Session 7: Integrating

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 Conversation 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

This chapter describes how to integrate a chatbot application quickly without coding and integrate it with the Watson Assistant service. For this use case example, you continue with your chatbot, however you can customize the chatbot to take any other role such as delivery service, Q&A, student assistant, and more.

To create the chatbot application, you use the Node-RED programming tool. With this powerful tool you can create, edit, and deploy applications quickly. Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click.

Node-RED, created by IBM but now part of JS Foundation, provides full integration with Watson APIs, allowing you to make great applications quickly and easy.

The following topics are covered in this chapter:

- Getting started
- Architecture
- Step-by-step implementation
- Quick deployment of application

3. Prerequisites

Before you start the exercises in this guide, you will need to complete the session building a dialog.

Bluemix 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/

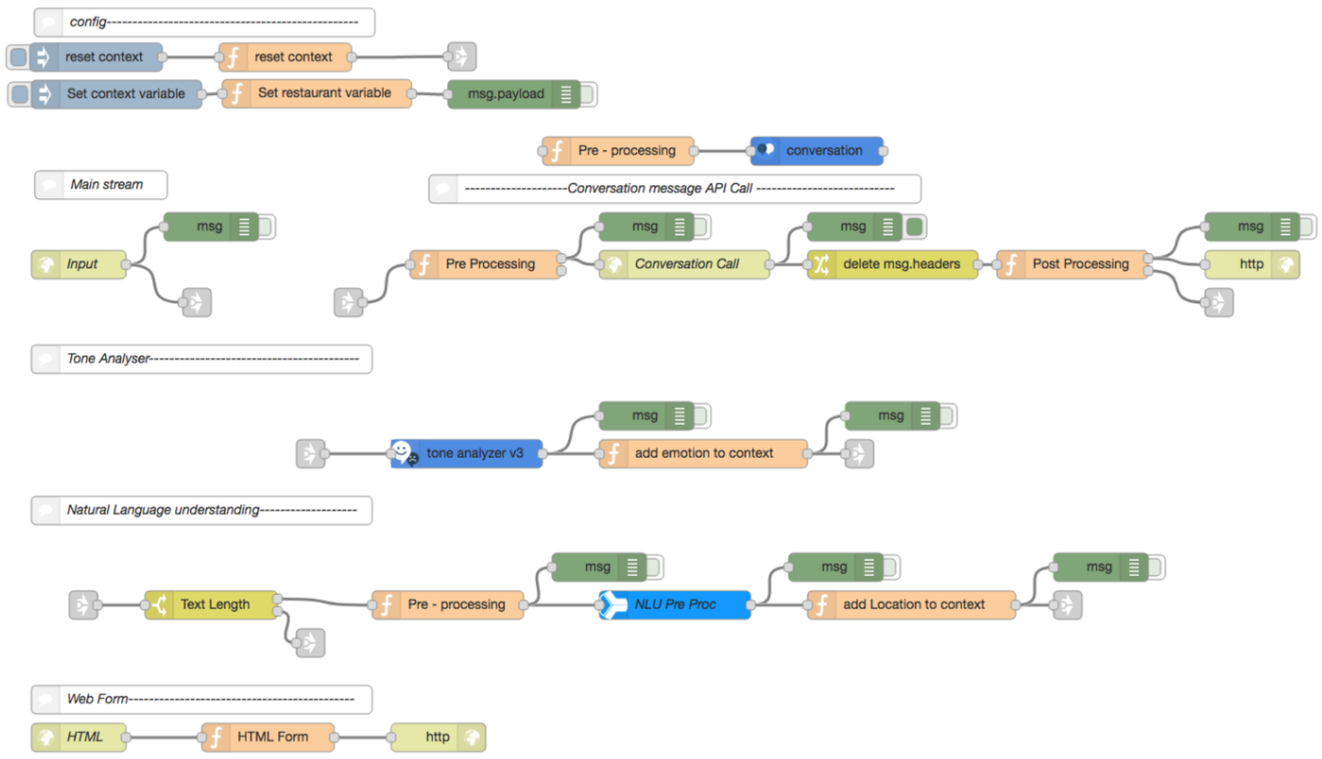
4. Scenario

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

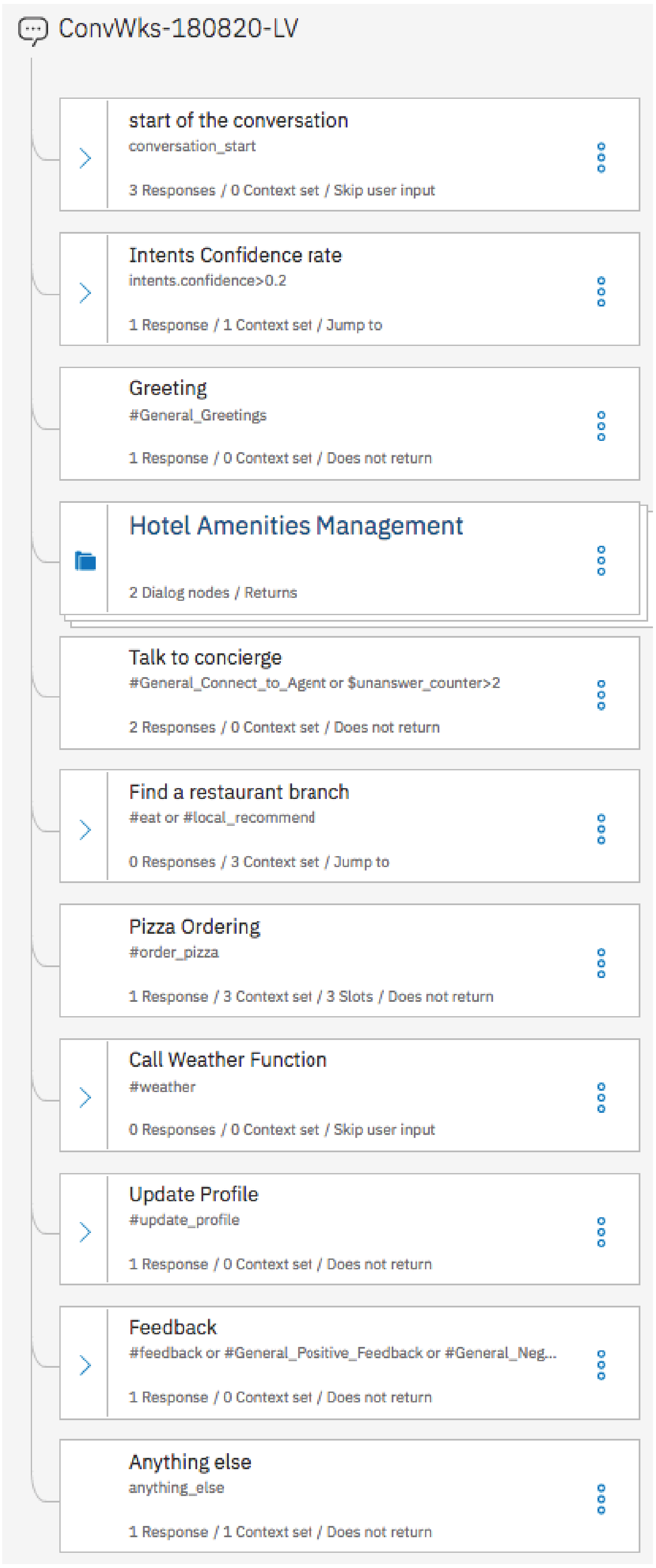
End-users: Hotel customers

5. What to expect when you are done

At the end of session, you should have a Node-Red flow to simulate your chatbot application.



and following dialog



ConvWks-180820-LV

start of the conversation
conversation_start
3 Responses / 0 Context set / Skip user input

Intents Confidence rate
intents.confidence>0.2
1 Response / 1 Context set / Jump to

Greeting
#General_Greetings
1 Response / 0 Context set / Does not return

Hotel Amenities Management
2 Dialog nodes / Returns

Talk to concierge
#General_Connect_to_Agent or $unanswer_counter>2
2 Responses / 0 Context set / Does not return

Find a restaurant branch
#eat or #local_recommend
0 Responses / 3 Context set / Jump to

Pizza Ordering
#order_pizza
1 Response / 3 Context set / 3 Slots / Does not return

Call Weather Function
#weather
0 Responses / 0 Context set / Skip user input

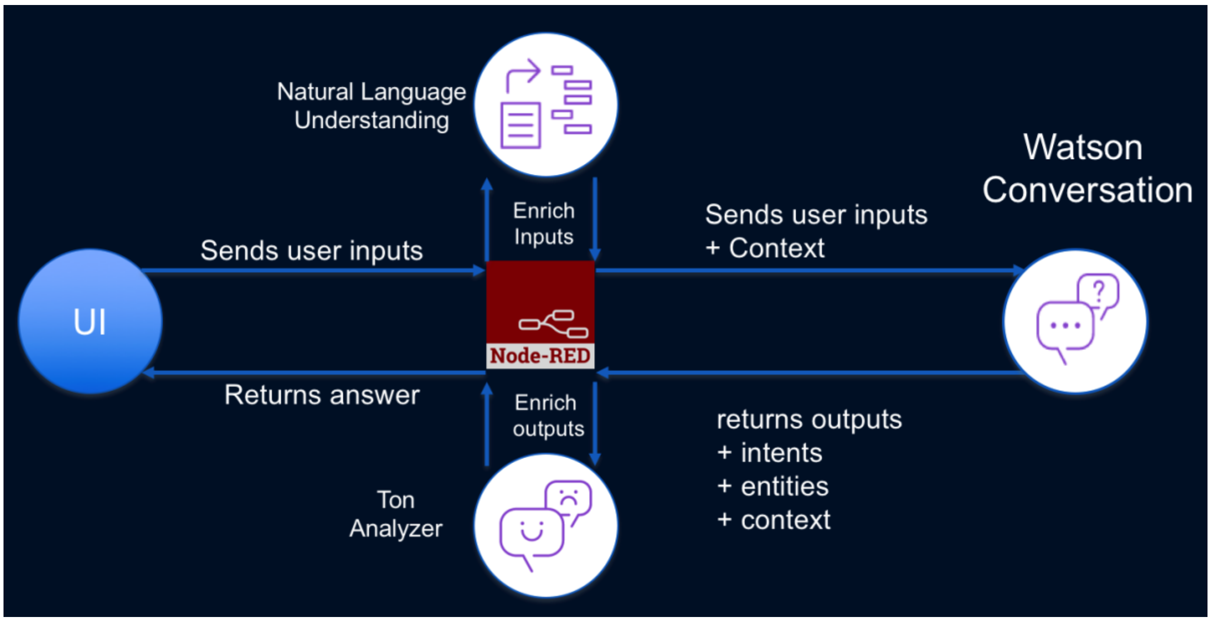
Update Profile
#update_profile
1 Response / 0 Context set / Does not return

Feedback
#feedback or #General_Positive_Feedback or #General_Neg...
1 Response / 0 Context set / Does not return

Anything else
anything_else
1 Response / 1 Context set / Does not return

Architecture

6. Overview


Natural Language Understanding
Watson Conversation
Enrich Inputs
Sends user inputs + Context
Sends user inputs
UI
Node-RED
Returns answer
Enrich outputs
returns outputs + intents + entities + context
Ton Analyzer

Notice that the flow shown in the figure represents one loop of a conversation, therefore this cycle repeats several times during a conversation:

1. The user sends a message to the web front-end (chatservice).
2. The chat service (for example, Slack, Facebook Messenger, webapp) determines whether the message is for the Assistant chatbot application. If the message is for the chatbot, then the chat service sends the message to your chatbot application (Node-RED).
3. Your application parses the message and sends the filtered message to the Watson Conversation service for processing.
4. The Watson Conversation service processes the message and provides a response.
5. The response is received and filtered by your application, which then sends the response to the chat service.
6. The chat service identifies that the input share from the Assistant chatbot and presents the message as a response from the chatbot to the user.

7. Project structure

These are the components you use in this use case:

- A Node-RED instance that is created in Bluemix, which is cloud-based, so installing software is not necessary
- A Watson Conversation service instance
- Watson Tone Analyser and Natural Language Understanding services

Chatbot application in Node-Red

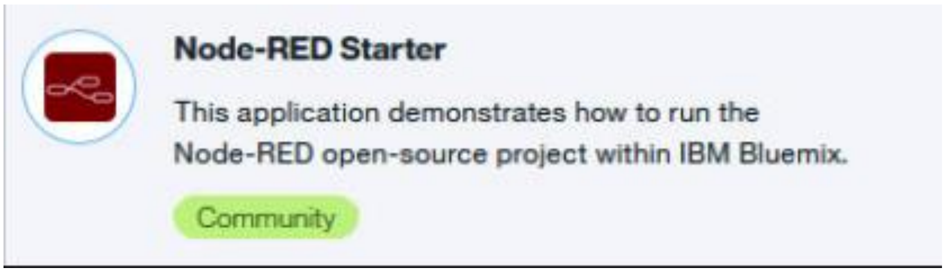
Node-RED is a useful tool to create applications without having to write code. Instead, it uses simple visual components that you configure and connect.

To make this task even easier, you do not need to install Node-RED, because it is available in Bluemix. In this section, you create a Node-RED application Bluemix and configure the flow.

- Create the Node-RED application in Bluemix
- Create the chatbot application flow with the Node-RED flow editor
- Configure the chatbot application in Node-RED

8. Create the Node-Red application in Bluemix

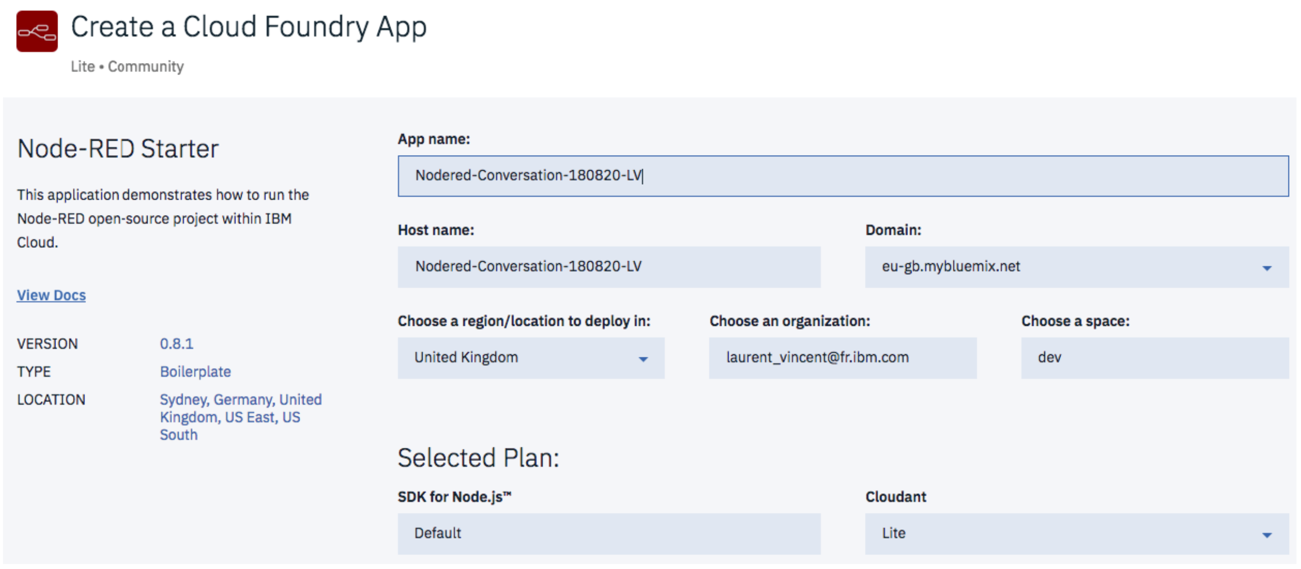
1. Go to the IBM Cloud Catalog.
2. In the **Catalog**, go to > **Starter Kits** and click on **Node-Red Starter**.



Node-RED Starter

This application demonstrates how to run the Node-RED open-source project within IBM Bluemix.

3. Enter the name of your application and host as *Nodered-Conversation-XXXX-YY*. Replace XXX with the date of the day and YY with your initials. Accept the default values for the remaining fields and click on **Create**



Create a Cloud Foundry App

Lite • Community

Node-RED Starter

This application demonstrates how to run the Node-RED open-source project within IBM Cloud.

View Docs

VERSION	0.8.1
TYPE	Boilerplate
LOCATION	Sydney, Germany, United Kingdom, US East, US South

App name: Nodered-Conversation-180820-LV

Host name: Nodered-Conversation-180820-LV

Domain: eu-gb.mybluemix.net

Choose a region/location to deploy in: United Kingdom

Choose an organization: laurent_vincent@fr.ibm.com

Choose a space: dev

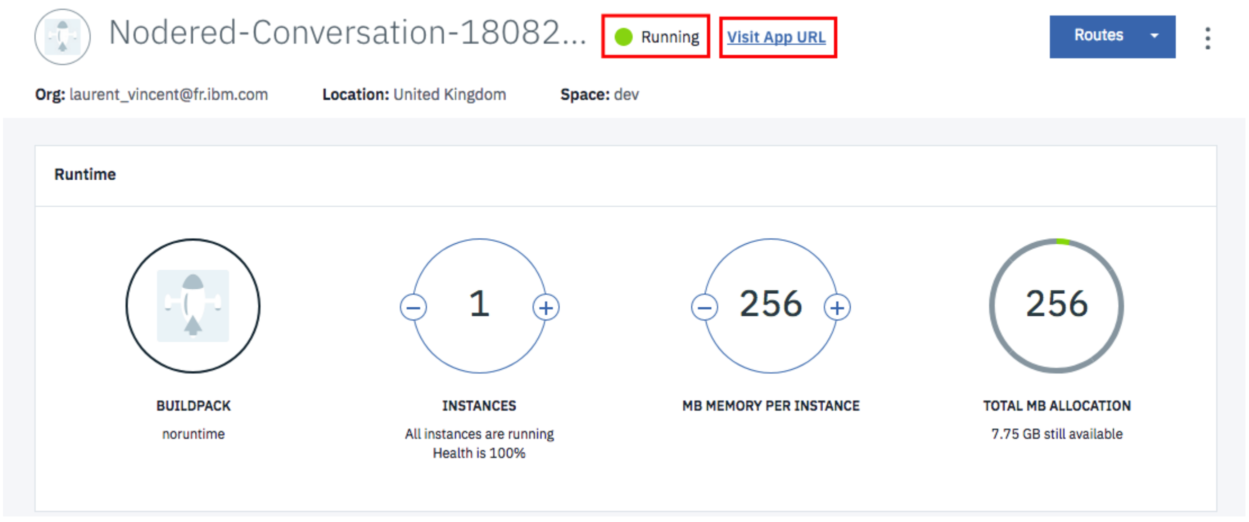
Selected Plan:

SDK for Node.js™: Default

Cloudant: Lite

Note: Wait until the application is created and it is started. The application status should be Running before you can proceed.

