By now you know the essentials of chatbot building. There are however some 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.

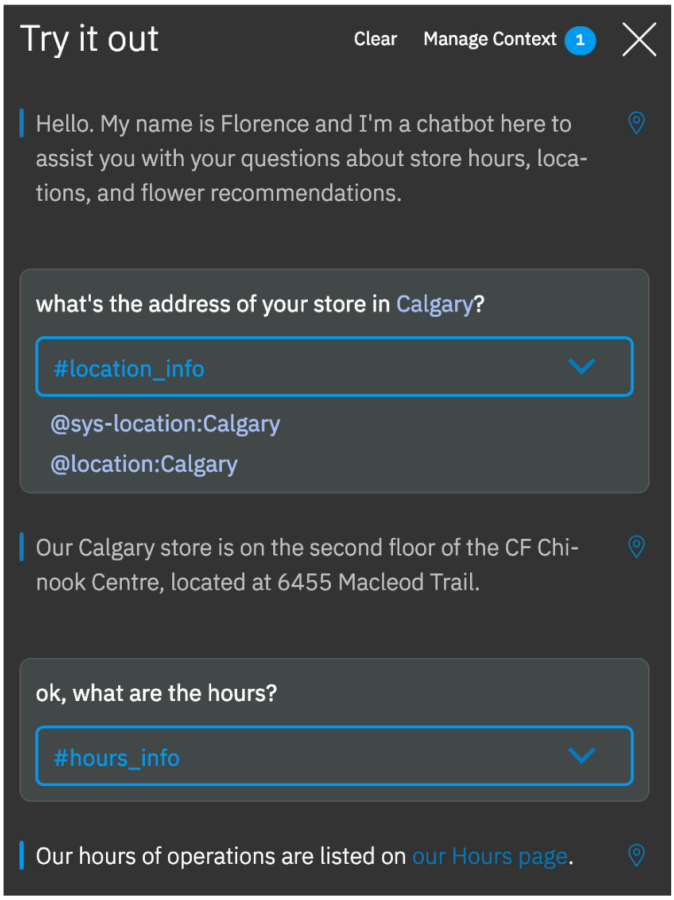
Keep in mind that some of these concepts are tougher to get, particularly if you have no prior programming experience. So don't be discouraged if you don't fully get everything right away.

You can try things, 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.

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 ability to appear smarter and remembering the context of the conversation so far. For example, consider the following interaction.



A human customer 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., Calgary). 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 it's available for the duration of the conversation? Enter the concept of context variables, which allows us to do just that. As we go about collecting information from the user, we can store them 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 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.

1. In the *Dialog* section of your skill, select the Welcome node and then **click *Add node*** 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 three dots to the right of Then respond with: and **select *Open context editor***.

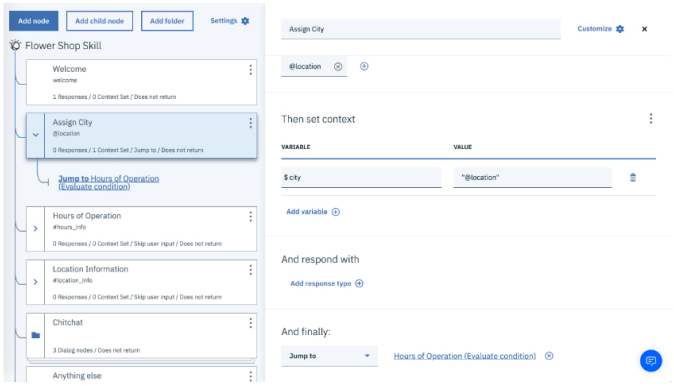
4. You'll be offered the ability to set one or more context variables whenever this node is executed. **Enter $city for the variable name, and @location for the value.**

In the And finally 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. Jumping to the response directly? No. **Select is *If assistant recognizes (condition)*** so that this node can be evaluated as it normally would.

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 this context variable assigning 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 like shown in the image below.

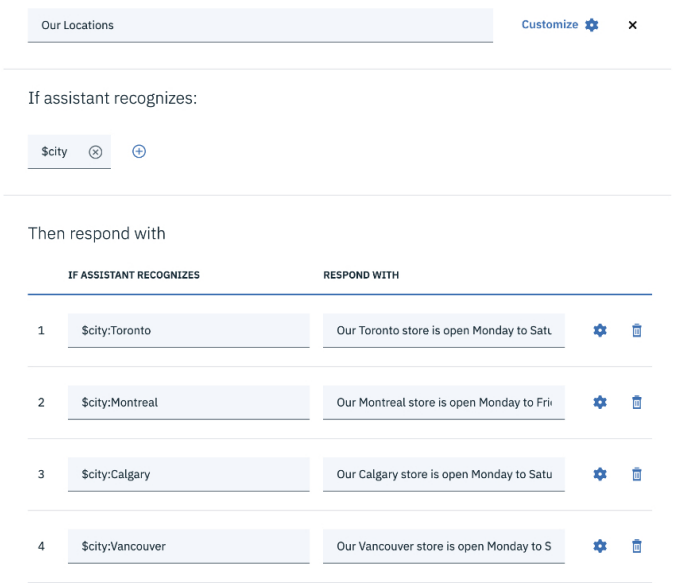


5. Head over to the *Try it out panel* and **ask What are your hours of operation?.** Click on *Manage Context* at the top of the panel to see the content of the Context (i.e., its variables). The $timezone variable will already be set for you automatically, but because we didn't specify a location, 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 panel starts a new conversation, and so context variables are cleared as well. Go head and close the context manager again.

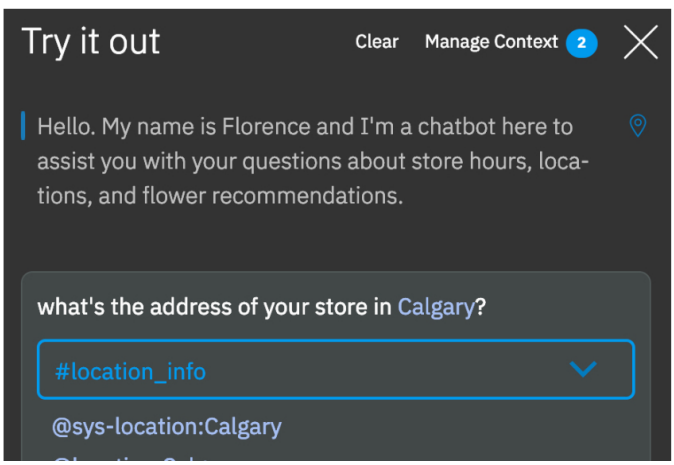
7. We want to make sure that $city variable is set whether it was specified along with a request for hours information (as we already did) or for location addresses. So as a sanity check, **try where is your Calgary store?.** You should see that the city in the context now changes to the string "Calgary".

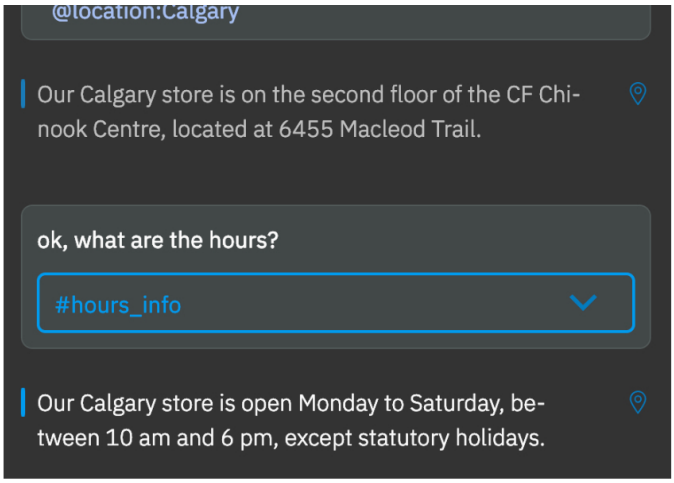
8. Alright, we now store city in our trusty $city context variable. To make use of it, we'll need to change our Our Locations child nodes under the *Hours of Operation* and *Location Information* parent nodes. There is an easy way to do this. **Simply replace @location with $city for every occurrence** 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.

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 Calgary?** followed by **ok, what are the hours?** You should now see a smarter response as shown in the image below!





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.

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 with no context variable 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, so as 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?

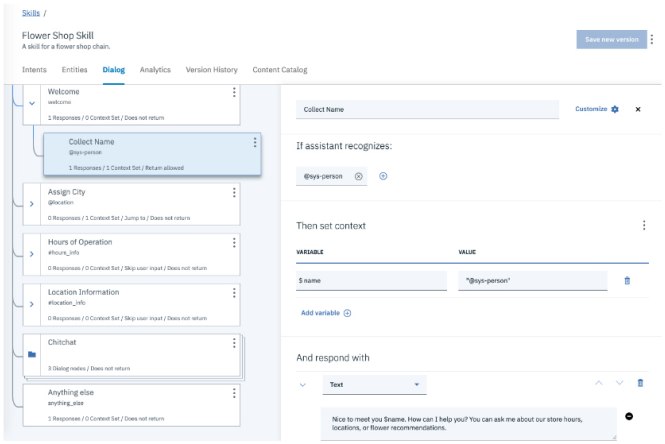
1. Since we are dealing with names, **let's start by enabling the @sys-person entity** from the *System entities* section. Watson will start training (as expected).

2. Back in the Dialog, 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?**

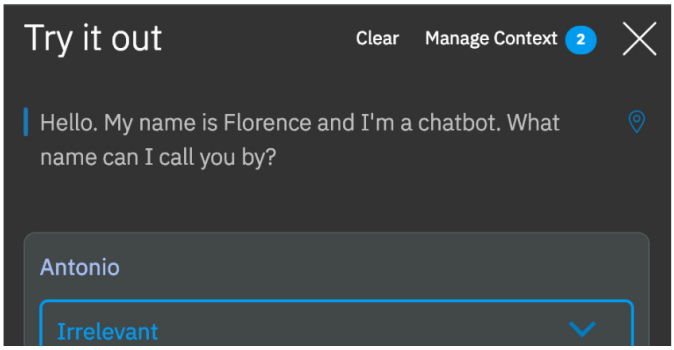
3. **We need a child node to actually collect the name** (the answer to our question, in other words). S**o go ahead and create a child node under *Welcome.*** Call it Collect Name. For the condition, we want to detect that a @sys-person name was provided.

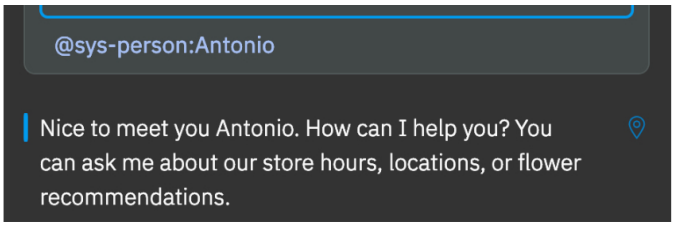
4. Click on the three dots icon in the response section and open the context editor. **Set the context variable $name to the value @sys-person**.

5. Next, we want to reply to the user from this node, so **add the following response, Nice to meet you, $name. How can I help you? You can ask me about our store hours, locations, or flower recommendations.**



Use the *Try it out* panel to **test out the interaction**, as shown below. (Click *Clear* to start a brand new conversation and see the new prompt.)





Nice! We are all set, right?

We have a problem

Well, not so fast. You might have spotted the problem if you tested it with your own name (depending on how common it is). We live in a beautiful and diverse world, and people have a variety of names. If you try it with the name, Reyansh it won't detect it. Heck, if you try antonio with a lowercase *a*, it won't detect the name either. I fully believe this latter limitation will be lifted in the future, but the former is much harder to address.

So we have something that sort of works, provided the name is common enough and properly capitalized. If that's not the case, the experience we provide is... poor at best, ending up with *I didn't understand* type of responses to the user's own name (the sweetest sound to their ears).

We have three possible approaches. All valid.

- Forget about collecting the name. We simply revert to the previous prompt, get rid of the child node, and that might be good enough for our chatbot.
- Collect the name in the node we created, if one is detected, as we currently have. Add a second node that simply doesn't set the context variable and replies with a generic *Hello*, without a name.
- Collect the name in the node we created, if the name is detected. Add a second node that collects what the user enters verbatim. In other words, we are collecting whatever answer the user provides and storing it exactly as stated.

We'll take this third route as it's the most refined of the three and it's the one that allows me to teach you the most concepts. The only downside to this is that the user might say, *I don't want to tell you* and we'll end up saying, *Nice to meet you I don't want to tell you*. Which is amusing, but an argument could be made that the user asked for it. 😀 Technically, we could implement even more sophisticated logic to detect such responses (with an intent) and not store the name if we get a flippant reply from the user, but it might be overkill or a refinement for a much later version of our chatbot.

Fixing the problem

As I mentioned, we'll take the third approach to fix this problem. If for no other reason than collecting the user's input exactly as provided is a useful notion you might need in future chatbots. The current *Collect Name* node works well for names detected by the system entity. However, we should **rename the node to something like Collect Sys-Person Name** to make it more descriptive.

Next, **add a peer/sibling node below it to handle the common case** where the user reply is not detected as a valid name. We'll want this second node to be executed every time the first child node (i.e., *Collect Sys-Person Name*) fails its condition, so **set the condition for this node to true** (it's essentially our fallback node in the *Welcome* tree). You can **call this node *Collect Other Names*** or something equally descriptive.

Watson stores the current user input in input.text. So open the context editor for this node and **set the $name context variable to <? input.text ?>**. The reason why we need the special syntax is that we don't want to say *Nice to meet you input.text.* but rather we are asking Watson to give us the actual value.

Doing so will collect the user input and assign it to the name. For the response, you can **use the same response as the node above**:

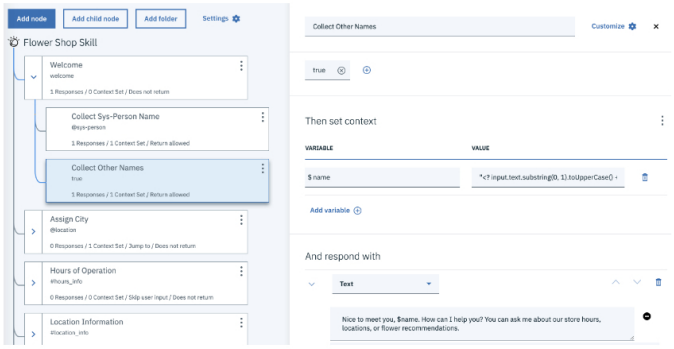
Nice to meet you, $name. How can I help you? You can ask me about our store hours, locations, or flower recommendations.

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 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. The image below shows what the node should look like.



Finally, test out a complete conversation with the chatbot:

(enter your name after the prompt)

What are you hours of operation of your Toronto store?

Where is it?

Thank you

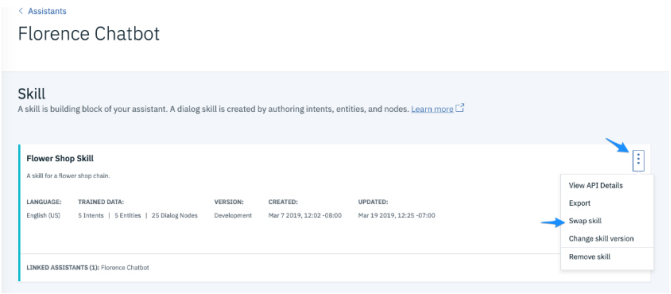
Goodbye

Pretty neat, right?

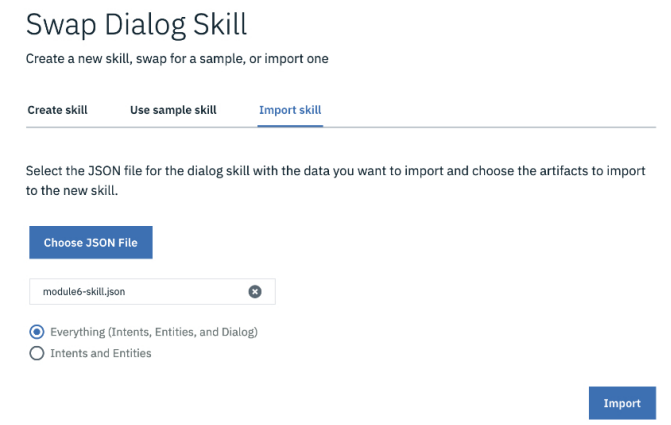
Help! It didn't work.

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. You can then click on *Assistants* in your Watson Assistant instance, then click on your assistant (e.g., Florence Chatbot). There you'll find your skill. Click on the three dots 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 create a new skill, use an existing 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*. You want to import everything as you need the dialog as well.



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.

Mark as completed