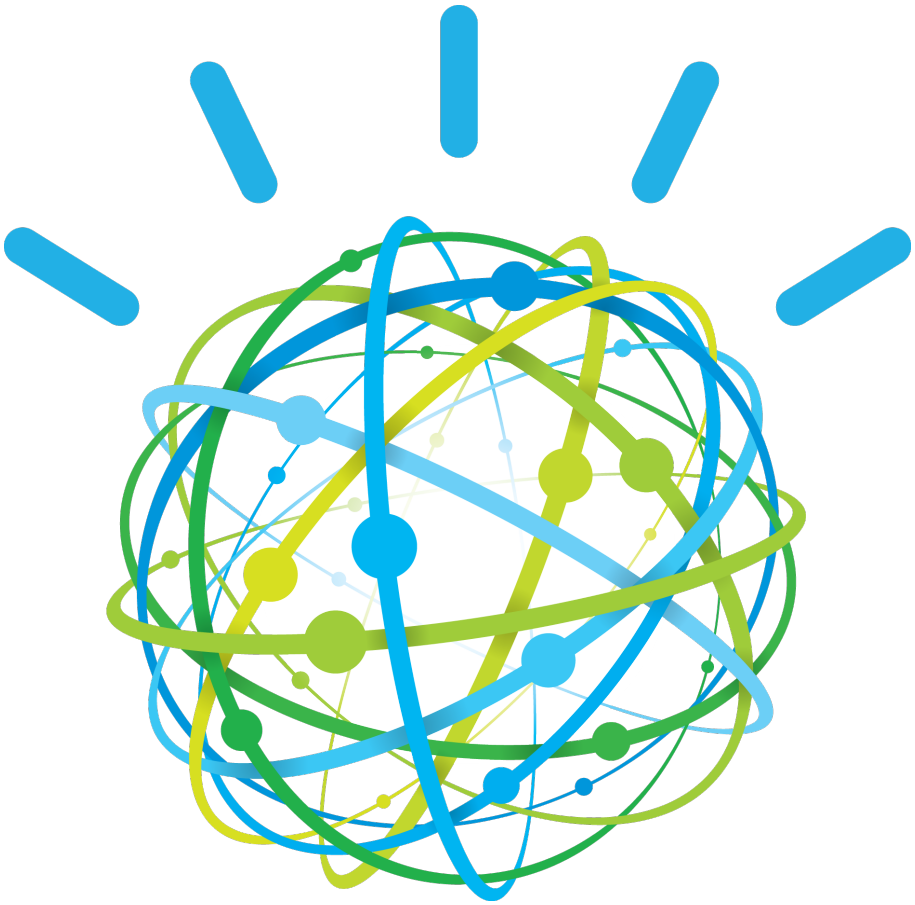
Lab 2 part 2: Building a Dialog

IBM Watson Assistant



Lab Instructions

Laurent Vincent

Content

Let's get started

1. Overview

The IBM Watson Developer Cloud (WDC) offers a variety of services for developing cognitive applications. Each Watson service provides a Representational State Transfer (REST) Application Programming Interface (API) for interacting with the service. Some services, such as the Speech to Text service, provide additional interfaces.

The Watson Assistant service combines several cognitive techniques to help you build and train a bot - defining intents and entities and crafting dialog to simulate conversation. The system can then be further refined with supplementary technologies to make the system more human-like or to give it a higher chance of returning the right answer. Watson Conversation allows you to deploy a range of bots via many channels, from simple, narrowly focused bots to much more sophisticated, full-blown virtual agents across mobile devices, messaging platforms like Slack, or even through a physical robot.

The **illustrating screenshots** provided in this lab guide could be slightly different from what you see in the Watson Assistant service interface that you are using. If there are colour or wording differences, it is because there have been updates to the service since the lab guide was created.

2. Objectives

Watson Assistant provides several options to manage Conditions, and possibility to have several answers to make your bot more human.

In this lab, you will:

- Gather information with slots
- Manage nodes and folders
- Manage digressions and disambiguations

3. Prerequisites

Before you start the exercises in this guide, you will need to complete the following prerequisite tasks:

- Lab 2 part 1 – building a dialog lab Instructions
- The instructor provided you the link to get labs content. You may download each file individually.

Reminder of IBM Cloud URLs per location:

Location	URL
worldwide	https://cloud.ibm.com/login

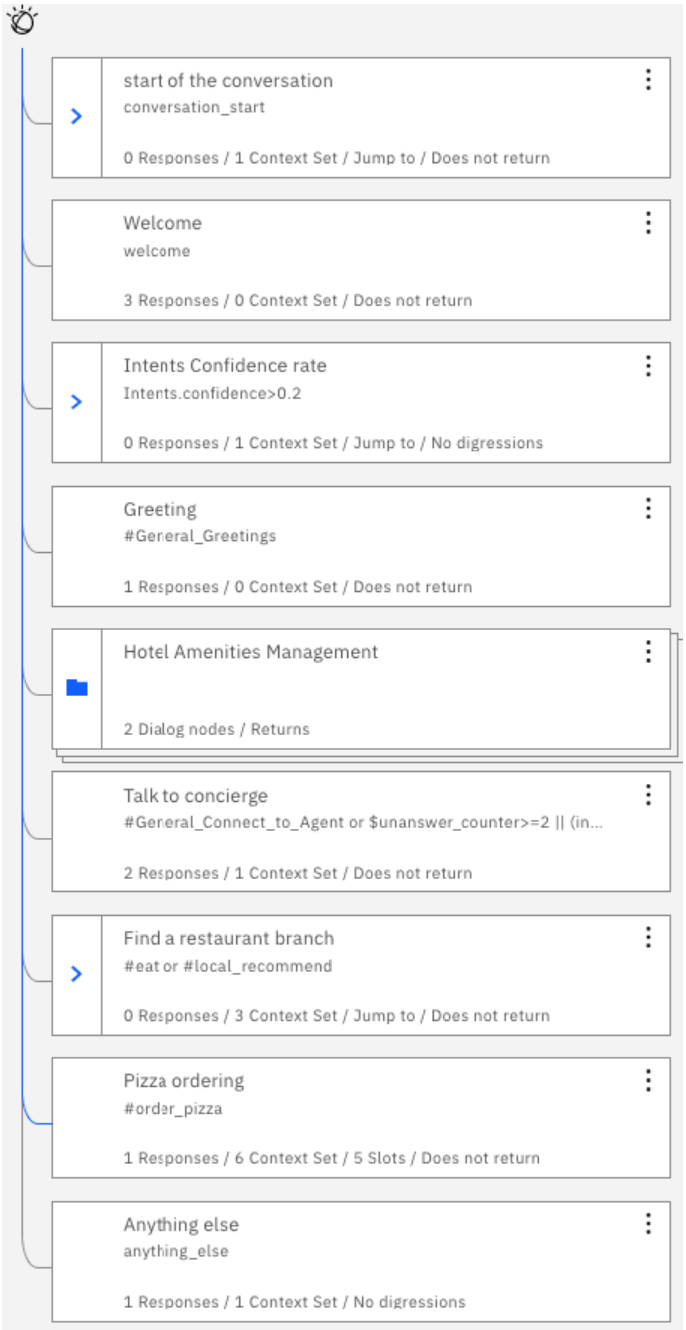
4. Scenario

Use case: A Hotel Concierge Virtual assistant that is accessed from the guest room and the hotel lobby.

End-users: Hotel customers

5. What to expect when you are done

At the end of session, you should get a more complex dialog using several conditions and answer in the same node.



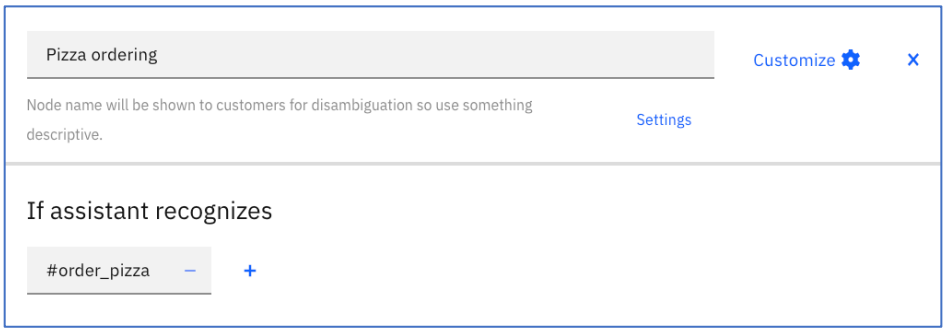
Gathering information with Slots

To organize better your dialog information, you have created a branch in previous lab. Now you will use Slots to simplify your dialog.
You can think of slots as the chatbot version of a web form in which users must fill out several required fields before they can submit the form. Similarly, slots prevent the flow of conversation from moving on to a new subject until the required values are provided
You are going to build a chatbot to order pizza. To do it, the chatbot must gather the size and the type of your pizza. We assume that your hotel can deliver such a service.

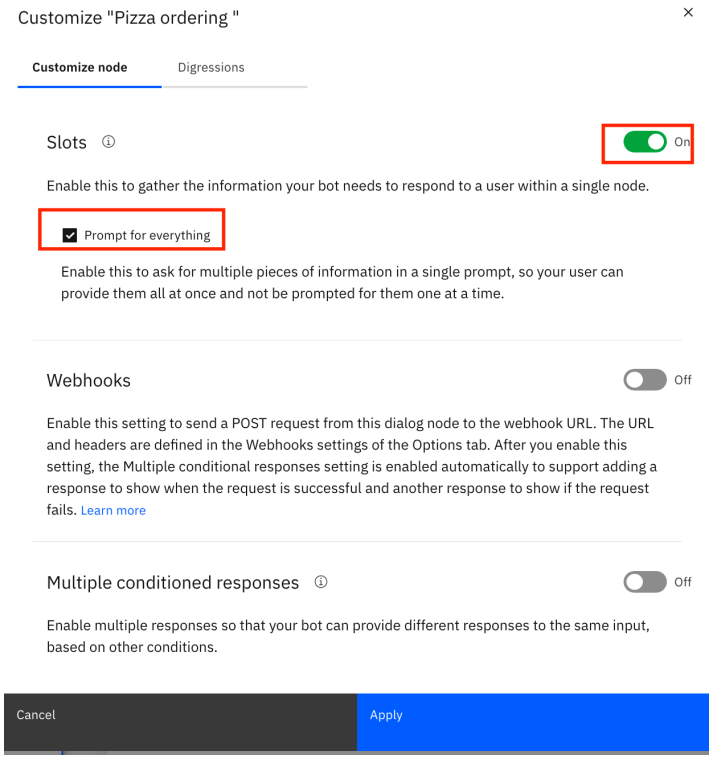
6. Add Pizza Ordering node and slots

The best should be to create a node to manage any orders, but we will simplify the lab and order only pizza which can be delivered in the guestroom.

1. Go back to **Dialog** page
2. Add a new node above the **Anything else** node of the main tree.
3. Set *#order_pizza* as condition and *Pizza ordering* as name
4. Click **Customize**



5. Switch on **Slots** on and enable **Prompt for everything**



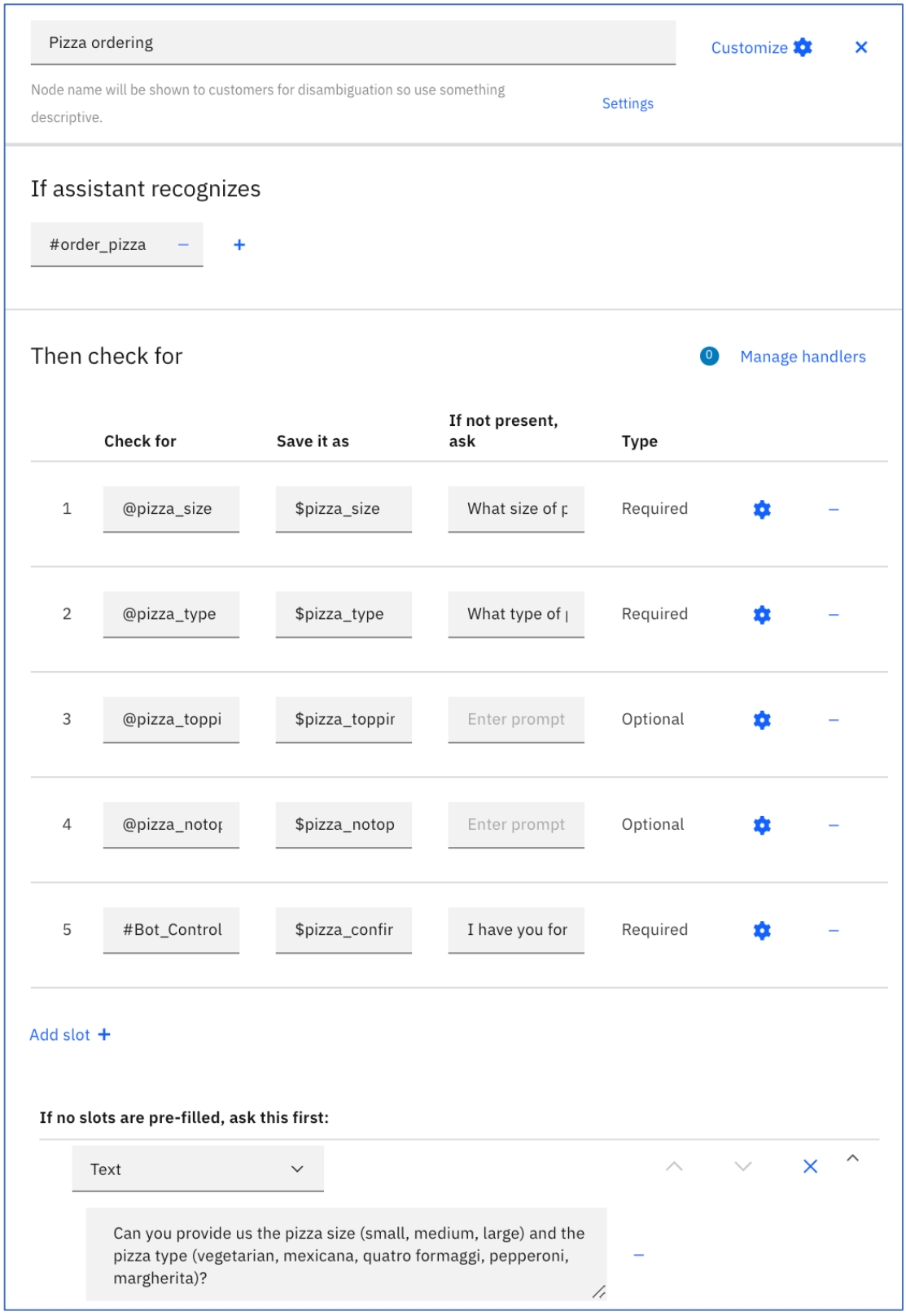
6. Click **Apply**
7. Click **Add slot +** four times.

8. Fill the slots like these

Check for	Save it as	If not present, ask
@pizza_size	*$pizza_size*	*What size of pizza do you want?*
@pizza_type	*$pizza_type*	*What type of pizza do you want?*
@pizza_toppings.values	*$pizza_toppings*	
@pizza_notoppings.values	*$pizza_notoppings*	
(#Bot_Control_Approve_Response || #Bot_Control_Reject_Response) && slot_in_focus	*$pizza_confirmed*	*I have you for $pizza_size $pizza_type $texttoppings. Is it correct?*

9. In the field **If no slots are pre-filled, ask this first** enter : *Can you provide us the pizza size (small, medium, large) and the pizza type (vegetarian, mexicana, quatro formaggi, pepperoni, margherita)?*

Now you should have:



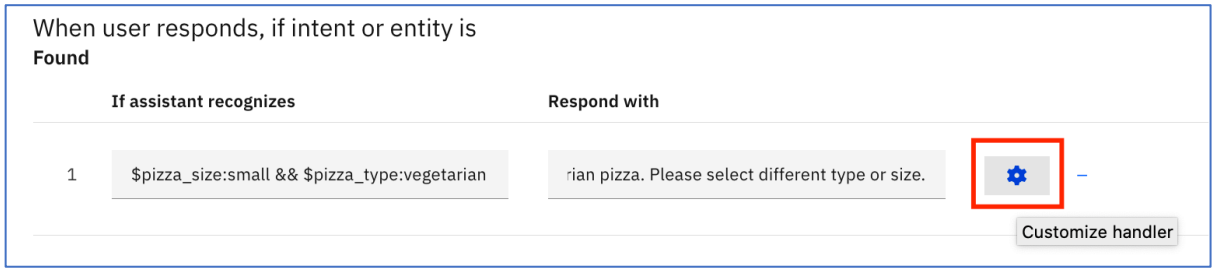
7. Manage the basic order information : pizza size

1. Click **Customize slot** icon (the wheel) of the **pizza_size** slot

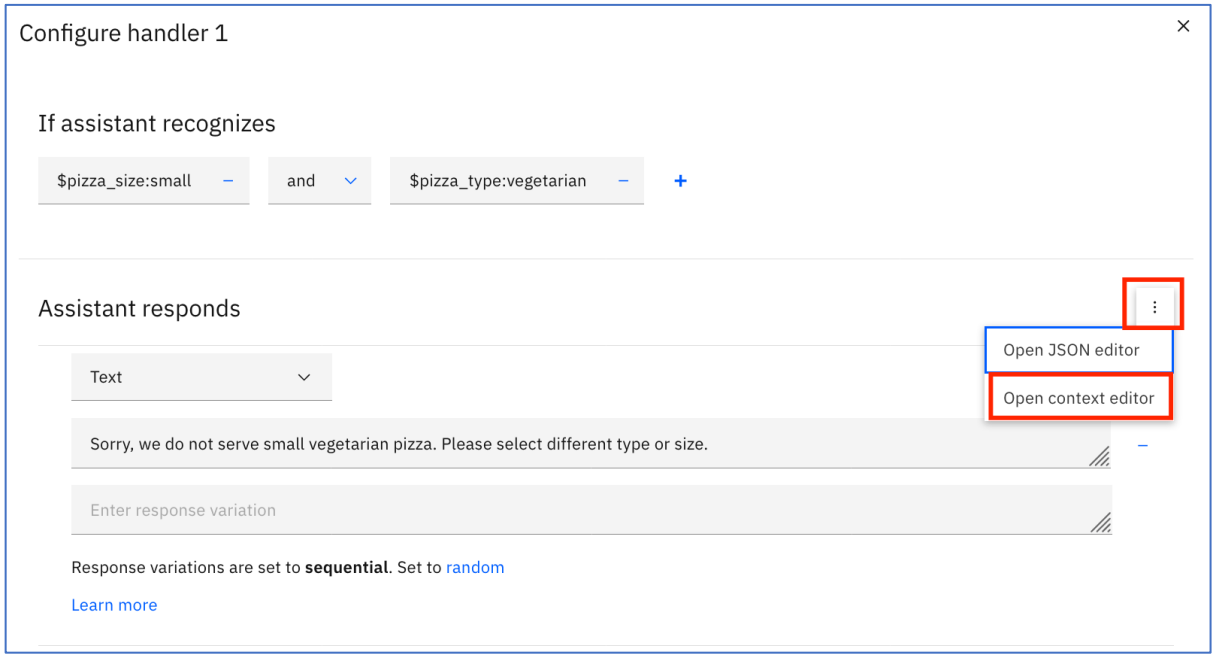
2. Scroll down and in **Found** frame, click **Add a response** on the first row,
3. Fill in the first 'Found' conditioned response like this

   If assistant recognizes: *$pizza_size:small && $pizza_type:vegetarian*

   Respond with: *Sorry, we do not serve small vegetarian pizza. Please select different type or size.*
4. On this row, click **Customize handler (the wheel)** icon



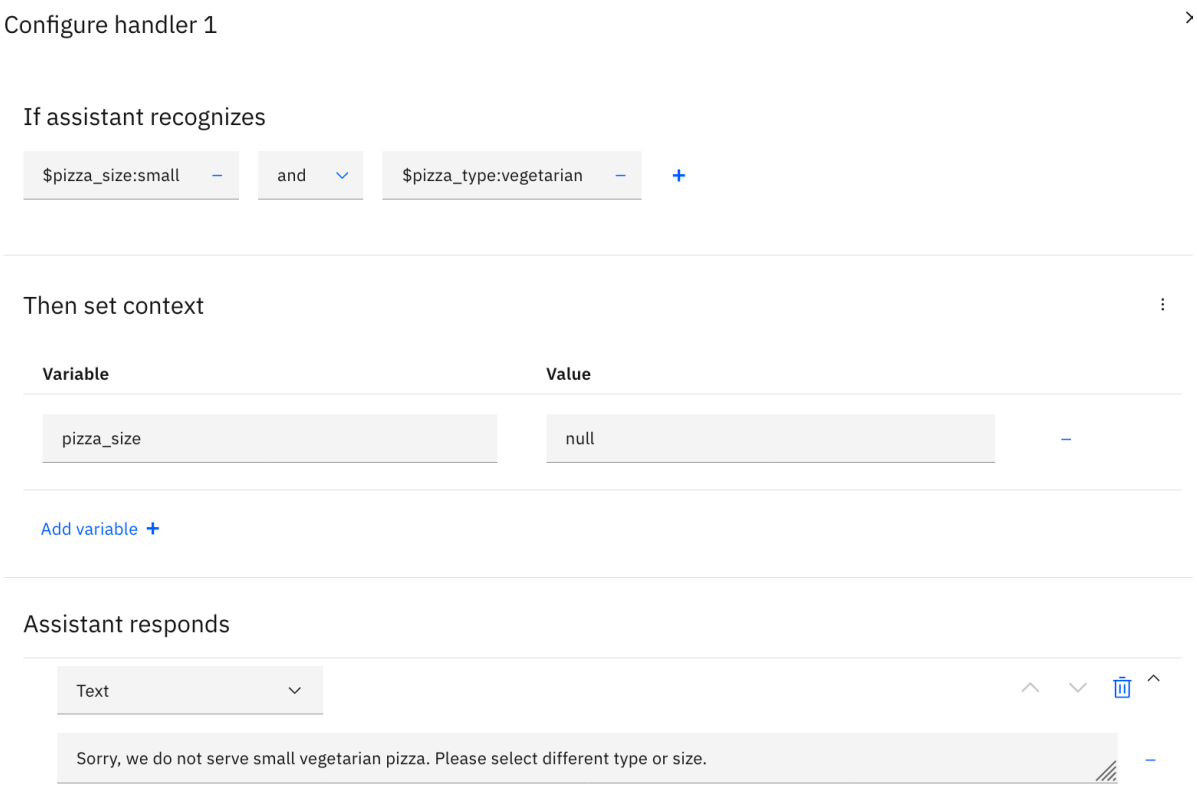
5. Click on the 3 dots menu, then open the context editor.



6. Fill the context variable like this

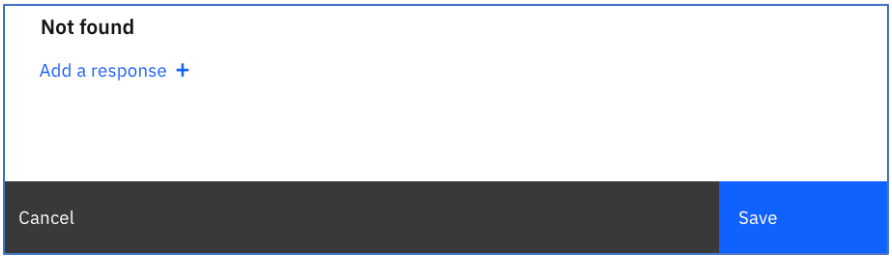
   Variable : *pizza_size*

   Value : *null*



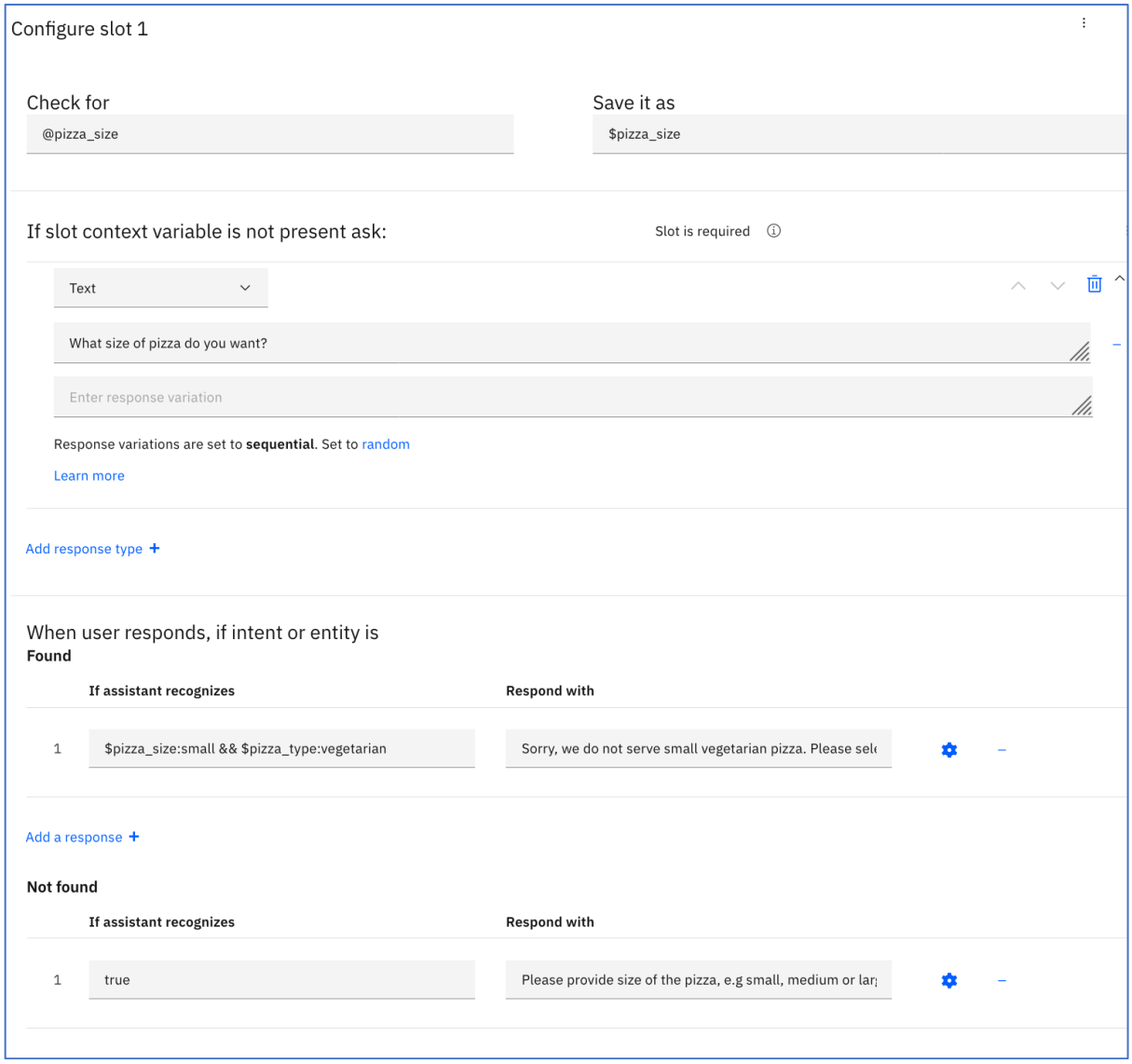
The simplest response example should be just to confirm the size of the pizza. Here we illustrate the capability to check the provided value according to some other context variables.

7. Click **Save**
8. In **Not found** frame at the bottom, click **Add a response**



9. Enter this response:
   condition: *true*
   response: *Please provide size of the pizza, e.g small, medium or large*.

