Session 5 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 Conversation Service provides several options to manage Conditions, and possibility to have several answers to make your bot more human.

In this lab, you will:

- Learn how to use IBM Cloud Function from the dialog
- Gather information with Slots

## 3. Prerequisites

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

- Session 5 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 |
|---|---|
| US | https://console.ng.bluemix.net/ |
| UK | https://console.eu-gb.bluemix.net/ |
| Sidney | https://console.au-syd.bluemix.net/ |
| Germany | https://console.eu-de.bluemix.net/ |

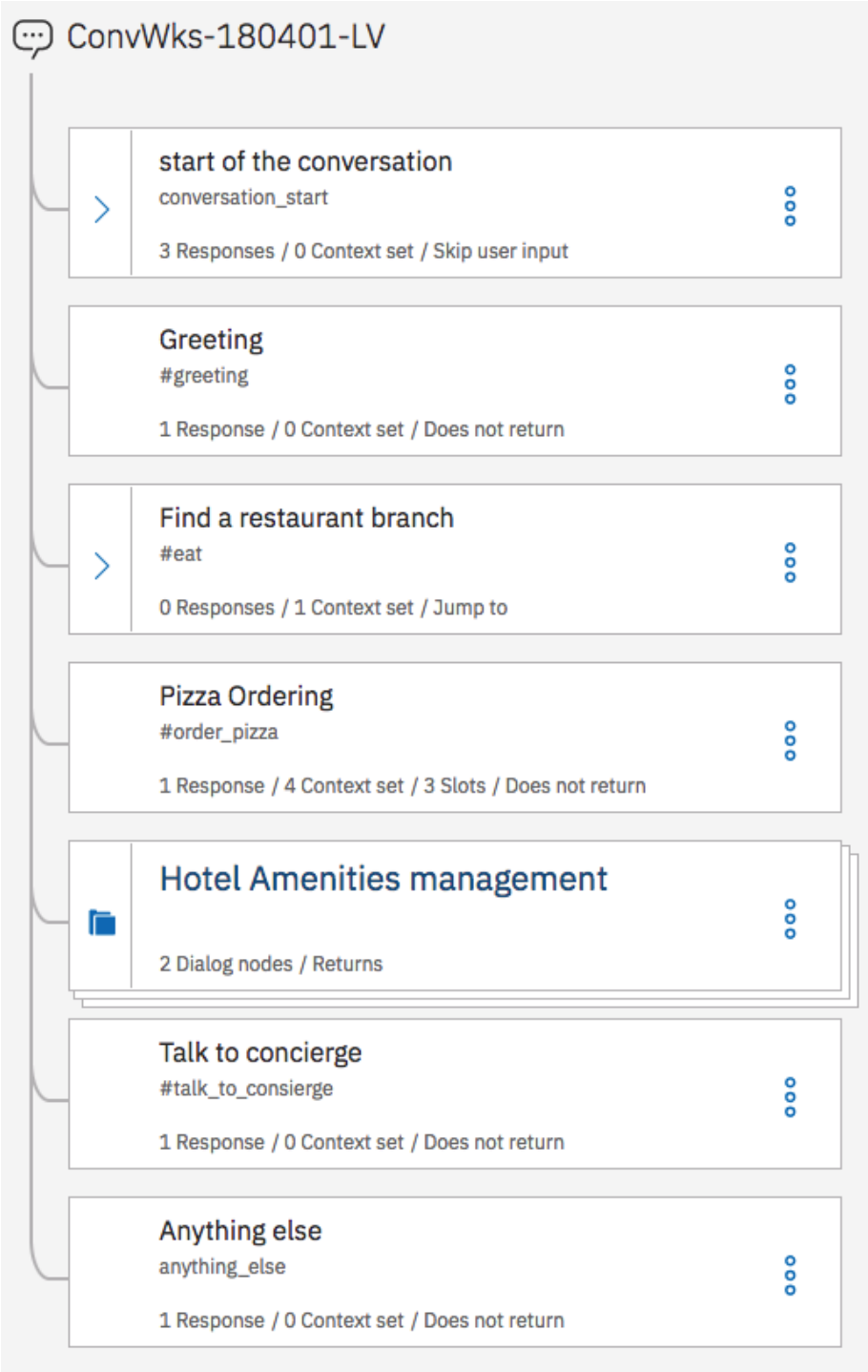
## 4. Scenario

**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.



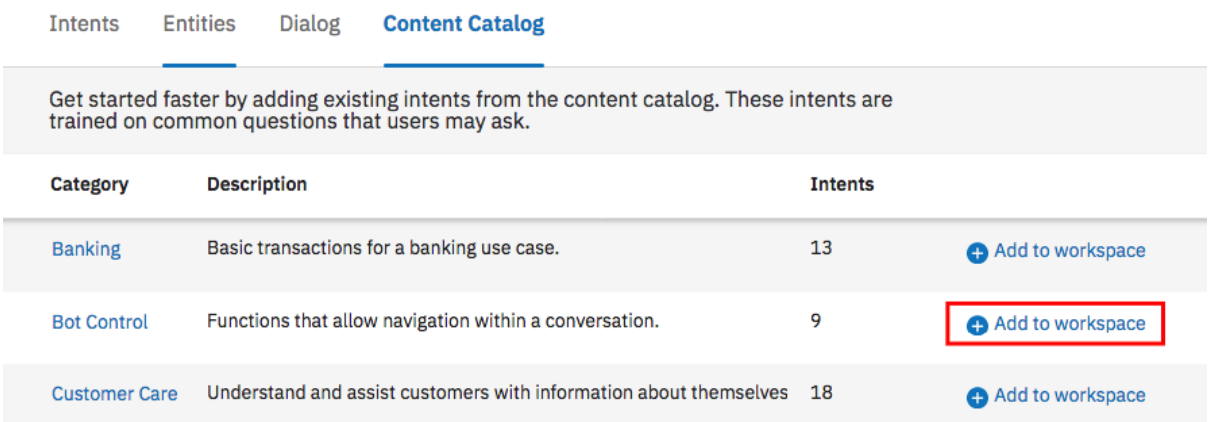
# Gathering information with Slots

To gather information, you have created a branch, now you can use Slots to do it and simplify your dialog.
You can think of slots as the chatbot version of a web form in which users must fill out 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. Bot Control precreated intents.

At the end of the acquisition of all information, we are going to request a validation of the order. To do this we are using 2 existing intents *#Bot_Control_Approve_Response* and *#Bot_control_reject_Response*.

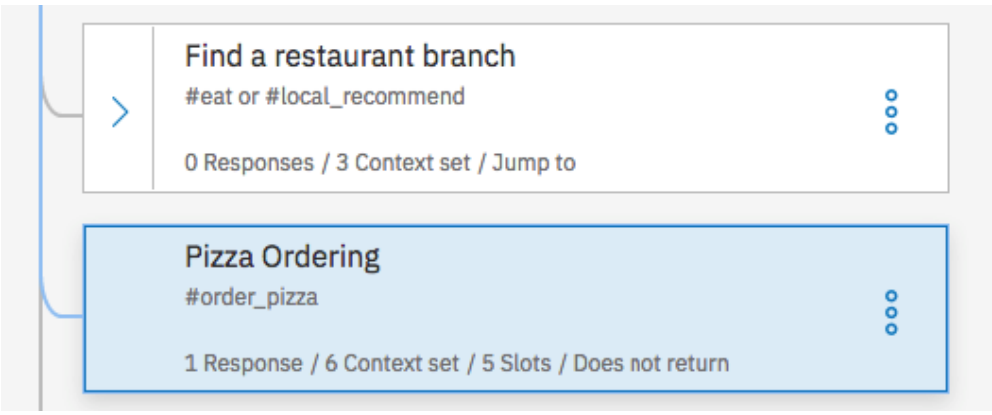
1. Go back to **Content Catalog** tab
2. On *Bot Control* row, Click Add to workspace



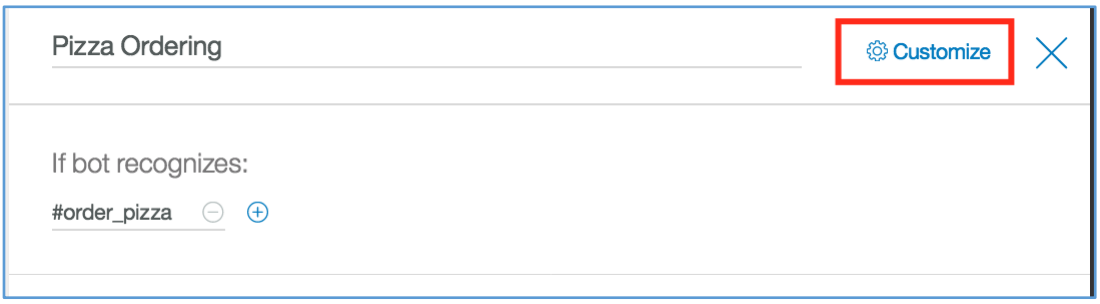
## 7. Add Pizza Ordering node and slots

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

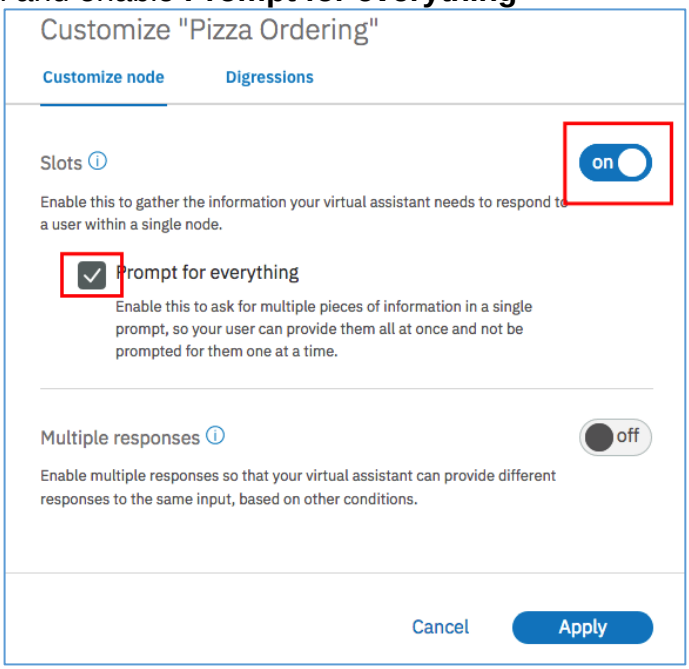
1. Go back to Dialog page
2. Add a node below **Find a restaurant branch** node




3. Set *#order_pizza* as condition and *Pizza Ordering* as name
4. Click **Customize**



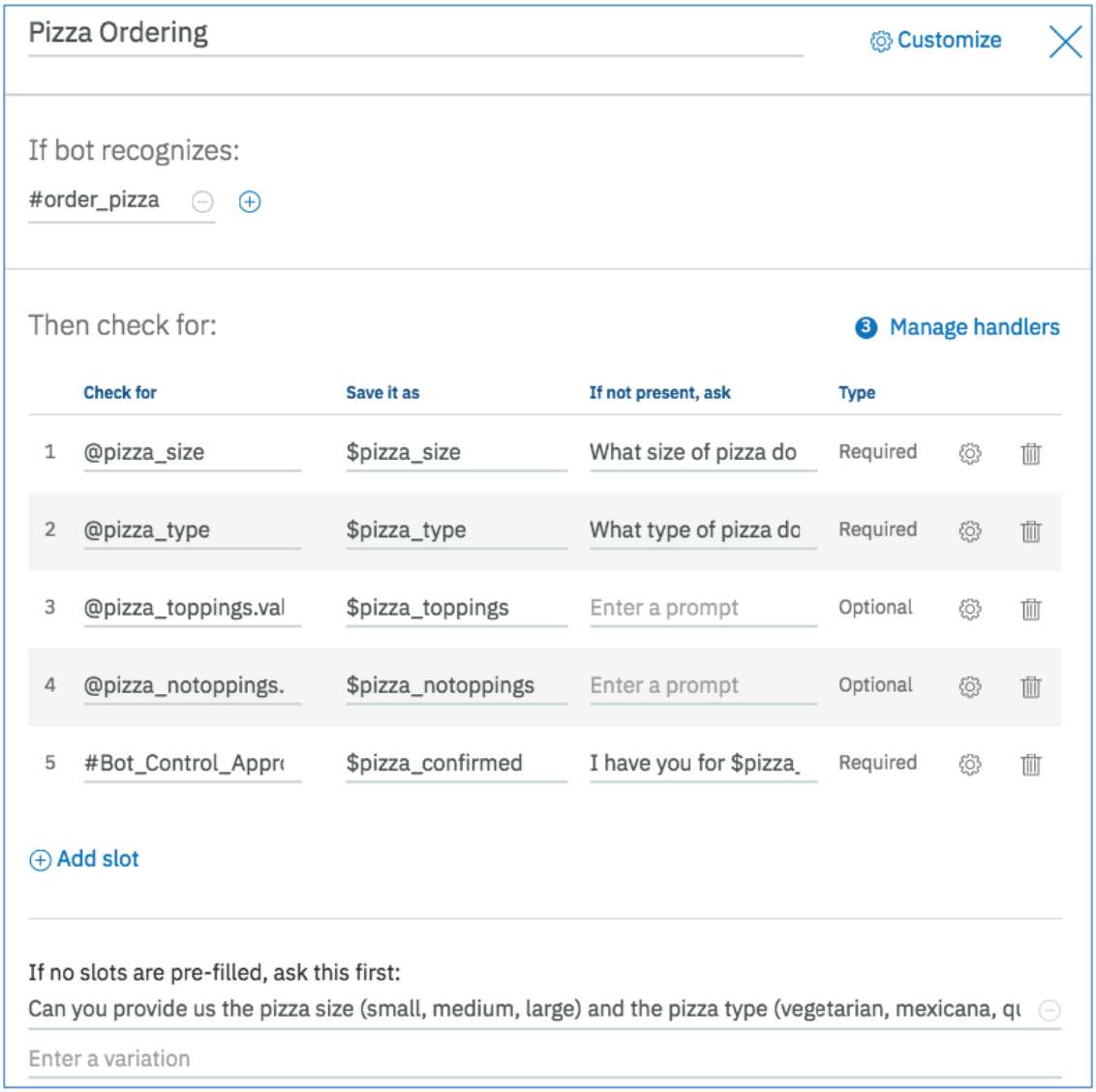

5. Switch **Slots** on and enable **Prompt for everything**



6. Click **Apply**
7. At the bottom of the edit page click **Add slot** 2 times
8. Fill the slots like these

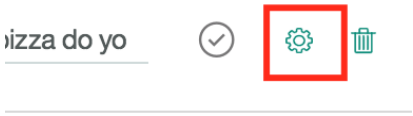
| Check for | Save it as | If not present, ask |
|---|---|---|
| *@pizza_size* | *$pizza_size* | *What size of pizza do you want?* |
| *@pizza_type* | *$pizza_type* | *What type of pizza do you want?* |
| *@pizza_toppings.values* | *$pizza_toppings* | |
| *@pizza_notoppings.values* | *$pizza_notoppings*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

