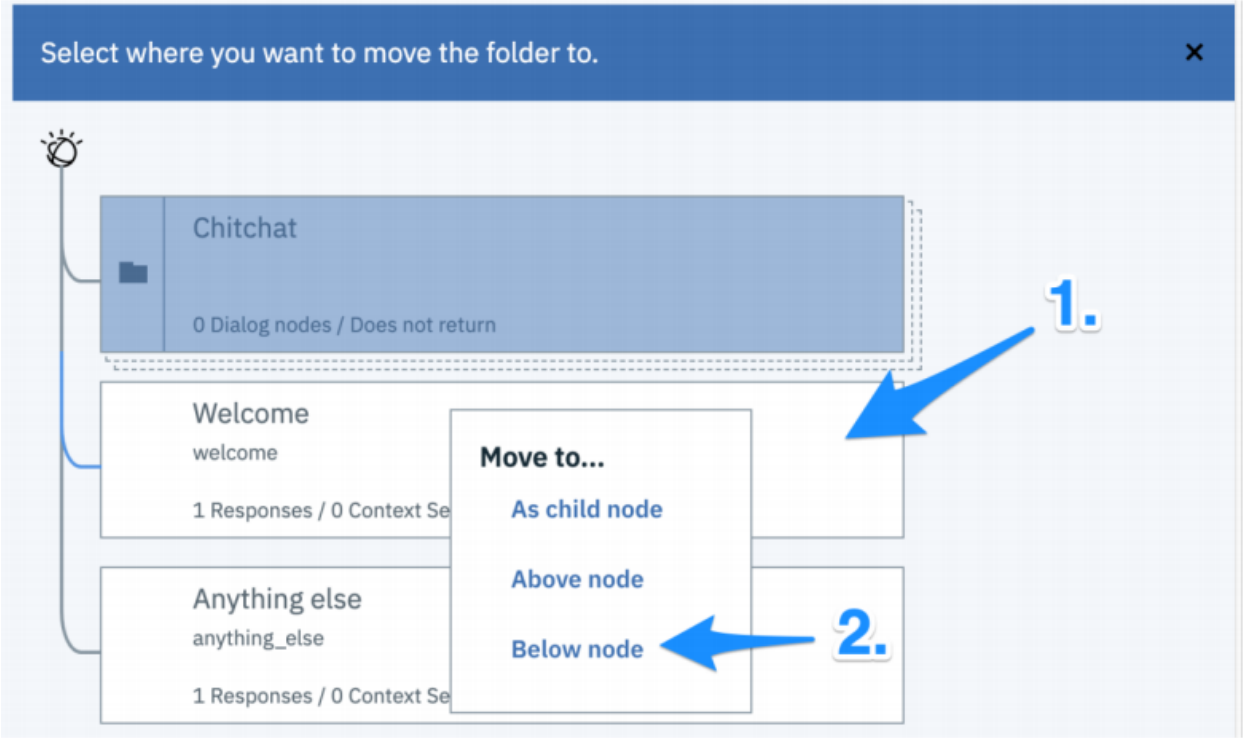
Follow these steps to implement this in your chatbot.

1. Click on the *Add folder* button. Name the folder Chitchat. You don't need to specify a

condition for the folder, as the conditions of the child nodes will suffice. 2. We need to ensure that the folder is located between our prompt node (i.e., *Welcome*) and our fallback node (i.e., *Anything else*). So, if it was created above the *Welcome* node (or below *Anything else*), you'll want to click on the three vertical dot icon to the right of the folder and then click *Move*.



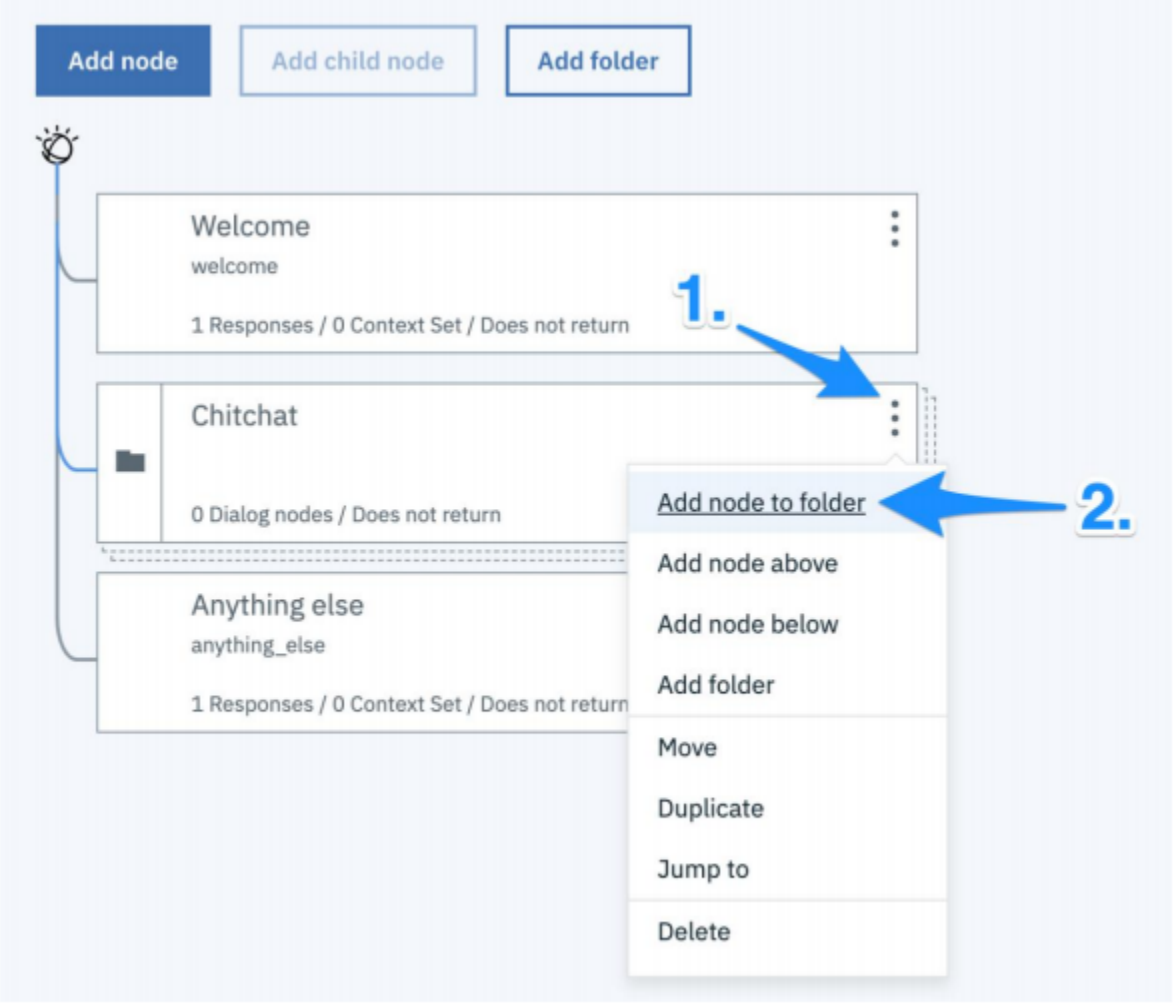
Next, you'll need to select the *Welcome* node, and click on *Below node*.



This will move the folder below *Welcome*. If it opened the folder for you, click on the X icon to close it and to return to the dialog.

3. Select the options for the node (by clicking on the three vertical dot icon, sometimes called

the more options or kebab menu) in the *Chitchat* folder, then click *Add node to folder* , as shown in the screenshot below.



This will create an empty child node within the folder.

4. Name this node Greetings. We want it to be executed when the #greetings intent is

detected, so under *If assistant recognizes* enter the #greetings intent. Autocomplete will help you find the intent (not that useful here, but quite handy in complex chatbots with many intents).

It's worth noting that you can make the condition of a node as complex or as simple as you'd like. You can use || (or its alias OR) and && (or its alias AND) to make the condition more complex. We don't want this here, but if you wanted to execute a node if the intent

detected was either #greetings or #goodbyes we could simply type #greetings OR #goodbyes in the node condition.

5. Enter a few appropriate responses. The scenario we are handling here is one in which we

already greeted the user with our prompt, and they replied with a greeting. So, we should greet them back without repeating the prompt verbatim.

Enter a few responses to offer some variation if we get a greeting-happy user. Examples could be:

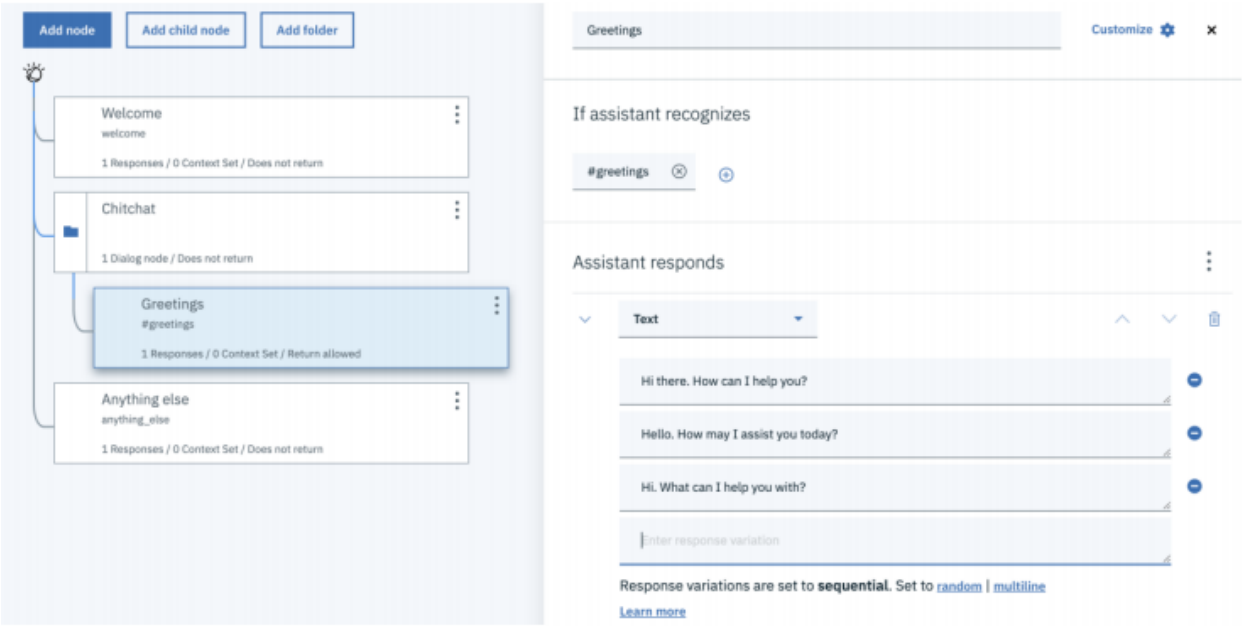
- Hi there. How can I help you?
- Hello. How may I assist you today?
- Hi. What can I help you with?

Normally, I would advise against open-ended questions such as “how can I help you?”, but since we already qualified the scope of the chatbot in our prompt, we can get away with it here.

6. You can leave the response variations set to sequential or set them to random if you prefer.

The third option, *multiline* , is not applicable here, as it would provide a response over multiple lines using each response you wrote as its own line, de facto asking the user three variations of “how can I help you?” all at once.

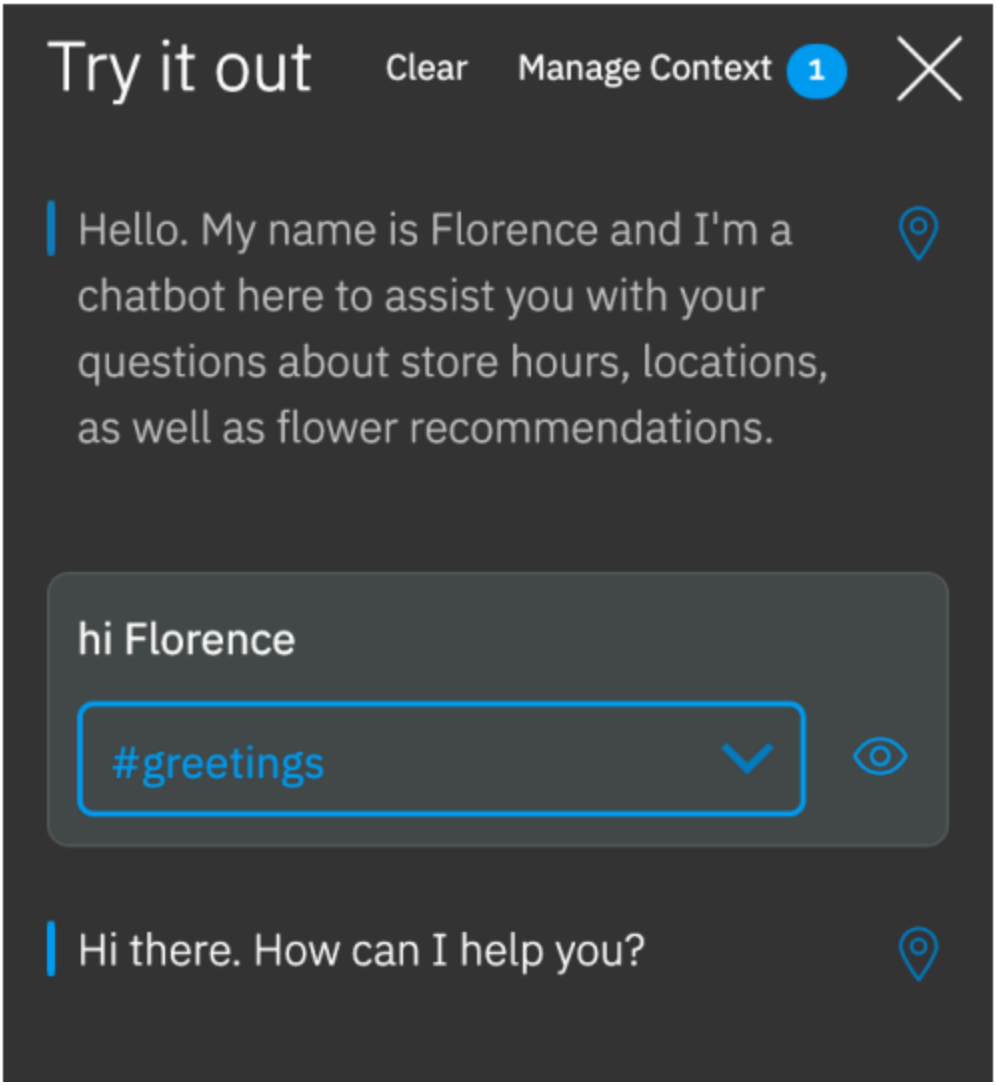
This is what the node will look like if you followed each step correctly.



7. The *Then assistant should* section at the bottom of the node defines what happens after this

node has been executed and a response was given to the user. In the case of this node, after we responded to the user, we expect them to enter some more questions, so you can also leave *Wait for reply* as the final action for this node. 8. Close the node and open the *Try it out* panel (if you closed it). Click the *Clear* link to start a new conversation. Try to reply hi Florence to the chatbot prompt.

If you see a proper response like in the image below, congratulations! You just had your first conversation with our chatbot. It's not a complex interaction, but it's a good start.

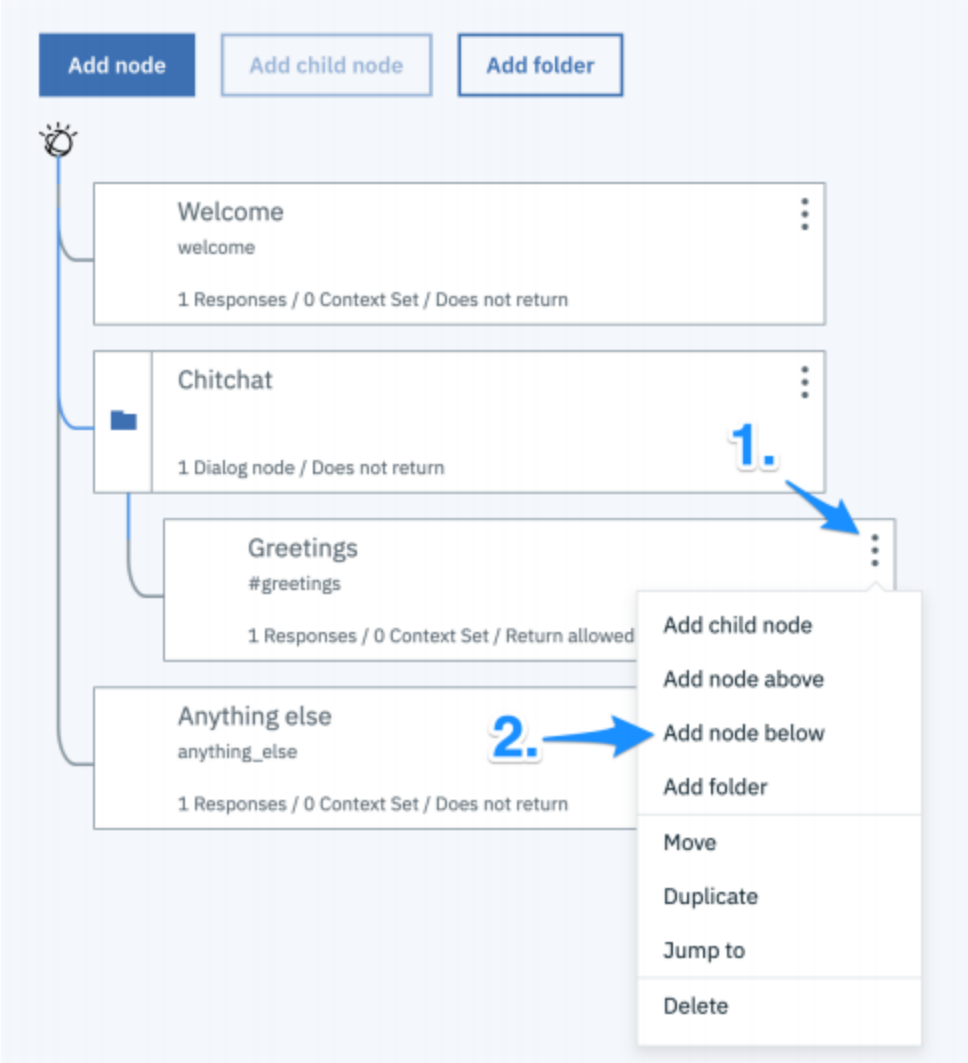


You can now close the *Try It out* panel.

9. Now we need to repeat the process to handle the scenario in which our user thanks us and

the #thank_you intent is detected.

What we want to do is create a new node inside of the *Chitchat* folder. Click on the more options menu (the three vertical dot icon, that is) for the *Greetings* node, and then click *Add node below*****.



This will create an empty peer node below *Greetings*.

The order of these chit chat nodes is not that important because they are all simple nodes with independent intents. However, the order can matter in more complex scenarios (as we'll see in a moment).

It also makes sense to place them in a logical manner that is roughly equivalent to how a normal conversation would go. Greetings first, thanks in the middle, and goodbyes at the end.

Go ahead and make this node handle the #thank_you intent. You can name it whatever you like but Thank You will do. For the responses, you'll likely want something like:

- You're welcome. Please let me know if you need anything else.
- My pleasure.

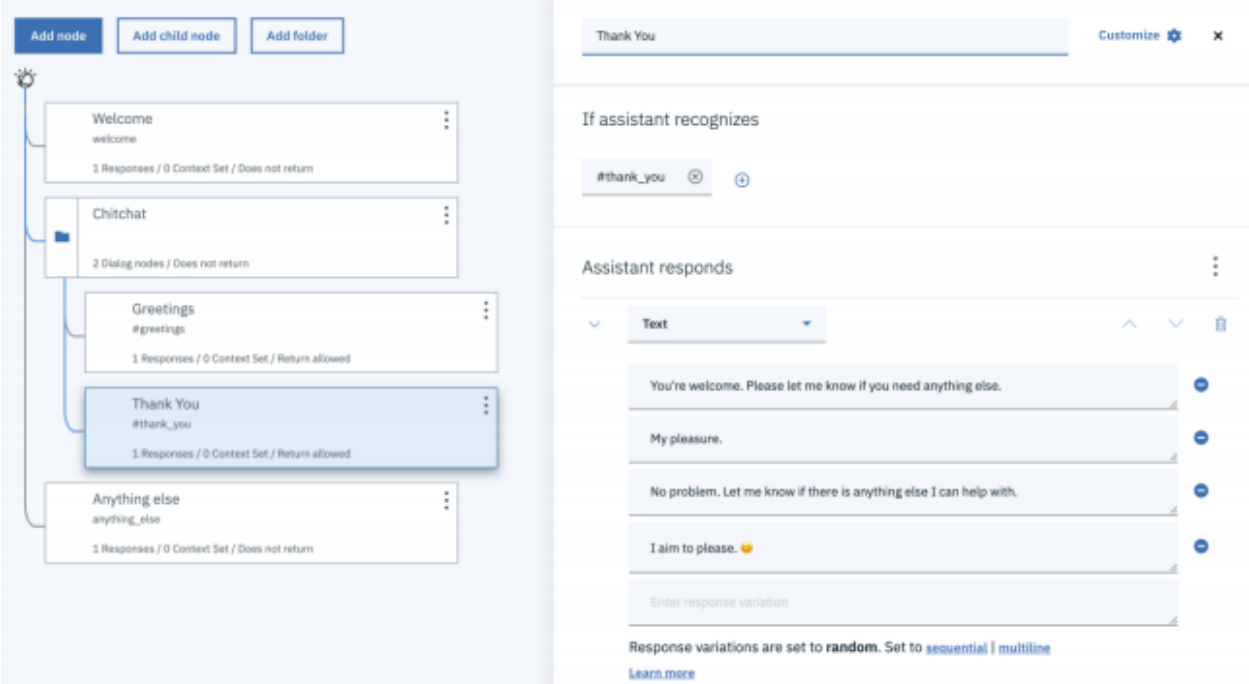
- No problem. Let me know if there is anything else I can help with.

You could get cheeky, and add:

- I aim to please.

Depending, of course, on how much personality you'd like to inject in your chatbot. BTW, yes, emojis are supported.

Set the response variation to random by clicking on the *random* link.



10. Repeat the process by adding a Goodbyes node that handles the #goodbyes intent. This

time make sure you use the *Thank You* node more options menu to select *Add node below* , since we want this third node to go below *Thank you*.

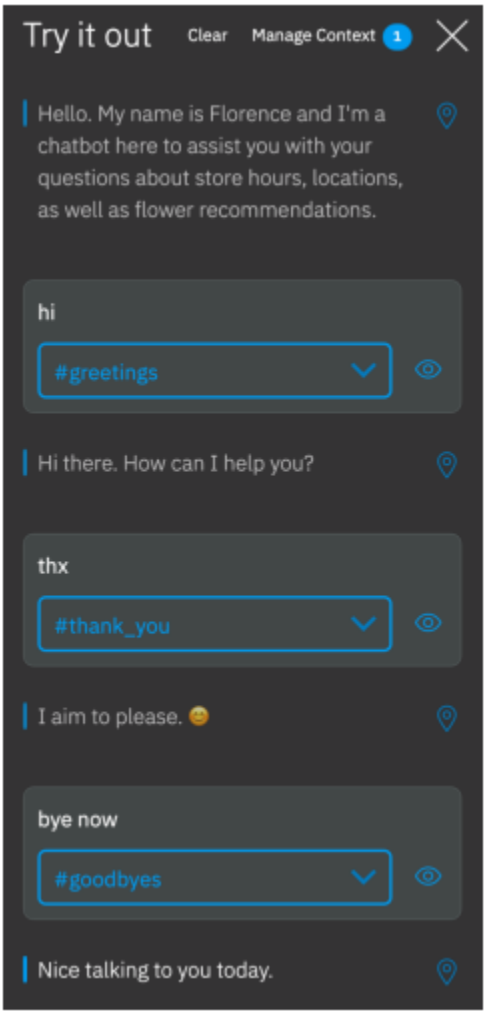
For now, you can use standard, polite goodbye responses such as:

- Nice talking to you today.
- Have a nice day.
- Goodbye.

(We'll improve these later on in the course.) Finally, set their order to random.

11. Start a new conversation in the *Try it out* panel and test all three intents to ensure you get

a proper response in each case. As you can see below, Florence is coming along quite well.



Lab 7 - Define Domain-Specific Intents

Objective for Exercise:

- How to define domain-specific intents.

Exercise 1: Respond to hours of operation requests

Chit chat interactions are necessary to make our chatbot more pleasant and human-like. However, what makes the chatbot actually useful is its ability to answer domain-specific questions. That is, business-related questions.

We defined intents for people inquiring about hours of operation and addresses of our fictional florist chain and even created an entity to be able to provide location-specific answers. However, much like the chit chat intents, intents alone don't offer responses to customers. We'll need to create nodes to handle these two business-specific intents.

Create the parent node

We'll start by creating a node for hours of operation. Follow these steps:

1. From the Dialog section, click on the *Add node* button. This will create an empty node just below the first node in the dialog.

> If the node was created elsewhere in the dialog, you know by now how to move it just below the *Welcome* node (hint: find the option in the three dotted/more options menu for that node).

2. Set the node name to Hours of Operation and use #hours_info for the node condition. This will ensure that the node will be executed when the user is inquiring about shop hours.
3. In the response, enter:

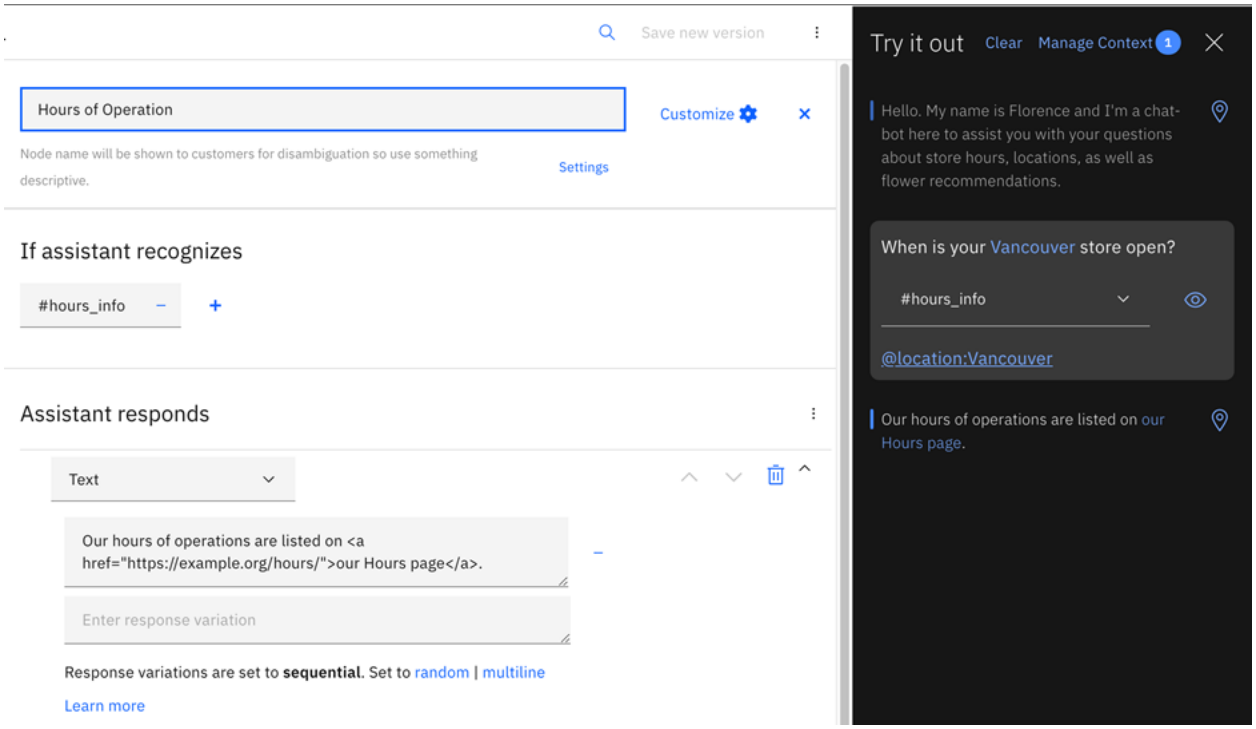
```
Our hours of operations are listed on <a
href="https://example.org/hours/">our Hours page</a>.
```

Notice how HTML code is allowed in responses, enabling us to provide more interactive and useful textual answers.

> Later, when you test this link, you'll be able to click on it and land on a sample page. If you were to get an error instead, consider replacing the double quotes with single quotes in the HTML above.

Next, close the node and head over to the *Try it out* panel to test that it works by asking:

When is your Vancouver store open?



It should hit the *Hours of Operation* node and give you its response as shown in the image above.

This works and it provides a somewhat useful answer to the user (assuming we are pointing them to a page with the right information listed). However, it feels... not very smart.

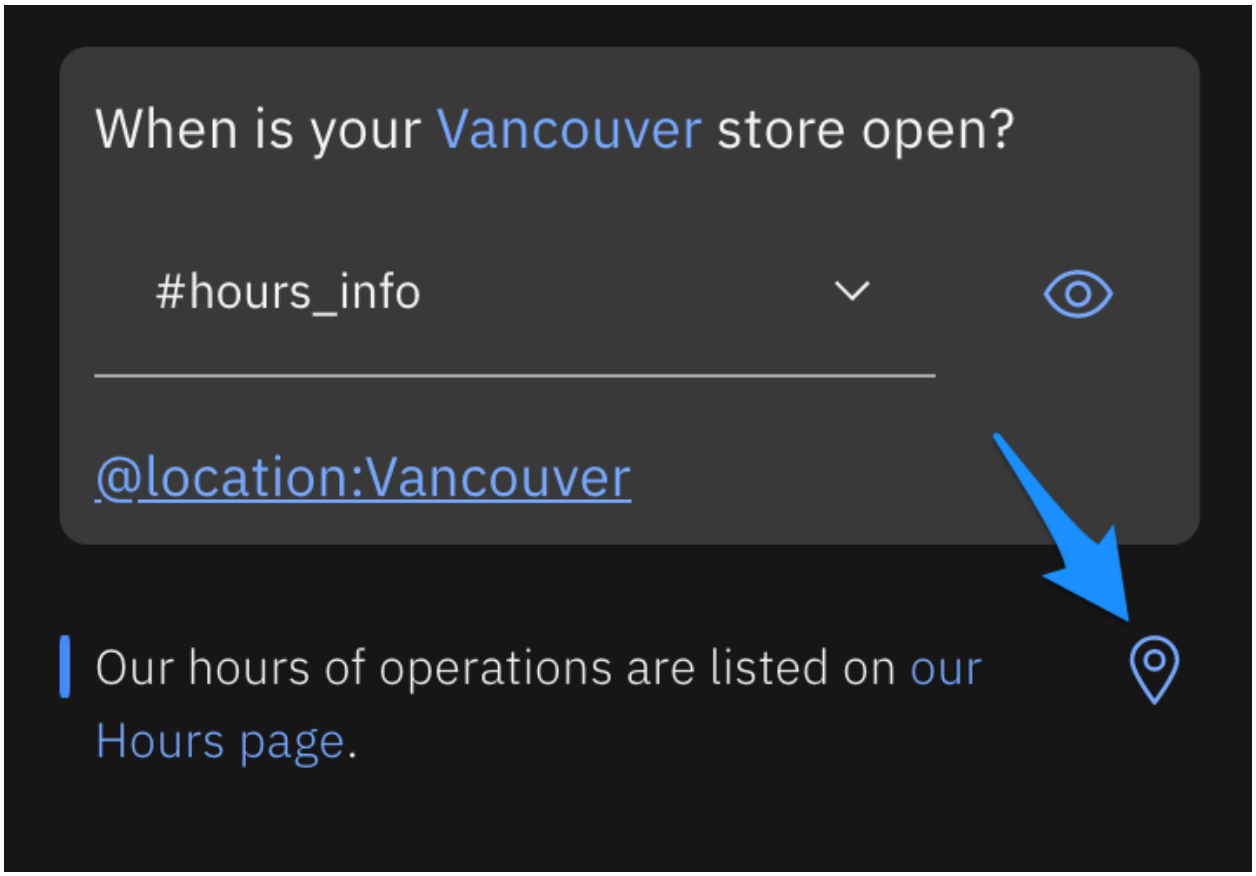
After all, the user asked us about a specific location. We even detected it with the @location entity and then proceeded to ignore it, opting instead for a generic answer. We can do better than that! (Close the *Try it out* panel to gain some breathing room as we work on the dialog.)

In order to handle this case properly, we'll have to consider two possible scenarios. One in which one of our locations is specified and one in which the user asks about hours of operation in general without indicating a city or indicating a city in which we don't have a store.

In the early days of Watson Assistant, before more powerful features were introduced, this scenario would be handled with child nodes. So, we'll use child nodes here and I'll show you the more advanced alternatives later on in the course.

We'll use our current node to capture the hours of operation request, and then jump to the child nodes to decide how to handle the request on the basis of the specific location information that was or wasn't provided.

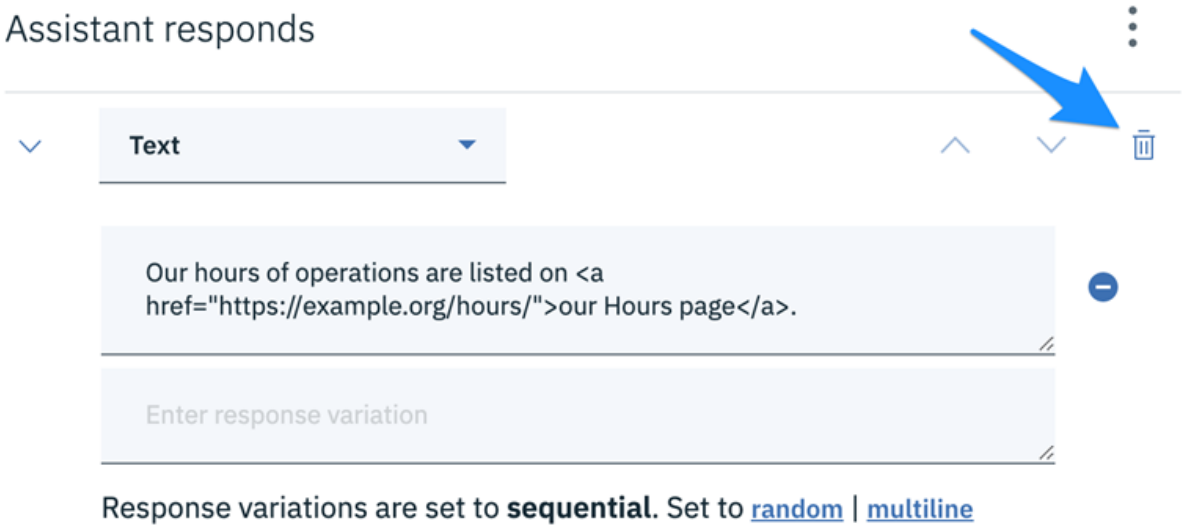
By the way, as the complexity of your dialog grows, you might have a hard time finding which node executed the response you're seeing during troubleshooting. To help you out, you can click on the map pin icon next to the response, and the current node will be highlighted for you.



