

Lab 10: Explore 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.

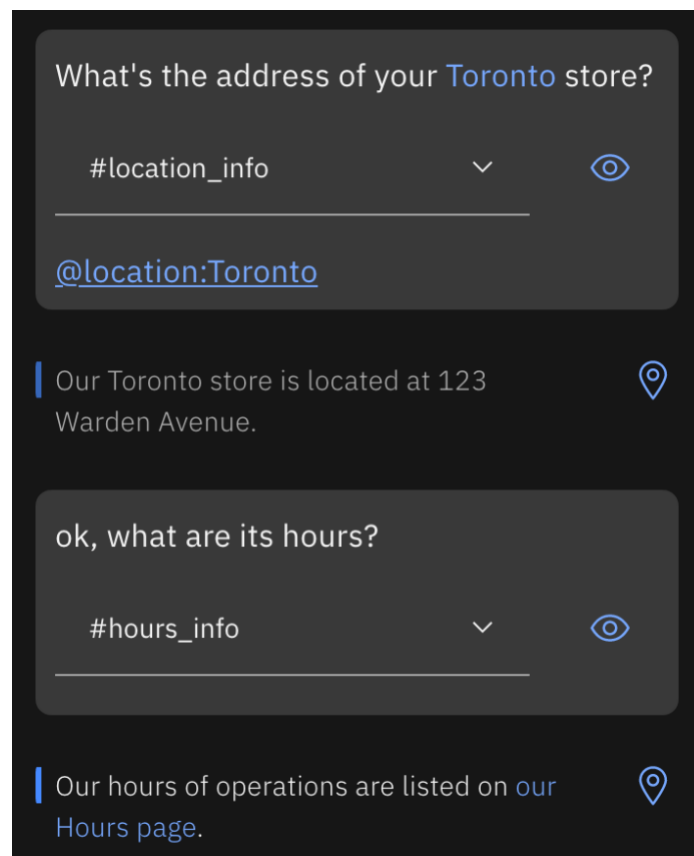
Keep in mind that some of these concepts are tougher to get, particularly if you have no prior programming experience. So, don't be discouraged if you don't fully get everything right away.

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.



Lab 10: Explore Context Variables

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.

Lab 10: Explore Context Variables

If assistant recognizes

@location — +

Then set context

Variable	Value
city	"@location"

[Add variable](#) +

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* node 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.

Lab 10: Explore Context Variables

The screenshot displays the IBM Watson Assistant interface. On the left, a workflow diagram shows a sequence of nodes: 'Welcome' (welcome), 'Assign City' (@location), 'Jump to Hours of Operation (Evaluate condition)', 'Hours of Operation' (#hours_info), 'Location Information' (#location_info), 'Chitchat', and 'Anything else' (anything_else). The 'Assign City' node is highlighted with a blue border and a checkmark icon.

On the right, the configuration panel for the 'Assign City' node is shown. The node name is 'Assign City'. Below it, the condition 'If assistant recognizes' is set to '@location'. The 'Then set context' section shows a table with one variable 'city' and its value '@location'. The 'Assistant responds' section is empty. The 'Then assistant should' section is set to 'Jump to' and points to the 'Hours of Operation (Evaluate condition)' node.

Variable	Value
city	"@location"

Then assistant should
Jump to Hours of Operation (Evaluate condition)

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"*).

Lab 10: Explore Context Variables

Context variables ⓘ ×

\$Enter variable name

\$timezone

⊖

"America/Vancouver"

\$city

⊖

"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.









Lab 10: Explore Context Variables

Our Locations

Customize  

\$city  

Assistant responds

	IF ASSISTANT RECOGNIZES	RESPOND WITH		
1	\$city:Toronto	Our Toronto store is open Monday to		
2	\$city:Montreal	Our Montreal store is open Monday t		
3	\$city:Calgary	Our Calgary store is open Monday to		
4	\$city:Vancouver	Our Vancouver store is open everydæ		