9. In the filed **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)?*

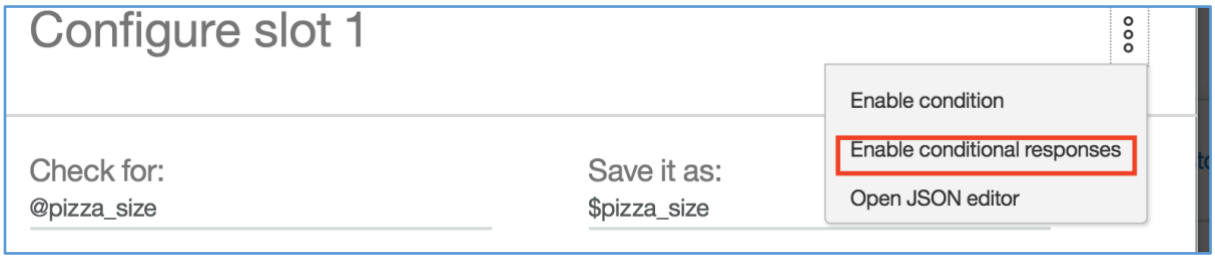


## 8. Manage the basic order information : pizza size

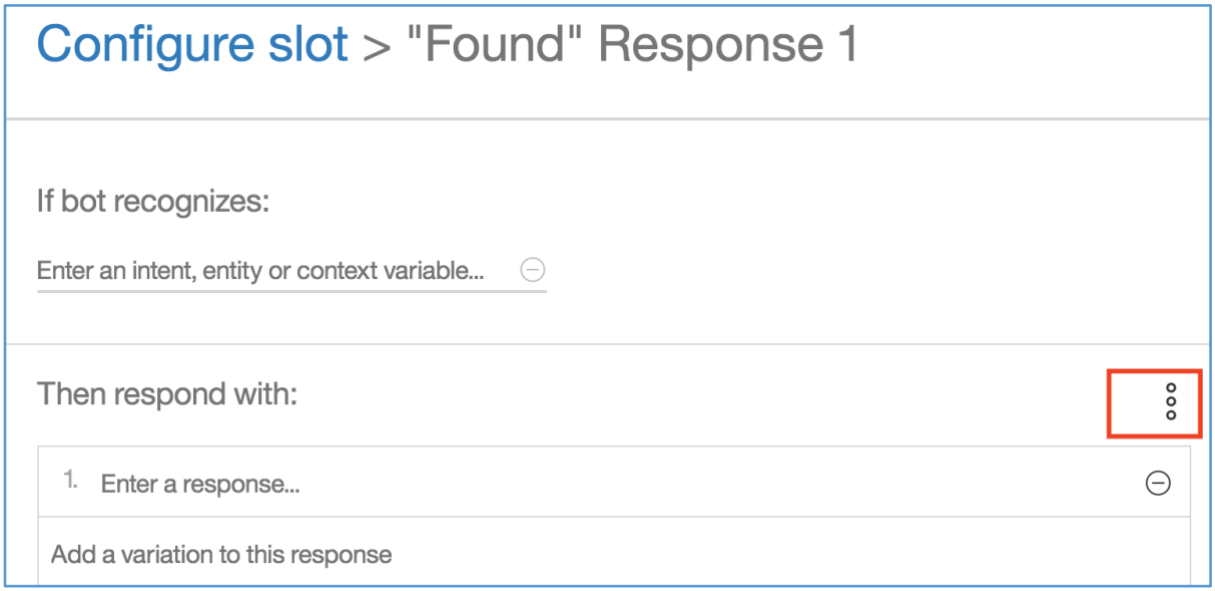
1. Click **settings/customize/edit slot** icon of the **pizza_size** slot




2. In the configure slot window, click on the 3 dots menu and select **enable conditional responses**




3. In **Found** frame, on the first row, click **edit** icon 
4. Open the context editor. (the 3 dots)



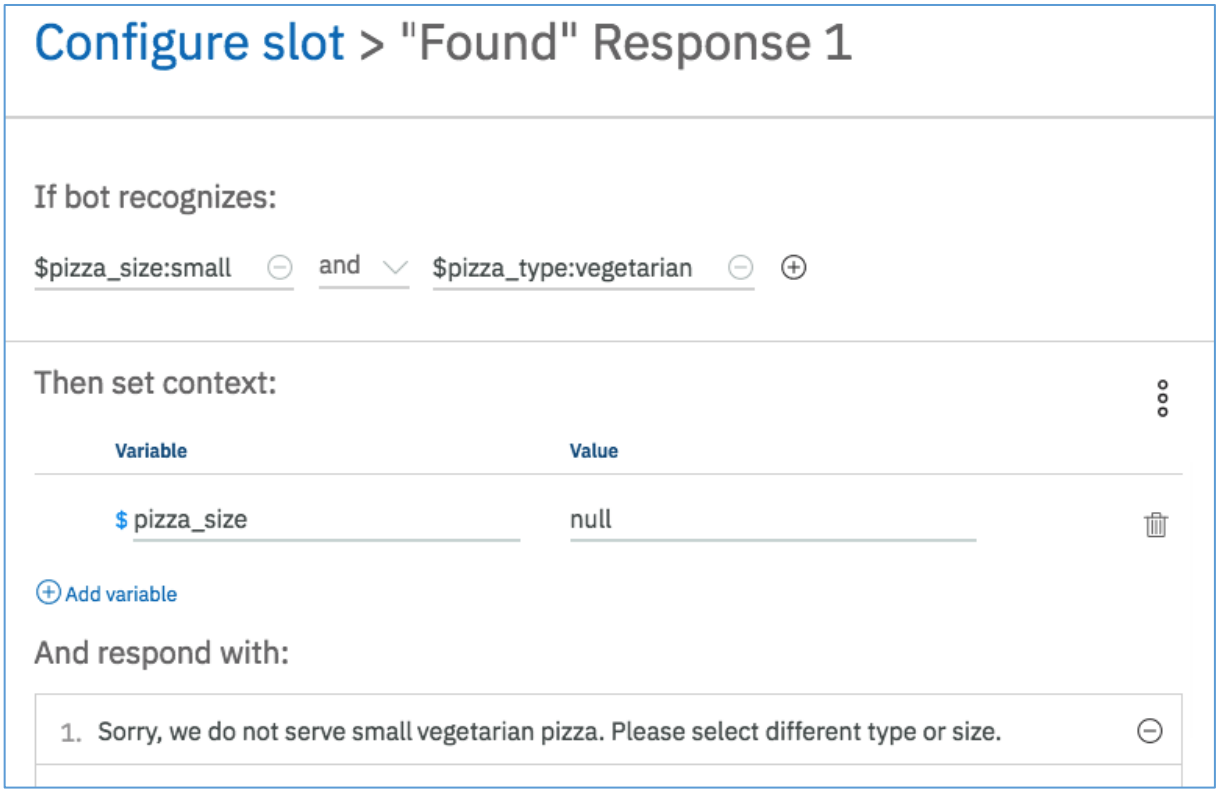
5. Fill the slot ‘found” like this

   Condition : *$pizza_size:small && $pizza_type:vegetarian*

   Context variable : *pizza_size*

   Context value : *null*

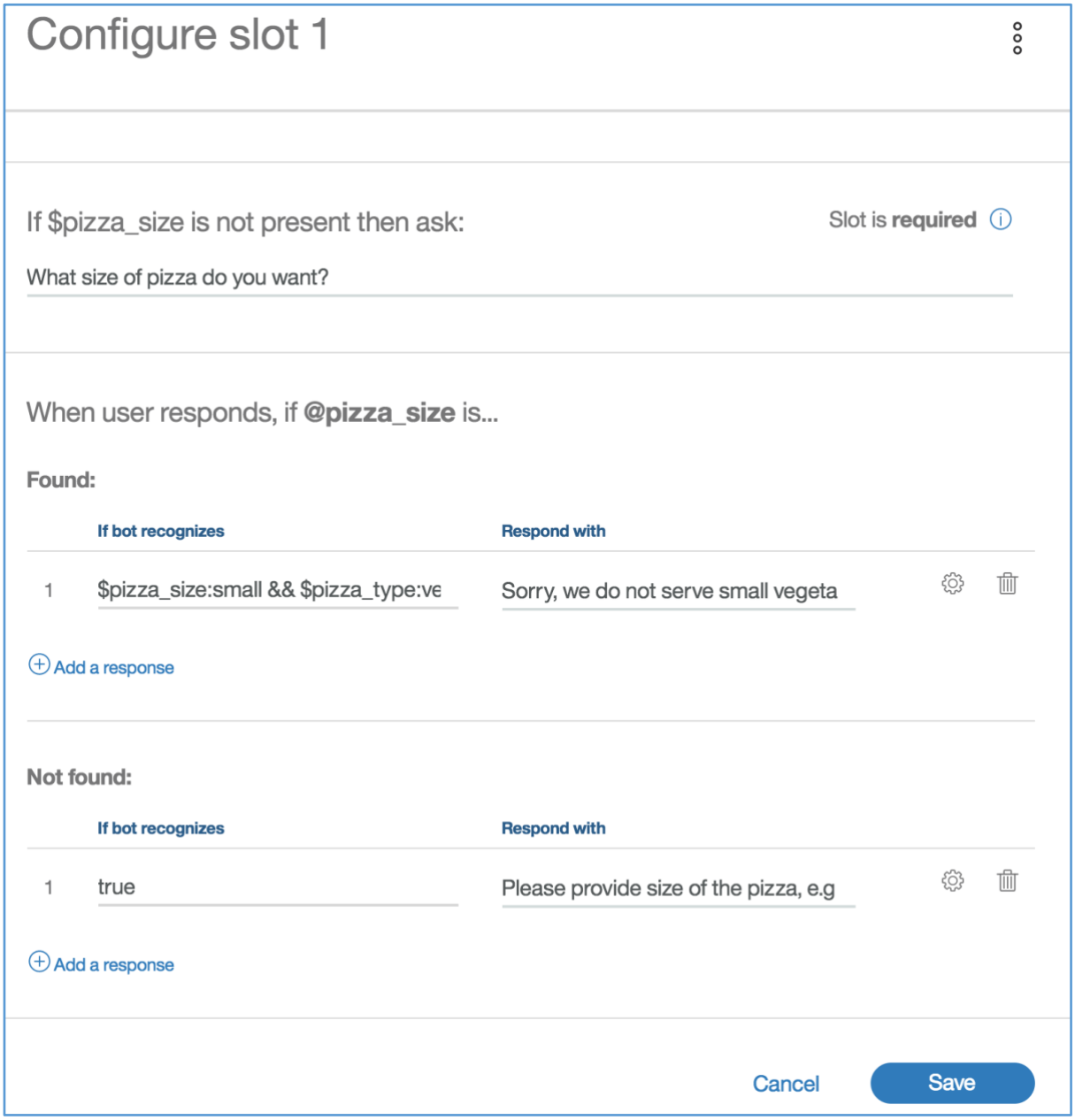
   Respond : *Sorry, we do not serve small vegetarian pizza. Please select different type or size.*




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.

6. Click **back**
7. In **Not found** frame, enter the respond:
   condition: *true*
   response: *Please provide size of the pizza, e.g small, medium or large*.

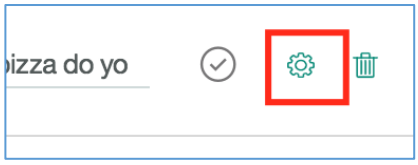
Below the first slot:



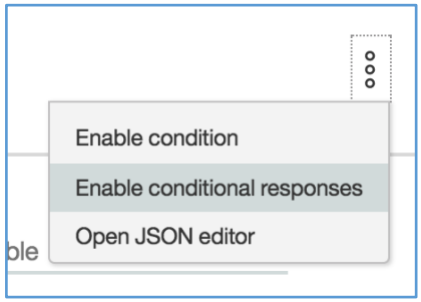

8. Click **Save**

## 9. Manage the basic order information : pizza type

1. Click settings icon of the **pizza_type** slot




9. In the configure slot window, click on the 3 dots menu and select **enable conditional responses**



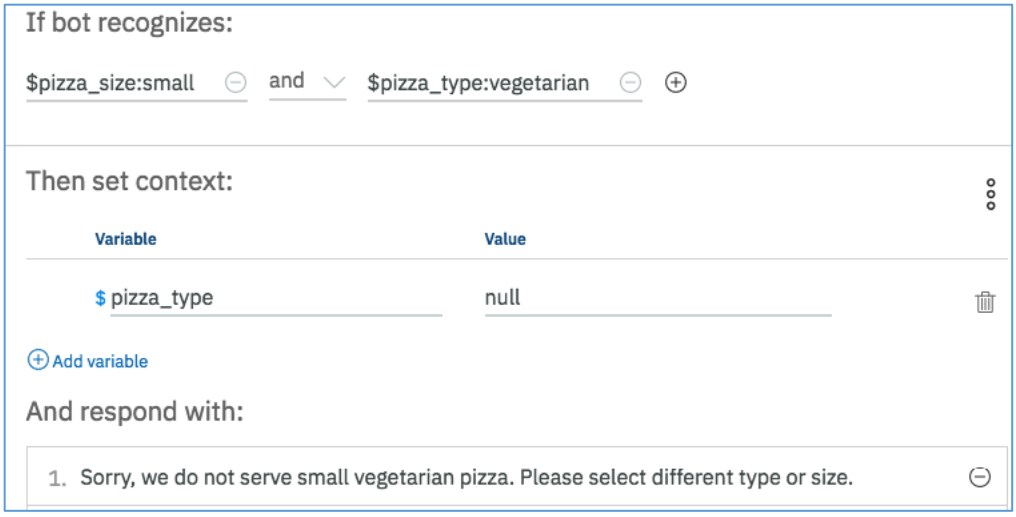

10. In **Found** frame, on the first row, click **edit** icon 
11. Open the context editor. (the 3 dots)
12. Fill the slot 'found" like this

    Condition : *$pizza_size:small && $pizza_type:vegetarian*

    Context variable : *pizza_type*

    Context value : *null*

    Respond : *Sorry, we do not serve small vegetarian pizza. Please select different type or size.*