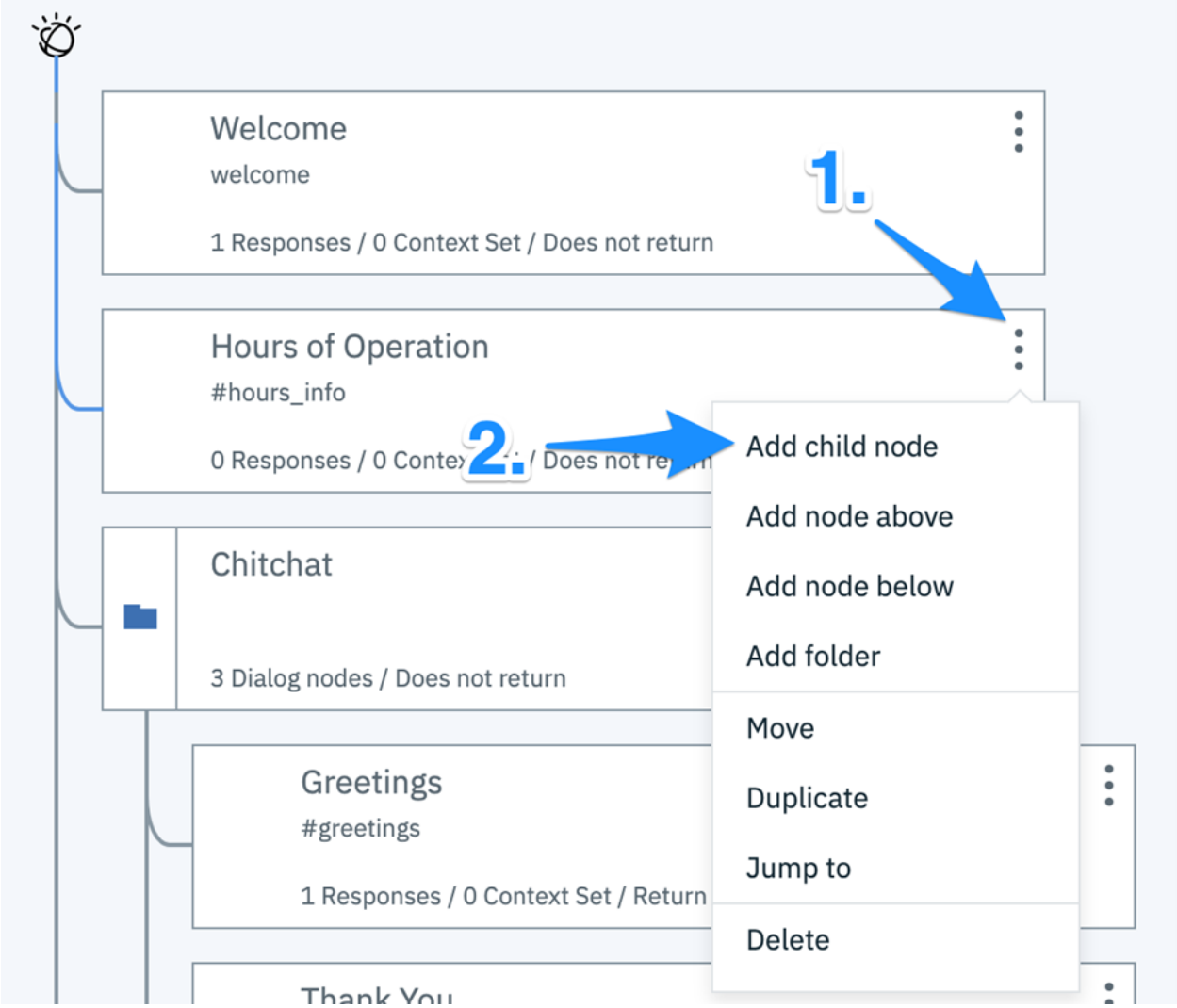
Create the Our Locations child node

1. Delete the response section from our *Hours of Operation* node by clicking on the trash bin icon in the *Assistant responds* section. We do that because we don't

want this parent node to provide the answer. We'll let the child nodes decide what's the right response.



2. Close the, now response-less, node. Using the more options menu for the *Hours of Operation* menu, , select the *Add child* node option, as shown in the screenshot below.



This creates the first child node. We'll use it for the case of the user providing us a city for which we have a flower shop. So, go ahead and name it Our Locations.

3. Set the condition to @location, as we want to execute this node only if the user is inquiring about hours of operation for one of our locations.

As a reminder, a child node is only executed if the parent node's condition was true, since we typically skip or jump to child nodes from the parent node. Or in the much less common case of another node jumping to it. In our dialog, if we are executing this child node, we can be certain about two conditions regarding the user input:

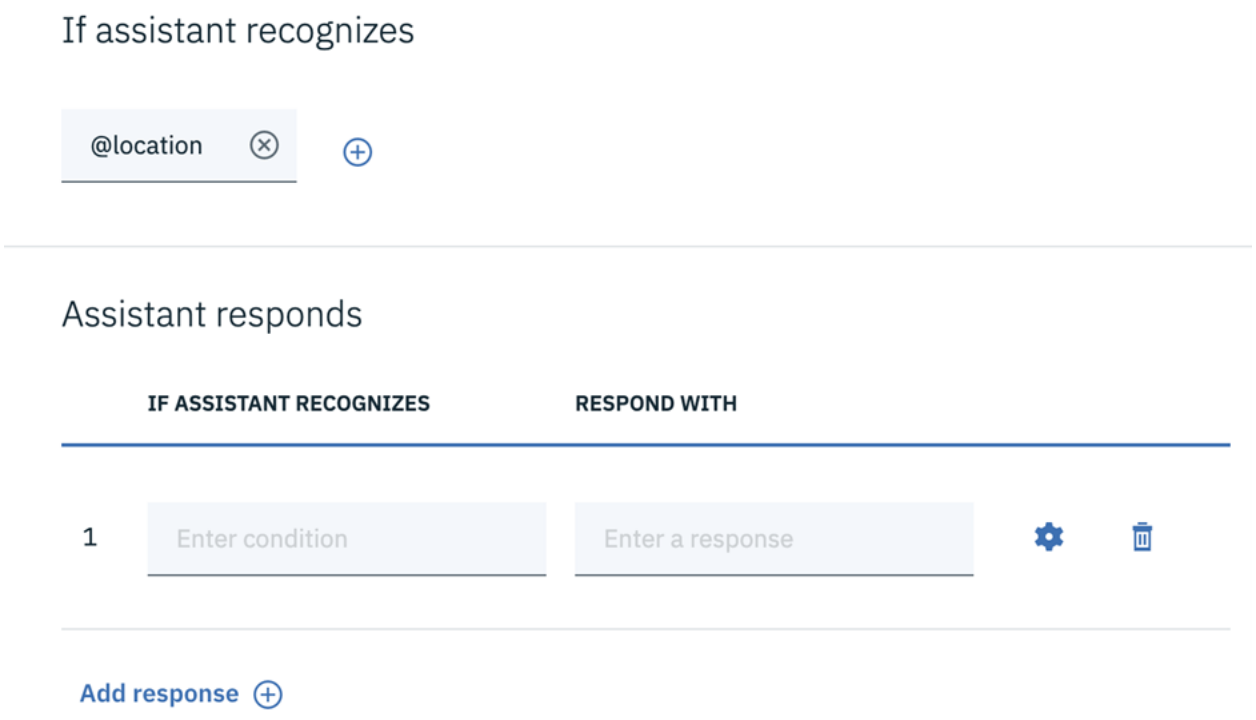
1. The intent will be #hours_info (because the parent node must have been executed to jump to this child node);

2. The input will contain the @location entity (since that's the condition for this node to be executed).

Knowing this allows us to provide very specific responses.

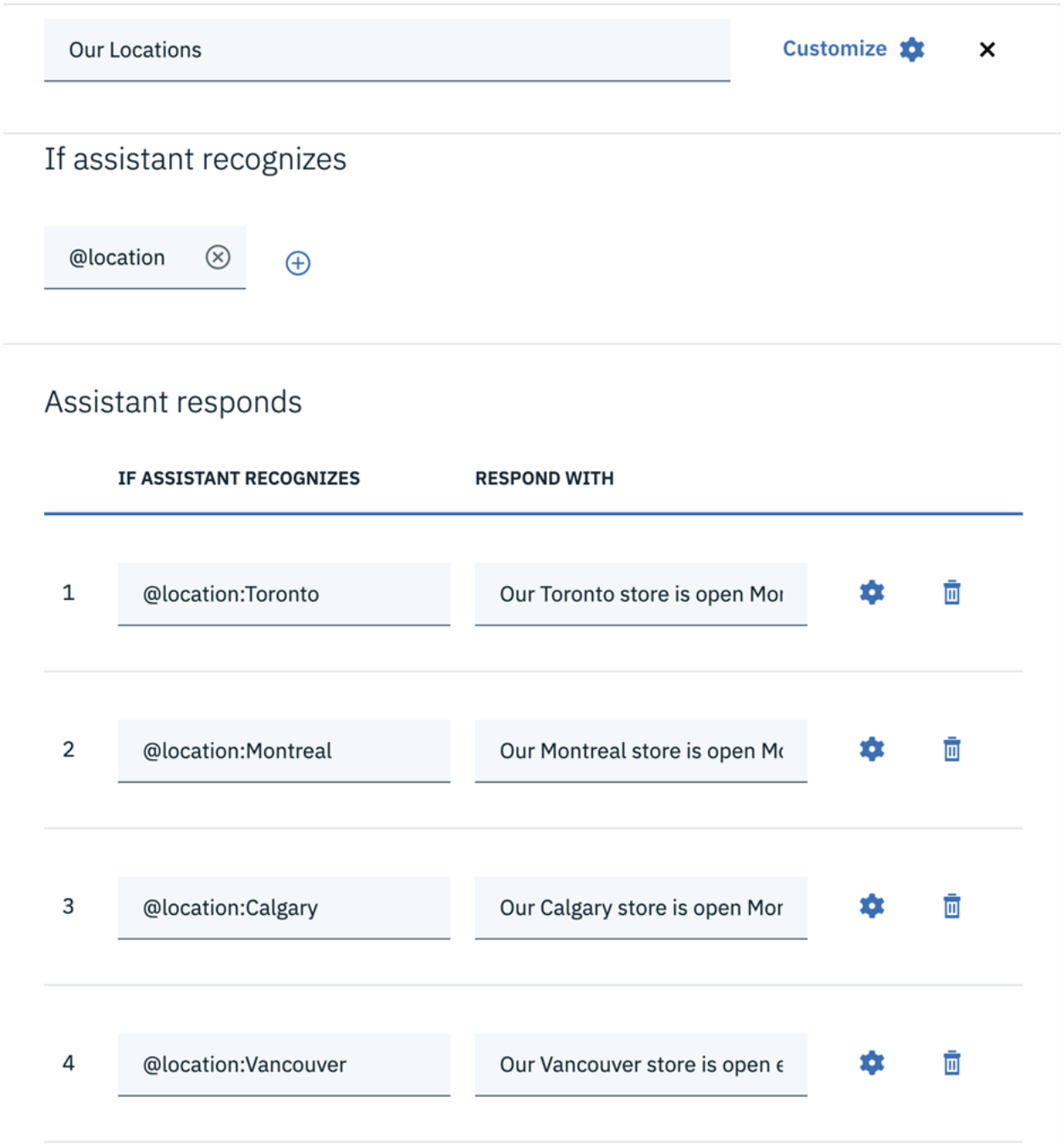
(Okay, technically we haven't made the parent, *Hours of Operation*, jump to its first child, *Our Locations* yet. Fear not, we'll do so as soon as we are finished setting up this child node.)

4. We need a way to offer a different response for each city, so we need to enable *Multiple conditioned responses*. To do so,click on the *Customize* link within the *Our Locations* child node. Scroll all the way to the bottom and switch on *Multiple conditioned responses* and click *Apply*. You'll notice that now we have the ability to attach a condition to each response, as shown below.



5. Go ahead and create a series of responses, one for each city. In the *IF ASSISTANT RECOGNIZES* column you'll want to enter the specific city (e.g., @location:Toronto) and in the *RESPOND WITH* the hours of our fictional flower shop location (e.g., Our Toronto store is open Monday to Saturday from 9 am until 6 pm, except statutory holidays.)

Use Add *response* to add additional entries for each location we have. Feel free to come up with fictional hours of operation, as it is, after all, a fictional retail chain. The end result should be similar to the image below.



It's worth noting that if the hours of operations where the same for all locations, we could have saved the trouble of switching to multiple conditioned responses and simply

included @location in our response. (e.g., Our @location store is open Monday to Saturday from 9 am until 6 pm, except statutory holidays.)

This would automatically output the detected entity value back to the user in the response. So, when enquiring about Calgary, the user would receive the response Our Calgary store is open Monday to Saturday from 9 am until 6 pm, except statutory holidays. Of course, if stores have different hours of operation, we need to opt for the multiple response approach like we did here.

Close the *Our Locations* node by clicking on X in the top corner.

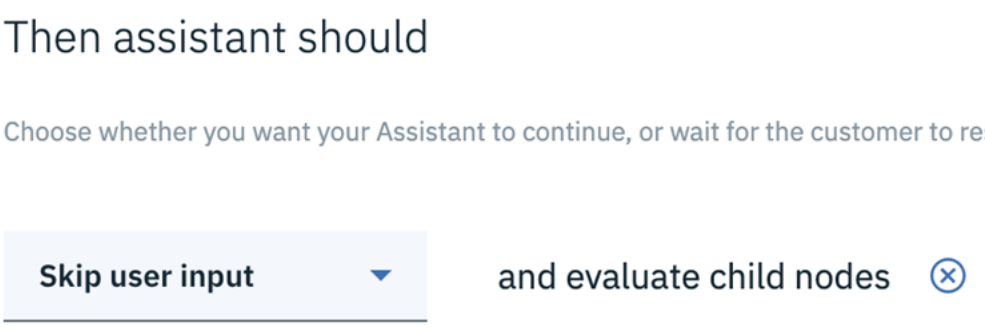
Skip the user input and evaluate the child nodes

Now that we have our child node defined, we need to set the jump from its parent node.

In other words, we need to make sure that the parent node (i.e., *Hours of Operation*) hands off control to the child nodes. We'll jump to the first node, and if the condition for that node is false, we'll continue evaluating other child nodes (that we'll define in a moment) until one child node's condition is met and that child node is finally executed.

Select the *Hours of Operation* node , and you'll notice that the *Then Assistant should* section is set to *Wait for reply*. This is not what we want. The user has already provided us with the question, and we haven't responded yet, since this node has no response.

Change this section of *Hours of Operation* to *Skip user input*. ** This will hand off the execution to the child nodes.

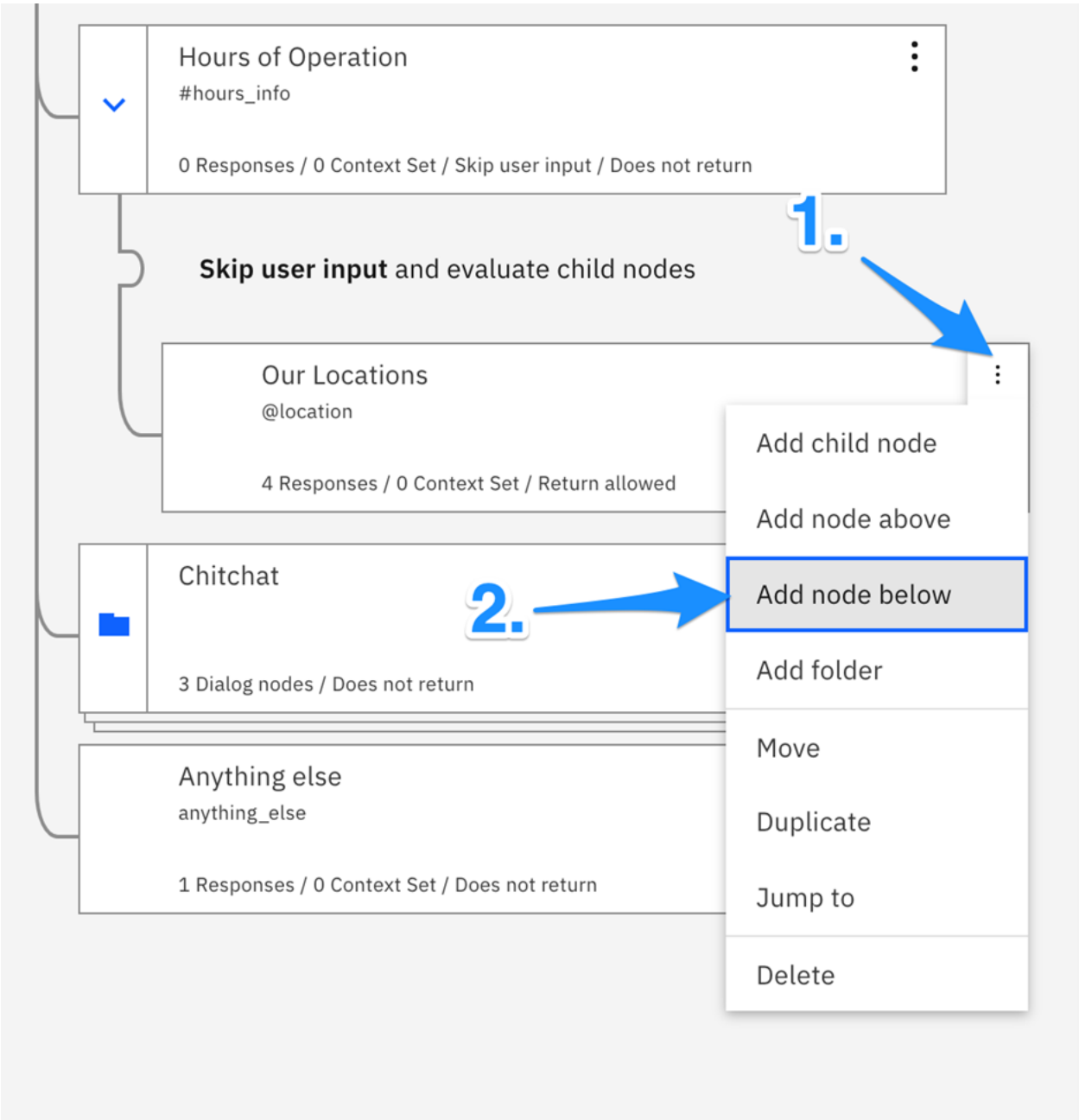


Finally, close the node.

Create the No Location child node

We now have a way to handle users asking about hours of operation for a specific location of ours, however, we also need a child node to handle the case in which the user didn't specify a location (or for a location that we don't recognize).

1. Using the more options menu for the *Our Locations* node, select *Add node below* to create a new node This will create a node below *Our Locations*, and *Hours of Operation* will end up with two child nodes in total.



2. Call this node No Location. Set its condition to true.

Here is why. When the user asks, “What are your hours of operation?” the #hours_info intent is detected, so when the dialog is evaluated, we enter the parent node *Hours of Operation*.

The *Our Location* child node is then first evaluated. We fail its condition because the user didn't specify any location, so the next child node is considered for execution.

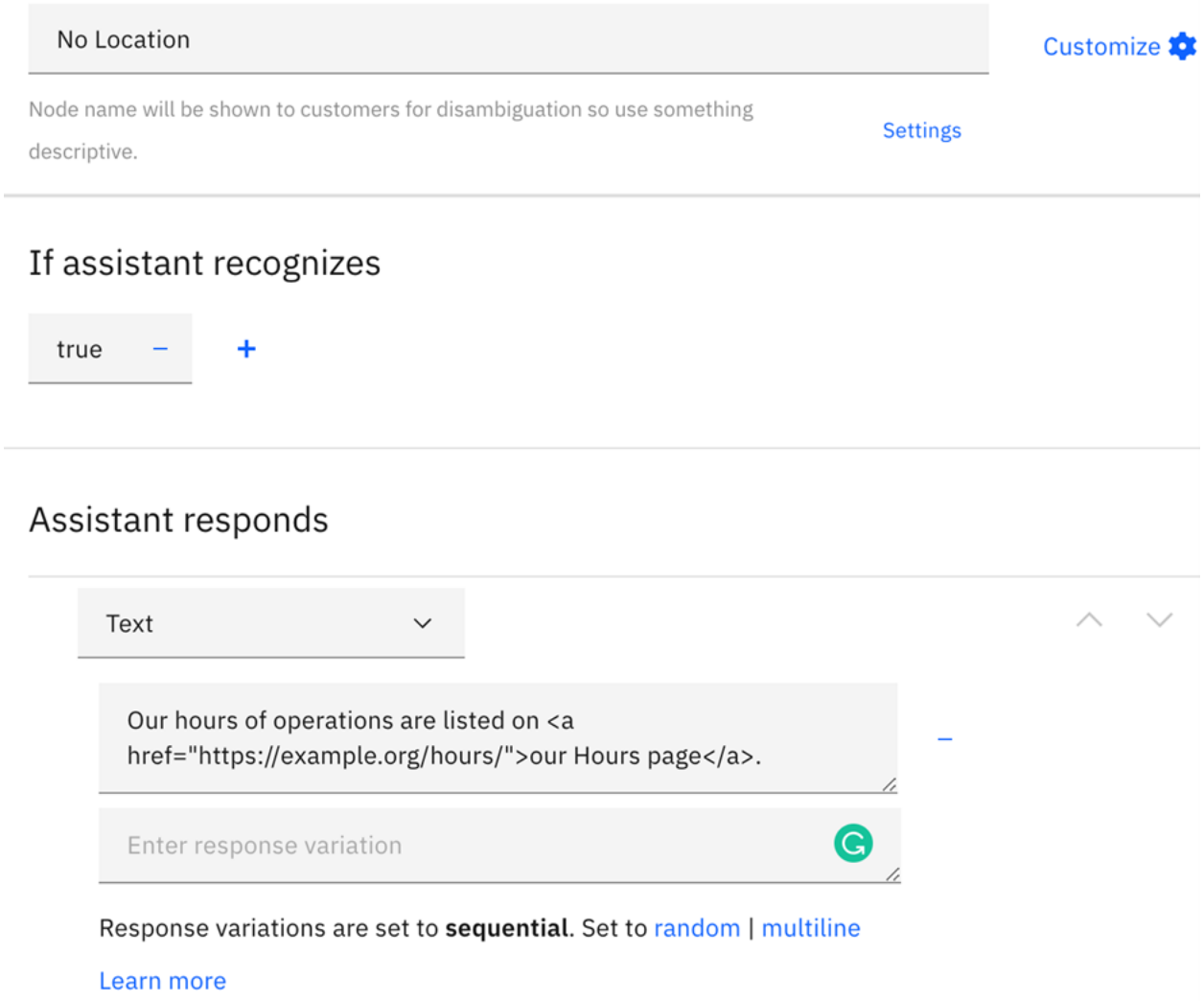
Since the condition is set to true it will automatically be executed. This is exactly what we want to happen since at this point, we know the user wants to know the hours of operation, but no location of ours was provided. (If we left the condition empty, we'd get an error because no child node was able to match the user request.)

Note that the order of your nodes can matter. For example, placing the *No Location* node with its *true* condition above *Our Locations* would overshadow *Our Locations* and it would never be executed. Instead, *No Location* would be executed each time, which is not what we want if the user specified one our locations.

So, the order of our nodes didn't matter in the case of the chit chat nodes, but it matters here. The general rule is to always put specific cases above more generic cases, to avoid overshadowing the specific nodes with the more generic ones.

3. We need a generic answer for when no location is specified, so go ahead and reuse the message we had originally.

```
Our hours of operations are listed on <a
href="https://example.org/hours/">our Hours page</a>.
```



4. Click on the *Try it* button, click *Clear*, then try the following inputs (one at a time):

What are you hours of operation in Toronto?

What are your hours in Calgary?

What are your hours in Seattle?

What are your hours of operation?

Try it out Clear Manage Context 1

Saturday from 9 am until 6 pm, except statutory holidays.

What are your hours in Calgary?

#hours_info

@location:Calgary

Our Calgary store is open Monday to Friday from 9 am until 6 pm, except statutory holidays.

What are your hours in Seattle?

#hours_info

Our hours of operations are listed on our Hours page.

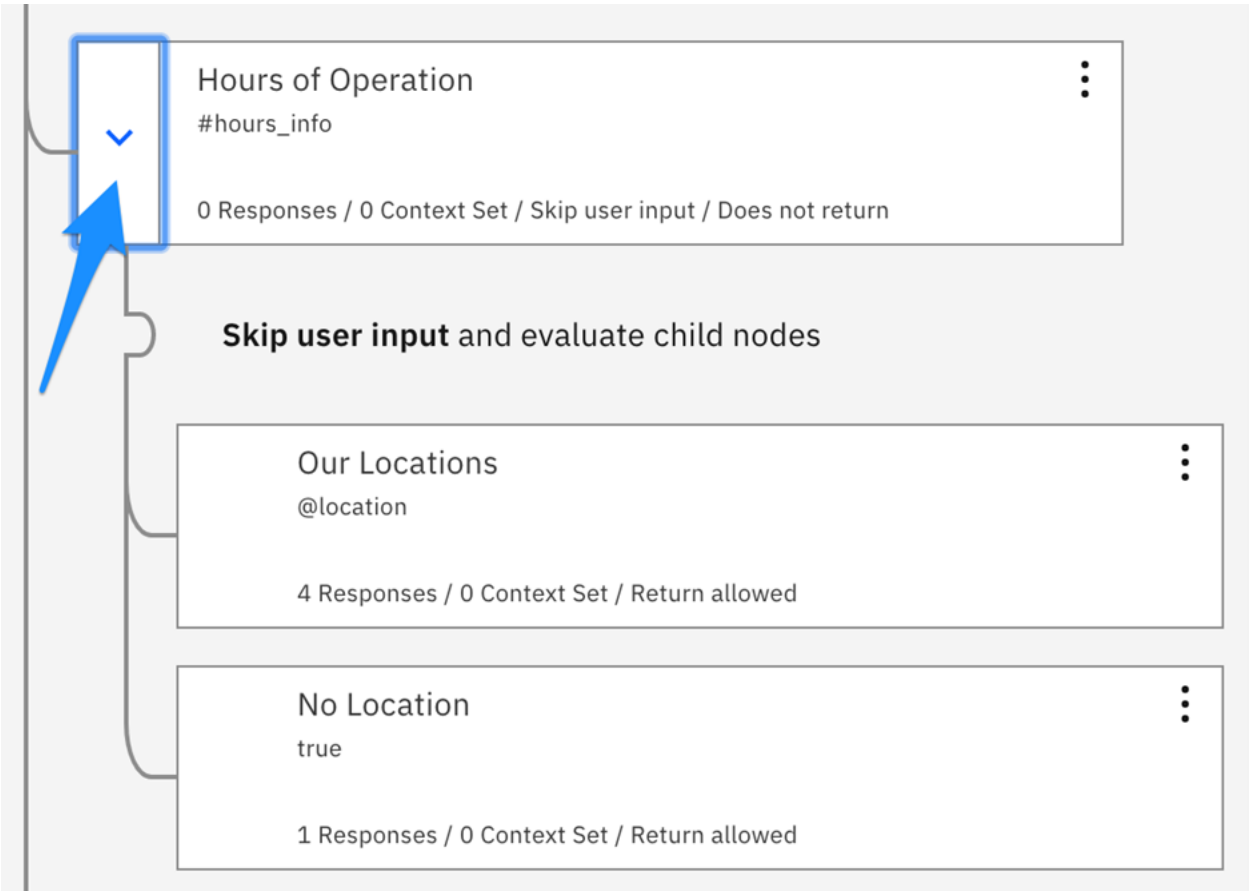
What are your hours of operation?

You should see a proper response for each of these inputs. Not bad!

Exercise 2: Respond to address requests

Our little chatbot is getting more useful by the minute. We now need to handle location address requests. And guess what? It's no different in terms of how this works. We'll have a parent node and two children to distinguish both scenarios.

By the way, at any time, you can click on the arrow on the left of the *Hours of Operation* node to collapse or expand its children.



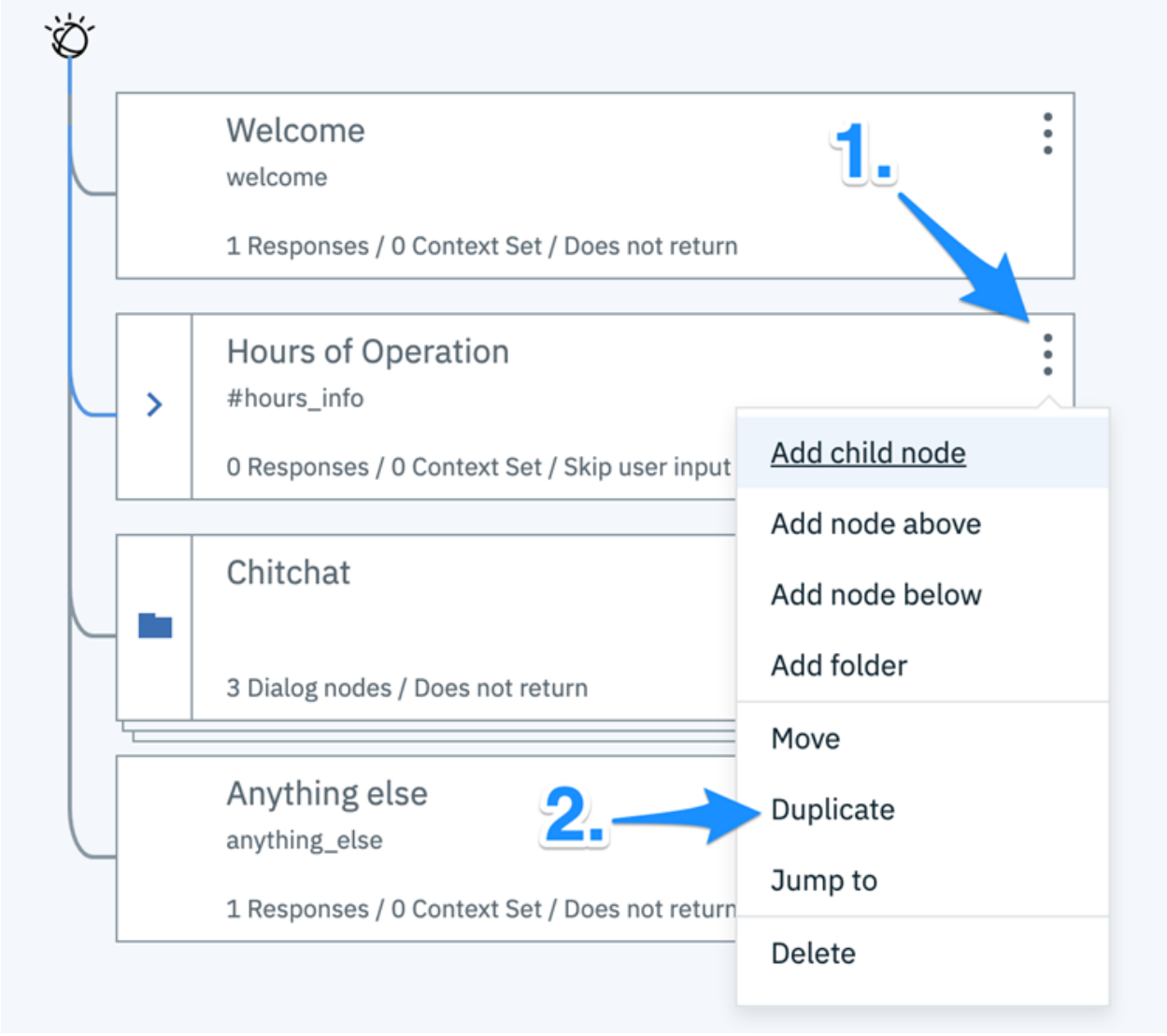
Collapse its child nodes now to gain some breathing room.

In the new parent and child nodes to handle address requests we'll need to change one condition (#location_info instead of #hours_info) and change the responses from hours of operation to actual addresses. Overall though, the structure is identical.

So, you could redo the process in Exercise 1 above, step by step, with those two minor adjustments, or we can be more efficient and simply duplicate *Hours of Operation* and change its copy to our needs.

We'll opt for this more efficient route:

1. Click on the more options menu for the *Hours of Operation* node and select *Duplicate* to make a copy of *Hours of Operation* and its children.



2. Select the *Hours of Operation - copy1* node that was generated. Change its name to Location Information and its condition to #location_info. Its name will allow us to distinguish it from its remarkably similar sibling *Hours of Operation*, and the

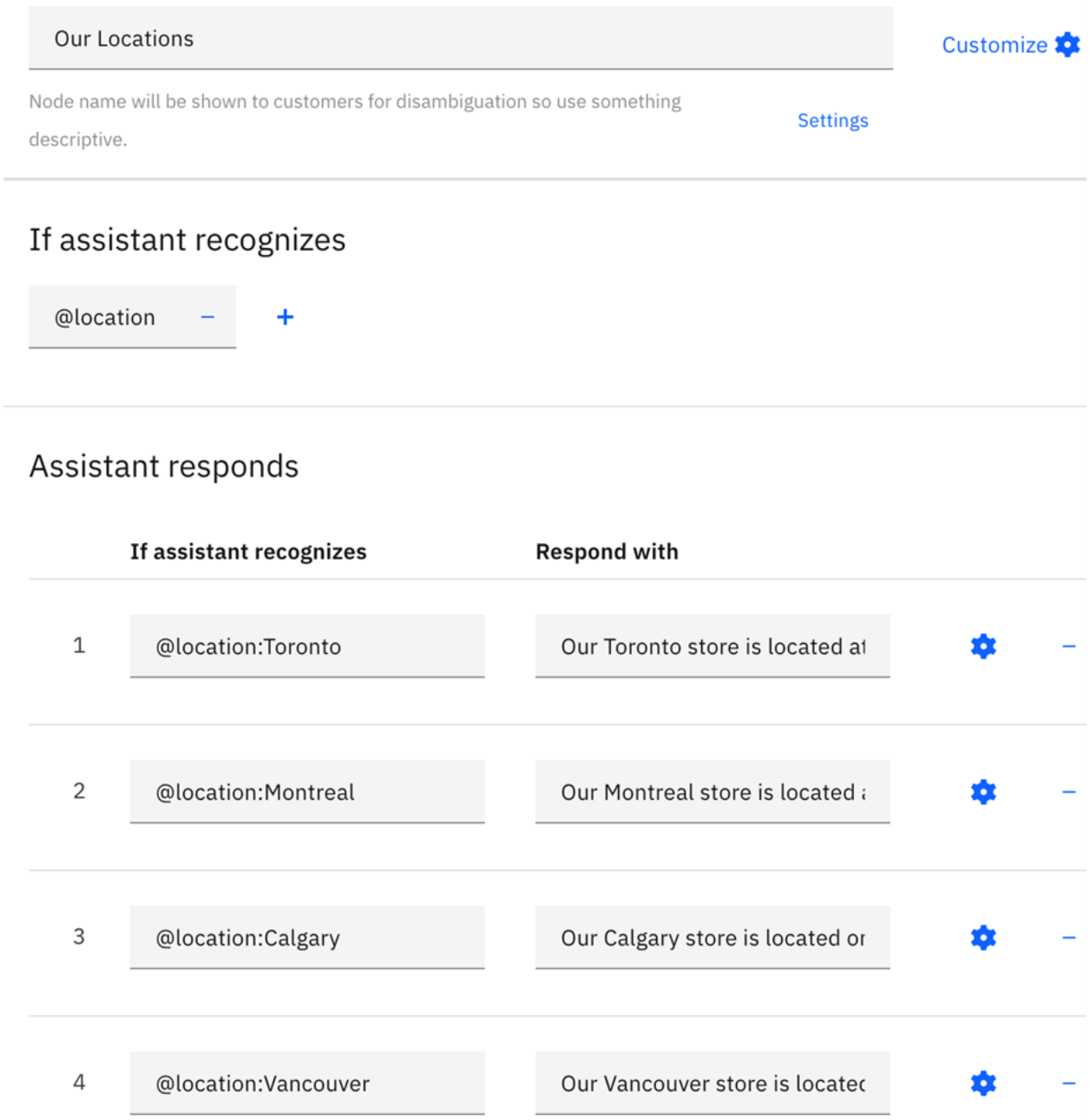
condition will ensure that Watson will only execute the node when the user asks for an address not hours of operation.

3. Next, we'll need to change the responses in two child nodes within the *Location Information* tree.

Feel free to get creative but here is the type of response you should assign to each city in the *Location Information* > *Our Locations* child node

Our Toronto store is located at 123 Warden Avenue.

Add fictitious address for all the cities in that node.



Likewise, in the more generic Location Information > No Location child node replace the response to say:

```
Our store locations are listed on our site on the <a
href="https://example.org/stores">stores page</a>.
```

4. Open the *Try it out* panel, press *Clear* to start a new conversation, and test out a full conversation a user might have with our chatbot. Enter in succession the following input.

hello

where are you stores located?

what are your hours of operations in Montreal?

thank you

bye

If you followed the instructions so far, you should have a pretty neat conversation. We can, of course, flesh out our chatbot much more, but if you got to this point, you have mastered the fundamentals of what you need to know to create something useful that cuts down of many common inquiries from your customers.

We'll soon see how to deploy the chatbot, and then tackle more advanced topics in the process of improving the chatbot's usefulness and apparent degree of intelligence.

Are child nodes really necessary here?

Technically speaking we don't need child nodes to handle the two scenarios we implemented above. We could simply add multiple conditional responses to the parent nodes and add responses for each of the cities and the catch-all true case, all from within the same node.

However, I wanted to show you how to work with child nodes, the importance of their ordering, and their flexibility. If the logic was more complex than just a generic response, having a dedicated child node to handle it would likely be a good idea, anyway. In some complex chatbots, you might even have child nodes that have their own child nodes!

Later in the course, we'll get rid of child nodes in favor of something called *Slots*. For now, please keep the two child nodes below both *Hours of Operation* and *Location Information*, as you defined them in this lab.

Congratulations on completing lab 7!

Objective for Exercise:

- How to add a preview link.
- How to generate a wordpress site.