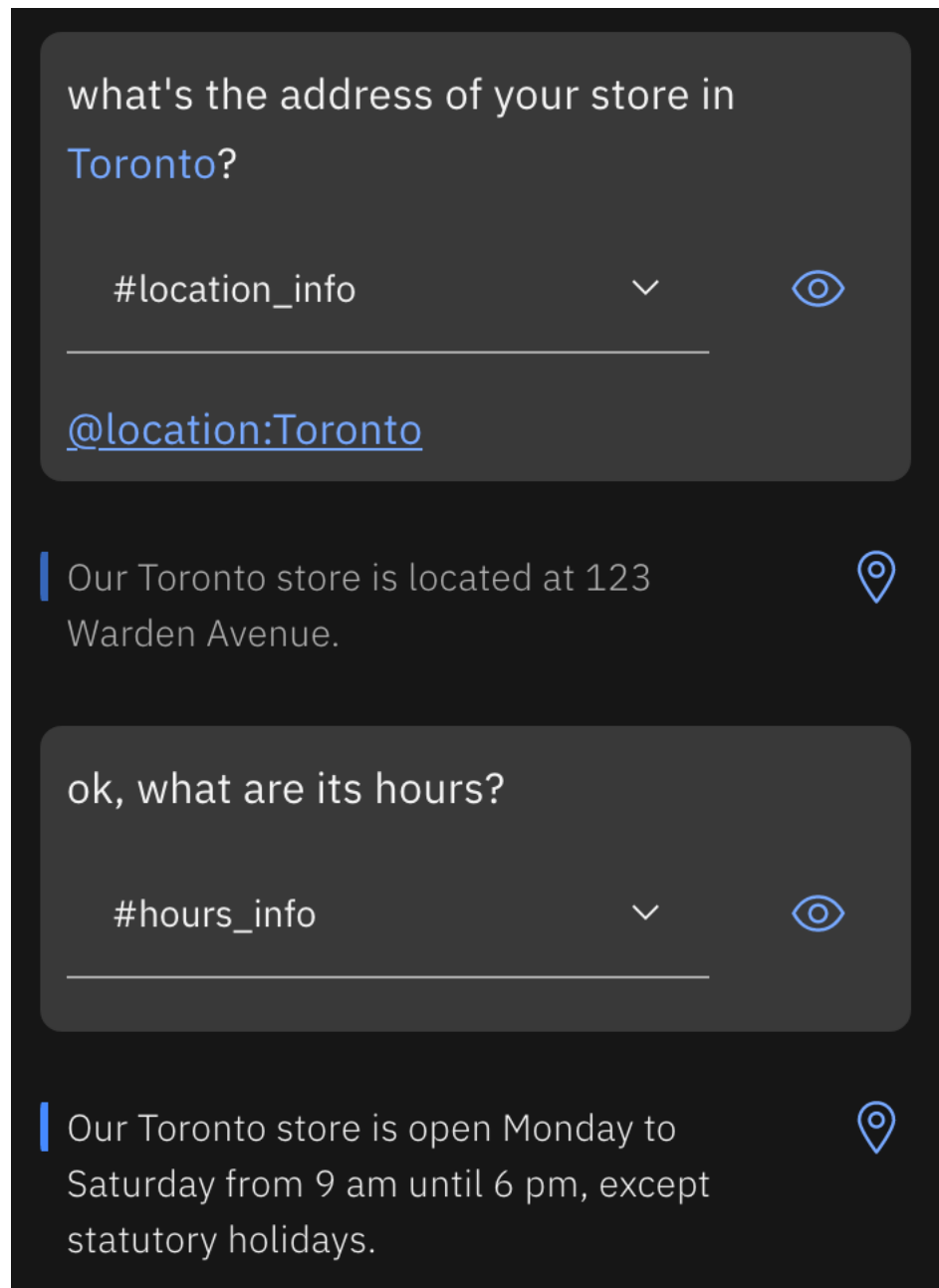
Make sure you repeat this process for both *Our Locations* child nodes.

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!

Lab 10: Explore Context Variables



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:

Lab 10: Explore Context Variables

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 `<? input.text ?>`

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 `<? input.text ?>`**.

Lab 10: Explore Context Variables

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.

The screenshot shows a chatbot configuration interface for a node titled "Collect Name".

- Node Name:** "Collect Name" (with a "Customize" link and gear icon).
- Description:** "Node name will be shown to customers for disambiguation so use something descriptive." (with a "Settings" link).
- Trigger:** "If assistant recognizes" with a dropdown set to "true".
- Context Setting:** "Then set context" with a table:

Variable	Value
name	"<? input.text.substring(0, 1).toUpperCase() + input.text.substring(1) ?>"
- Response:** "Assistant responds" with a dropdown set to "Text". The response text is: "Nice to meet you, \$name. How can I help you? You can ask me about our store hours, locations, or flower recommendations." Below this is a text input field labeled "Enter response variation".

Lab 10: Explore Context Variables

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.

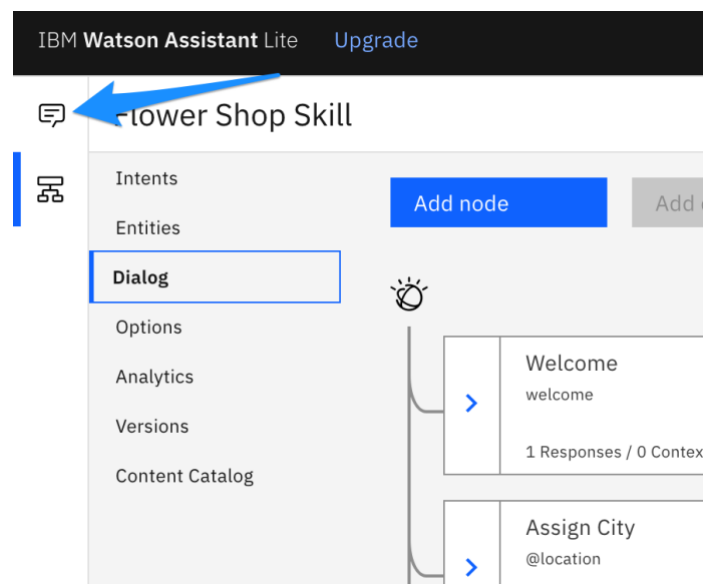
[Help! It didn't work.](#)

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 file](#) with 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.



Lab 10: Explore Context Variables

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*.

IBM Watson Assistant Lite Upgrade

← Assistants

Flower Shop Chatbot

A chatbot to assist our flower shop customers.

Skill

A dialog skill provides specific responses you've created. Choose one for your assistant. [Learn more](#)

Dialog

Flower Shop Skill

LANGUAGE: English (US) **TRAINED DATA:** 5 Intents | 3 Entities | 22 Dialog nodes

DESCRIPTION: ---

LINKED ASSISTANTS (1): Flower Shop Chatbot

- View API details
- Export
- Swap skill
- Remove 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 Skill* tab, upload the JSON file you just downloaded (by clicking on *Choose JSON File*) and then click *Import*.

Swap dialog skill

Create a new skill or swap for an existing one

Add existing skill

Create skill

Use sample skill

Import skill

Select the JSON file for the dialog skill with the data you want to import.

Choose JSON File

Flower-Shop-Skill.json ×

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.