13. Click **back**

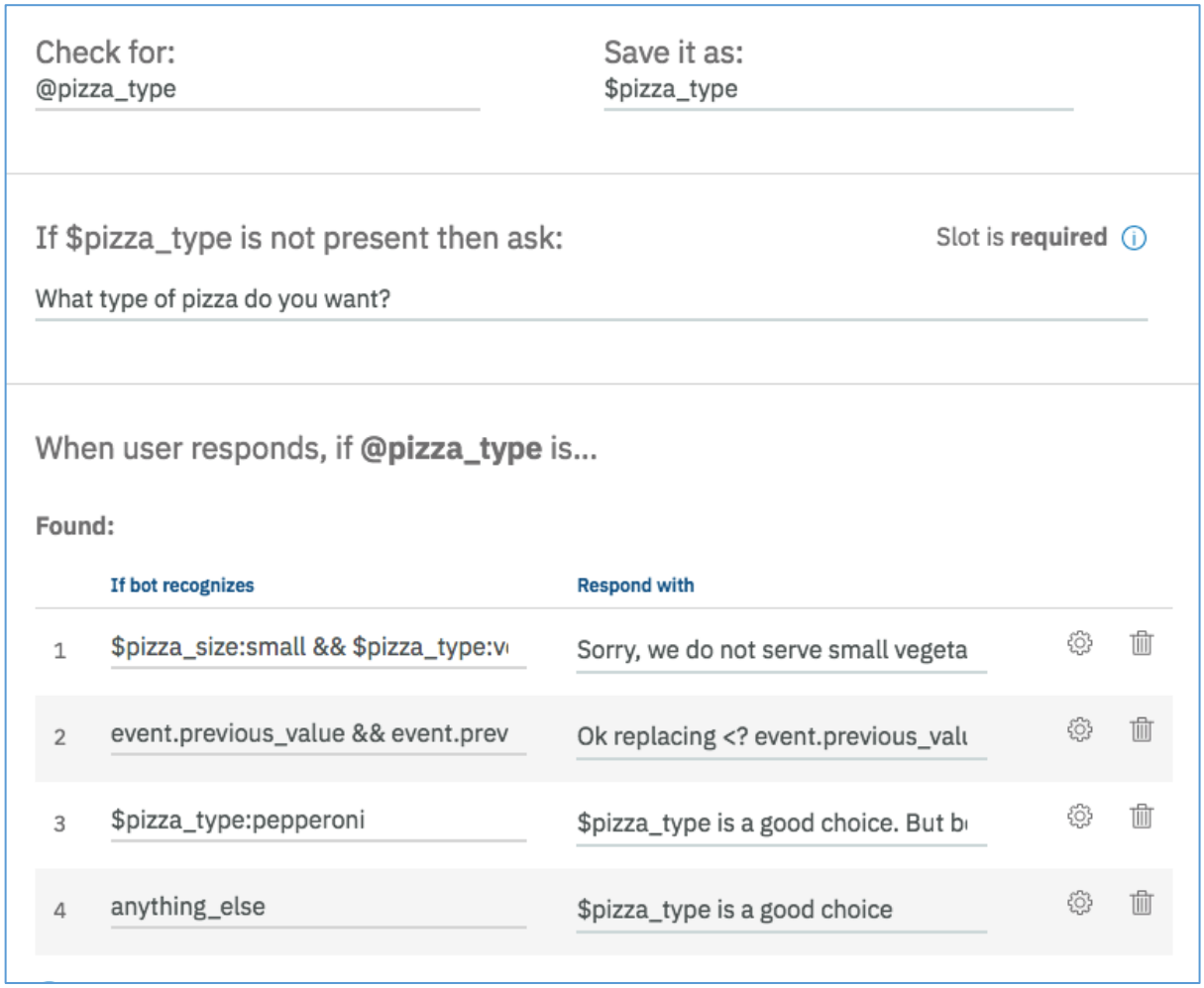
2. In the Found frame, add 3 more responses and condition like that:

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

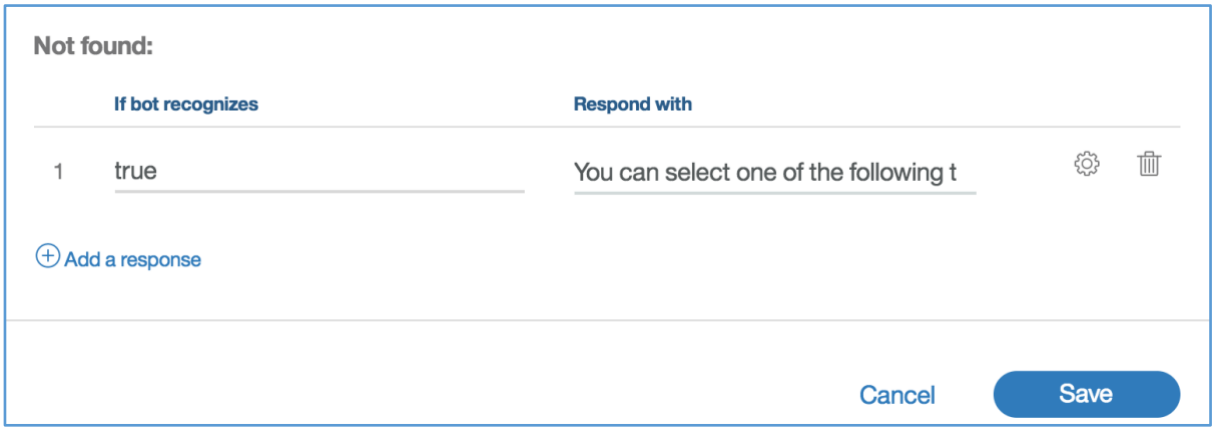
   Resp3 condition: *$pizza_type:pepperoni*
   Resp3 response: *$pizza_type is a good choice. But be warned, pepperoni is very hot!*

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

That's the way to enrich the chatbot responses and make it more human like.



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

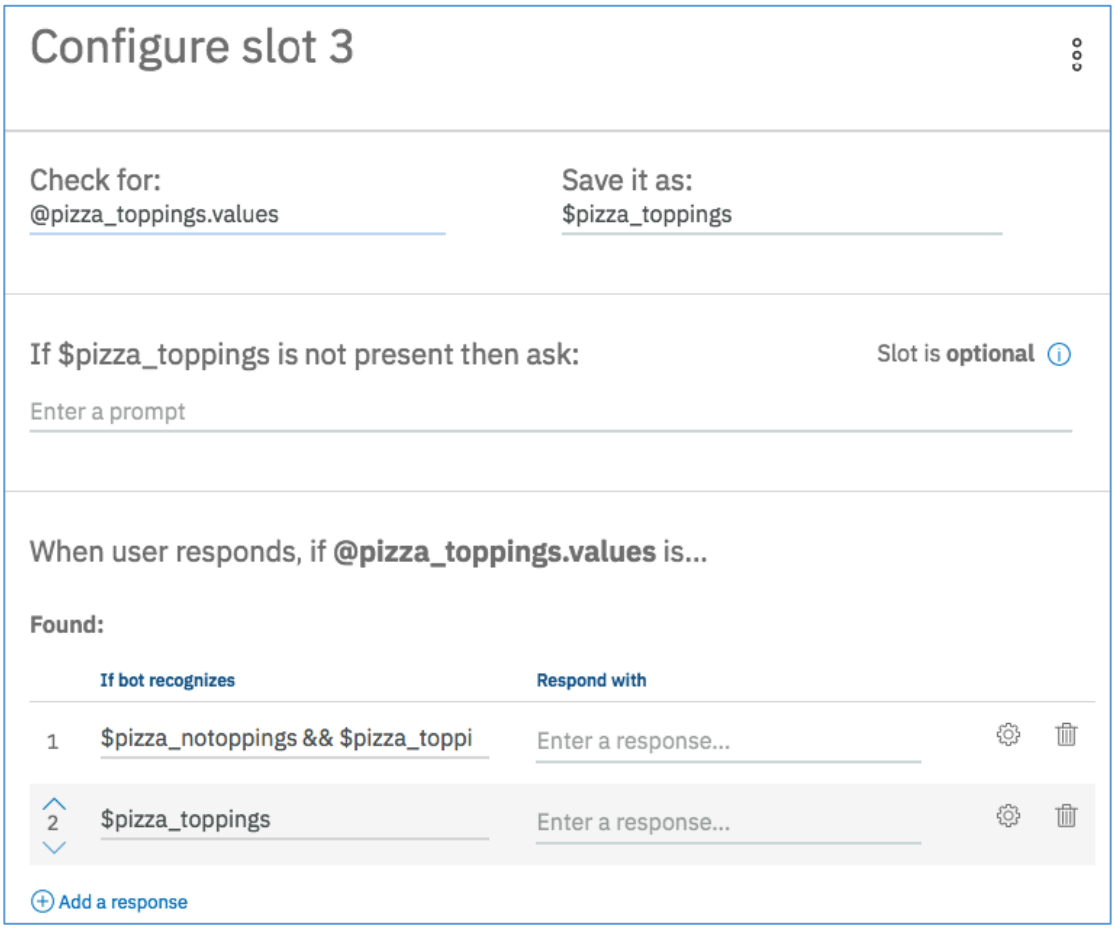



4. Click **Save**

## 10. Manage the toppings the user would like to add

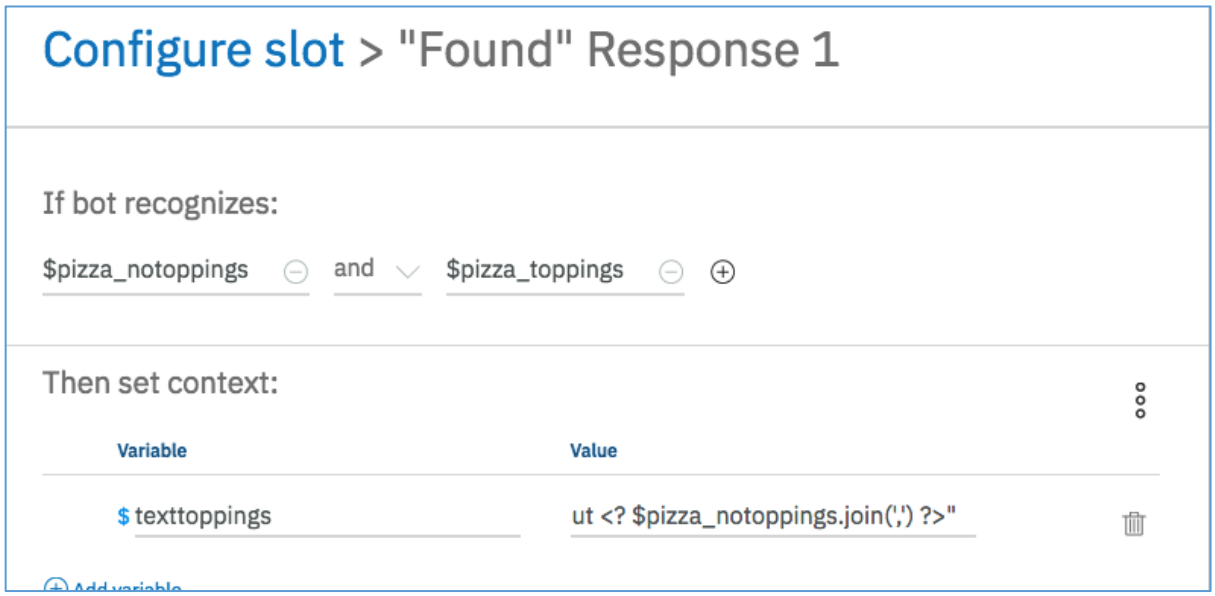
1. Click **settings** icon of the **pizza_toppings** slot
2. In the configure slot window, click on the 3 dots menu and select **enable conditional responses**
3. In **Found** frame, add 2 responses and condition like that:

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



4. In **Found** frame, Click **Edit** icon for the first condition, 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 **Back**
6. In **Found** frame, Click **Edit** icon for the second condition, then open the **Context editor** and fill it like this:

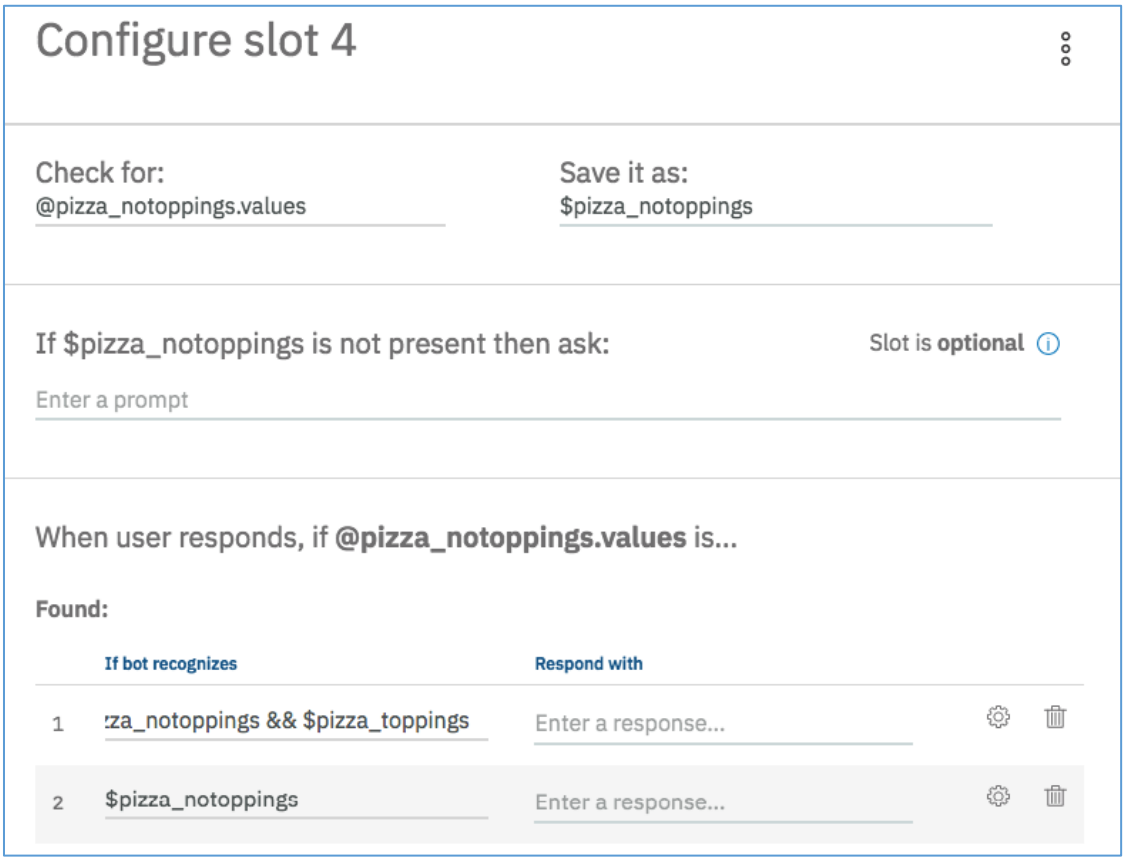
   Resp1 context variable: *texttoppings*
   Resp1 context value: *with <? $pizza_toppings.join(',') ?>*