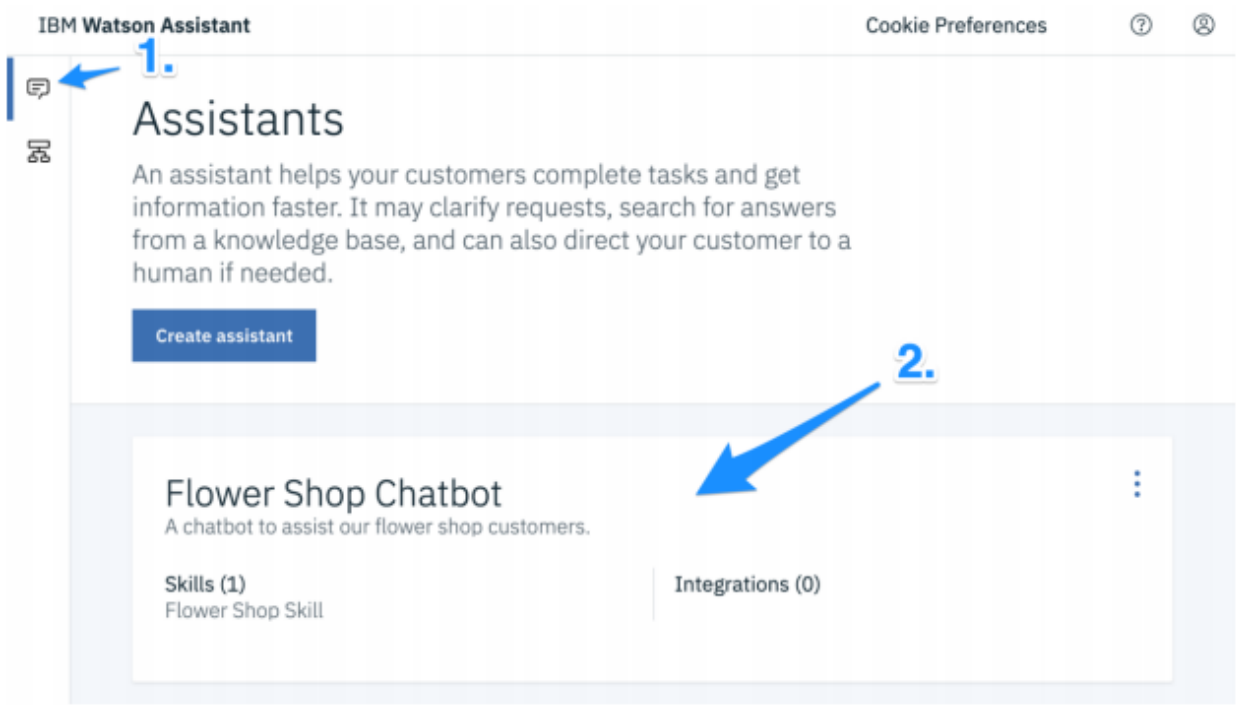
The small chatbot we built so far works well enough from the *Try it out* panel. That's great and all but our customers won't have access to it unless we deploy it somewhere. Let's see how to accomplish that.

Exercise 1: Add a Preview link

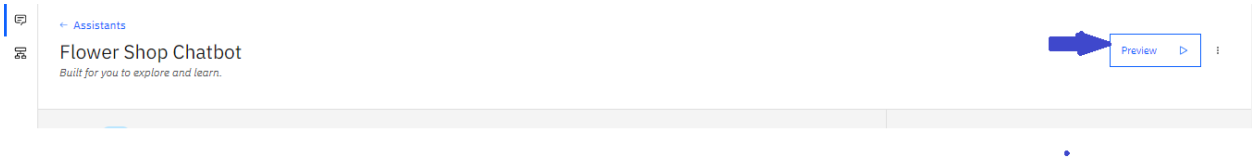
Assistants within Watson Assistant have an *Integrations* section from which we can select various way to deploy our chatbot. Before we look at how to deploy our chatbot on WordPress, in particular, it would be good to take advantage of the *Preview link* integration.

This *Preview link* can be shared with friends and colleagues who'd like to try out your chatbot.

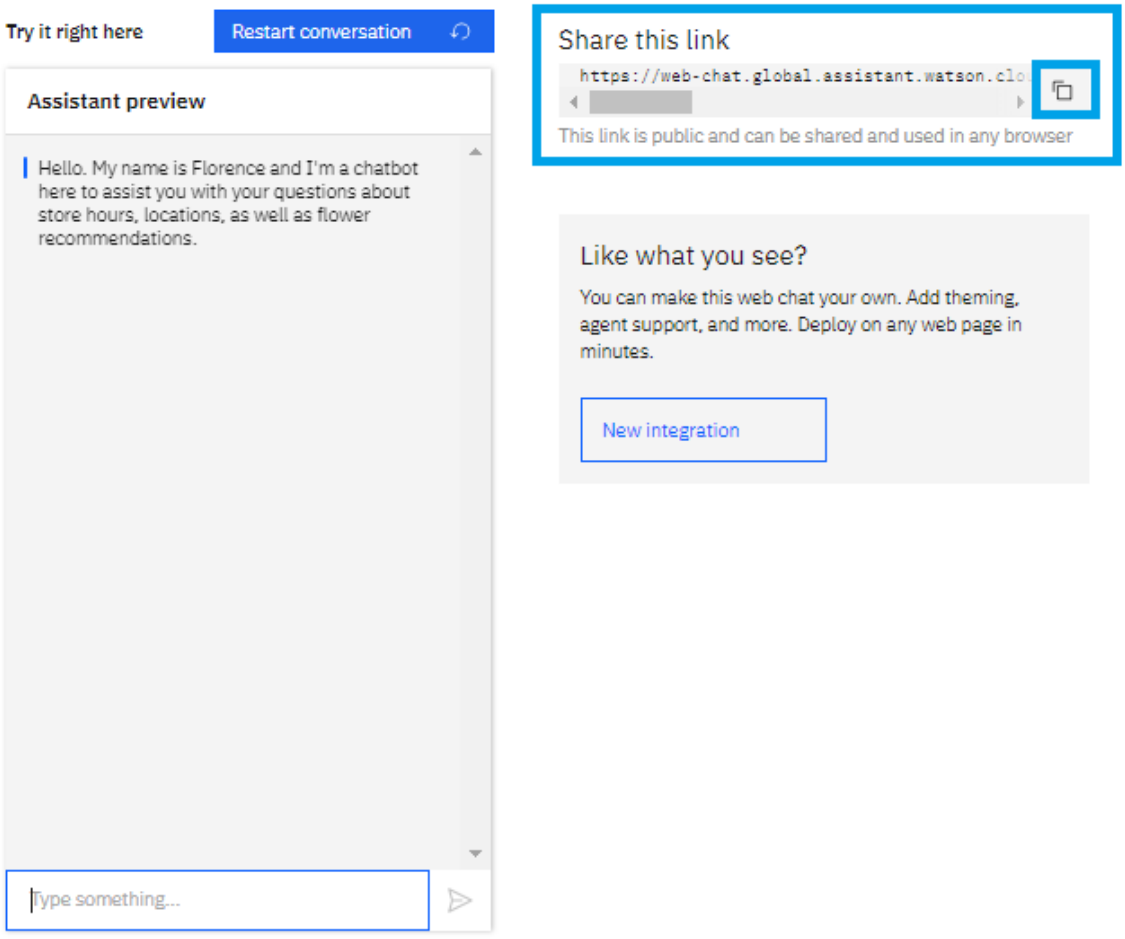
To enable the *Preview link* , first head over to the *Assistants* tab. You'll notice that 0 next to Integrations. That's because we don't have any integrations enabled quite yet. Click on the tile for your chatbot.



From within your assistant, click on Preview.

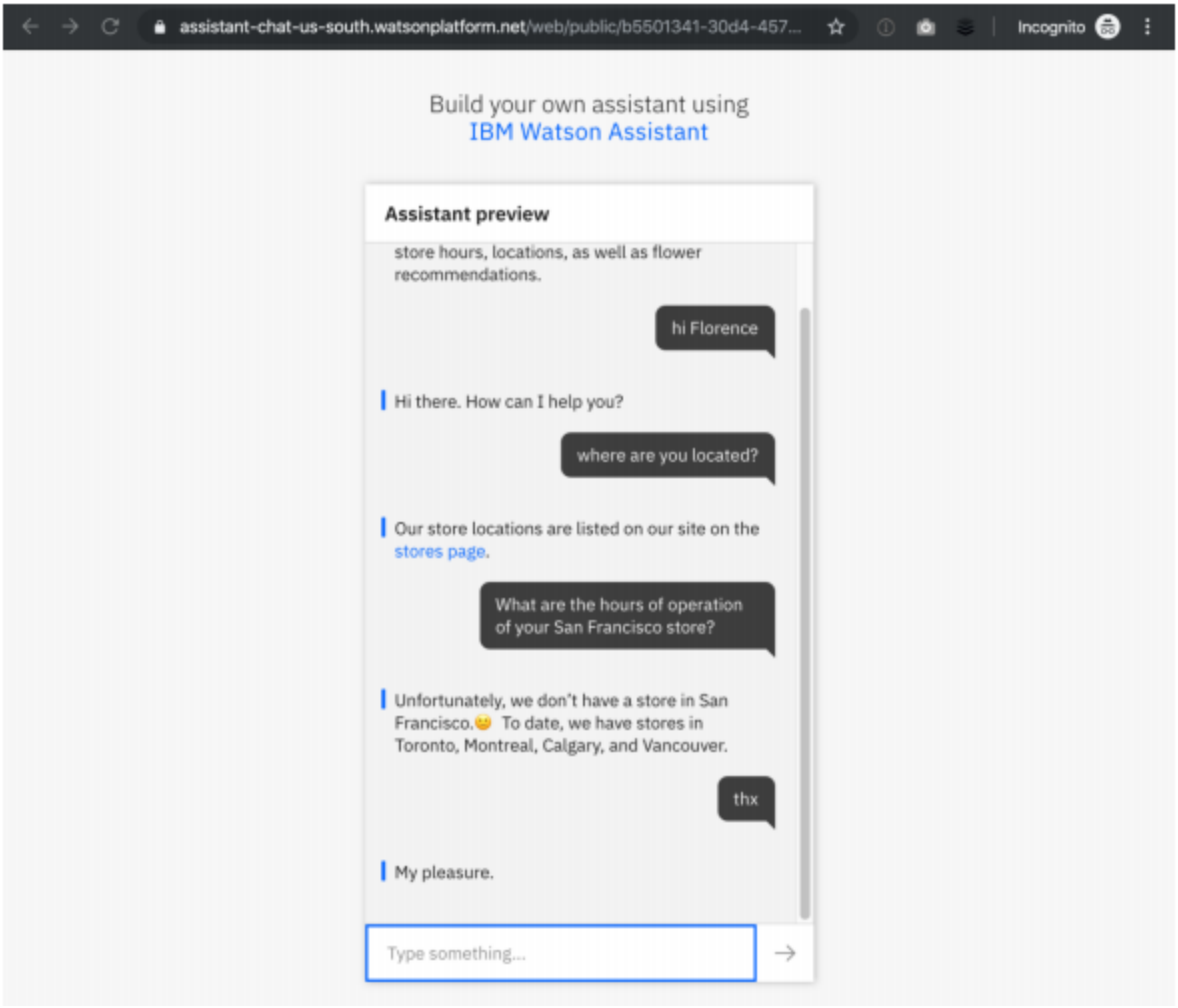


A link will be generated for you. Click on the Copy icon, that's your *Preview link* that you can share with others who are interested in trying out your chatbot. then .

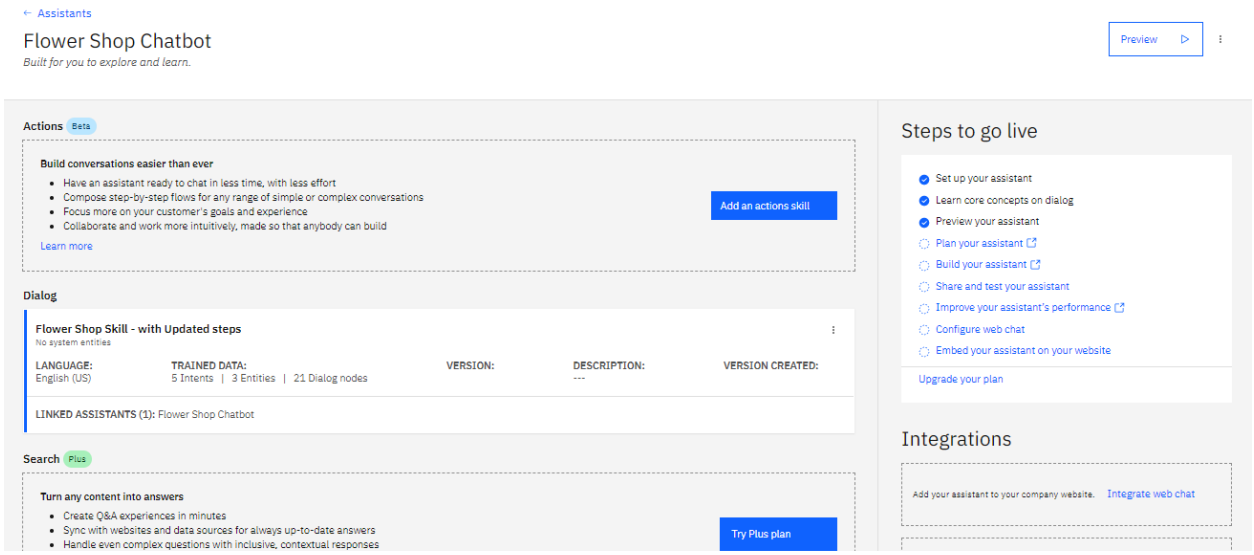


Do bear in mind that every time someone sends a message to the chatbot, one API call is made, and it counts towards your free allowance (10,000 API calls per month in the Lite plan).

Test it out to verify that the chatbot we built so far, does indeed work correctly from this user interface.



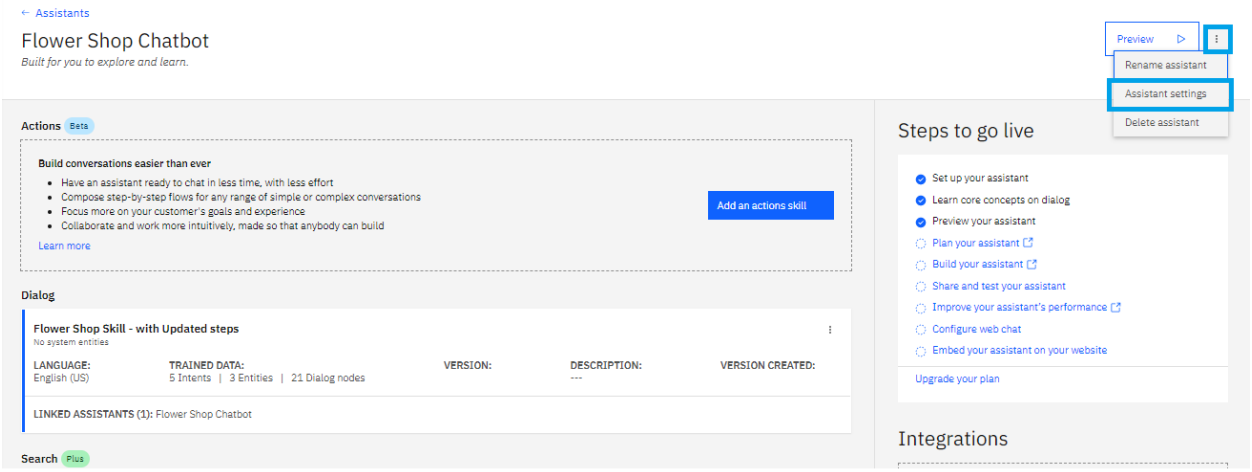
This will add the preview link to your Integrations.



Exercise 2: Retrieve your chatbot credentials

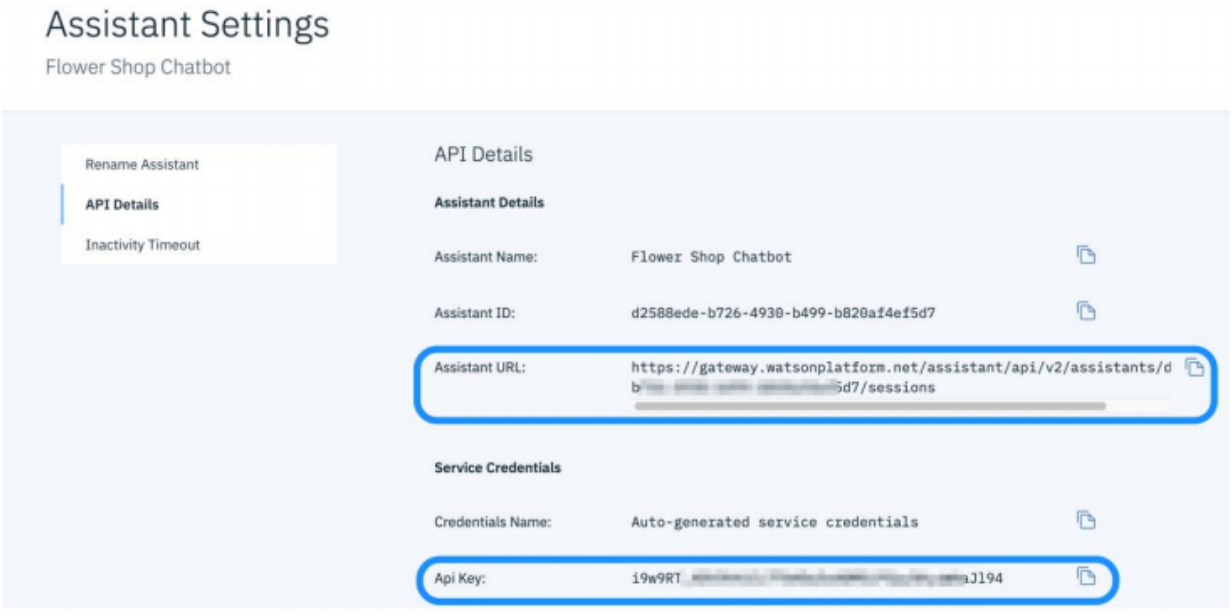
The preview link is quite handy but to actually deploy our chatbot in production, we'll want to collect our assistant's credentials and make note of them.

Click on the more options menu for your *Flower Shop Chatbot* assistant, then select *Assistant Settings*.



From the settings, click on *API Details* .

Make note of the *Assistant URL* and the *API Key* . You'll need to know them in order to successfully deploy your chatbot later on.



Make note of them now and then click on the X to close the API credentials page.

Exercise 3: Generate a WordPress site

You followed along and now have a simple Flower Shop chatbot running in your Watson Assistant service. That's great, but how do we place it on an actual site?

WordPress is a content management system that allows anyone to quickly have a website up and running. This platform has a lot of features out of the box, and many more can be added through plugins.

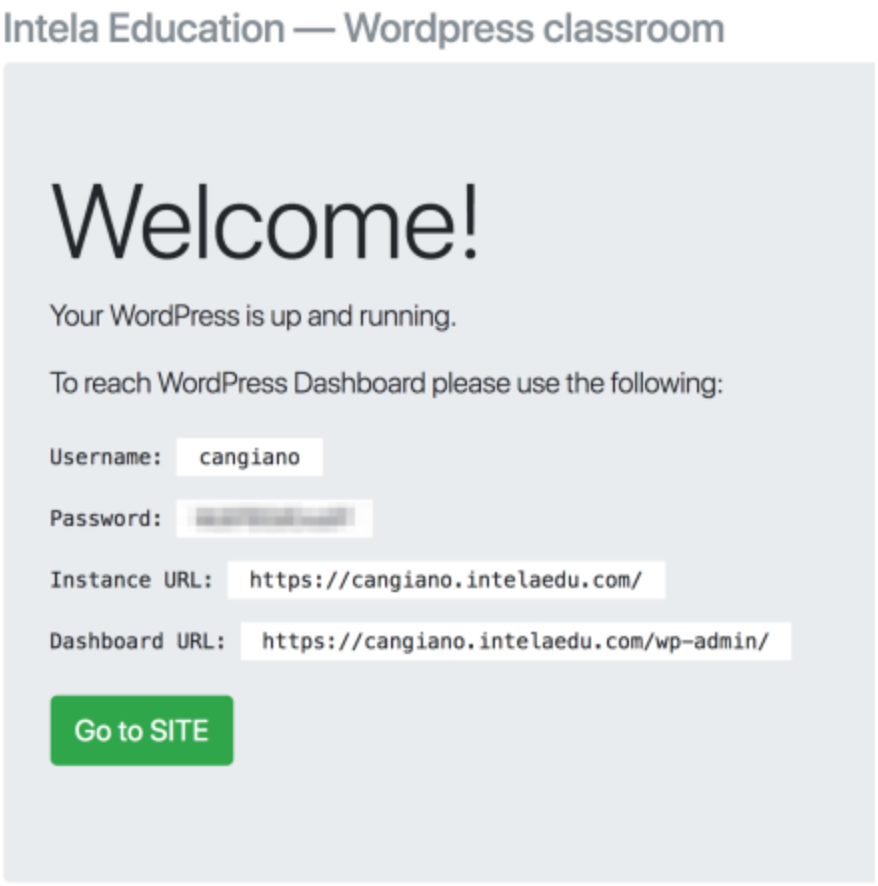
We developed one such a plugin for Watson Assistant to make it extremely easy to place a chatbot on a WordPress site.

We'll discuss the plugin in the next lab. But first, we need to create a WordPress site.

In the next section of this module, you'll find a button that will allow you to *Generate a WordPress site*. Click on it to generate your site.

```
Do not create a WordPress.com site. Generate the site using the tool
provided. WordPress.com
expects you to pay to be able to install plugins. The WordPress(.org)
installation we give you
already has the plugin installed.
```

You'll be given details about your site, similarly to the ones shown in the figure below.



Please make note of these WordPress credentials you'll be given upon generating the site, you'll need them to log into the site in the next lab.

In particular, write down somewhere your generated WordPress *Dashboard URL* (where you'll log in), your *username* , and your *password*. (Note that these are WordPress credentials and therefore different from the API ones you wrote down earlier in this lab).

If you lose them, you can always come back to the next section of this module (i.e., *Generate a WordPress site*) and obtain them again.

Without further ado, go ahead to the next section and actually generate your WordPress site.

Please note that this site is for testing purposes only. Do not use it as your main site as it might be shut down after a certain period of time.

A note about Assistants and Skills

Assistants have one or more skills. Skills are linked to particular assistants.

You don't normally have to worry about this because a default assistant and a skill (already linked to each other) were automatically generated for us when we created our Watson Assistant instance.

If, in the future, you were to create a new assistant, you'll want to make sure to link it to a skill.

Note that when you deploy your chatbot, you generally want to use the credentials from the assistant (like we did in this lab) and not from its skill.

Lab 9: Deploy your Chatbot

Objective for Exercise:

- How to deploy a chatbot to your website.
- How to customize the chatbot window.

Exercise 1: Deploy a chatbot to your website

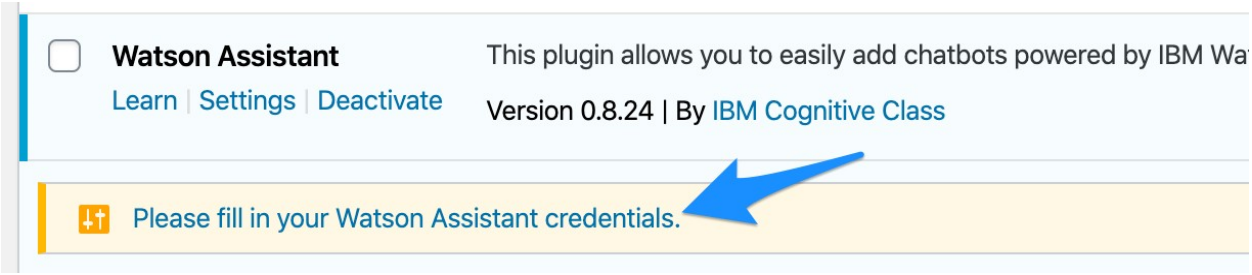
1. Log into the WordPress site you just generated. Visit your *Dashboard URL* and use the

credentials you obtained in the previous lab to log in.

2. Activate the Watson Assistant plugin. In the *Plugins* section of your WordPress Dashboard,

you'll find a few plugins that were installed for you. Click on the *Activate* link under the plugin for Watson.

3. Click on the link prompting you to provide your credentials.



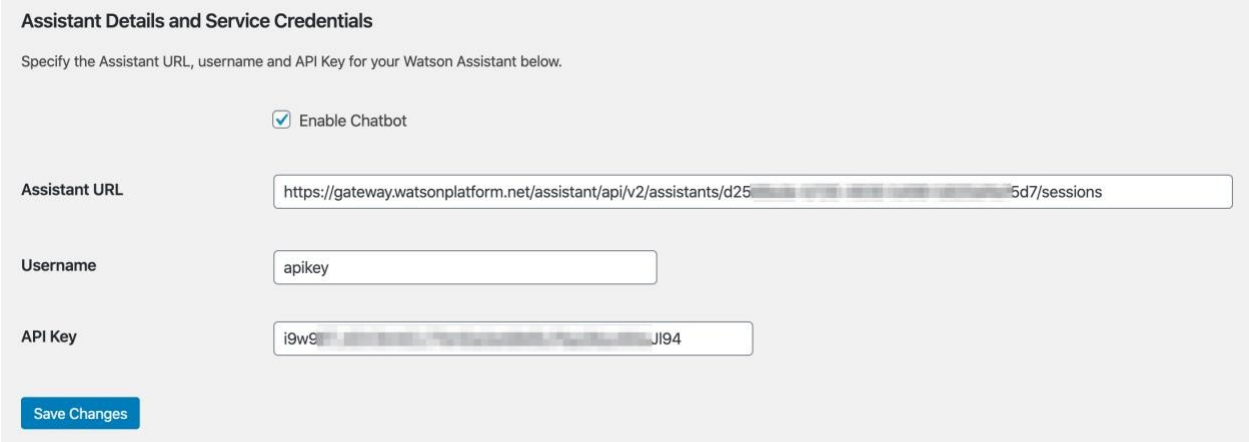
4. Next, click on the *Plugin Setup* tab. Here specify your Watson Assistant credentials for the

chatbot assistant we created in the previous lab.

```
As a reminder, you'll just need your Assistant URL and API Key. If you
don't know where to find
them, review the previous lab again as instructions are provided in Lab 8.
```

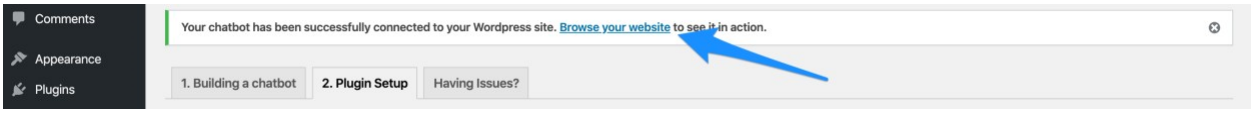
4. Make sure that the chatbot is enabled. In the future, should you decide to temporarily

disable the chatbot, you'll be able to do so from this page by deselecting the checkmark next to *Enable Chatbot*.



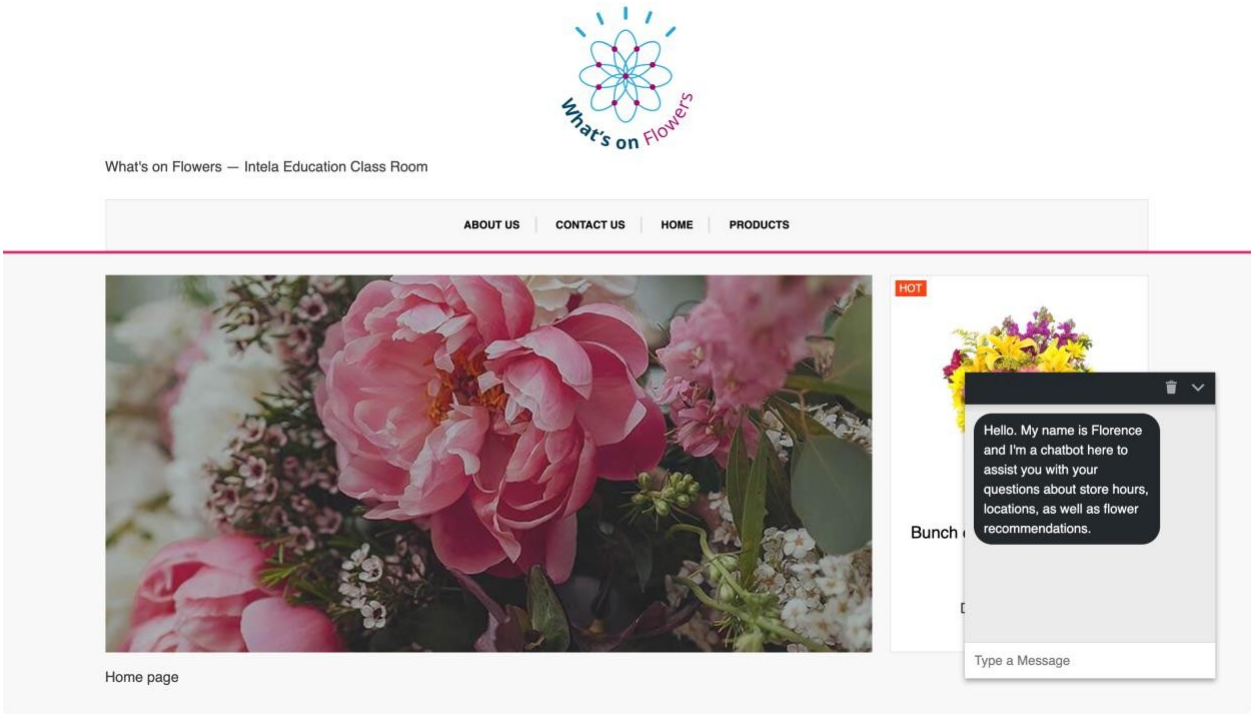
5. Click on *Save Changes* and a message should appear inviting you to Browse your website to

see the chatbot in action at the top. *Click on that link* or simply head over to the Instance URL you made note of earlier on.



6. If you see a chatbot pop up greeting you with the prompt you defined, congratulations, **you

just deployed your chatbot**. That was quite straightforward, wasn't it? Go ahead and *test your chatbot directly through this chat box*.



Exercise 2: Customize the chat box window

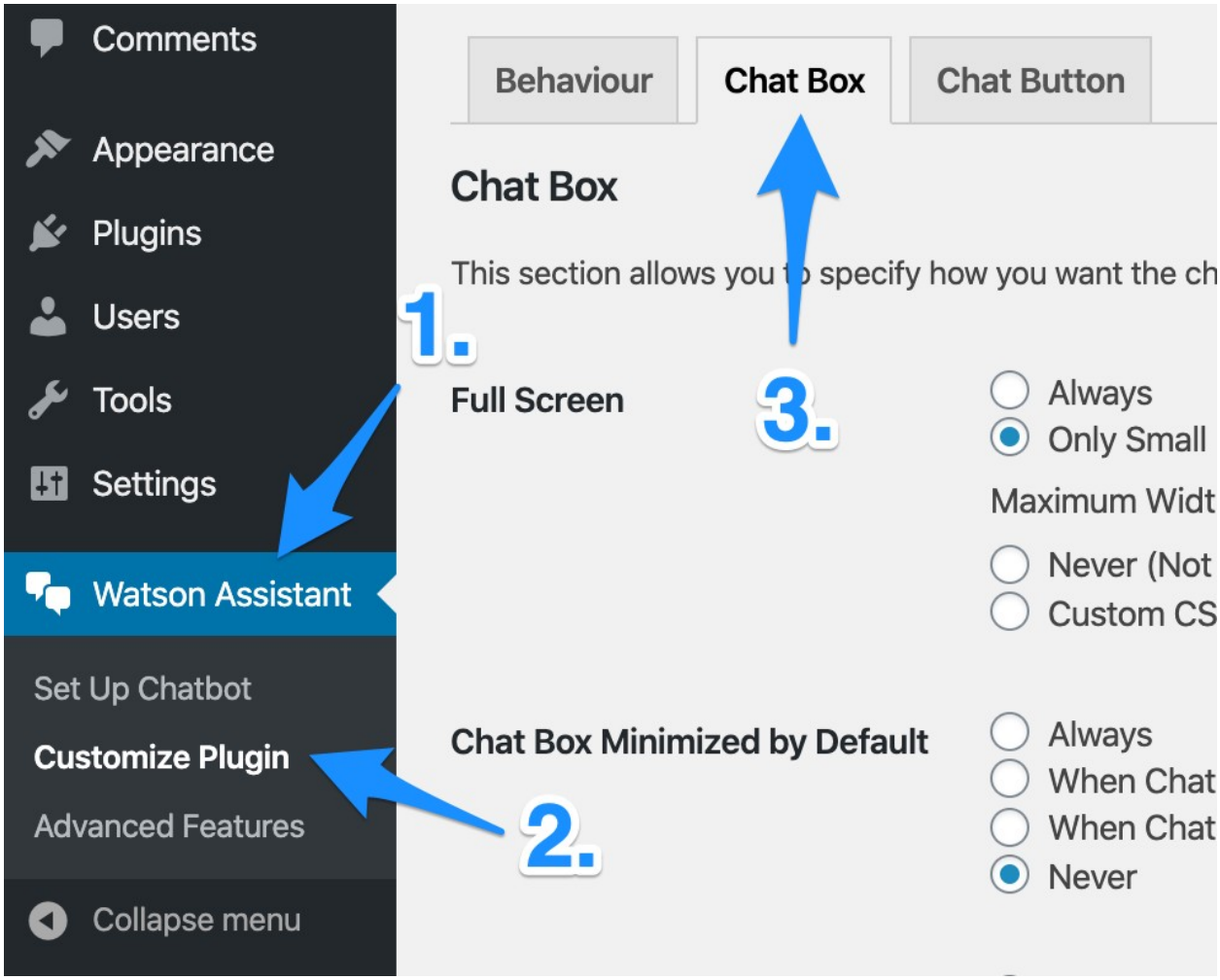
Now that our chatbot is deployed we can take a moment to celebrate our accomplishment. Whenever we make changes to the chatbot on Watson Assistant, these changes will be reflected on our already deployed chatbot. So, we could literally walk away from the WordPress site now, and just focus on Watson Assistant improvements.

However, before we do so, I'd like for you to spend some time customizing the chatbot look and feel on the site. Specifically, the look of the chat box.

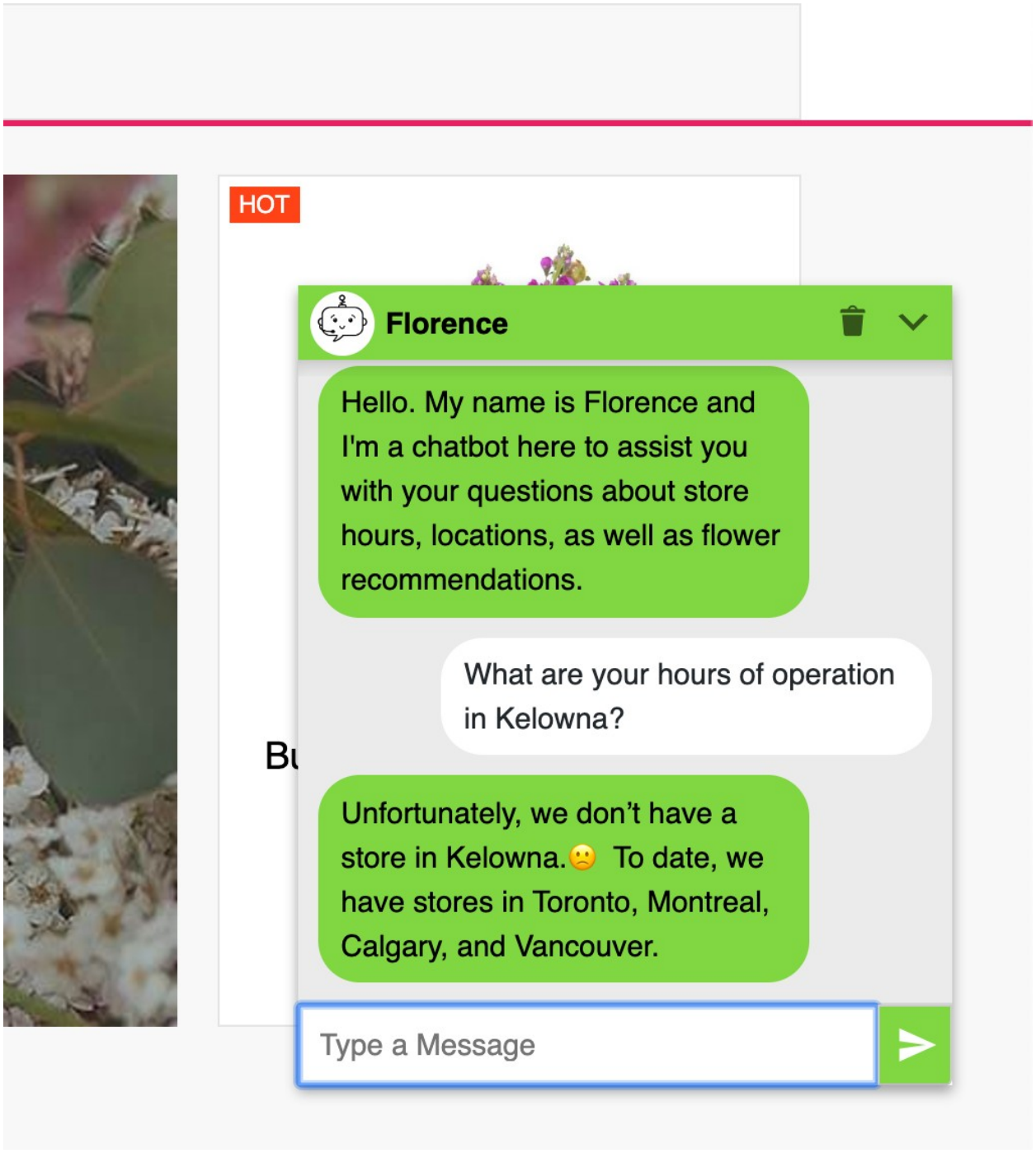
1. Go back to your WordPress Dashboard. **Click on the *Watson Assistant* link within your

WordPress dashboard** sidebar (towards the bottom of the page).