4. Once the application is Running, Click **Visit App URL** (on the right of the status)


Nodered-Conversation-18082... Running Visit App URL

Org: laurent_vincent@fr.ibm.com Location: United Kingdom Space: dev

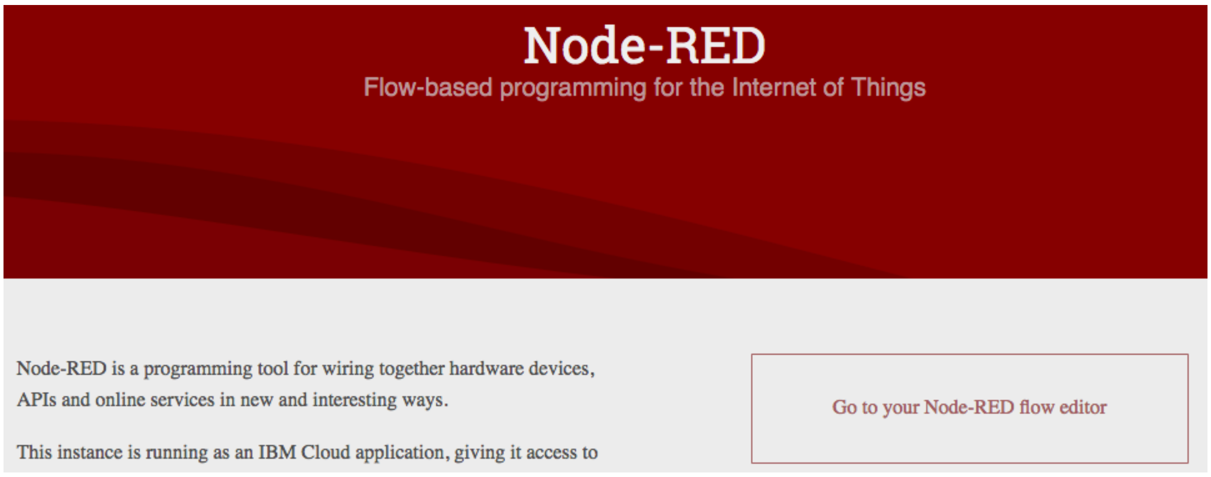
Runtime

BUILDPACK	1 INSTANCES	256 MB MEMORY PER INSTANCE	256 TOTAL MB ALLOCATION
noruntime	All instances are running Health is 100%		7.75 GB still available

5. Click **Next** for each step of the service creation (don't secure your editor)

6. Click **Go to your Node-Red flow Editor**

Note: When you first run this application, you are presented with some options to secure the Node-RED flow editor with a username and password. Securing the editor is optional but it is a good practice to do so. Skip through optional windows for this example until you get to the window below.


Node-RED

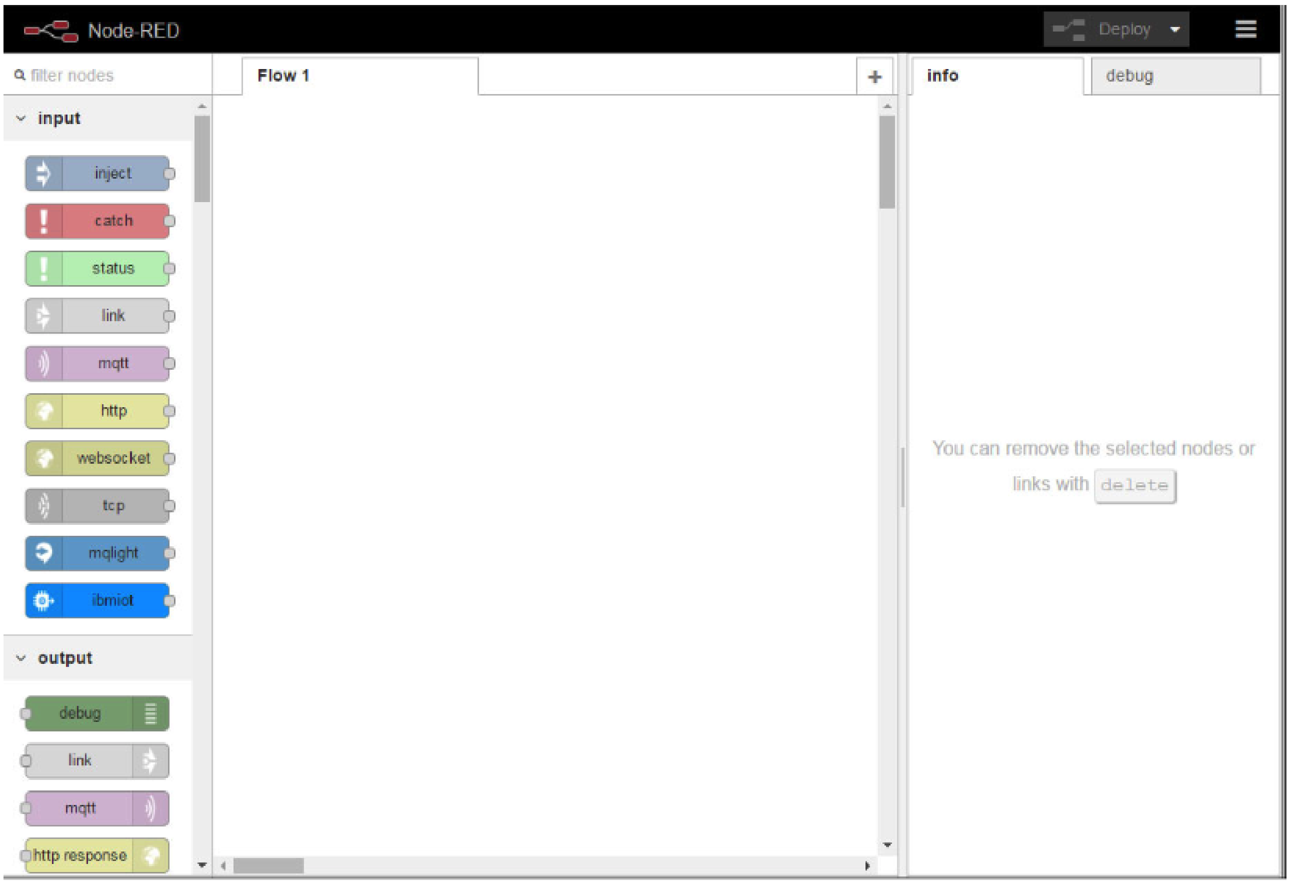
Flow-based programming for the Internet of Things

Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways.

This instance is running as an IBM Cloud application, giving it access to

9. Create the Chatbot application flow with the Node-Red flow editor.

The Node-RED flow editor opens. The panel on the left shows a palette of nodes that you can drag to the workspace. You can connect them together (wire them) to create an application. After dragging a node to a workspace, you can double-click the node to open the Edit (configuration) dialog to provide values for the node.

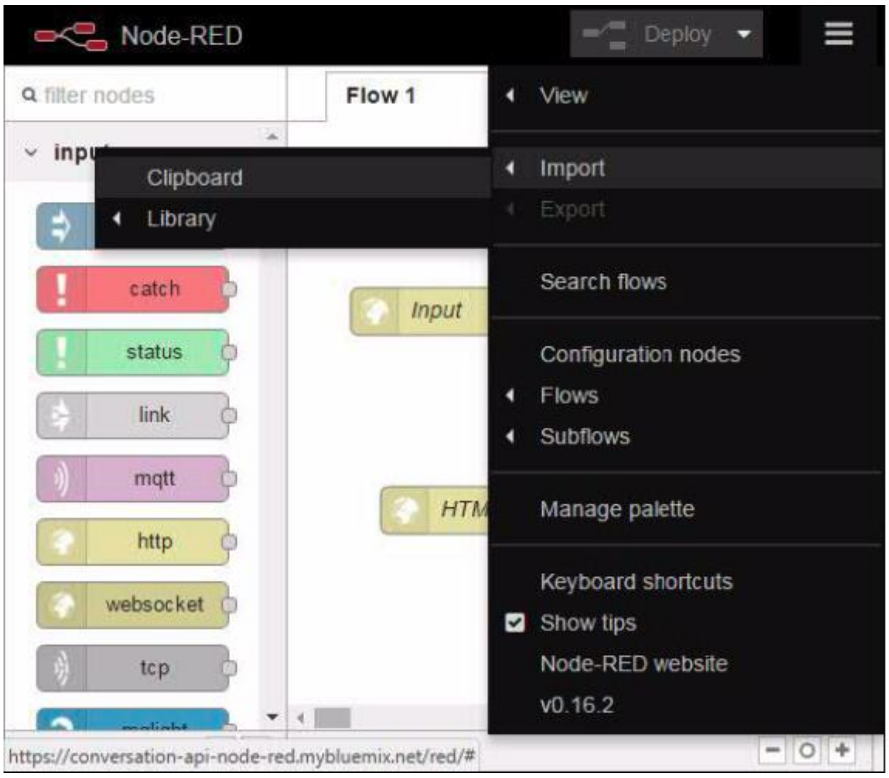


Now you can start to create flows. You use the Node-Red flow editor to add node and Values and create and wire flows.

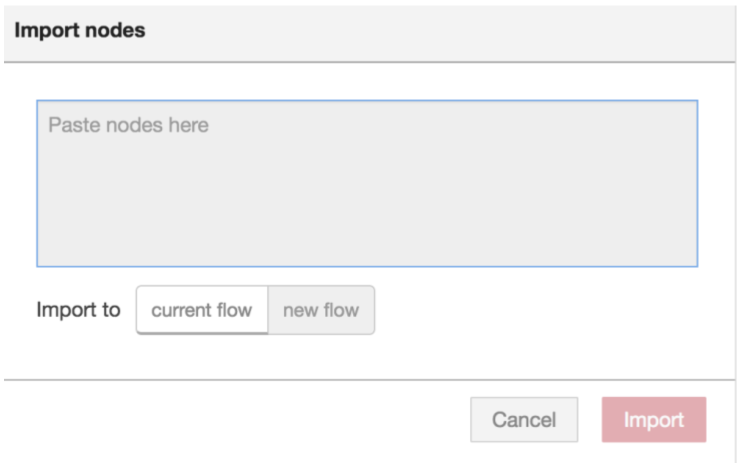
On the IBM box folder, open **conversation-nodered-project** file with any text editor.

1. Copy the content of this files to your clipboard

2. To import the nodes, click hamburger menu at the top-right and select **Import** -> **Clipboard**.



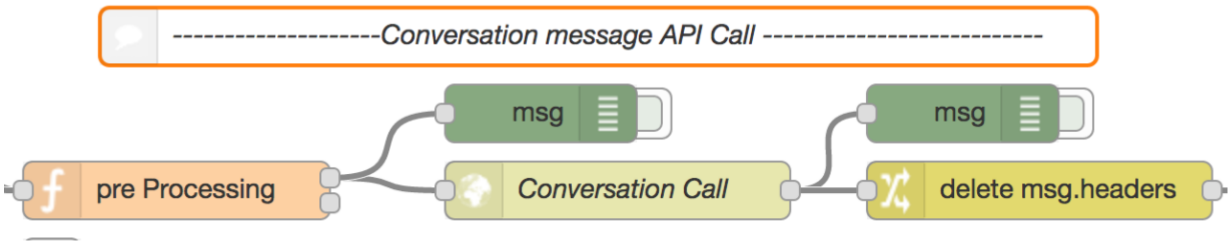
3. Paste your content in the window.



Now your flow is ready to use, you must edit the nodes and add the authentication values based on your Conversation service instance credentials, workspace ID.

10. Configure the Message Conversation API call

Below the nodes which manages the call to your Conversation service:


--------------------Conversation message API Call ---------------------------
msg
msg
pre Processing
Conversation Call
delete msg.headers

1. Double click on **Conversation Call**.

   As describe in the API documentation here

   You are going to use

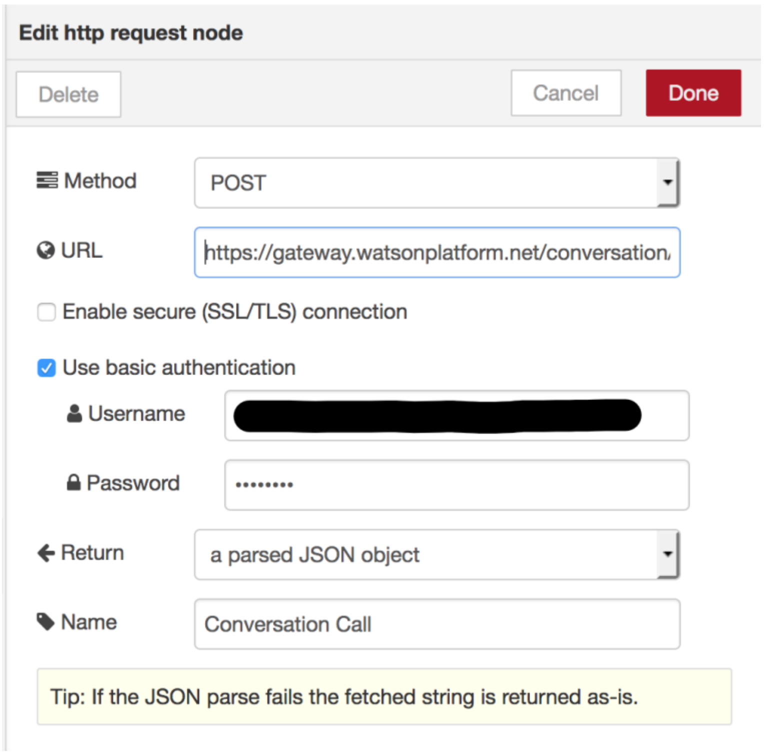
   Method : POST

   URL : https://gateway.watsonplatform.net/assistant/api/v1/workspaces/ ***<WorkspaceID>***/message?version=2018-07-10

   Username : ***apikey***

   Password : ***<API_KEY>***

2. Set the properties according to your inputs, select “Use basic authentication” option.



Method: POST

URL: https://gateway.watsonplatform.net/conversation,

☐ Enable secure (SSL/TLS) connection

☑ Use basic authentication

Username: [redacted]

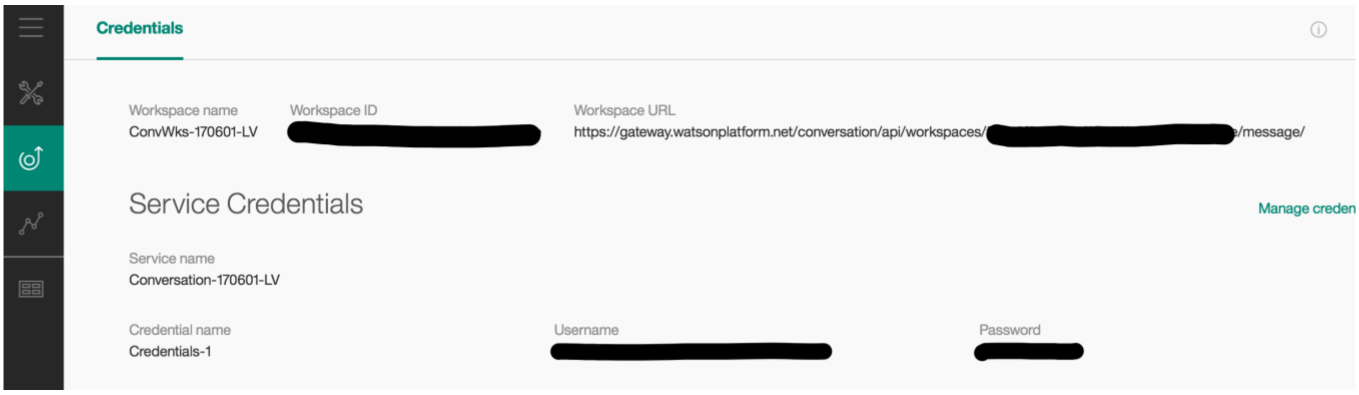
Password: ••••••••

Return: a parsed JSON object

Name: Conversation Call

Tip: If the JSON parse fails the fetched string is returned as-is.

you can retrieve your WorkspaceID, Username and Password on the Credentials page of your Assistant Service.


Credentials

Workspace name
ConvWks-170601-LV

Workspace ID
[redacted]

Workspace URL
https://gateway.watsonplatform.net/conversation/api/workspaces/[redacted]/message/

Service Credentials

Manage creden

Service name
Conversation-170601-LV

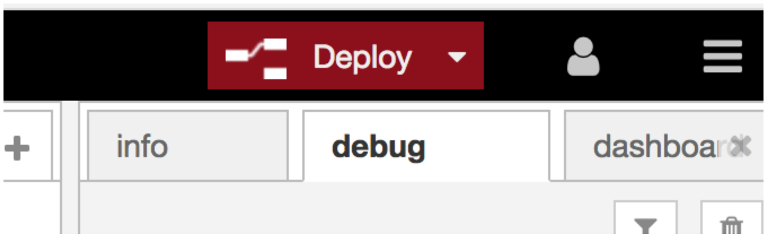
Credential name
Credentials-1

Username
[redacted]

Password
[redacted]

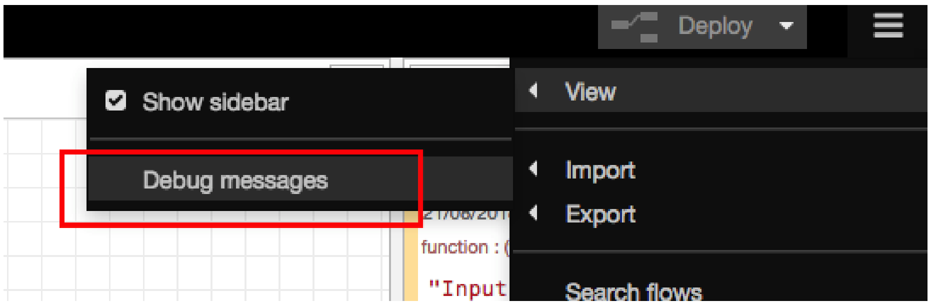
3. Click **Done** to save your node updates (top right)

4. Click **Deploy** to save your flow updates (top right)

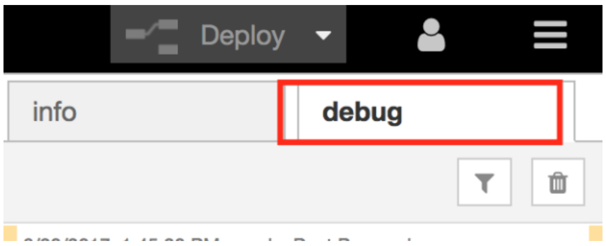


To understand the data exchanged between your chatbot application and Conversation Service, you are going to use the debug pane.