7. Click **Back**
8. Click **Save**

## 11. Manage the toppings the user would like to remove

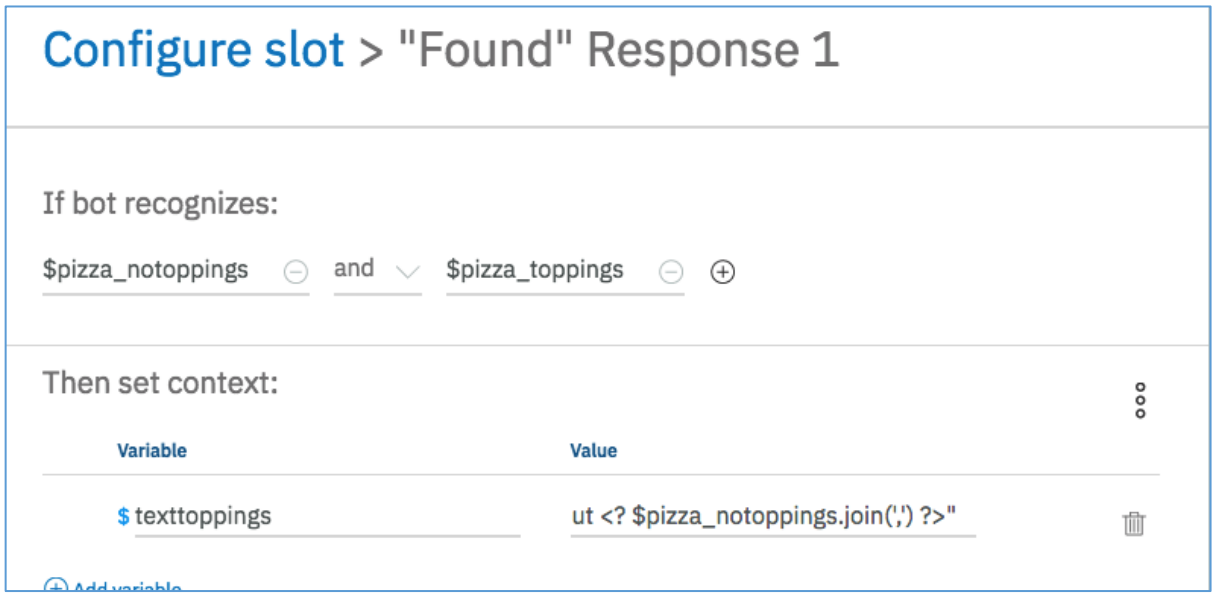
1. Click **settings** icon of the **pizza_notoppings** slot
2. In the configure slot window, click on the 3 dots menu and select **enable conditional responses**
3. In **Found** frame, add 2 responses and condition like that:

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



4. In **Found** frame, Click **Edit** icon for the first condition, 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(',') ?>*



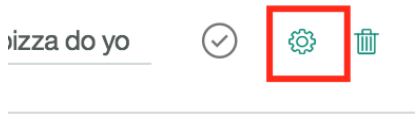

5. Click **Back**
6. In **Found** frame, Click **Edit** icon for the second condition, then open the **Context editor** and fill it like this:

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


7. Click **Back**
8. Click **Save**

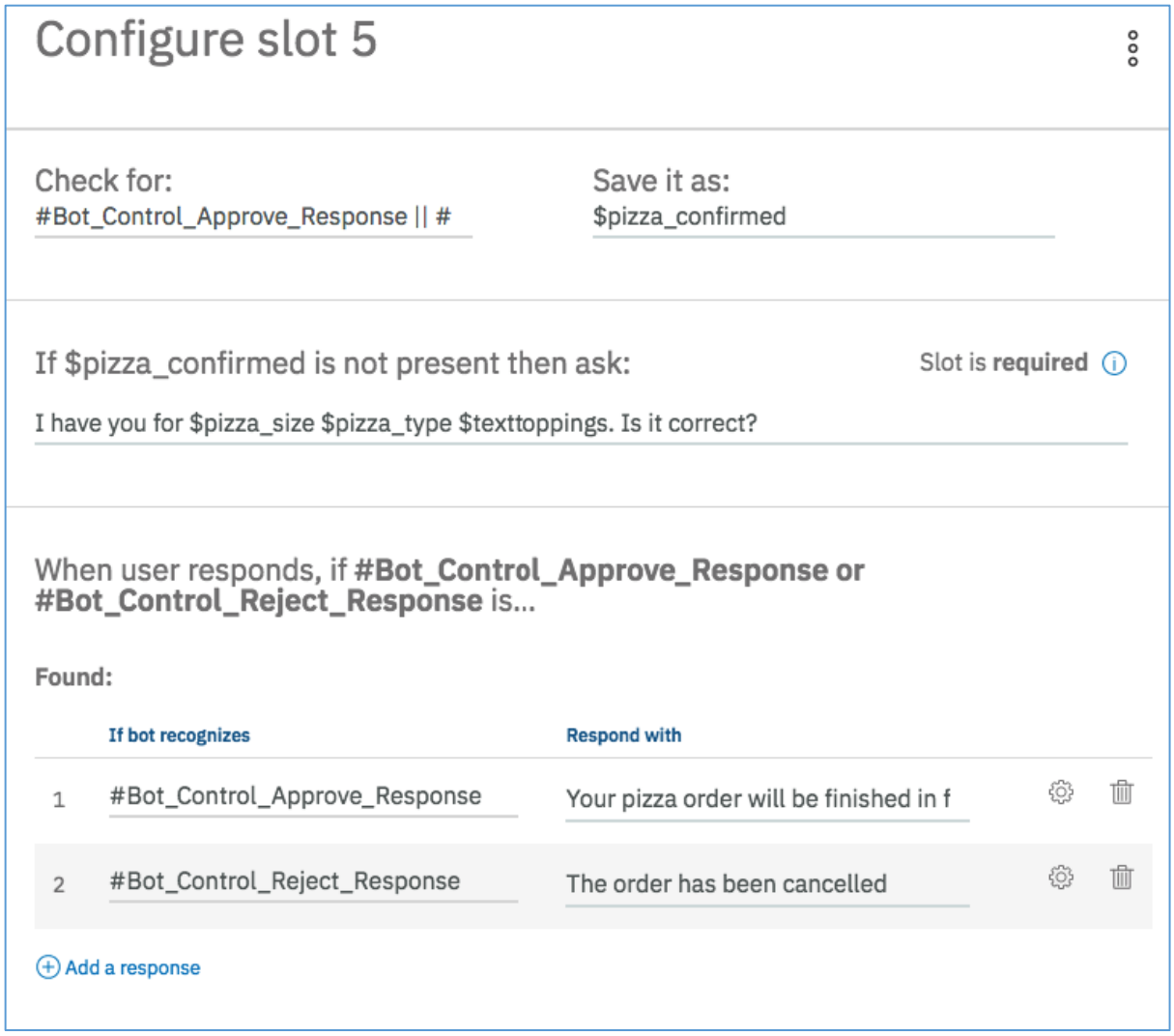
## 12. Manage the confirmation

9. Click settings icon of the **pizza_confirmed** slot

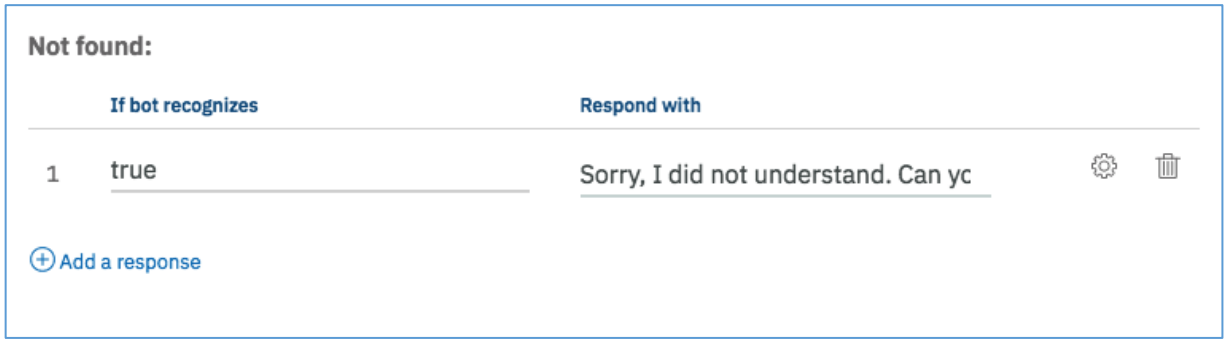



10. In the configure slot window, click on the 3 dots menu and select **enable conditional responses**
11. In **Found** frame, add 2 responses and condition like that:

    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: *The order has been cancelled.*



1. In **Not found** frame, 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.*

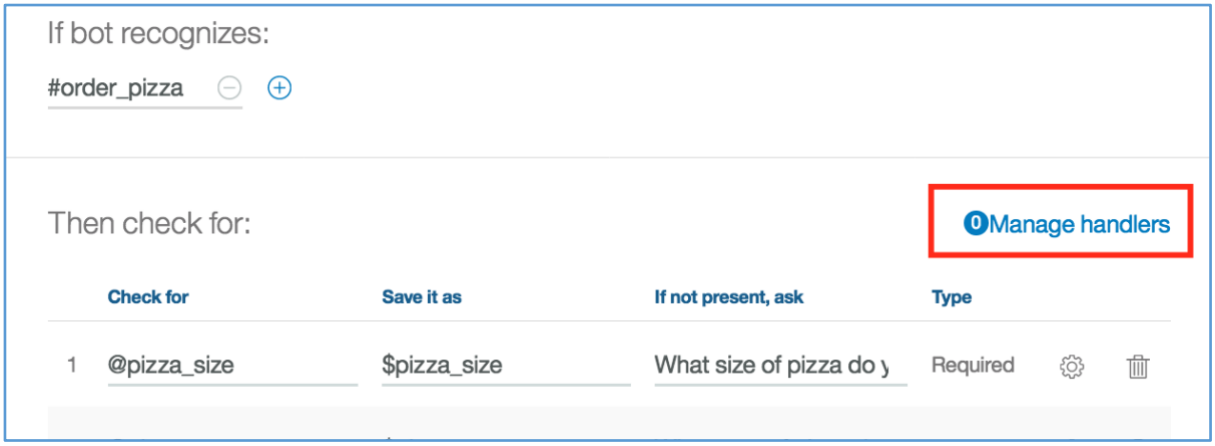



12. Click **Save**

## 13. Manage Handlers

You can optionally define node-level handlers that provide responses to questions users might ask during the interaction that are tangential 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**



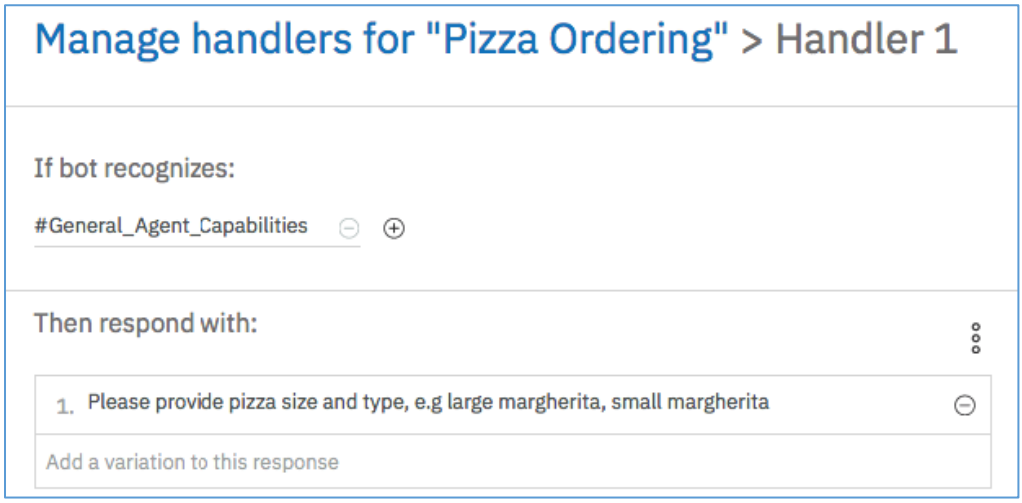

You are going to add 3 handlers.