2. Next, click on *Customize Plugin* and then the *Chat Box* tab.



Spend some time to customize the chat box. Change some of the options within this tab and then visit your WordPress site to see how they affect the look of your chat box. For inspiration, see how I customized it in the image below.



Exercise 3: Familiarize yourself with the plugin options

The plugin is divided into three sections: *Setup Chatbot, Customize Plugin*, and *Advanced Features*. Take some time to explore these three sections.

Some of the options are for features we haven't discussed yet or are out of scope for this introductory course. But it's good to know what options the plugin has to offer and, as always, if you have any plugin-specific questions feel free to ask them in the forum.

Lab 10: Explore Context Variables

Objective for Exercise:

- How to use context variables.

By now you know the essentials of chatbot building. There are however more advanced concepts that will enable you to create better and smarter chatbots.

I could list them all here at once, but I think it makes more sense to organically introduce them as their need arises in the process of improving our chatbot.

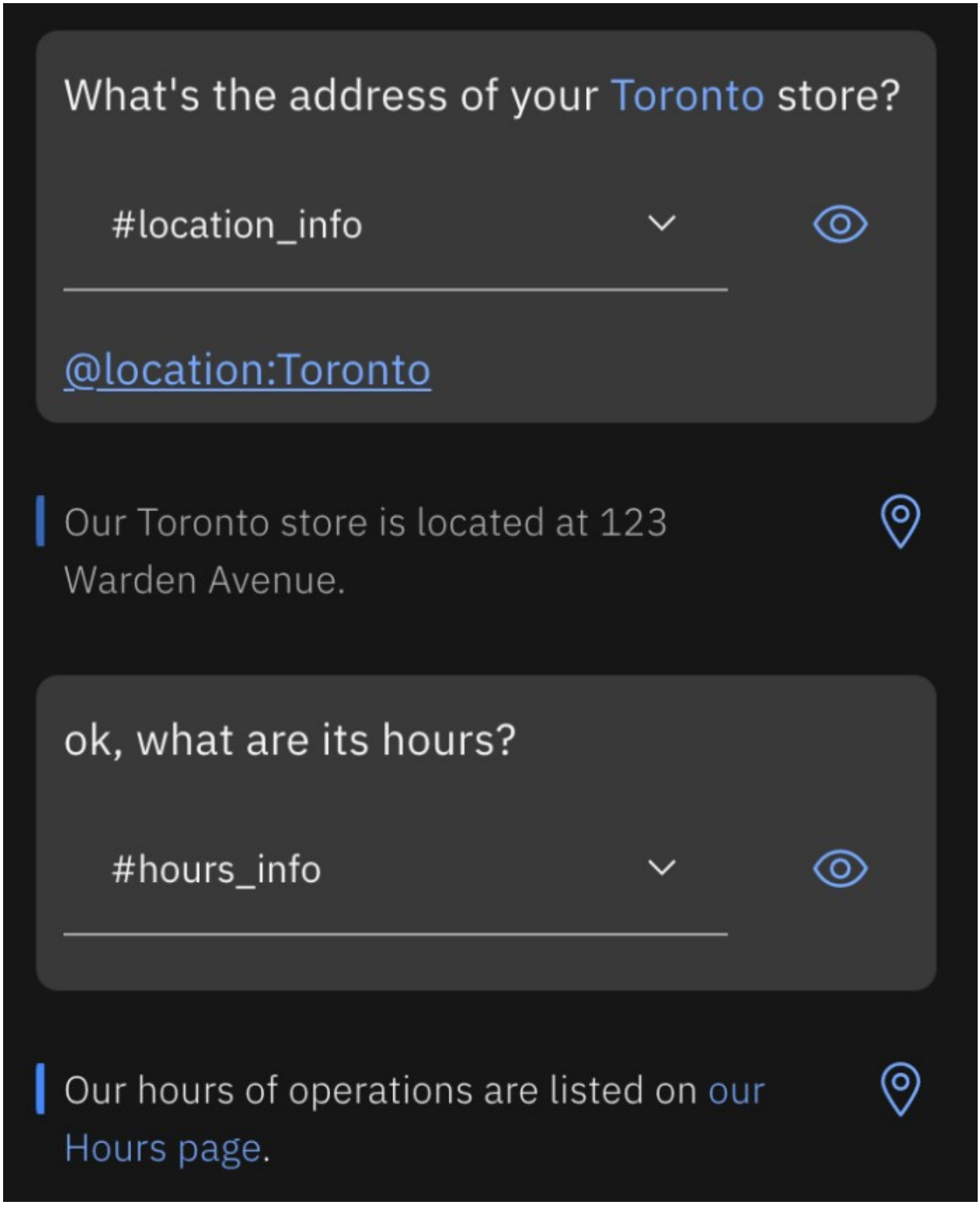
Keep in mind that some of these concepts are tougher to get, particularly if you have no prior programming experience. So, don't be discouraged if you don't fully get everything right away.

You can try things out, test to see if they work, and if they don't, try something else. That's why the *Try it out* panel is so useful. It allows you to build chatbots one feature at the time. Stick with it, and if you practice, you'll quickly become familiar with the advanced concepts as well.

Exercise 1: Remember the city with context variables

Whenever a user enters a new input, the intent and entities that are detected don't stick around for the rest of the conversation. They exist at the moment, for the current input, and are forgotten once the user types more questions.

This is generally fine, but it limits the chatbot's ability to appear smarter and remembering the context of the conversation so far. For example, consider the following interaction.



A human customer care agent responding to the second question would have inferred that the user is asking about the hours of operation for the city they just inquired about in the previous question (i.e., Toronto).

However, the entity detected in the first input only lives for the duration of that input, so our chatbot has no memory of it when the user enters a second question.

How can we store this information so that it’s available for the duration of the conversation? Enter the concept of context variables, which allow us to do just that. As we go about collecting information from the user, we can store it in the Context and then reuse it when it makes sense.

One way to achieve this is to create a passthrough node that checks for the *@ location* entity and sets it to the *$city* context variable if one is detected. It then jumps to the next node in the dialog and hands off the execution to the rest of the nodes as if this node didn't exist.

Keep in mind that this is not necessarily the best approach, but it allows us to demonstrate a couple of things:

1. The passthrough node technique which can come in handy in complex chatbots;
2. How context variables work.

So, let's see how this would work in practice.

Creating a passthrough node

1. In the *Dialog* section of your skill, select the *Welcome* node more options menu, and **click

Add node below** to create a sibling node underneath (as a reminder, all nodes must be contained between the *Welcome* and the *Anything else* node).

2. Call the node Assign City or something similar. **Set the condition to *@ location*. Delete the

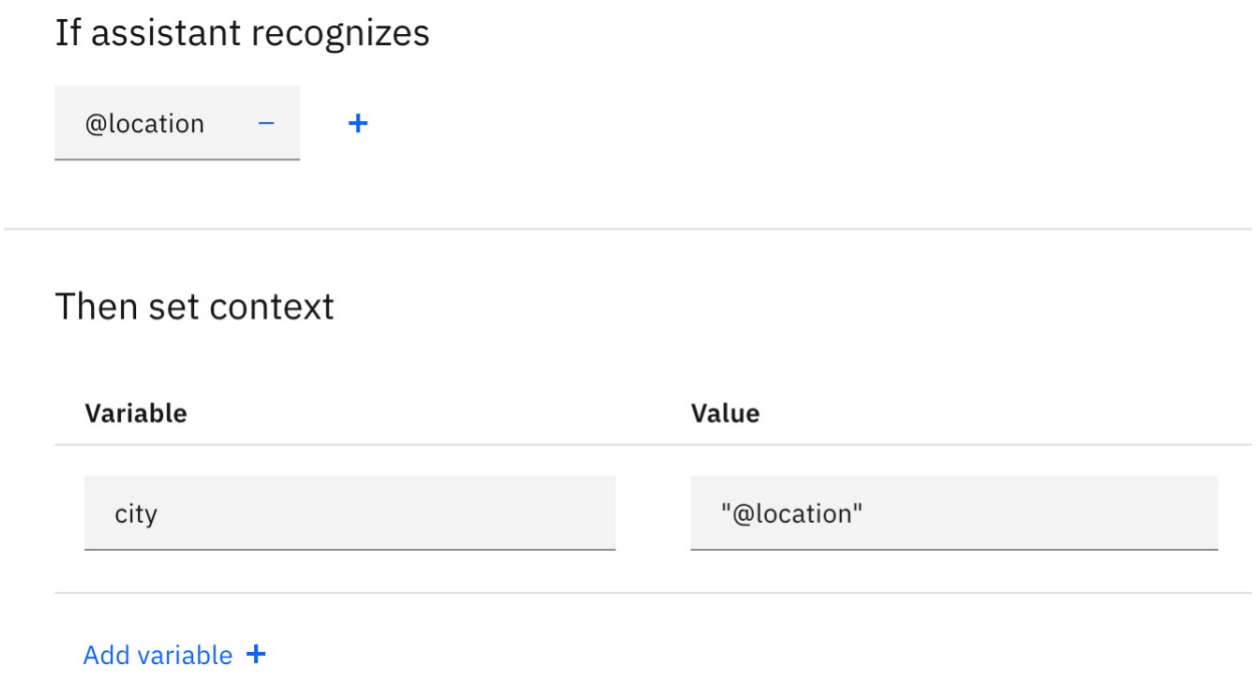
response by clicking on the trash can icon in the response area**, as we don't want this node to issue the response, only to set the variable in the context.

3. Next click on the more options menu to the right of *Assistant responds* and **select *Open

context editor***.

4. You'll be offered the ability to set one or more context variables whenever this node is

executed. Enter city for the variable name, and @ location for the value, as shown in the figure below.



In the *Then assistant should* section we don't want to wait for the user input (they already gave us input to process) we just want to jump to the rest of the nodes as if nothing happened. To do so select *Jump to* from the drop-down list. You'll be asked to specify which node to jump to. Select the first node just below the current one(i.e., Hours of Operation).

You'll then be asked to specify what to do after the jump. Wait for the user input? No. Respond directly? No. Select is *If assistant recognizes (condition)* so that this node can be evaluated as it normally would.

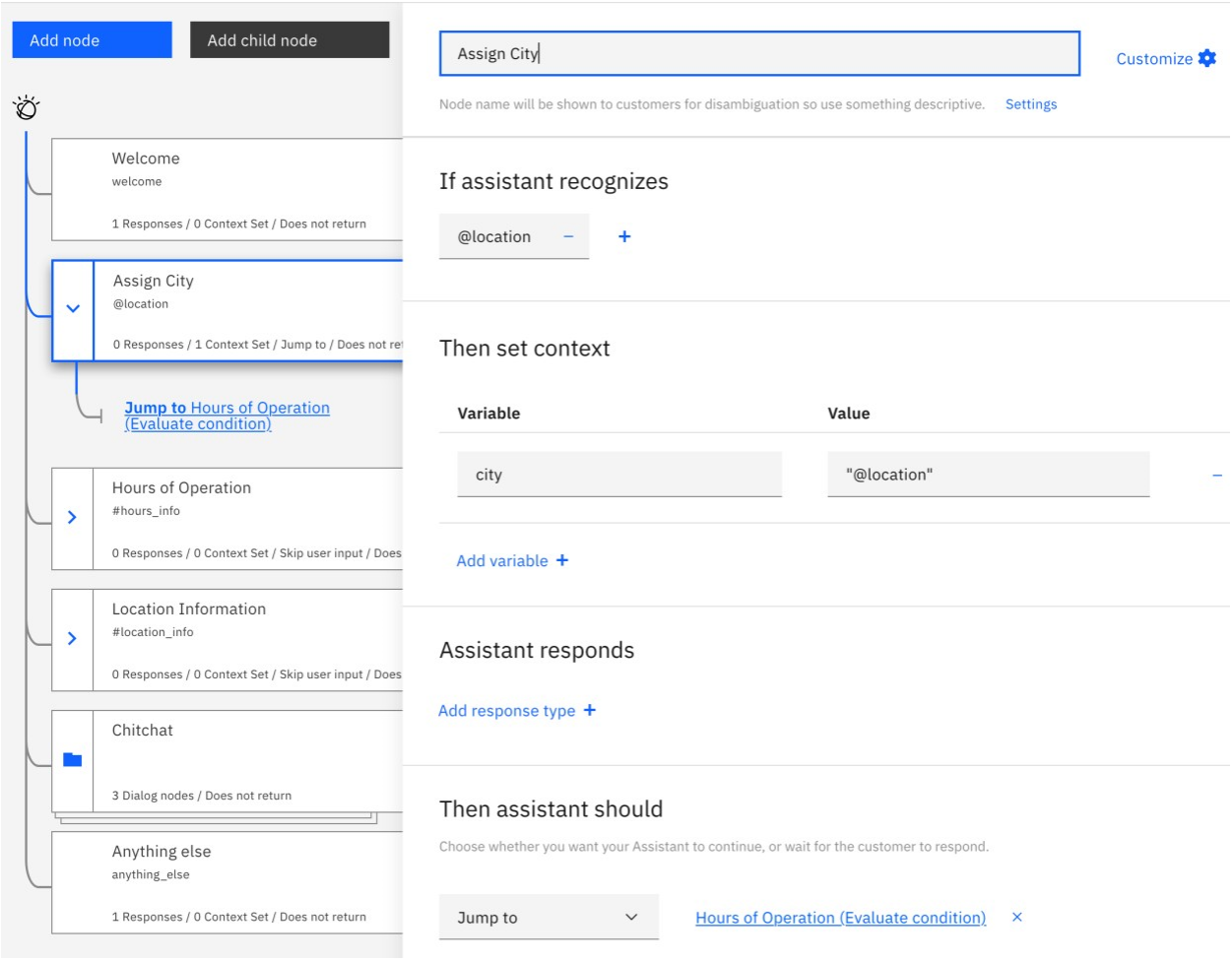
The *Assign City* now is now ready and can be closed.

To recap, our node detects if there is a *@ location* specified in the input. If there is, we execute the node which does nothing but set the context variable *$city* to the entity value (e.g., Vancouver).

Then we jump to evaluating the condition of the first node beneath us so that the flow is the same as if the *Assign City* node wasn't there.

If that node's condition is successful it will be executed. If not, the nodes beneath will be evaluated in their order of appearance. If none of the nodes satisfy the current input, we hit the fallback *Anything else* node as usual.

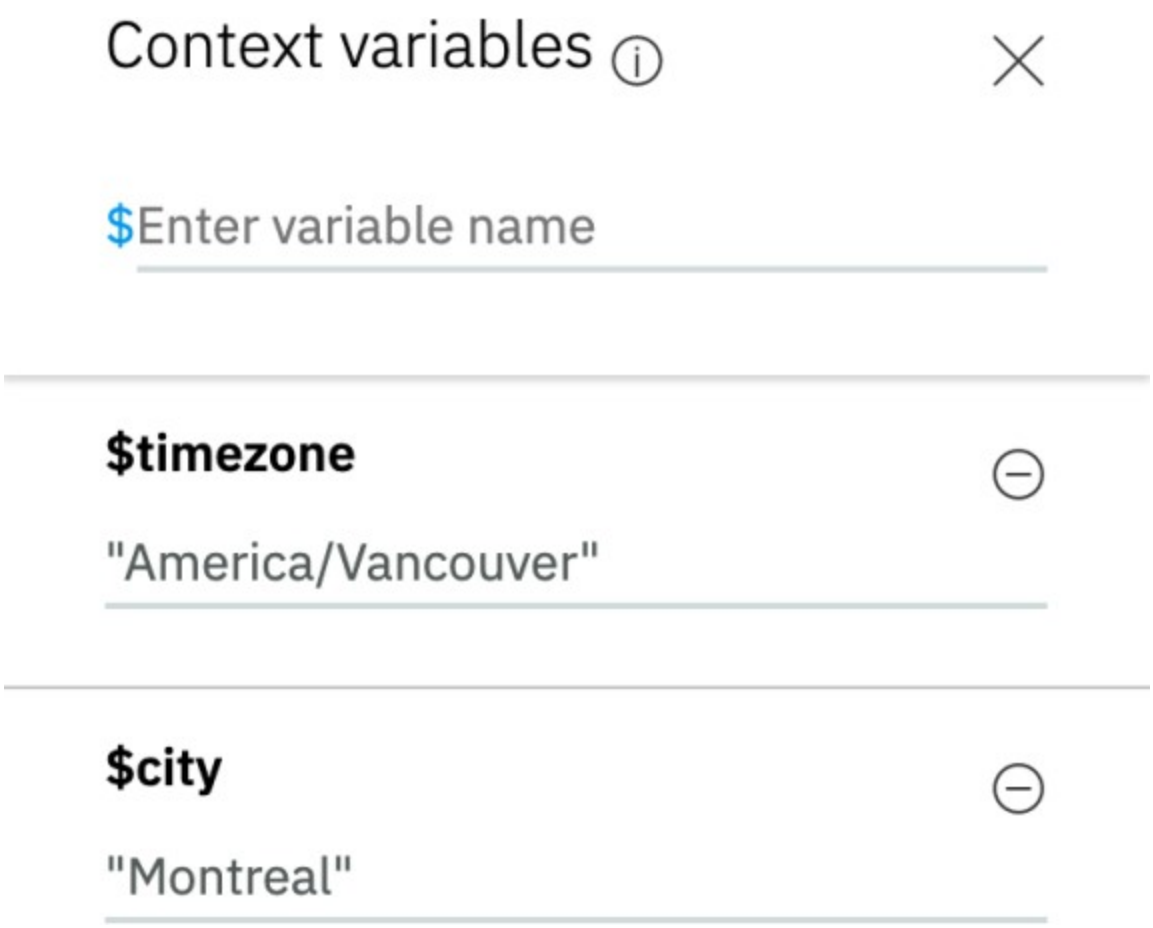
Your *Assign City* node should look as shown in the image below.



5. Head over to the *Try it out panel*, click *Clear*, and ask **What are your hours of operation?**

Click on *Manage Context* at the top of the panel to see the content of the *Context* (i.e., its variables). The *$timezone* variable will already be set for you automatically, but because we didn't specify a location, the *Assign City* node was not executed, and therefore no *$city* context variable was set.

6. Close the context and now **try entering What are your hours of operation in Montreal?** in input. Next, click on *Manage Context* again. You'll notice that this time the *$city* context variable has been set to the entity value (i.e., the string *"Montreal"*).



We'll have access to this variable for the entire duration of the conversation with the user (or until we set its value to something else). It's worth noting that pressing *Clear* in the *Try it out* panel starts a new conversation, and so context variables are cleared as well. Go ahead and close the context manager again.

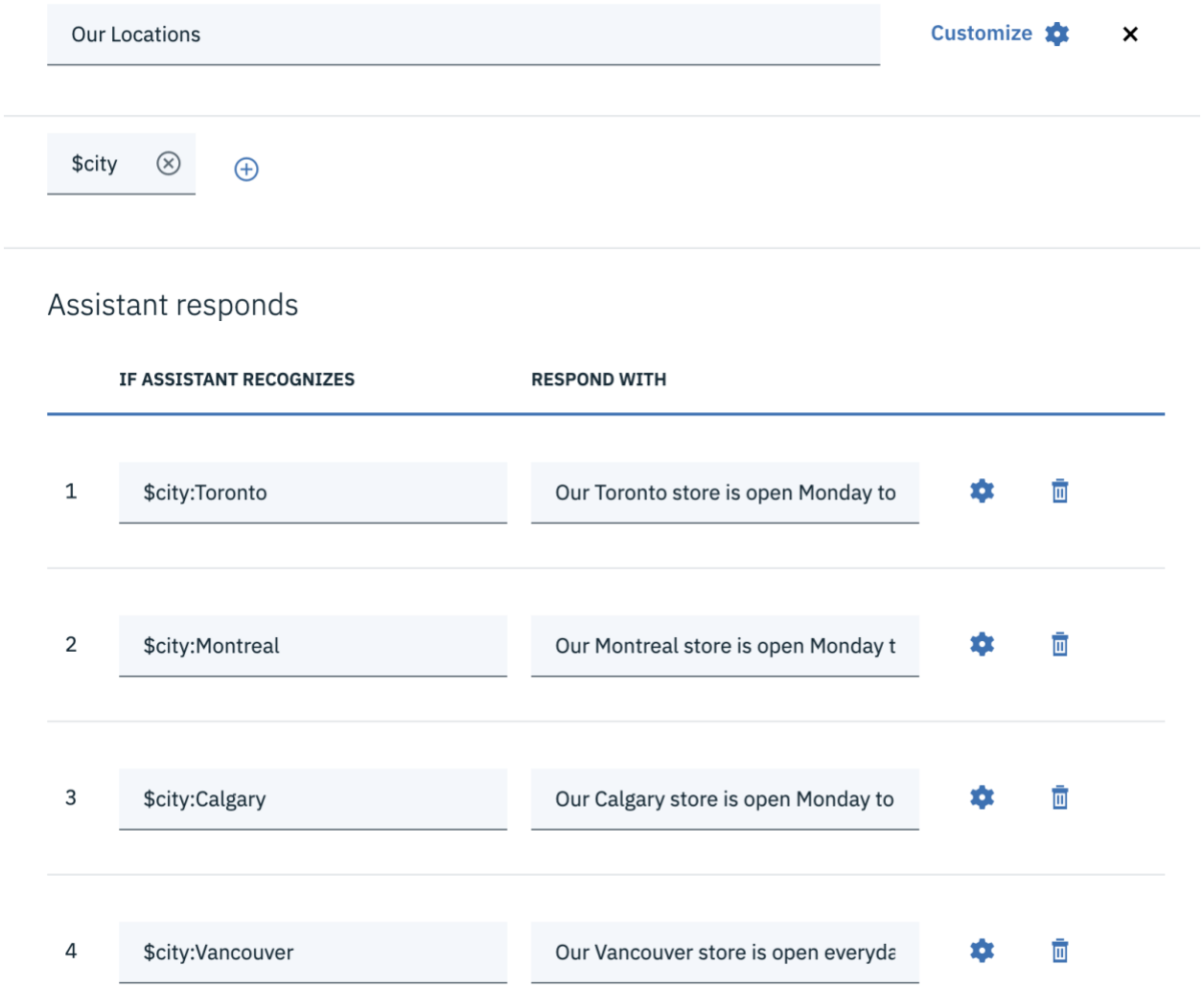
7. We want to make sure that *$city* variable is set whether it was specified along with a request

for hours information (as we already did) or for location addresses. So as a sanity check, try where is your Calgary store?. You should see that the city in the context now changes to the string *"Calgary"*.

8. Alright, we now store the city in our trusty *$city* context variable. To make use of it, we'll

need to change our *Our Locations* child nodes under the *Hours of Operation and Location Information parent nodes*. We need to do so both in the condition and in the responses.

There is an easy way to do this. Simply replace *@ location* with *$city* for every occurrence in the two *Our Locations* child nodes as I did in the image below.



Make sure you repeat this process for both *Our Locations* child nodes.

```
Please note that $city:Vancouver (no spaces) is just a shorthand for $city
== "Vancouver".
If one of our cities contained a space, we'd need to use the == comparison
(e.g., $city ==
"Quebec City").
```

9. Next, test the original interaction again. As a reminder, you can save time by recalling

previous input through the *Up* key on your keyboard, instead of retyping the same questions in.

Enter, what's the address of your store in Toronto? followed by ok, what are its hours? You should now see a better response as shown in the image below!

what's the address of your store in Toronto?

#location_info

@location:Toronto

Our Toronto store is located at 123 Warden Avenue.

ok, what are its hours?

#hours_info

Our Toronto store is open Monday to Saturday from 9 am until 6 pm, except statutory holidays.

The chatbot definitely comes across as smarter and it's more useful to the end-user.

10. But wait... now that we have the *$city* variable, can we use it to help our business even

further? It would be a nice touch to tell the user we hope they'll visit our store when they wave us goodbye.

Simply change the *Goodbyes* node responses to include the *$city* variable. If it's set to a specific city, it will be shown. If it's not set, it will not be displayed. So, go ahead and change the first response for that node to:

Nice talking to you today. We hope you visit our *$city* store.

If the *$city* is set to, say, *Calgary*, the response to the user will be *Nice talking to you today. We hope you visit our Calgary store*. If no city is set, simply *Nice talking to you today*. *We hope you visit our store*. A small, but still nice touch that invites our customers to shop with us.

Now replace the other two goodbyes with variations such as:

We hope you visit our $city store. Have a nice day.

Nice talking to you today. We hope to see you at our $city store.

Go ahead and test that it works in the *Try it out* panel. Next, click on the *Clear* link at the top to clear your variables and try typing bye now that no context variable is set. You should see that the response still makes sense.

```
As a general rule, always clear the context whenever you are running a new test.
```

Context variables are quite useful, as I hope this small example allowed to illustrate.

Exercise 2: Collect the user name with

Sometimes you'll see chatbots asking for the user name to make the interaction more personable. We know that we'd want to store it in a context variable once we acquire it so that we can refer to it throughout the conversation to sound more friendly. However, how would we go about collecting the name?

In the past, we could use a *@ sys-person* entity that would detect names for us. However, this wasn't a very robust solution and it failed to detect the beautiful variety of names in existence. It is now deprecated and cannot use it. So we'll have to roll our own solution.

1. Select the *Welcome* node. We need to change the prompt so that it asks for a name. **Enter,

Hello, my name is Florence and I'm a chatbot. What name can I call you by?**

2. We need a child node to actually collect the name (the answer to our question, in other

words). So, go ahead and create a child node under *Welcome*. Call it Collect Name.

We want to always execute this child node after the prompt, so set its condition to true.

3. Watson stores the current user input in input.text. So open the context editor (from the

more options icon to the right of *Assistant responds*) for this new node and set the *name* context variable to .

The reason why we need the special syntax is that we don't want to literally say *Nice to meet you input.text.* but rather we are asking Watson to give us the actual value of the variable.

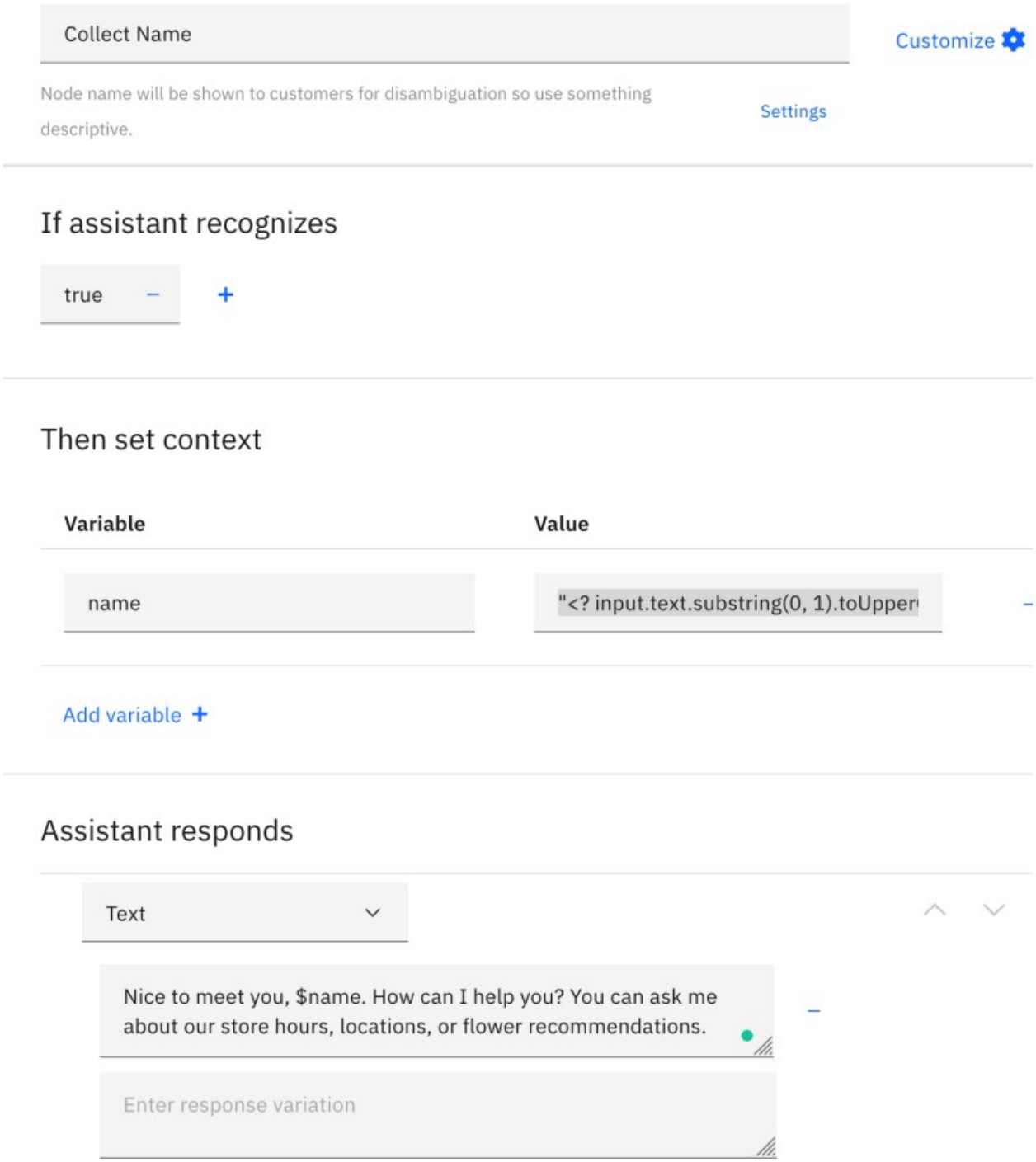
Doing so will collect the user input and assign it to the name.

```
If you want to always capitalize the name, so that antonio is stored as
Antonio, you can use a
bit of code and replace <? input.text ?> with:

             <? input.text.substring(0, 1).toUpperCase() +
input.text.substring(1) ?>

This will capitalize the first letter of the user reply for you. If you
are not a programmer, don't
worry too much about the details. Simply know that it capitalizes the
input text and you can
copy and paste it whenever you have such a need in your chatbots.
```

For the response, you can use the response below:

Nice to meet you, $name. How can I help you? You can ask me about our store hours, locations, or flower recommendations.

The image below shows what the node should look like.



Finally, test out a complete conversation with the chatbot entering these, one at the time (after clicking *Clear* in the *Try it out* panel to start a brand new conversation:

(enter your name after the prompt)

What are your hours of operation of your Toronto store?

Where is it?

Thank you

Goodbye

Pretty neat, right? If it worked, congratulations on completing Lab 10.

Help! It didn't work.

```
You can skip this section if the conversation above worked well for your chatbot.
```

If the conversation above didn't work well for your chatbot, it's likely because some mistake (or happy little accidents as Bob Ross would have called them) was made in the process of following the instructions.

If that's the case, no worries, you can import this JSON filewith the current chatbot we built so far. As usual, you might have to save the file if it opens up in your browser instead of automatically downloading it (Ctrl+S on Windows and Command+S on Mac). Feel free to call it Flower-Shop-Skill.json or something like that.

You can then click on the *Assistants* section of your Watson Assistant instance.

Objective for Exercise:

- How to handle follow-up questions and using slots.

Exercise 1: Follow-up questions and slots

What we just did, besides learning about <? input.text ?\is handle a follow-up question in a child node. This is a common pattern in which a parent node asks for information or clarification from the user and then one of its child nodes handles the response to the user.

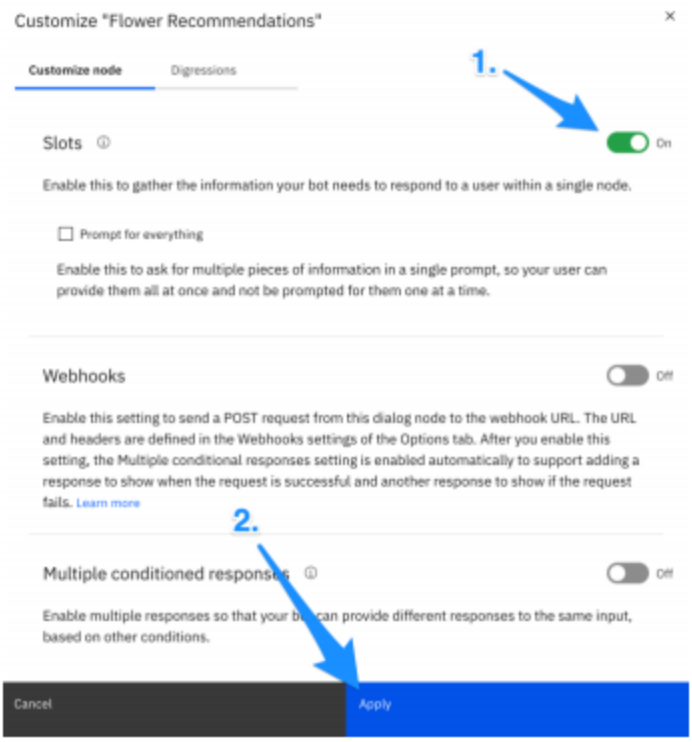
If multiple follow up questions that are dependent on each other have to be asked by the chatbot, you'll end up with a cascade of child nodes, each asking the next question in the chain and having their child process it. This works but it's not ideal in terms of reasoning about or structuring your chatbot dialog flow.

Another shortcoming of this approach is that if the user asks a side question or just says, *wait a second*, instead of replying to what we asked, we'll end up losing our "position" in the dialog cascade and therefore end up treating the delayed answer as a brand new input, failing (most likely) to provide an appropriate response or collect the information we wanted.

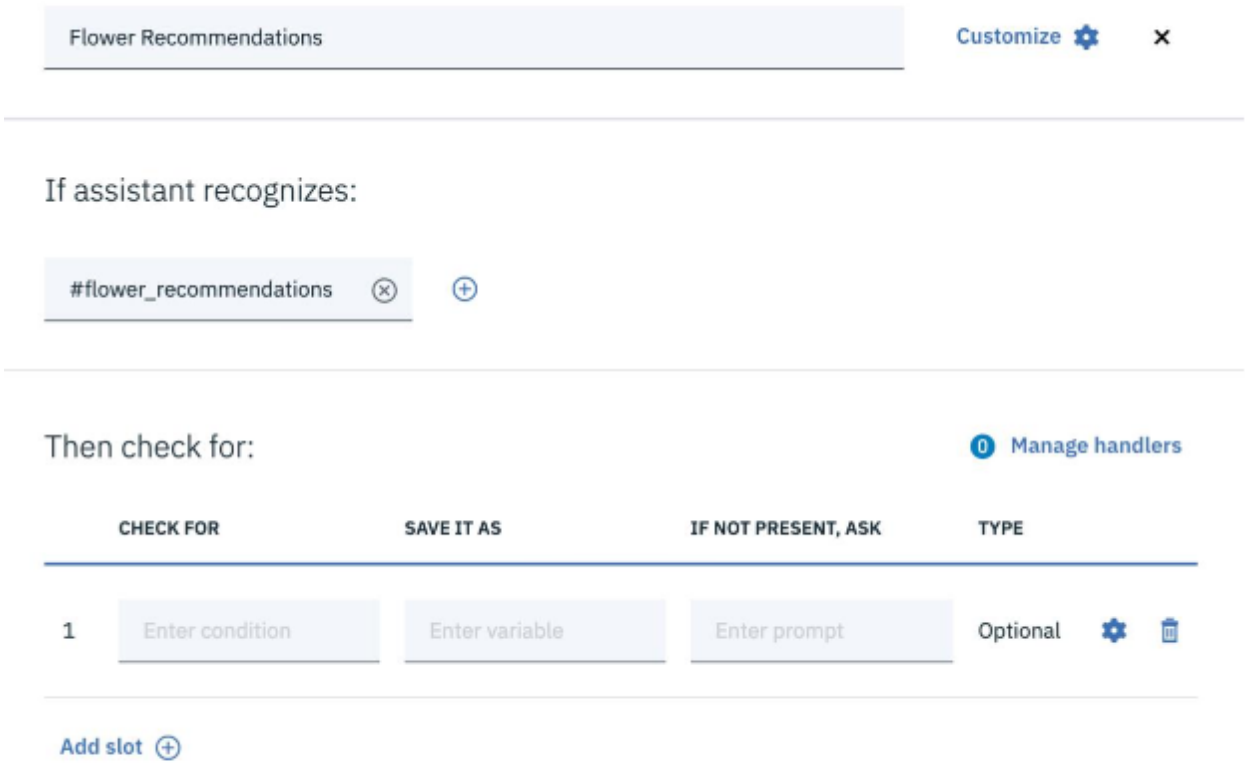
There is a much better tool to help us collect information from the user and store it in context variables. Namely, I'm talking about *Slots*.

Let's see a practical example of how they work.

1. Define an intent called #flower_recommendations with at least 5 examples of ways people might ask for flower suggestions (e.g., Flower recommendations, flowers suggestions for my girlfriend, Which flowers for Valentine's Day?, etc.). Watson will train on it as usual.
2. Create a node called Flower Recommendations below the *Welcome* node (as a peer, not a child node). Set the condition to #flower_recommendations. This is the node that will handle our flower recommendations.
3. Click on the *Customize link* in the node and turn on the *Slots* feature. Leave *Prompt for everything* unchecked, as this option is only useful if you have multiple slots/questions for the user and you want to ask them all at once, rather than one at the time. Not a common scenario. Finally, click on the *Apply* button.



4. This will automatically add one empty slot to our node. We use slots to collect information from the user and store it in a context variable.



The three key components of a slot are *Check for* (often an entity), *Save it as* (a context variable), and *If not present, ask* (an optional question to explicitly request the information if not already provided by the user in their question).

Enter @occasion, $occasion, and What occasion are the flowers for? respectively. You'll notice that the slot type changes from *Optional* to *Required* the moment we add a question.

This node will be executed when its condition #flower_recommendations is detected. In other words, when the user is asking for flower suggestions. However, we want to know

for which occasion the flowers are meant, so as to have an appropriate response for different occasions.

The slot will automatically assign *@occasion* to the *$occasion* context variable if the user- provided an entity value in their original question (e.g., *flowers suggestions for Valentine's Day*) and not ask the question in that case.

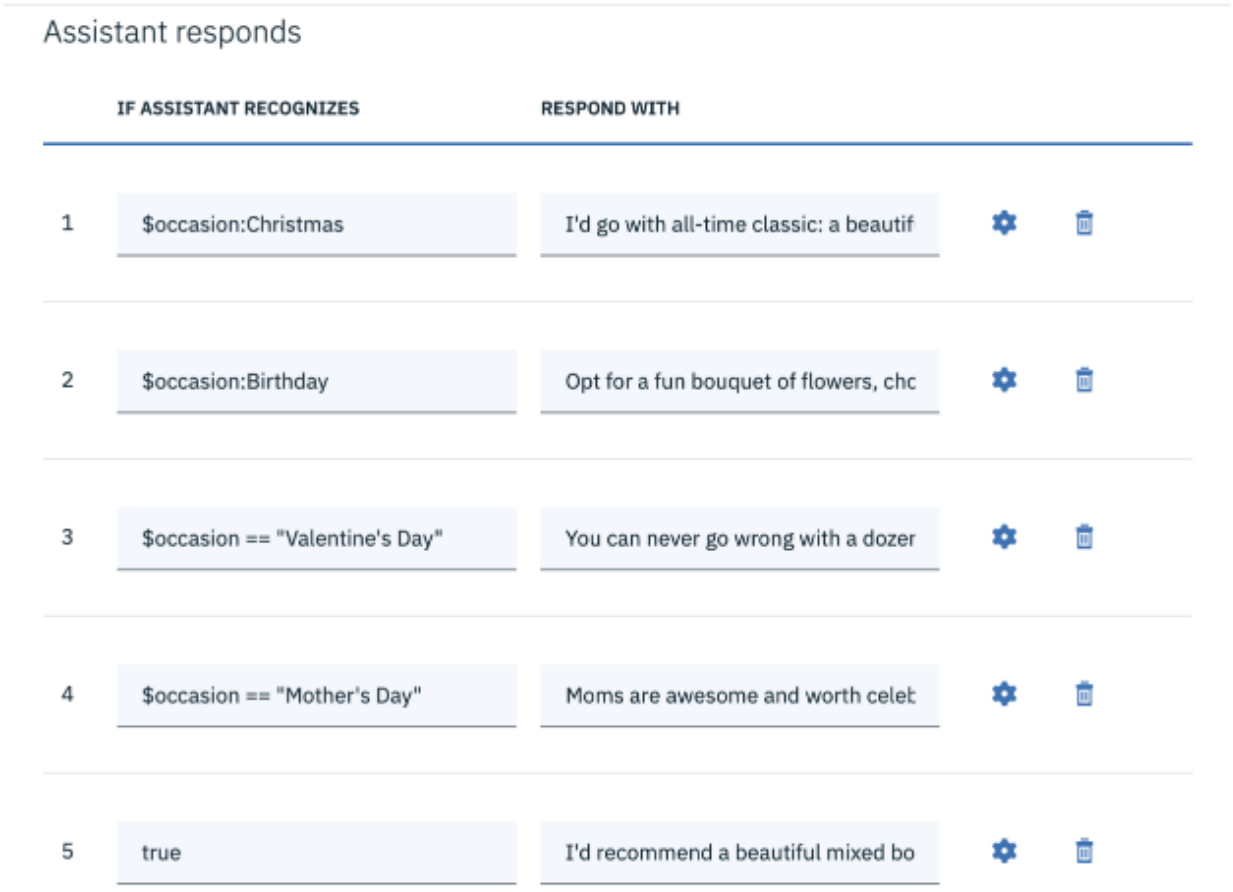
If the *@occasion* entity is not detected, because the user simply asked, *I'd like some flower recommendations* without specifying a particular occasion, then the slot will ask *What occasion are the flowers for?* until the user replies with a relevant *@occasion*. The slot is like a dog with a bone and will keep asking the question until the user enters a valid occasion. So, if the user enters an irrelevant reply, the slot will ask the question again.

By the way, a node can have multiple slots (through that *Add slot* link), if multiple pieces of information need to be collected in the same node. (Think about a restaurant reservation where you need several bits and pieces of information from the user.)

5. After the slot does its job of clarifying with the user which occasion we are talking about, it will store it in the *$occasion* context variable. So, we can use it directly in the response section of the same node, without the need to create a child node. We want to provide a different answer for each occasion, so enable *Multiple conditioned responses* for the node from the *Customize* link as well.
6. Now you can add different answers leveraging the content of the context variable $occasion, as shown in the image below. Go ahead and replicate it in your *Flower Recommendations* node, handling at least a few occasions from *@occasion*. If you don't implement them all, make sure you add a *true* fallback response for the occasions you don't handle otherwise the user will receive no response at all (a cardinal sin of chatbot design).

For the generic response, you might recommend a mixed bouquet that is versatile enough for different occasions. (Admittedly I know much more about chatbots than flowers.)

The slot sets the context variable *$occasion* for you. Make sure you use *$occasion* not *@occasion* in your multiple responses. This way, if the user-specified the occasion earlier in the conversation, we still have the memory of it and can reply appropriately.



To save you some time, here are the responses you could use:

```
For Christmas: I'd go with an all-time classic: a beautiful Red
Poinsettia.


For Birthdays: Opt for a fun bouquet of flowers, choosing a colorful

    one from <a href="https://example.org/catalog">our catalog</a>.


For Valentine's Day: You can never go wrong with a dozen red roses.


For Mother's Day: Moms are awesome and worth celebrating every day.

Consider our <a href="https://example.org/mothers-day">Mother's Day
special bouquet</a>.
```

```
For the fallback, *true* case: I'd recommend a beautiful mixed bouquet
from <a href="https://example.org/catalog">our catalog</a>.
```

Make sure you type the values exactly as they appear in the corresponding entity. For example, Valentine’s Day and not Valentine’s day or Valentines Day.

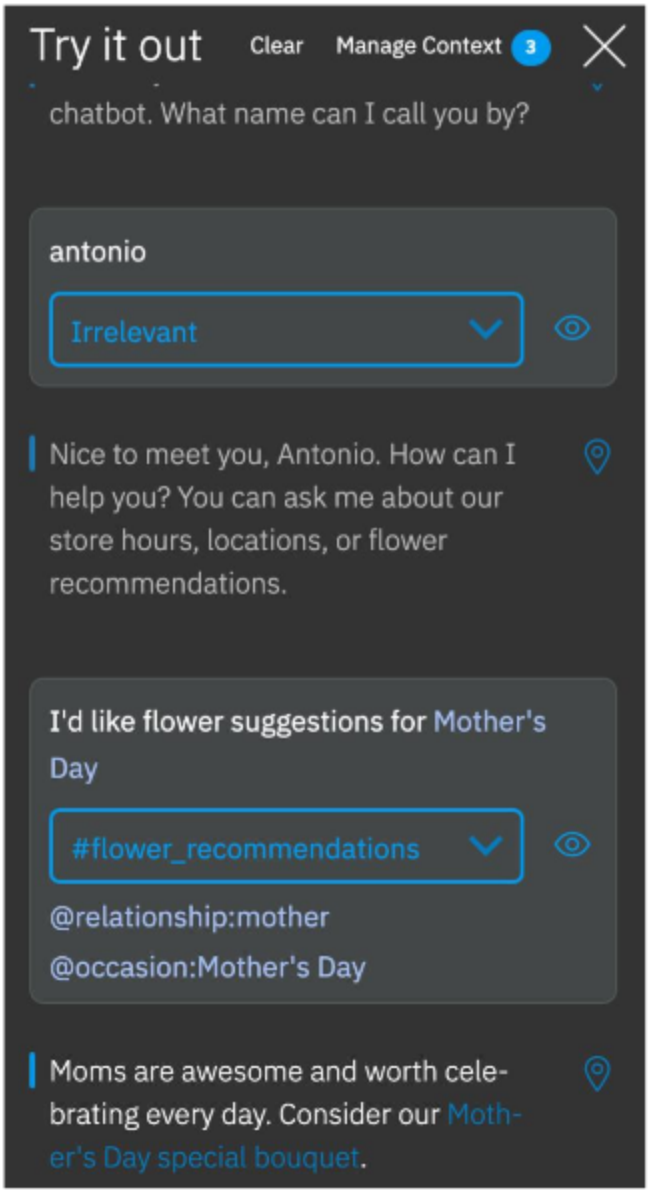
7. Once you've added the responses above, open the *Try it out* panel, press the *Clear* link if needed, and test that this is actually working. For example, for your turns, try entering:

(enter your name)

I'd like flower suggestions for Mother's Day

You should get the response you specified (provided you added one for the condition

$occasion == "Mother's Day"). Something similar to the conversation shown in the image below.



As a reminder, we can normally use the : shorthand when working with entity values that have no spaces. So $occasion:Birthday is equivalent to explicitly saying $occasion == "Birthday" which means *the value stored in $occasion is Birthday.* However, if the entity value contains a space, as it's the case for @occasion:(Mother's Day), you'll want to use the explicit form with the "equal equal" symbols (e.g., $occasion == "Mother's Day").

Using a slot saved us from having to implement collecting *$occasion* somewhere (e.g., in a passthrough node like we did for *Assign City*), handling everything neatly in one node. With a required slot we can count on *$occasion* existing as we formulate our responses.

Note that if you don't specify a question, the slot becomes optional, which means that the entity value will be stored in the context variable of your choice only if detected in the user input, but the user won't be asked explicitly for it (since you didn't provide a question). If you add two required slots to a node, then the node will ask the first question, store the information in your first context variable, then proceed with asking the second question and storing that answer in the second context variable you specified. In our case, we could have used the second slot to ask for the *@relationship* entity. Knowing both occasion and relationship would then allow us to come up with really fine-tuned answers. In the responses, we would be able to combine the two through logical AND and OR logical operators

(e.g., $occasion:Birthday AND $relationship:wife).

As mentioned, the classic example of multiple slots in a node is a chatbot that makes a restaurant reservation.

Let's say that the information it needs to collect is the name, phone number, date and time, and party size. The node can define a slot for each of these values with their respective questions. This greatly simplifies the dialog flow, as it reduces what would require several nodes, to a single node that does all the work. It also ensures that the answers are collected before the conversation proceeds further which is crucial in a scenario where, say, you are making a reservation.

To handle complex logic, you can use a combination of slots and child nodes. Slots to collect the info, child node to do the processing of that information according to your logic/preferences.

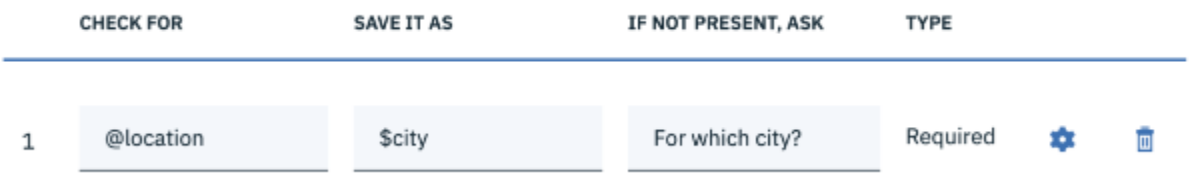
And since slots collect the information in context variables, we can refer to their values throughout the conversation with the user. So, in the example of the reservation, we might be able to confirm the reservation as we wave the user goodbye.

Exercise 2: Reimplement Hours of Operation

Now that we know how to work with slots, we can greatly simplify our *Hours of Operation* (and eventually the *Location Information*) node.

1. Get rid of the *Assign City* node by clicking on the more options menu in that node, and then selecting *Delete*.

2. Define a slot with the condition @location inside of *Hours of Operation.* Assign the value to $city. Make the slot required, that is, explicitly ask the user For which city?, if they didn't specify it in their original question.



3. Enable *Multiple conditioned responses* for the node. Then move the response information from the *Our Locations* child node into these responses within *Hours of Operation*.

If assistant recognizes	Respond with
$city:Toronto	Our Toronto store is open Monday to Saturday from 9 am until 6 pm, except statutory holidays.
$city:Montreal	Our Montreal store is open Monday to Saturday from 9 am until 5 pm, except statutory holidays.
$city:Calgary	Our Calgary store is open Monday to Friday from 9 am until 6 pm, except statutory holidays.
$city:Vancouver	Our Vancouver store is open Monday to Saturday from 10 am until 6:30 pm, except statutory holidays.

At this point, you will have the basic scenario for our locations handled by the combination of the slot and the multiple conditioned responses.

4. Delete the *Our Locations* and *No Location* child nodes below *Hours of Operation*.
5. Since *Hours of Operation* now issues a response, we need to change the *And finally* action to *Wait for user* input so that the conversation with the user can continue. In other words, we are no longer going to use the child nodes to handle the interaction.

If you test it with what are your hours of operation the chatbot will ask your *For which city?* and if you reply with one of our cities such as Vancouver, you'll get the right response.

Great.

A small problem

Having made the slot required, we have lost the ability to provide an answer for the generic,

what are your hours of operation? question. The user will get the reply, *For which city?*

This might be good enough and for some virtual assistants it might even be required. But in our case, if the user doesn't know which locations we have, we'll keep asking them the question as they helplessly reply, *I don't know* or type in cities for which we don't have a store.

That's not very user-friendly. One option we have is to remove the question from the slot and add a fallback response in the node with the condition set to true.

This way if the user provides a valid location in their question expressing an *#hours_info* intent, the slot will detect it, assign it to $city, and then respond with the relevant information for that city. If they don't provide a location, the optional slot is skipped, none of the $city responses apply, and we end up issuing the response for the true case.

Doing this is equivalent to implementing what we had through the child nodes.

1. Go ahead and delete the question from the slot.
2. Add a true response with the default text we've been using:

Our hours of operations are listed on our Hours page

3. Test the chatbot to ensure it's working correctly. After clearing the conversation try the following:

(Enter your name)

What are your hours of operation?

What are your hours of operation in Calgary?

It should still work as expected, giving the right answers for both.

The image below shows what the *Hours of Operation* node should look like.

Hours of Operation

Customize

Node name will be shown to customers for disambiguation so use something descriptive.

Settings

#hours_info

Then check for

Manage handlers

	Check for	Save it as	If not present, ask	Type
1	@location	$city	Enter prompt	Optional

Add slot +

Assistant responds

	If assistant recognizes	Respond with
1	$city:Toronto	Our Toronto store is open Monc
2	$city:Montreal	Our Montreal store is open Mor
3	$city:Calgary	Our Calgary store is open Mond
4	$city:Vancouver	Our Vancouver store is open Mc
5	true	Our hours of operations are list

Add response +

Then assistant should

Choose whether you want your Assistant to continue, or wait for the customer to respond.

Wait for reply

Exercise 3: Reimplement Location Information

Repeat the whole process in Exercise 2 for the *Location Information* node adjusting the responses accordingly.

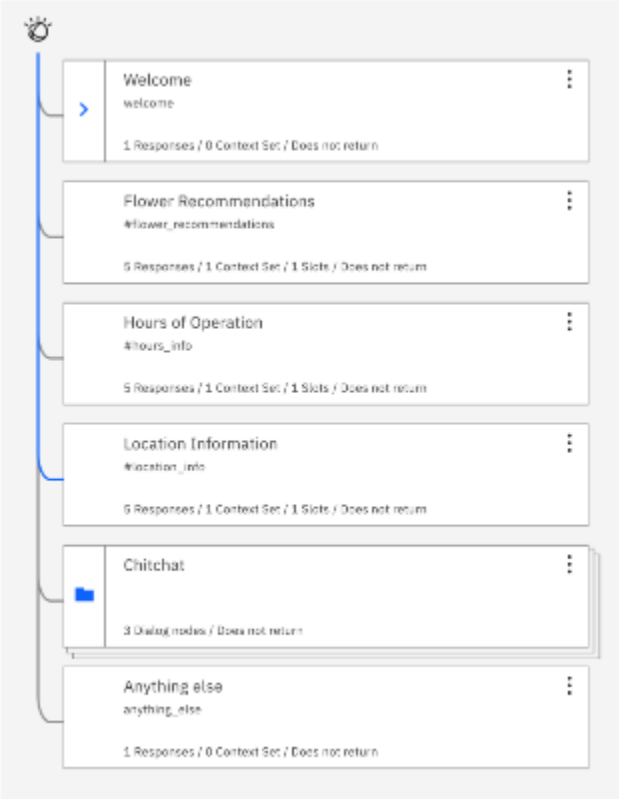
If you'd like to speed up the process, you could duplicate Hours of Operation, rename it to Location Information, then change the condition and responses to be about locations instead of hours, and finally delete the old Location Information node and its children.

Finally, test the *Location Information* node with the following text (again, click *Clear* in between each test):

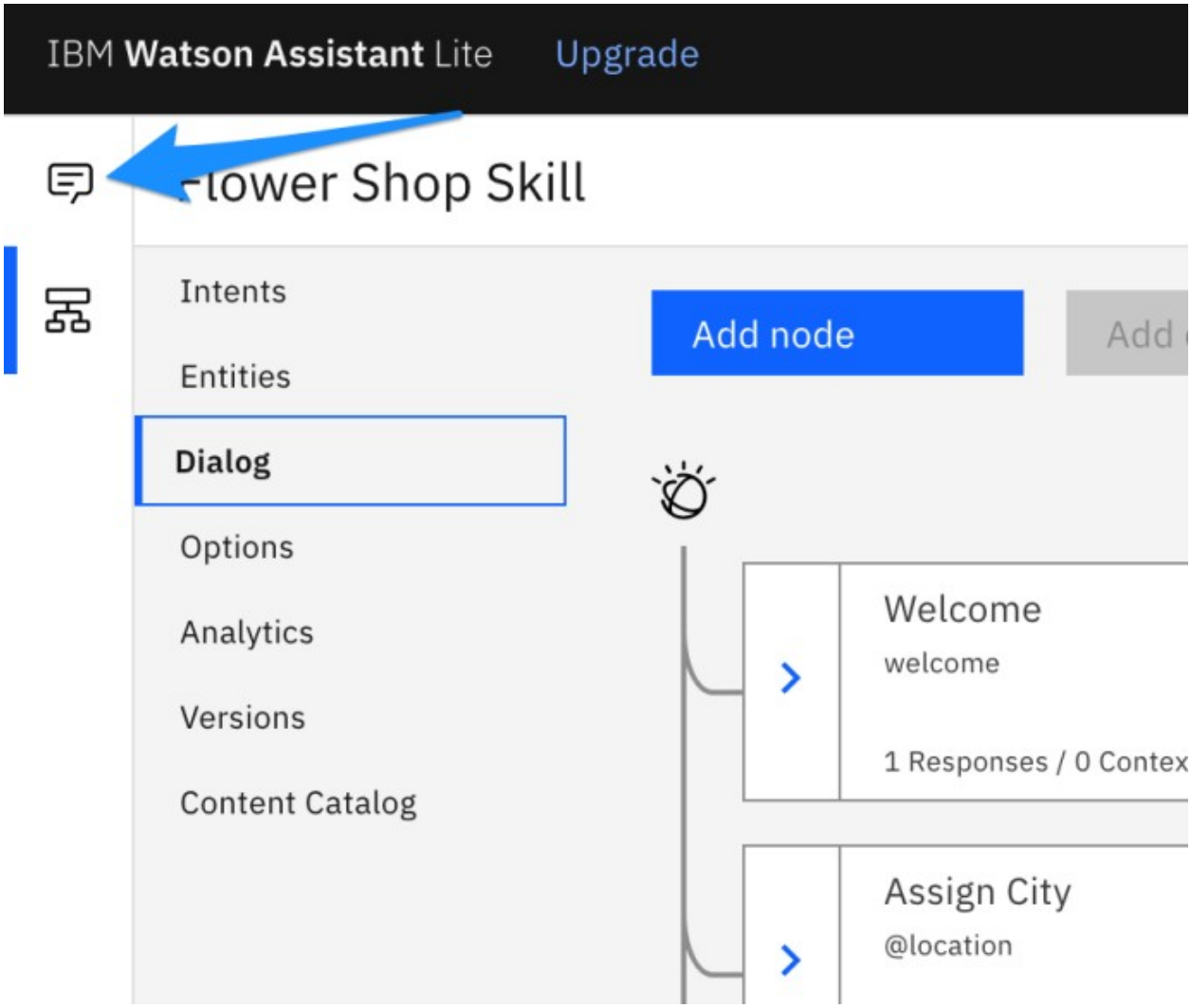
(Enter your name) list of locations

what's the address of your Vancouver store? And when is it open?

Did it work? It should. The image below shows what the dialog will look like after all the changes have been correctly implemented.

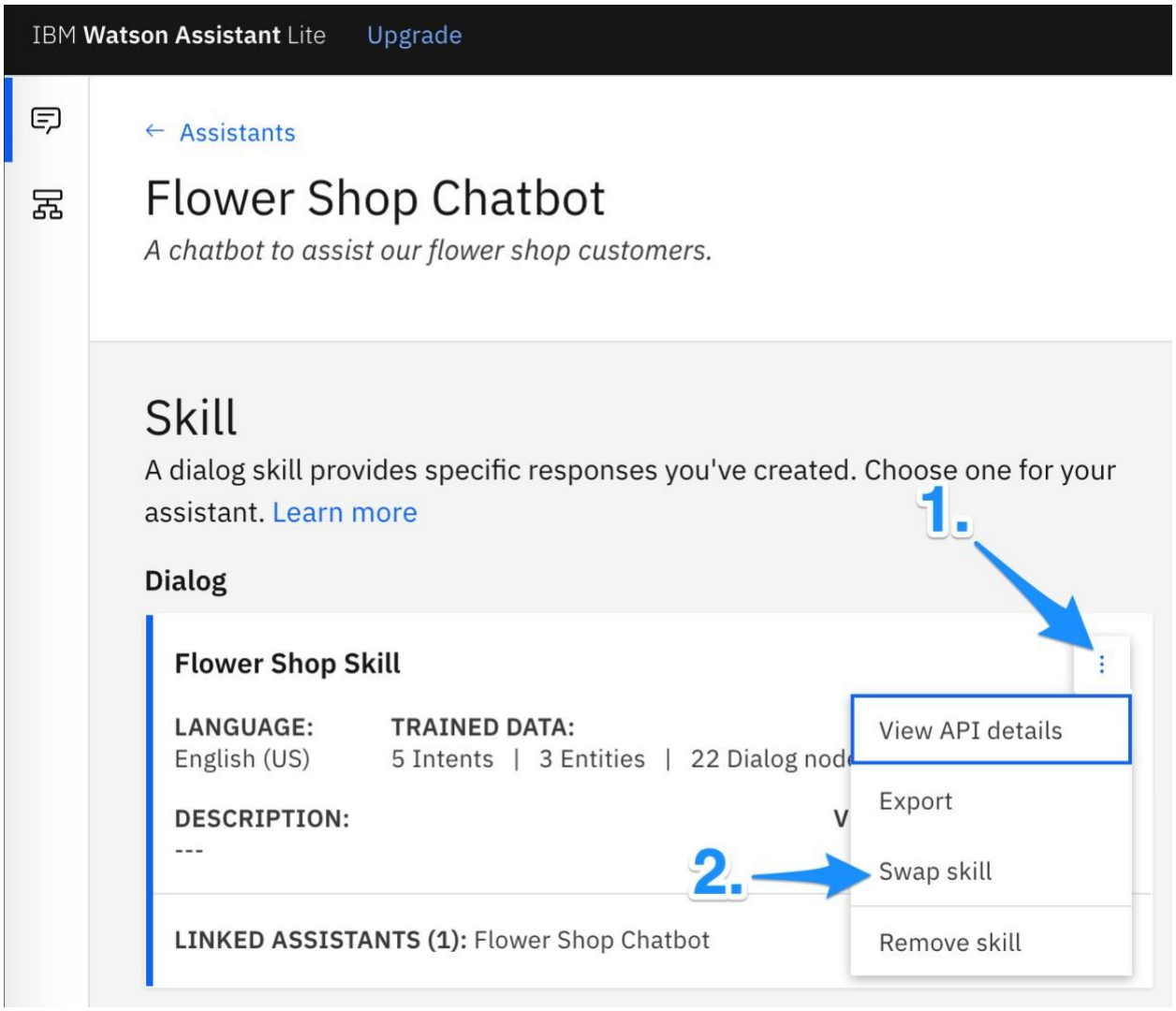


If you are lost or encountered problems when testing the chatbot, you can download the JSON file for the dialog skill we developed so far.



There you'll find your assistants. Click on *Flower Shop Chatbot*.

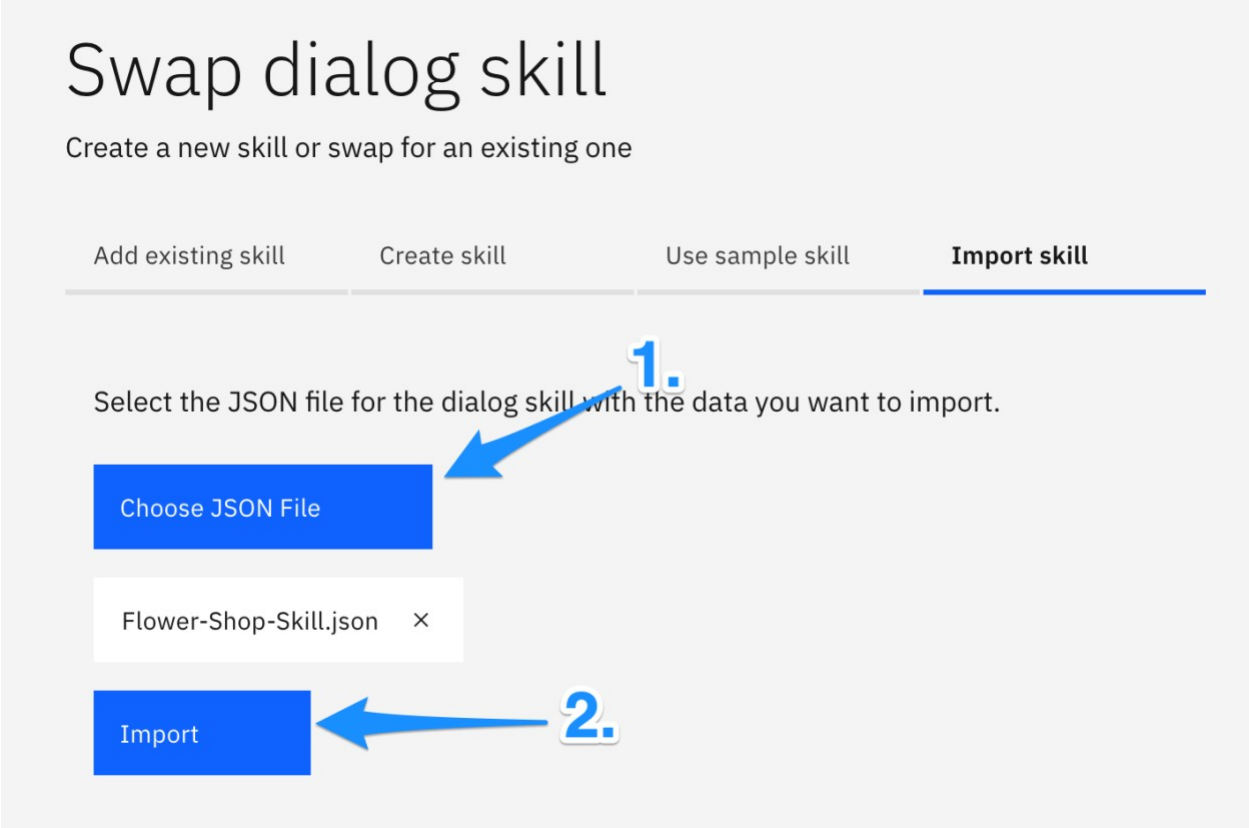
Now that you're inside of your chatbot, you should see its dialog skill. Click on the more options menu and then select *Swap Skill*.



This enables you to replace the current skill with a different one.

As you click that, a new page will appear allowing you to use an existing skill you already created, create a new skill, use a sample one, or importing a skill.

Select the *Import Skil*l tab, upload the JSON file you just downloaded (by clicking on *Choose JSON File*) and then click *Import*.



Once the import is done, you'll have the skill we developed so far linked to your assistant. A successful notification will appear.

Try the conversation again and this time it should work for you.

Objective for Exercise:

- How to handle follow-up questions and using slots.

Exercise 1: Follow-up questions and slots

What we just did, besides learning about <? input.text ?\is handle a follow-up question in a child node. This is a common pattern in which a parent node asks for information or clarification from the user and then one of its child nodes handles the response to the user.

If multiple follow up questions that are dependent on each other have to be asked by the chatbot, you'll end up with a cascade of child nodes, each asking the next question in

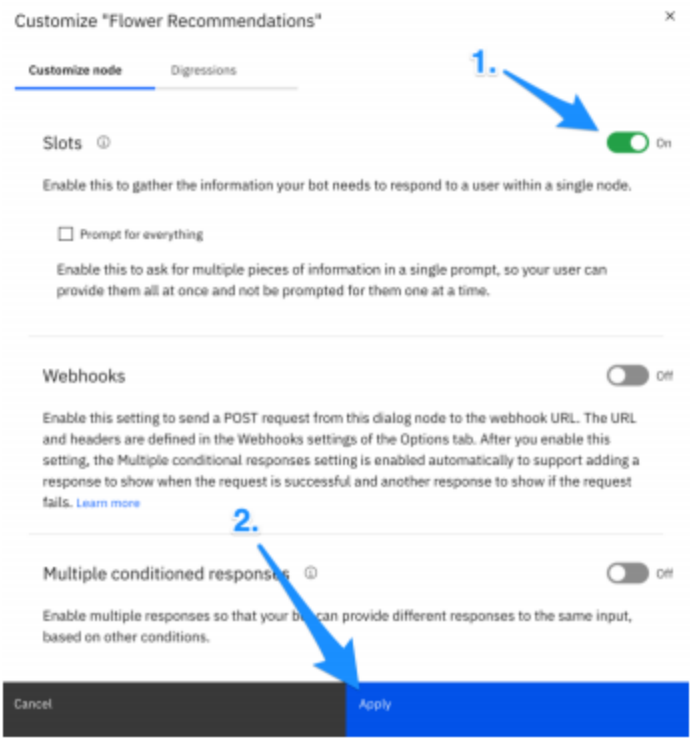
the chain and having their child process it. This works but it's not ideal in terms of reasoning about or structuring your chatbot dialog flow.

Another shortcoming of this approach is that if the user asks a side question or just says, *wait a second*, instead of replying to what we asked, we'll end up losing our "position" in the dialog cascade and therefore end up treating the delayed answer as a brand new input, failing (most likely) to provide an appropriate response or collect the information we wanted.

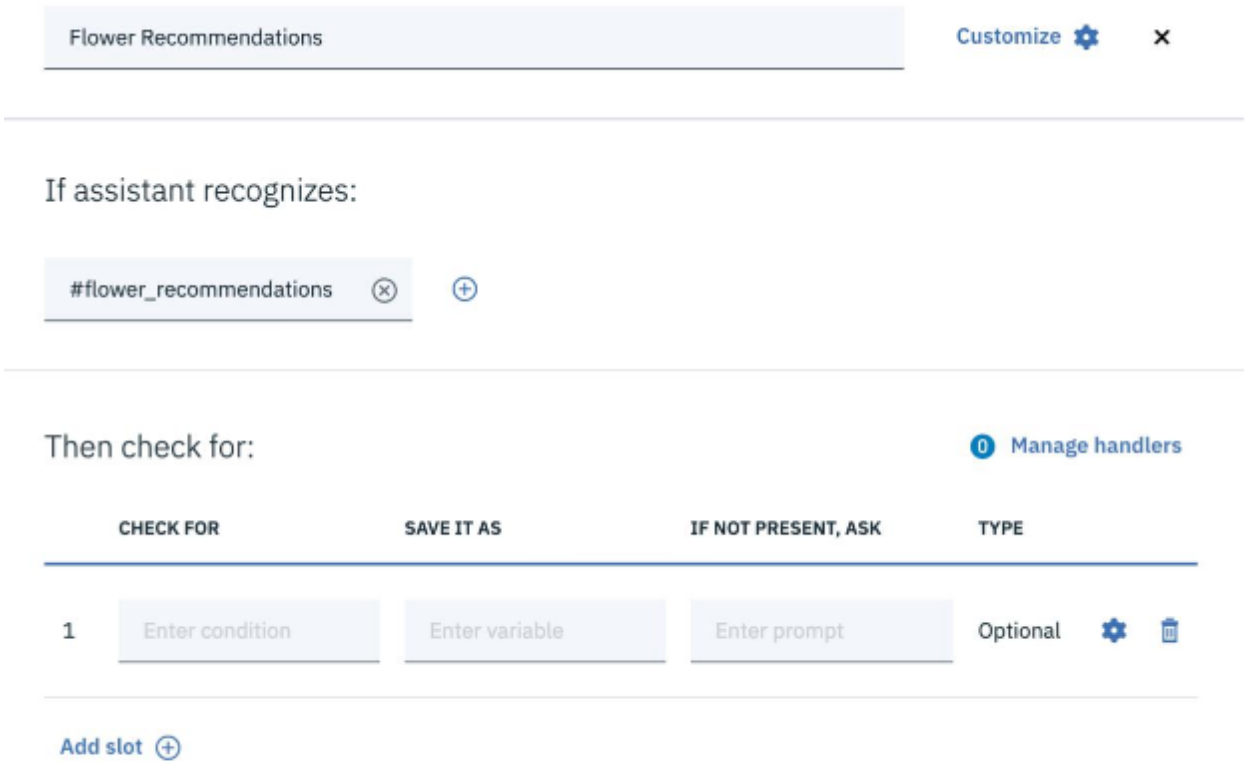
There is a much better tool to help us collect information from the user and store it in context variables. Namely, I'm talking about *Slots*.

Let's see a practical example of how they work.

1. Define an intent called #flower_recommendations with at least 5 examples of ways people might ask for flower suggestions (e.g., Flower recommendations, flowers suggestions for my girlfriend, Which flowers for Valentine's Day?, etc.). Watson will train on it as usual.
2. Create a node called Flower Recommendations below the *Welcome* node (as a peer, not a child node). Set the condition to #flower_recommendations. This is the node that will handle our flower recommendations.
3. Click on the *Customize link* in the node and turn on the *Slots* feature. Leave *Prompt for everything* unchecked, as this option is only useful if you have multiple slots/questions for the user and you want to ask them all at once, rather than one at the time. Not a common scenario. Finally, click on the *Apply* button.



4. This will automatically add one empty slot to our node. We use slots to collect information from the user and store it in a context variable.



The three key components of a slot are *Check for* (often an entity), *Save it as* (a context variable), and *If not present, ask* (an optional question to explicitly request the information if not already provided by the user in their question).

Enter @occasion, $occasion, and What occasion are the flowers for? respectively. You'll notice that the slot type changes from *Optional* to *Required* the moment we add a question.

This node will be executed when its condition #flower_recommendations is detected. In other words, when the user is asking for flower suggestions. However, we want to know

for which occasion the flowers are meant, so as to have an appropriate response for different occasions.

The slot will automatically assign *@occasion* to the *$occasion* context variable if the user- provided an entity value in their original question (e.g., *flowers suggestions for Valentine's Day*) and not ask the question in that case.

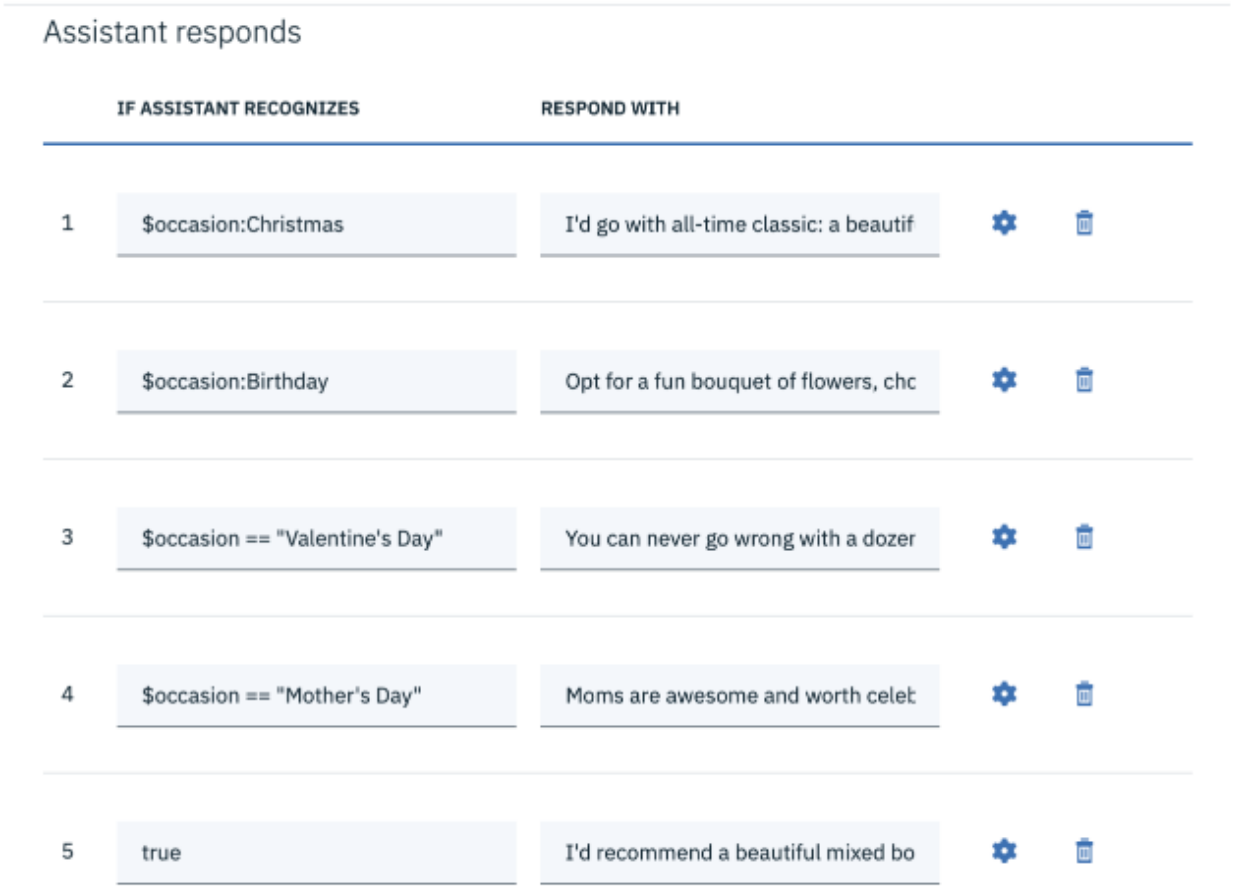
If the *@occasion* entity is not detected, because the user simply asked, *I'd like some flower recommendations* without specifying a particular occasion, then the slot will ask *What occasion are the flowers for?* until the user replies with a relevant *@occasion*. The slot is like a dog with a bone and will keep asking the question until the user enters a valid occasion. So, if the user enters an irrelevant reply, the slot will ask the question again.

By the way, a node can have multiple slots (through that *Add slot* link), if multiple pieces of information need to be collected in the same node. (Think about a restaurant reservation where you need several bits and pieces of information from the user.)

5. After the slot does its job of clarifying with the user which occasion we are talking about, it will store it in the *$occasion* context variable. So, we can use it directly in the response section of the same node, without the need to create a child node. We want to provide a different answer for each occasion, so enable *Multiple conditioned responses* for the node from the *Customize* link as well.
6. Now you can add different answers leveraging the content of the context variable $occasion, as shown in the image below. Go ahead and replicate it in your *Flower Recommendations* node, handling at least a few occasions from *@occasion*. If you don't implement them all, make sure you add a *true* fallback response for the occasions you don't handle otherwise the user will receive no response at all (a cardinal sin of chatbot design).

For the generic response, you might recommend a mixed bouquet that is versatile enough for different occasions. (Admittedly I know much more about chatbots than flowers.)

The slot sets the context variable *$occasion* for you. Make sure you use *$occasion* not *@occasion* in your multiple responses. This way, if the user-specified the occasion earlier in the conversation, we still have the memory of it and can reply appropriately.



To save you some time, here are the responses you could use:

```
For Christmas: I'd go with an all-time classic: a beautiful Red
Poinsettia.


For Birthdays: Opt for a fun bouquet of flowers, choosing a colorful

    one from <a href="https://example.org/catalog">our catalog</a>.


For Valentine's Day: You can never go wrong with a dozen red roses.


For Mother's Day: Moms are awesome and worth celebrating every day.

Consider our <a href="https://example.org/mothers-day">Mother's Day
special bouquet</a>.
```

```
For the fallback, *true* case: I'd recommend a beautiful mixed bouquet
from <a href="https://example.org/catalog">our catalog</a>.
```

Make sure you type the values exactly as they appear in the corresponding entity. For example, Valentine’s Day and not Valentine’s day or Valentines Day.

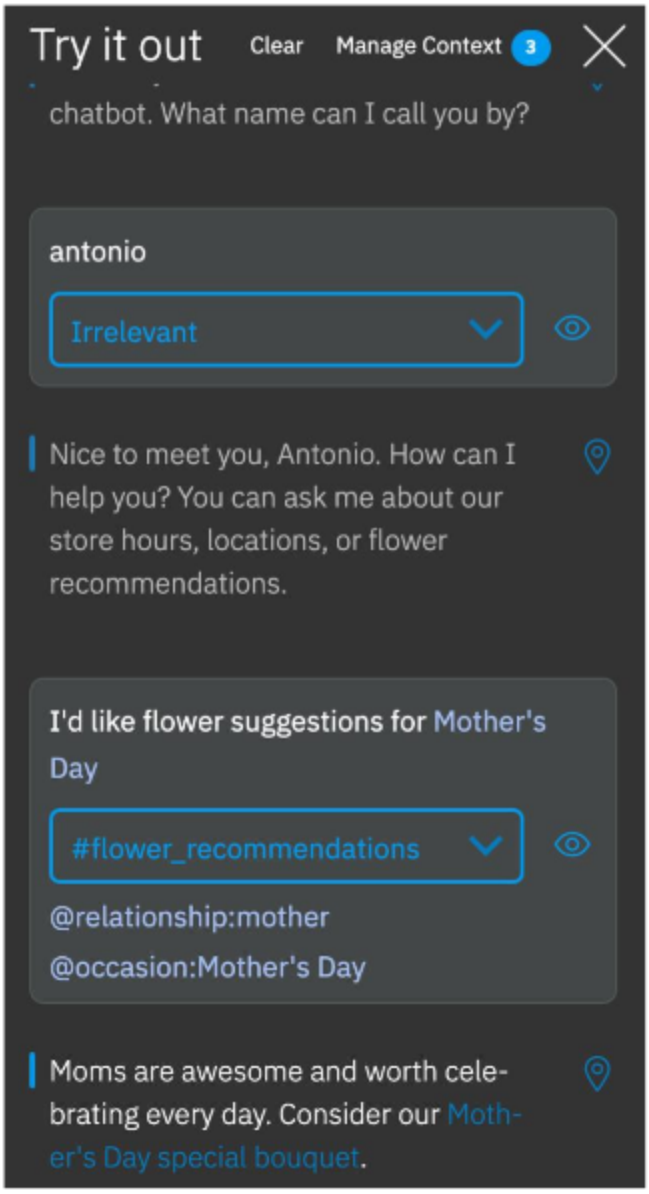
7. Once you've added the responses above, open the *Try it out* panel, press the *Clear* link if needed, and test that this is actually working. For example, for your turns, try entering:

(enter your name)

I'd like flower suggestions for Mother's Day

You should get the response you specified (provided you added one for the condition

$occasion == "Mother's Day"). Something similar to the conversation shown in the image below.



As a reminder, we can normally use the : shorthand when working with entity values that have no spaces. So $occasion:Birthday is equivalent to explicitly saying $occasion == "Birthday" which means *the value stored in $occasion is Birthday.* However, if the entity value contains a space, as it's the case for @occasion:(Mother's Day), you'll want to use the explicit form with the "equal equal" symbols (e.g., $occasion == "Mother's Day").

Using a slot saved us from having to implement collecting *$occasion* somewhere (e.g., in a passthrough node like we did for *Assign City*), handling everything neatly in one node. With a required slot we can count on *$occasion* existing as we formulate our responses.

Note that if you don't specify a question, the slot becomes optional, which means that the entity value will be stored in the context variable of your choice only if detected in the user input, but the user won't be asked explicitly for it (since you didn't provide a question). If you add two required slots to a node, then the node will ask the first question, store the information in your first context variable, then proceed with asking the second question and storing that answer in the second context variable you specified. In our case, we could have used the second slot to ask for the *@relationship* entity. Knowing both occasion and relationship would then allow us to come up with really fine-tuned answers. In the responses, we would be able to combine the two through logical AND and OR logical operators

(e.g., $occasion:Birthday AND $relationship:wife).

As mentioned, the classic example of multiple slots in a node is a chatbot that makes a restaurant reservation.

Let's say that the information it needs to collect is the name, phone number, date and time, and party size. The node can define a slot for each of these values with their respective questions. This greatly simplifies the dialog flow, as it reduces what would require several nodes, to a single node that does all the work. It also ensures that the answers are collected before the conversation proceeds further which is crucial in a scenario where, say, you are making a reservation.

To handle complex logic, you can use a combination of slots and child nodes. Slots to collect the info, child node to do the processing of that information according to your logic/preferences.

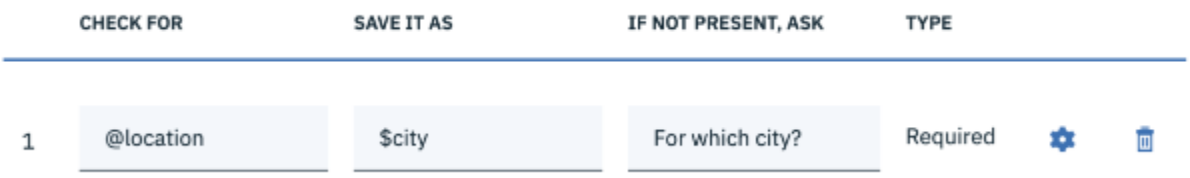
And since slots collect the information in context variables, we can refer to their values throughout the conversation with the user. So, in the example of the reservation, we might be able to confirm the reservation as we wave the user goodbye.

Exercise 2: Reimplement Hours of Operation

Now that we know how to work with slots, we can greatly simplify our *Hours of Operation* (and eventually the *Location Information*) node.

1. Get rid of the *Assign City* node by clicking on the more options menu in that node, and then selecting *Delete*.

2. Define a slot with the condition @location inside of *Hours of Operation.* Assign the value to $city. Make the slot required, that is, explicitly ask the user For which city?, if they didn't specify it in their original question.



3. Enable *Multiple conditioned responses* for the node. Then move the response information from the *Our Locations* child node into these responses within *Hours of Operation*.

If assistant recognizes	Respond with
$city:Toronto	Our Toronto store is open Monday to Saturday from 9 am until 6 pm, except statutory holidays.
$city:Montreal	Our Montreal store is open Monday to Saturday from 9 am until 5 pm, except statutory holidays.
$city:Calgary	Our Calgary store is open Monday to Friday from 9 am until 6 pm, except statutory holidays.
$city:Vancouver	Our Vancouver store is open Monday to Saturday from 10 am until 6:30 pm, except statutory holidays.

At this point, you will have the basic scenario for our locations handled by the combination of the slot and the multiple conditioned responses.

4. Delete the *Our Locations* and *No Location* child nodes below *Hours of Operation*.
5. Since *Hours of Operation* now issues a response, we need to change the *And finally* action to *Wait for user* input so that the conversation with the user can continue. In other words, we are no longer going to use the child nodes to handle the interaction.

If you test it with what are your hours of operation the chatbot will ask your *For which city?* and if you reply with one of our cities such as Vancouver, you'll get the right response.

Great.

A small problem

Having made the slot required, we have lost the ability to provide an answer for the generic,

what are your hours of operation? question. The user will get the reply, *For which city?*

This might be good enough and for some virtual assistants it might even be required. But in our case, if the user doesn't know which locations we have, we'll keep asking them the question as they helplessly reply, *I don't know* or type in cities for which we don't have a store.

That's not very user-friendly. One option we have is to remove the question from the slot and add a fallback response in the node with the condition set to true.

This way if the user provides a valid location in their question expressing an *#hours_info* intent, the slot will detect it, assign it to $city, and then respond with the relevant information for that city. If they don't provide a location, the optional slot is skipped, none of the $city responses apply, and we end up issuing the response for the true case.

Doing this is equivalent to implementing what we had through the child nodes.

1. Go ahead and delete the question from the slot.
2. Add a true response with the default text we've been using:

Our hours of operations are listed on our Hours page

3. Test the chatbot to ensure it's working correctly. After clearing the conversation try the following:

(Enter your name)

What are your hours of operation?

What are your hours of operation in Calgary?

It should still work as expected, giving the right answers for both.

The image below shows what the *Hours of Operation* node should look like.

Hours of Operation

Customize

Node name will be shown to customers for disambiguation so use something descriptive.

Settings

#hours_info

Then check for

Manage handlers

	Check for	Save it as	If not present, ask	Type
1	@location	$city	Enter prompt	Optional

Add slot +

Assistant responds

	If assistant recognizes	Respond with
1	$city:Toronto	Our Toronto store is open Monc
2	$city:Montreal	Our Montreal store is open Mor
3	$city:Calgary	Our Calgary store is open Mond
4	$city:Vancouver	Our Vancouver store is open Mc
5	true	Our hours of operations are list

Add response +

Then assistant should

Choose whether you want your Assistant to continue, or wait for the customer to respond.

Wait for reply

Exercise 3: Reimplement Location Information

Repeat the whole process in Exercise 2 for the *Location Information* node adjusting the responses accordingly.

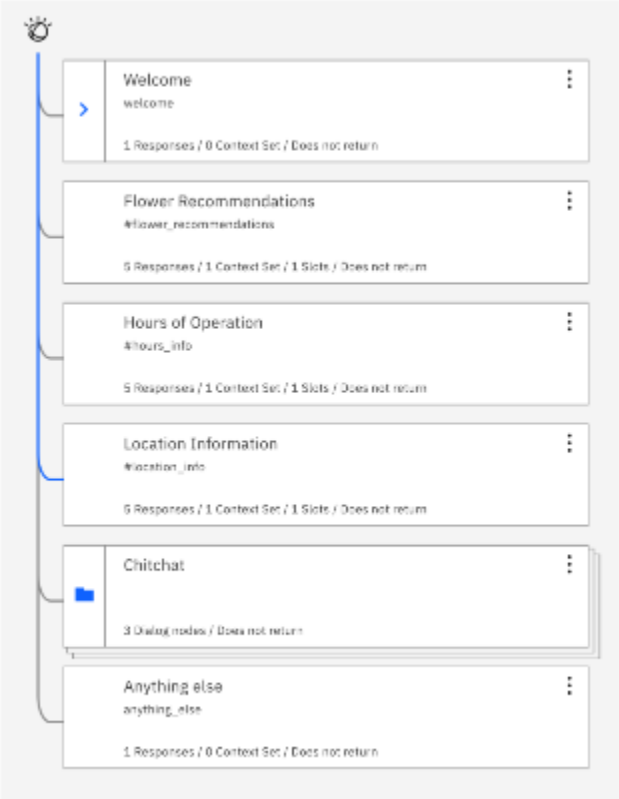
If you'd like to speed up the process, you could duplicate Hours of Operation, rename it to Location Information, then change the condition and responses to be about locations instead of hours, and finally delete the old Location Information node and its children.

Finally, test the *Location Information* node with the following text (again, click *Clear* in between each test):

(Enter your name) list of locations

what's the address of your Vancouver store? And when is it open?

Did it work? It should. The image below shows what the dialog will look like after all the changes have been correctly implemented.



If you are lost or encountered problems when testing the chatbot, you can download the JSON file for the dialog skill we developed so far.

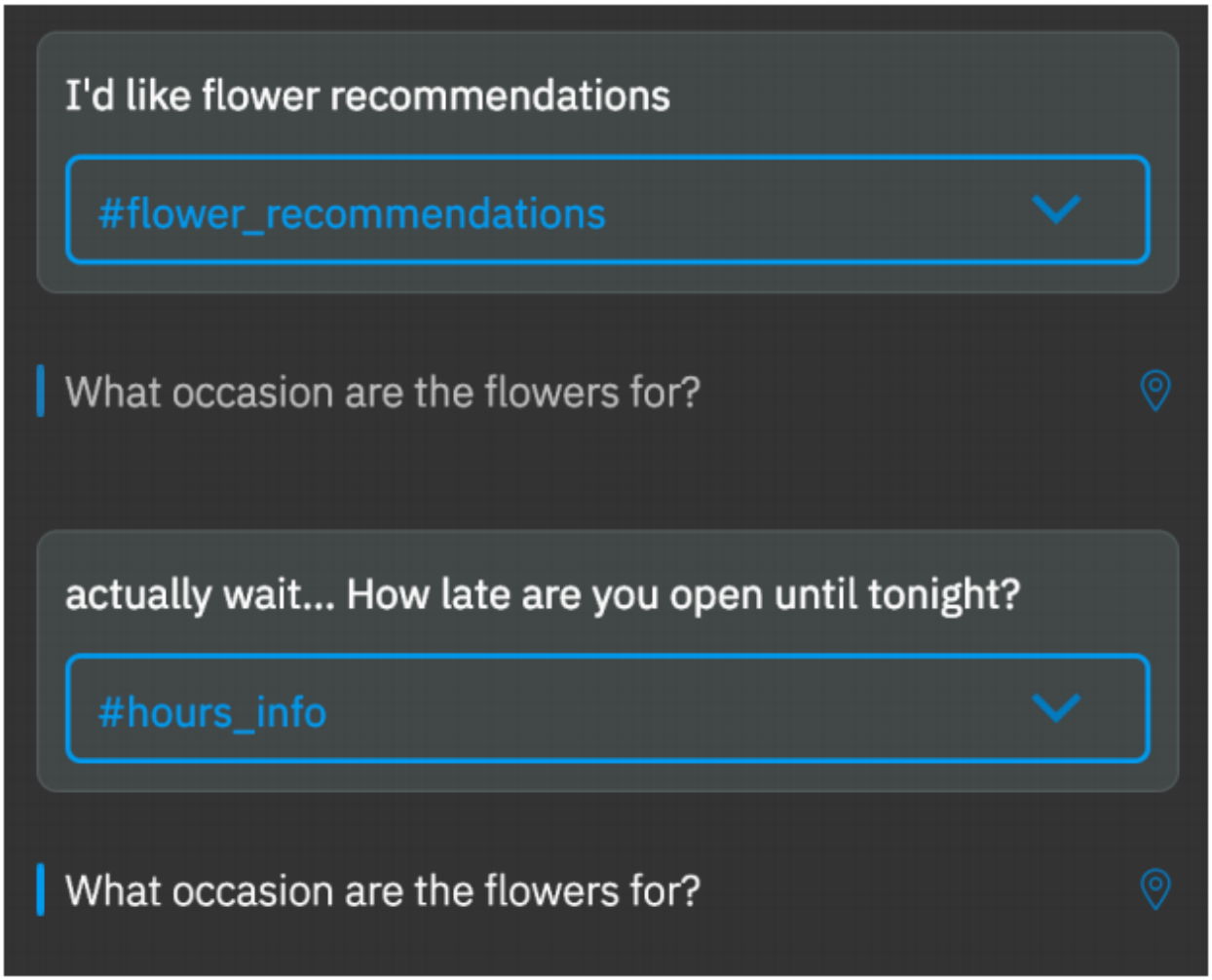
Objective for Exercise:

- Configuring Slot's Found and Not Found.
- Enabling Digressions.
- Understanding Handlers.

Exercise 1: Configuring Slot’s Found and Not Found

Slots are awesome. However, their stubborn nature (for required slots that specify a question) can come across as rude if we are not careful. They keep the user to the task, which might be fine if the user enters something irrelevant. It's definitely less okay if the user is asking a legitimate side question, however.

Consider the following interaction.



That's not great. It looks like we completely ignored the user's legitimate question. Notice also that the chatbot understood what the user wanted (i.e., #hours_info was detected) but we were still somewhat rude to them by not addressing their digression.

Configuring Found and Not Found responses

A first course of action we have is to use the Slot's *Found* and *Not Found* responses. These are issued to the user before the node's own responses (in our case the actual flower suggestions for the given occasion), so they allow us to interject a message before the node replies to the user.

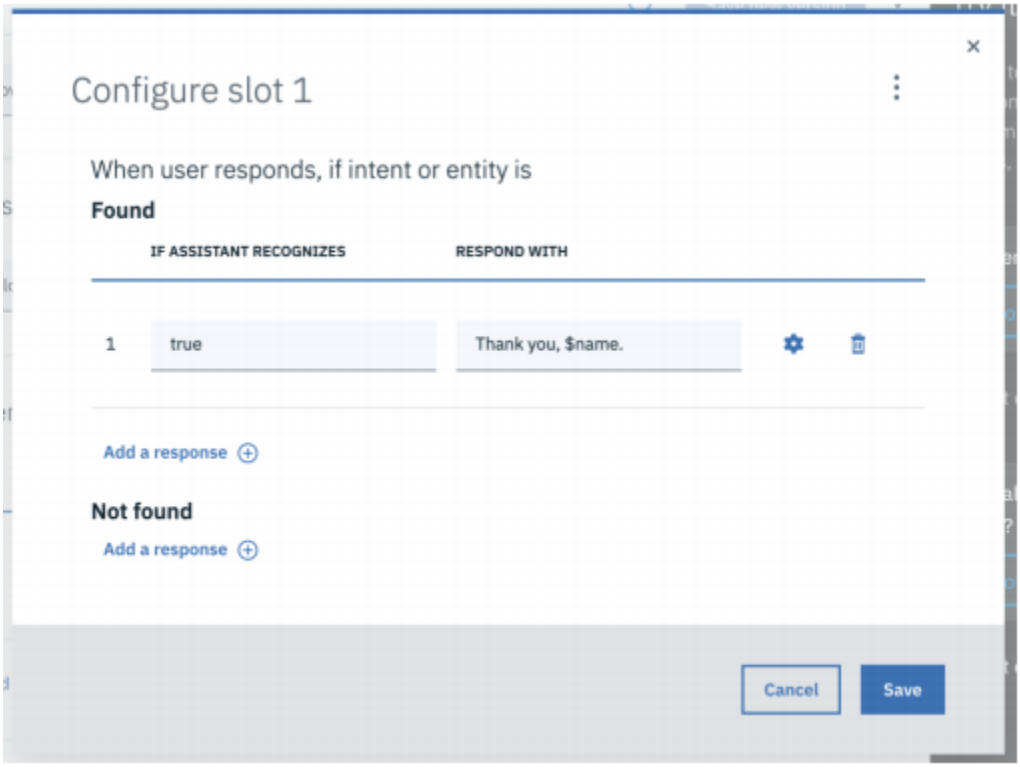
Let's see them in action.

1. Select the *Flower Recommendations* node and click on the gear icon next to *Required* **in our

node's slot.**

2. Configure the slot by scrolling to the bottom of the dialog that appears, and by clicking **_Add a

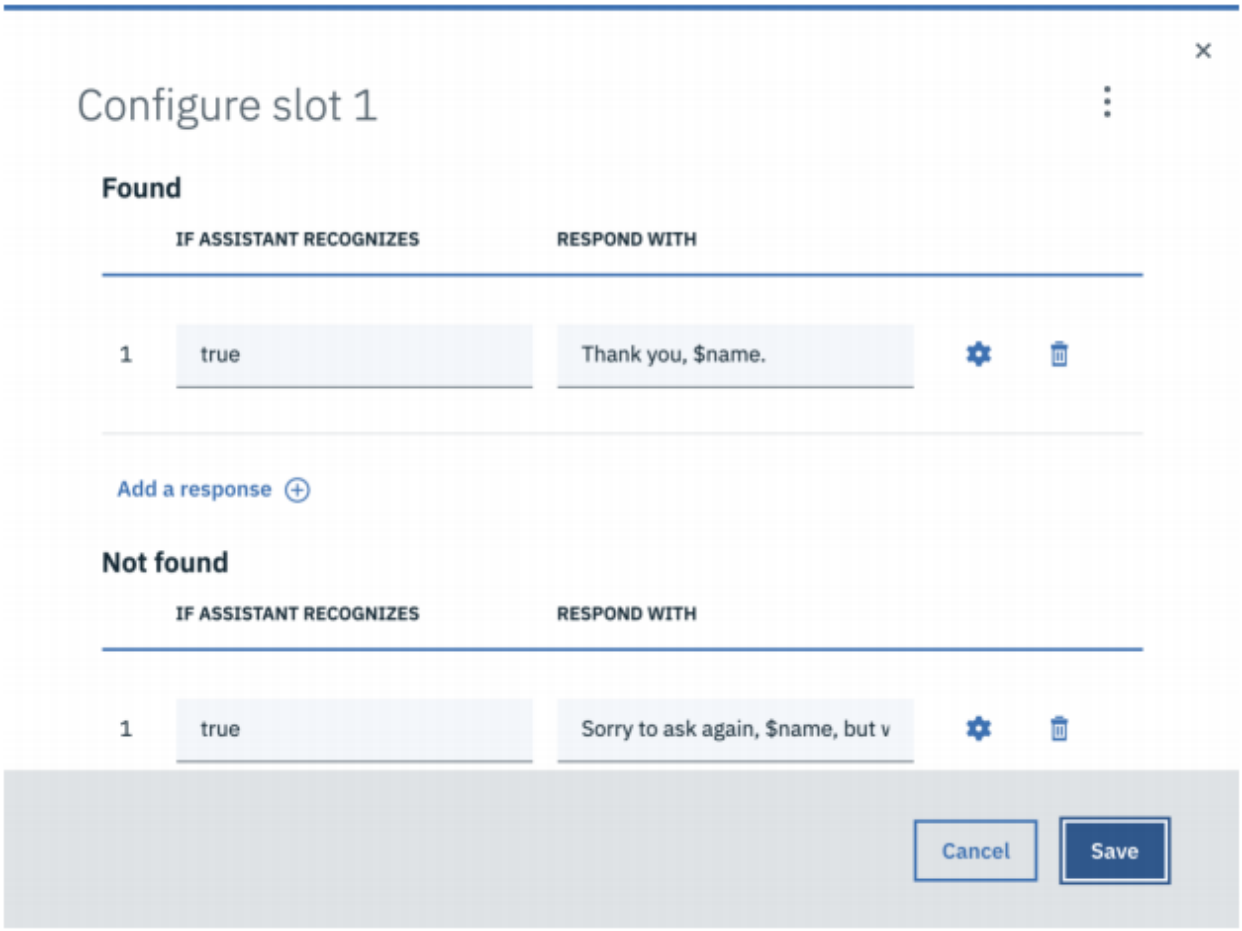
response_** below the *Found* section. Then enter true as the condition and Thank you, $name. as the response in the *Found* section.



This will ensure that when the slot receives the answer that it's looking for (the occasion in this case), we will always issue a thank you. If we wanted more refined control, we could add different responses for different inputs from the user. In our case, we are happy with a standard thank you response.

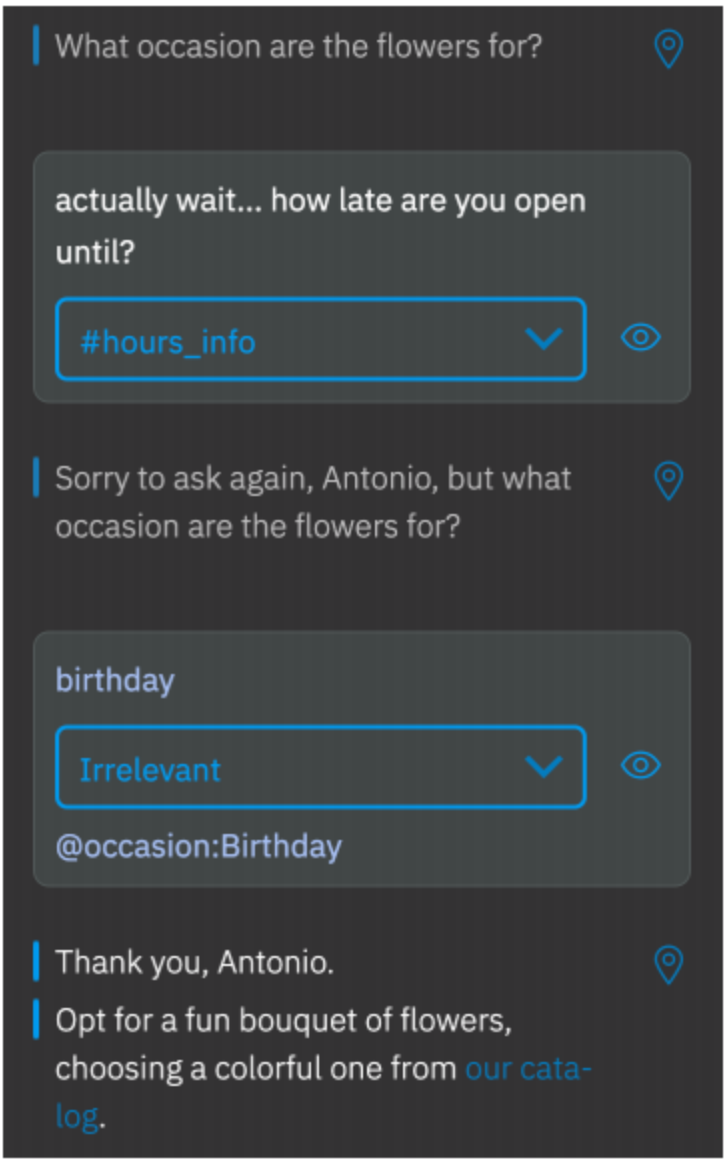
3. Similarly, add a *Not Found* **response with a true condition that responds with, Sorry to

ask again, $name, but what occasion are the flowers for?** as shown in the image below.



Finally, click *Save*.

Now, when we try the interaction again, we get a slightly more friendly interaction as shown below.



That's better, maybe even good enough! We are apologizing when we have to ask again, and we are thanking the user when they provide the answer we need. Since we have their name, we sweetened the deal by including it in our messages.

But... we are still not answering the user's legitimate side question about hours of operation. To properly handle that, we'll need *Digressions*.

Exercise 2: Enabling Digressions

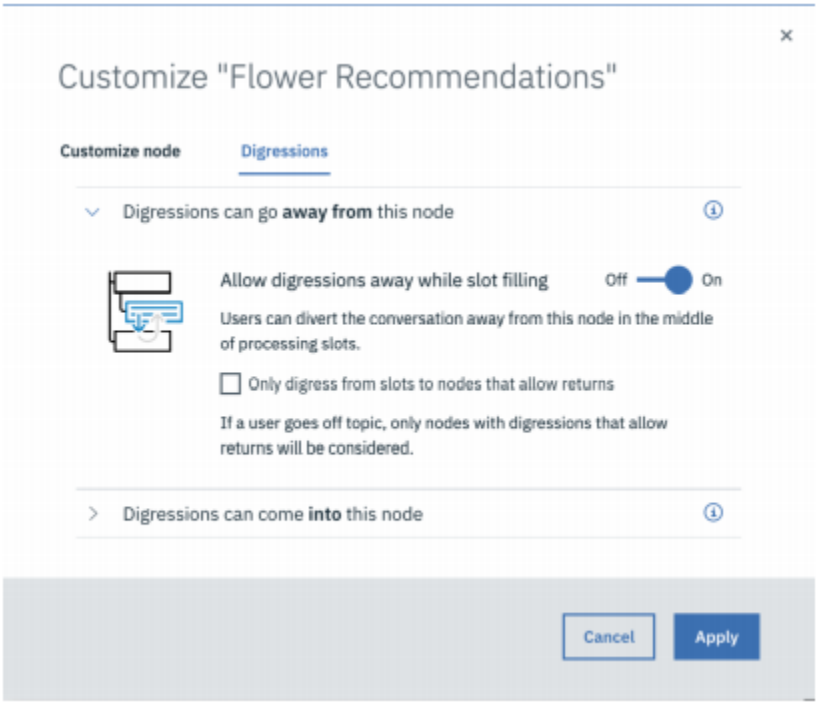
Digressions are a feature that enables nodes to pass control to a different node while they are processing a slot. What this means is that they allow the dialog to respond to a user's side question, while they are in the middle of answering a slot.

Let's see them in action.

1. The first thing we need to do is ensure that the node containing the slot allows digressing

away from it. Select the *Flower Recommendations* node again and click on *Customize* , and then its *Digressions* tab.

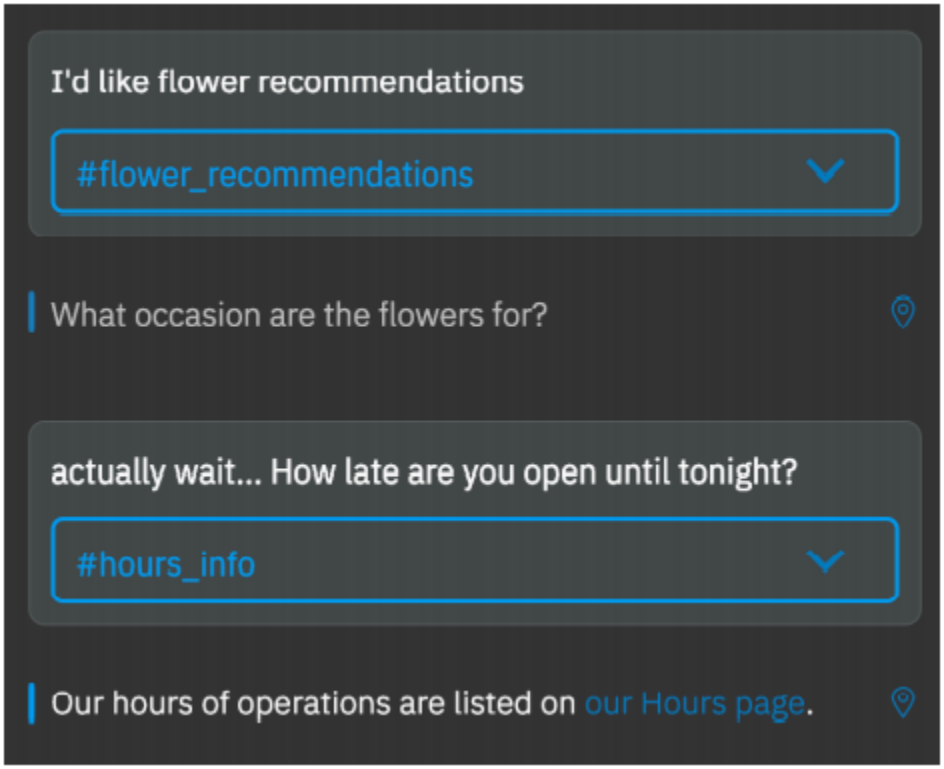
Here you'll want to expand *Digressions cannot go away from this node* by clicking on the > next to it and turn on the option to allow digressions away from this node. Make sure you click *Apply*.



With the digression away enabled in our node, we'll now be able to ask other questions in the middle of answering the slot's question and get a proper response for them.

Technically, the nodes we digress into need to allow the digression, but that's the default setting so we don't have to worry about it. (Should you ever need to prevent a node from being digressed into, you can disable the *Digressions can come into this node* for that node.)