5. Click on Hamburger menu (top right), Select View / Debug messages.

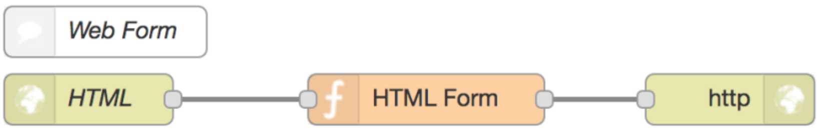


Then the debug is ready to use:

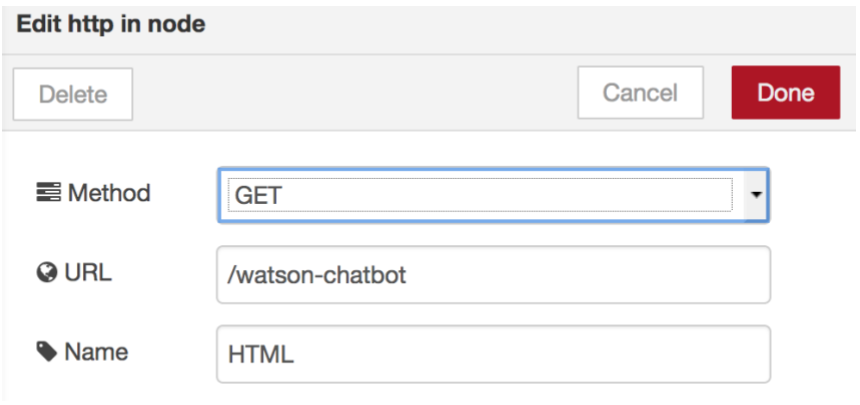


11. Test your settings

For test purpose a HTML page have been embedded in your Node-Red Flow



1. DoubleClick on the HTML node, it gives you the extension of the URL you will use in your browser.



Edit http in node

Delete | Cancel | Done

Method: GET

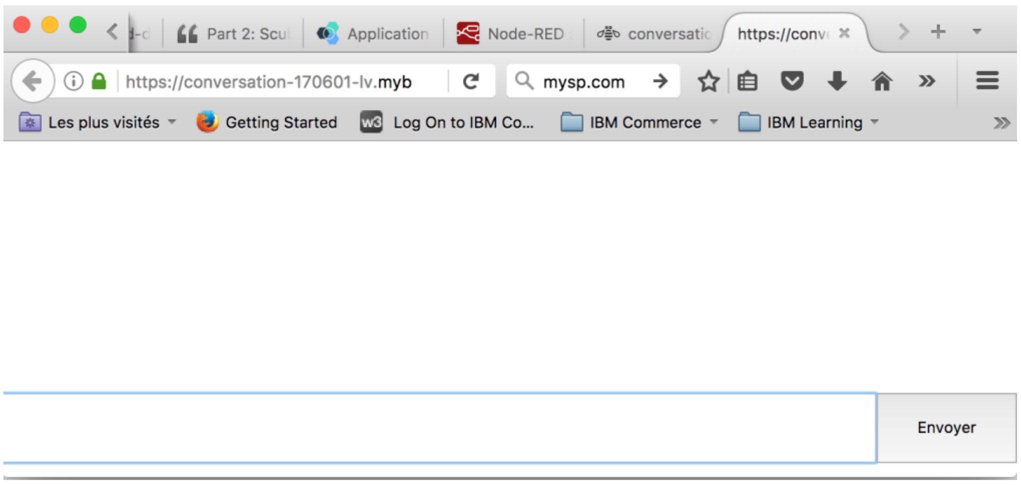
URL: /watson-chatbot

Name: HTML

Note: So the Final URL to access to the chatbot UI should be the aggregation of your node red application and the extension defined in the HTML node: https://<Node-Red_App_Name>.mybluemix.net/watson-chatbot
it means :
https://nodered-conversation-XXXX-YY.eu-gb.mybluemix.net/watson-chatbot

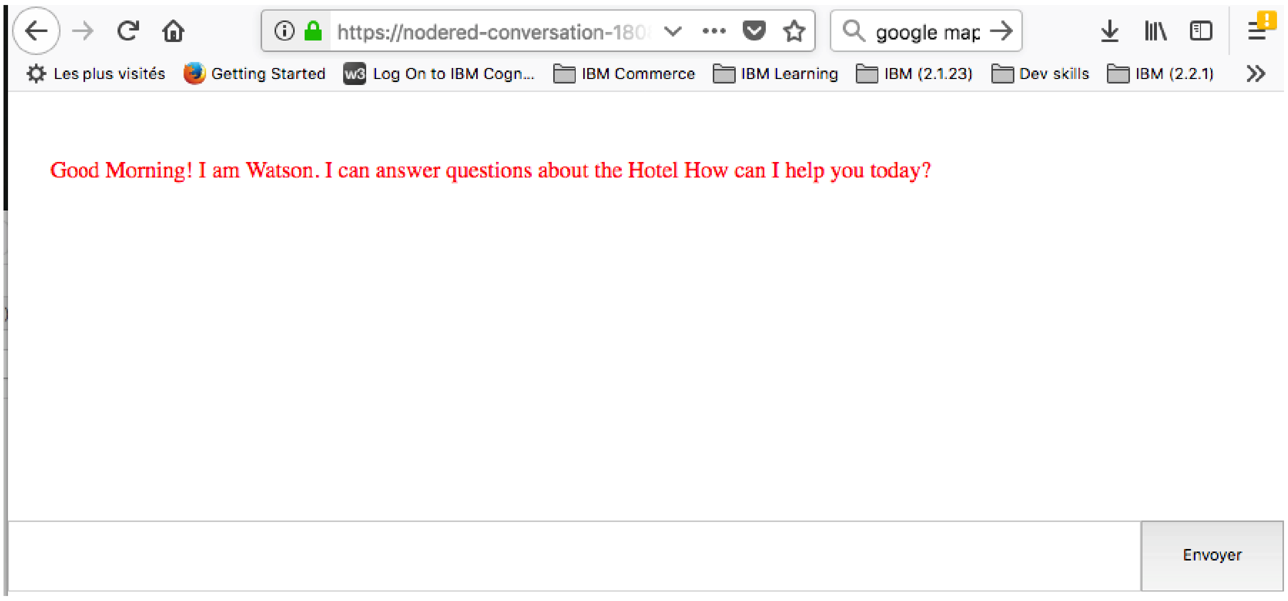
2. Click **Cancel** to close the window

3. Use the URL in your browser to access to a simple page to enter your inputs:



4. Click **Submit**

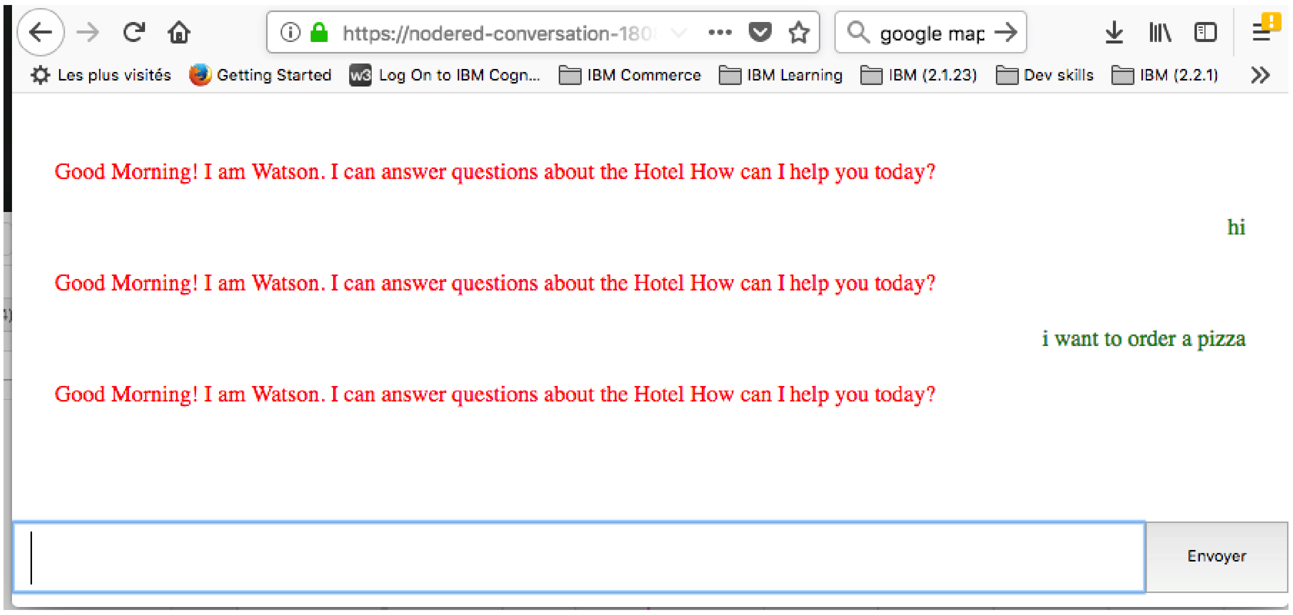
As you don't provide any input, the service returns the "Start of the conversation" responses:



Good Morning! I am Watson. I can answer questions about the Hotel How can I help you today?

5. Enter *Hi* and try to enter some other inputs

   You should get the same responses from the service



Good Morning! I am Watson. I can answer questions about the Hotel How can I help you today?

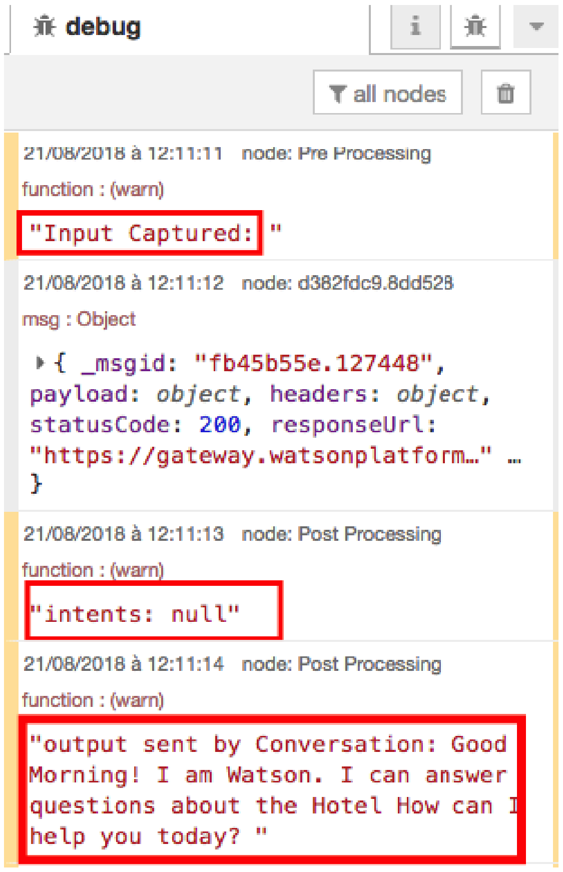
hi

Good Morning! I am Watson. I can answer questions about the Hotel How can I help you today?

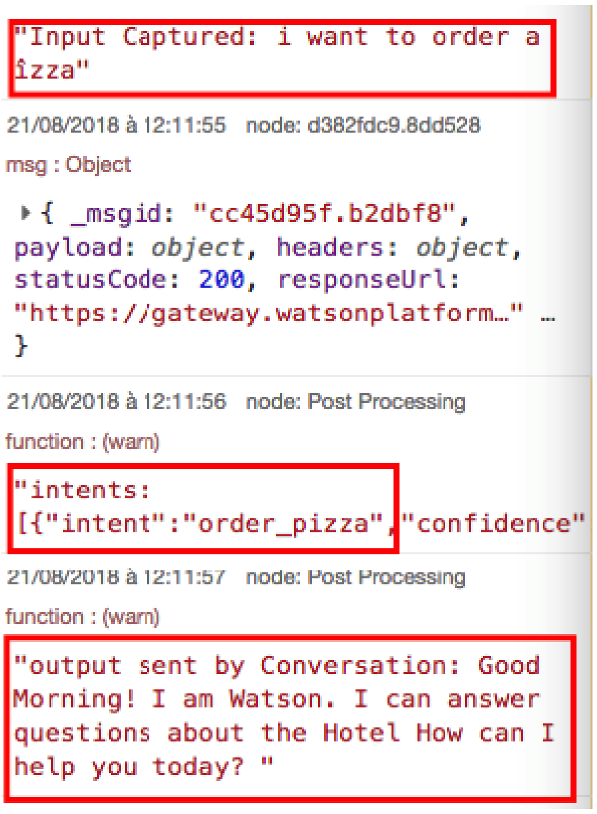
i want to order a pizza

Good Morning! I am Watson. I can answer questions about the Hotel How can I help you today?

Go back to your Node-Red application and review the information displayed in the debug panel.



```
21/08/2018 à 12:11:11   node: Pre Processing
function : (warn)
"Input Captured: "

21/08/2018 à 12:11:12   node: d382fdc9.8dd528
msg : Object
▸{ _msgid: "fb45b55e.127448",
payload: object, headers: object,
statusCode: 200, responseUrl:
"https://gateway.watsonplatform…" …
}

21/08/2018 à 12:11:13   node: Post Processing
function : (warn)
"intents: null"

21/08/2018 à 12:11:14   node: Post Processing
function : (warn)
"output sent by Conversation: Good
Morning! I am Watson. I can answer
questions about the Hotel How can I
help you today? "
```



```
"Input Captured: i want to order a
pizza"

21/08/2018 à 12:11:55   node: d382fdc9.8dd528
msg : Object
▸{ _msgid: "cc45d95f.b2dbf8",
payload: object, headers: object,
statusCode: 200, responseUrl:
"https://gateway.watsonplatform…" …
}

21/08/2018 à 12:11:56   node: Post Processing
function : (warn)
"intents:
[{"intent":"order_pizza","confidence"

21/08/2018 à 12:11:57   node: Post Processing
function : (warn)
"output sent by Conversation: Good
Morning! I am Watson. I can answer
questions about the Hotel How can I
help you today? "
```

The Conversation service is detecting the correct intents, and yet every turn of the conversation returns the welcome message from the **start of conversation** node (*Good Afternoon! I am Watson, …*).

This is happening because the Conversation service is stateless; it is the responsibility of the application to maintain state information. Because you are not yet doing anything to maintain state, the Conversation service sees every round of user input as the first turn of a new conversation, triggering the **conversation_start** condition.

Context variables

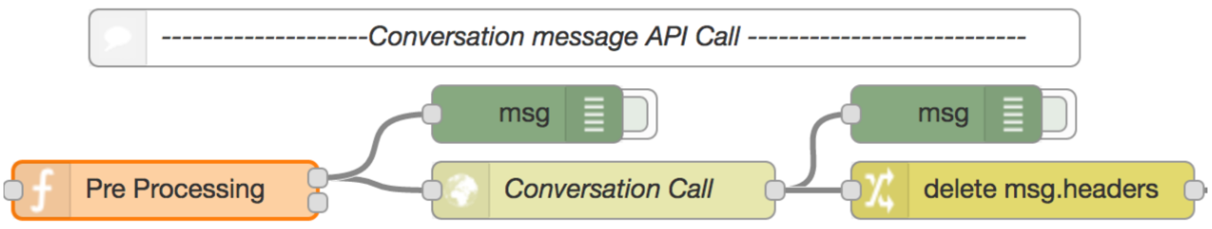
State information for your conversation is maintained using the *context*. The context is a JSON object that is passed back and forth between your application and the Conversation service. It is the responsibility of your application to maintain the context from one turn of the conversation to the next.

The context includes a unique identifier for each conversation with a user, as well as a counter that is incremented with each turn of the conversation. Our previous version of the example did not preserve the context, which means that each round of input appeared to be the start of a new conversation. We can fix that by saving the context and sending it back to the Conversation service each time.

In addition to maintaining our place in the conversation, the context can also be used to store any other data you want to pass back and forth between your application and the Conversation service. This can include persistent data you want to maintain throughout the conversation (such as a customer's name or account number), or any other data you want to track (such as the current status of option settings).

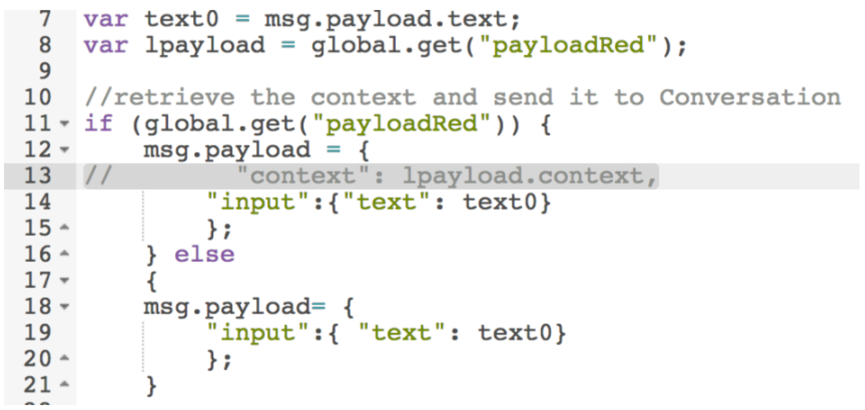
12. Maintaining state

The only change from the previous example is that with each round of the conversation, we now send back the *context* object we received in the previous round:


--------------------Conversation message API Call ---------------------------
msg
msg
Pre Processing
Conversation Call
delete msg.headers

1. DoubleClick on **Pre Processing**

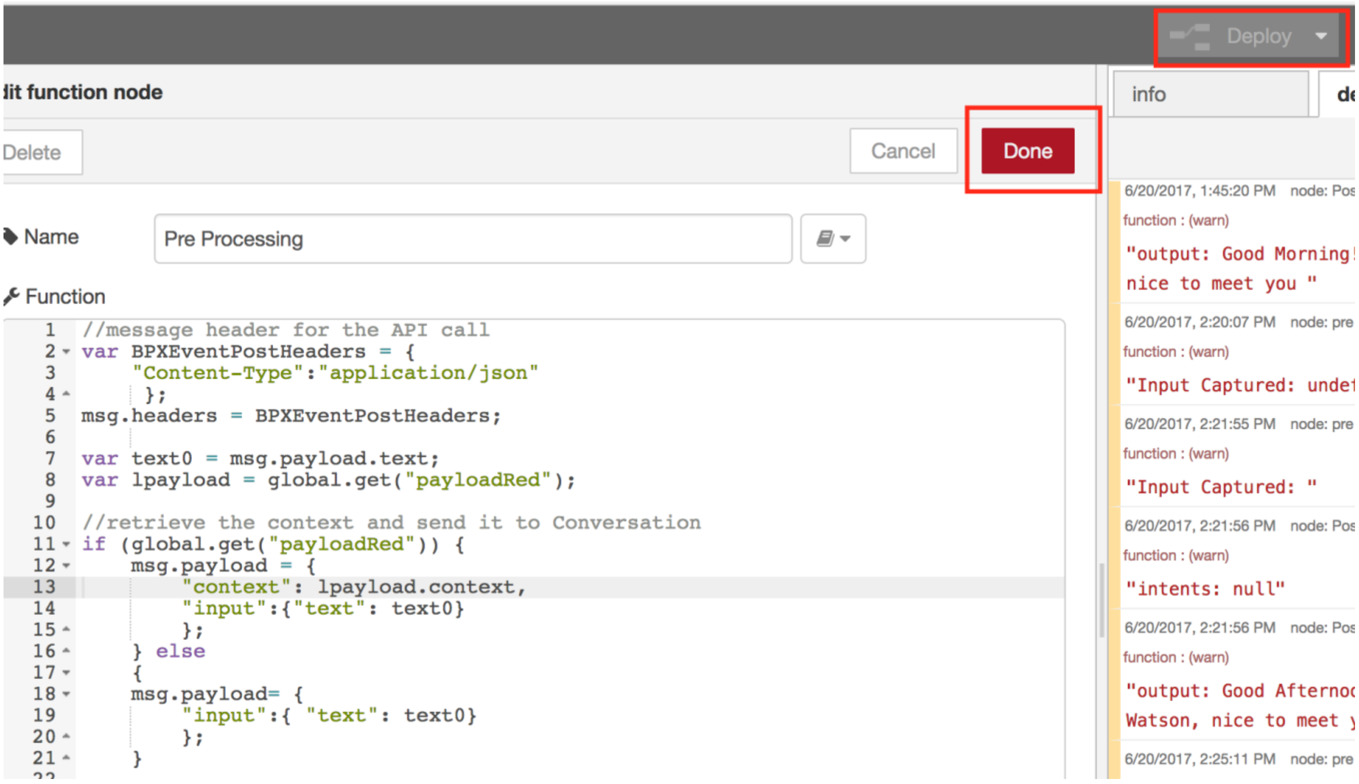
The application manages store the context in global variable *payloadRed*. This object is populated during the first turn of the conversation service.



```
var text0 = msg.payload.text;
var lpayload = global.get("payloadRed");

//retrieve the context and send it to Conversation
if (global.get("payloadRed")) {
    msg.payload = {
//          "context": lpayload.context,
          "input":{"text": text0}
          };
    } else
    {
    msg.payload= {
          "input":{ "text": text0}
          };
    }
```

2. Uncomment the line 13, “Context” : lpayload.context,

3. Click **Done**, Click **Deploy**.



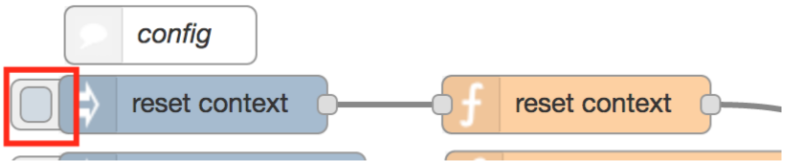
Name: Pre Processing

Function

```
1   //message header for the API call
2   var BPXEventPostHeaders = {
3       "Content-Type":"application/json"
4       };
5   msg.headers = BPXEventPostHeaders;
6
7   var text0 = msg.payload.text;
8   var lpayload = global.get("payloadRed");
9
10  //retrieve the context and send it to Conversation
11  if (global.get("payloadRed")) {
12      msg.payload = {
13          "context": lpayload.context,
14          "input":{"text": text0}
15          };
16      } else
17      {
18      msg.payload= {
19          "input":{ "text": text0}
20          };
21      }
```

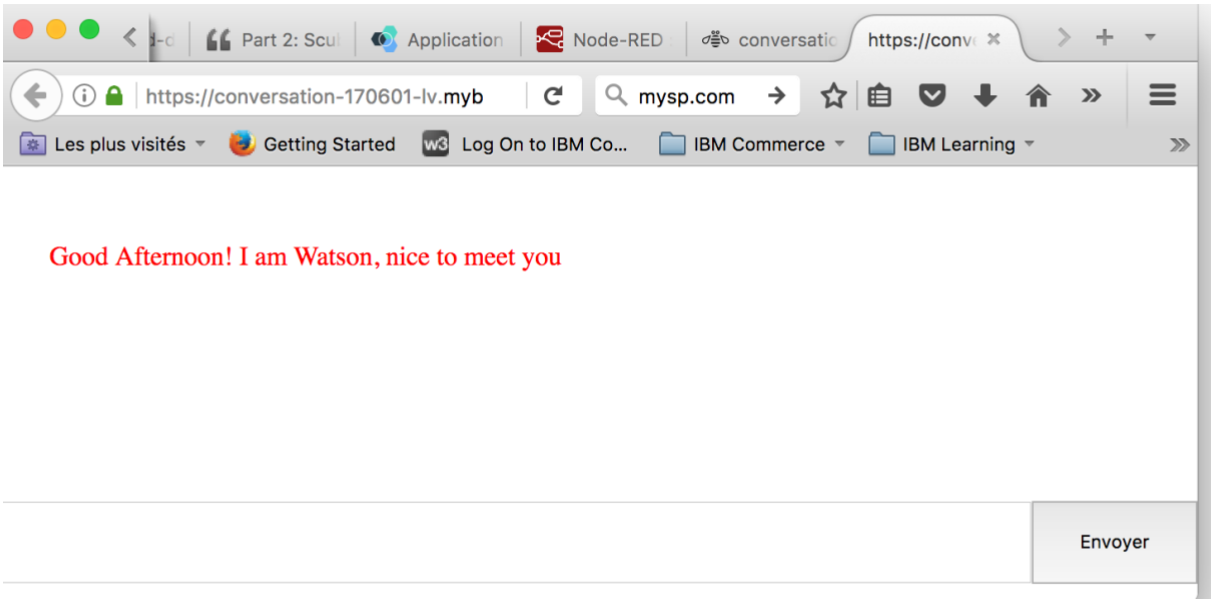
```
6/20/2017, 1:45:20 PM   node: Pos
function : (warn)
"output: Good Morning!
nice to meet you "

6/20/2017, 2:20:07 PM   node: pre
function : (warn)
"Input Captured: undei

6/20/2017, 2:21:55 PM   node: pre
function : (warn)
"Input Captured: "

6/20/2017, 2:21:56 PM   node: Pos
function : (warn)
"intents: null"

6/20/2017, 2:21:56 PM   node: Pos
function : (warn)
"output: Good Afternoo
Watson, nice to meet y

6/20/2017, 2:25:11 PM   node: pre
```

4. To reset the context, Click **reset context** inject node.



5. Go back to the Web Form, Click **Submit**

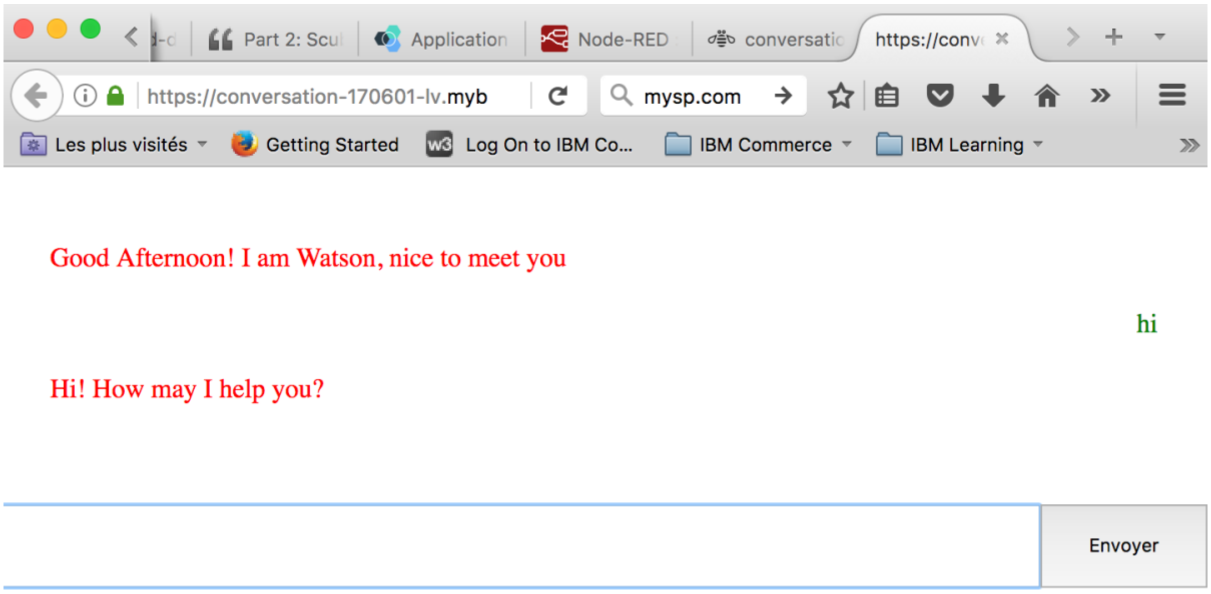
As you don't provide any input, the service returns the "Start of the conversation" responses:



Good Afternoon! I am Watson, nice to meet you

6. Enter *Hi* inputs

You should get the right responses from the service

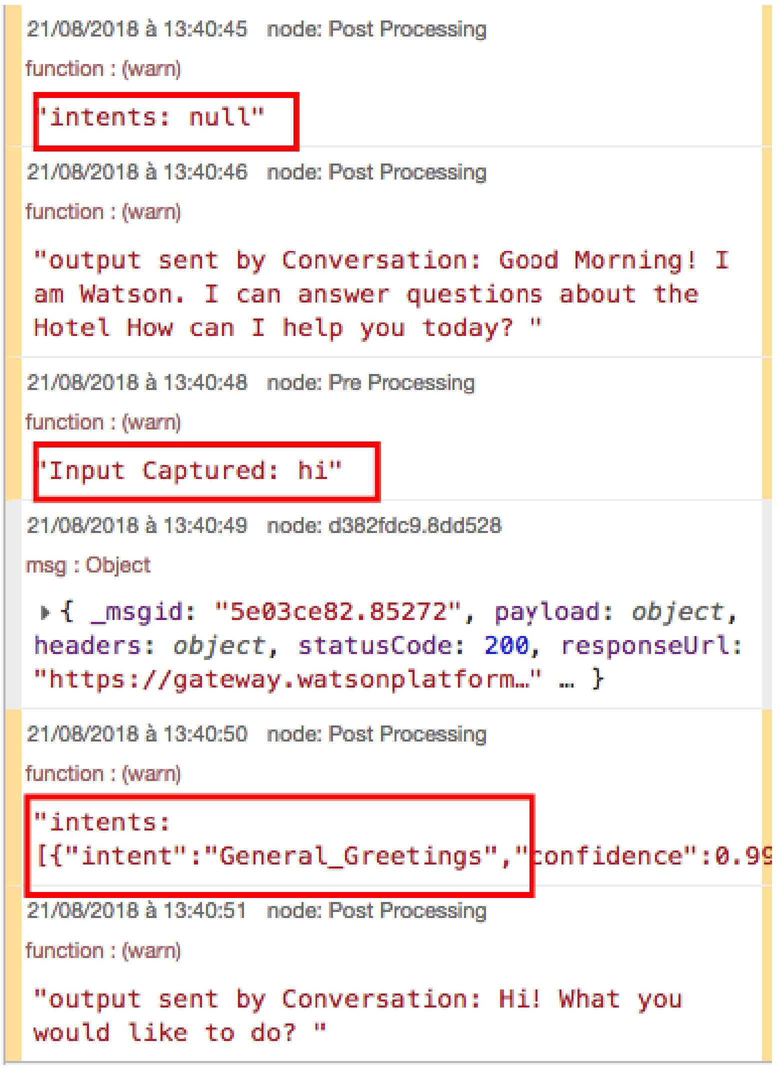


Good Afternoon! I am Watson, nice to meet you

hi

Hi! How may I help you?

7. Go back to your Node-Red application and review the information displayed in the debug pane.

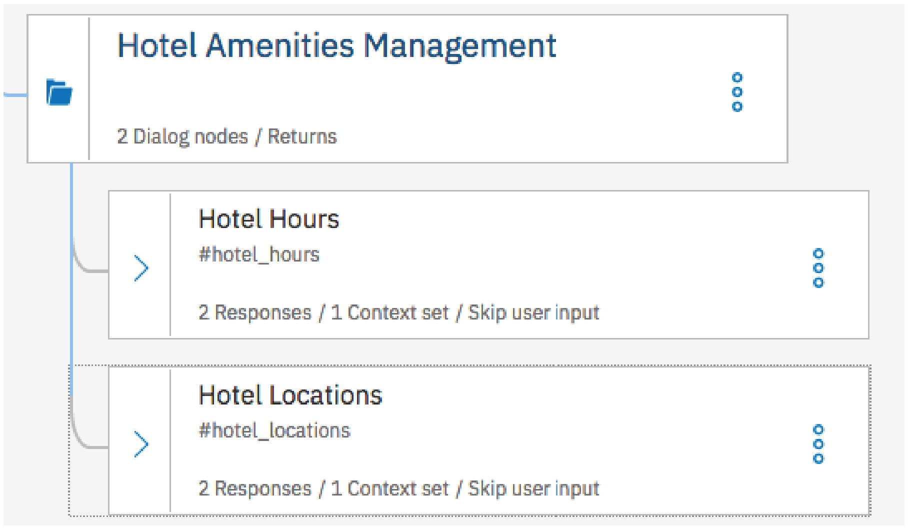


```
21/08/2018 à 13:40:45   node: Post Processing
function : (warn)
"intents: null"

21/08/2018 à 13:40:46   node: Post Processing
function : (warn)
"output sent by Conversation: Good Morning! I
am Watson. I can answer questions about the
Hotel How can I help you today? "

21/08/2018 à 13:40:48   node: Pre Processing
function : (warn)
"Input Captured: hi"

21/08/2018 à 13:40:49   node: d382fdc9.8dd528
msg : Object
▸{ _msgid: "5e03ce82.85272", payload: object,
headers: object, statusCode: 200, responseUrl:
"https://gateway.watsonplatform…" … }

21/08/2018 à 13:40:50   node: Post Processing
function : (warn)
"intents:
[{"intent":"General_Greetings","confidence":0.99

21/08/2018 à 13:40:51   node: Post Processing
function : (warn)
"output sent by Conversation: Hi! What you
would like to do? "
```

So now, it works as expected.

13. Public Context Variables

To illustrate the way to manage context variable, you are going to use the **Hotel Amenities Management** of your dialog. To manage it, you are using the entity *@hotel_amenity* and the context variable *$hotel_amenity*.



Hotel Amenities Management
2 Dialog nodes / Returns

Hotel Hours
#hotel_hours
2 Responses / 1 Context set / Skip user input

Hotel Locations
#hotel_locations
2 Responses / 1 Context set / Skip user input

When you enter *where is it?*, Conversation needs more details about the amenity you are looking for. So,if you are doing some tests you will get this behavior:

Good Morning! I am Watson. I can answer questions about the Hotel
How can I help you today?

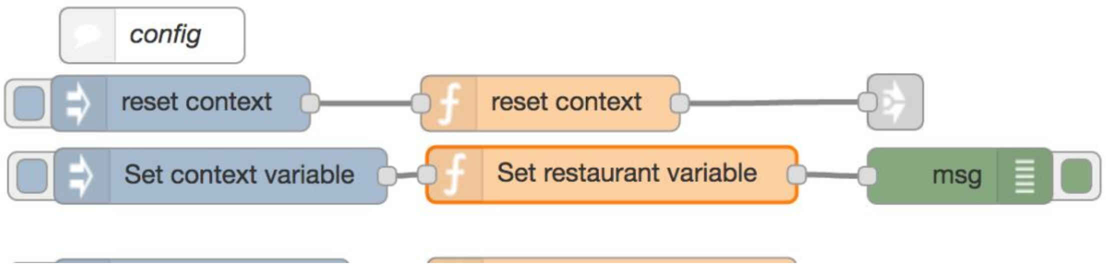
where is it?

I understand that you'd like to locate something in the hotel. Please ask me for the specific amenity you are looking for, (gym, restaurant, pool, sauna) or you may call the front desk for directions.

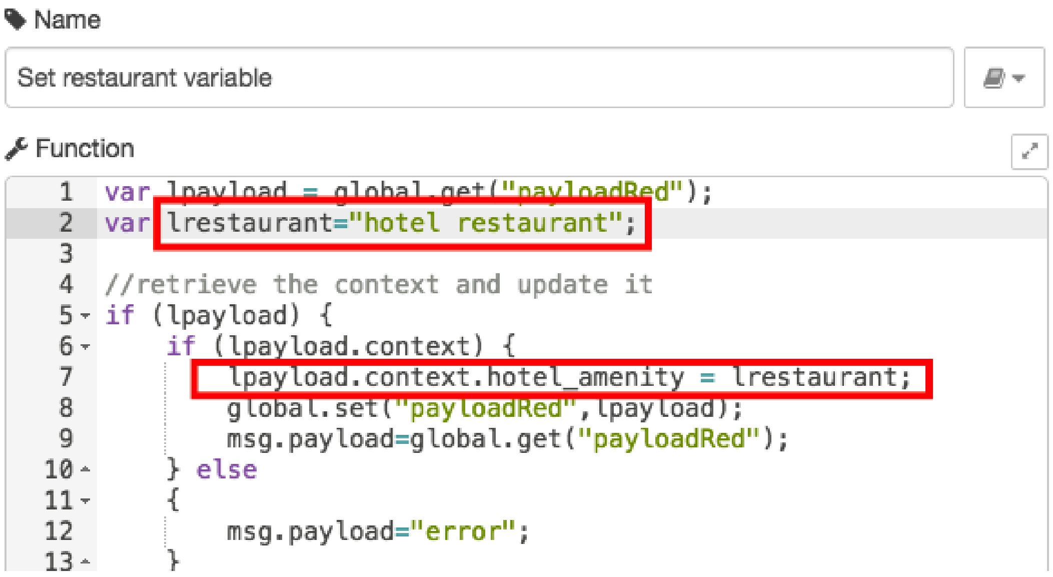
That's because *$hotel_amenity* variable must be captured by Assistant. Such an information could be stored in your profile preference for instance and could be populated by your chatbot application.

You will simulate the acquisition of *$hotel_amenity* and set up to *hotel restaurant*.

1. Go back to your Node-Red application
2. DoubleClick on **Set restaurant variable**



the global variable is updated like defined below:


Name

Set restaurant variable

Function

```
var lpayload = global.get("payloadRed");
var lrestaurant="hotel restaurant";

//retrieve the context and update it
if (lpayload) {
    if (lpayload.context) {
        lpayload.context.hotel_amenity = lrestaurant;
        global.set("payloadRed",lpayload);
        msg.payload=global.get("payloadRed");
    } else
    {
        msg.payload="error";
    }
```

3. Click **Cancel** button
4. To update the variable, click **reset context** inject node
5. click **set context variable** inject node.
6. On the chat window, click **submit** then enter *where is it?*

The variable has been set correctly and you should get the following behaviour.

Good Afternoon! I am Watson. I can answer questions about the Hotel How can I help you today?

where is it?

The hotel restaurant is located on the ground floor of the hotel.

Private Context Variables

An application must be based around the idea of managing two sets context information, *public* and *private*

Public Context is the context object that is sent to Watson Conversation as part of the request. It will be available inside of Watson Conversation through the standard API and will be visible in plain text through the application.

Private Context is maintained in the application's memory and is never sent to Watson Conversation. This makes it appropriate for data that may be sensitive, or simply if it's not important to send to Watson Conversation.

Both *public* and *private* context can be used to store information to use in API calls as well as to augment the response to the user.

14. Storing a user input

Sometimes a developer will need to store a user's next response, for instance, Watson may ask the user a question and need to store that information for later. Your Node-Red application allows for this situation to be quickly and simply addressed with the following syntax on Watson Assistant.

```
{
"output": {
        "generic": {…},
        "updatesContext": {
                "value": "name",
                "context": "private"
                }
        }
}
```

or

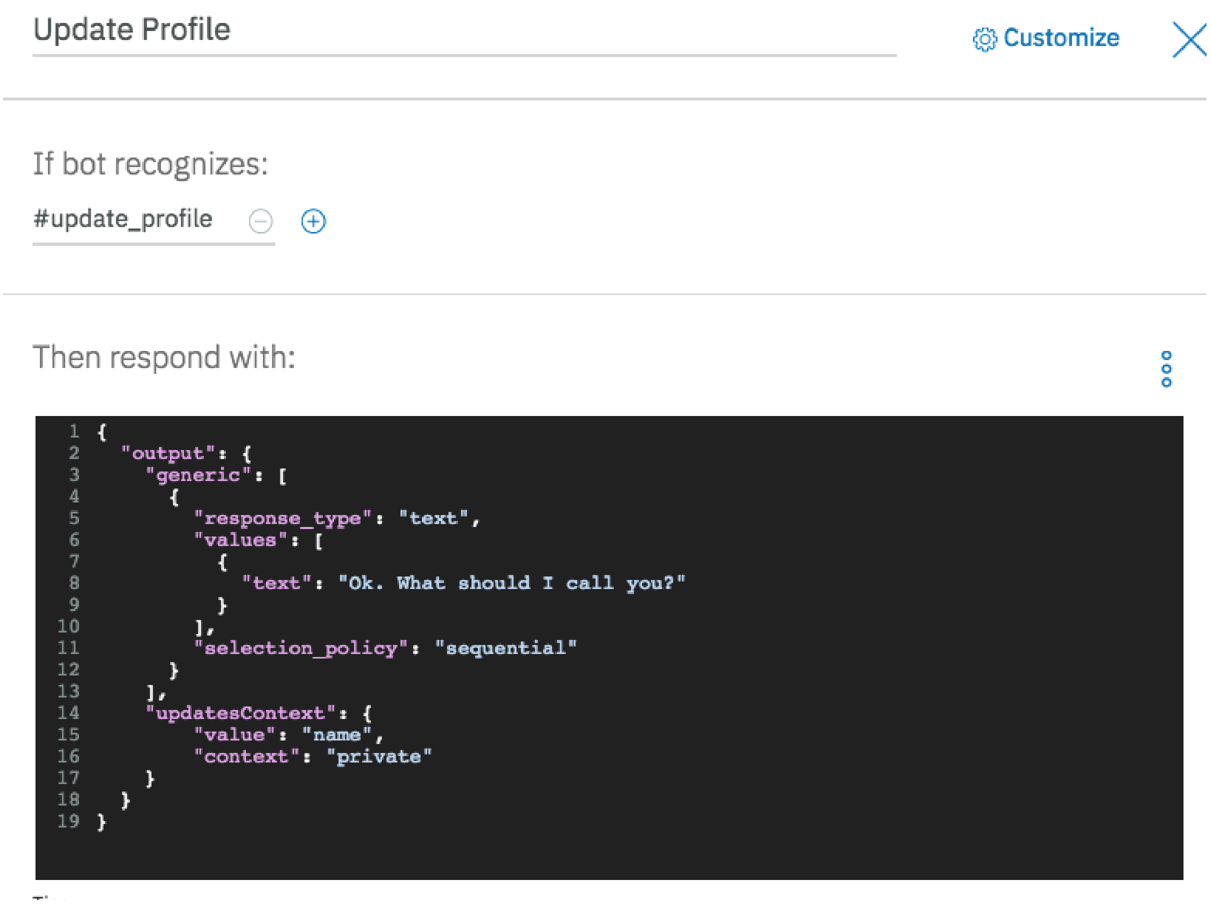
```
{
"output": {
        "generic": {…},
        "updatesContext": {
                "value": "name",
                "context": "public"
                }
        }
}
```

The presence of this property will indicate to the application that the **next** response from the user will be stored in the variable name as public or private context.

1. Go back to your Conversation service and open **Dialog** tab

2. Add a new node before the **anything else** node.

3. Fill it as below:



```
{
  "output": {
    "generic": [
      {
        "response_type": "text",
        "values": [
          {
            "text": "Ok. What should I call you?"
          }
        ],
        "selection_policy": "sequential"
      }
    ],
    "updatesContext": {
        "value": "name",
        "context": "private"
    }
  }
}
```

Copy of the response:

```
{
  "output": {
    "generic": [
      {
        "response_type": "text",
        "values": [
          {
            "text": "Ok. What should I call you?"
          }
        ],
        "selection_policy": "sequential"
      }
    ],
    "updatesContext": {
                        "value": "name",
                        "context": "private"
          }
  }
}
```

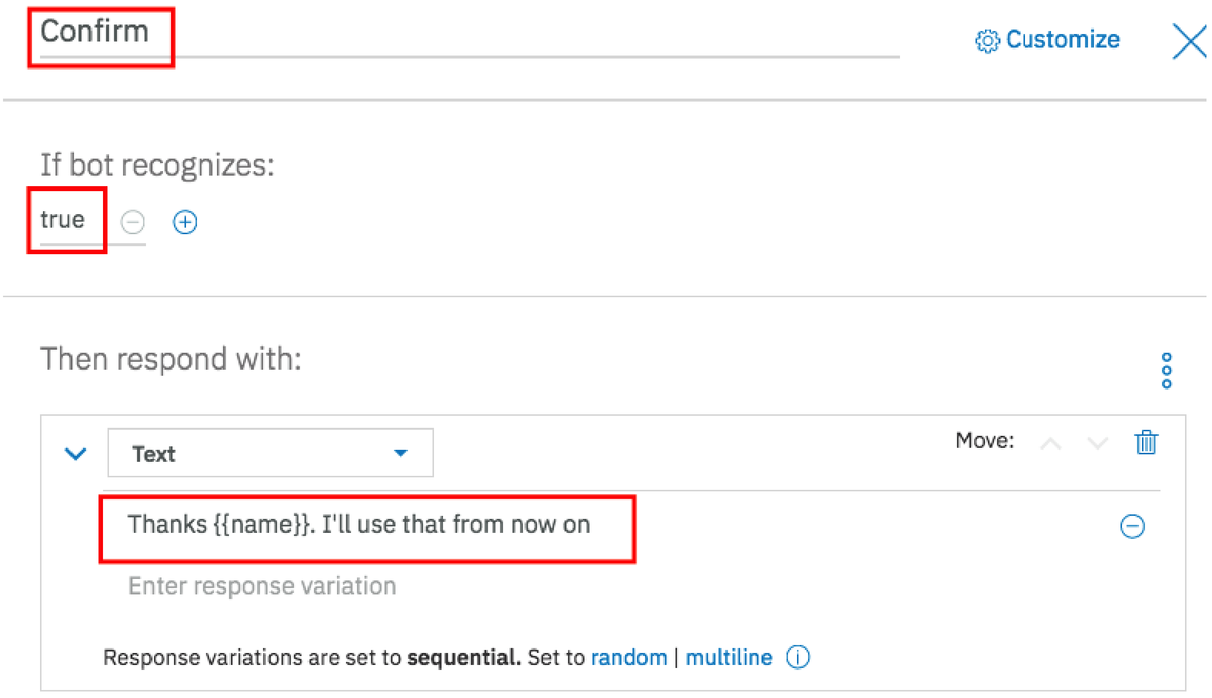
15. Augmenting a response

For the chatbot to truly be dynamic, it's not enough for the bot to simply call external APIs and internal functions, but it needs to tailor its response based on the information retrieved in these integrations. Since each API call requires that it returns a Promise that will update context and privateContext, we need a way to access this information quickly.

The Node-Red application allows you to include the following syntax {{fieldName}} in their responses. The application will update these references and replace them with the first matching option from the ordered list:

- publicContext.fieldName ,
- privatecontext.fieldName
- or keep the placeholder indicating the value was not found.

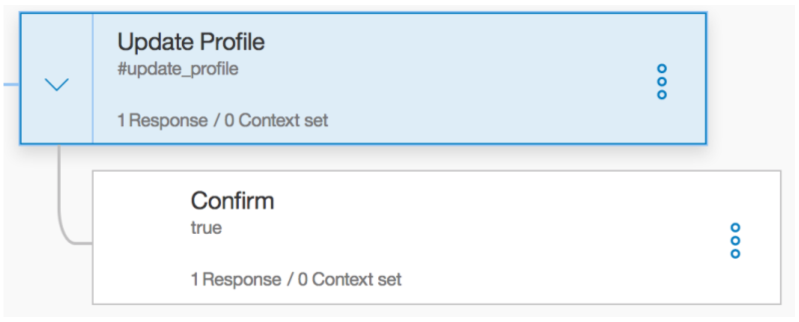
1. Select **Update Profile** node, and add a child node
2. Fill it as below:


Confirm
Customize
If bot recognizes:
true
Then respond with:
Text
Move:
Thanks {{name}}. I'll use that from now on
Enter response variation
Response variations are set to **sequential.** Set to random | multiline

Copy of the response:

Thanks {{name}}. I'll use that from now on.

Storing a user's next response as a context or privateContext field just takes 1 dialog node. In this example, we'll confirm that the value was stored with the second node.


Update Profile
#update_profile
1 Response / 0 Context set
Confirm
true
1 Response / 0 Context set

16. Review the stored Data

1. In the chatbot window, enter *update my profile*
2. Conversation will ask you to provide your new name, enter the one you want to use.
3. enter *your name*

   You should get something like below:

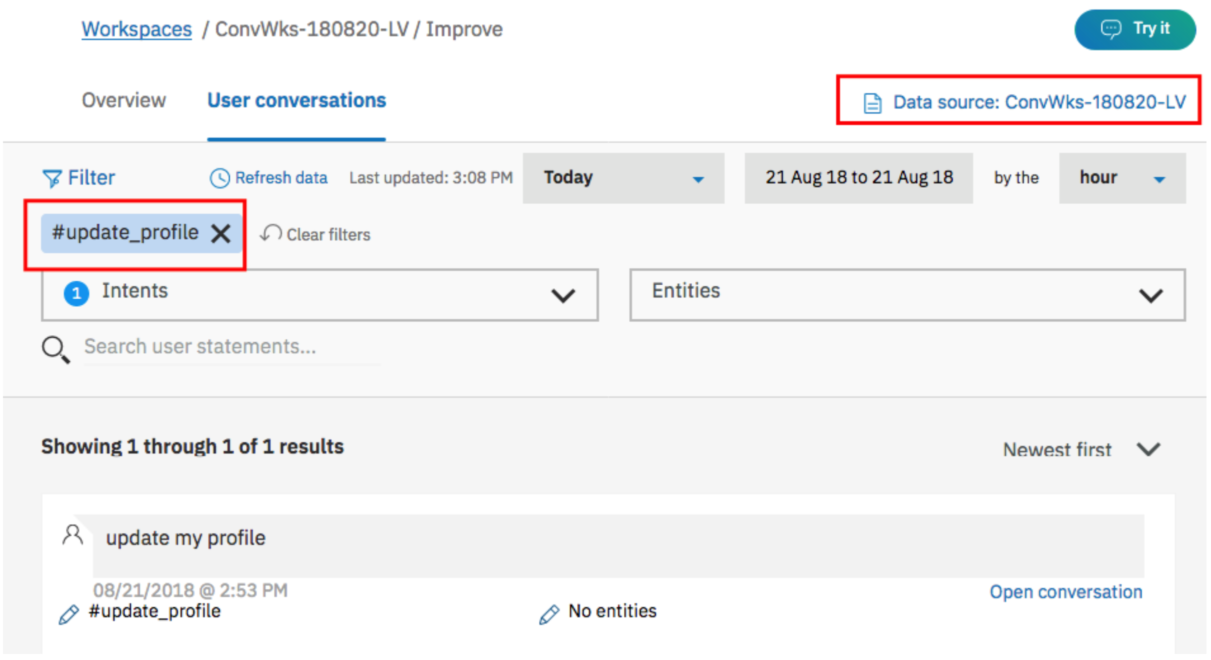
update my profile

Ok. What should I call you?

laurent

Thanks laurent. I'll use that from now on.

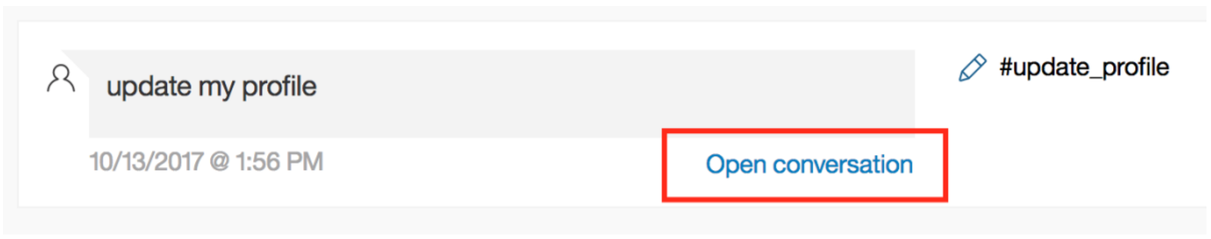
4. Let's go to **improve** page of your Assistant service
5. Select the right data source (*ChabotProduction* alias is not used in Nodered)
6. Select **User Conversations** tab and select *#update_profile* intent as filter


Workspaces / ConvWks-180820-LV / Improve

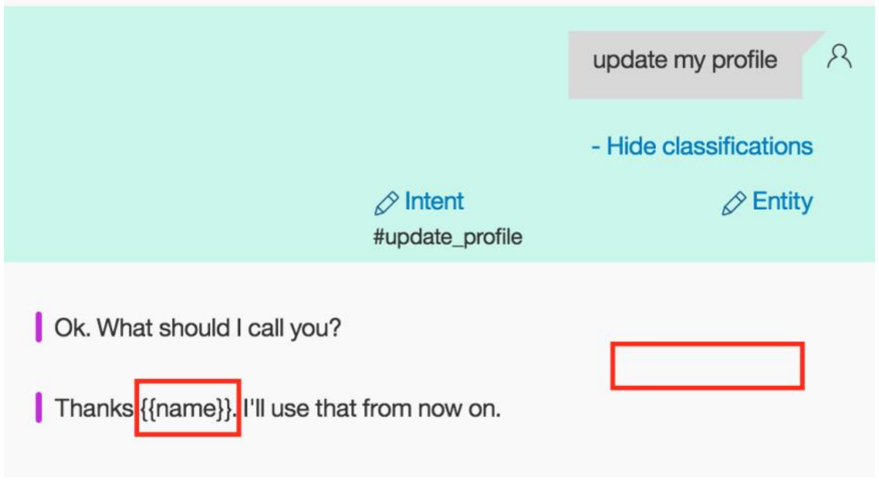
Showing 1 through 1 of 1 results

update my profile
08/21/2018 @ 2:53 PM
#update_profile
No entities

7. To review the saved data, click **Open conversation** of the latest record


update my profile
10/13/2017 @ 1:56 PM
#update_profile

You have got the confirmation that the variable *name* is not captured by Conversation.


update my profile

#update_profile

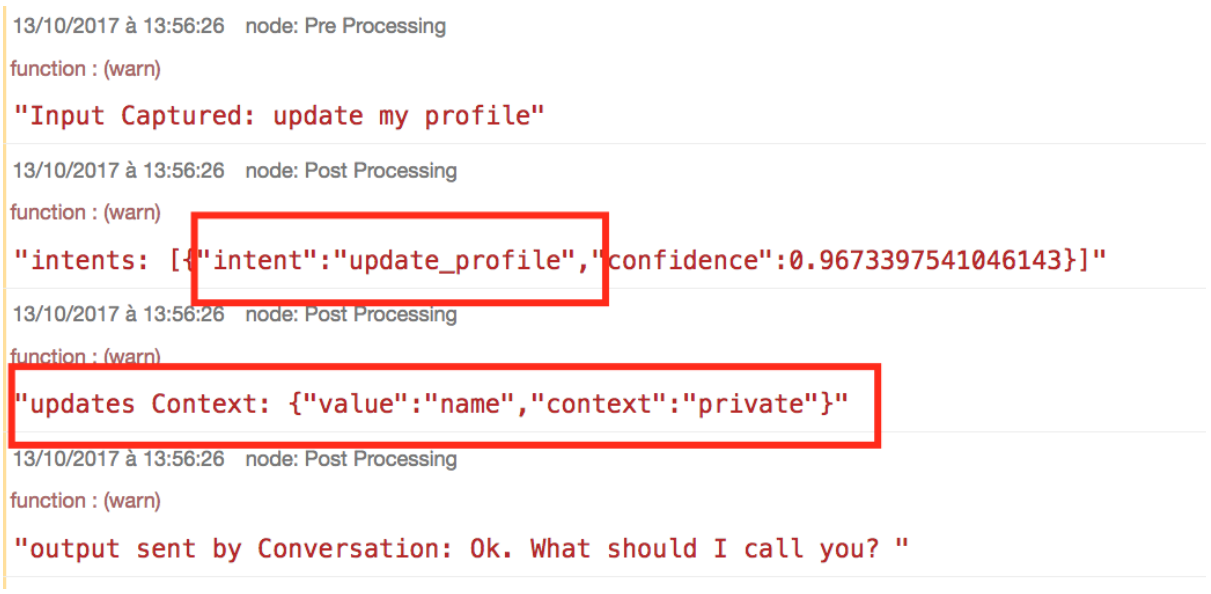
Ok. What should I call you?

Thanks {{name}}. I'll use that from now on.

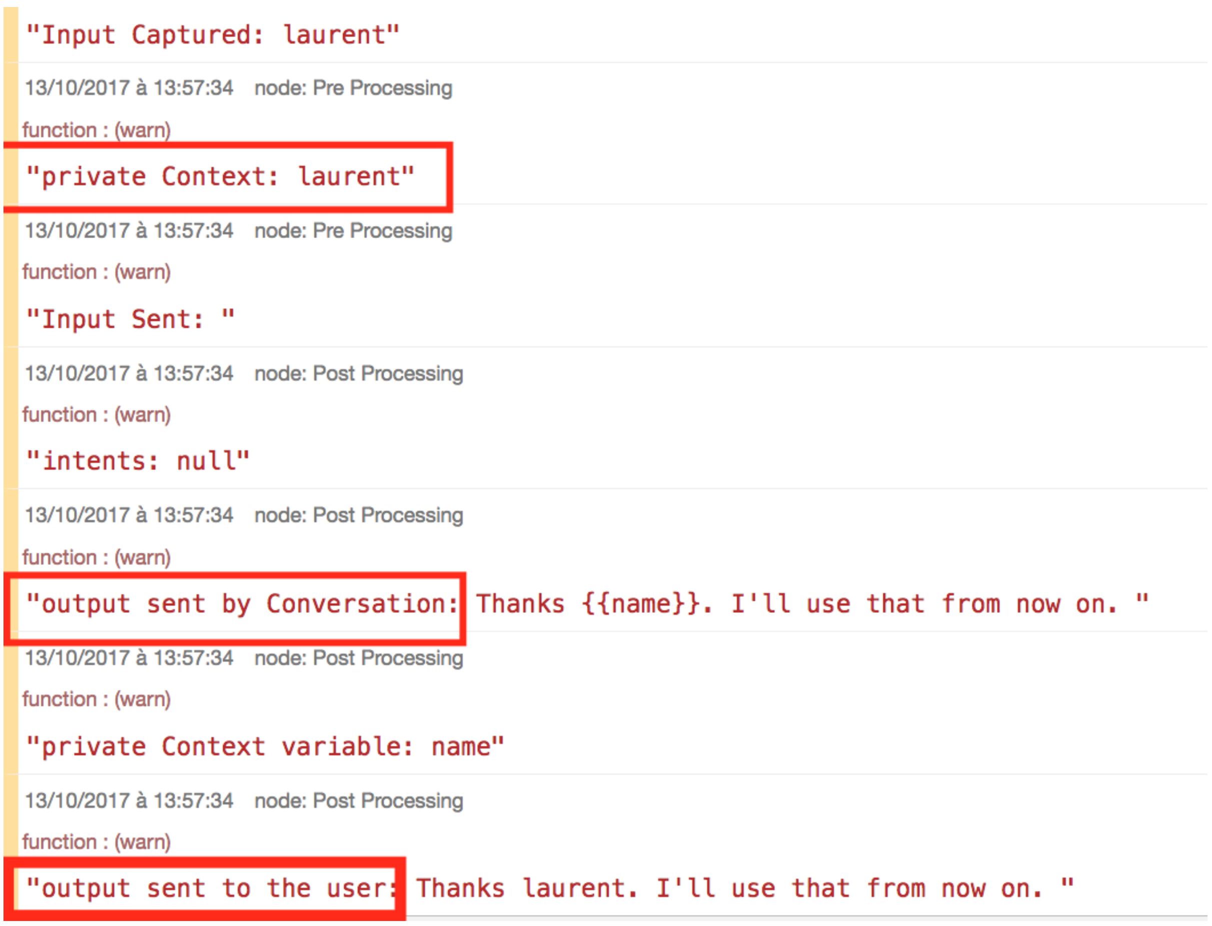
On the debug pane of your Node-Red application, you can see that

the data captured is *laurent*

but the data sent to conversation is empty

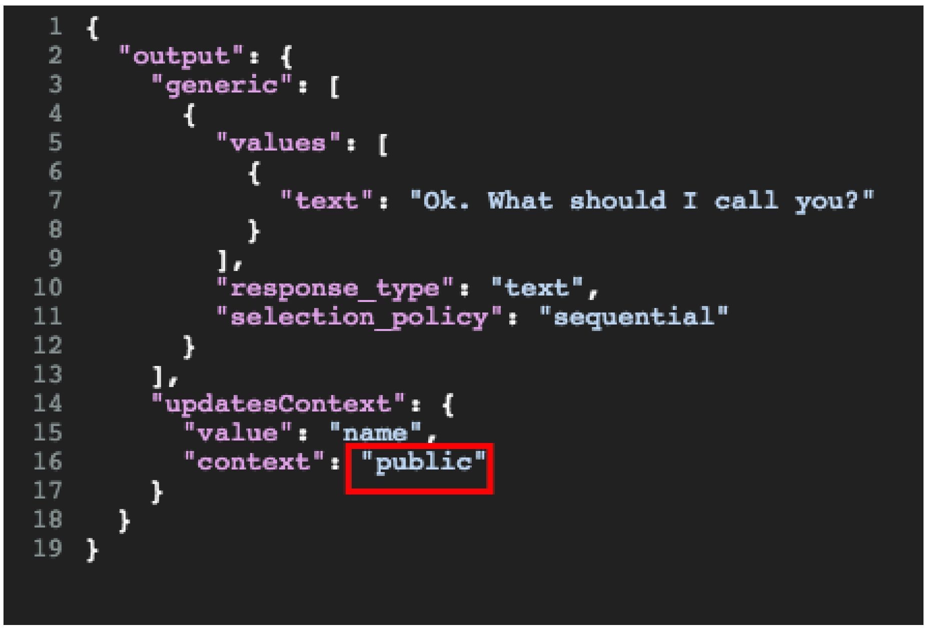


```
13/10/2017 à 13:56:26   node: Pre Processing
function : (warn)
"Input Captured: update my profile"
13/10/2017 à 13:56:26   node: Post Processing
function : (warn)
"intents: [{"intent":"update_profile","confidence":0.9673397541046143}]"
13/10/2017 à 13:56:26   node: Post Processing
function : (warn)
"updates Context: {"value":"name","context":"private"}"
13/10/2017 à 13:56:26   node: Post Processing
function : (warn)
"output sent by Conversation: Ok. What should I call you? "
```



```
"Input Captured: laurent"
13/10/2017 à 13:57:34   node: Pre Processing
function : (warn)
"private Context: laurent"
13/10/2017 à 13:57:34   node: Pre Processing
function : (warn)
"Input Sent: "
13/10/2017 à 13:57:34   node: Post Processing
function : (warn)
"intents: null"
13/10/2017 à 13:57:34   node: Post Processing
function : (warn)
"output sent by Conversation: Thanks {{name}}. I'll use that from now on. "
13/10/2017 à 13:57:34   node: Post Processing
function : (warn)
"private Context variable: name"
13/10/2017 à 13:57:34   node: Post Processing
function : (warn)
"output sent to the user: Thanks laurent. I'll use that from now on. "
```

8. You can test the application behaviour if the variable is public. In the dialog page, edit the node **Update Profile**
9. Open the **Json Editor**
10. Replace *private* with *public*



```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Ok. What should I call you?"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ],
    "updatesContext": {
      "value": "name",
      "context": "public"
    }
  }
}
```

11. Repeat the previous test, and review the collected data.

The behaviour doesn't change from the user point of view

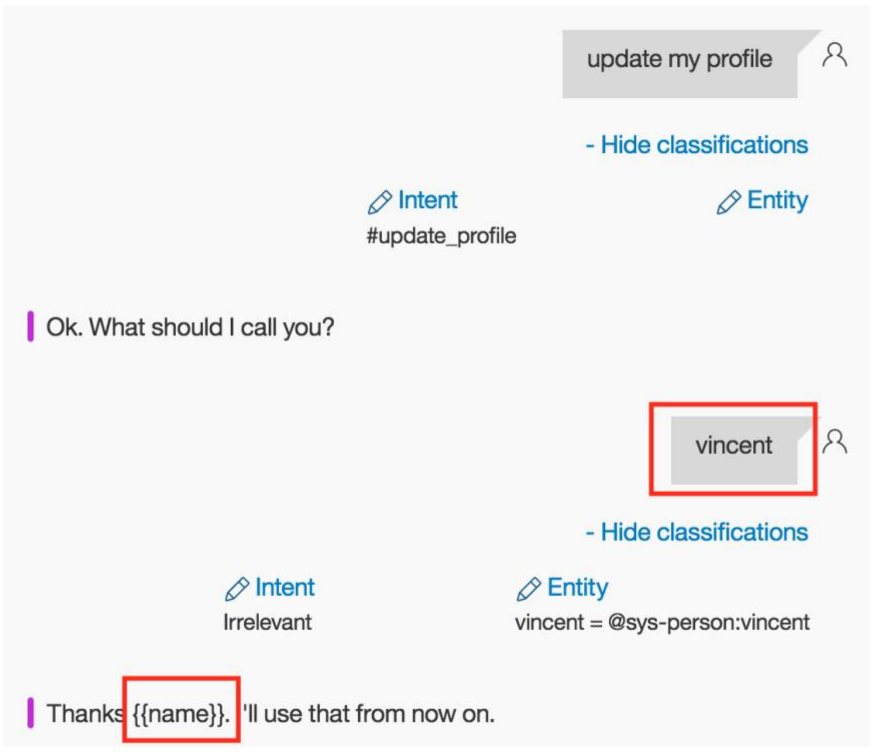
update my profile

Ok. What should I call you?

vincent

Thanks vincent. I'll use that from now on.

Conversation capture the provided name


update my profile

- Hide classifications

Intent
#update_profile

Entity

Ok. What should I call you?

vincent

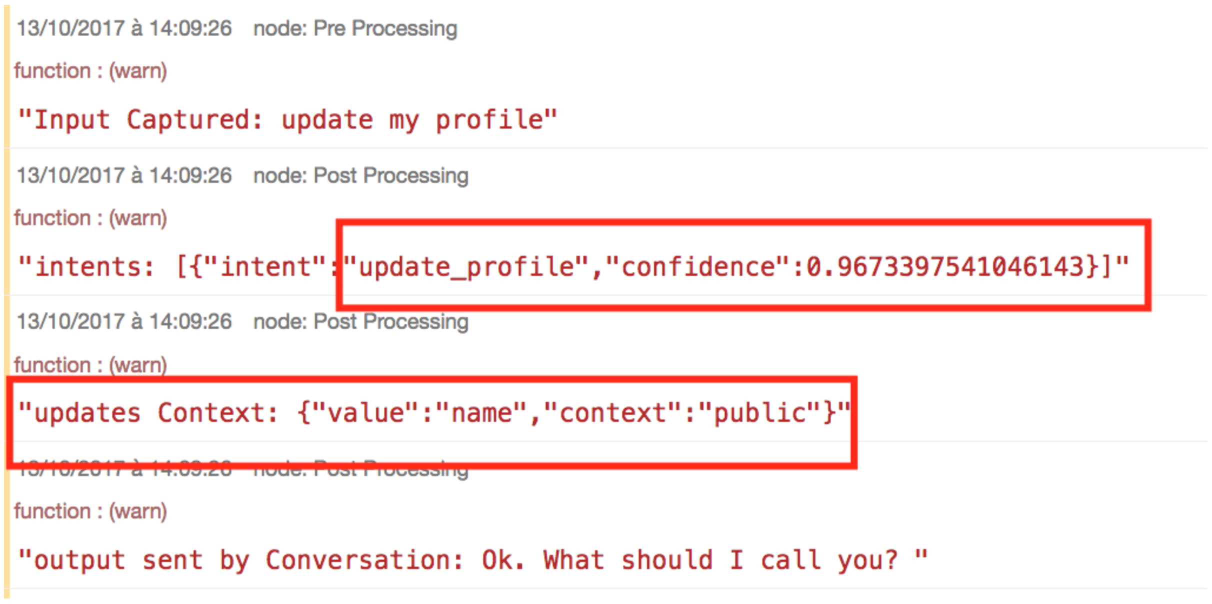
- Hide classifications

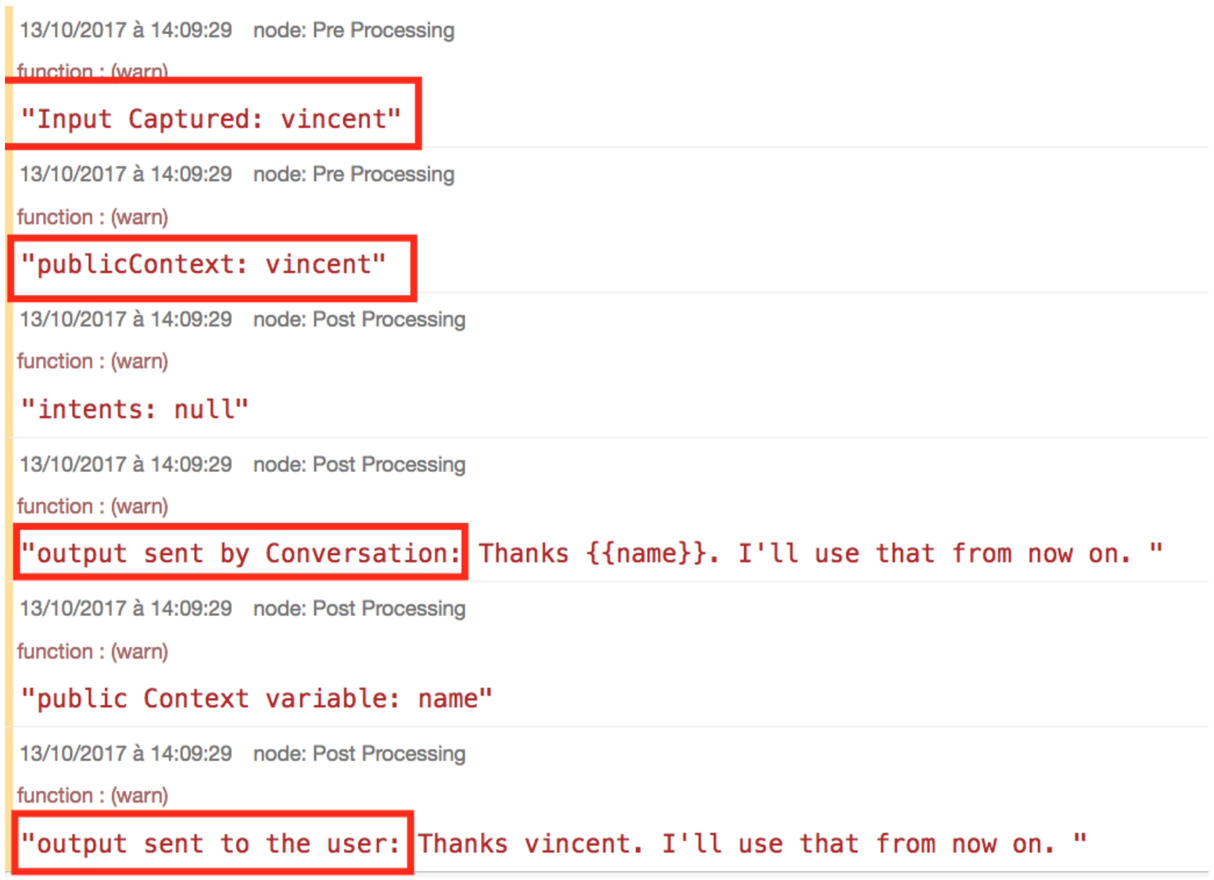
Intent
Irrelevant

Entity
vincent = @sys-person:vincent

Thanks {{name}}. 'll use that from now on.

You should get the log details as below.


```
13/10/2017 à 14:09:26   node: Pre Processing
function : (warn)
"Input Captured: update my profile"
13/10/2017 à 14:09:26   node: Post Processing
function : (warn)
"intents: [{"intent":"update_profile","confidence":0.9673397541046143}]"
13/10/2017 à 14:09:26   node: Post Processing
function : (warn)
"updates Context: {"value":"name","context":"public"}"
13/10/2017 à 14:09:26   node: Post Processing
function : (warn)
"output sent by Conversation: Ok. What should I call you? "
```



```
13/10/2017 à 14:09:29   node: Pre Processing
function : (warn)
"Input Captured: vincent"
13/10/2017 à 14:09:29   node: Pre Processing
function : (warn)
"publicContext: vincent"
13/10/2017 à 14:09:29   node: Post Processing
function : (warn)
"intents: null"
13/10/2017 à 14:09:29   node: Post Processing
function : (warn)
"output sent by Conversation: Thanks {{name}}. I'll use that from now on. "
13/10/2017 à 14:09:29   node: Post Processing
function : (warn)
"public Context variable: name"
13/10/2017 à 14:09:29   node: Post Processing
function : (warn)
"output sent to the user: Thanks vincent. I'll use that from now on. "
```

Call an external API (Tone Analyzer)

In addition to the output text to be displayed to the user, our dialog uses the *output* object in the response JSON to signal when the application needs to carry out an action, based on the detected intents.

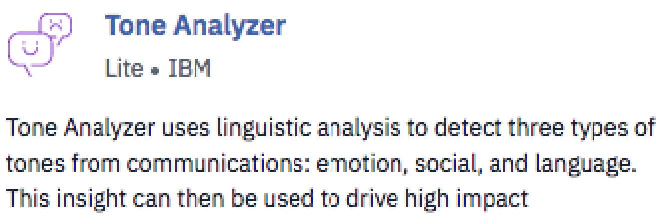
These action flags are sent using the *action* property, which our dialog defines as part of the response JSON. When the dialog determines that the application needs to do something, it sets the value of *action* to the appropriate value in your case a call to the Tone Analyzer API.

Keep in mind that *output* is just a JSON object, and you can add any valid content to it. For a more complex application, you might use an array with multiple action flags.

But in our example, we're using a simple key/value pair that supports a single action flag. Our application code needs to check the value of the *action* property in the response and then carry out any specified action.

17. Configure the Tone Analyzer service and node

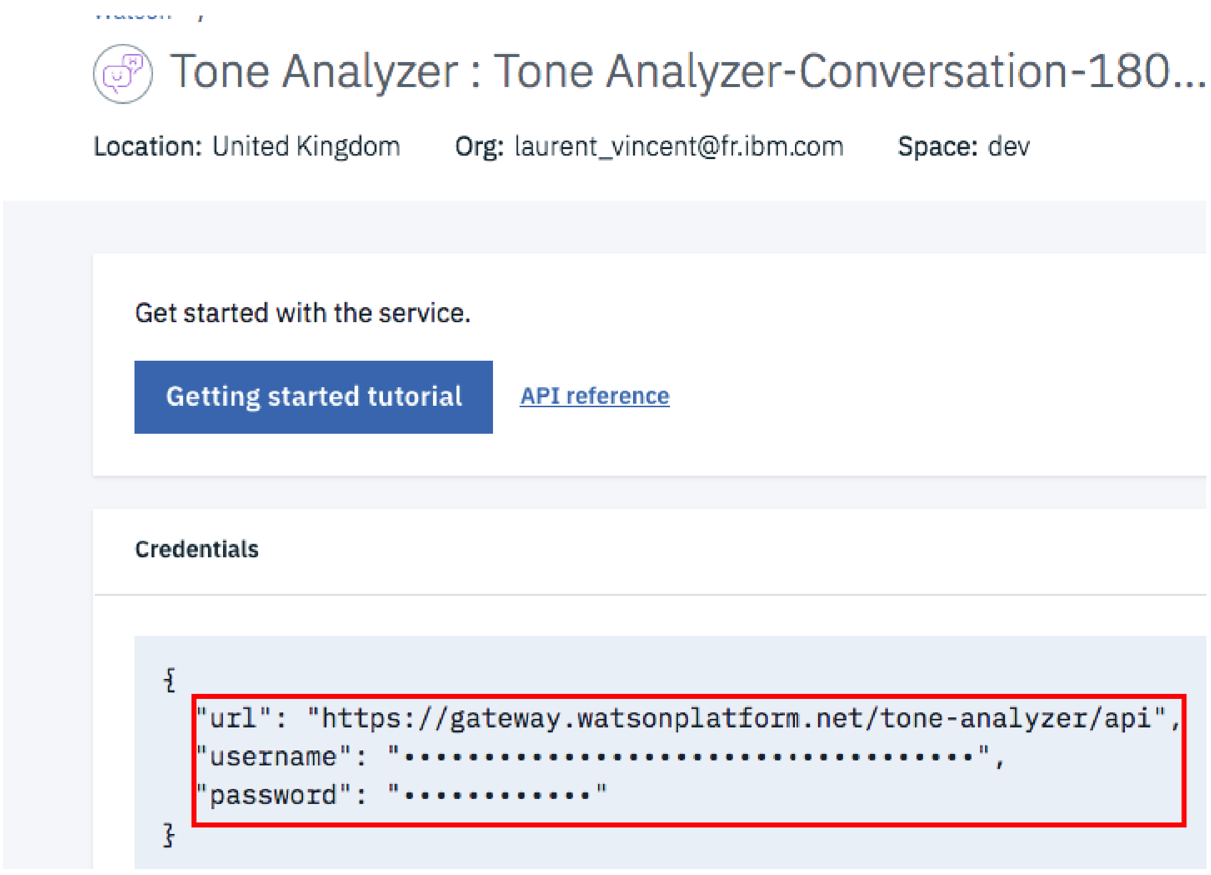
1. Go back to the IBM Cloud catalog,
2. Look for **Tone Analyzer**, Click on **Tone Analyzer** tile



Tone Analyzer
Lite • IBM

Tone Analyzer uses linguistic analysis to detect three types of tones from communications: emotion, social, and language. This insight can then be used to drive high impact

3. Set a name and keep United Kingdom as region, hen click **Create**



Tone Analyzer : Tone Analyzer-Conversation-180...

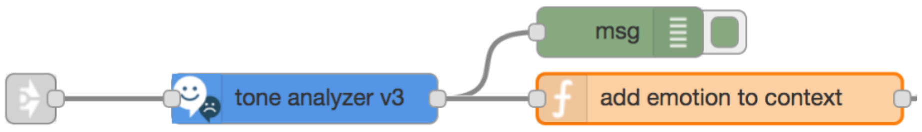
Location: United Kingdom Org: laurent_vincent@fr.ibm.com Space: dev

Get started with the service.

Credentials

```
{
  "url": "https://gateway.watsonplatform.net/tone-analyzer/api",
  "username": "••••••••••••••••••••••••••••••••••••",
  "password": "••••••••••••"
}
```

4. Copy the credentials *username* and *password* on your clipboard.
5. Go back to your Node-Red application
6. Doubleclick on **Tone Analyser** node



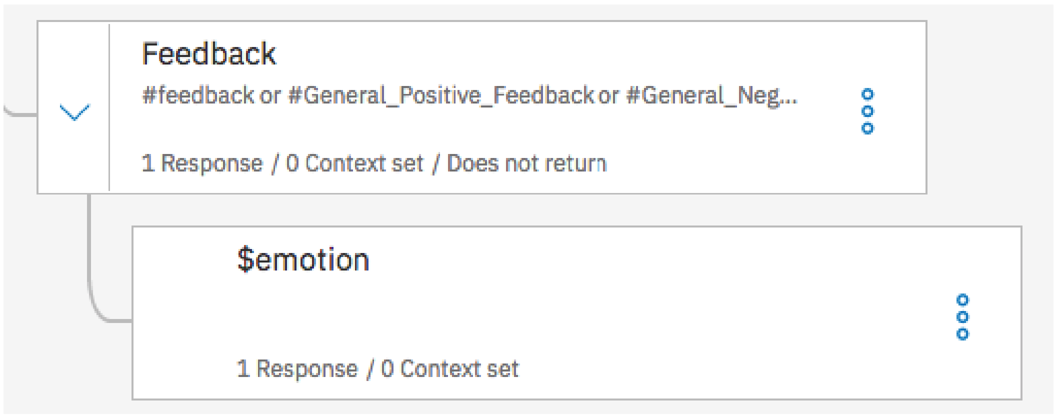
7. Fill the *username* and *password* with the values of your Tone Analyzer service
8. Click **Done,** then **Deploy**

Your Tone Analyzer is ready!

18. Update Dialog

You are going to create 2 nodes.

The first captures the *#feedback* intent and request a call to action
the second displays the value *$emotion* with the best confidence rate returned by Tone Analyzer.



Feedback

#feedback or #General_Positive_Feedback or #General_Neg...

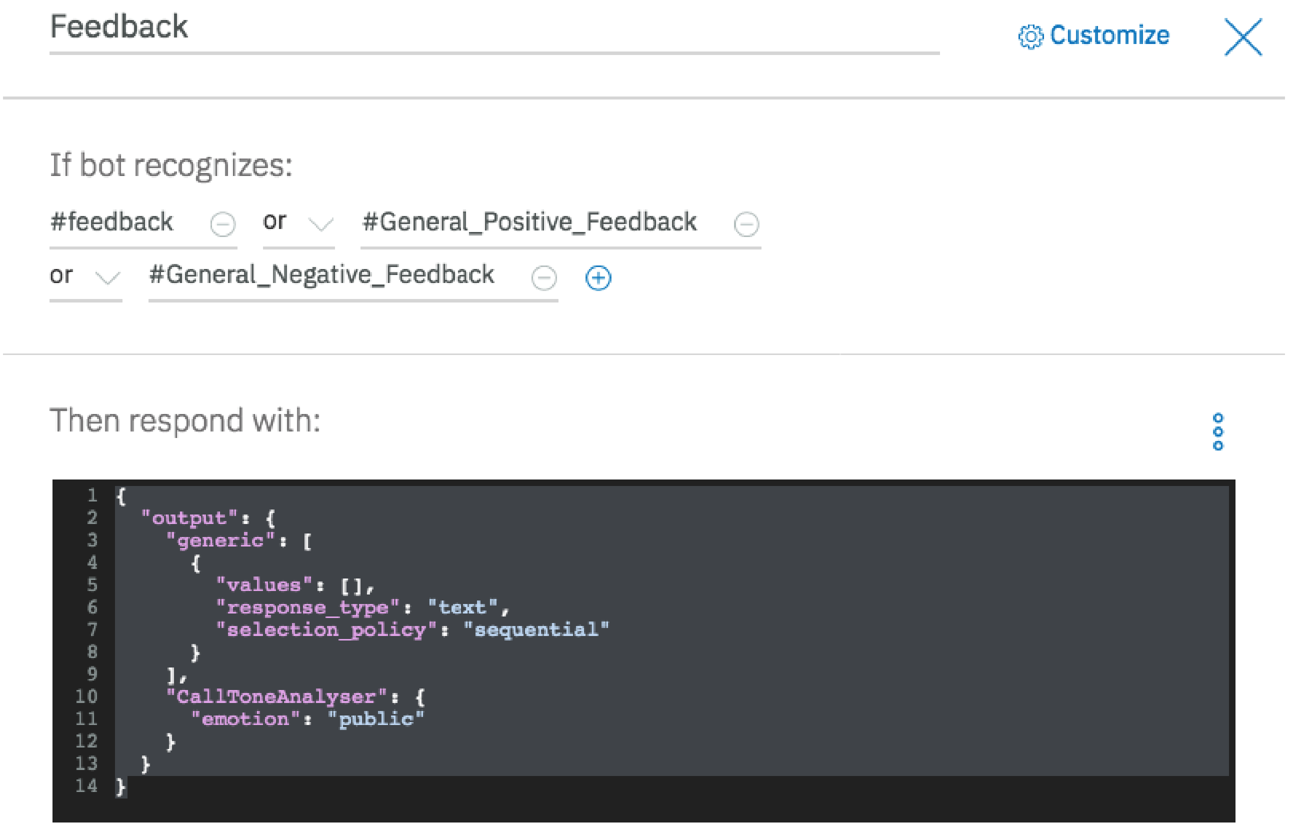
1 Response / 0 Context set / Does not return

$emotion

1 Response / 0 Context set

1. Create the branch as described above, you can get the details of each node below:

   First node



Feedback

If bot recognizes:

#feedback or #General_Positive_Feedback or #General_Negative_Feedback

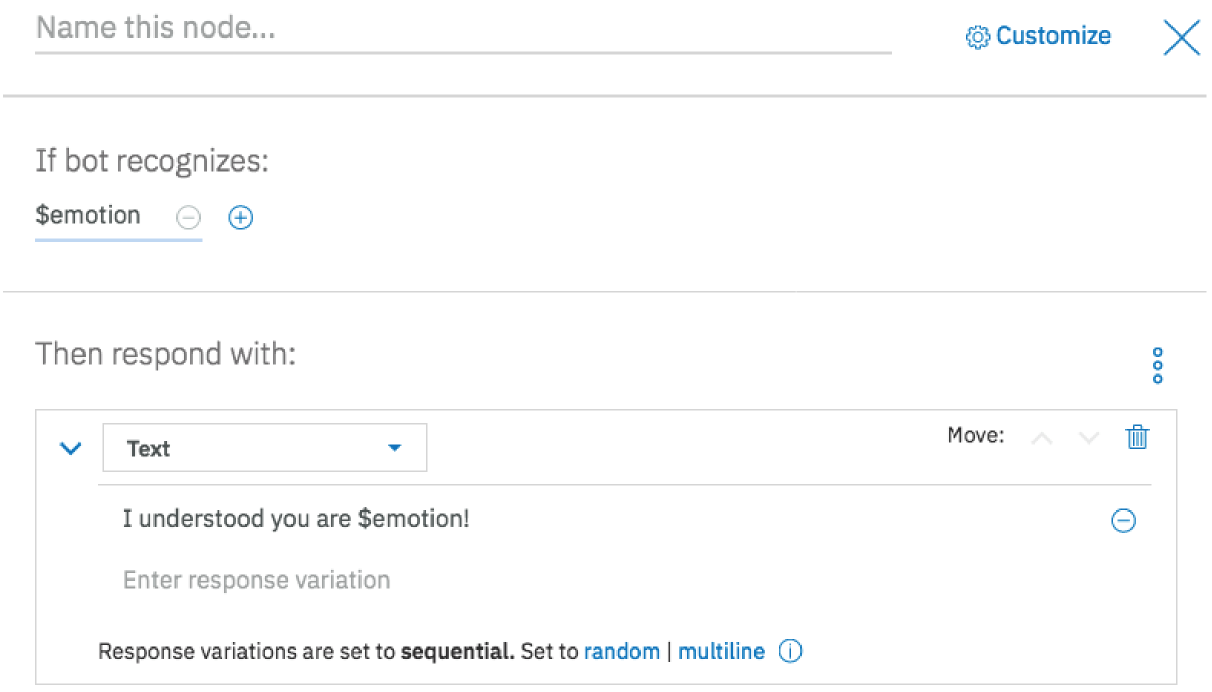
Then respond with:

```
{
  "output": {
    "generic": [
      {
        "values": [],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ],
    "CallToneAnalyser": {
      "emotion": "public"
    }
  }
}
```

Copy of the response:

```
{
  "output": {
    "generic": [
      {
        "values": [],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ],
    "CallToneAnalyser": {
      "emotion": "public"
    }
  }
}
```

Second node



Name this node...

Customize

If bot recognizes:

$emotion

Then respond with:

I understood you are $emotion!

Enter response variation

Response variations are set to **sequential.** Set to random | multiline

Copy of the response:

I understood you are $emotion!

19. Test it

You can work with this new integration:

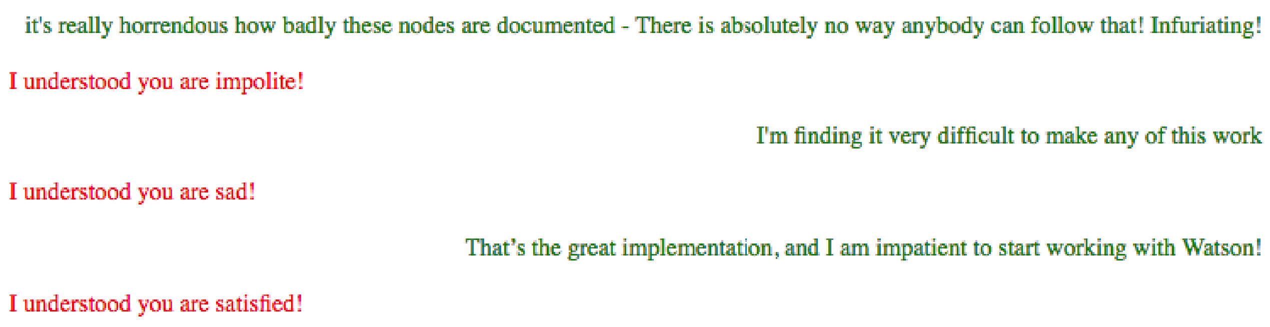
1. On the Chatbot form enter successively the following sentences:

   it's really horrendous how badly these nodes are documented - There is absolutely no way anybody can follow that! Infuriating!

   I'm finding it very difficult to make any of this work

   That's the great implementation, and I am impatient to start working with Watson!

You should get:



it's really horrendous how badly these nodes are documented - There is absolutely no way anybody can follow that! Infuriating!

I understood you are impolite!

I'm finding it very difficult to make any of this work

I understood you are sad!

That's the great implementation, and I am impatient to start working with Watson!

I understood you are satisfied!

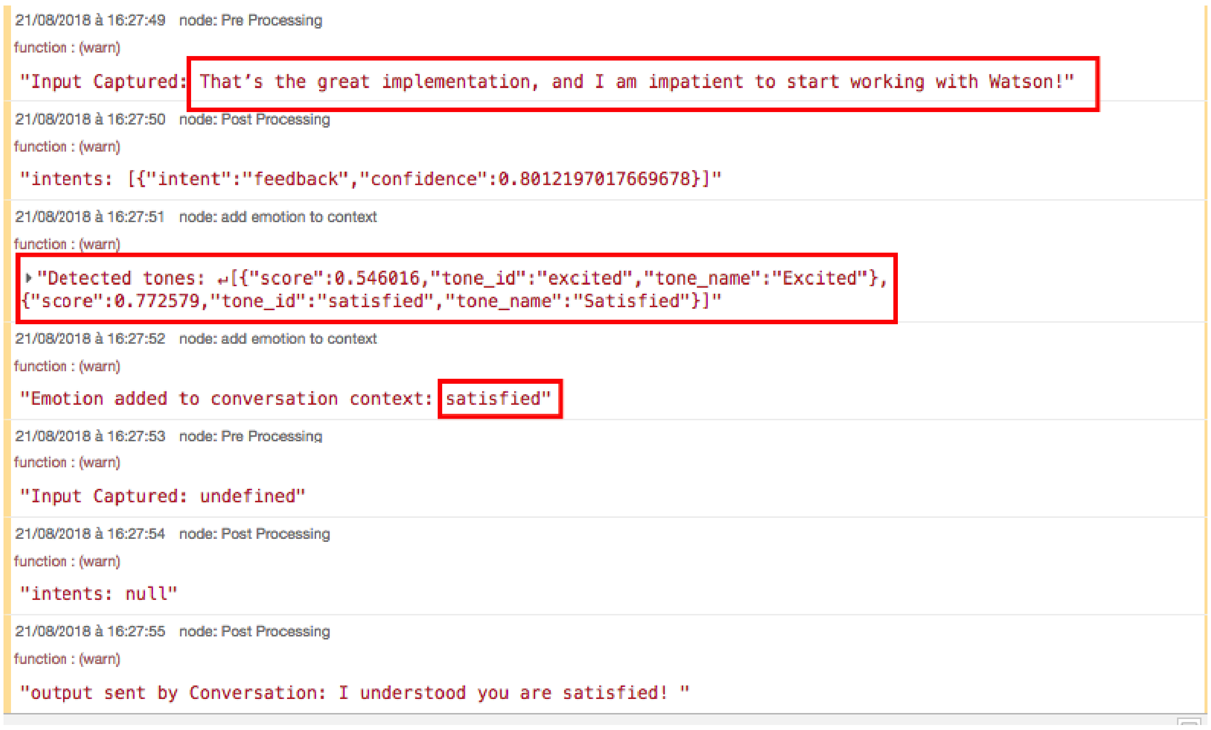
Tone analyzer returns successively:

$emotion = impolite

$emotion = sad

$emotion = satisfied

The debug pane should display for the last utterance



```
21/08/2018 à 16:27:49   node: Pre Processing
function : (warn)
"Input Captured: That’s the great implementation, and I am impatient to start working with Watson!"
21/08/2018 à 16:27:50   node: Post Processing
function : (warn)
"intents: [{"intent":"feedback","confidence":0.8012197017669678}]"
21/08/2018 à 16:27:51   node: add emotion to context
function : (warn)
▸"Detected tones: ↵[{"score":0.546016,"tone_id":"excited","tone_name":"Excited"},
{"score":0.772579,"tone_id":"satisfied","tone_name":"Satisfied"}]"
21/08/2018 à 16:27:52   node: add emotion to context
function : (warn)
"Emotion added to conversation context: satisfied"
21/08/2018 à 16:27:53   node: Pre Processing
function : (warn)
"Input Captured: undefined"
21/08/2018 à 16:27:54   node: Post Processing
function : (warn)
"intents: null"
21/08/2018 à 16:27:55   node: Post Processing
function : (warn)
"output sent by Conversation: I understood you are satisfied! "
```

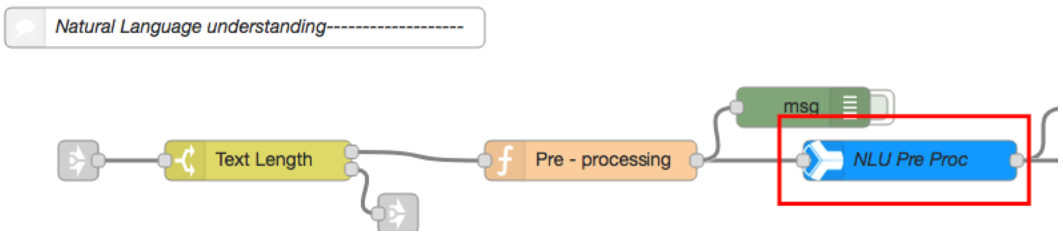
Enrich user inputs with Natural Language Understanding (NLU)

In addition to the input text provided by the user, it could be useful to extract entities, keywords which cannot be identify directly by Watson conversation Service.

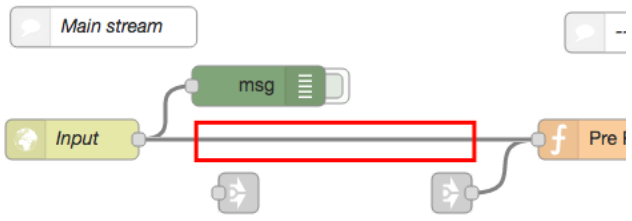
To do this , you are going to use NLU to extract entities, keywords, document emotion or language. It will be the opportunity to enrich variable context.

20. Configure the NLU service and node

1. Go back to the IBM Cloud catalog,
2. Look for **Natural Language Understanding,** click on the tile
3. Set the name of the service, keep *United Kingdom* as region
4. Click Create
5. Copy the Credentials *username* and *password* on your clipboard.
6. Go back to your Node-Red application
7. Doubleclick on **NLU Pre Proc** node

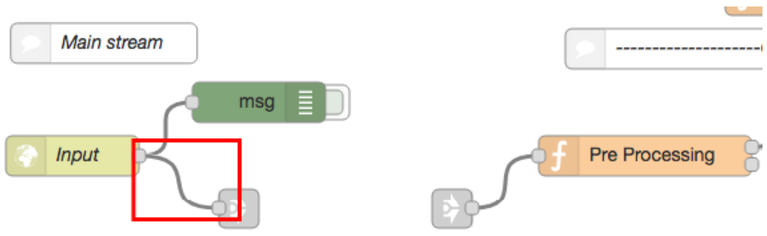


8. Fill the *username* and *password* with the values of your Tone Analyser service
9. Click **Done,** then **Deploy**
10. On the Main stream, select the link between **Input** and **Pre Processing** nodes

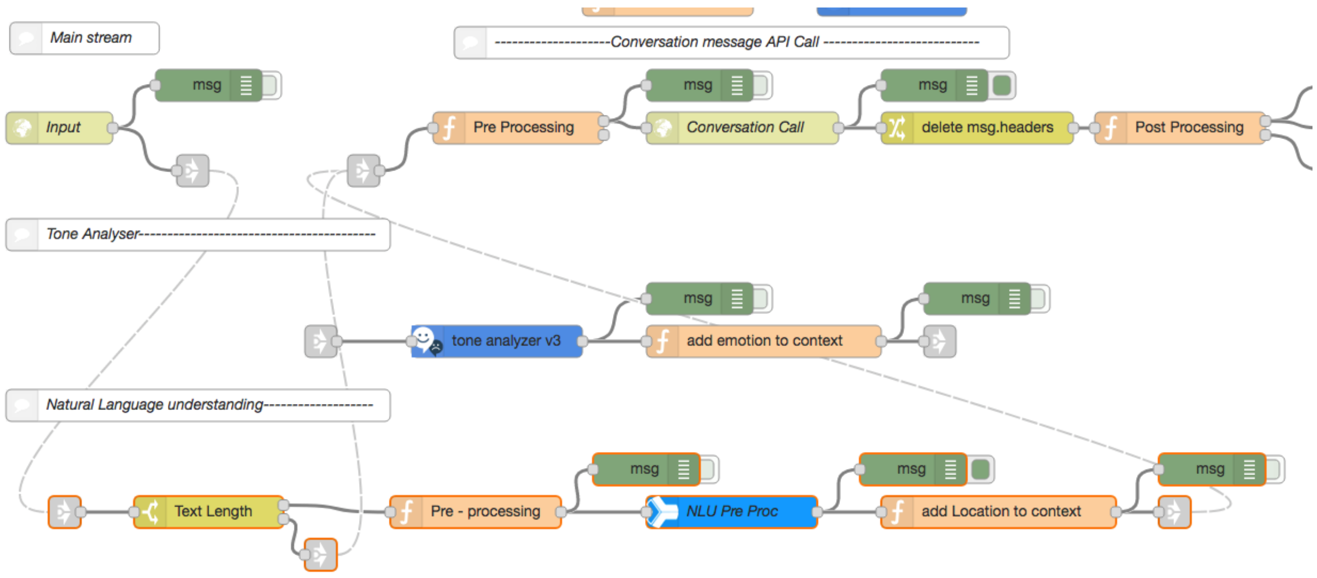


11. Delete it

12. Create a link between **Input** and **link** nodes



That the way to add NLU in pre-processing of the user input



Now your new component is ready to use.

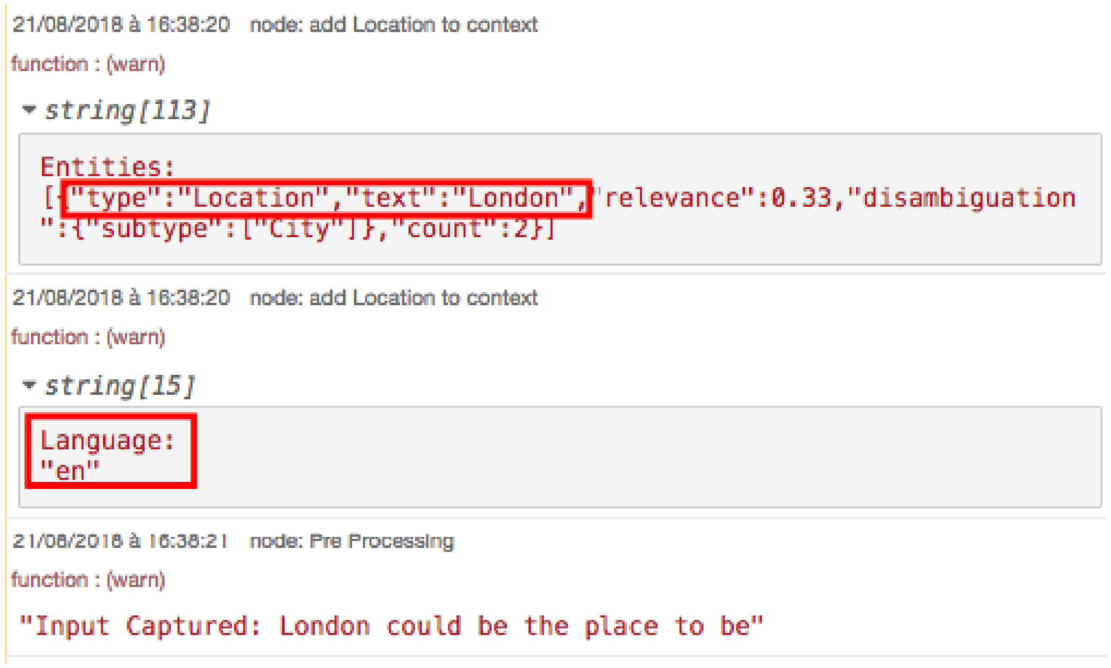
21. Test it

You are using input not manage by Watson conversation. Right now, you are just going to review the information provided by NLU and which could be leverage during a conversation.

1. On the Chatbot form enter the following sentence:

   London could be the place to be

In the debug panel, you should get the entity location London and the language en (English):

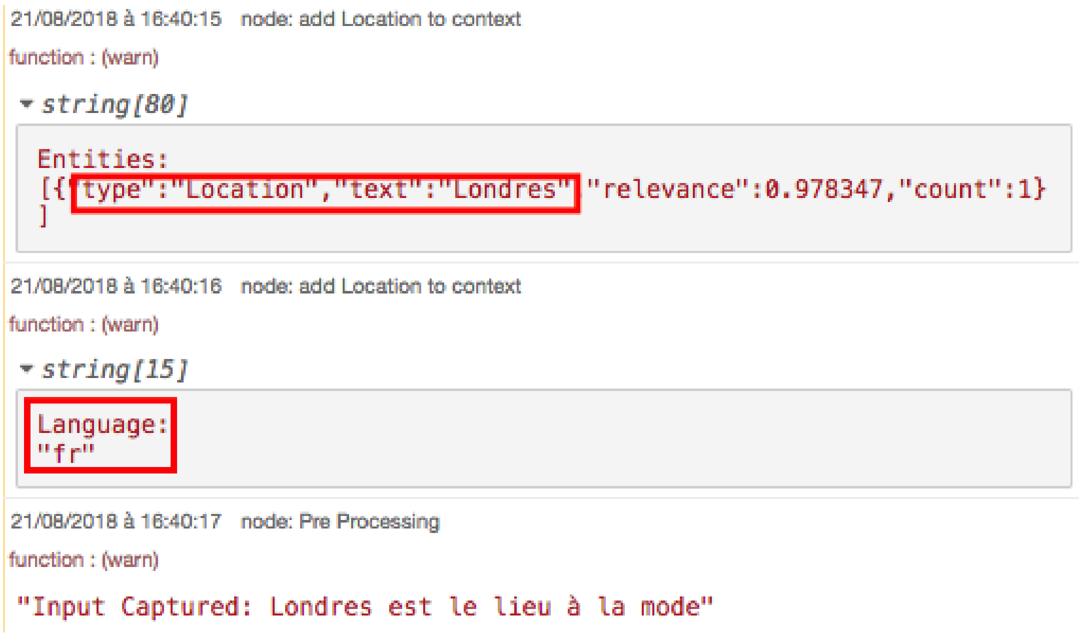


```
21/08/2018 à 16:38:20   node: add Location to context
function : (warn)
▾ string[113]
Entities:
[{"type":"Location","text":"London","relevance":0.33,"disambiguation
":{"subtype":["City"]},"count":2}]

21/08/2018 à 16:38:20   node: add Location to context
function : (warn)
▾ string[15]
Language:
"en"

21/08/2018 à 16:38:21   node: Pre Processing
function : (warn)
"Input Captured: London could be the place to be"
```

2. On the Chatbot form enter the following sentence:

   Londres est le lieu à la mode

In the debug panel, you should get the entity location Londres and the language fr (French):

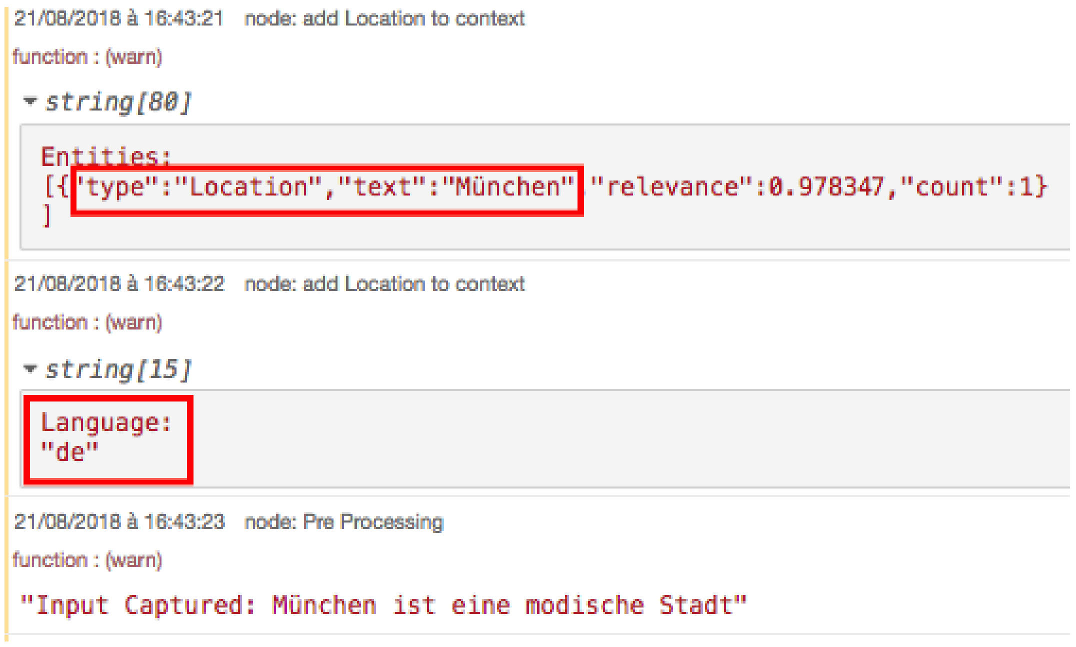


```
21/08/2018 à 16:40:15   node: add Location to context
function : (warn)
▾ string[80]
Entities:
[{"type":"Location","text":"Londres","relevance":0.978347,"count":1}
]

21/08/2018 à 16:40:16   node: add Location to context
function : (warn)
▾ string[15]
Language:
"fr"

21/08/2018 à 16:40:17   node: Pre Processing
function : (warn)
"Input Captured: Londres est le lieu à la mode"
```

3. On the Chatbot form enter the following sentence:

   München ist eine modische Stadt

In the debug panel, you should get the entity location München (Munich) and the language de (german):



```
21/08/2018 à 16:43:21   node: add Location to context
function : (warn)
▾ string[80]
Entities:
[{"type":"Location","text":"München","relevance":0.978347,"count":1}
]

21/08/2018 à 16:43:22   node: add Location to context
function : (warn)
▾ string[15]
Language:
"de"

21/08/2018 à 16:43:23   node: Pre Processing
function : (warn)
"Input Captured: München ist eine modische Stadt"
```

It is an illustration of the capabilities , in pre-processing of WCS, to :

- determine the language and then call the right workspace,
- extract information which could be useful to manage the conversation.