3. Click twice **Add handler**

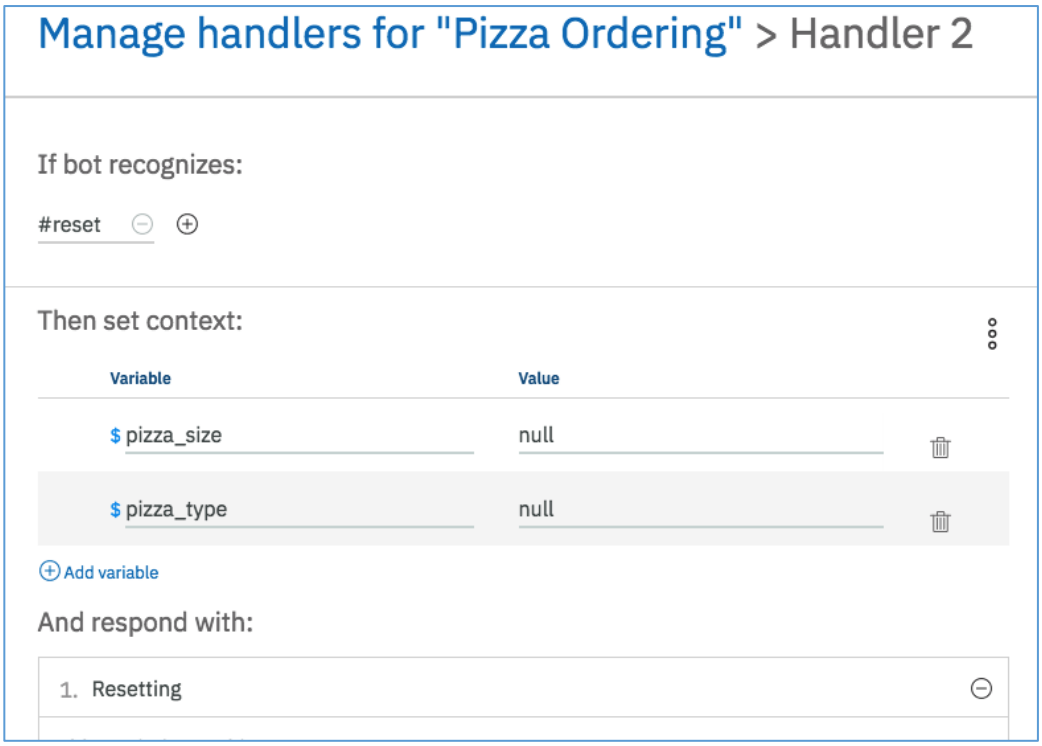
   **Note:** to open the json editor, you must click on edit icon of the selected row then the 3 dots on the new window

4. Fill the 3 handlers as defined below (for second and third open context editor)

   Handler1 condition: *#General_Agent_Capabilities*
   Handler1 response: *Please provide pizza size and type, e.g large margherita, small margherita.*

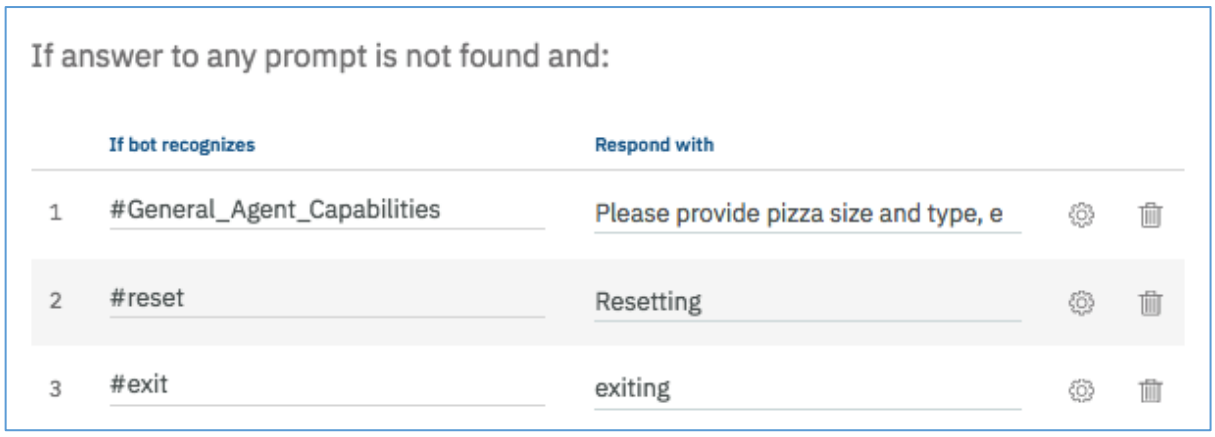



   Handler2 condition: *#reset*
   Handler2 response: *Resetting*
   Handler2 Context 1 Variable: *pizza_size*
   Handler2 Context 1 Value: *null*
   Handler2 Context 2 Variable: *pizza_type*
   Handler2 Context 2 Value: *null*



Handler3 condition: *#exit*
Handler3 response: *exiting*
Handler3 Context 1 Variable: *pizza_size*
Handler3 Context 1 Value: *"no_def"*
Handler3 Context 2 Variable: *pizza_type*
Handler3 Context 2 Value: *"no_def"*
Handler3 Context 3 Variable: *pizza_confirmed*
Handler3 Context 3 Value: *false*


Then you should have

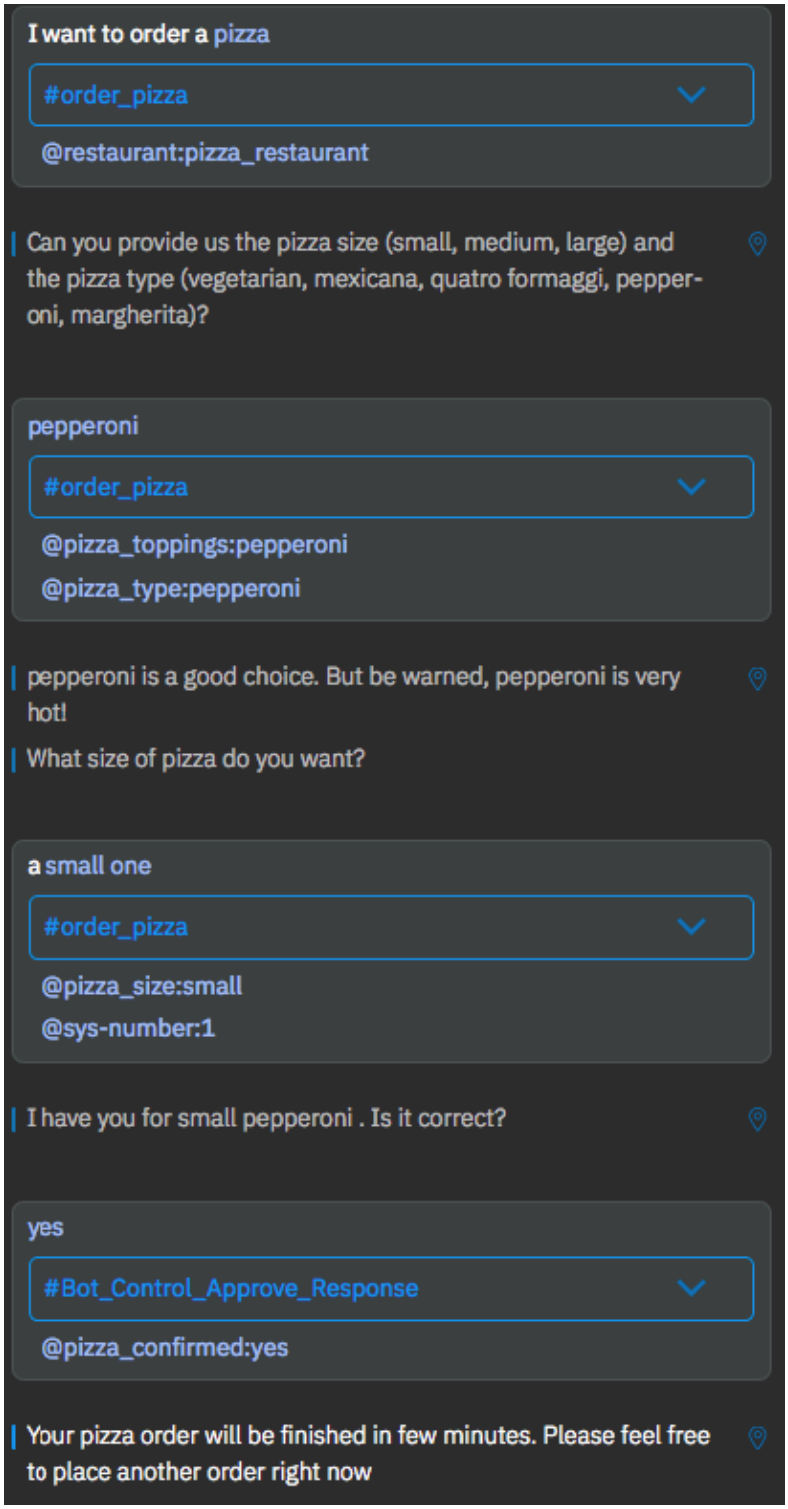



5. Click **Save**

## 14. 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**

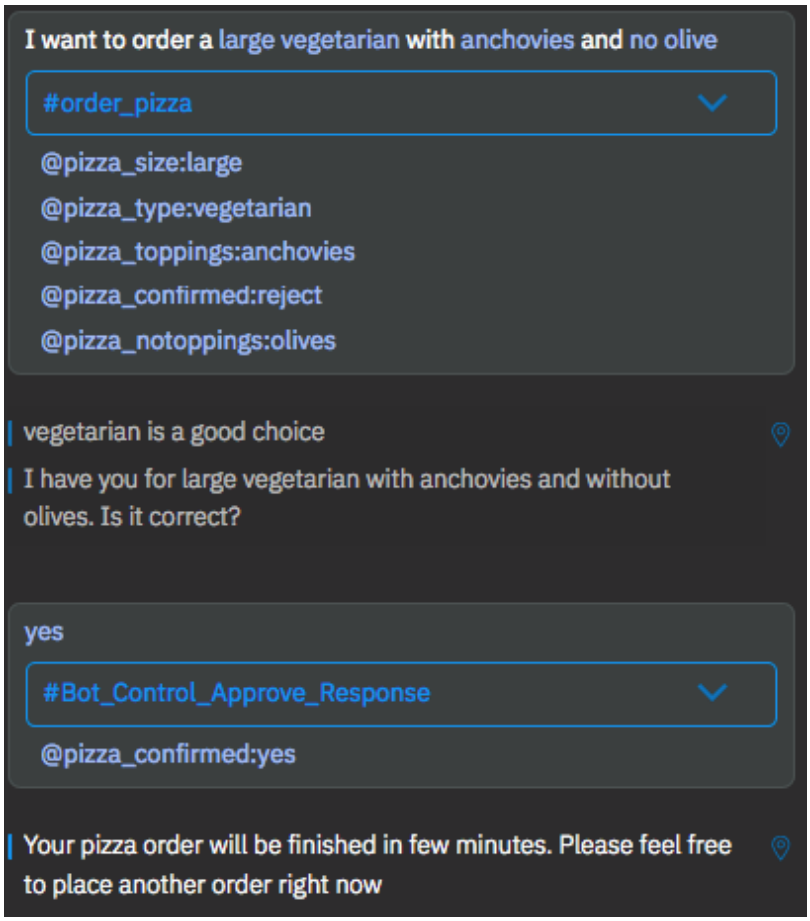
3. Enter successively

   *I want to order a small vegetarian*
   *so, a large one*
   *Yes*


4. Click **Clear**

5. Enter successively

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

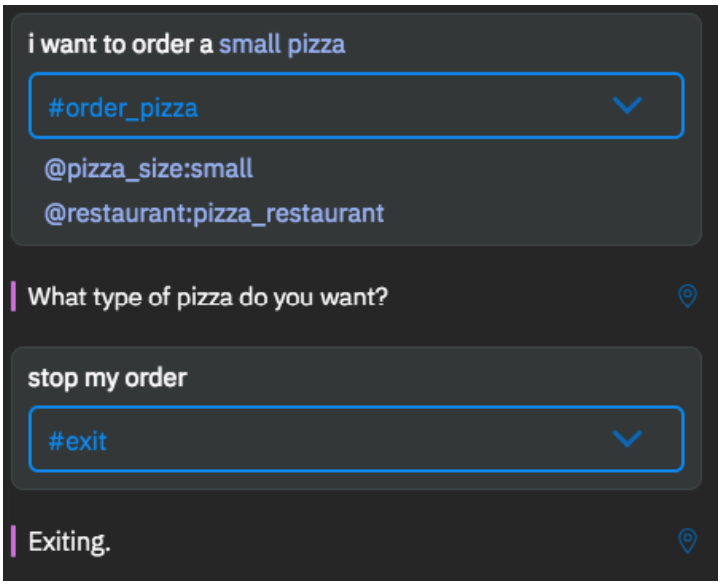



6. Click **Clear**

7. Enter successively

   *I want to order a small pizza*

   *stop my order*




If you open the context variable panel, you retrieve the values set by the handler. The client application has to understand that the command was cancelled. That's a possibility to stop the slot.


You can run some other tests and order a pizza by using your chatbot.

# Managing nodes and folders

We can group dialog nodes together by adding them to a folder.

- It allows a dialog designer to organize content based on topics
- It is a 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
- 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, the service first evaluate the folder conditions to determine whether to process the nodes within it.

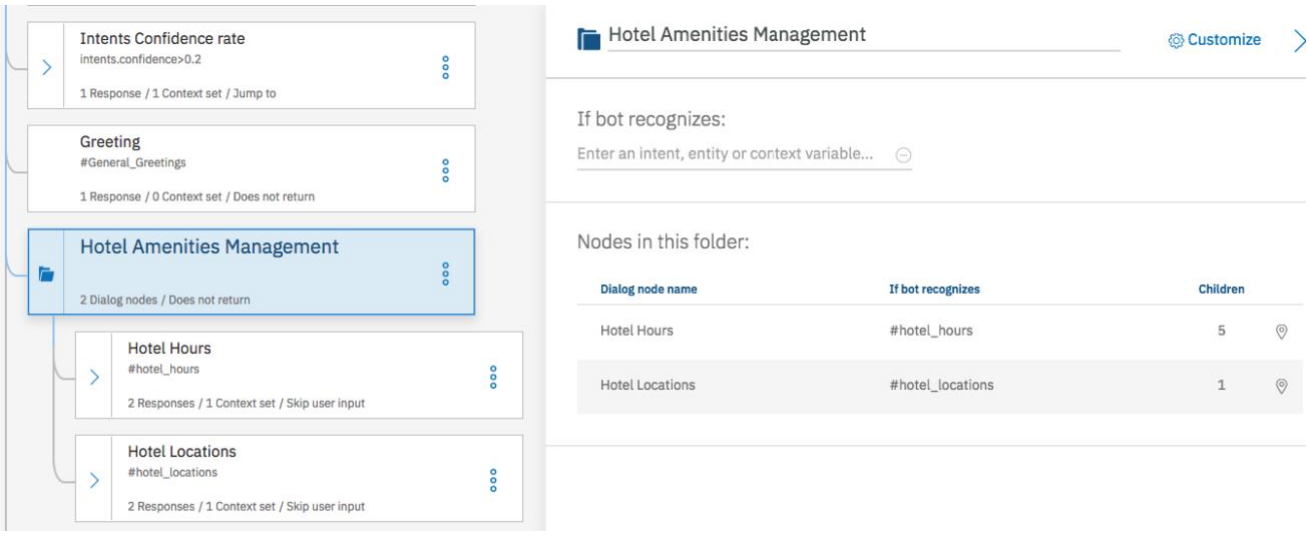
The nodes inherit of the digression settings of the folder.

## 15. Add a folder

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

1. On the dialog tab, Select **Greeting** node and click **Add folder**
2. Name it *Hotel Amenities Management*

   We don't apply neither condition nor settings, as we just want to organise our dialog.
3. Move **Hotel Hours** and **Hotel Locations** nodes in it.




If we make some test the behaviours of the conversation stay the same.

# Understanding digressions

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

## 16. Configure your digressions

We are going to configure 2 nodes and 1 folder.