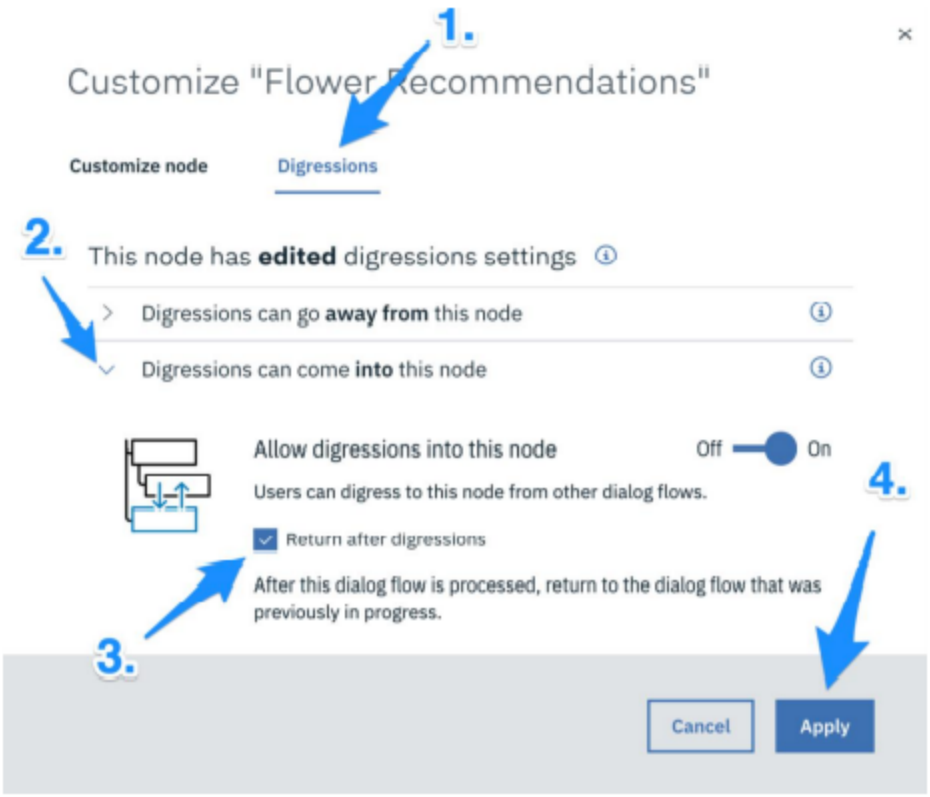
Let's see how this altered our interaction.



Awesome. We fixed our original problem. However, you might notice that we don't come back to the original question about flower recommendations.

This may or may not be what we want, depending on our chatbot. If we'd like to return, we'll need to explicitly set *Return after digression* in the nodes we might digress to.

Go ahead and set the option for all three nodes with slots (i.e., *Hours of Operation* , *Location Information* , and *Flower Recommendations*) as shown below. As a reminder, you can access the *Digressions* section by clicking on *Customize* in each of these three nodes.



Don't forget to click on *Apply* , and then test the interaction again, as shown in the image below.

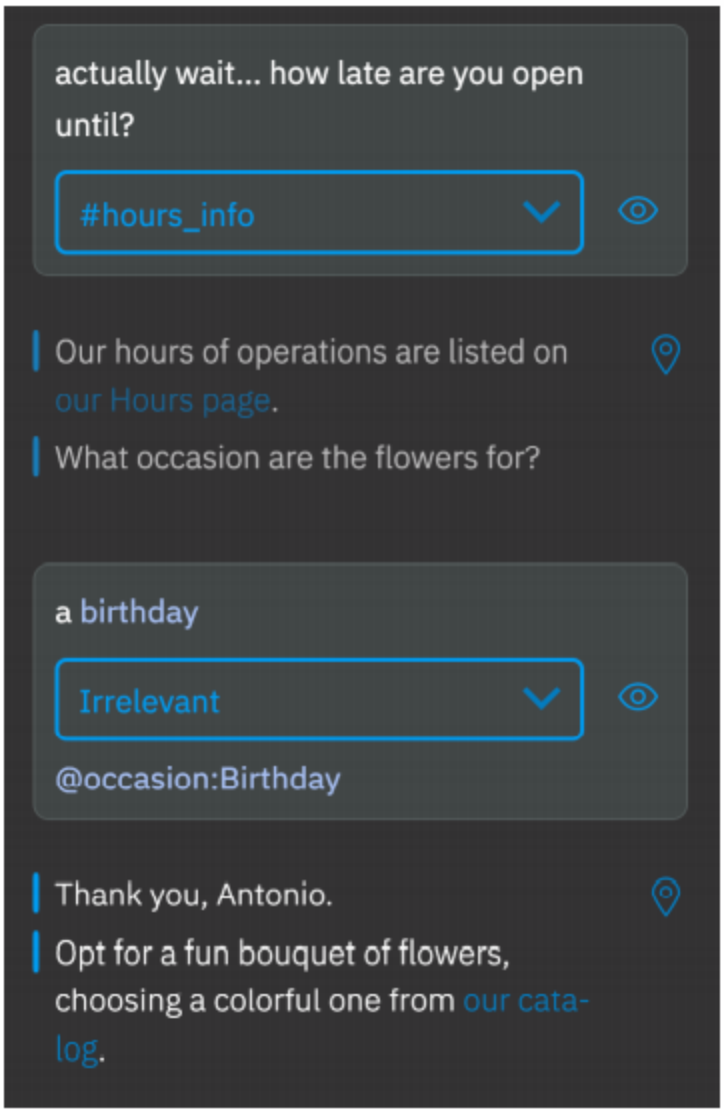
Next, clear the conversation and test it again.

(enter your name)

I'd like flower recommendations

actually wait... how late are you open until tonight?

birthday



That's quite nice! Our chatbot interacts like a polite human would in such a conversation.

2. Technically, neither *Hours of Operation* nor *Location Information* have required slots, so we

don't need digressions. But in the future, we might decide to make their slots mandatory, so as an exercise, there is no harm in enabling the digression for both nodes now. Go ahead and enable *Allow digressions away* for both nodes. 3. Test a few conversations with your chatbot from the WordPress site you deployed to earlier in the course.

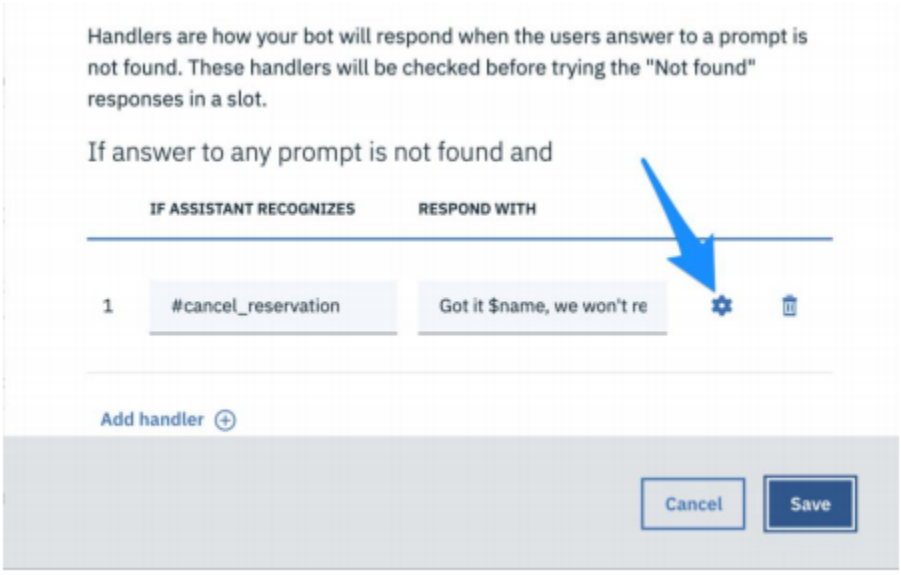
Found and *Not Found* responses, along with *Digressions* , are useful tools to add further polish to our chatbot and make it appear more human-like in its interactions with our users.

Handlers

We didn't need it in our chatbot, but sometimes we might want to execute certain actions if the user responds to a slot in a certain way. For example, if we detect that the user is trying to cancel a reservation, we want to escape/exit the slot rather than continue to pester the user for details until they get frustrated and just leave the chat.

To skip a slot when a certain condition is met, you might use *Handlers* , accessible via the *Manage Handlers* link in the slot section of your node. Handlers are evaluated even before the *Not Found* response is issued.

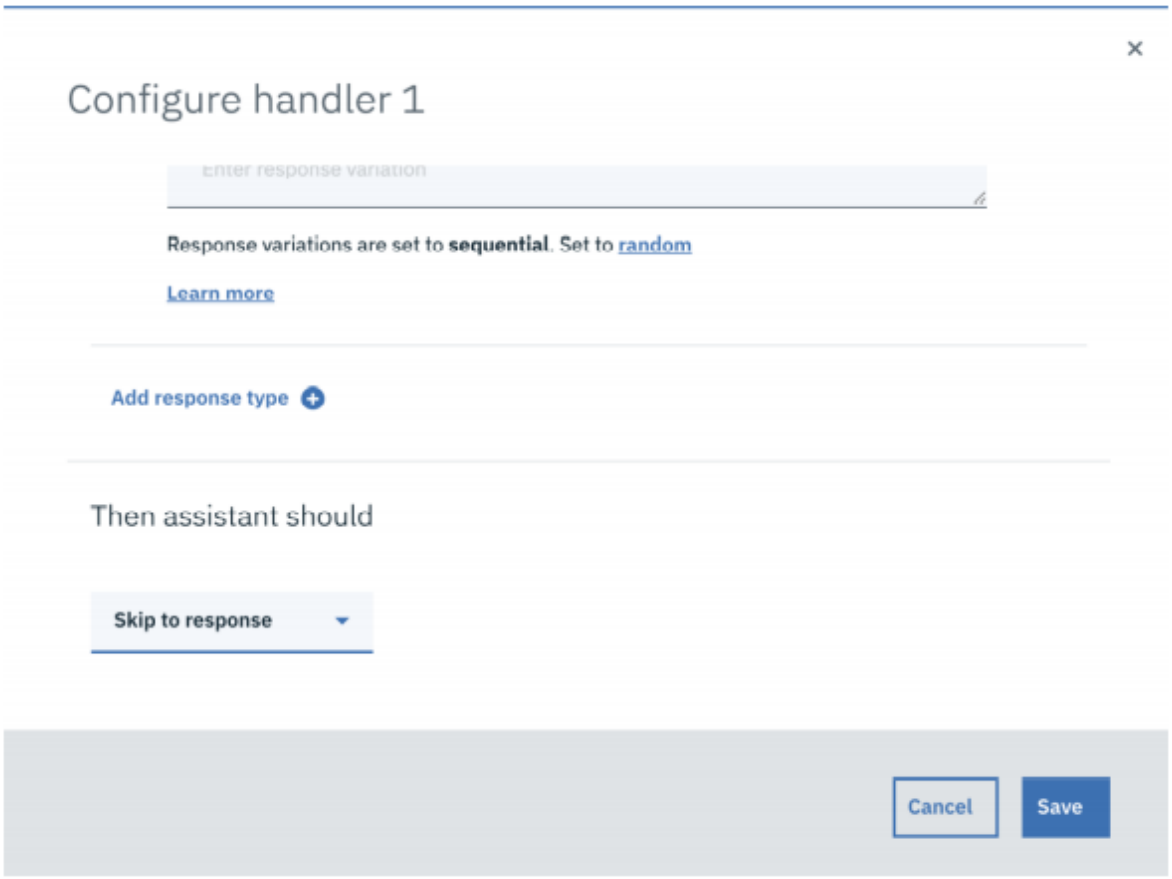
For example, we could have the following response in an handler within a reservation chatbot. Clicking on the gear icon offers us more detailed control.



This includes specifying that the handler should escape the current slot by skipping it when the handler's condition (e.g., a cancellation intent) is detected in response to the slot's question.

We can skip the current slot or skip all the slots in the node jumping straight to the response.

In the theoretical scenario of a reservation cancelled mid-booking, we'd want to skip any further question and would therefore opt to skip straight to response.



The fallback/ *true* response for such a reservation node, will likely have some kind of message inviting the user to reserve again at a later time or something equally user-friendly.

Again, this is not relevant to our chatbot, but I wanted you to know about the existence of *Handlers* and why they might be useful at times.

Our chatbot

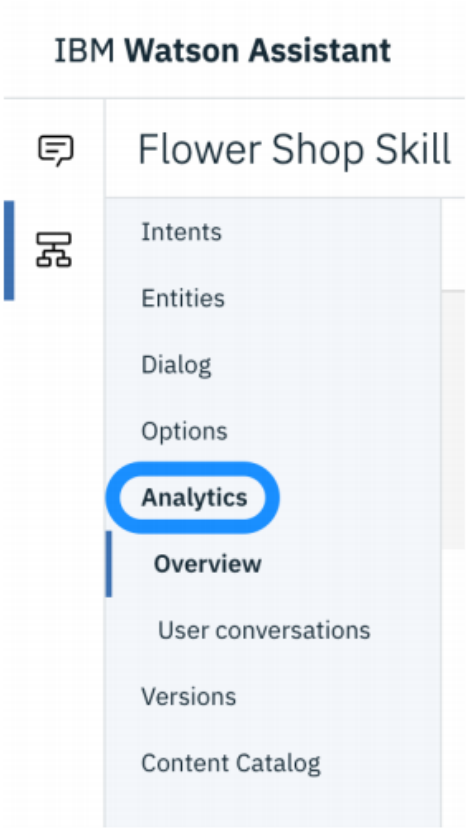
Congratulations on getting so far into the course. We're almost done. Meanwhile, if you need it you can download and import a JSON file of the dialog skill we created so far.

Objective for Exercise:

- Getting familiar with the Analytics Tab.

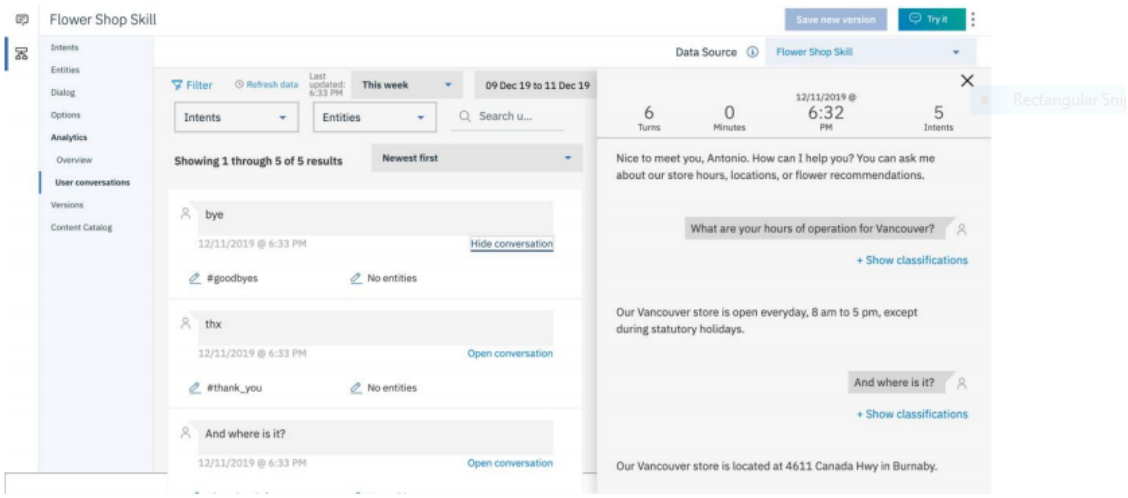
Exercise 1: Familiarize yourself with the Analytics tab

While working on your dialog skill, you might have noticed an *Analytics* tab.



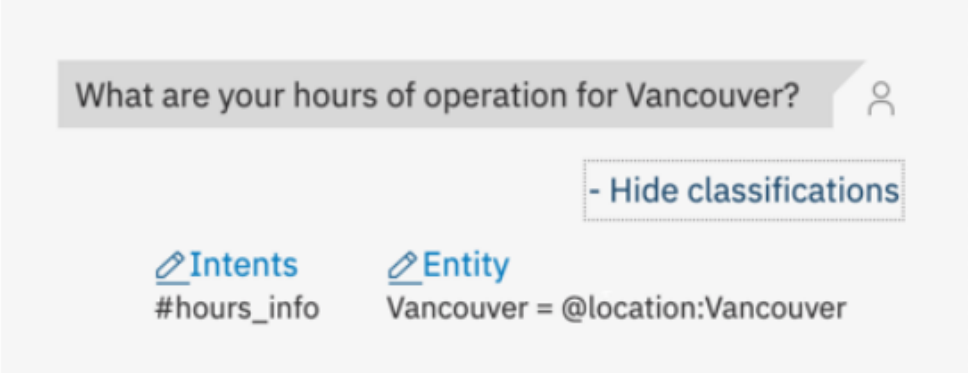
If you click on it , you'll see a dashboard with statistics and details about how your chatbot is being used.

This is quite useful once you deploy your chatbot in the real world. You want to know how your chatbot is being used, observe the conversation people are having with it (as shown in the image below), and see if there are ways to improve the chatbot accordingly.



For example, if you find out that a lot of people are asking about something that your chatbot doesn't know how to handle, it might be time to create a new intent and node to handle that scenario and provide helpful responses to the user.

Likewise, if Watson misclassified an intent, it would be good to correct it and you can do so directly from the *Analytics* > *User conversations* tab.



The statistics are also helpful because they tell you what your customers are focusing on. With that knowledge in hand, you can invest more time to refine your chatbot to answer very in- demand questions.

All this feedback can even be useful to refine the product itself at times.

For example, if the users complain to your chatbot that they are unable to find a certain feature, it might be wise to improve the UI to make that feature more obvious or easy to find in the app itself.

It's worth noting that this *Analytics* tab will be empty for you if you haven't had some conversation with the chatbot via the WordPress chat box. The reason for this is that *Try it out* sessions are not included in the *Analytics* tab.

So, go ahead and chat with your chatbot through the WordPress pop up chat window , if you haven't done so already. Then spend some time to familiarize yourself with the *Analytics* capabilities built in Watson Assistant , by exploring this tab.

Congratulations

Congratulations on completing all the labs within this course! With the knowledge acquired so far, you should be able to build simple but useful chatbots and deploy them on websites powered by WordPress.

Watson Assistant in the Private Cloud

Due to regulations concerning data handling, some organization may require their chatbot to be available on premises (in a private rather than public cloud).

Watson Assistant (and other IBM Cloud services) are now available on premises as well.

If your company or a client of yours has such requirements, consider reading the official announcement here.

Welcome to the new experience on Watson Assistant

Objective for Exercise:

- Create an instance of new Watson Assistant.
- Create a Banking Assistant.

Overview

The new Watson Assistant experience, focused on using actions to build customer conversations, is designed to make it simple enough for anyone to build a virtual

assistant. Building, testing, publishing, and analyzing your assistant can all now be done in one simple and intuitive interface.

- New navigation provides a workflow for building, previewing, publishing, and analyzing your assistant.
- Each assistant has a home page with a task list to help you get started.
- Build conversations with actions, which represent the tasks you want your assistant to help your customers with. Each action contains a series of steps that represent individual exchanges with a customer.
- A new way to publish lets you review and debug your work in a draft environment before going live to your customers.
- Use a new suite of analytics to improve your assistant. Review which actions are being completed to see what your customers want help with, determine if your assistant understands and addresses customer needs, and decide how can you make your assistant better.

Create an instance of new Watson Assistant

To access the resources and services that the IBM Cloud provides, you need an IBM Cloud account.

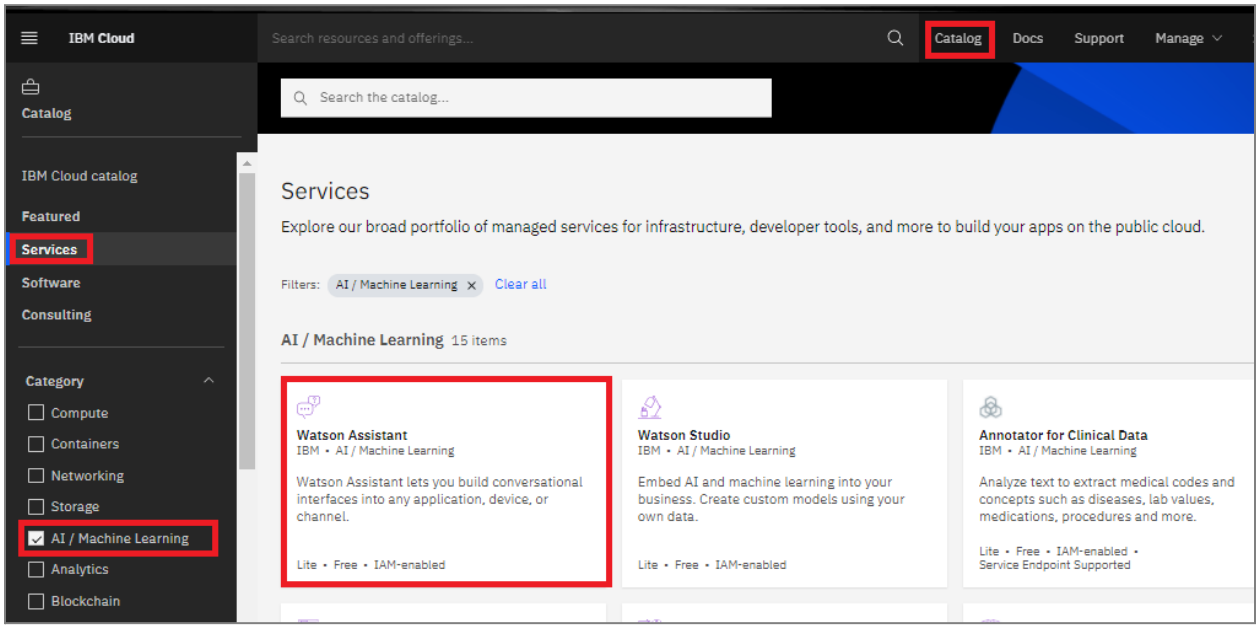
If you already have an IBM Cloud account, you can directly move on to creating Watson Assistant Service. Otherwise, please follow this lab create an IBM Cloud account.

Task 1: Add a Watson Assistant Service Instance

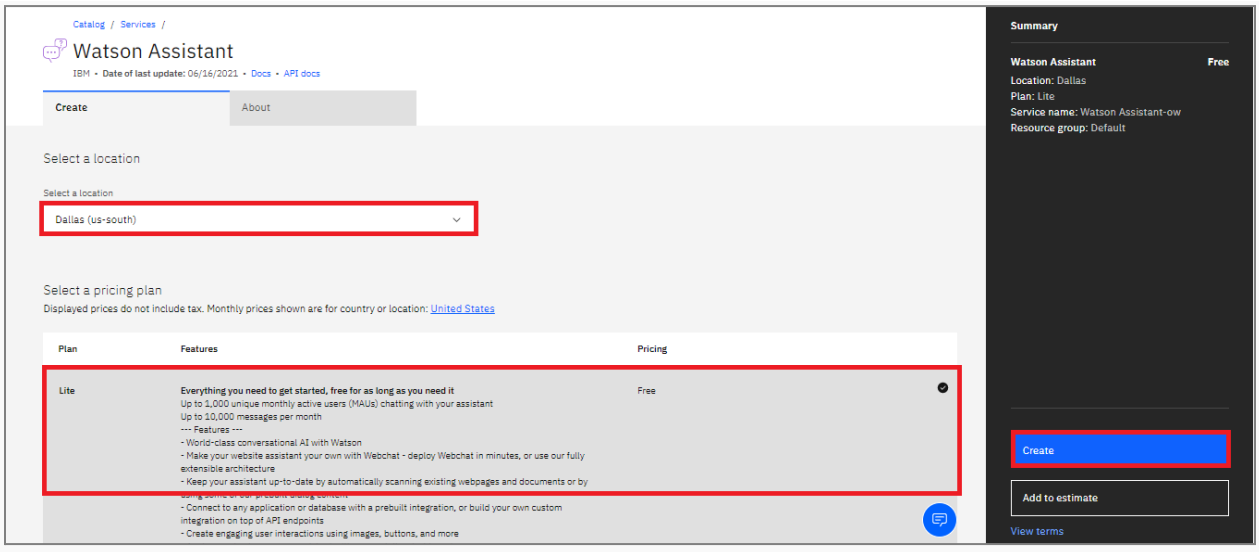
We need to add an instance of the Watson Assistant service to your account. This will be used later to create chatbot.

1. Click on the IBM Cloud Catalog
   From the catalog page that loads, Click on the services, select the AI category

and then select the Watson Assistant service from the list.

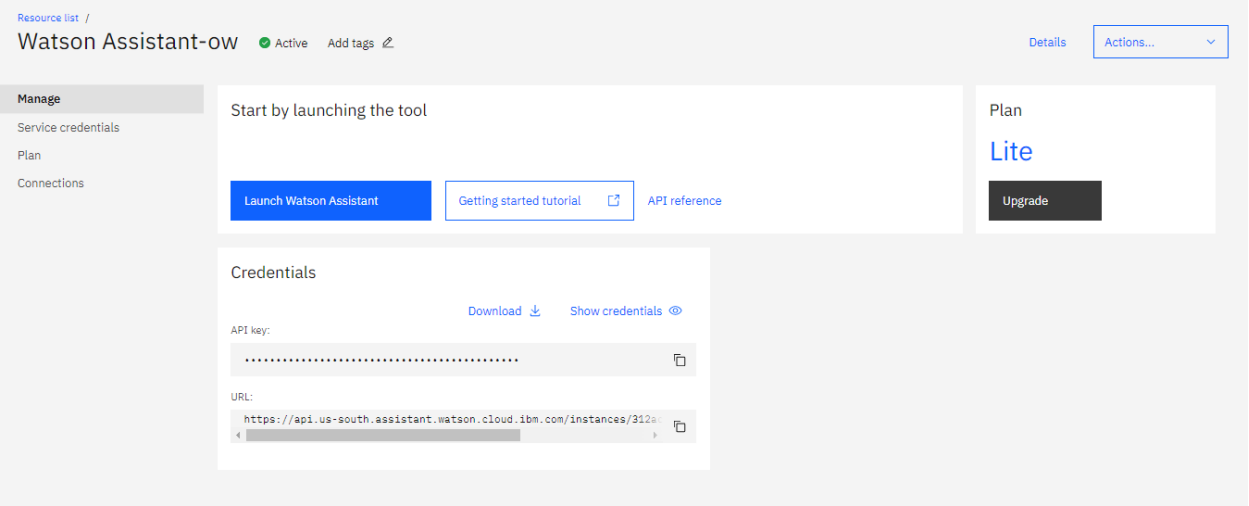


2. In the Watson Assistant setup page, choose a region depending on where you are located. For example, here we have chosen Dallas. Verify that the Lite plan is selected, and then click Create.
   Note: If you created a Watson Assistant instance in the past using the Lite plan, you won't be allowed to create a second instance. You can use the instance you already created for this course or you'll need to delete your previous instance and create a new one.



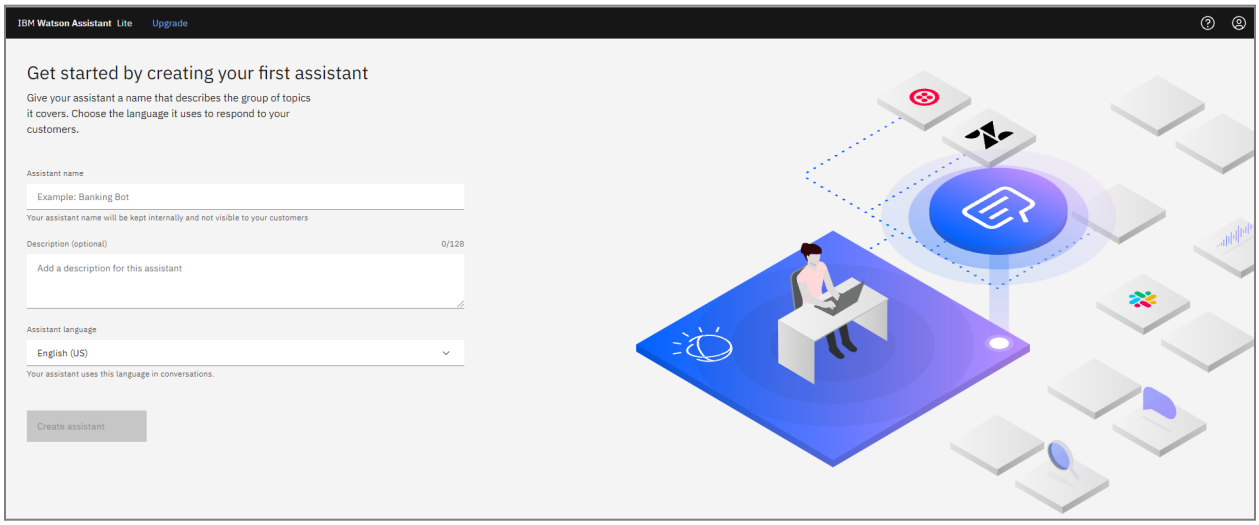
3. Once the service is created you will see the below page, from where you can launch Watson Assistant. Click on the Launch Watson Assistant button to access

the web application that will allow you to create chatbots.



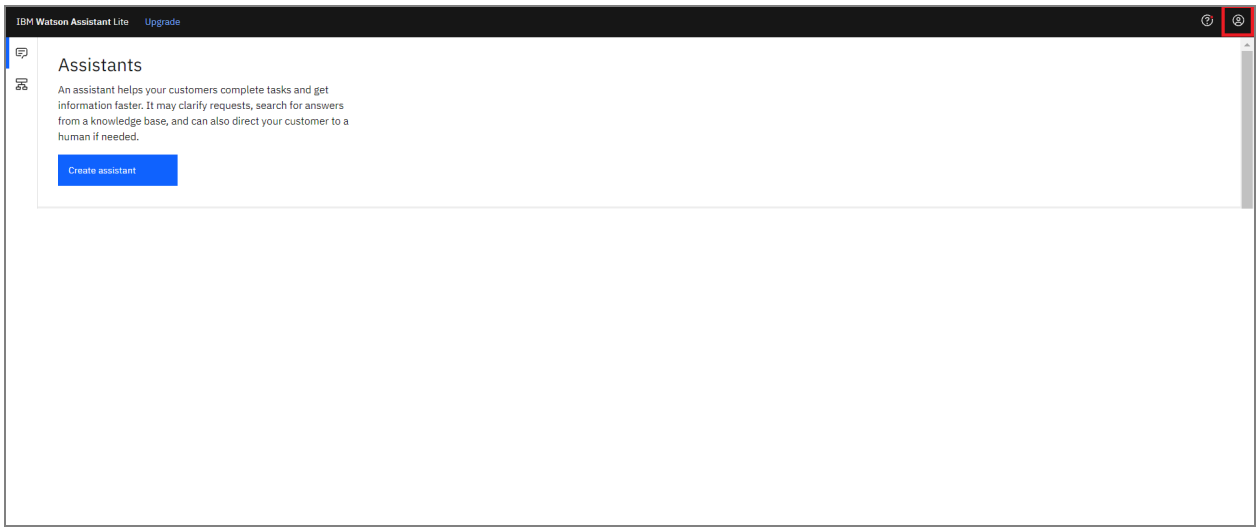
Task 2: New Watson Assistant instance

Some instances are automatically provisioned with the new Watson Assistant.

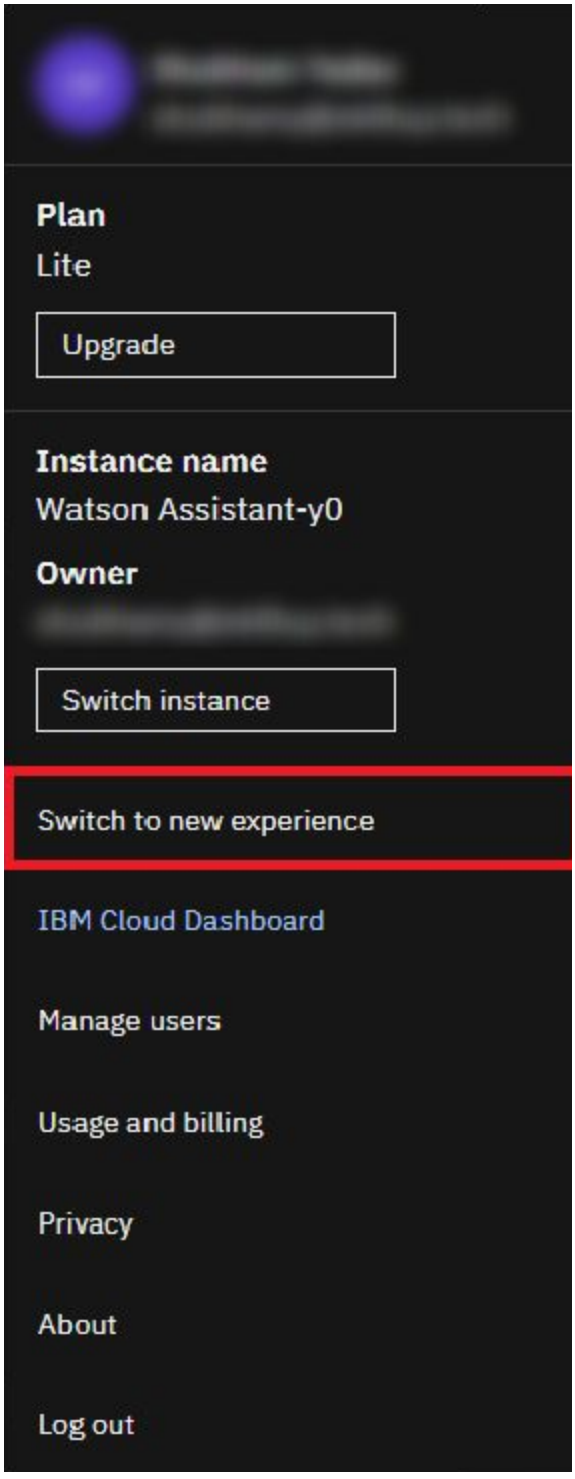


If your instance is still using the classic Watson Assistant, you can switch to the new experience by following these steps:

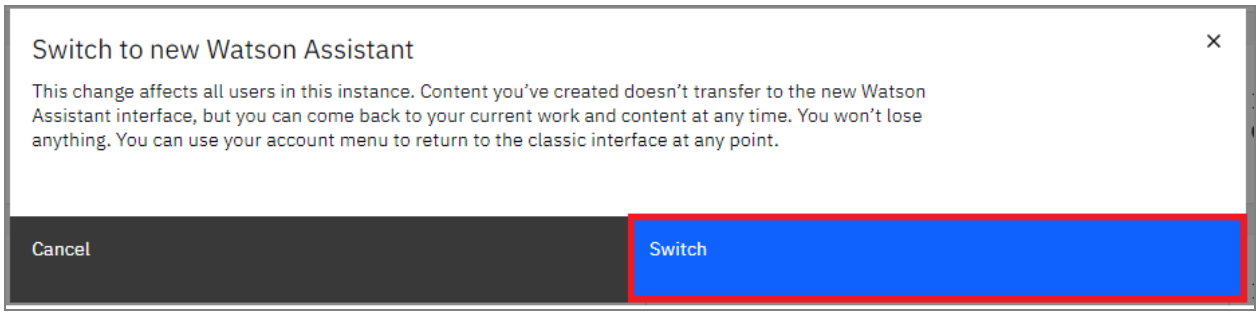
1. From the Watson Assistant interface, click the User Panel icon to open your account menu.



2. Select Switch to new experience from the account menu.



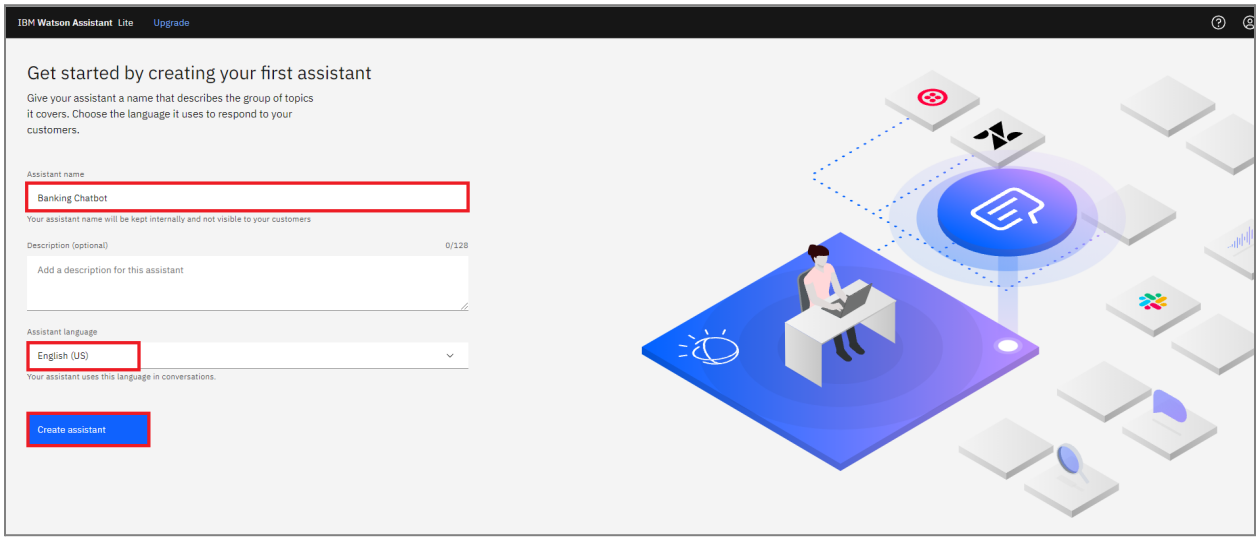
3. Click Switch to confirm that you want to start using the new experience.



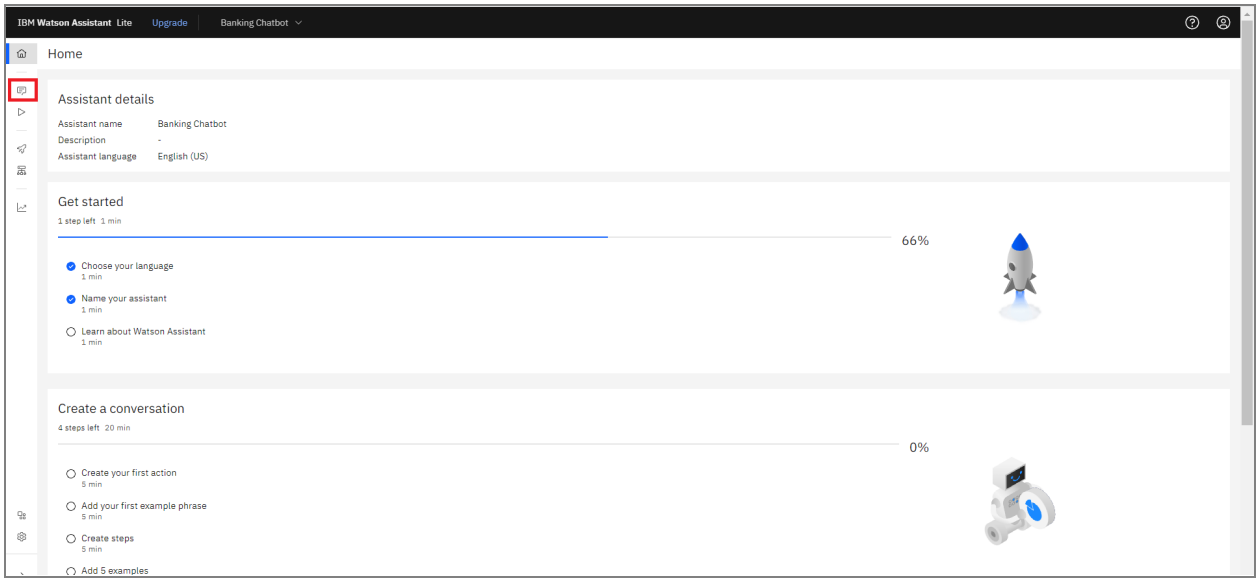
> Note: You won't lose any work if you switch to the new experience, and you can switch back to the classic experience at any time by clicking Switch to classic experience from the account menu.

Create a Banking Assistant

Give your assistant the name Banking Chatbot. Choose the language it uses to respond to your customers, For this lab please select English and click on Create.

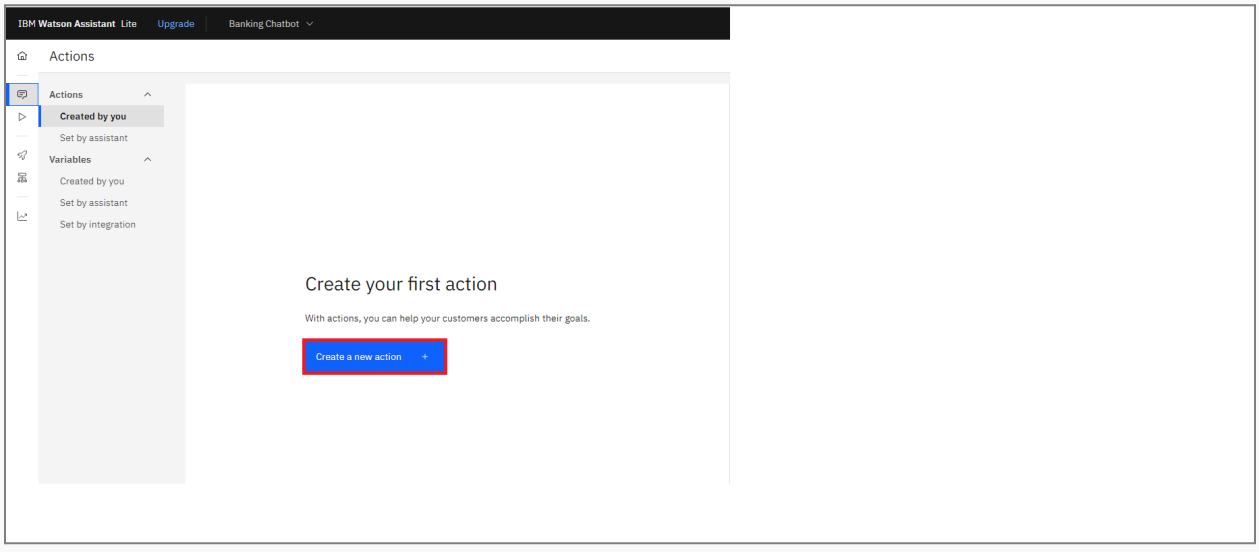


You are on the Home page of your Assistant, you can also see the Assistant details here. Please click on the Actions represented by the message icon, see the below screenshot for reference.



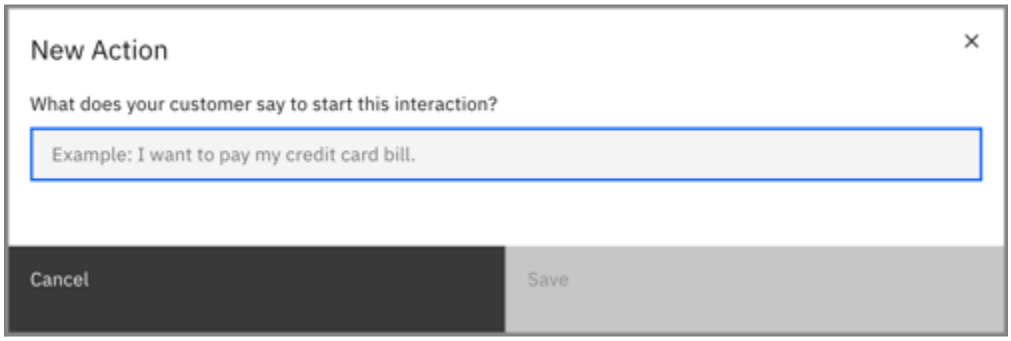
Now, you can create your first action, with actions you can help your customers accomplish their goals. Please click on Create a new action button.

> Note: An action represents a discrete outcome you want your assistant to be able to accomplish in response to a user's request. An action comprises the interaction between a customer and the assistant about a particular question or request.



Creating and editing an action

When you create a new action, Watson Assistant prompts you for an example of the customer input that starts the action. This text is also used as the default name for the action, but you can edit the action name later.

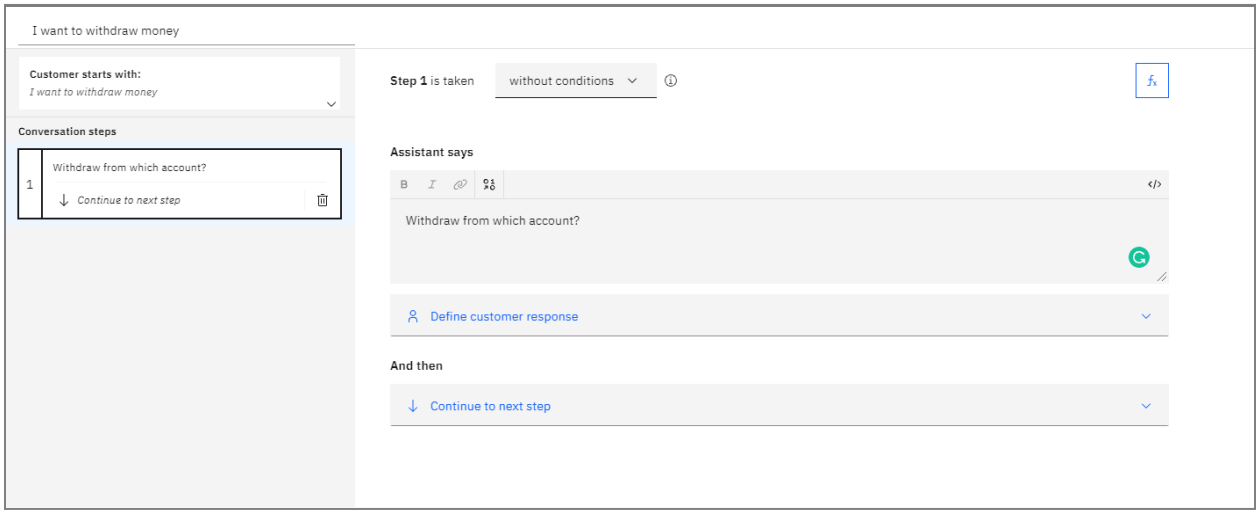


Type *I want to withdraw money* and then click Save to create the action.

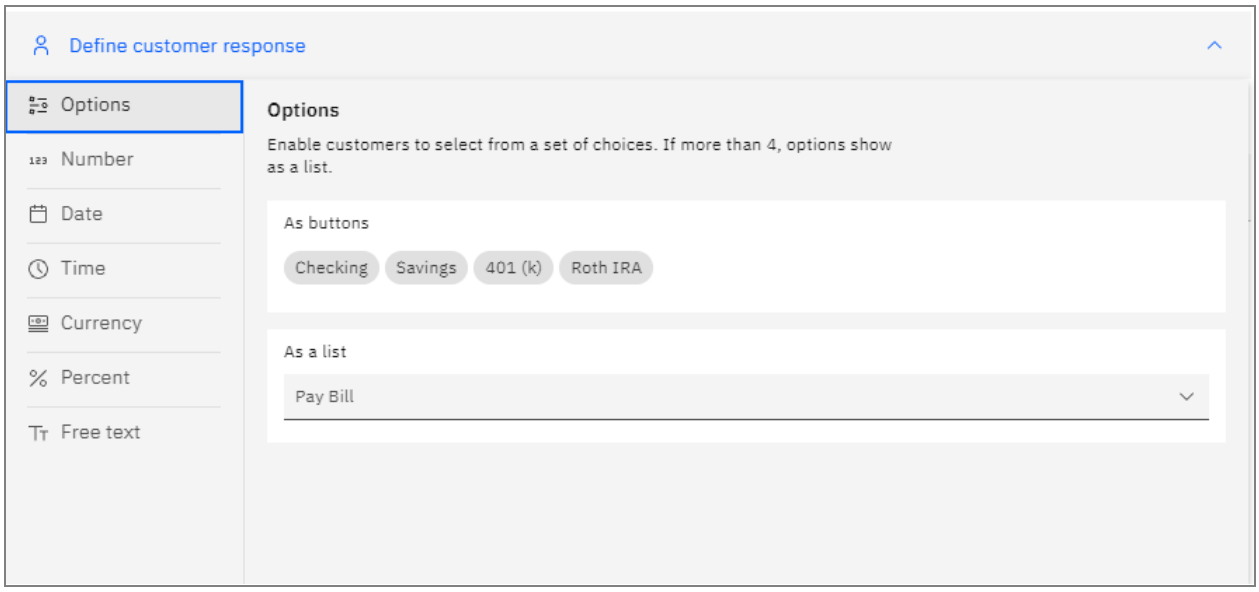
> Note: Initially, you only need to specify one example of typical user input that starts the action. You can add more examples of user input later if you want to.

After you create the action, the action editor opens. In the Step 1 is taken field, leave the default value of without conditions. This step is always required for any withdrawal.

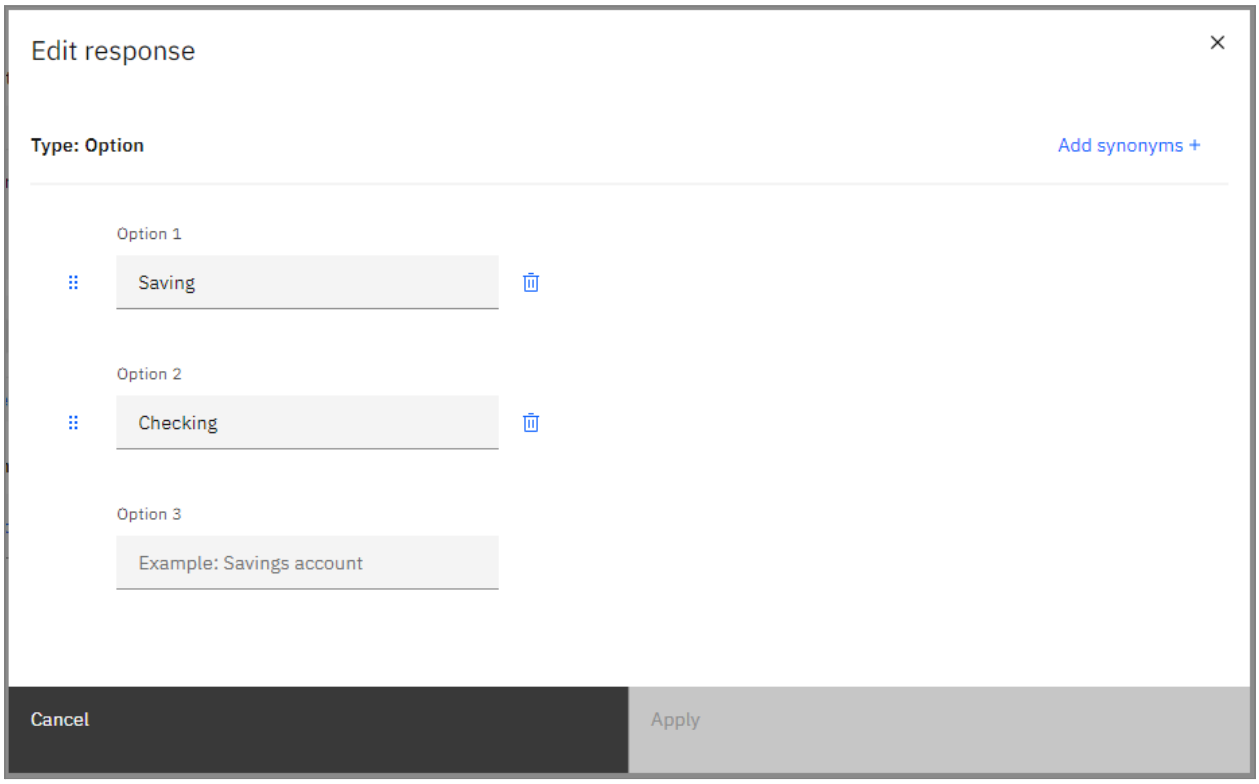
In the Assistant says field, type *Withdraw from which account?*.



Now, Click Define customer response and select Options because we will be asking the user to select from a list of predefined choices(Saving or Checking)

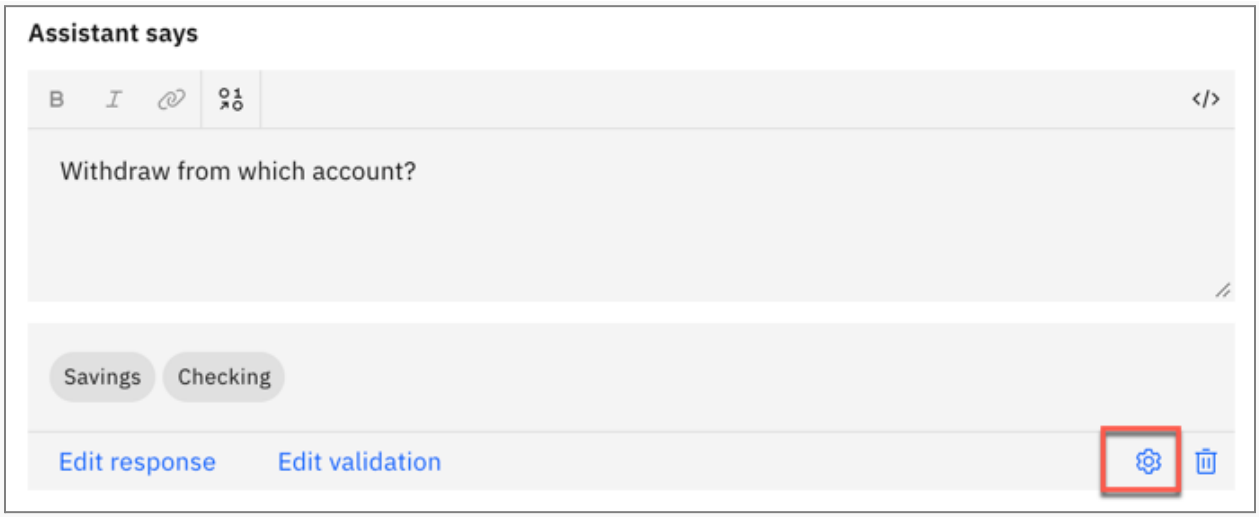


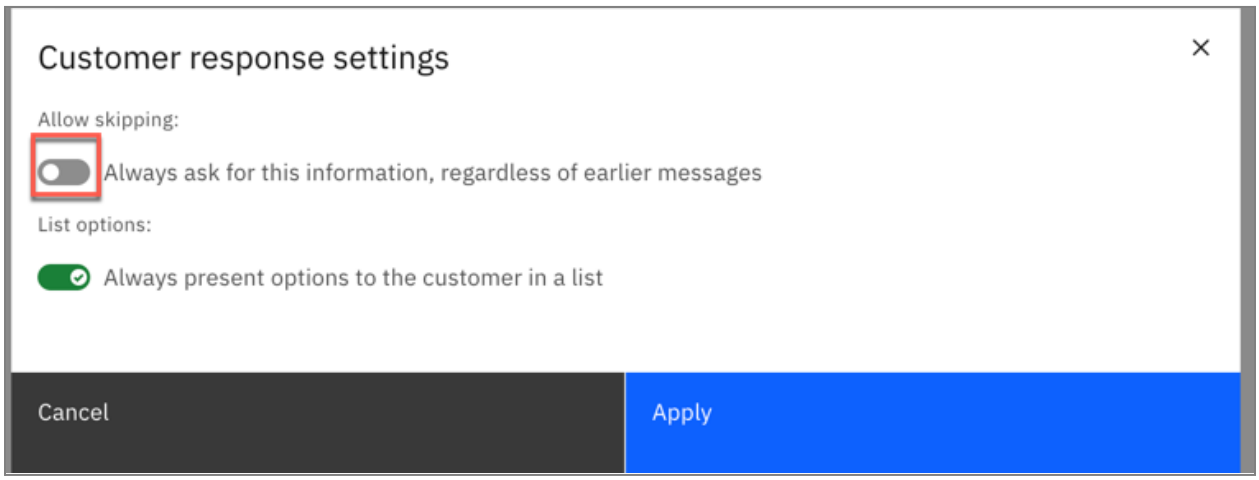
In the Option 1 field, type Savings. As soon as you enter a value for option 1, a field appears for options 2. Navigate to Option 2 and type Checking.



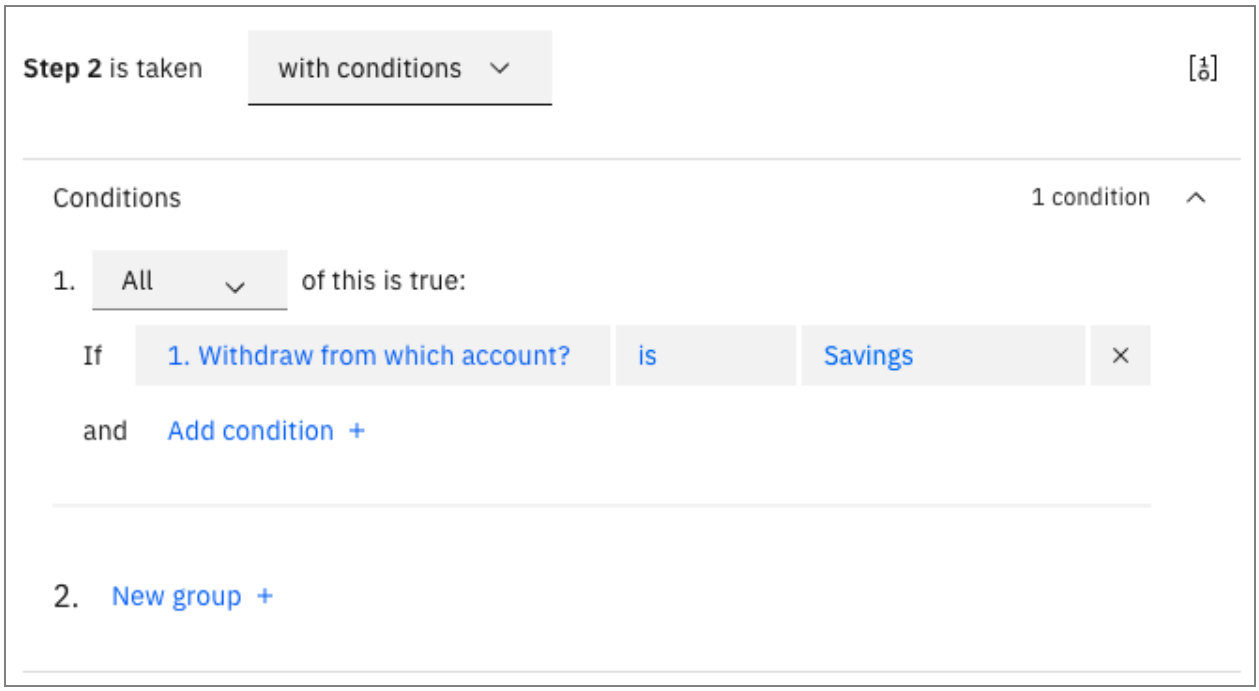
Click Apply to save the customer response.

Note: If the customer already specified the account type previously; for example, if the initial customer input was *I want to withdraw money from my savings account* then it won't present the options to you. But if you want the options to always be present you can do that by clicking on the Settings icon and selecting the Always ask for this information, regardless of the earlier message.

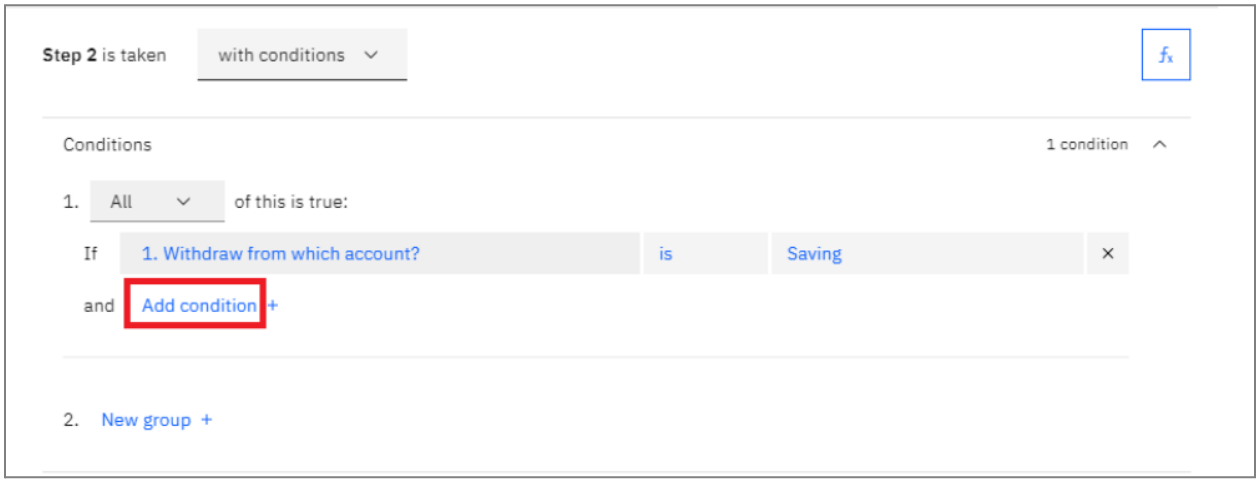




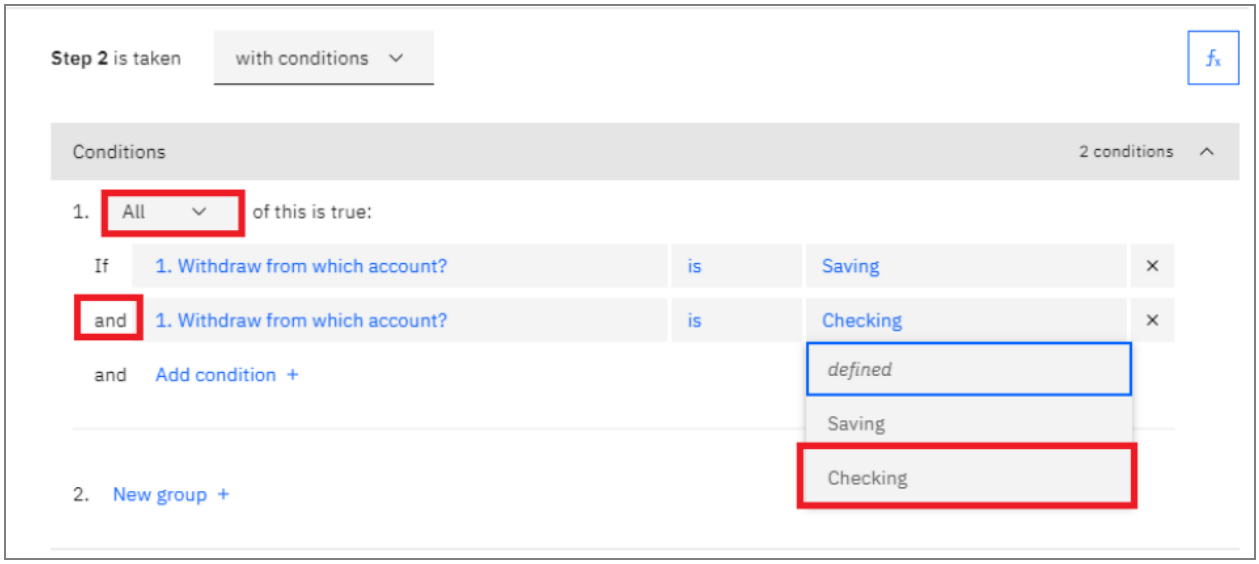
Now, click New step. In the Step 2 is taken field, select with conditions. The Conditions section expands.



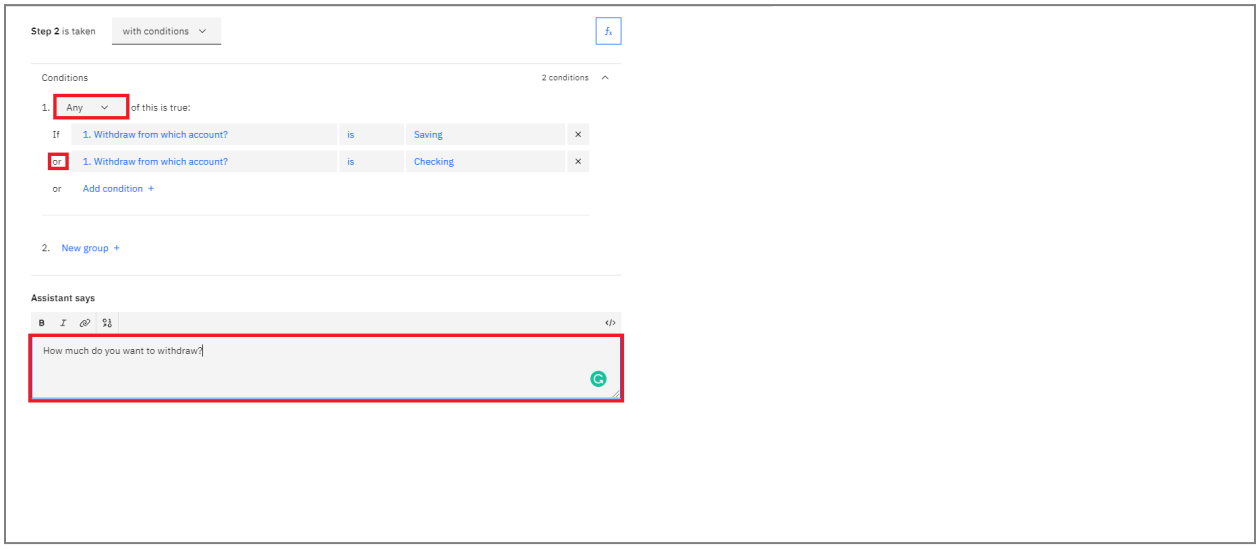
By default, a condition is automatically created based on the action variable stored by the previous step (Withdraw from which account?). However, by default it is checking for a value of Savings.



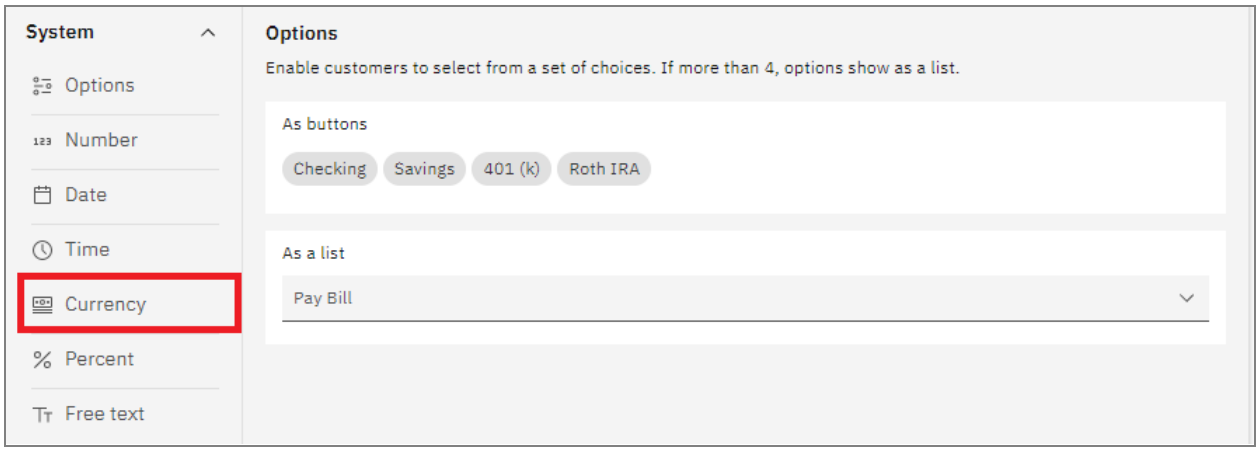
Now, click on the Add condition and select Checking. For both conditions Saving and Checking the assistant should ask *How much do you want to withdraw?*.



Also, change from All option to Any. In the Assistant says field, type *How much do you want to withdraw?*.



Click Define customer response. This time we need the customer to specify a monetary amount, so select Currency. There are no more details you need to specify for a currency amount, so it is immediately added to the step.



Finishing the action

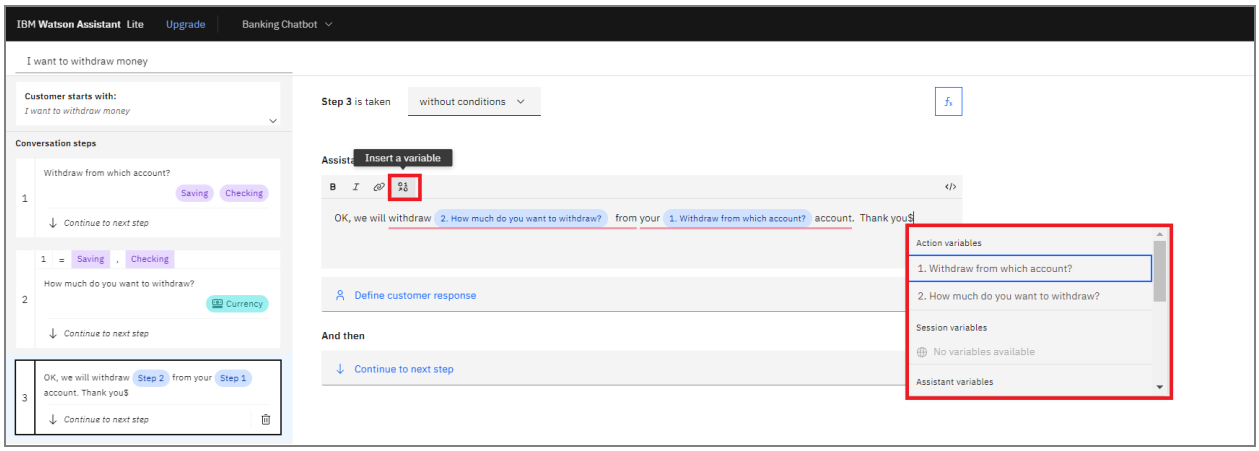
We now have all the information we need. For our example, we're not going to implement any real logic for making a withdrawal, but we can send a final message summing up what we're doing.

To do this, we need to insert action variables (representing customer responses from previous steps) into our response. At run time, these action variables will be replaced with the actual values supplied by the customer.

Click New step. Now we need to build a confirmation message that says "OK, we will withdraw *amount* from your *account_type* account.Thank you."

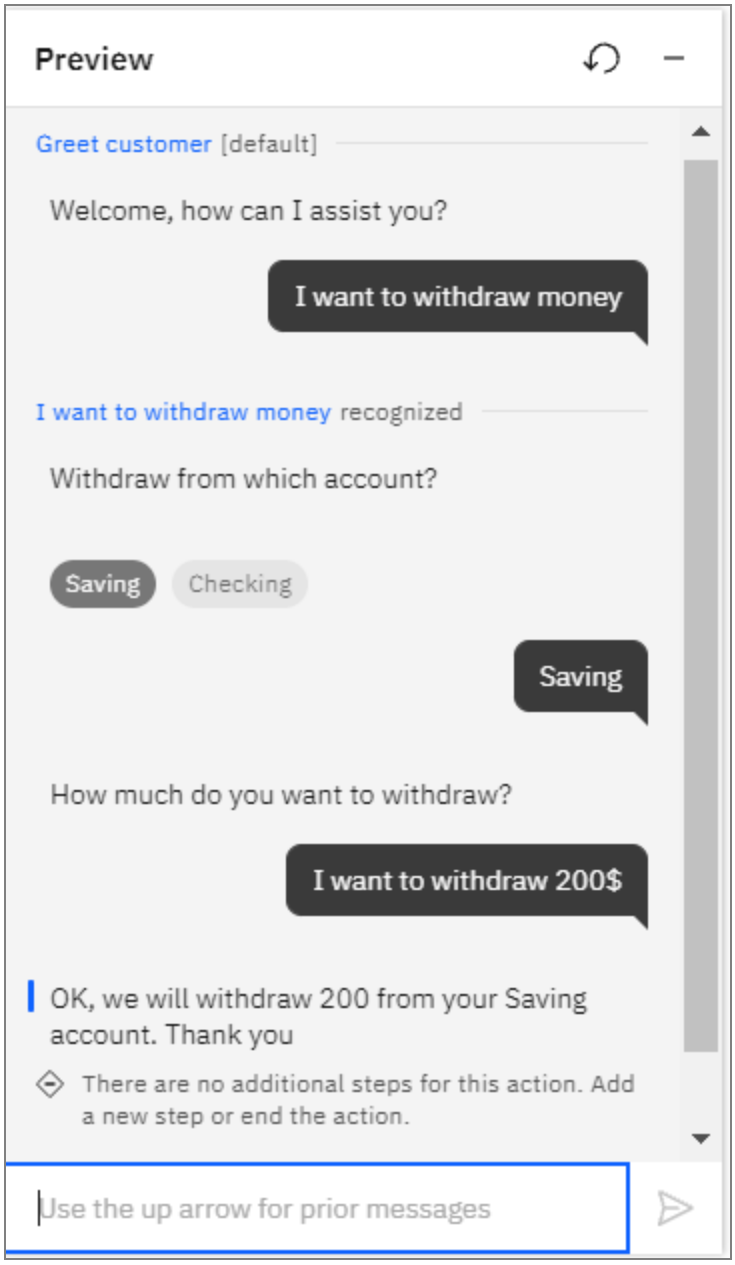
To create this response, type the text of the message in the Assistant says field, but in place of the variable values, click the Insert a variable icon to insert references to action variables:

- For amount, select 2. How much do you want to withdraw?.
- For account_type, select 1. Withdraw from which account?.



Because this is the last step in the action, you don't need to specify any customer response.

Please click on Preview button to verify your chatbot responses.



Thank you for completing this lab.