Your first slot should look like this:



10. Click **Save**

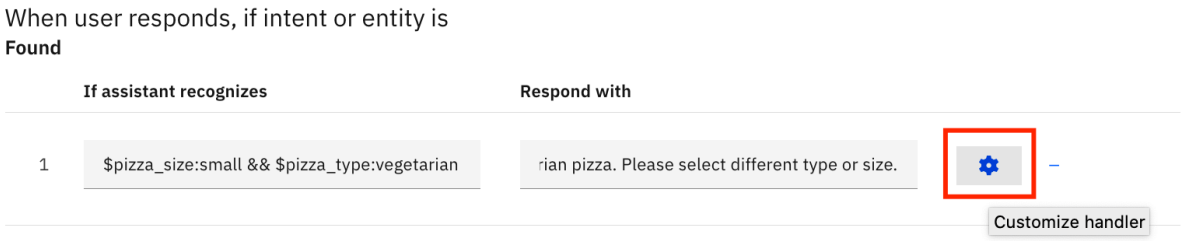
8. Manage the basic order information : pizza type

1. Click **Customize slot** icon (wheel) of the **pizza_type** slot

11. In the **Found** frame,click **Add a response**
12. Fill the first 'Found' conditioned response like this

    If assistant recognizes: *$pizza_size:small && $pizza_type:vegetarian*

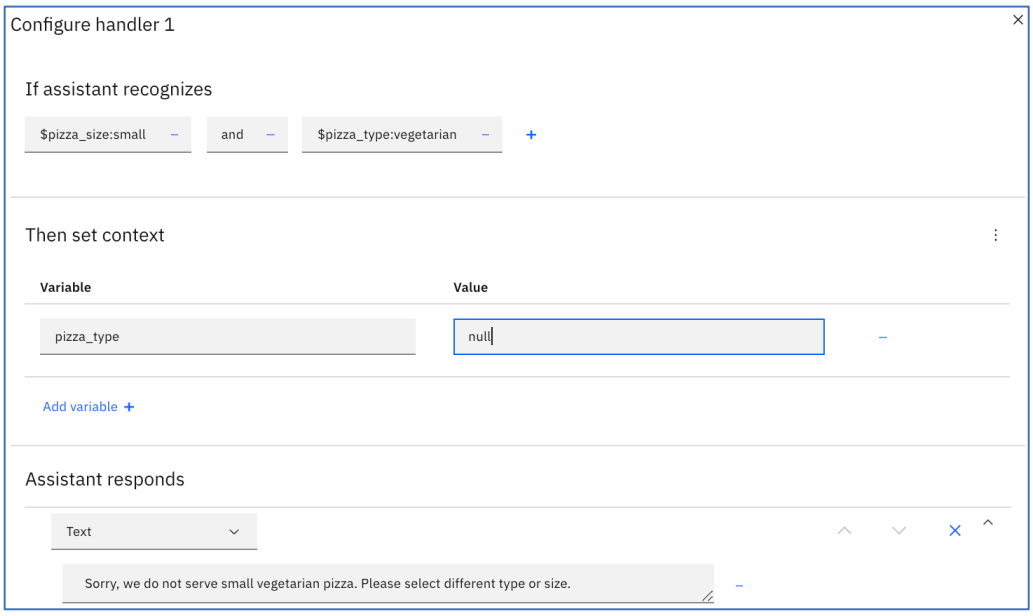
    Respond with: *Sorry, we do not serve small vegetarian pizza. Please select different type or size.*
13. On this row, click **Customize handler** (the wheel) icon



14. Click on the 3 dots menu, then open the context editor.
15. Fill the context variable like this

    Variable : *pizza_type*

    Value : *null*



16. Click **Save**

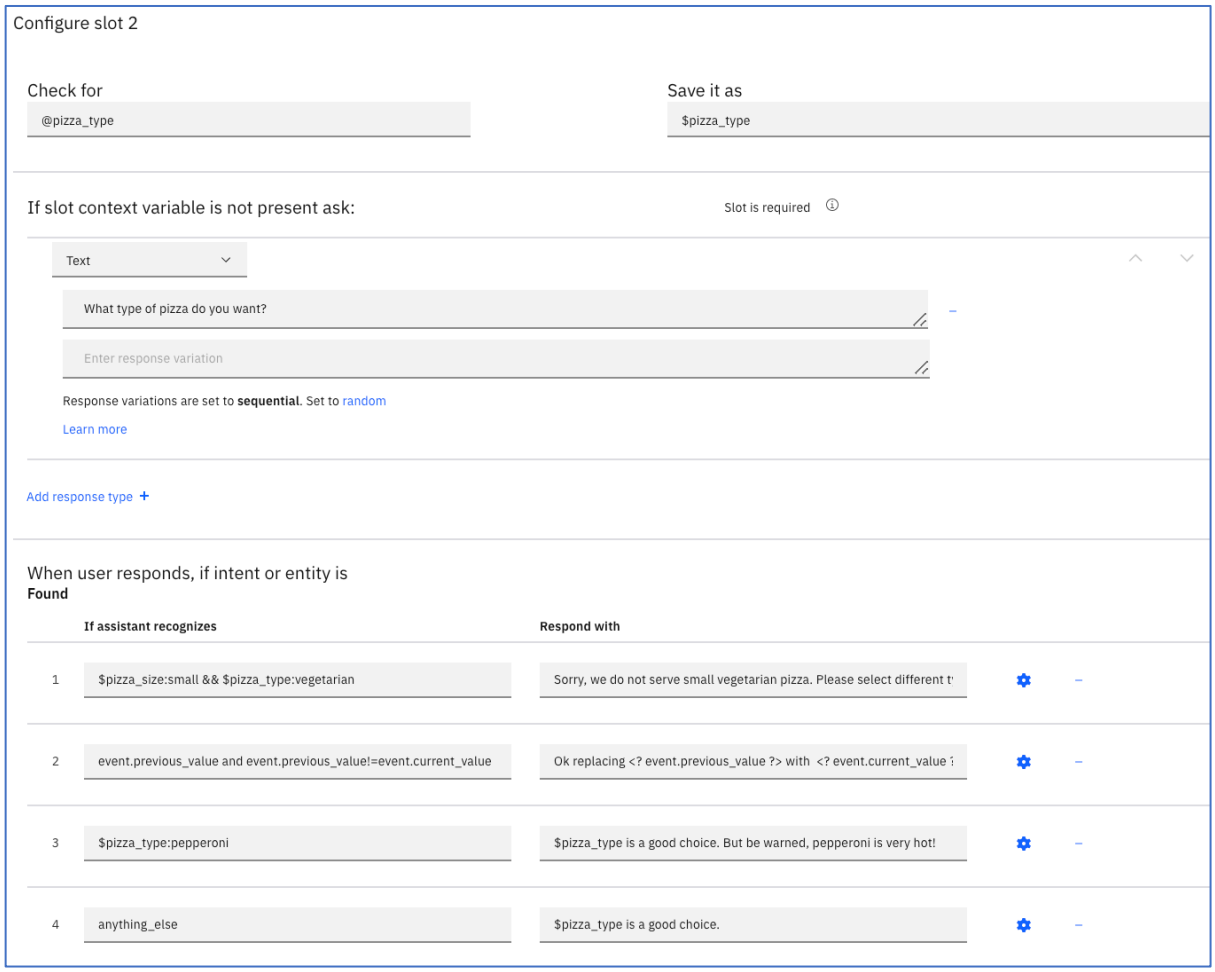
2. In the Found frame, add 3 more responses and condition like that:

   Resp2 condition: *event.previous_value and event.previous_value!=event.current_value*
   Resp2 response: *Ok replacing <? event.previous_value ?> with <? event.current_value ?>.*

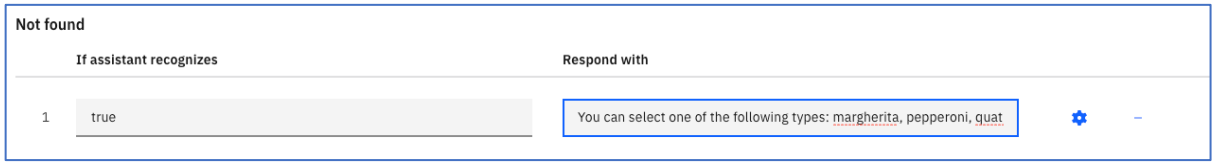
   Resp3 condition: *$pizza_type:pepperoni*
   Resp3 response: *$pizza_type is a good choice. But be warned, pepperoni is very hot!*

   Resp4 condition: *anything_else*
   Resp3 response: *$pizza_type is a good choice.*

This is the way to enrich the chatbot's responses and make them more human like.



3. In **Not found** frame, enter the response:
   Resp3 condition: *true*
   Resp3 response: *You can select one of the following types: margherita, pepperoni, quatro formaggi, mexicana, vegetarian.*



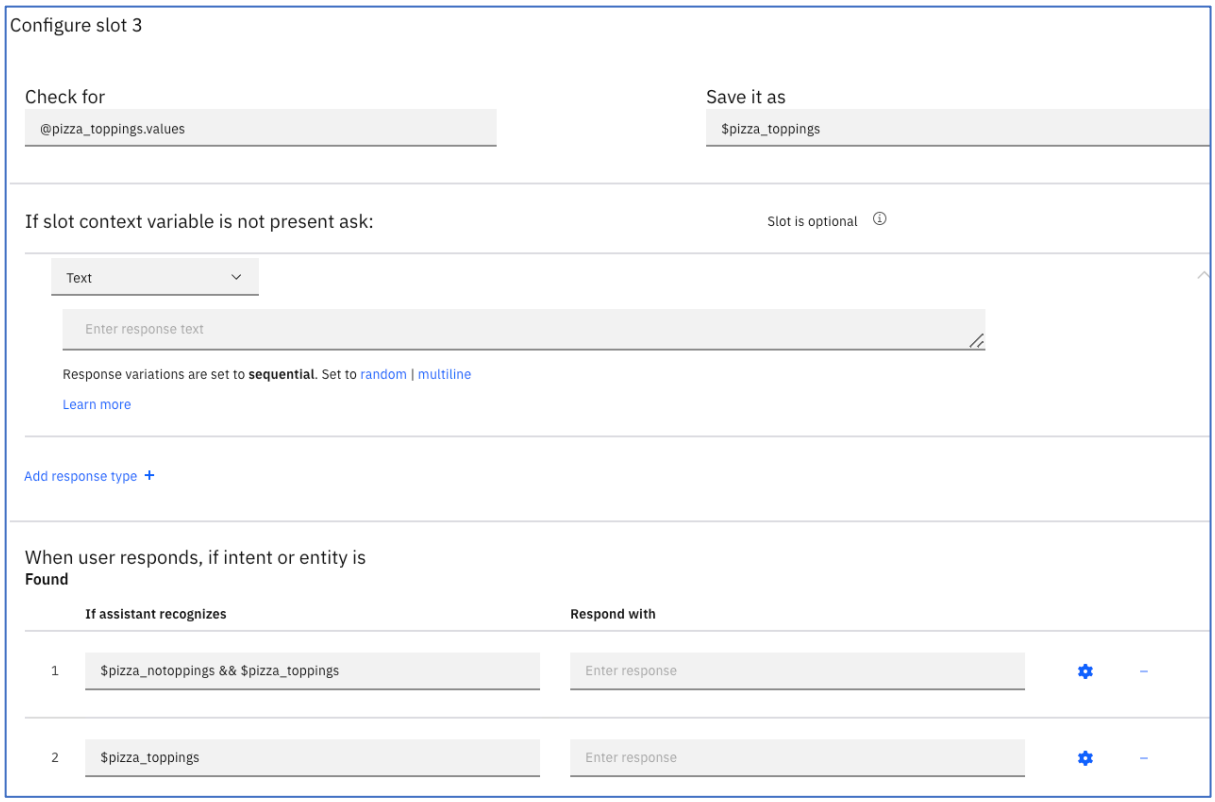
4. Click **Save**

9. Manage the toppings the user would like to add

1. Click **Customize slot** icon (wheel) of the **pizza_toppings** slot

2. In **Found** frame, add 2 responses and condition like this:

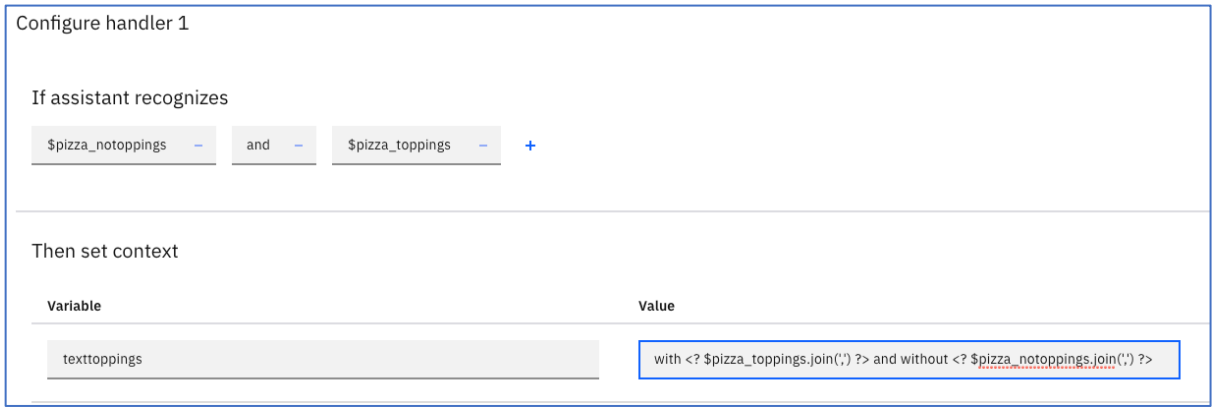
   Resp1 condition: *$pizza_notoppings && $pizza_toppings*
   Resp1 response:
   Resp2 condition: *$pizza_toppings*
   Resp2 response:.



3. In **Found** frame, first row, click **Customize handler** (the wheel) icon .

4. Click the 3 dots menu, then open the **Context editor** and fill it like this:

   Resp1 context variable: *texttoppings*
   Resp1 context value: *with <? $pizza_toppings.join(',') ?> and without <? $pizza_notoppings.join(',') ?>*



5. Click **Save**
6. In **Found** frame, second row, click **Customize handler** (the wheel) icon

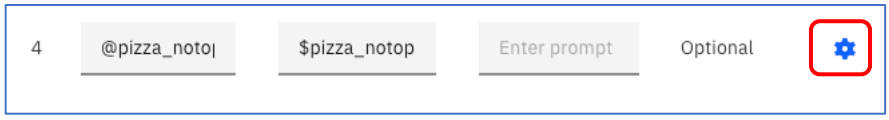
7. Click the 3 dots menu, then open the **Context editor** and fill it like this:

   Resp1 context variable: *texttoppings*
   Resp1 context value: *with <? $pizza_toppings.join(',') ?>*

8. Click **Save**
9. Click **Save** again

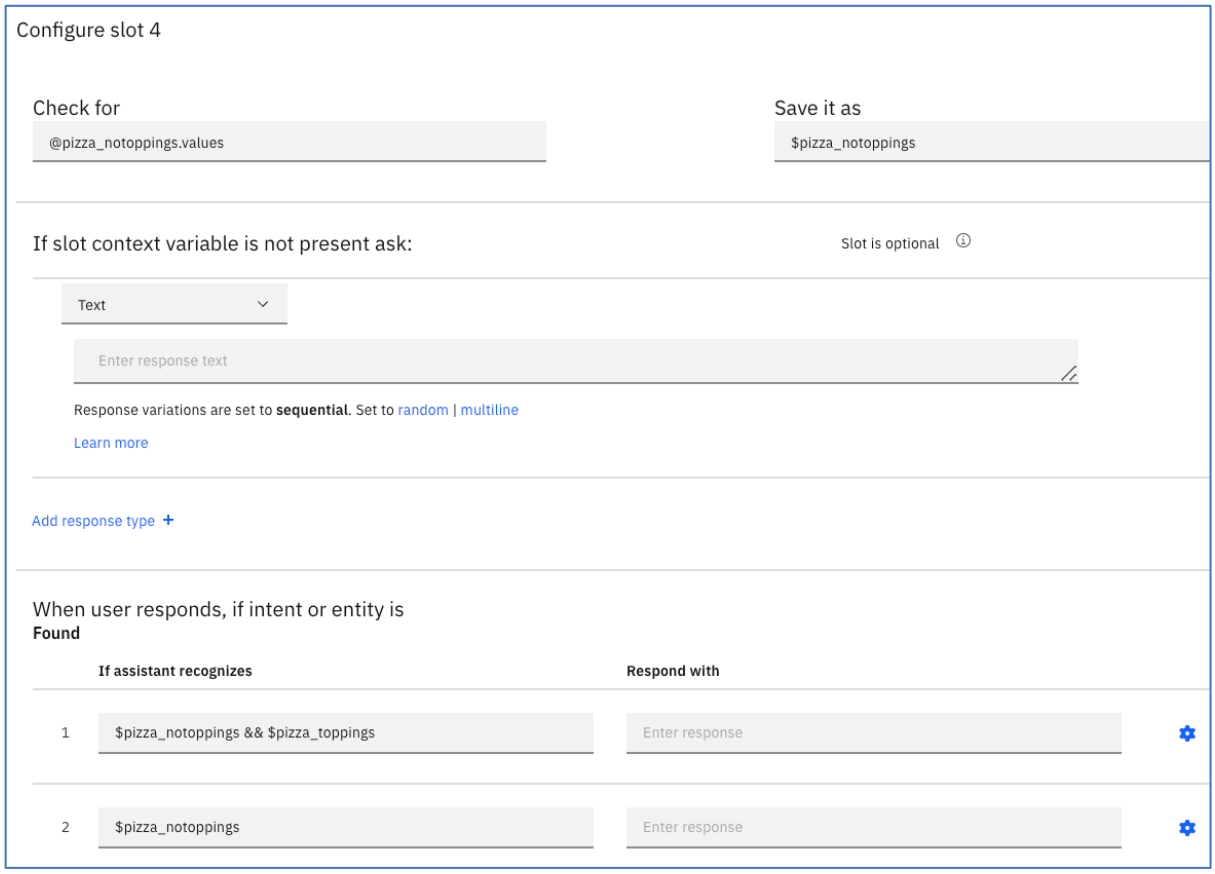
10. Manage the toppings the user would like to remove

1. Click **Customize slot** icon (wheel) of the **pizza_notoppings** slot



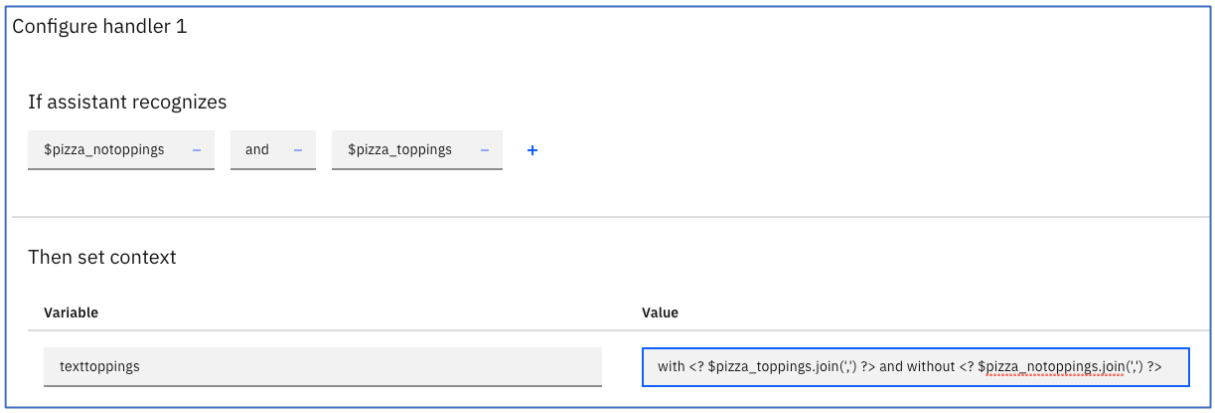
2. In **Found** frame, add 2 responses and condition like this:

   Resp1 condition: *$pizza_notoppings && $pizza_toppings*
   Resp1 response:
   Resp2 condition: *$pizza_notoppings*
   Resp2 response:.



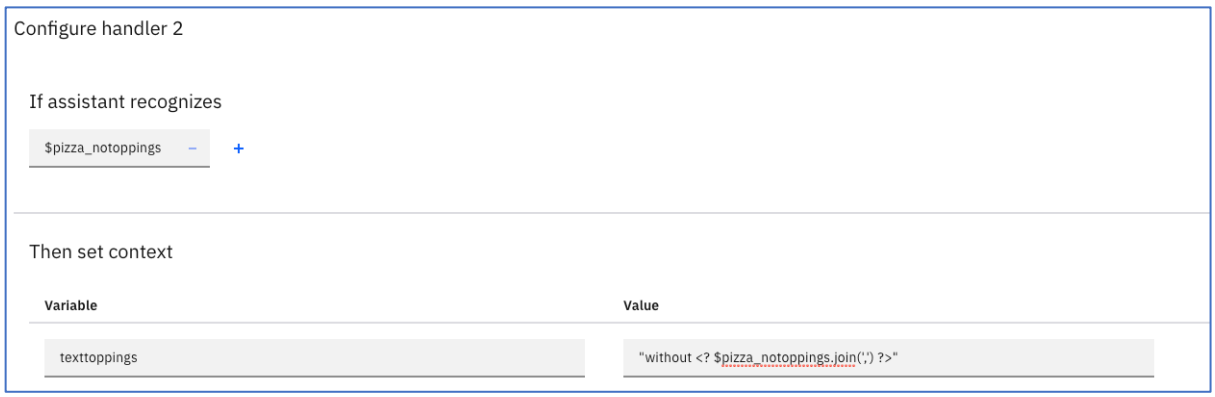
3. In **Found** frame, first row, click **Customize handler** (the wheel) icon, then open the **Context editor** and fill it like this:

   Resp1 context variable: *texttoppings*
   Resp1 context value: *with <? $pizza_toppings.join(',') ?> and without <? $pizza_notoppings.join(',') ?>*



4. Click **Save**
5. In **Found** frame second row, click **Customize handler** (the wheel) icon, then open the **Context editor** and fill it like this:

   Resp2 context variable: *texttoppings*
   Resp2 context value: *without <? $pizza_notoppings.join(',') ?>*



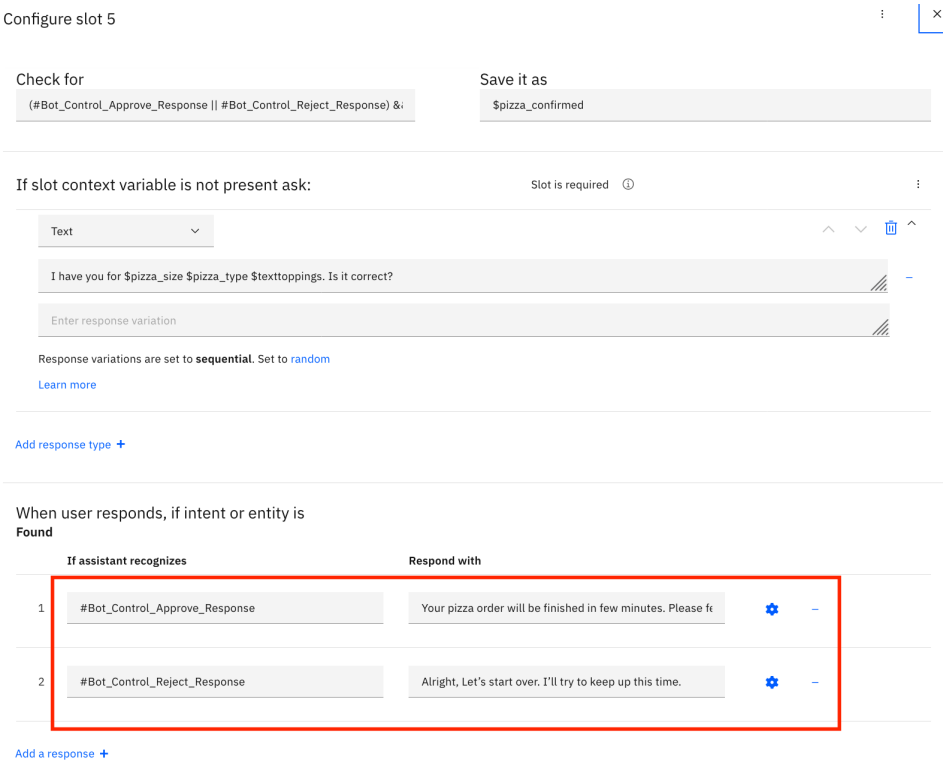
6. Click **Save**.
7. Click **Save** again.

11. Manage the confirmation

1. Click **Customize slot** icon (wheel) of the **pizza_confirmed** slot

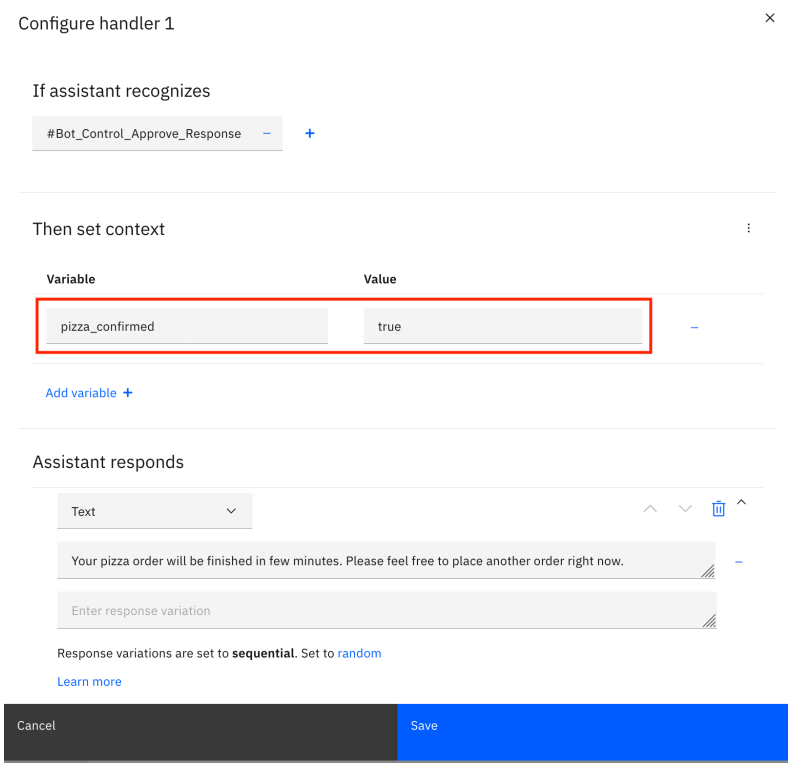
2. In the **Found** frame, add 2 responses and conditions like that:

   Resp1 condition: *#Bot_Control_Approve_Response*
   Resp1 response: *Your pizza order will be finished in few minutes. Please feel free to place another order right now.*
   Resp2 condition: *#Bot_Control_Reject_Response*
   Resp2 response: *Alright, Let's start over. I'll try to keep up this time.*



8. In **Found** frame, first row, click **Customize handler** (the wheel) icon, then open the **Context editor** and fill it like this:

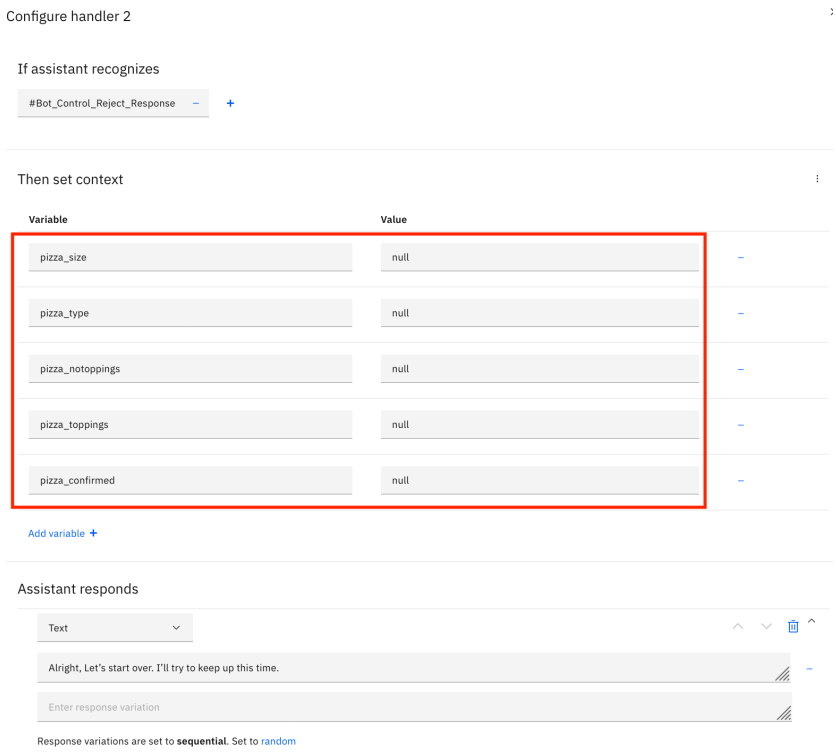
   Resp1 context variable: *pizza_confirmed*
   Resp1 context value: *true*



9. Click **Save**.

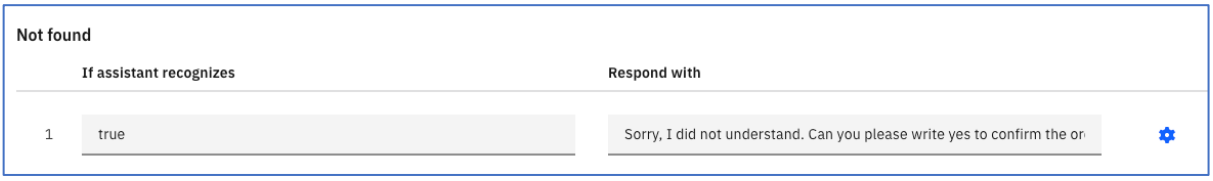
10. In **Found** frame, second row, click **Customize handler** (the wheel) icon, then open the **Context editor** and fill it like this:

    Context 1 Variable: *pizza_size*
    Context 1 Value: *null*
    Context 2 Variable: *pizza_type*
    Context 2 Value: *null*
    Context 3 Variable: *pizza_notoppings*
    Context 3 Value: *null*
    Context 4 Variable: *pizza_toppings*
    Context 4 Value: *null*
    Context 5 Variable: *pizza_confirmed*
    Context 5 Value: *null*



11. Click **Save**.

1. In the **Not found** frame, click **Add a response** then enter the respond:
   Resp1 condition: *true*
   Resp1 response: *Sorry, I did not understand. Can you please write yes to confirm the order or no to cancel the order all together? You can also yet change the type or size. Just say e.g. '"small Margherita.*

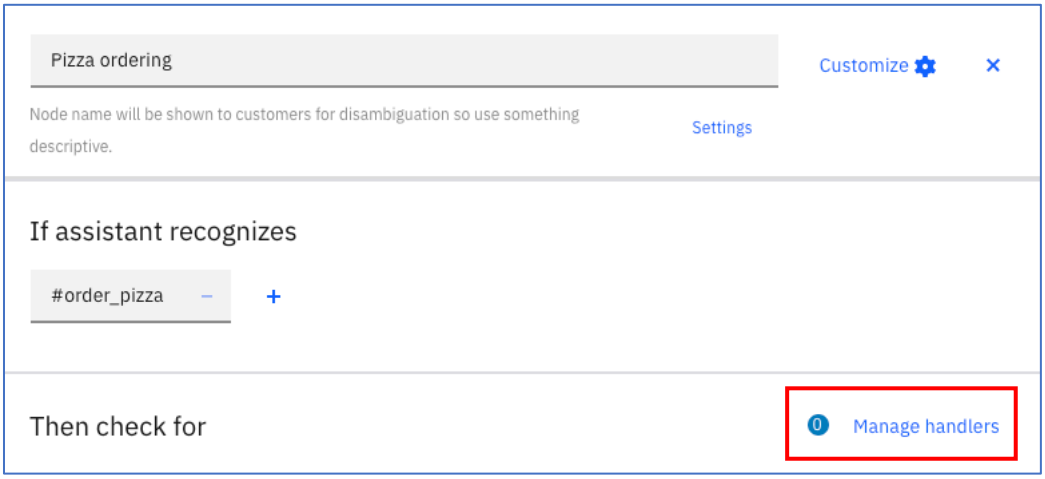


3. Click **Save.**

12. Manage Handlers

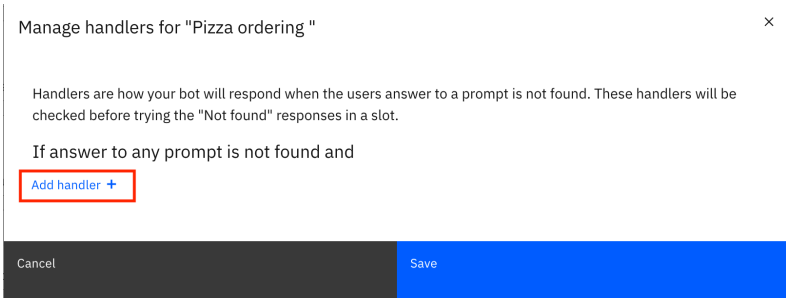
You can optionally define node-level handlers that provide responses to questions users might ask during the interaction that are tangential (ie. not essential) to the purpose of the node. Right now, the handlers enable users to leave the order or get some help.

1. Edit the **Pizza_ordering** node
2. Click **Manage handlers**



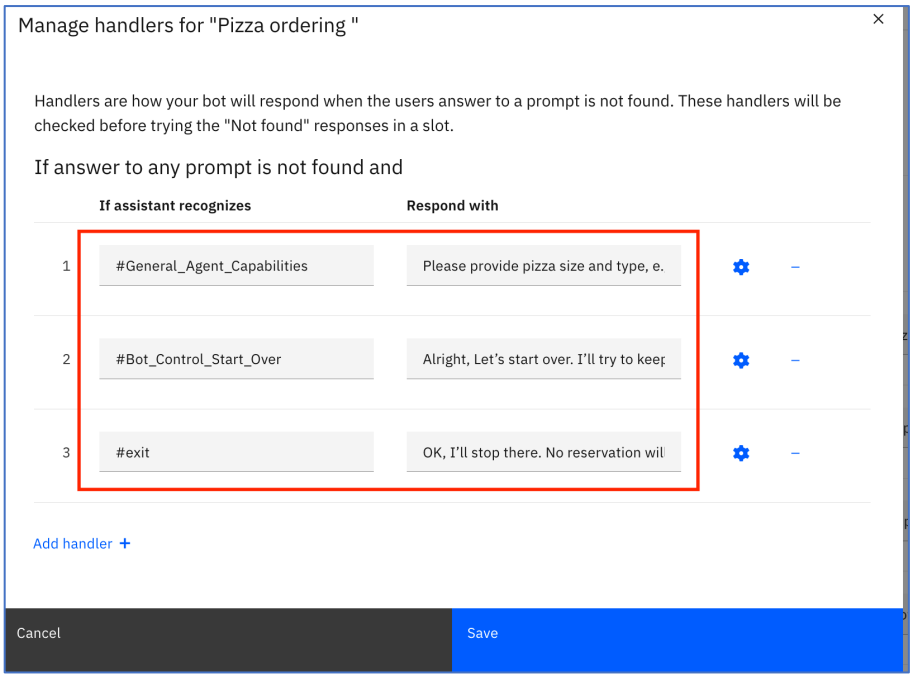
You are going to add 3 handlers.

3. Click 3 times **Add handler**



4. Fill the 3 handlers like below

   Handler1 condition: *#General_Agent_Capabilities*
   Handler1 response: *Please provide pizza size and type, e.g large margherita, small margherita.*
   Handler2 condition: *#Bot_Control_Start_Over*
   Handler2 response: *Alright, Let's start over. I'll try to keep up this time.*
   Handler3 condition: *#exit*
   Handler3 response: *OK, I'll stop there. No reservation will be made.*



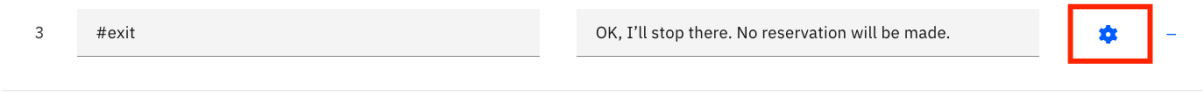
5. On the second row, click **Customize handler** icon (the wheel),

6. Open context editor and create the five context variables:

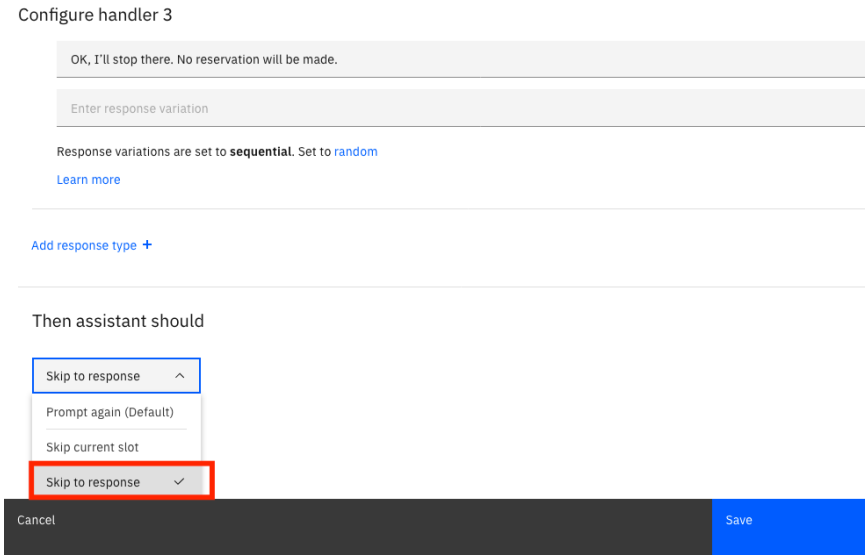
   Handler2 Context 1 Variable: *pizza_size*
   Handler2 Context 1 Value: *null*
   Handler2 Context 2 Variable: *pizza_type*
   Handler2 Context 2 Value: *null*
   Handler2 Context 3 Variable: *pizza_notoppings*
   Handler2 Context 3 Value: *null*
   Handler2 Context 4 Variable: *pizza_toppings*
   Handler2 Context 4 Value: *null*
   Handler2 Context 5 Variable: *pizza_confirmed*
   Handler2 Context 5 Value: *null*

7. Click **Save**

8. On the third row, click **Customize handler** icon (the wheel)



9. Scroll down, in the **then assistant should** frame, in the drop box select **Skip to response** option

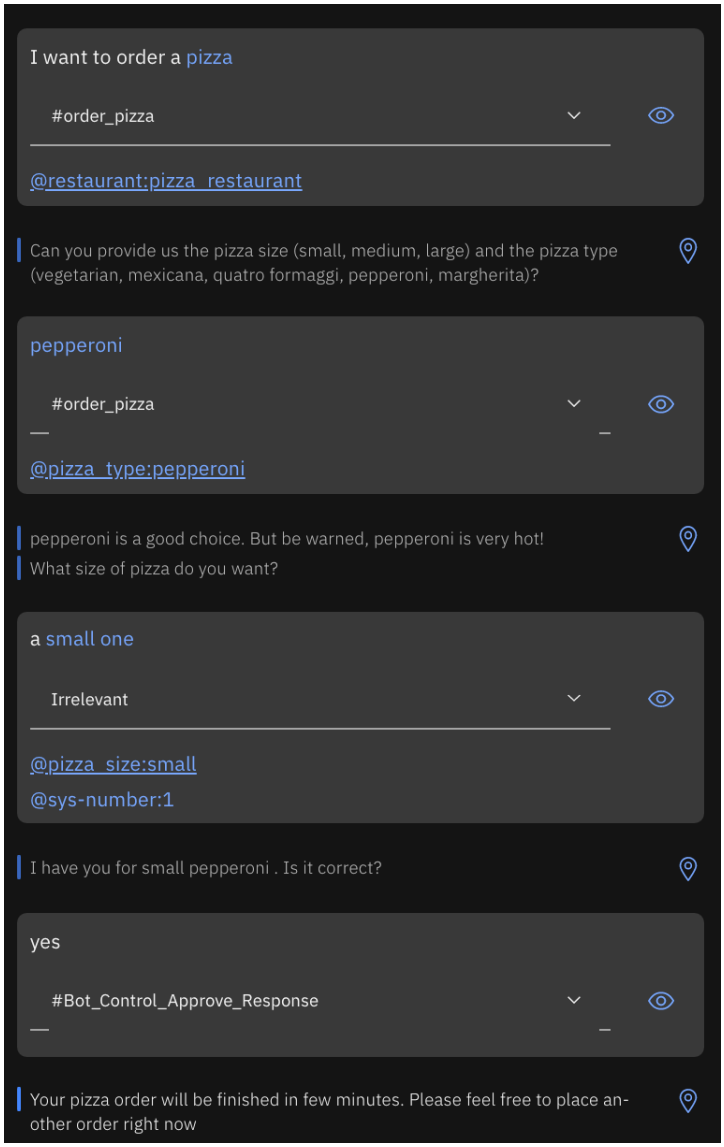


10. Click **Save**
11. Click **Save** again

13. Test your Slots

1. Open **Try it out** panel and Enter successively:

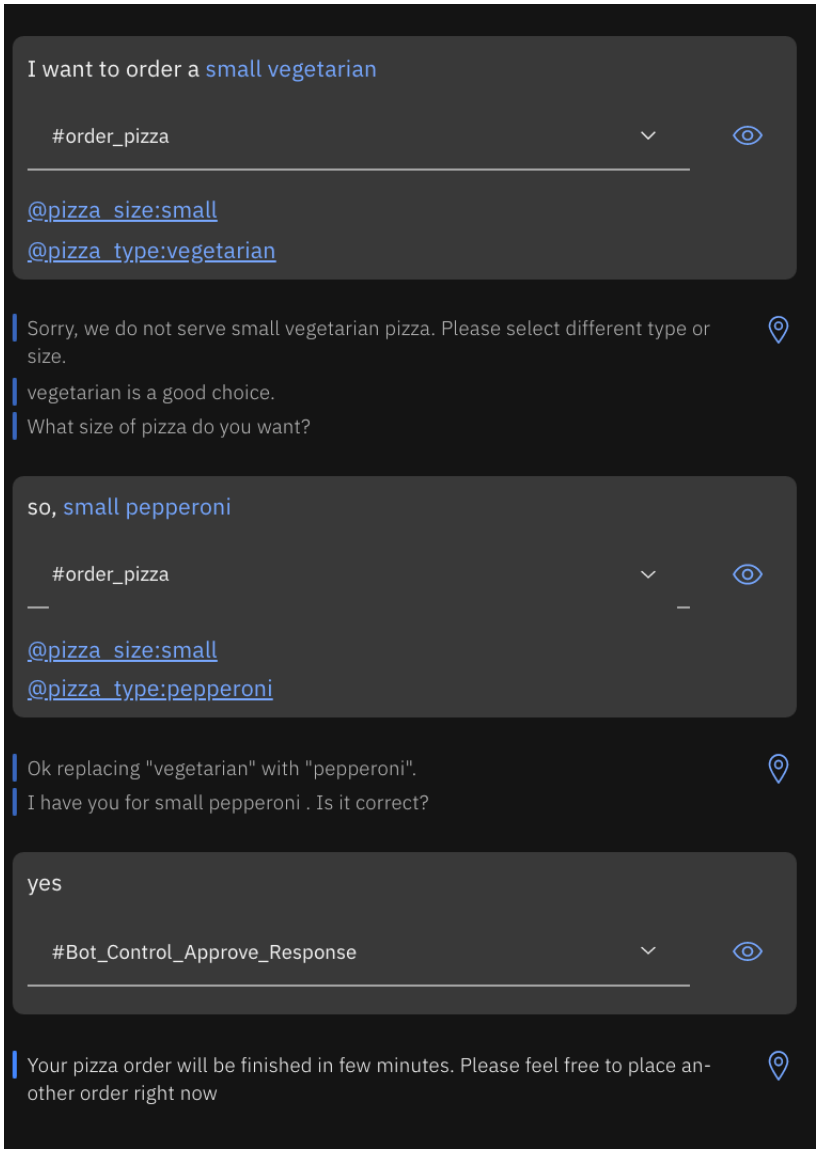
   I want to order a pizza
   pepperoni
   A small one
   Yes



2. Click **Clear**

3. Enter successively

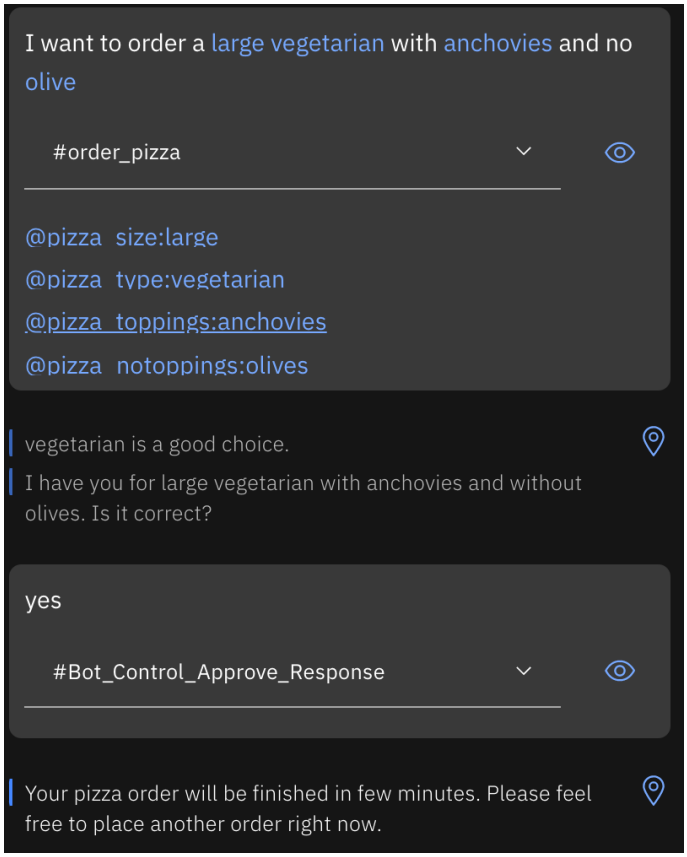
   I want to order a small vegetarian
   so, small pepperoni
   Yes



4. Click **Clear**

5. Enter successively

   I want to order a large vegetarian with anchovies and no olive
   Yes

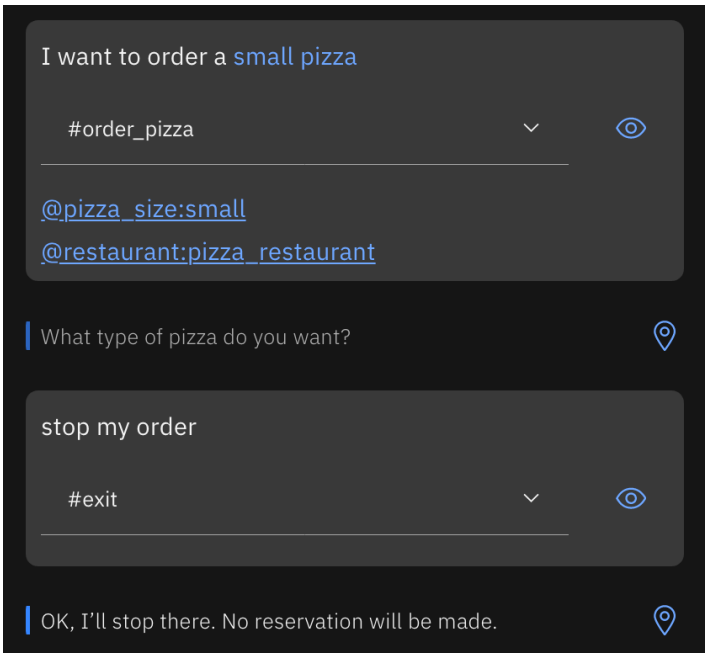


6. Click **Clear**

7. Enter successively

   I want to order a small pizza

   stop my order



You can run some other tests and order a pizza by using your Preview link – the external URL to your chatbot.

Managing nodes and folders

We can group dialog nodes together by adding them to a folder. Benefits of using a folder are:

- It allows a dialog designer to organize content based on topics
- Much easier dialog tree navigation and understanding
- It allows performing of bulk setting of node settings at the folder level instead of one by one node
- It is an easier separation of duties for multiple people working on the same bot

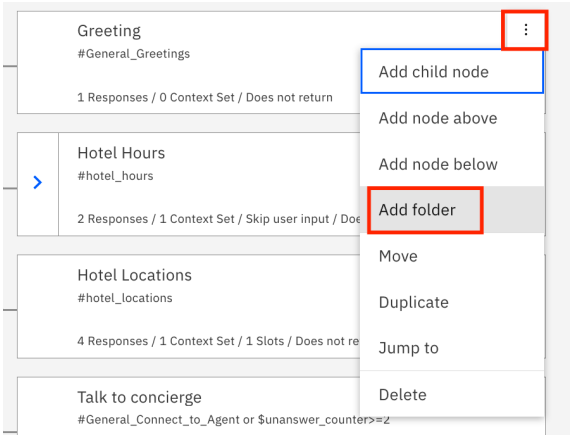
Folders have no impact on the order in which nodes are evaluated. But if a condition is specified, WA first evaluates the folder conditions to determine whether to process the nodes within it.

The nodes inherit of the digression settings of the folder.

14. Add a folder

The best should be to create a node to manage any orders,but we will simplify the lab and order only pizza which can be delivered in the guestroom.

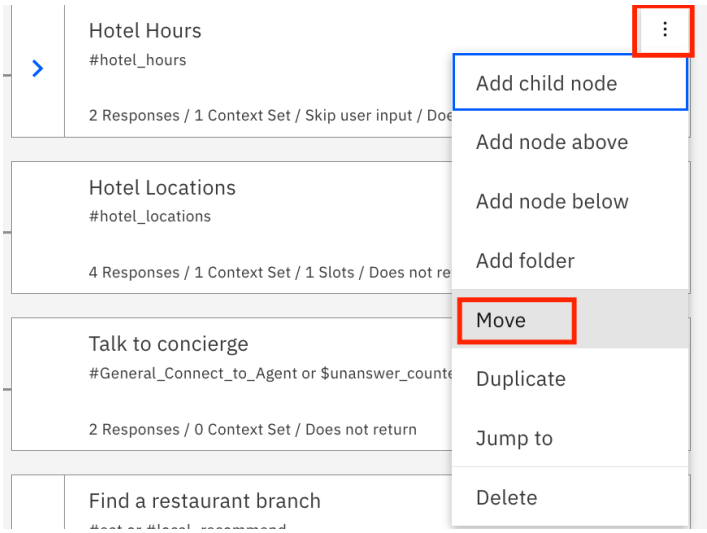
1. On the dialog tab, select the **Greeting** node, open its menu and click **Add folder.**



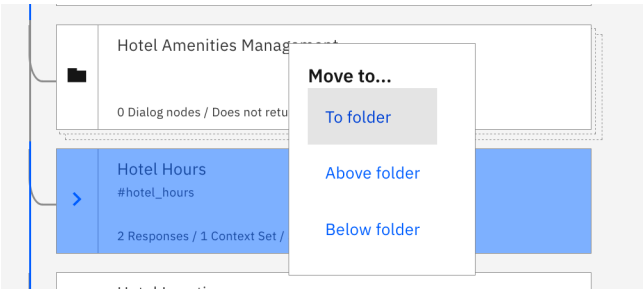
2. Name it *Hotel Amenities Management*

   We don't apply neither condition nor settings on this folder, as we just want to organise our dialog.

3. Open the menu of **Hotel Hours** node, and click **Move**.

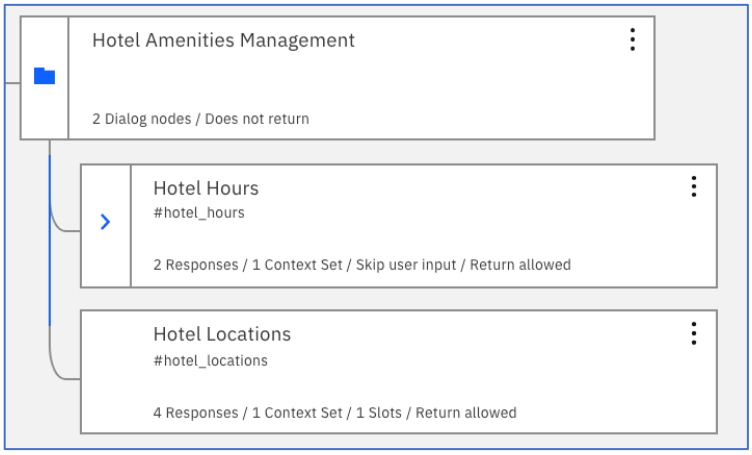


4. Select the destination location, the folder **Hotel Amenities Management**



5. Then click **To folder**.
6. Repeat the steps 3 to 5 to move **Hotel Locations** node to the folder **Hotel Amenities Management**.

   You will have this:



   If we make some tests, the behaviour of our conversation stays the same.

Understanding digressions

Digressions allow for users to break away from a dialog branch in order to temporarily change the topic before returning to the original dialog flow.
In this section, you will start to order a pizza, then digress away to ask for the restaurant's hours. After providing the opening hours information, WA will return back to the pizza ordering dialog flow.

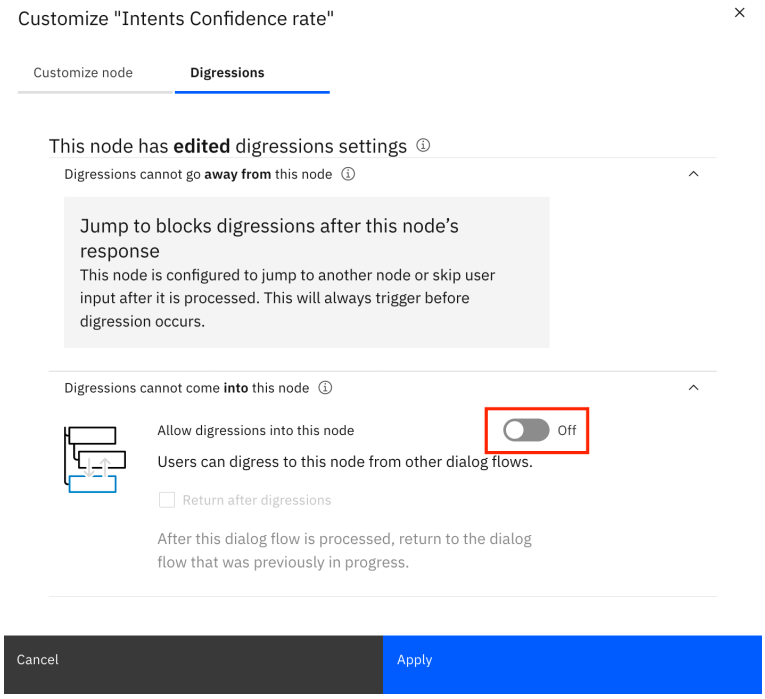
15. Configure your digressions

We are going to enable 5 nodes and 1 folder for digressions.

We don't want **Intents Confidence rate** and **Anything else,** our technical nodes, to be a target of any digression.

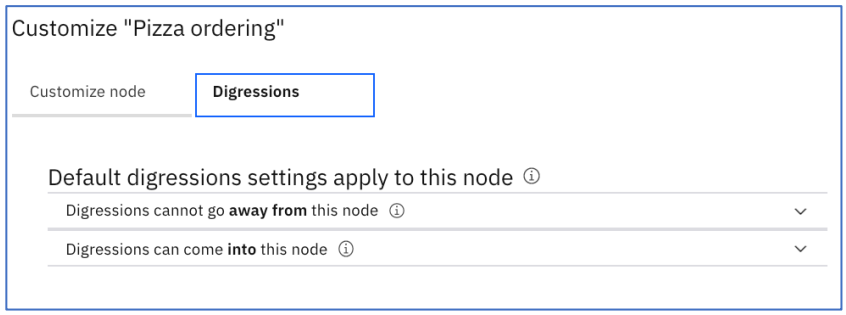
1. Select **Intents Confidence rate** node.
2. Click **Customize**, then go to **Digressions** tab

   As this is a technical node, we don't want any digression from or to this node. We switch off the option come into this
3. Turn off the second option



4. Click **Apply**
5. Repeat the previous steps to disable the digression for the **Anything else** node.

6. Select **Pizza Ordering** node
7. Click on **Customize** button then go to the **Digression** tab

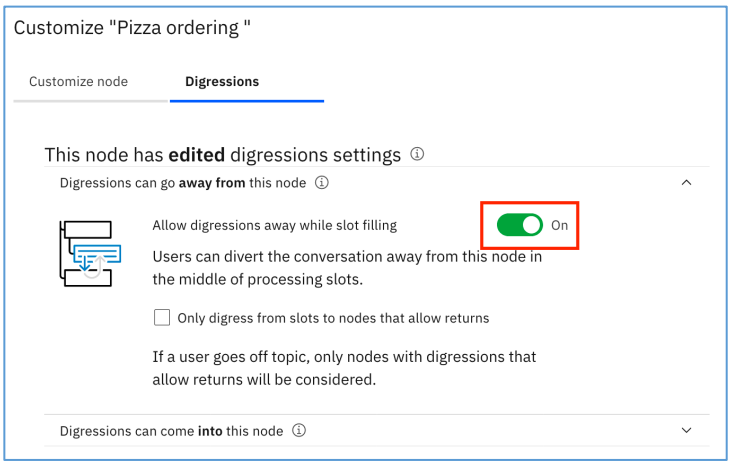


These are the Digressions default settings:

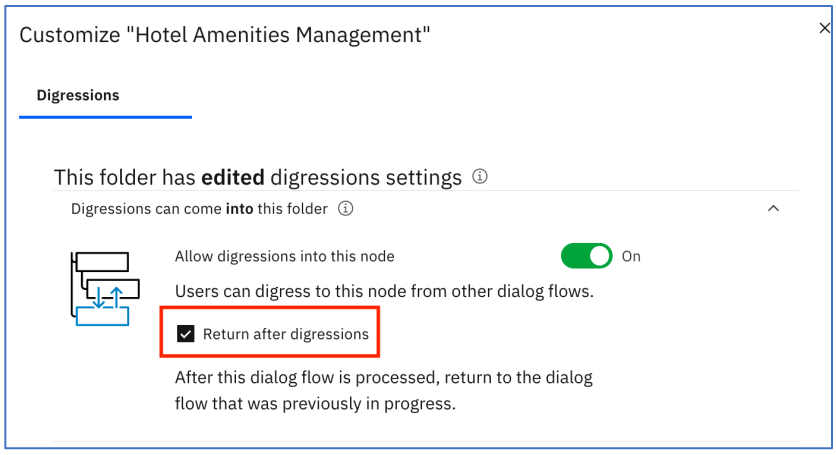
- Digression cannot go away from this node
- Digression can come into this node

We want to enable Digressions to go away from this node and come into this node.

8. Enable the option **go away from** this node,and don't update the second option.



9. Click **Apply.**
10. Select **Hotel Amenities management** folder.
11. Click on **Customize** button.
12. Select the option **Return after digression**



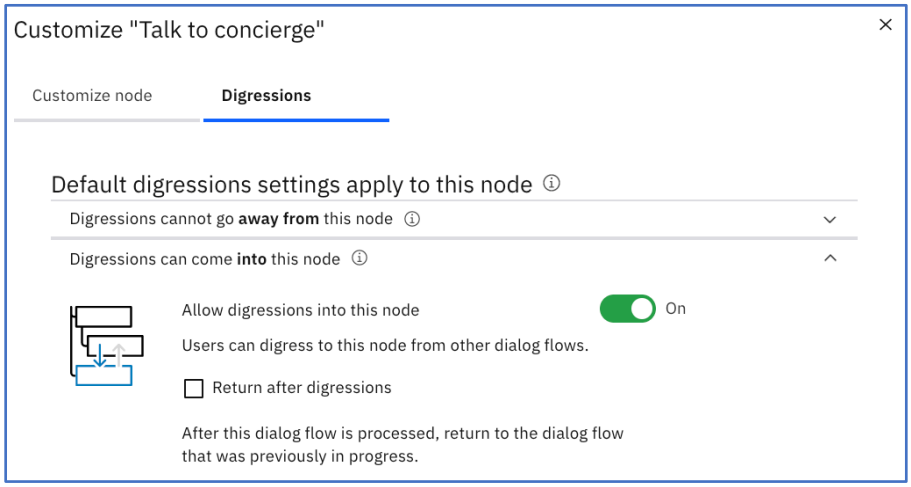
13. Click **Apply**

    The settings will be applied to all nodes into the folder: **Hotel Locations** and **Hotel Hours**

Now, you are going to update **Talk to concierge** node which requires to not return after digression.

14. Select **Talk to concierge** node.
15. Click **Customize**, then go to **Digressions** tab

    We keep the default digression behaviour as we want to be able to come into this node without return after digression



16. Click **Cancel** to close the window

16. Digression navigation tips.

1. How to know if we're in a digression?

You can use the context variable $system.digressed. This variable is set to true while a digression. It can be used as a condition if **return after digression** is check in the **Digression Settings** of your node; if not WA clear the whole dialog stack to prevent any digression returns. So *$system.digressed* is null in this case.

2. How to know if we're return from a digression?

There also *returning_from_digression* Spel condition, which we can use to customize how people experience dialog flow within digressions. The text message response can be set like this :

<? (returning_from_digression)? "post-digression message" : "first-time message" ?>
e.g.
<? (returning_from_digression)? "So, where were we... ah, what color shirt did you want again?" : "What color shirt would you like?" ?>

3. How to prevent digression return?

In some cases, you might want to prevent a return to the interrupted conversation flow based on a choice the user makes in the current dialog flow. You can use special syntax to prevent a return from a specific node.
The call to the function *<? clearDialogStack() ?>* will clear the whole dialog stack hence preventing any digression returns to occur

You can use it like these e.g.

in context variable *"MyVariable":"<? clearDialogStack() ?>"*

and it sets MyVariable to null as the result of that function.

Or in the response: *Pre cancellation message <? clearDialogStack() ?>*

For example, you might have a node that conditions on *#General_Connect_To_Agent* or a similar intent. When triggered, if you want to get the user's confirmation before you transfer them to an external service, you might add a response such as, *Do you want me to transfer you to an agent now?* You could then add two child nodes that condition on *#yes* and *#no* respectively.

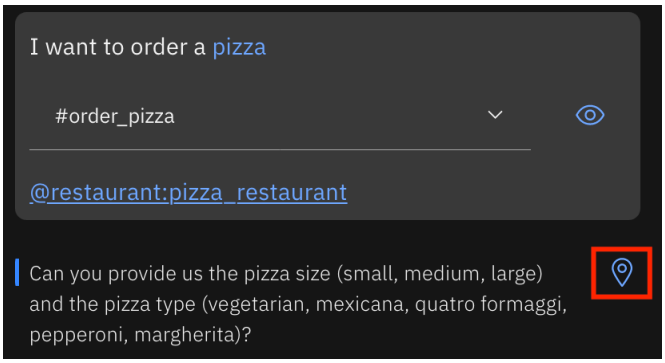
The best way to manage digressions for this type of branch is to set the root node to allow digression returns. However, on the #yes node, include the SpEL expression *<? clearDialogStack() ?>* in the response. For example:

OK. I will transfer you now. <? clearDialogStack() ?>

This SpEL expression prevents the digression return from happening from this node. When a confirmation is requested, if the user says yes, the proper response is displayed, and the dialog flow that was interrupted is not resumed. If the user says no, then the user is returned to the flow that was interrupted.

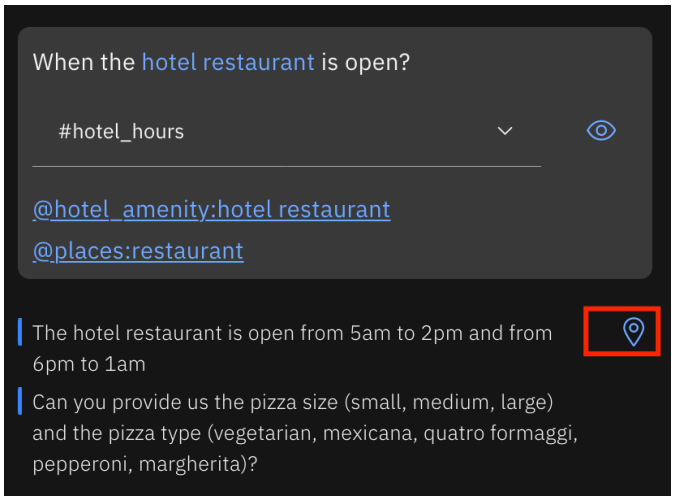
17. Test your digressions

1. Open the **Try is out** panel and click **Clear**
2. Enter : *I want to order a pizza*
3. Click on the location icon (right from the answer). When we click on this icon, we can see in which node we are currently positioned.



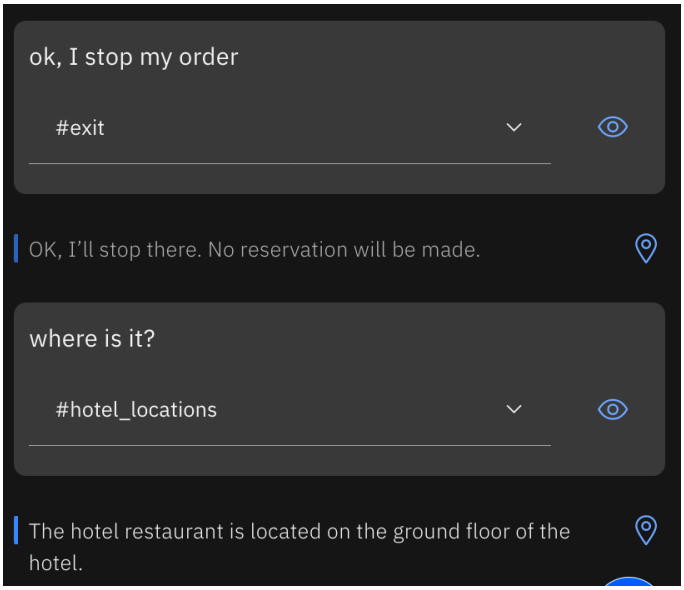
On the left, the Pizza ordering node is highlighted, which was expected.

4. Enter : *When the hotel restaurant is open?*



The bot digresses away from the **Pizza ordering** node to process the **Hotel Hours** node. (You can find it out again by clicking on the location icon) The service then returns to the **Pizza Ordering** node, and prompts you again for the size of pizza.

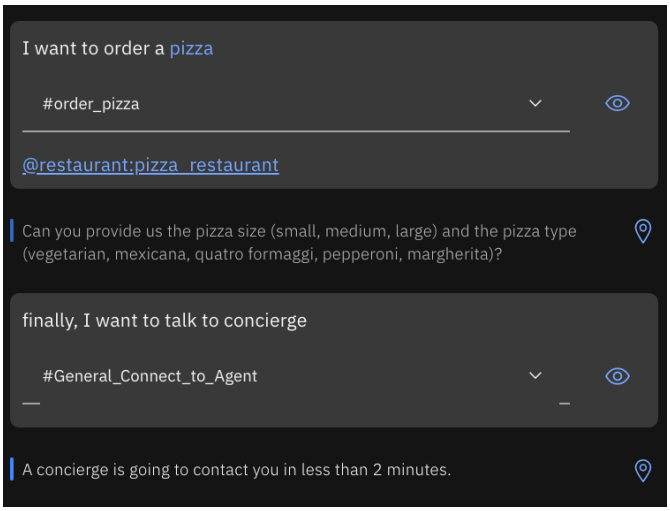
5. Enter : *ok, I stop my order* to conclude the ordering
6. Enter : *where is it?* to illustrate that the service kept the context.



7. Click **Clear**
8. Enter successively :

   I want to order a pizza

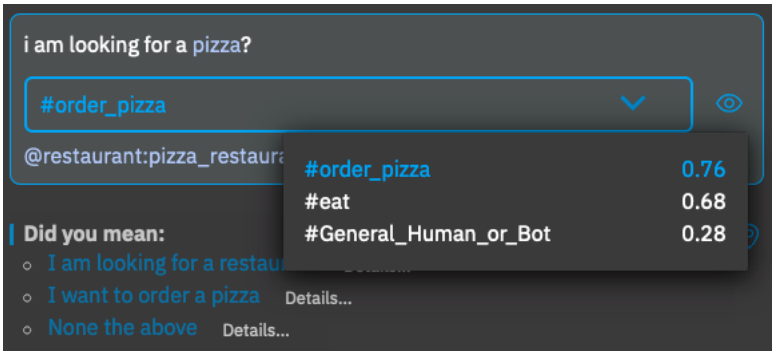
   finally, I want to talk to concierge



   The bot digresses away from the **Pizza Ordering** node to process the **Talk to concierge** node and does not return to the **Pizza Ordering** node, as expected.

Understanding disambiguation

Disambiguation instructs your assistant to ask the customer for help when more than one dialog node can respond to a customer's input. Instead of guessing which node to process, your assistant shares a list of the top node options with the user, and asks the user to pick the right one.
Such as:



Disambiguation is triggered when the following conditions are met:

- The confidence scores of the runner-up intents that are detected in the user input are close in value to the confidence score of the top intent.
- The confidence score of the top intent is above 0.2.

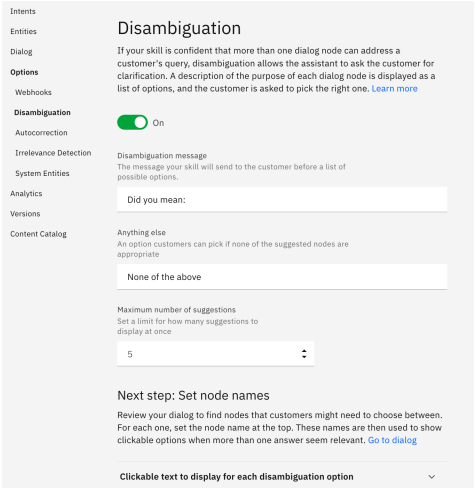
Even when these conditions are met, disambiguation does not occur unless two or more independent nodes in your dialog meet the following criteria:

- The node condition includes one of the intents that triggered disambiguation.
- A description of the node's purpose is provided for the node in the node name field. (Alternatively, a description can be included in the external node name field.)

18. Configure your Disambiguation

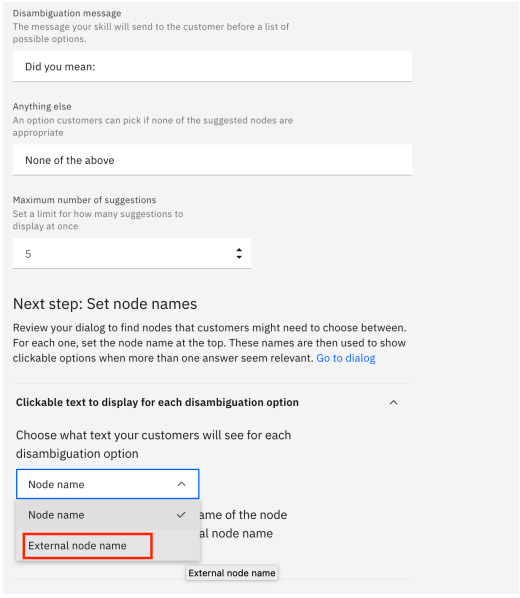
Disambiguation is enabled automatically for all new dialog skills. You can change the settings that are applied automatically to disambiguation from the **Options** page.

1. Select **Options** tab then **Disambiguation** tab
2. **Turn on** the option



3. Keep the default messages :
    a. Disambiguation message: *Did you mean:*
    b. Anything else: *None the above*

    As we want to set, later on, the list of potential nodes eligible for a Disanbiguation, you are going to leverage External node name.
4. At the bottom, expand **Clickable text to**
5. Select *External node name* as Node name



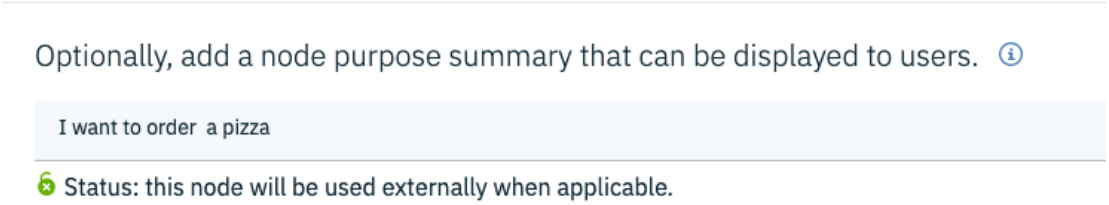
19. Configure eligible nodes

We are going to configure 2 nodes.

1. Return to **Dialog** tab and select **Pizza Ordering** node

You can add a node summary description to the **external node name** field instead of the **name** field. The **external node name** field serves two purposes. It provides information about the node to customers when it is included in a disambiguation list. It also describes the node in a chat summary that is shared with service desk agents when a conversation is transferred to a person. The **external node name** field is only visible in skills that are part of a Plus or Premium plan instance. If the **external node name** field contains text, its text is used, whether or not there is text in the **name** field.

2. At the bottom of the node editor page fill the field like this: *I want to order a pizza*

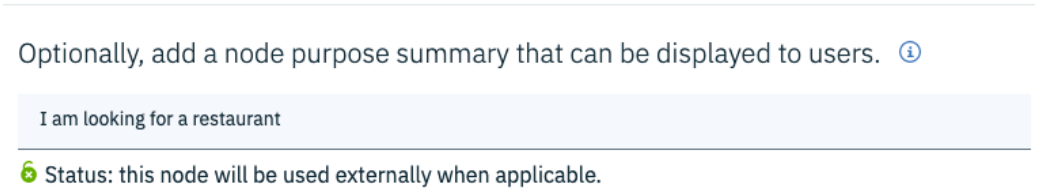


3. Select the **Find a restaurant branch** node

If you didn't create the node in the previous lab, just create a new node as below

Field	Value
Name of the node	*Find a restaurant branch*
Triggered by	*#eat*
Watson responses	You are looking for a restaurant

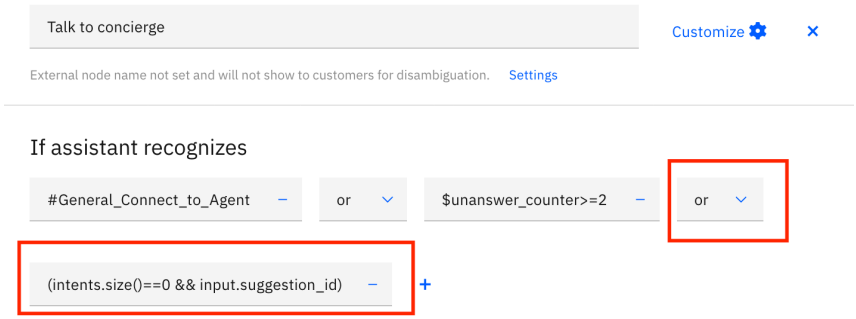
4. At the bottom fill the field like this: *I am looking for a restaurant*



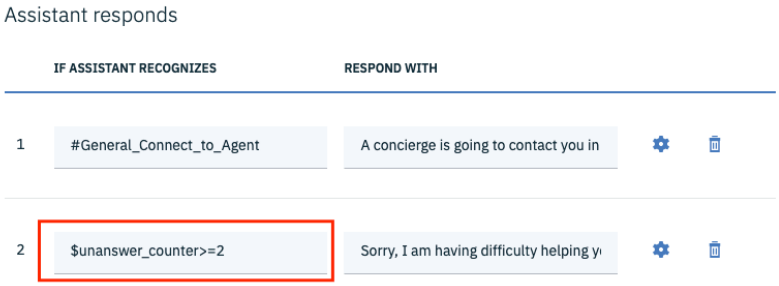
When WA will display the different options to the end user, it will propose also none of the above. When a user clicks the *None of the above* option, your assistant strips the intents that were recognized in the user input from the message and submits it again. This action typically triggers the anything else node in your dialog tree.
To customize the response that is returned in this situation, you can add a root node with a condition that checks for a user input with no recognized intents (the intents are stripped, remember) and contains a suggestion_id property. A suggestion_id property is added by your assistant when disambiguation is triggered.
In our use case we want to trigger **Talk to concierge** node.

5. Select the **Talk to concierge** node
6. Add the following condition:

   or (intents.size()==0 && input.suggestion_id)

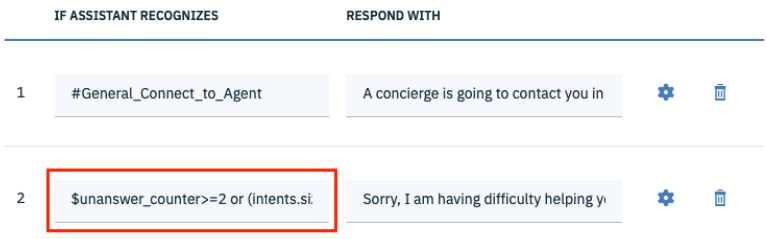


7. Select the second conditioned response



8. Add the following condition:

   or (intents.size()==0 && input.suggestion_id)



9. Close the node editor page.

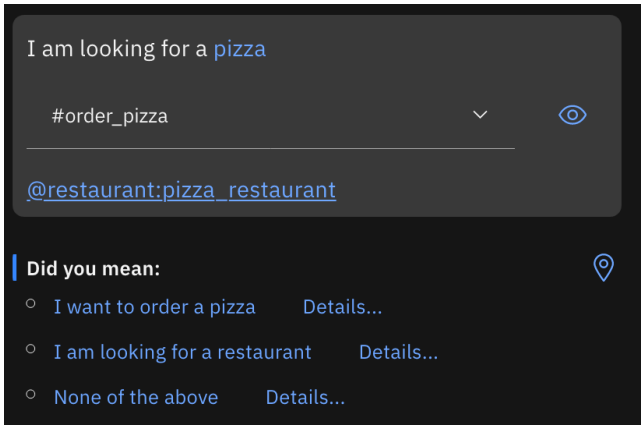
20. Test the disambiguation

you are going to assess our disambiguation.

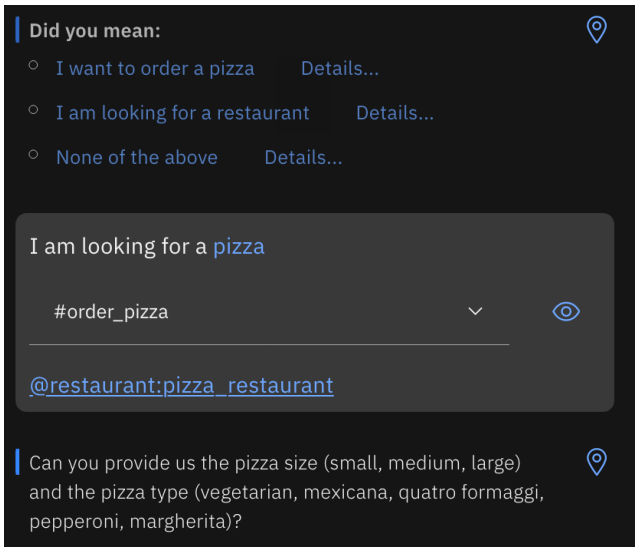
1. Open the **Try it out**
2. Enter *I am looking for a pizza*

WA displays the messages defined in the nodes using the identified intents:

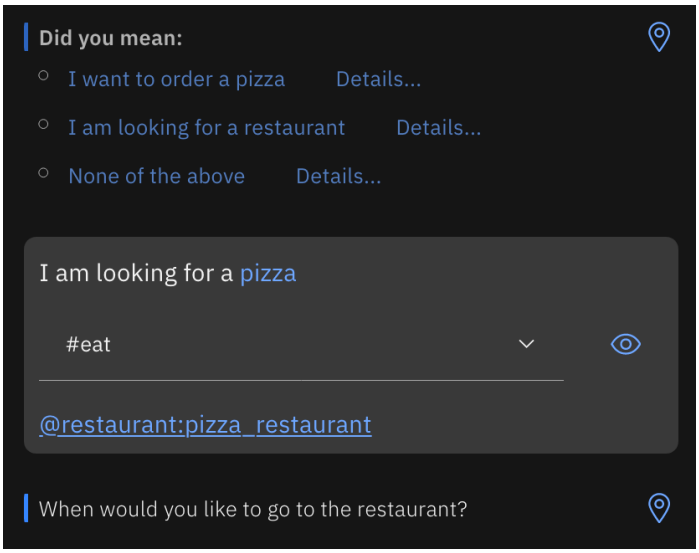
- Pizza ordering
- Find a restaurant



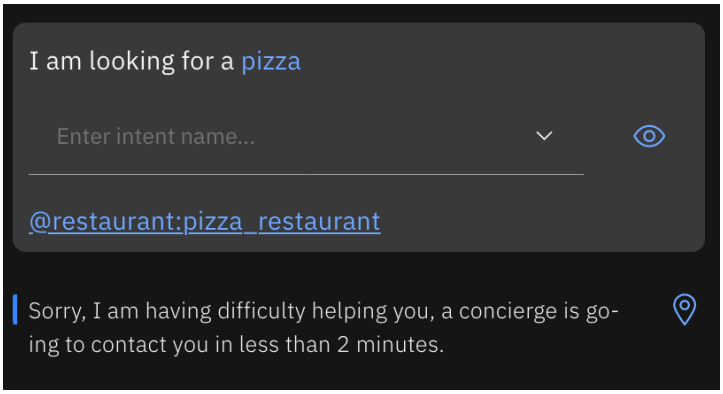
If you select the first option, you start the process to order a pizza:



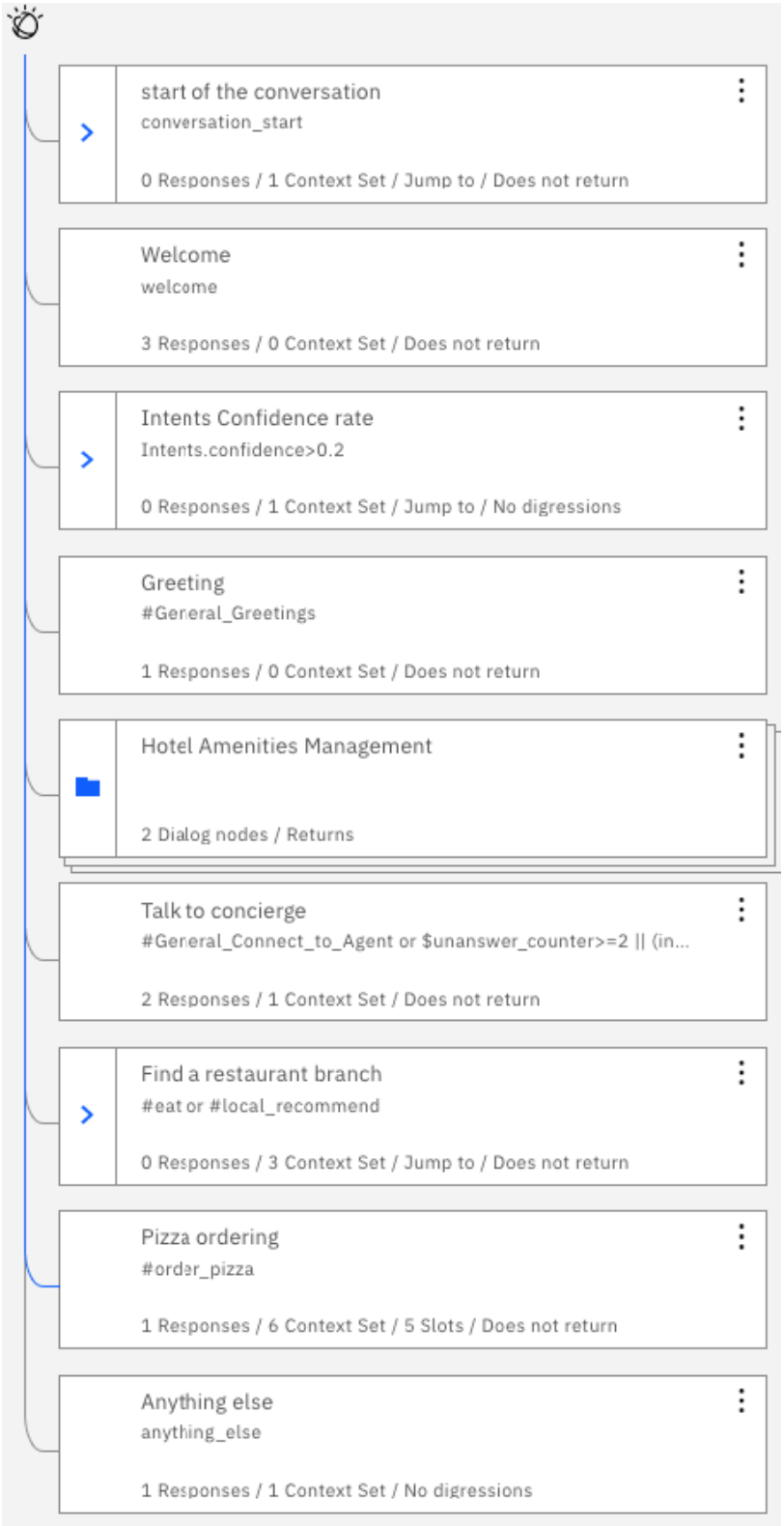
If you select the second option, you start the process to look for a restaurant:



If you select the third option, you start the process to contact a human agent:



The final dialog should look like this:



Note: if you want to compare your dialog with the one we created, feel free to import into your Assistant our Skill as it was after the end of this exercise. This skill is in file *skill-My-Concierge-Skill-Lab2part2.json*.