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

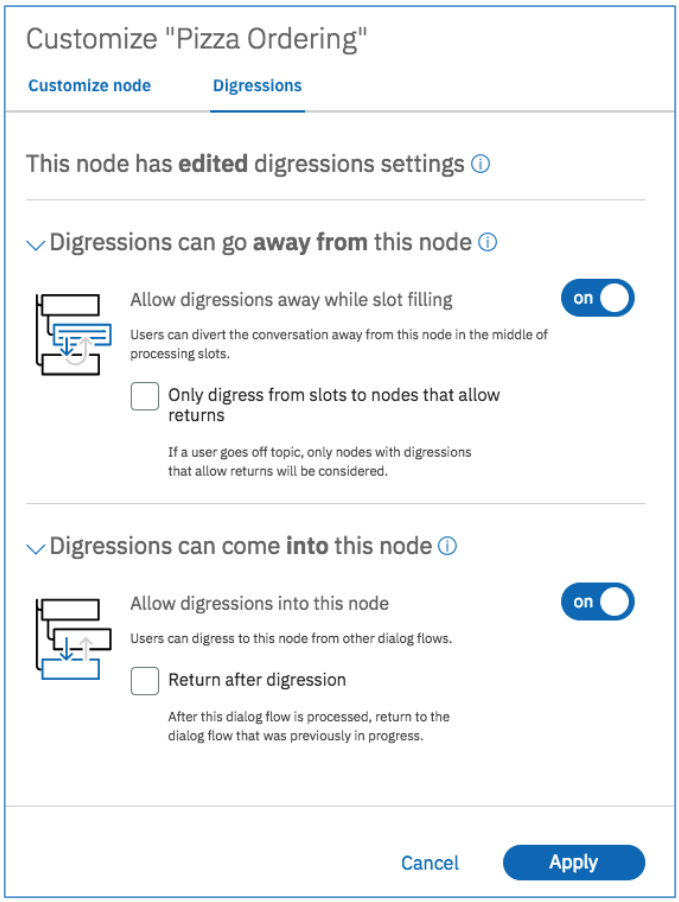

This is the default settings :

Digression cannot go away from this node

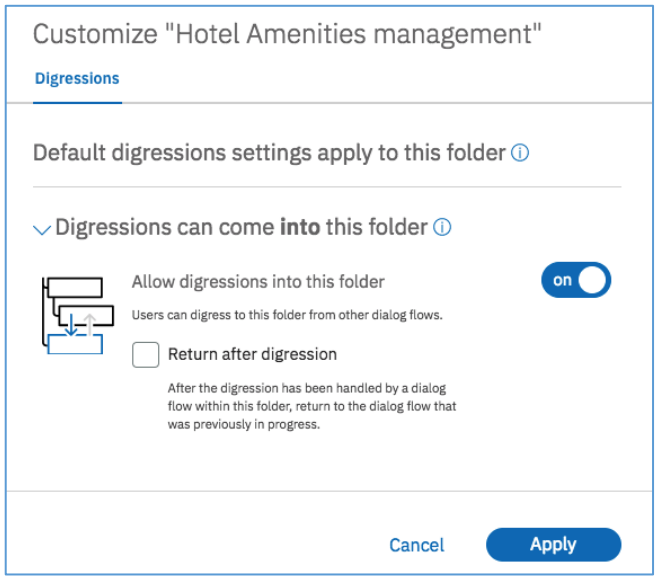
Digression can come into this node

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

6. Enable this option go away from this node, don't update the second option




7. Click **Apply**
8. Select **Hotel Amenities management** folder
9. Click on **Customize** button

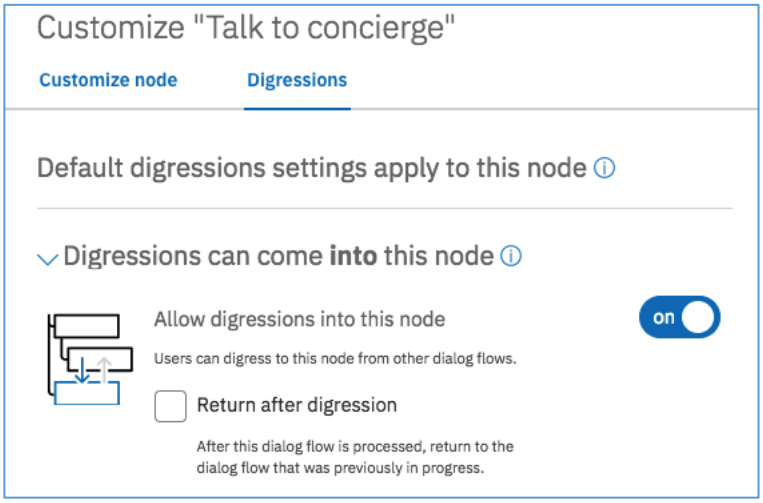


10. Select the option **Return after digression**
11. Click **Apply**

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

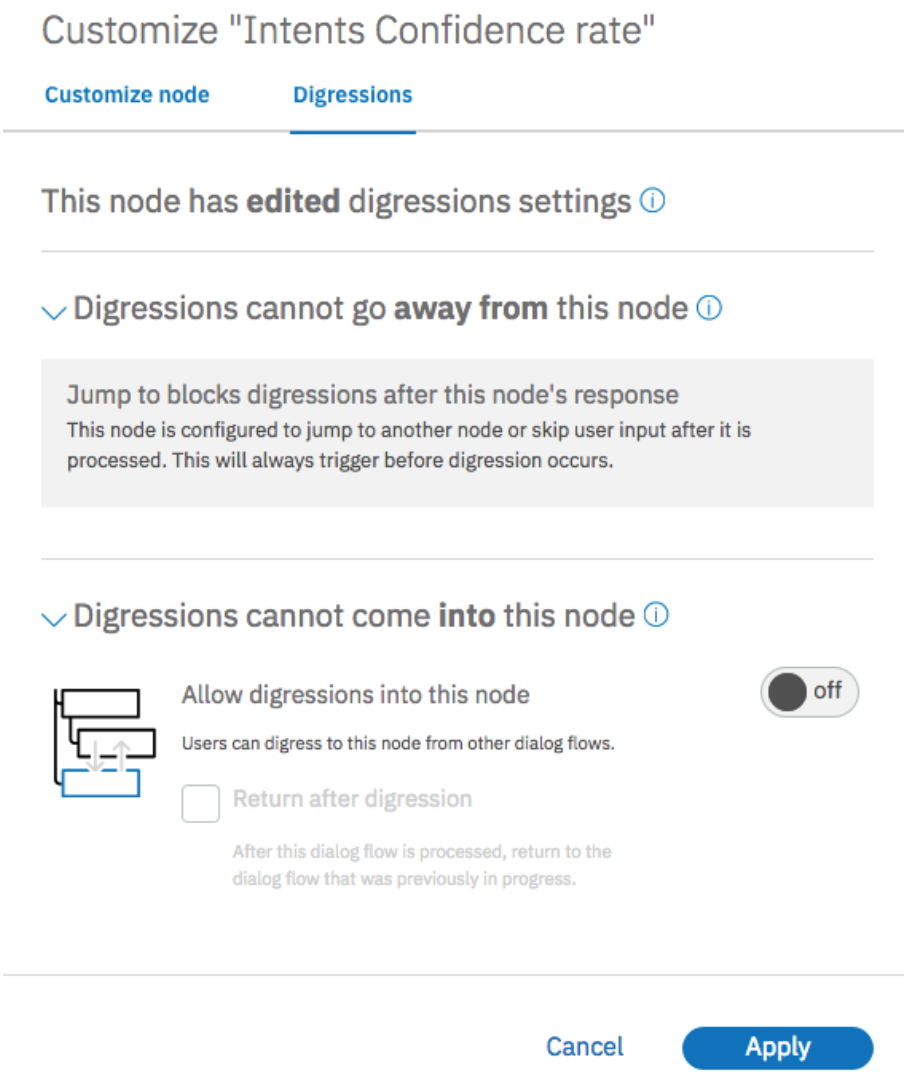
    Now, you are going to create **talk to concierge** node which requires to not return after digression.
12. Select **talk to concierge** node.
13. Click **Customize**, then go to **Digressions** tab

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



14. Select **Intents Confidence rate** node.
15. Click **Customize**, then go to **Digressions** tab

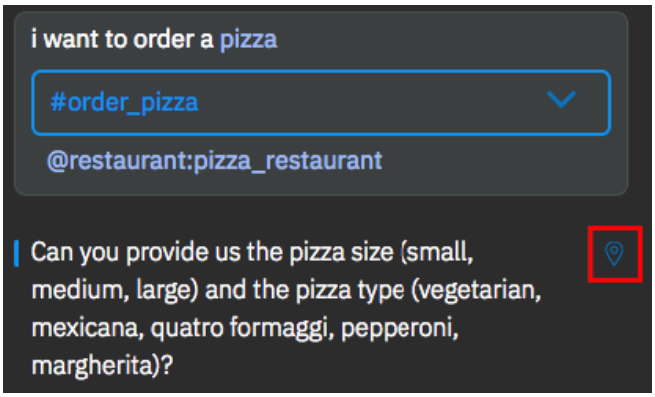
    As this is a technical node, we don't want any digression from or to this node. We switch off the option come into this
16. Turn off the second option




17. Click **Apply**
18. Repeat the previous steps to disable the digression for the **Anything else** node.

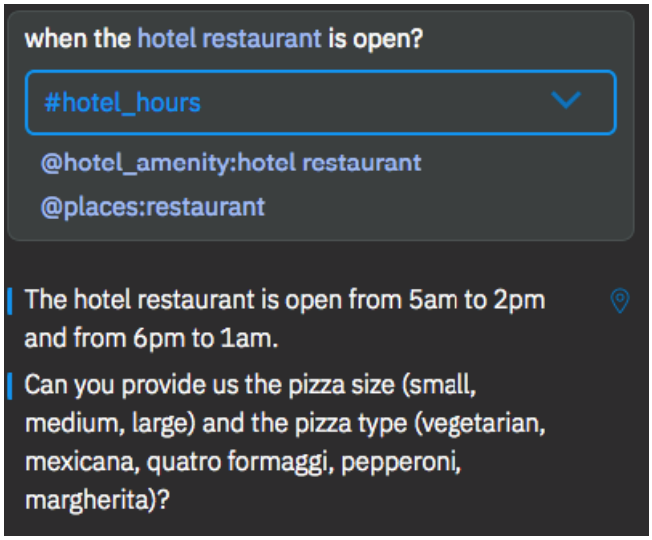
## 17. Test your digressions

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 to the answer)



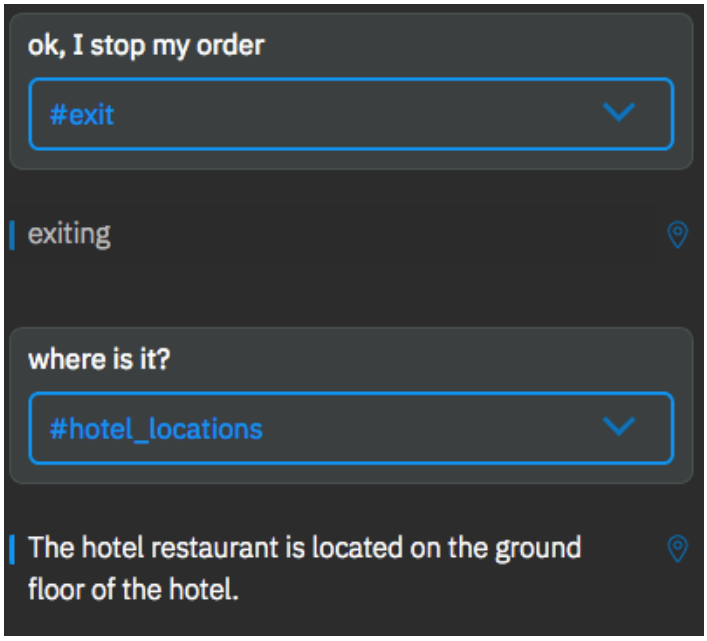

The Pizza Ordering node is highlighted, which was expected/

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




The bot digresses away from the **Pizza Ordering** node to process the **Hotel Hours** node. The service then returns to the **Pizza Ordering** node, and prompts you again for the size of pizza.

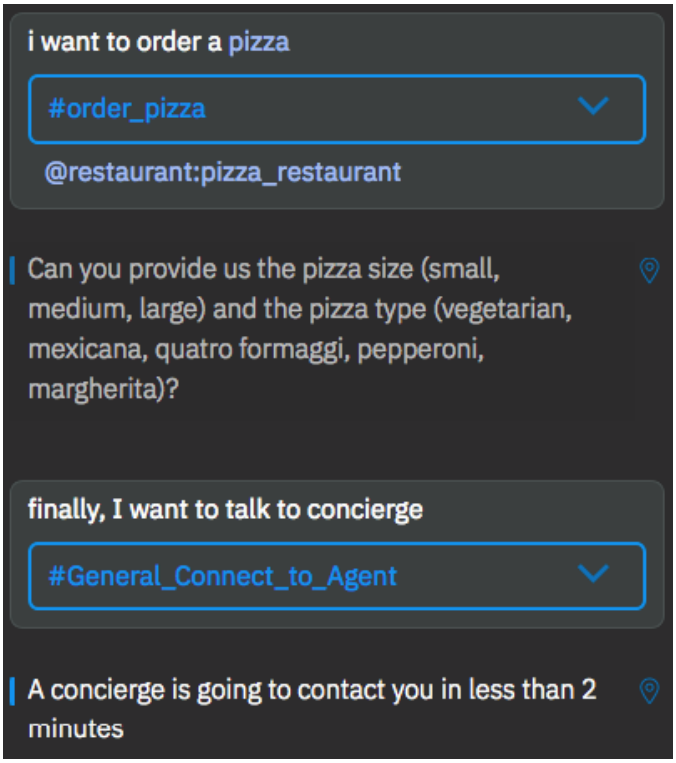
5. Enter : *ok, I stop my order* to conclude the ordering
6. Enter : *where is it?* to illustrate that the service kept the context.




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

   *I want to order a pizza*

   *finally, I want to talk to concierge*



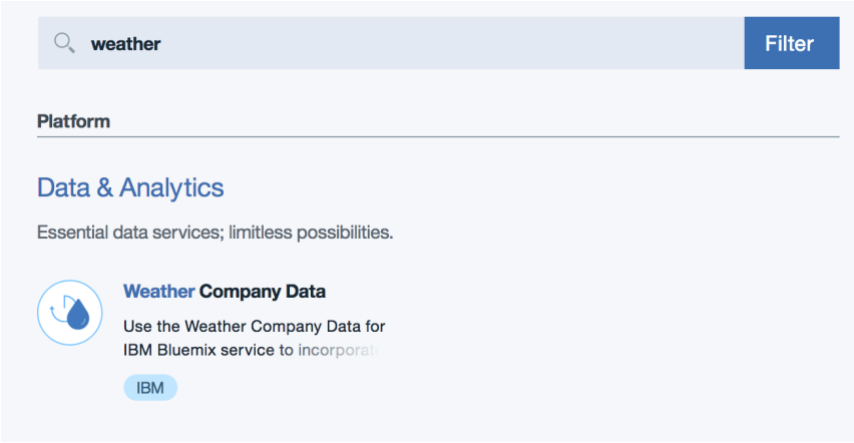

The bot digresses away from the **Pizza Ordering** node to process the T**alk to concierge** node and not returns to the **Pizza Ordering** node.

# Serverless Conversation

The objective of the section is to define actions that can make programmatic calls to external applications or services and get back a result as part of the processing that occurs within a dialog turn.
You can use an external service to validate information that you collected from the user or perform calculations or string manipulations on the input which are too complex to be handled by using supported SpEL expressions and methods. Or you can interact with an external web service to get information, such as an air traffic service to check on a flight's expected arrival time or a weather service to get a forecast. You can even interact with an external application, such as a restaurant reservation site, to complete a simple transaction on the user's behalf.
By today, you will add a new to get information about the weather forecast in Nice. We limited the location in Nice for our lab, but you can get such an information for any city around the world.

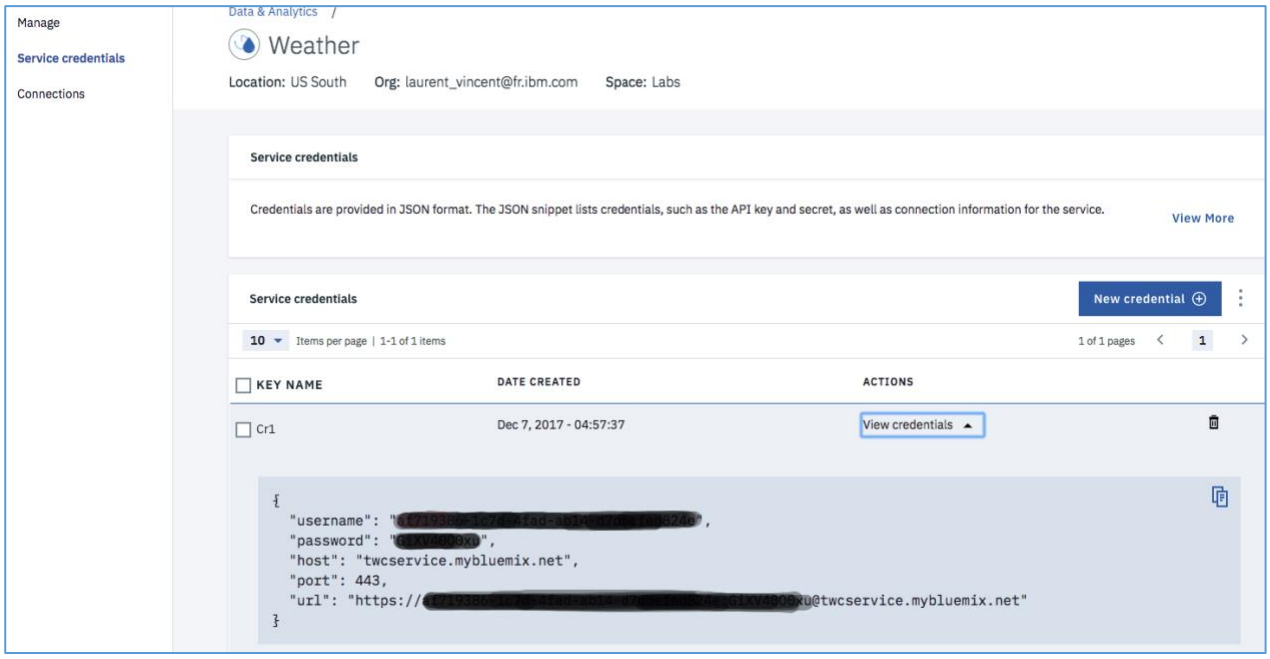
## 18. Create a weather service

1. Go back to your **Dashboard**
2. Click **Create resource**
3. Look for weather service




4. Click **Weather Company Data** tile.
5. Determine a name for your service and Select the region / location used for your Watson Assistant service. It should be *US South* or *Dallas*.
6. Click **Create**

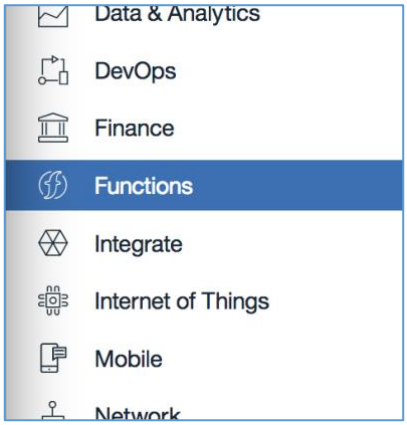
7. Go to the **Service credentials** page
8. Create a new credential and copy **username** and **password** and **host**




## 19. Instanciate a IBM Function using Weather service

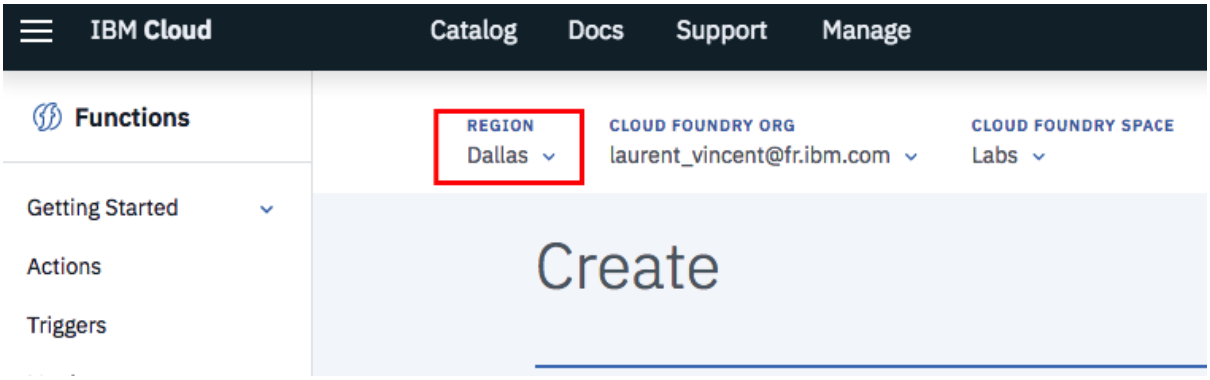
You are going to create the Function called by Watson Conversation. This Function will call the Weather company service.

1. Click on Hamburger menu
2. Click Functions

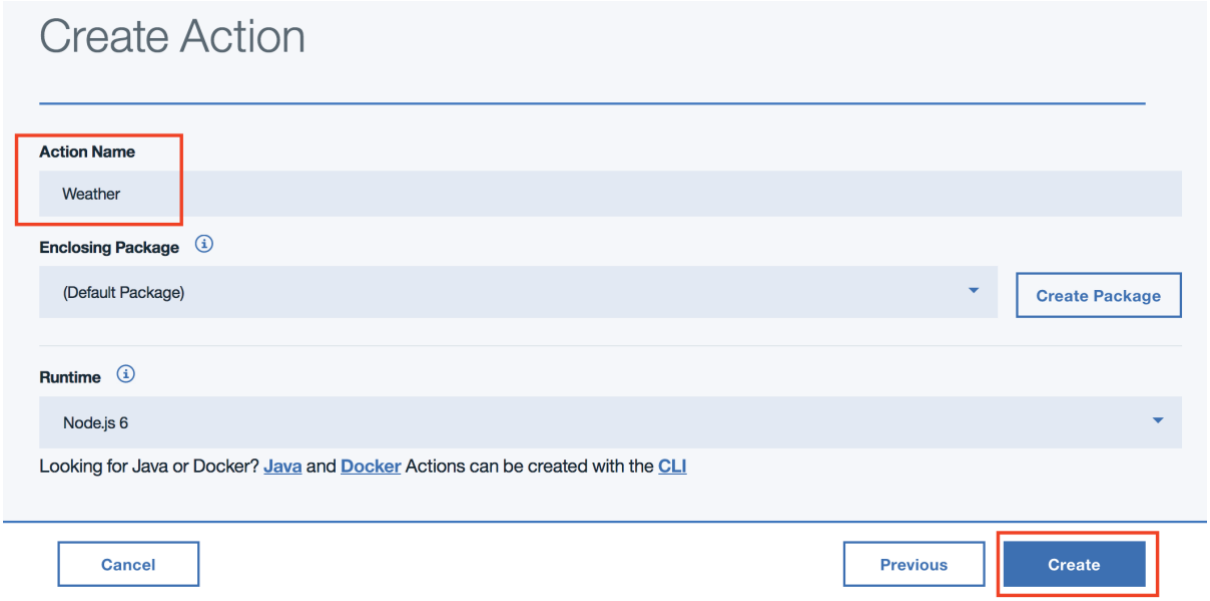



3. Click **Start Creating**

4. Be sure to select the region used by your Watson Assistant




5. Click **Create Action**
6. Enter *Weather* as name, keep *Node.js 6*, click **Create**



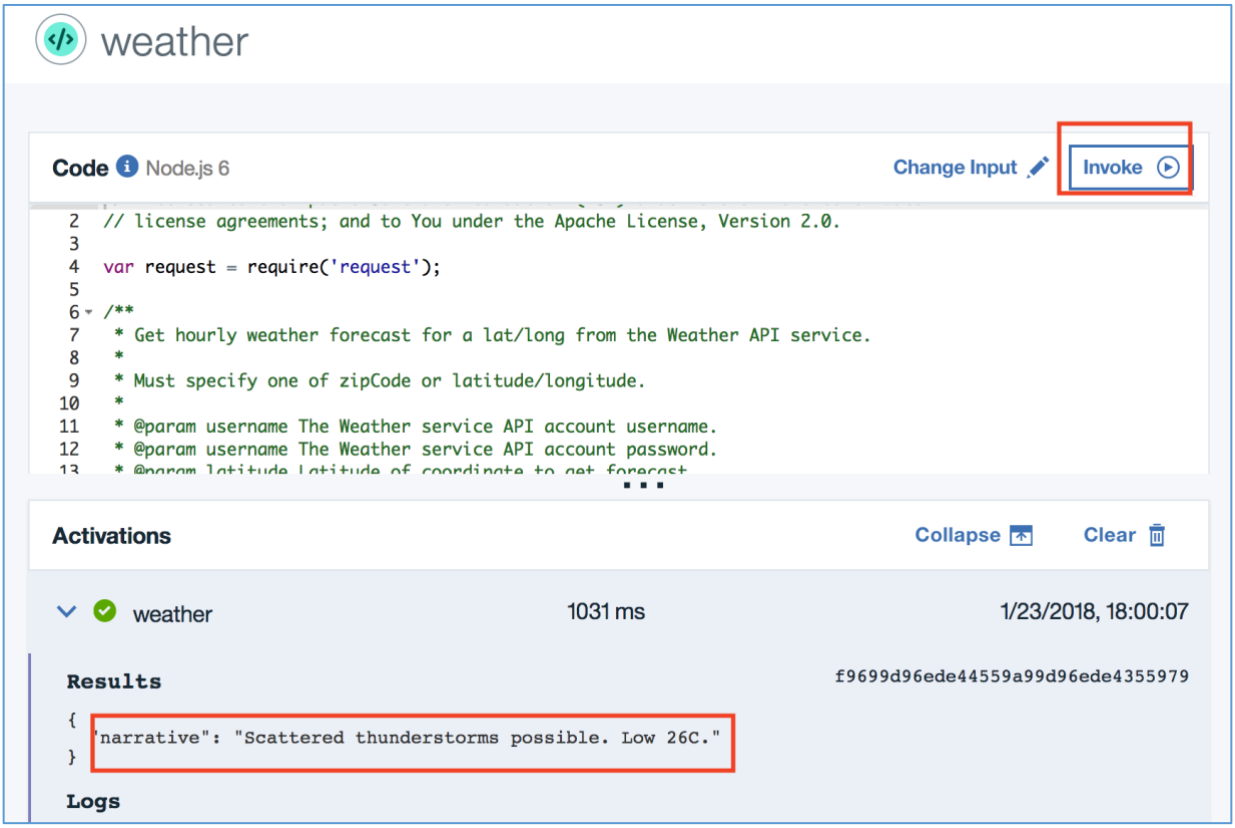
7. Copy / paste the code below into the Code frame of the Weather IBM Cloud Function. (the code can also be copied from the file Weather_Cloud_Function.js)

```
// Licensed to the Apache Software Foundation (ASF) under one or more contributor
// license agreements; and to You under the Apache License, Version 2.0.

var request = require('request');

/**
 * Get hourly weather forecast for a lat/long from the Weather API service.
 *
 * Must specify one of zipCode or latitude/longitude.
 *
 * @param username The Weather service API account username.
 * @param username The Weather service API account password.
 * @param latitude Latitude of coordinate to get forecast.
 * @param longitude Longitude of coordinate to get forecast.
 * @param zipCode ZIP code of desired forecast.
 * @return The hourly forecast for the lat/long.
 */

function main(params) {
   console.log('input params:', params);
   var username = params.username || '<user name>';
   var password = params.password  || '<password>';
   var lat = params.latitude || '43.659';
   var lon = params.longitude ||  '7.192';
   var language = params.language || 'en-US';
   var units = params.units || 'm';
   var timePeriod = params.timePeriod || '10day';
   var host = params.host || '<host>';
   var url = 'https://' + host + '/api/weather/v1/geocode/' + lat + '/' + lon;
   var qs = {language: language, units: units};

   switch(timePeriod) {
      case '48hour':
         url += '/forecast/hourly/48hour.json';
         break;
      case 'current':
         url += '/observations.json';
         break;
      case 'timeseries':
         url += '/observations/timeseries.json';
         qs.hours = '23';
         break;
      case '3day':
         url += '/forecast/daily/3day.json';
         qs.hours = '23';
         break;
      default:
         url += '/forecast/daily/10day.json';
         break;
   }
```

```
    console.log('url:', url);

    var promise = new Promise(function(resolve, reject) {
        request({
            url: url,
            qs: qs,
            auth: {username: username, password: password},
            timeout: 30000
        }, function (error, response, body) {
            if (!error && response.statusCode === 200) {
                var j = JSON.parse(body);
                console.log('body:', body);
                console.log('j:', j.forecasts[0].narrative);
                var tmp = { narrative: j.forecasts[0].narrative};
                resolve(tmp);
             //   resolve(j);

            } else {
                console.log('error getting forecast');
                console.log('http status code:', (response || {}).statusCode);
                console.log('error:', error);
                console.log('body:', body);
                reject({
                    error: error,
                    response: response,
                    body: body
                });
            }
        });
    });

    return promise;
}
```

8. Replace *<user name>*, *<password>* and *<host>* in the source code with the username, password and host of your weather service instance.

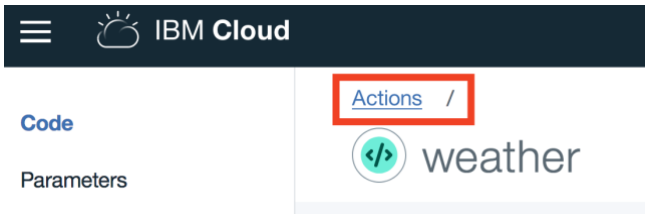
9. To test your action, click **Invoke**



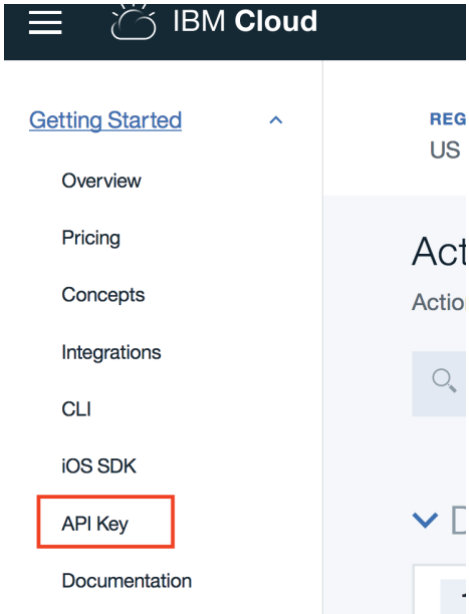

The action should return the current weather in Nice. You can imagine to provide the Hotel location to get the local weather forecast.

## 20. Get IBM Function credential

1. On top of the page click **Actions** /




2. Expand **Getting Started** menu and click **API Key**




3. Click **copy** icon


4. Paste it in any text editor

   You should get something like this:

   <Function User ID>**:**<Function Password>

   The User ID is all characters before **:**

   The Password is all characters after **:**

5. Copy also the **Current Namespace**

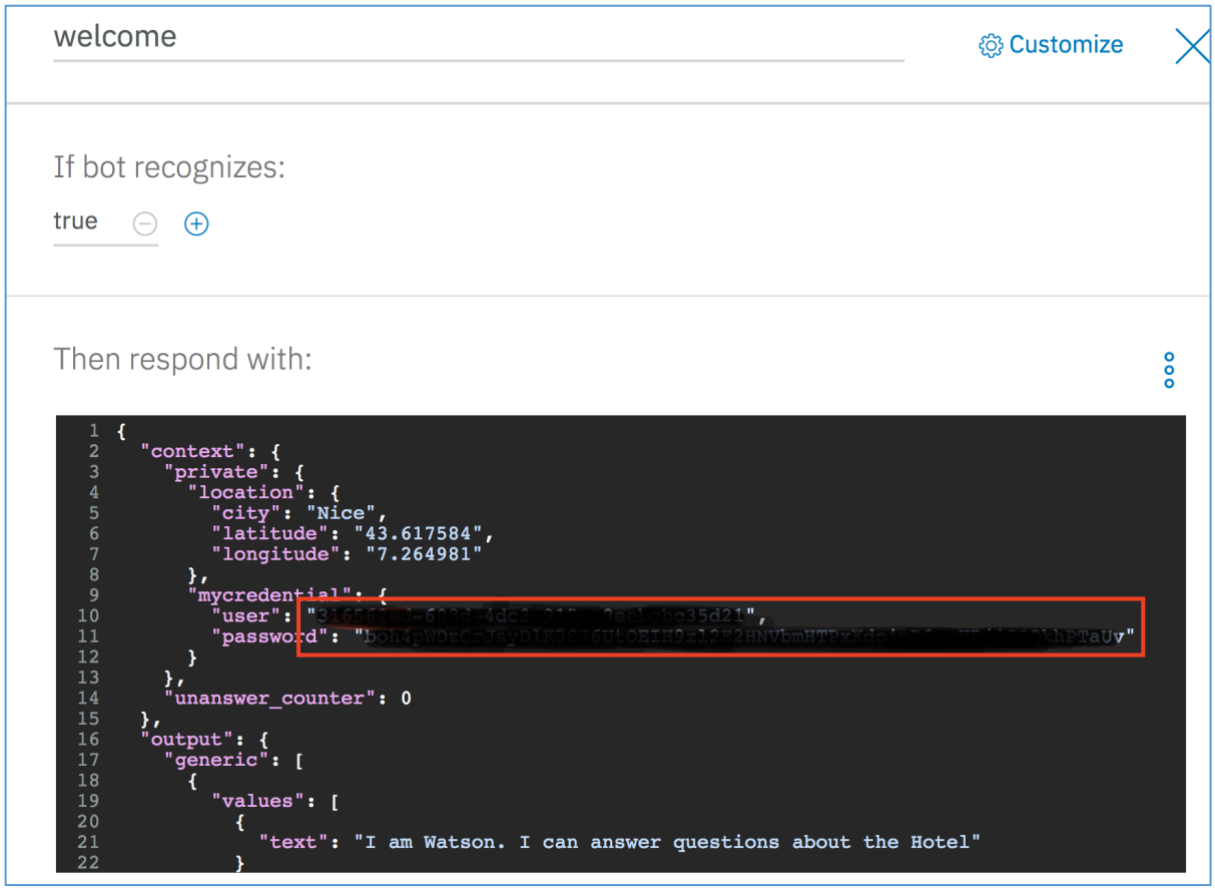
## 21. Update the Welcome statement

You are going to add in the Welcome statement information about the weather forecast.

1. Go back to Watson Assistant user interface
2. Select and expand **Start conversation** node
3. Select **welcome** node to edit it
4. Open the **Json editor** and update in the Context variable : *private.mycredential*

   The Context value : *{"user":"<Function user ID>","password":"<Function Password>"}*

   User and password must be replaced with your IBM Cloud Functions credentials




In the real implementation the credential must be manage by the Client Application which orchestrate the conversation. So this node is useless in this case.

## 22. Create the weather branch

We are going to create nodes to leverage the Weather forecast provided by the weather company service via IBM Cloud Function.

1. Select the  anything else node and Add a node above and fill it this

   Name : *Call Weather Function*

   Condition : *#weather*

2. Open the JSON editor
3. Copy Paste the code below

```
{
  "output": {},
  "actions": [
    {
      "name": "/<Your name space>/Weather",
      "type": "cloud_function",
      "parameters": {
        "latitude": "$private.location.latitude",
        "longitude": "$private.location.longitude",
        "timePeriod": "10day"
      },
      "credentials": "$private.mycredential",
      "result_variable": "context.weather"
    }
  ]
}
```

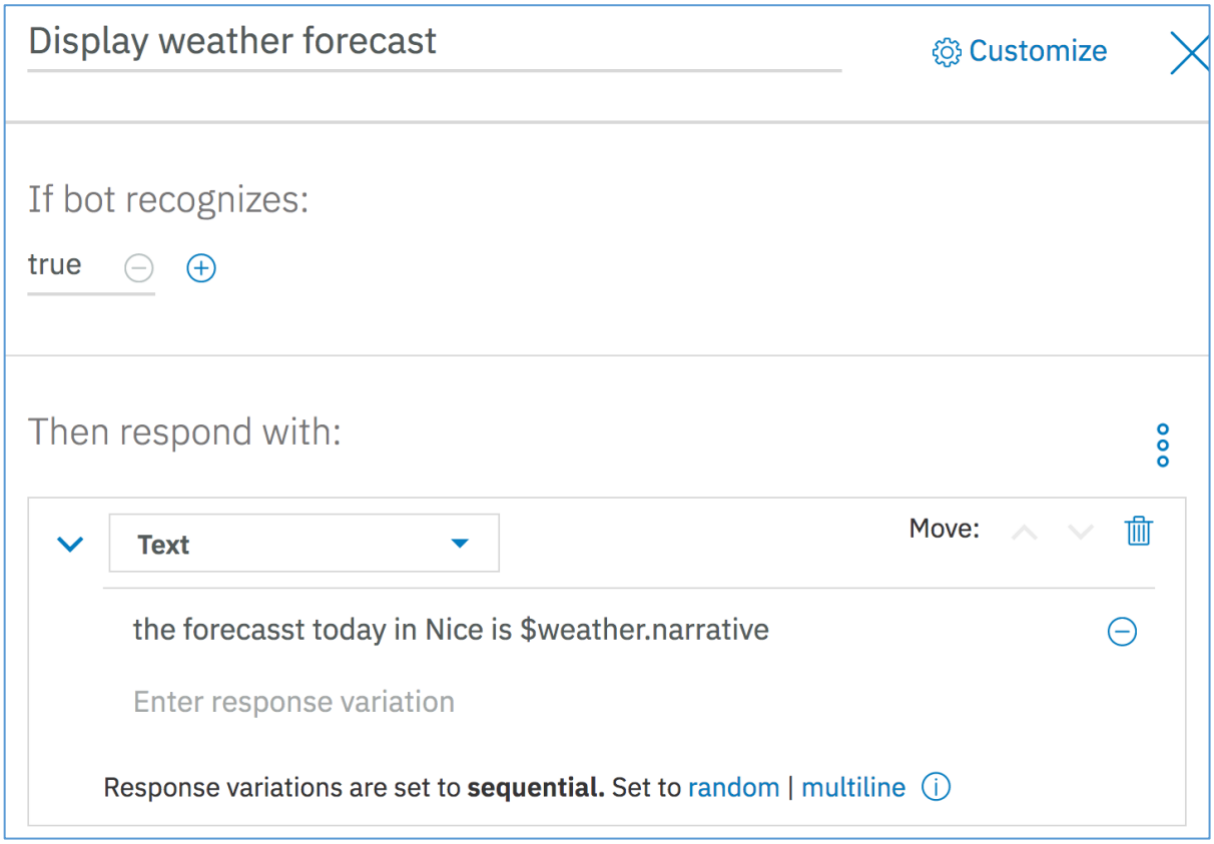
4. Replace the name space with yours

Then respond with:

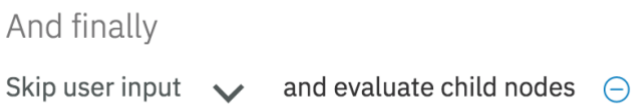
```
1  {
2    "output": {},
3    "actions": [
4      {
5        "name": "/laurent_vincent@fr.ibm.com_Labs/weather",
6        "type": "cloud_function",
7        "parameters": {
8          "latitude": "$private.location.latitude",
9          "longitude": "$private.location.longitude",
10         "timePeriod": "10day"
11       },
12       "credentials": "$private.mycredential",
13       "result_variable": "context.weather"
14     }
15   ]
16 }
```

5. Add a child to **Call Weather Function** node and fill it like this:

   Name: *Display weather forecast*
   condition: *true*
   response: *The forecast today in Nice is $weather.narrative*

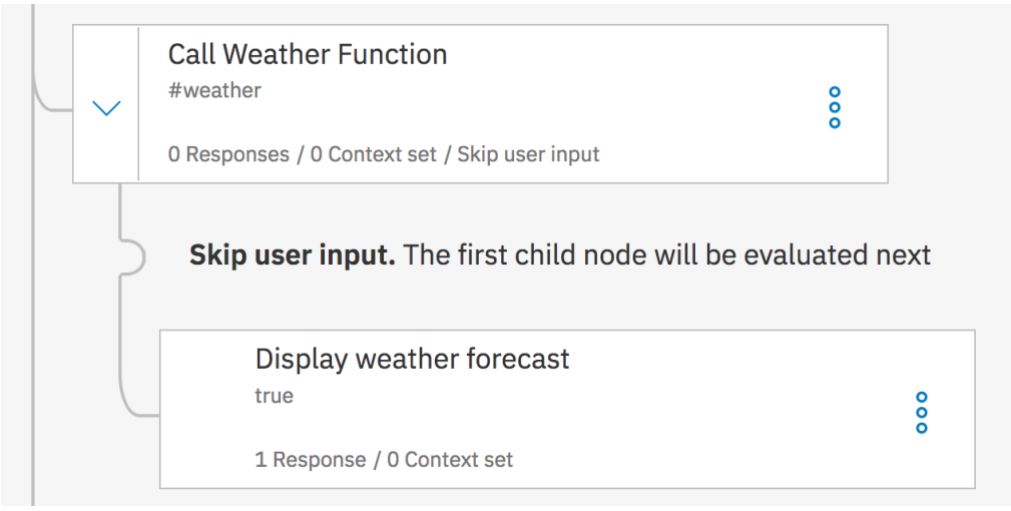



6. Return to **Call Weather Function** node and select the option **Skip user input**




7. Close the node editor
8. Return to **Call Weather Function** node and select the option **Skip user input**
9. Close the node editor

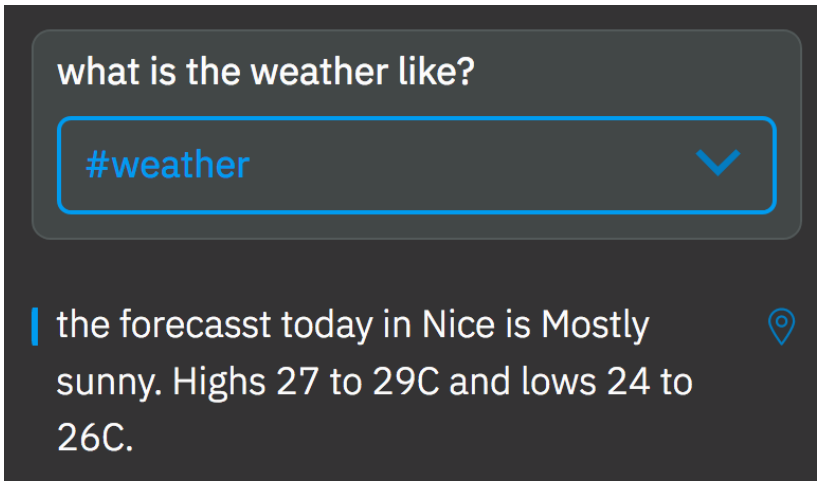
The final Weather branch should look like this:




## 23. Test your serverless conversation

1. Open **try it out** frame
2. Enter *What is the weather like?*

   The service should display the weather forecast:



The final dialog should look like this: