Watson Training

Watson Conversation Service

Lab Exercises

Course Code: Watson Training – Watson Conversation service

Version: **1.01**

Updated: November 14th, 2016

For questions about the Watson workshop and materials, contact:
Stephan Woehl *E-mail: swoehl@us.ibm.com*

Lab Overview

Lab 1

Lab 2

Lab 3

Lab 4

Lab 5

Lab 6

Lab 7

Lab 8

Lab 9

Change Log

Revision History: Changes to this document are summarized in the following table in reverse chronological order (latest version first).

Revision	Date	Change	Short Description of Changes
1.01	11/15/16	Added Lab 9	Lab 9 covers

Credits

Contents

Course overview

Credits

This workshop was created in collaboration with several teams and experts, who deserve credit for their dedication and great work.

- Joseph Kozhaya Scenario, code and insructions for Lab 2 and Lab 5, 6, 7
- Hiroaki Komine Scenario and instructions for lab 1
- Armen Pischdotchian Instructions for lab 2

If I missed anyone in this list, please shoot me a note. This project will continue as a collaboration between many experts within and outside of Watson in the hopes of broading skills within IBM, our clients, partners and the developer community.

If you have suggestions for how to improve this guide, please send a note to swoehl@us.ibm.com.

Objectives

During these lab exercises, you will use different IBM Watson services to build sample scenarios that can directly translate into

Lab 1 will use the new Conversation flow using the IBM Watson Conversation service. You will develop a conversation solution from scratch using the conversation tooling.

The tasks and tools that are covered should allow you to perform and complete all necessary steps to efficiently and effectively build and refine Intents, Entities and Dialogs for your Conversation flow.

Lab Structure

You should only perform numbered steps.
All other explanations are output or informational only.

This lab exercise is designed for every student to work independently using a Mac or PC. Each student requires their own IBM Bluemix account to create an IBM Conversation service instance in Bluemix.

Exercise Environment

The lab exercise will be performed on the Conversation Tool provided as part of the Watson Developer Cloud and create a Conversation flow hosted on a Conversation service instance.

Course Focus

The lab consists of three portions.

- The first one guides you through creating your own conversation flow from scratch.
- The second part uses those results to expand on the service and build and app on Bluemix.
- During the third part, you enhance the solution through speech-to-text (STT) and text-to-speech (TTS).

Preparation

To prepare for the lab exercises, locate the zip file provided to you by your instructor. Unzip the .zip file into a directory on your local hard drive.

The lab file contains the following files:

File name	Description
car_intents.csv	List of intents used in this lab
car_entities.csv	List of entities used in this lab
car_workspace_start.json	Starting point JSON file for the Conversation workspace
car_workspace.json	Final JSON file for the Conversation workspace

0 Getting Started

While labs can be performed using Windows and iOS, the current version of this guide assumes the use of a Windows computer to complete all tasks. The instructions will be updates for iOS in the coming weeks.

To perform these lab exercises, follow these instructions to set up your computer.

0.1 Overview

0.1.1 Bluemix

Bluemix is an implementation of IBM's Open Cloud Architecture, leveraging Cloud Foundry to enable developers to rapidly build, deploy, and manage their cloud applications, while tapping a growing ecosystem of available services and runtime frameworks. You can view a short introductory video here: http://www.ibm.com/developerworks/cloud/library/cl-bluemix-dbarnes-ny/index.html

The purpose of this guide is not to introduce you to Bluemix, which you should already be familiar with, at least on a high level.

0.1.2 Watson Developer Cloud (WDC)

IBM Watson Developer Cloud (**WDC**) is a platform of cognitive services, designed to help developers build solutions to help users extract insight from Big Data. Cognitive computing systems learn and interact naturally with humans to augment their ability to make better decisions from data.
As such, the Watson Developer Cloud services offer a variety of services that cover various aspects of natural interaction including text (**Alchemy Language**, **natural language classifier**, **Conversation**), images (**visual recognition**), and speech (**speech to text** and **text to speech**).

Furthermore, WDC offers services to understand a user's personality (**Personality Insights**) and emotional/social tone (**Tone Analyzer**) in a scalable manner.

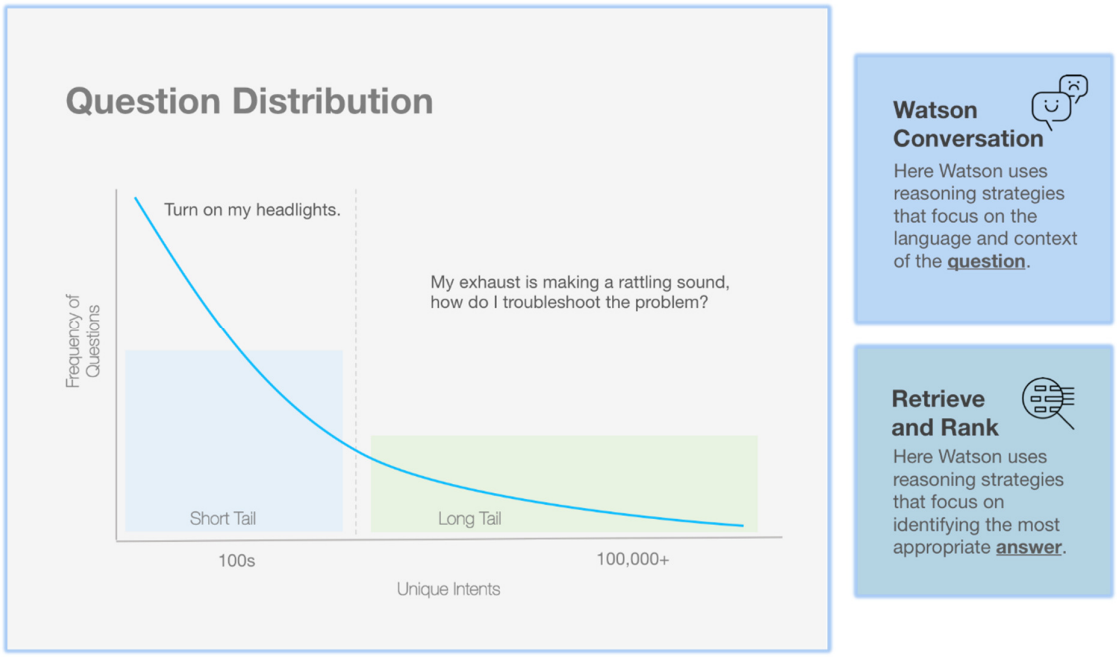
0.1.3 Watson Conversation Service (WCS)

This guide takes an instructional approach to working with the IBM Watson™ Conversation service, which you can use to create virtual agents and bots that combine machine learning, natural language understanding, and integrated dialog tools to provide automated customer engagements. Watson Conversation comes with an easy-to-use graphical interface to create natural conversation flows between your apps and your users.
Creating your first conversation using the IBM Watson™ Conversation service (WCS) entails the following steps:

- Train WCS to understand user input with example utterances and intents
- Identify Entities as the terms that may vary in your users' inputs
- Create Dialog to respond to your user's questions
- Test and Improve

IBM Watson Conversation service is designed to answer questions that are asked frequently by users. We call this the 'short tail' of a typical input distribution. The following graphic depicts that distribution.

Watson also provides 'long-tail' infrequently asked input/question solutions, such as Retrieve & Rank.
This guide will be expanded to address those capabilities ones they are made available.



0.2 Lab Scenarios

The scenario you will work with in this lab guide, consists of a cognitive car. Imagine you are sitting in such a car and can communicate with that car in natural language to control appliances. This means you can turn on and off certain devices, such as wipers, music, air conditioning, etc..

This solution also lends itself to being expanded to ask for the weather, get the latest news and of course communicate with your vehicle through speech. For example, the car can read the latest news for you that it locates based on a natural language search.

As part of the weather scenario, you will connect the Conversation service to an external service from a Weather Company (http://api.wunderground.com/?MR=1) to inform you of real time actual weather in the exact location that you are sitting and performing this lab. The service obtains your location using the browser's geo-location capabilities.

To see what your scenario is that you will be creating, watch this 14-minute video:
https://youtu.be/ELwWhJGE2P8

There are several labs in this document that can be completed independently.

0.2.1 Lab 1

This lab guides you through the use of the Conversation tool to create your own conversation flow (dialog). This will provide you with the basic understanding of the tasks to complete when creating your own conversation solution. What this lab does not cover are the broader best practices and considerations that you should follow when creating an enterprise-grade solution. We offer a separate workshop on those details.

0.2.2 Lab 2

It covers the conversation service more broadly, taking you through the tasks that need to be performed to run the application locally and to upload it to Bluemix.

0.2.3 Lab 3

This lab explains how to integrate the solution with cognitive speech capabilities such as speech-to-text (SST) and text-to-speech (TTS).

0.2.4 Lab 4

This lab expands the solution into a long-tail solution using the Retrieve and Rank (R&R) service. It will be added shortly.

0.2.5 Lab 5 & 6

The lab uses the Alchemy language service to expand entities.

At the end of this workshop, you should be ready to build your own integrated cognitive solutions using a variety of available services. Enjoy the cognitive journey you are about to undertake.

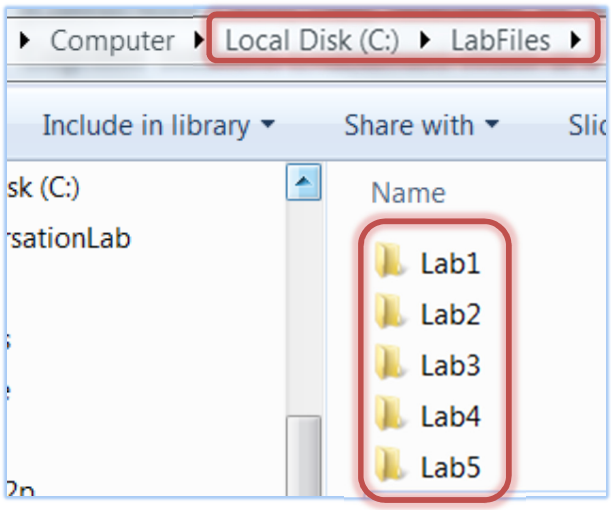
0.3 Prerequisites

This section provides instructions to help you get started quickly with the IBM Watson™ Developer Cloud services using Node.js as your programming runtime environment. To make it easy to get up and running with a functional application that uses the REST Application Programming Interface (API) for any Watson service, IBM provides a Node.js package with wrappers that simplify application development. The package includes simple command-line example applications to let you experiment with any of the available services.

1. To get set up, complete the following steps:

0.3.1 Creating Lab Directory Structure

2. Depending on your platform, create a directory structure, such as the screen capture below.



This directory structure will be used to complete all 5 labs currently covered in this lab guide.

0.3.2 Obtaining a Bluemix account

Bluemix is a cloud PaaS (Platform as a Service), which allows you to host your application on-line and bind it to a variety of SaaS service offerings from IBM including Watson analytic services.

Learn more at http://www.ibm.com/bluemix and if you are new to Bluemix, you can create a trial account at http://www.bluemix.net/.

3. Direct your browser to the Bluemix home page: https://console.ng.bluemix.net/home/ to access your dashboard
4. If you do not yet have a Bluemix account, click **Sign Up** on the top right
5. Enter requested information and click **Create Account**

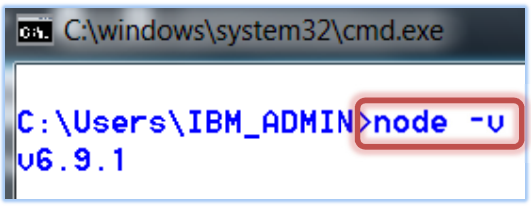
If you use your personal e-mail address, you have 30 days to evaluate Bluemix. Some services, such as Conversation, continue to be free for limited use after the trial period.

0.3.3 Installing the node.js runtime

The default installation includes both the runtime and package manager. Make sure to include the installed binaries on your PATH environment variable after installation (typically, the default installation locations that the installer selects does the inclusion).

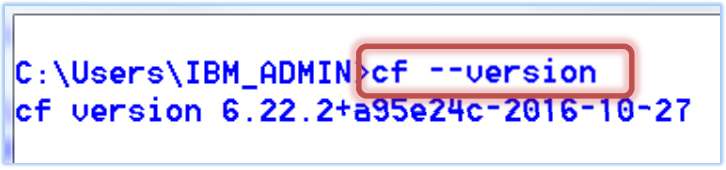
6. Direct your browser to the nodejs.org web site: https://nodejs.org/en/
7. Click **DOWNLOADS** in the menu bar
8. Select and install the installer (not the binary), appropriate for your operating system (i.e. Windows Installer.msi (64 bit)

9. Complete the installation
10. Verify that node.js has been installed by opening a command line and issuing `node -v`



0.3.4 Installing cf (Cloud Foundry) CLI

11. Direct your browser to a GitHub repository: https://github.com/cloudfoundry/cli/releases
12. Download and install the most recent installer appropriate for your operating system
13. You may need to open Preferences > Security and Privacy > General tab (in Mac)
14. Unlock and change the **Allow applications downloaded from Anywhere**
15. Verify that installation was successful by issuing `cf --version` from a command line



0.3.5 Downloading OS-specific coding-friendly editing tool

16. If you are using a PC, you can use **Notepad ++** (https://notepad-plus-plus.org/download)
17. For Mac, use **Sublime Text** (http://www.sublimetext.com/3/) → also works for Windows

0.3.6 Obtaining the Weather Company API

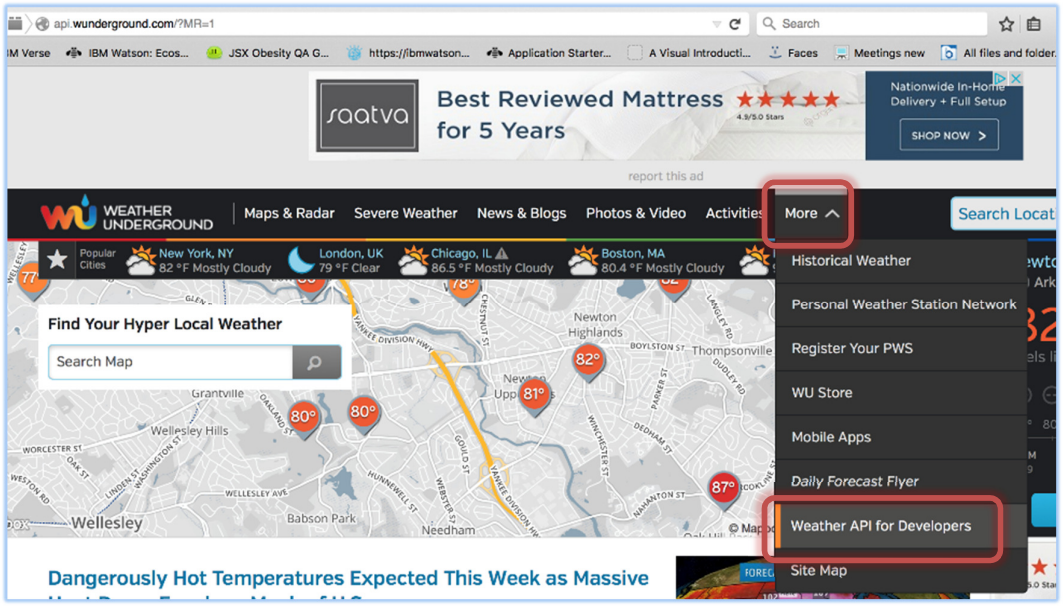
This section is only required for labs 2 and 5 in this guide. You can skip this if you are not planning on going through those labs.

Currently, the following desktop browsers support the W3C Geolocation API:

- Firefox 3.5+
- Chrome 5.0+
- Safari 5.0+
- Opera 10.60+
- Internet Explorer 9.0+

18. Point your browser to the following site: http://api.wunderground.com/
19. Create an account and sign in.

20. From the **More** pull down menu, click **Weather APIs for Developers**



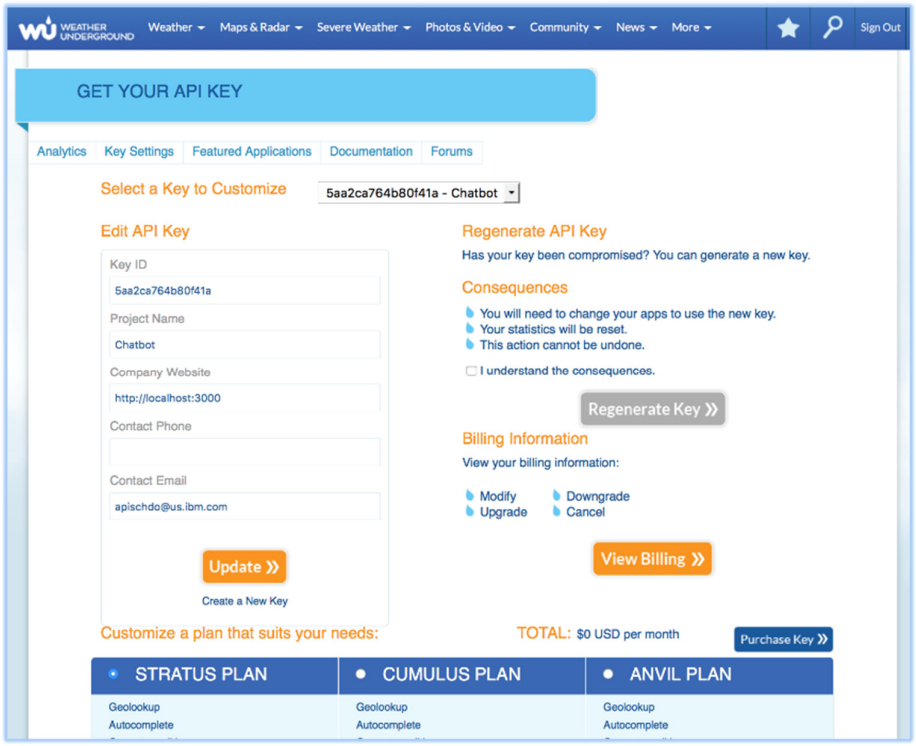
21. Go to **Pricing**

22. Make sure that Cick **Purchase Key**

23. Follow the on-screen instructions to obtain the key. All fields must be filled.
Below is an example from my experience (use http://localhost:3000 for the purposes of this lab).

24. Under the **Key Settings** tab, request your own JSON key

25. Refer to your email for verification.



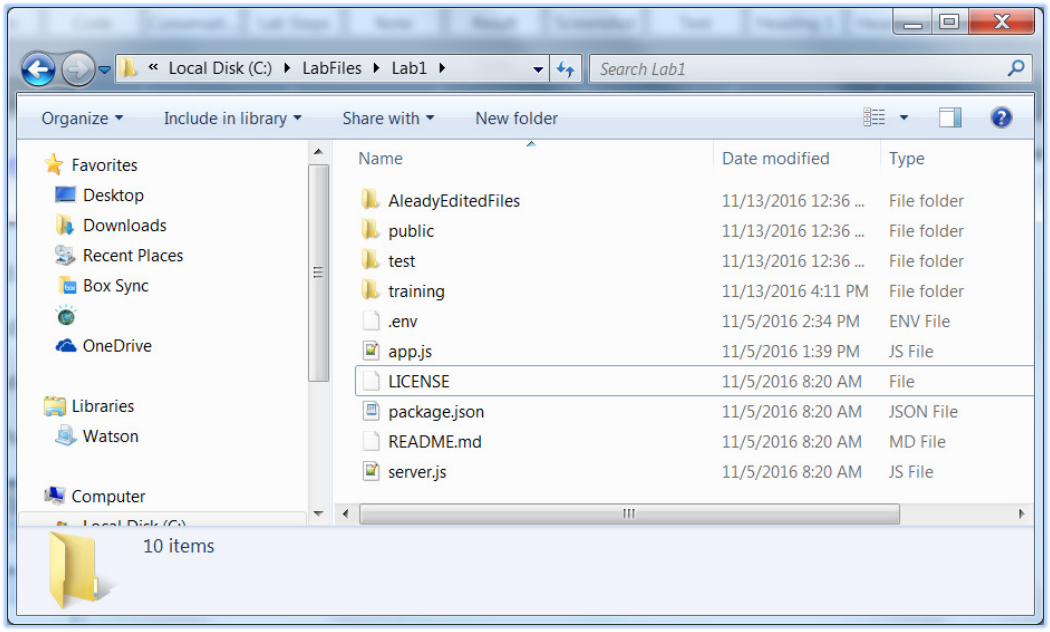
26. Once you have your key, copy and paste it into a text file and store it under your \LabFiles\ directory.

0.3.7 Obtaining Lab Files

Each lab requires a set of files you download from github

27. Access https://github.com/swoeh/Conversation.git → You do not need to sign up
28. Type **ConversationLab** into the Search field
29. Click and download **Lab1.zip**
30. Copy and extract to your **\LabFiles** directory

    This creates a subdirectory, called \Lab1
31. Access the folder and review the content



You are ready to begin with Lab 1 of this guide.

1 Lab 1: Creating a Conversation using Watson Conversation Service

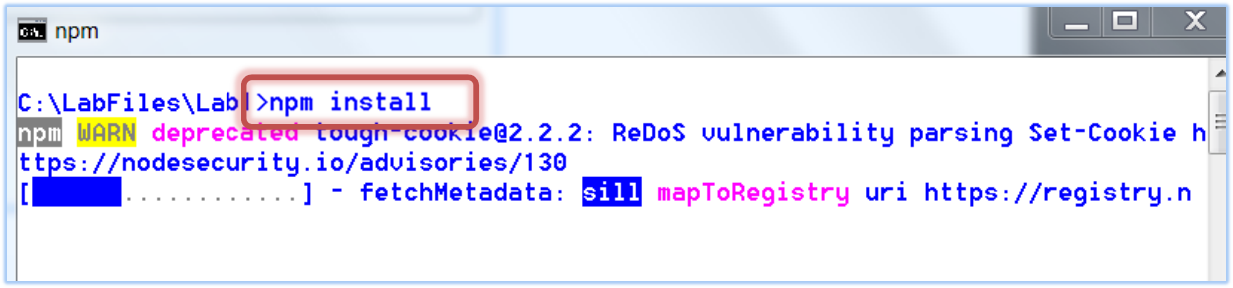
During this lab exercise, you will learn how to develop a Conversation application by using the Conversation service tool. At first you will review the documenttion for Conversation service which contains information for developers on how-to-develop a Conversation application. You will also review demo applications that utilize the Conversation service as part of a use case.

1.1 Prerequisites

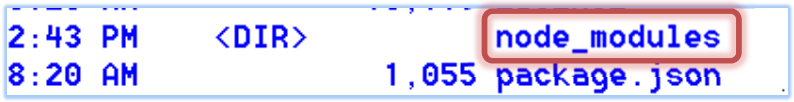
1.1.1 Installing node packages defined in package.json

1. Open a Command Line Interface or a terminal and change directory to **\LabFiles\Lab1**

2. Install package with npm by issuing the command `npm install`



Notice that this adds a new folder named node_modules.

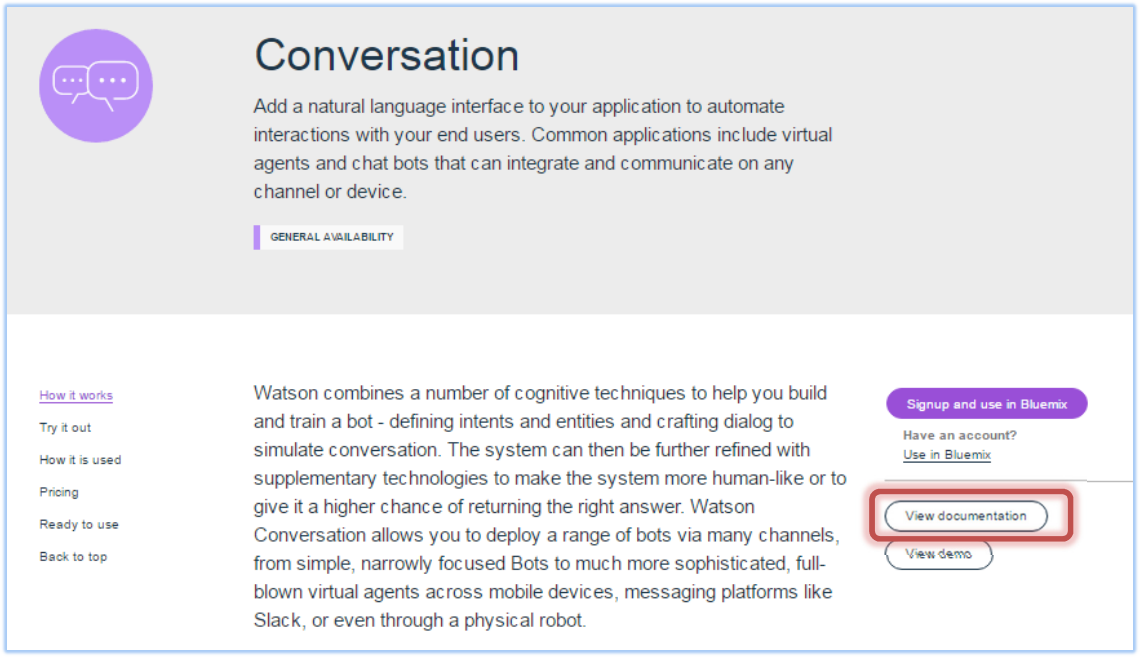


1.2 Reviewing the Conversation Service Documentation

Check the location and contents of Conversation service document

1. Open the following URL to open the Conversation service page:
   http//www.ibm.com/watson/developercloud/conversation.html

   From this page, you can navigate to the documentation page and demonstrations of Conversation service.

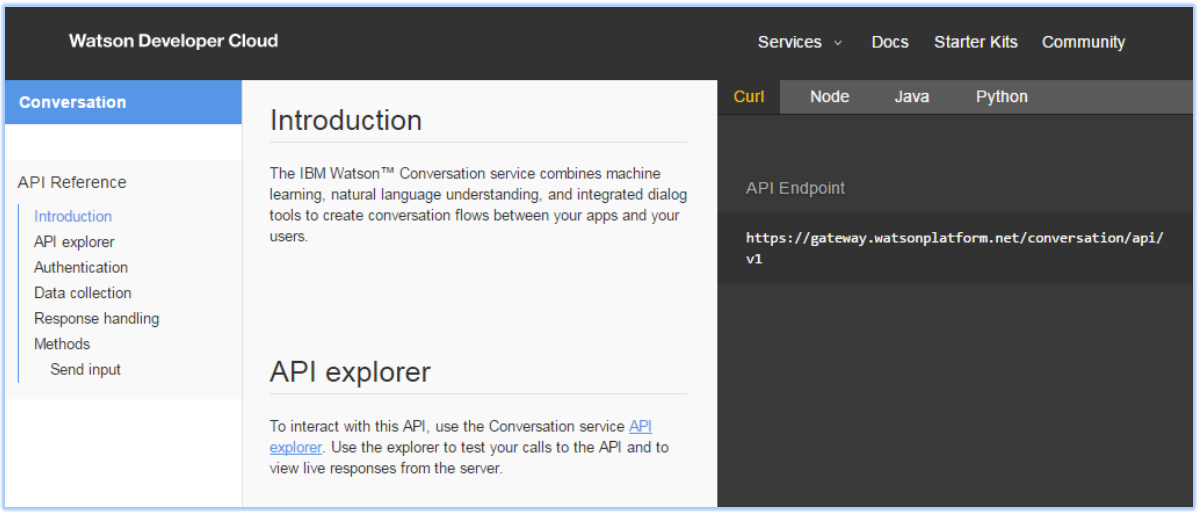


2. Click **View documentation** to open the Conversation documentation.

The Conversation service documentation explains the basic concepts, key features (Intents, Entities, Dialog) and detail of application development.

During this lab exercise, you will perform all steps of developing a Conversation flow for a Car Dashboard use case.

3. Click **API Reference** to find more information about the Conversation APIs you can use on the Conversation service.



At this point, we publish only one API for sending a user input text to the Conversation workspace. The API returns a response message from the Conversation workspace along with addition information related to the current conversation session, including context, detected intents and

entities. These can be used by an application to provide a better user experience on Conversation enabled applications.

Although you can test your Conversation flow on the Conversation tools, you can also submit direct API calls to review detailed data and error messages for an API call response.

1.2.1 Conversation demo applications

Watson Developer Cloud (WDC) provides three types of demo applications on Conversation service:

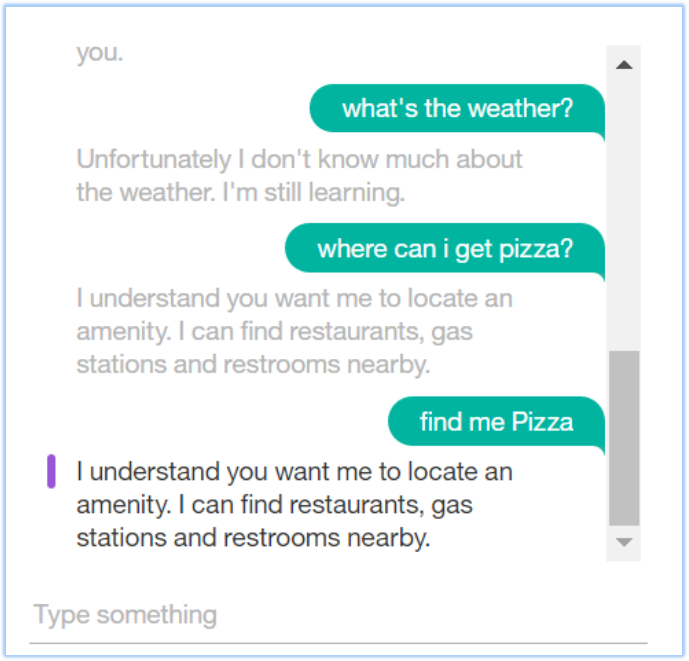
Conversation simple app

See the demo

This Node.js app shows how the Conversation service uses intents in a simple chat interface. It demonstrates the conversation with an end user and the JSON responses at each turn of the conversation.

4. Click the link or enter http://conversation-simple.mybluemix.net/ to see the demo

5. Type a few commands or questions to see how the application performs



You will build a similar application from scratch during Lab 1 of this guide.

Conversation Enhanced app

This Java app demonstrates the combination of the Conversation service and the Retrieve and Rank service. First, users pose questions to the Conversation service. If Conversation service is not able to confidently answer, a call is executed to the Retrieve and Rank service to provide the user with a list

of helpful answers. Use this user interface to accelerate the development of your own conversational apps.

See the demo

Cognitive car demo

This Node.js app is a fully developed example of the type of app you can build with the Conversation service. It shows how the Conversation service uses intents, entities, and dialog.

See the demo

1.3 Your lab 1 exercise example

In this Lab exercise, you will create a Conversation Workspace that is used in the **Conversation simple app** while using the Conversation Tool.

Conversation Enhanced app and **Cognitive car demo** contain similar Conversation workspace designs, but these applications are enhanced by including other Watson services such as **Retrieve and Rank** within the **Conversation Enhanced app** and an animation UI in the **Cognitive car demo**.

Those details can be found in the source code published on GitHub.

1.4 Creating a Conversation service instance and importing a workspace (JSON)

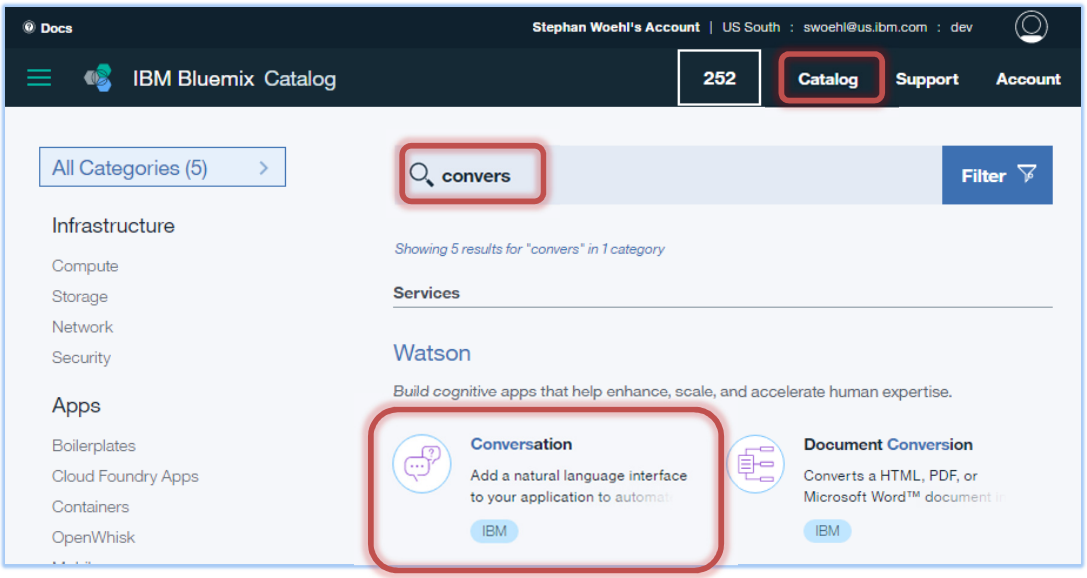
1.4.1 Creating a Conversation service instance

A Conversation service instance is created on the IBM Bluemix console. The procedure is identical to that of creating other Watson services.

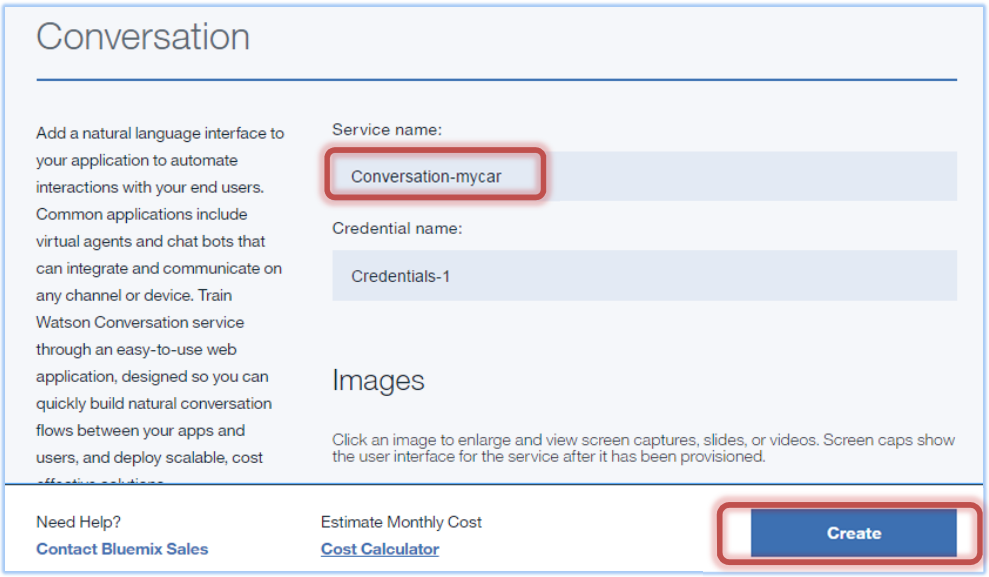
6. Access the Bluemix console by entering the URL https://console.ng.bluemix.net/ and logging-in with your Bluemix ID.

7. Access the **Catalog** link

8. Locate **Conversation service** by starting to type "convers…" into the search field

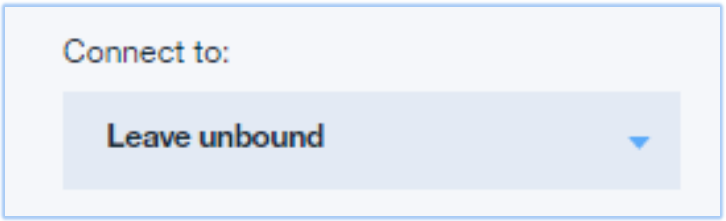


9. Click **Conversation**



10. Enter a meaningful **Service name**, such as *Conversation-mycar*

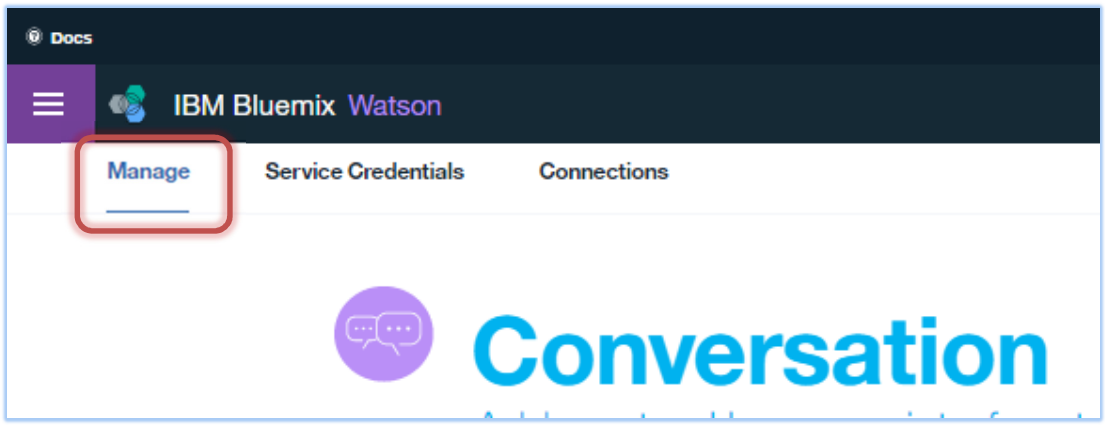
11. Accept the default **Credential name** and under **Connect to**, leave the service unbound



12. Click **Create**

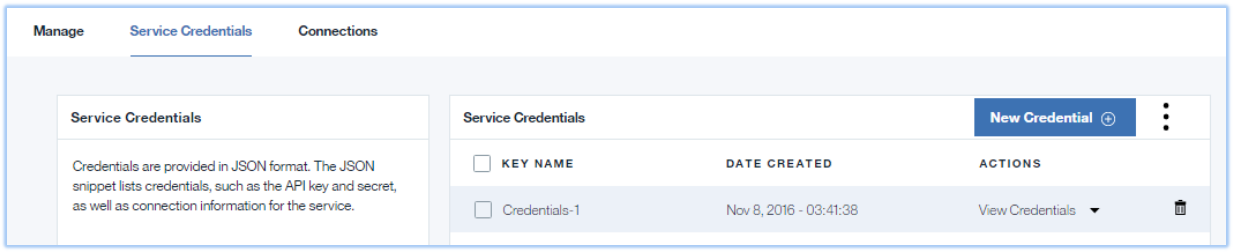
Creation of the service is usually finishes within a few seconds but may take up to a few minutes.

Once the Conversation service instance has been created, the **Manage** tab in the Service dashbord opens.



The page includes a green **Launch tool** button that launches the conversation tool to build a conversation flow, or dialog.

13. Click **Service Credentials** in the upper left corner of your dashboard



The service credential tab shows your credential information.

You need these credentials in order to access the Conversation API through cURL, or SDK.

1.5 Importing a Workspace (JSON) with Intents and Entities

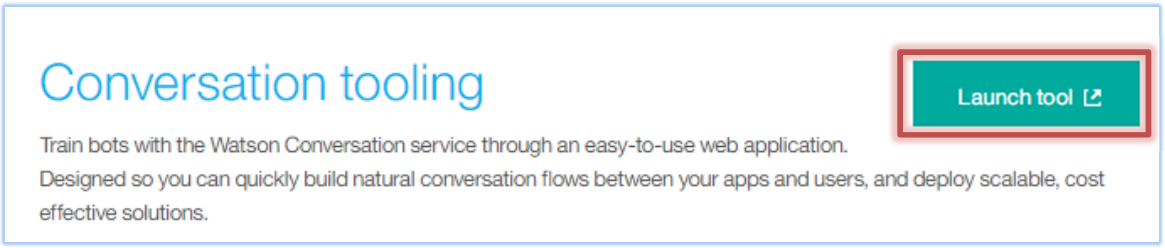
When you create a solution for your own use case, you develop a new Conversation workspace from scratch. This requires you to specify each intent and entity as part of your workflow.

This guide does not cover rules and best practices for collecting and developing questions, intents and entities.

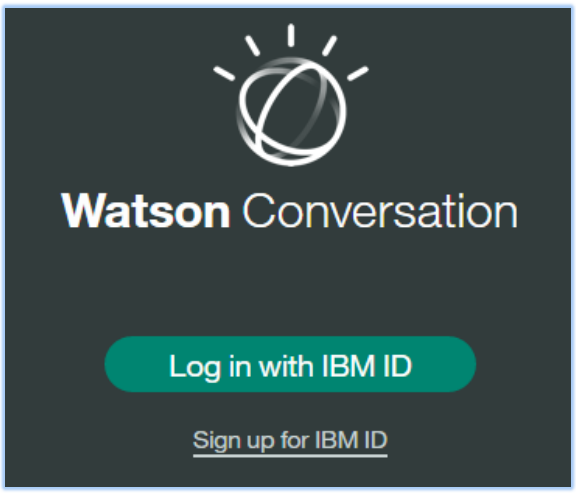
To expedite this lab, you will import an existing JSON file that contains a partial design of conversation workspace, including all intents and entities. You resume your design from there by building the conversation (dialog).

Use an existing JSON file that contains all intents and entities.

14. Go back to **Manage** within the Conversation Service dashboard

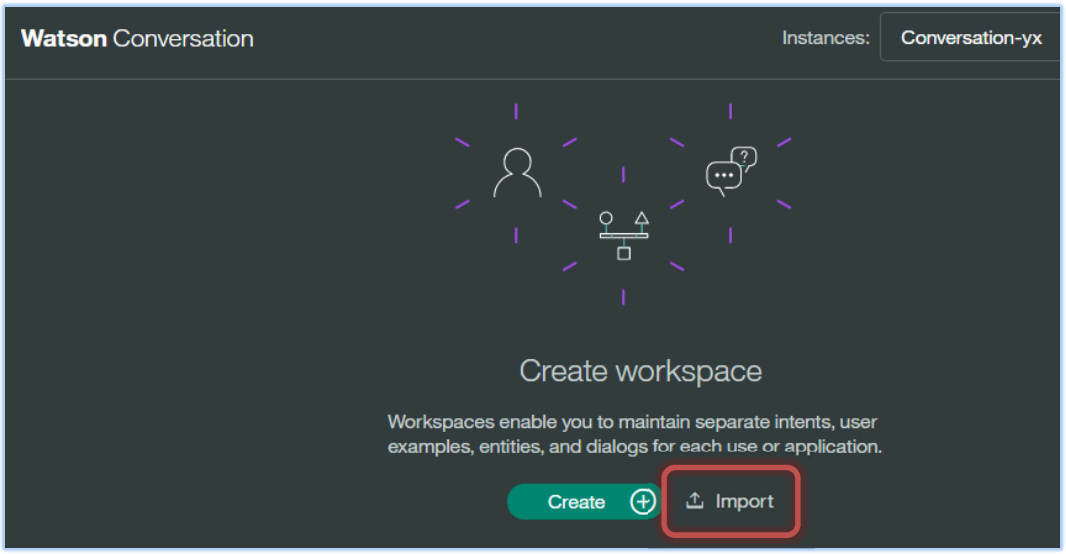


15. Click the **Launch Tool** button to launch the Conversation tool

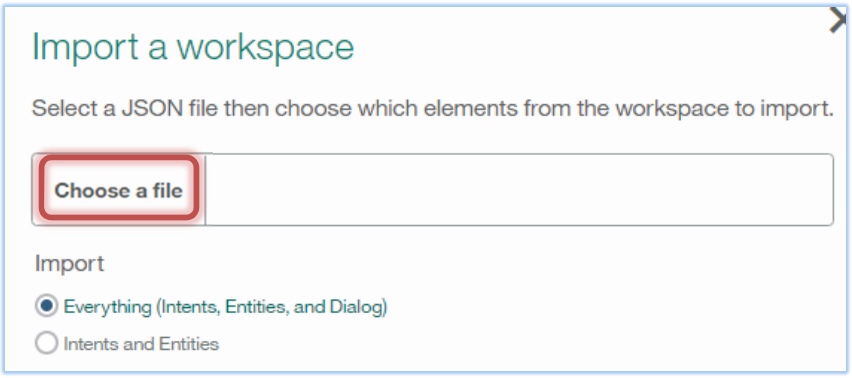


16. If you are prompted to log in, click the **Log in with IBM ID** button

    You should be logged-in automatically using the Bluemix credentials you entered earlier.

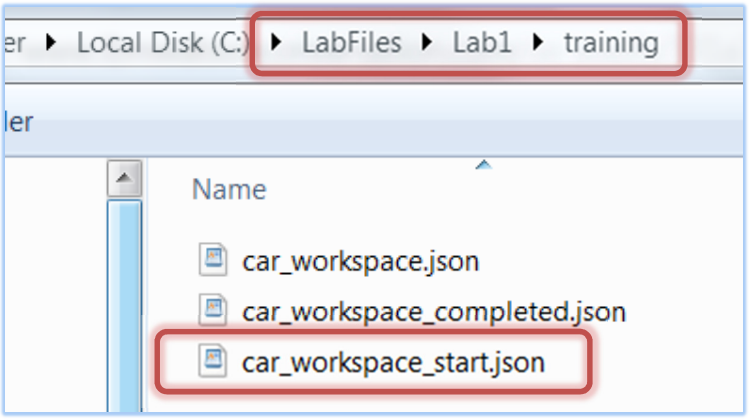


17. On the **Create workspace** page, select **Import**



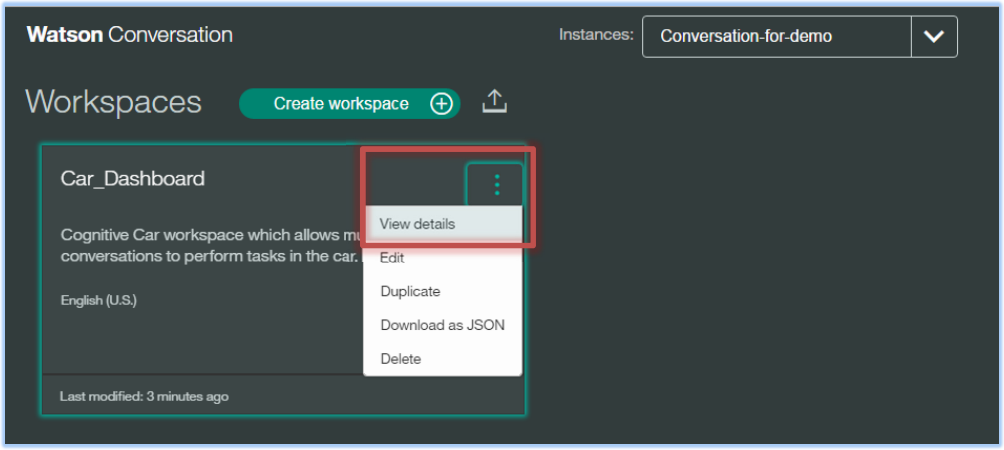
18. Select **Choose a file**

19. Locate the **car_workspace_start.json** file in your **\LabFiles\Lab1\training** directory



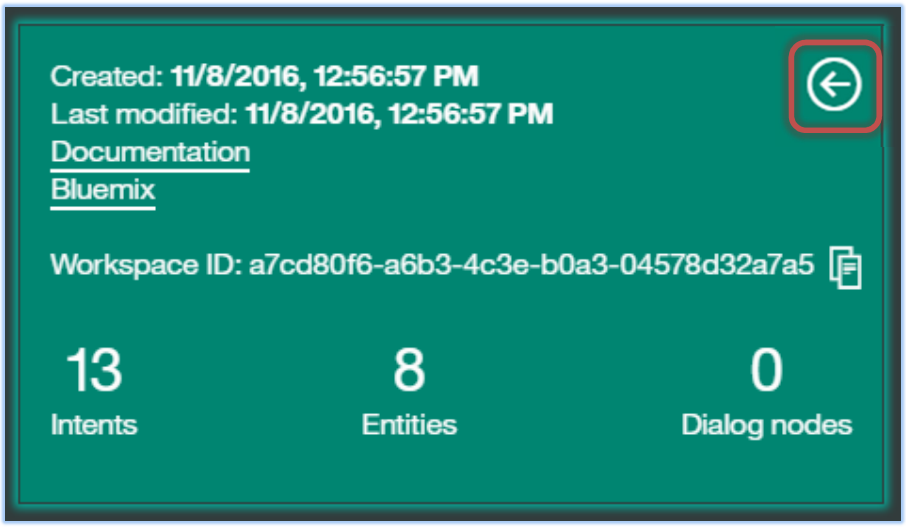
20. Click **Open** and then **Import**

This creates a Car_Dashboard workspace.

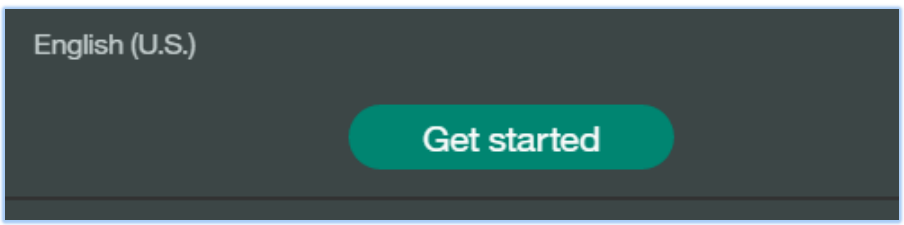


21. To access additional details, click the three dots in the upper right corner of the Workspace rectangle and select **View details**

    This opens additional details about the workspace, including the Workspace ID and information about number of intents, entities and dialog nodes.



22. Click the back arrow to go back to the workspace front page.

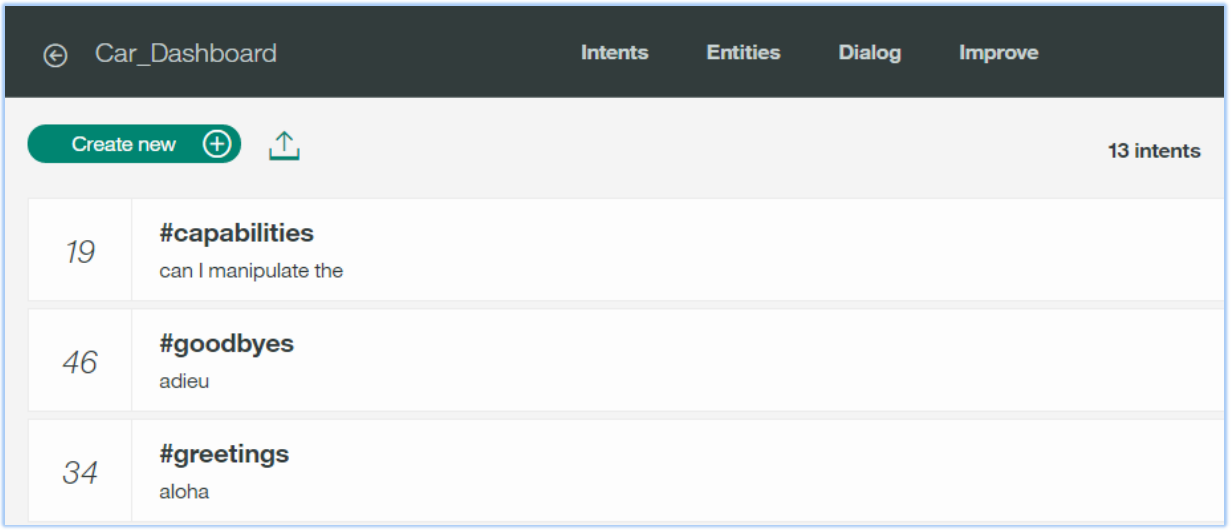


23. To open the workspace and start working on intents, entities and dialog nodes, click **Get started**

1.6 Reviewing Intents and Entities

This opens the Intents page, showing the intents, including the number of user examples for each.

Intents are displayed with a "#" prefix.



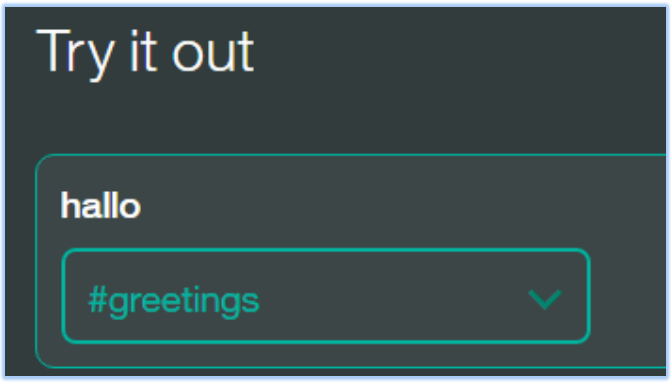
24. Click the second intent (#goodbyes) to view the different ways the service was trained for users to say "goodbye".

    This does not mean that these are the only ways the system will be able to recognize a user ending a conversation. Machine learning allows the system to recognize other variations of saying goodbye that will also lead to ending a conversation.

25. To test this, click the **Ask Watson 'Try it out'** icon in the upper right corner of the window

26. Enter **`hallo`** (which is German for hello)

    As you can see, the service correctly identified the intent as #greetings

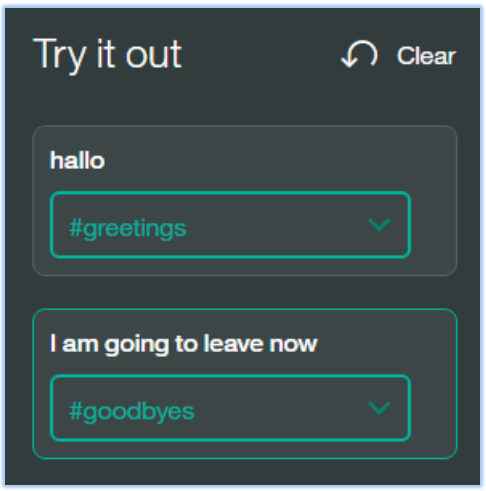


27. Click and expand the #greetings intent on the left and determine whether the system had been trained for "hallo" → You will find it has not

    You can see that Watson has been trained in multiple languages, but not German. Yet it works.

28. Enter `I am going to leave now` into the **Try it out** section

    You see that the service recognizes your first entry as #greetings intent and your second one as #goodbyes intent, even though neither was specifically mentioned in the intents section.

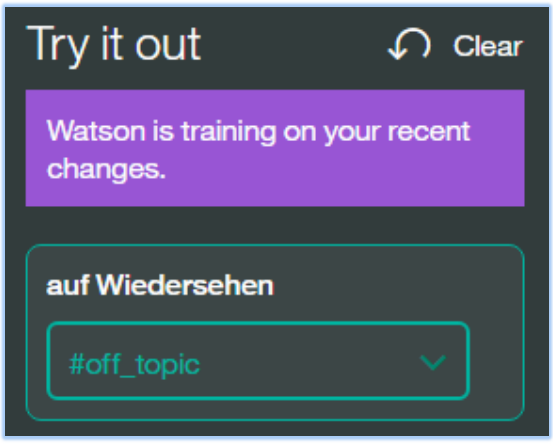


29. To verify that this intent does not exist either, expand the #goodbyes intent in the left pane of the window.

30. If you have a different way to say hello or goodbye and the system does not recognize it, try adding it ot the respective intent. As an example, add **"auf Wiedersehen",** which is German for goodbye

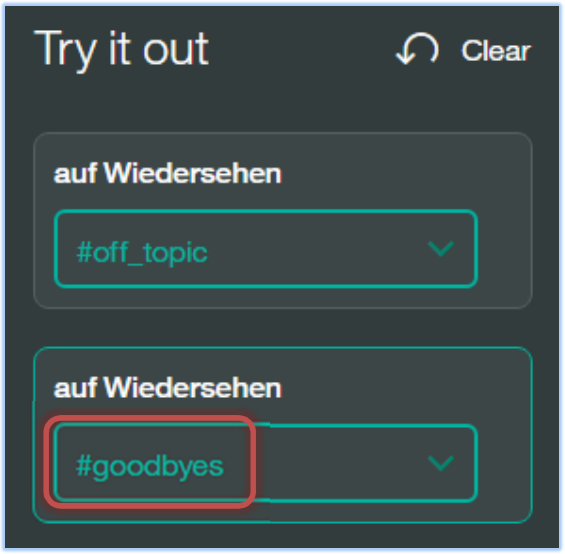
31. Type the new greeting into the **Try it out window**

    It shows that the service is being trained. During that time, the Intent is not being recognized correctly.

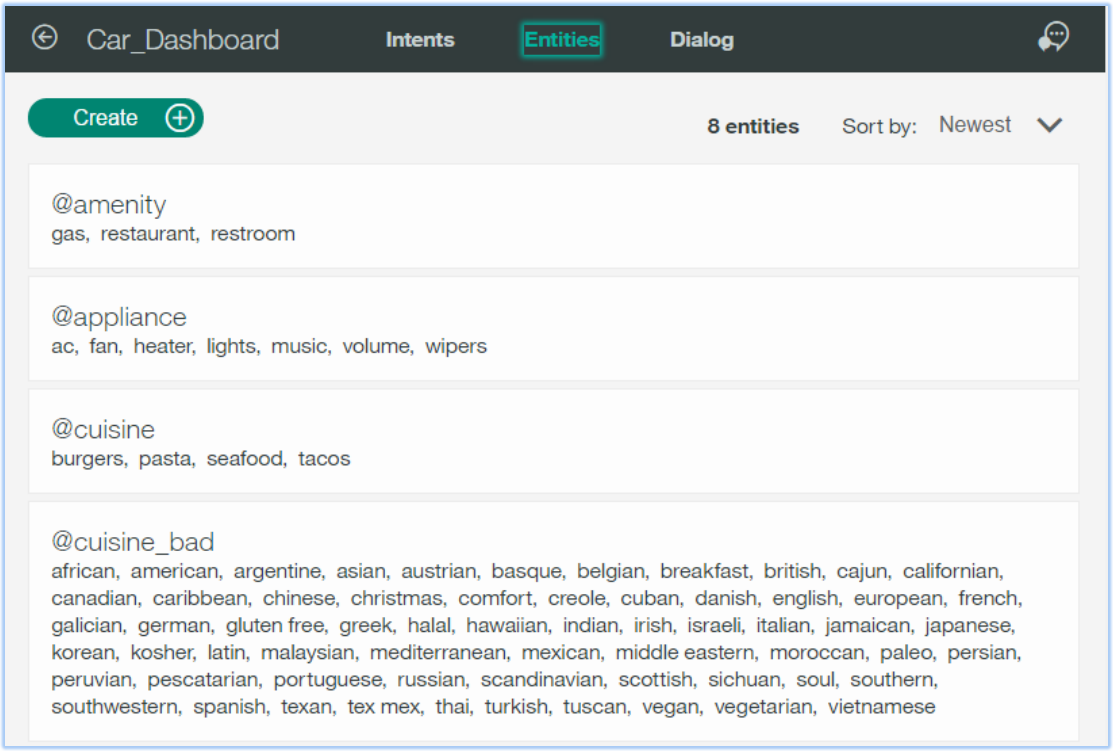


It may take some time for the training to finish.

32. Once the message disappears (turns green), try entering the message again.
→ If it takes too long, continue with the exercise and try again later.



33. Now access the **Entities** section by clicking the entry in the window header



Entities are displayed with a "@" prefix. Each entity is collasped and displays all entity values.

1.7 Your Use Case

The conversation flow you will create in this lab exercise, demonstrates how the Conversation service uses Intents and Entities in a simple chat interface.

You are creating a car dashboard that allows a driver of a car to make requests, or ask questions in natural language.

Intents and entities are extracted from those requests and presented as JSON data in the response from the Conversation API call.

An application using the Conversation workspace, can take appropriate action based on that information. Usually, a combination of intent and entity results in a separate response or action.

To get started, the Conversation service should accept the following requests from the driver:

```
"Turn on radio."
"Start wiper"
"Where is the nearest gas station?"
```

The system should also be trained on various iterations of those statements and requests.

Your intents and entities you already uploaded into your workspace, should handle the majority of those iterations already.

1.7.1 Intents used in this Lab Exercise

Here are the intents the system has been trained for already:

Intents	Representative questions	Description
greetings	Hello, how are you?	Just speaking a greeting message
goodbyes	bye now	Just speaking a goodbye message
turn_on	turn my lights on or me	Request for turning on an equipment
turn_off	turn off my wiper please	Request for turning off an equipment
turn_up	can you increase the	Request to increase music volume
turn_down	turn it down	Request for decreasing music volume
phone	dial Home	Request for making phone call
traffic_update	Estimated time left?	Question for traffic or driving status
weather	how cold is it out?	Question for weather
locate_amenity	i need to stop for gas	Request for finding local amenity
capabilities	what can you do for me	Question for dashboard's capability
off_topic	book a flight for NY on sunday	Request or question that this dashboard can not handle
out_of_scope	how do i adjust exterior mirrors	Request or question that this system is not design to handle

The complete list of intents is available in **car_intents.csv** within the \LabFiles\lab1\training\ folder.

1.7.2 Entities used in this Lab Exercise

Here are the entities that are already included in the workspace.

Entities	Values defined in the entity	Description
amenity	gas, restaurant, restroom	Amenity list that this dashboard can find.
appliance	ac, fan, heater, lights, music, volume, wipers	Appliance list that you can control through this dashboard.
cuisine	brgers, pasta, seafood, tacos	Food types of the resultant that this dashboard can find.
cuisine_bad	african, japanese, vegetarian	Food types of the resultant that this dashboard cannot find.
Genre	Classical, jazz, Pop, rock	Music types that the dashboard like to play.
genre_bad	Blues, J-Pop, Latin	Music types that the dashboard do not like to play.
option	closest, first, second, third, fourth, fifth	Optional preference on finding amenities.
Phone	call, text	Type of phone call

All Entities can be found in **car_entities.csv** in the \LabFiles\Lab2\training\ directory.

1.8 Creating a simple dialog node

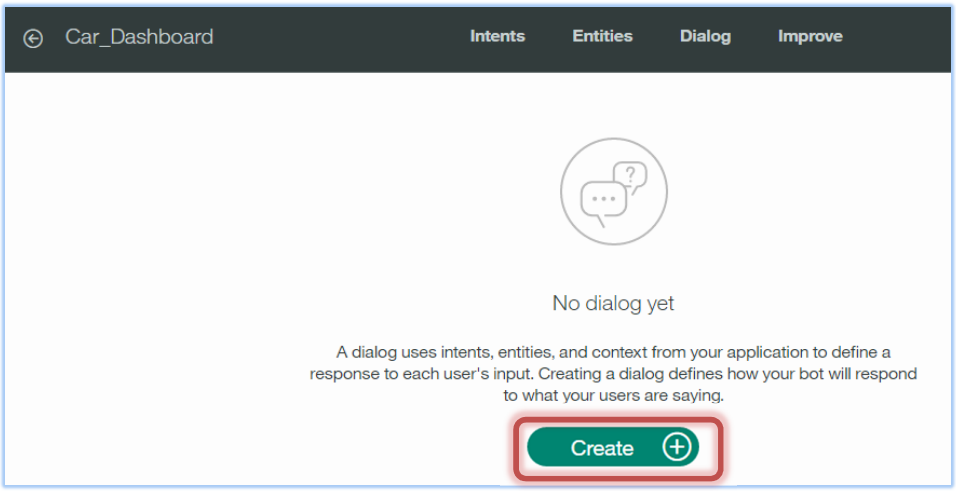
In this chapter, you will create a simple conversation flow that uses basic intent matching conditions. You then test the conversation flow you created using the **Try it out** testing tool and providing feedback to the current Conversation flow design.

1.8.1 Creating a simple dialog node for verifying intents

Even though the system recognizes most intents and entities, it does not yet return any responses.

To respond to user input, create a simple dialog node which matches the intent of the user input and returns a response.

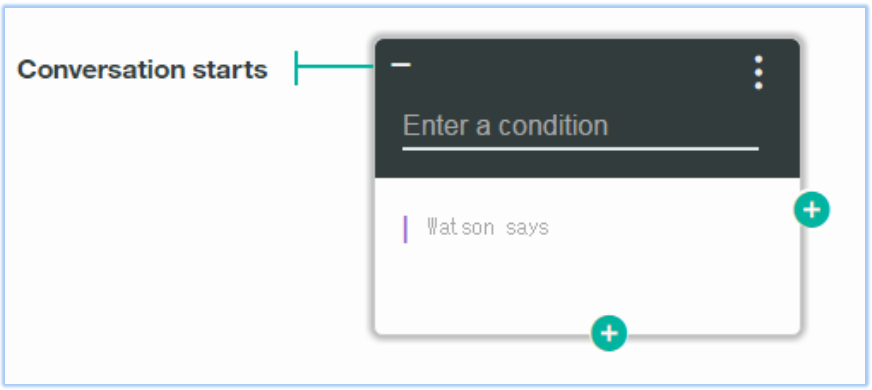
34. Select the **Dialog** tab in the Conversation tool.



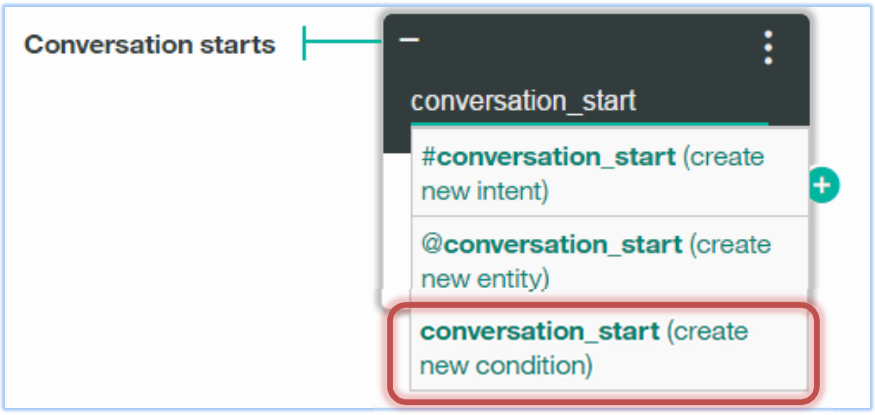
The page shows that no dialog has been defined.

35. Click **Create**

An empty dialog node is automatically created.



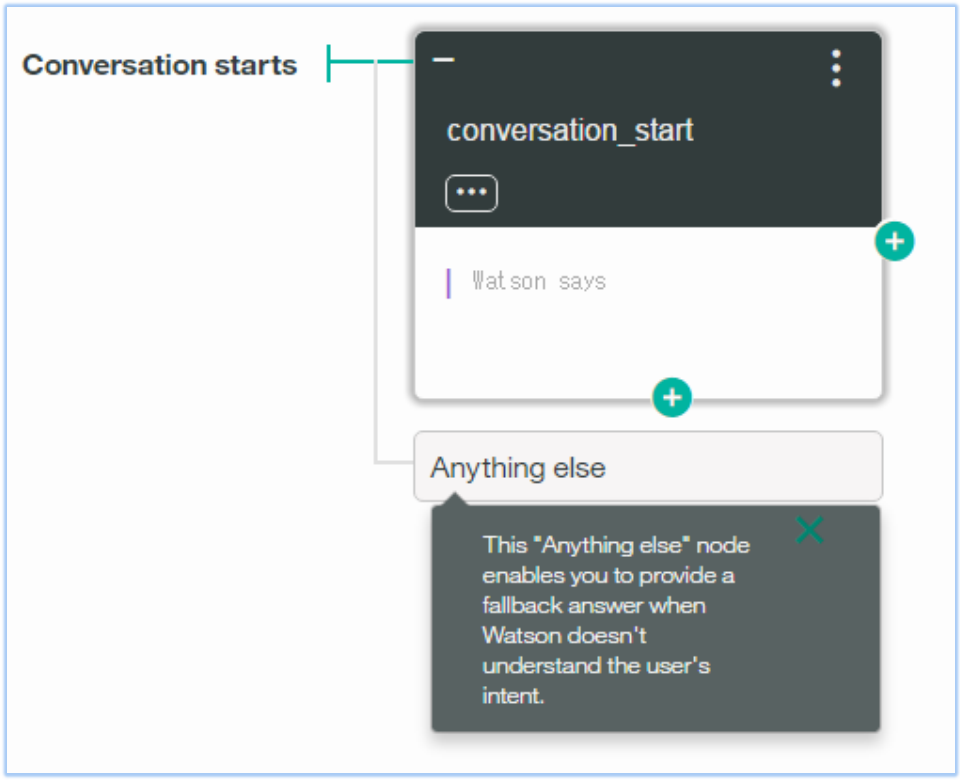
36. Type **`conversation_start`** in Enter a condition field



37. From the options displayed, select conversation_start (create new condition)

The [conversation_start] dialog node is the starting node for any new conversation the user initiates. It is also the contextual node for when there are no more child dialog nodes available as part of a conversation.

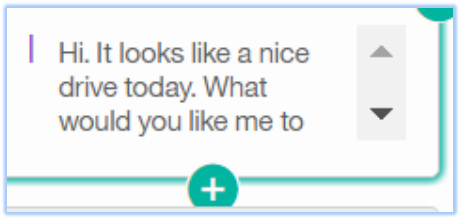
Besides the [conversation start] node, an [Anything else] node has also been created automatically.



The [Anything else] node should be the last node in every conversation. It is being used when no match occurred in the conditions of the top-level nodes. By specifying an [Anything else] node, the conversation will always return a response to the user.

38. Select the [conversation_start] node and in the **Watson says** field, enter:

    Watson says: **`Hi. It looks like a nice drive today. What would you like me to do?`**



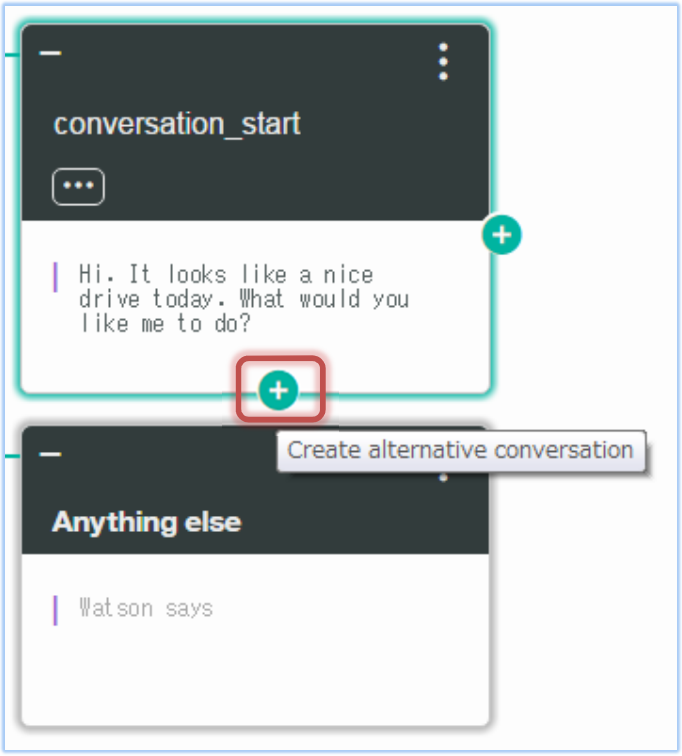
39. Select the [Anything else] node and in the **Watson says** field, enter:

    Watson says: **`Sorry, I do not understand.`**

You will now add some dialog nodes to cover greetings and goodbyes.

40. Select the **conversation_start** node and click the + sign at the bottom of the node

    This creates an alternative conversation sibiling node.



41. In the new node, click the dark grey condition field and specify: Condition: **`#greetings`**

    Note: The type-a-head feature helps you type into the condition field. Typing the first few characters of an intent displays possibly input candidates.

42. When prompted, select the first entry #greetings, which represents the '#greetings' Intent

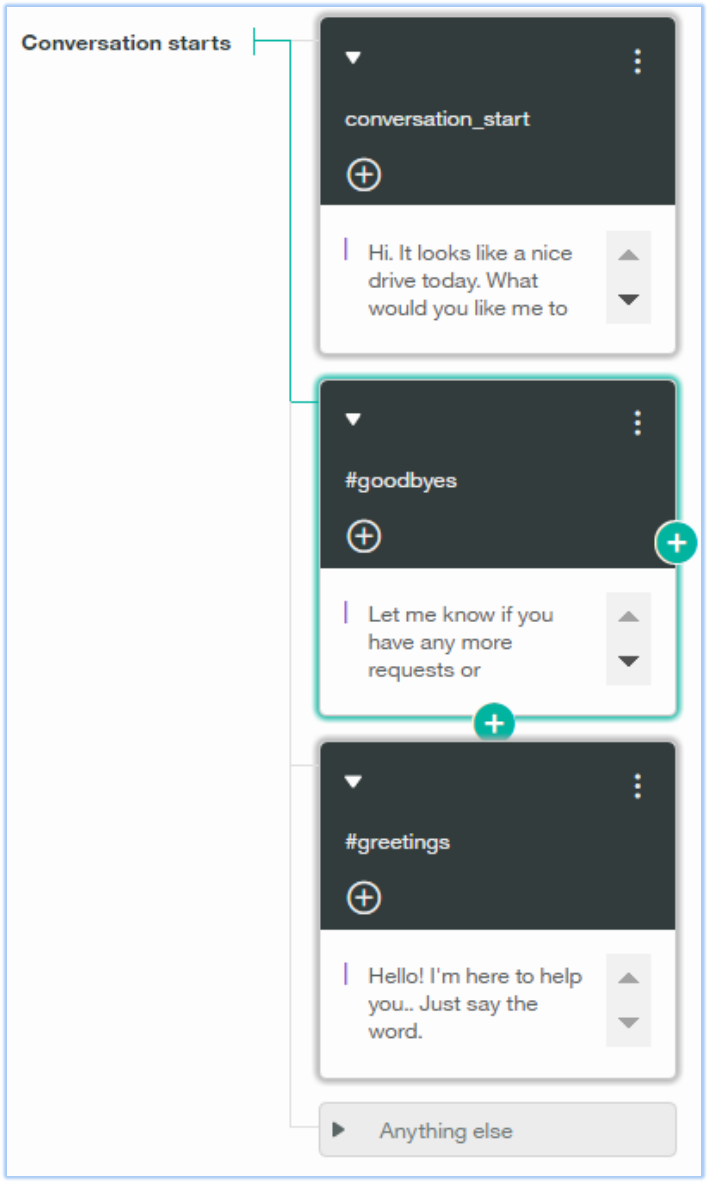
43. In the Watson says: field, enter **`Hello! I'm here to help you.. just say the word.`**

44. Create a new dialog node as a new sibling of the **#greetings** node and enter:

    Condition: **`#goodbyes`**

    Watson says: **`Sure! Let me know if you have more requests or questions.`**

    Now your dialog nodes should look like this:



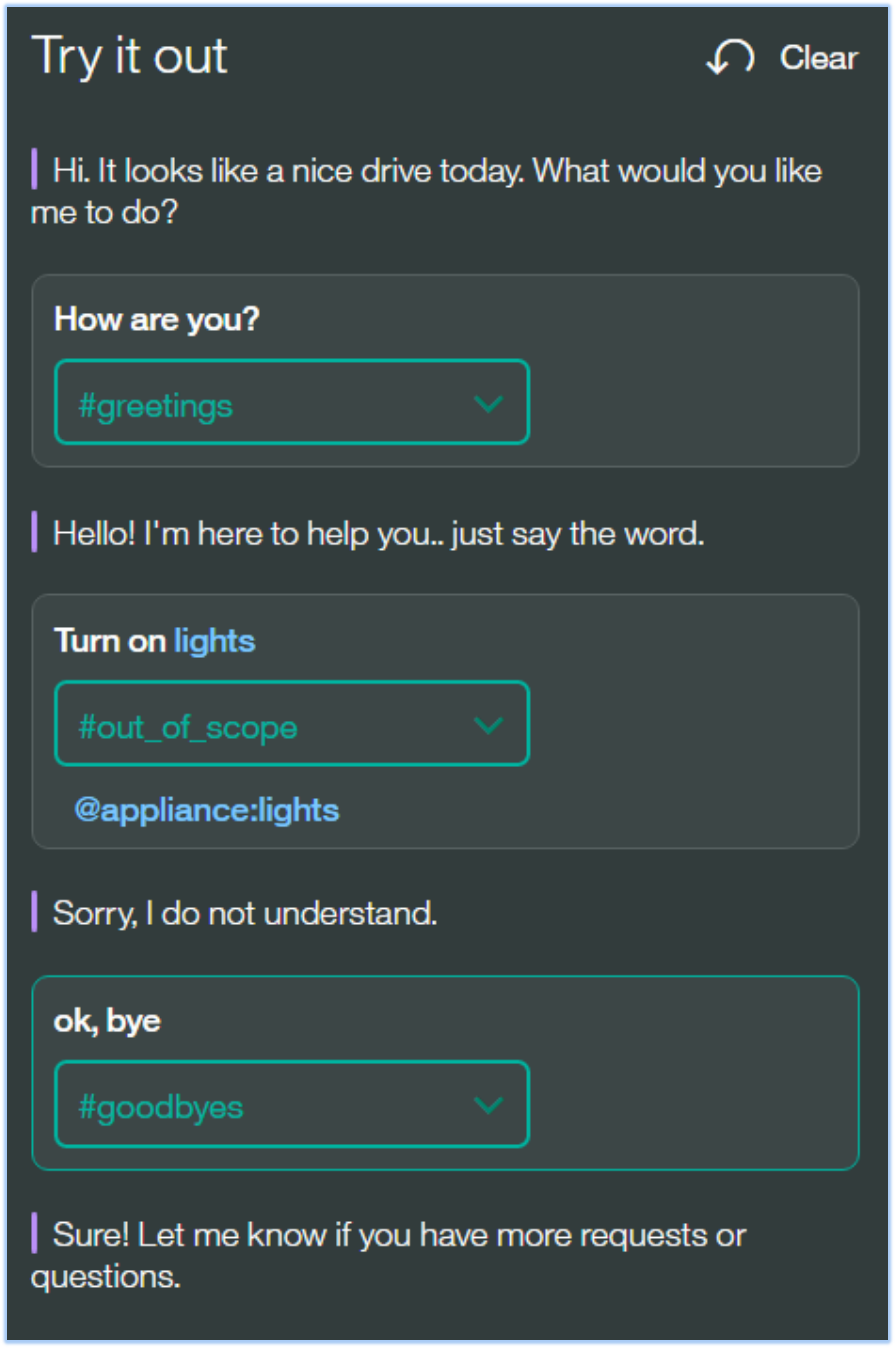
Test your first simple dialog by clicking the **Try it out** icon in the upper right corner.

45. Click  in the top right of the Conversation Tool to open **Try it out** window.

    Recognize the greeting that Watson already displays to start the conversation.

46. Enter a few statements that were defined in the intents such as:

```
How are you?
Turn on lights
Ok, bye
```

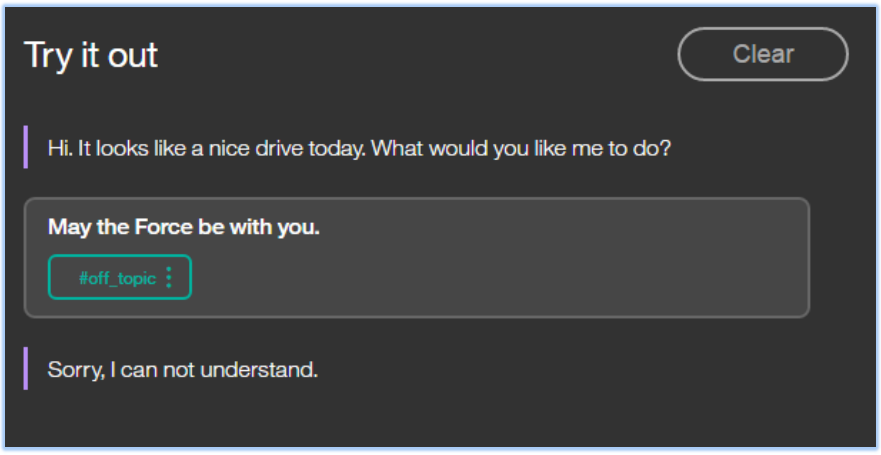


You can see that the Intents are working pretty well already. However, it did not know how to respond to the 'turn on lights' command yet.

1.9 Monitoring conversation results

When you enter an utterance for which the system does not recognize the Intent, it will return the statement that you defined under **Anything** else.

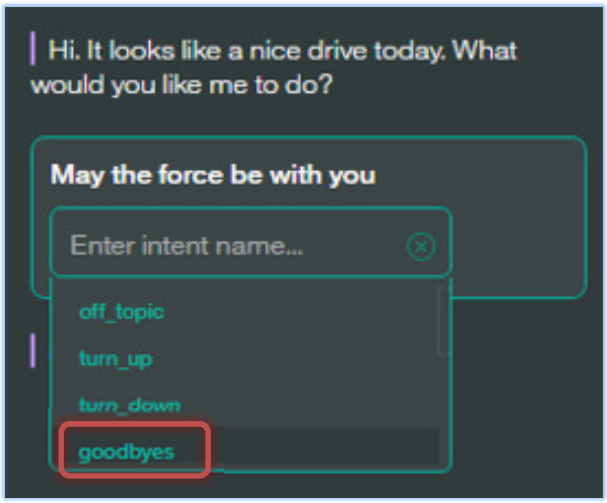
47. Type in **`May the force be with you.`** , which you know the system does not understand.



48. The system identifies the intent as ***#off_topic***.

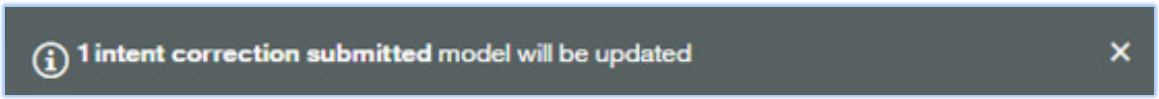
However, we tend to recognize this statement as a possible goodbye greeting. So why not add it to that intent.

49. To do so, click the field of the detected intent (#off_topic) and select or type the intent name you want to assign to classify the utterance.

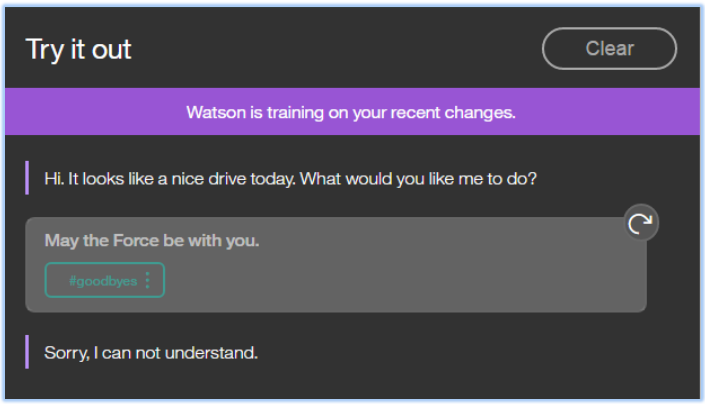


50. Select **goodbyes**

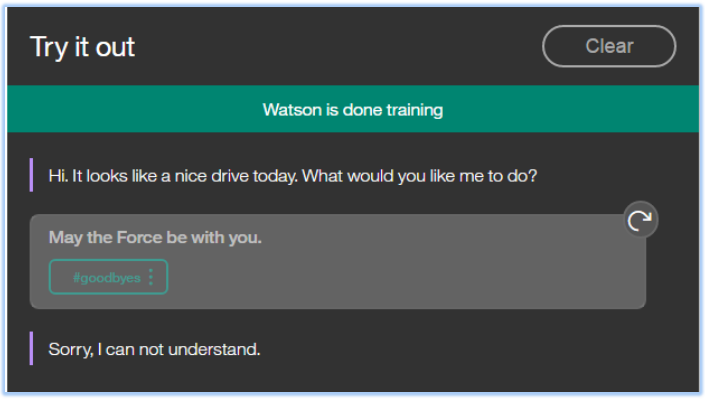
By doing this, you just added added a new example to the **#goodbyes** intent and a message is displayed accordingly.



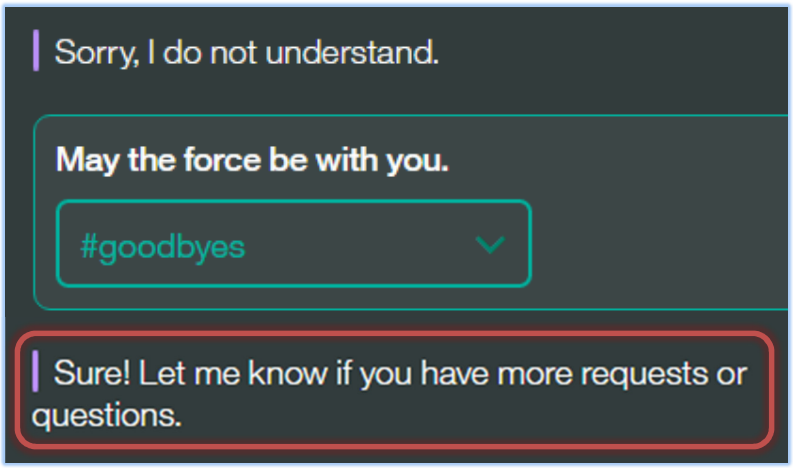
You also see that the **Try it out** window now displays a message that **Watson is training on your recent changes.**



Once the system is updated (trained), the message is replaced by a green message indicating that Watson is done training.



51. Re-enter **`May the force be with you.`** in the input field and confirm the text is correctly recognized as expected intent.



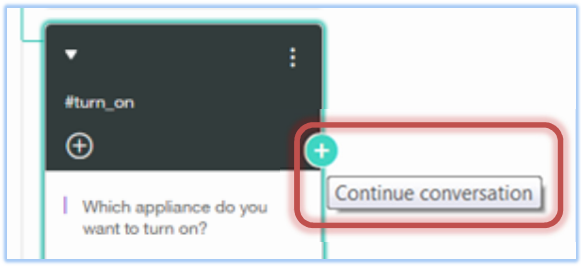
1.10 Creating hierarchical notes to match multiple conditions

52. Select [conversation_start] and create a sibling node ("Create alternative conversation")

53. Enter the following details:

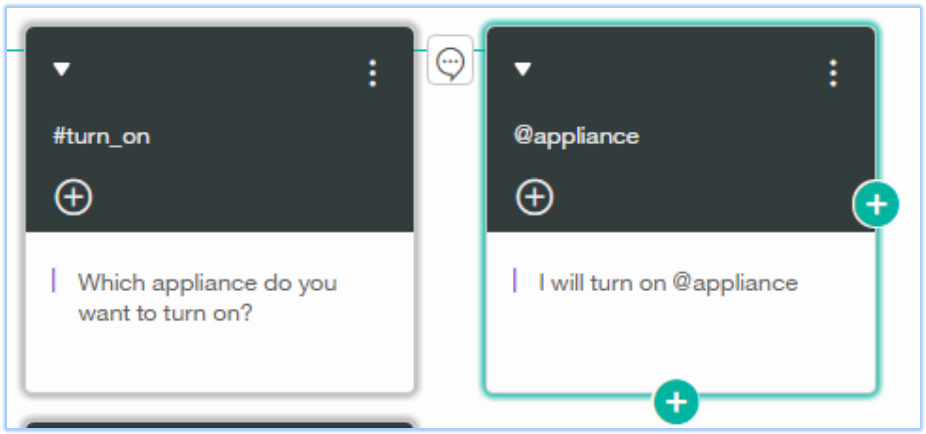
    `Condition:` **`#turn_on`** → select the intent when prompted
    `Watson says:` **`Which appliance do you want to turn on?`**

54. In the [#turn_on] dialog node you have just created, click the ⊕ on the right (Continue conversation) to create child dialog node



55. Specify the following setting in the new node:

    `Condition:` **`@appliance`**
    `Watson says:` **`I will turn on @appliance.`**

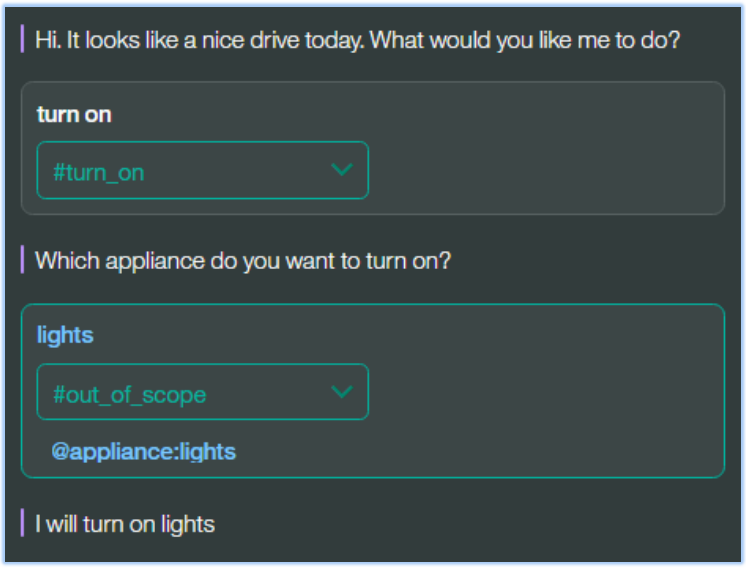


If you have created a child as part of a parent node they follow the sequence of the **user input**. The Conversation system processes the parent node first and returns the response message. It then waits for additional user input to continue the execution of the conversation flow into the child node.

56. Test your implementation in the **Try it out** window and enter the following utterances:

    `You>` **`turn on`**
    `Watson> which appliance do you want to turn on?`
    `You>` **`lights`**

```
Watson> I will turn on lights
```



Now you add a target option for the **#turn_on** intent by adding a sibling node with another matching rule and adding a fall-back node when an invalid appliance name has been specified.

57. Select the **@appliance** dialog node and click (Create alternative conversation) at the bottom.

58. Add a sibling node and specify the following settings:

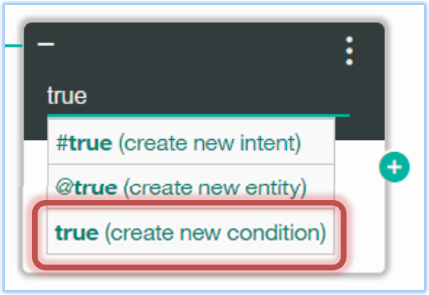
    ```
    Condition: @genre
    Watson says: Great choice! Playing some @genre music for you.
    ```

    This allows the dialog flow to address a new appliance for **#turn_on**, such as "Turn on rock."

59. Select the [@genre] dialog node you just created and click to add another sibling node.

60. Enter:

    ```
    Condition: true
    ```



61. When prompted, select **true (create new condition)**

Note: When you type a new condition, you are presented with several options, such as "#true (create new intent)", "@true (create new entity)", or "true (create new condition).

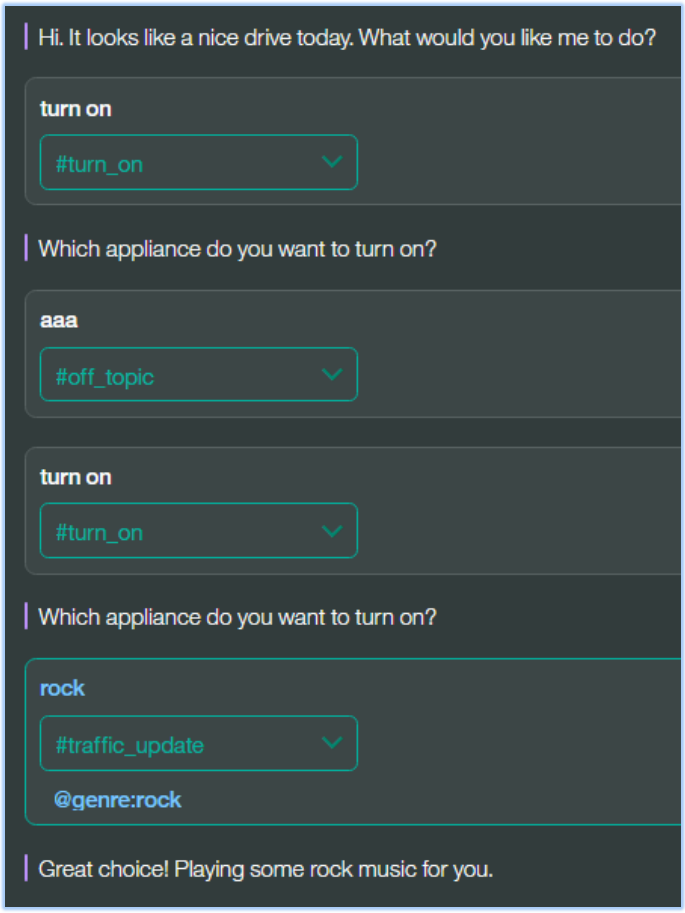
Selecting "intent or @entity would result in the creation of a new intent or entity.
If you create these accidentally, you can remove them in **Intents** or **Entities** tab.

62. Now enter:

```
Watson says: I understand you want me to turn on something. You can
say turn on the wipers or switch on the lights.
```

63. Test your implementation by entering the following statements:

```
You> turn on
You> aaa
You> turn on
You> rock
```



1.11 Using **continue from** statements

Continue from statements tell the system which dialog node to go to next. Moreover, you can specify which portion of the target dialog node the conversation flow will continue from. You can use the **Continue from** function to accomplish many goals, such as:

- Altering the default flow
- Bypassing the default behavior of asking for user input when a dialog goes to the child nodes
- Routing the flow from multiple locations in the tree to a common dialog node

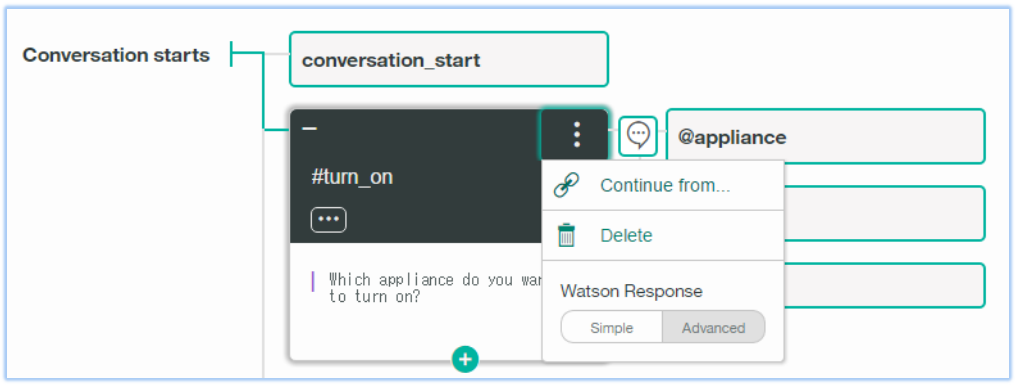
In this section, you will try several types of **continue from** statements.

1.11.1 Continue from - Targeted condition

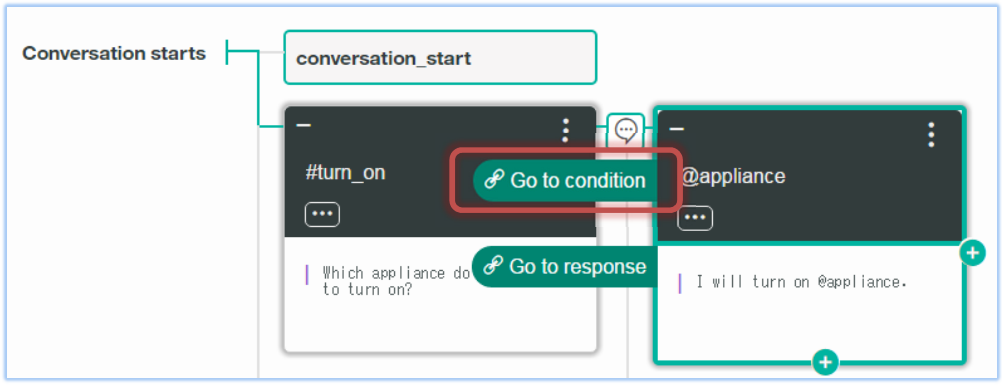
Although you implemented a dialog to accept a *#turn_on* request, the chat flow did not have a lot of flexibility to handle several types of user input. For example, user input like "Turn on lights" contains *#turn_on* intent and *@appliance:lights* in a single text. We therefore need additional sibling nodes for the [#turn_on] dialog node to handle complex conditions.

In this chapter, you will use the **continue from statement with targeted condition** to create a well-structured dialog flow.

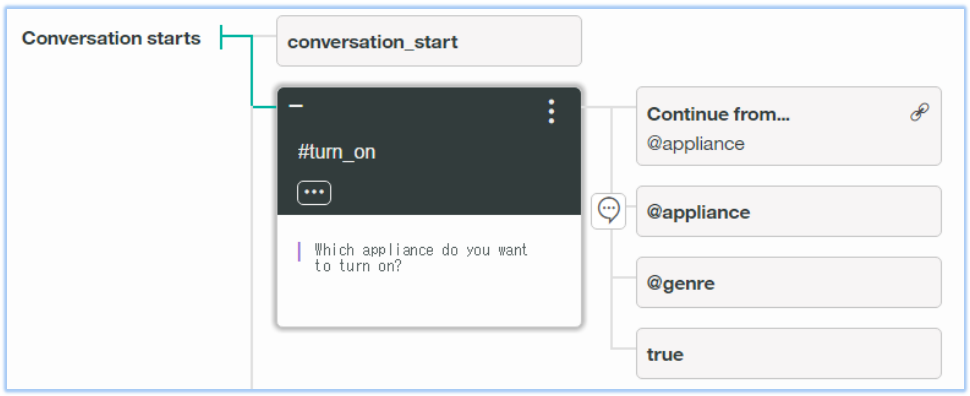
64. Select the [#turn_on] dialog node, click the three dots in the top right corner of the dialog rectangle and select **Continue from**...



65. After selecting **Continue from**, click the [@appliance] dialog node and select **Go to condition**.



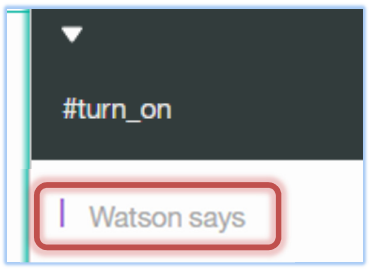
The dialog node structure should now look like this (you can collapse individual notes if they are expanded):



This change results in the input text being evaluated at the [#turn_on] dialog node. The condition evaluation then moves to the [@appliance] node without additional user input.

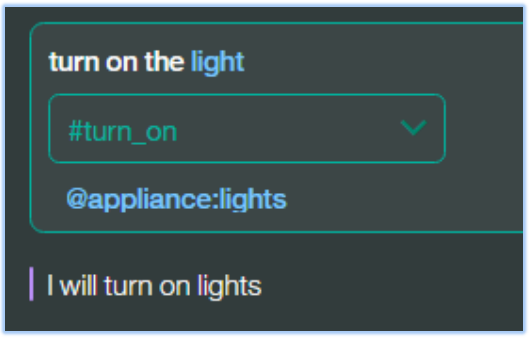
If the user entered a statement that only represents the *#turn_on* intent, it will trigger the [true] node.

66. Before testing your implementation, remove the output text from the **Watson says** field of [#turn_on] dialog node. Otherwise, this message is also returned to the user.



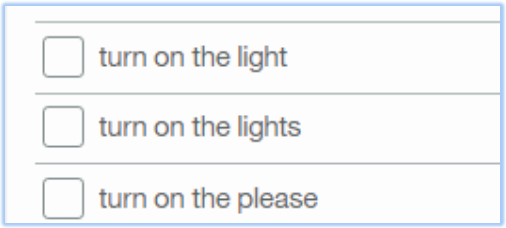
67. Test your implementation

```
You> turn on the light
```



Side-note:

If you entered **turn on the lights** (plural), the system may come back with intent being **out of scope**. Can you determine why? (hint … read through the intents that are listed under the out of scope intent.
While this is not the behavior you would expect, this points to an insufficiently trained system. More examples may need to be added to the #turn_on intent.
To get around this, you can add **"turn on the lights"** user example directly to the intent.



After re-training, the system should now understand the request.

Back to the flow…

If the condition in the dialog node, which is pointed to by a **continue from** statement, does not match the user input, the system moves to a sibling node and continues the evaluation.
As an example, you will add alternative nodes to accept a variety of *#turn_on* requests.

68. Select the [@genre] dialog node, create a sibling and enter:

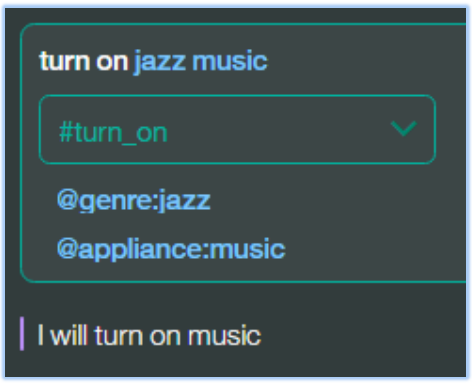
```
Condition: @genre_bad
Watson says: Unfortunately I don't have any @genre_bad music in my
collection. Try genres like jazz or rock.
```

69. Test your implementation.

```
You> turn on radio
Watson> I will turn on music.
You> turn on rock
Watson> Great choice! Playing some rock music for you.
You> turn on hip hop
Watson> Unfortunately I don't have any Hip Hop music in my
collection. Try genres like jazz or rock.
```

```
You> turn on abc
Watson> I understand you want me to turn on something. You can say
turn on the wipers or switch on the lights.
```

70. Now type: **`turn on jazz music`**

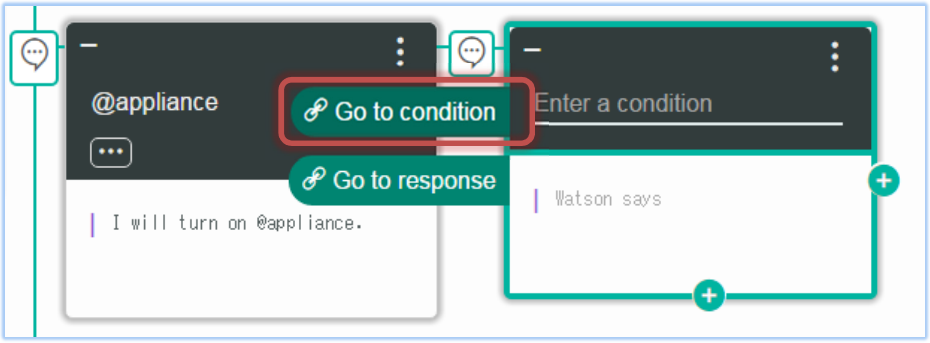


The system correctly identifies intent and entities but has no way of responding yet.

Now add a capability for the system to detect the genre of music in a request like "turn on jazz music."

71. Select the [@appliance] dialog node, click three dots at right top of the dialog rectangle and select **Continue from**...

72. Create a child node for the [@appliance] node and select **Go to condition**



73. In the new child node, enter:

```
Condition: @genre
Watson says: Great choice! Playing some @genre music for you.
```

74. Create a sibling node

75. Enter:

```
Condition: @genre_bad
Watson says: Unfortunately I don't have any @genre_bad music in my
collection. Try genres like jazz or rock.
```

76. Create a sibling node for [@genre_bad] and enter:

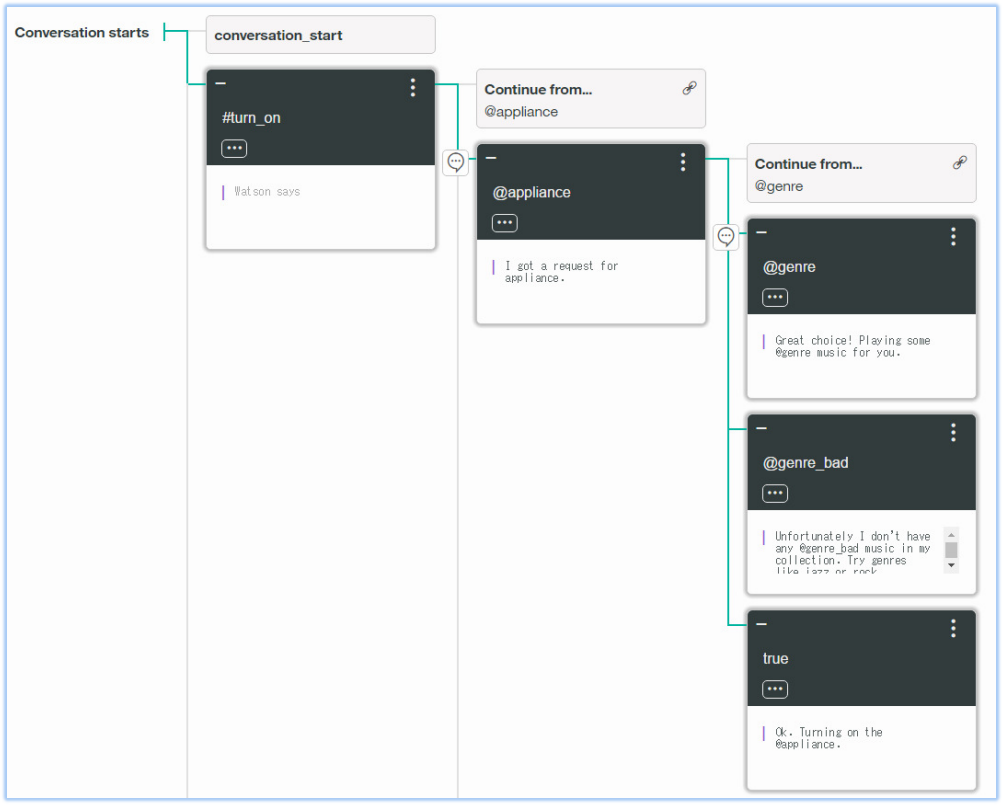
    ```
    Condition: true
    Watson says: Ok. Turning on the @appliance.
    ```

    This dialog node is used when *@appliance* is specified without *@genre* for *#turn_on* intent and displays a simple message like "I will turn on lights."

77. Select the [@appliance] dialog node and replace Watson says field text with:

    ```
    Watson says: I got a request for appliance.
    ```

    Your flow should now look like this:



78. Test your updates:

    ```
    You> turn on rock music.
    Watson> I got a request for appliance.
    Watson> Great choice! Playing some rock music for you.
    You> turn on hip hop music.
    Watson> I got a request for appliance.
    Watson> Unfortunately I don't have any Hip Hop music in my
    collection. Try genres like jazz or rock.
    You> turn on music.
    Watson> I got a request for appliance.
    Watson> Ok, Turning on the music.
    ```

1.11.2 **Continue from** - Targeted user input

As you just saw, the last statement did not ask for the type of genre the user wanted to listen to.

When the users issues a *#turn_on* intent with the *@appliance:music* entity, but without *@genre*, the system should ask the user which genre of music the user wants to listen to.

This is done through reprompting and can be implemented by adding a child node without the **continue from** statement.

However, in the case that the additional input is not recognized as music genre, the system should reprompt continuously. In that case, you can use **Continue from statement with targeting user input.**

At first, you create a simple reprompt for music genre.

79. Select the [@genere_bad] node under the [#turn_on] > [@appliance] node branch and create a sibling node

80. Enter:

    Condition: **@appliance:music**
    Watson says: **Sure thing! Which genre would you prefer? Jazz is my personal favorite..**

81. Select the [@appliance:music] node you just created, and create 2 child nodes.

82. Enter the following names and settings:

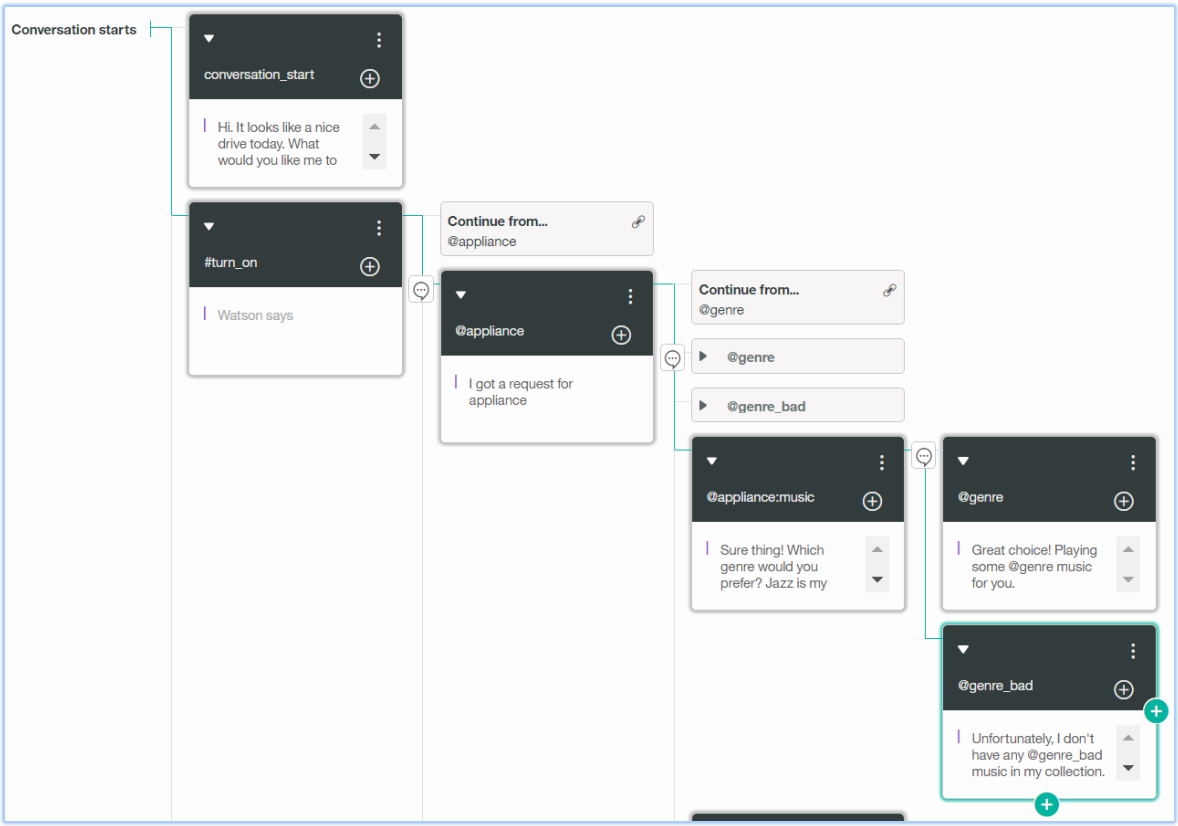
1st node:

Condition: **@genre**
Watson says: **Great choice! Playing some @genre music for you.**

2nd node:

Condition: **@genre_bad**
Watson says: **Unfortunately, I don't have any @genre_bad music in my collection. Try genres like jazz or rock.**

Here is what your node structure should now look like:



83. Test your new structure

```
You> turn on music
Watson> I got a request for appliance.
Watson> Sure thing! Which genre would you prefer? Jazz is my
personal favorite..
You> Rock
Watson> Great choice! Playing some rock music for you.
You> turn on radio
Watton> I got a request for appliance.Watson> Sure thing! Which
genre would you prefer? Jazz is my personal favorite..
You> Hip hop
Unfortunately, I don't have any Hip Hop music in my collection. Try
genres like jazz or rock.
```

Note: An interesting side-note is that by entering **Rock**, the intent came back as #traffic_update. Of course, this does not matter here since you were in the middle of a conversation flow. Try entering **rock** by itself and the system will not know what to do with the input.

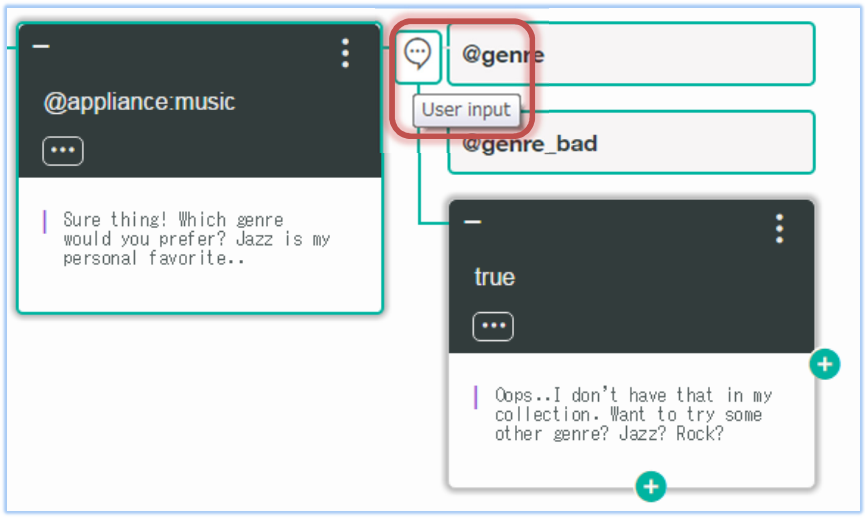
Now you add a third option for the scenario where a valid music genre name is not being provided. In that case, you want to loop the reprompt until the users enters a valid music genre. Use **continue from** with targeting user input to implement this feature.

84. Select the [@genre_bad] node you just created under the [@appliance:music] node and create a sibling node. Then enter:

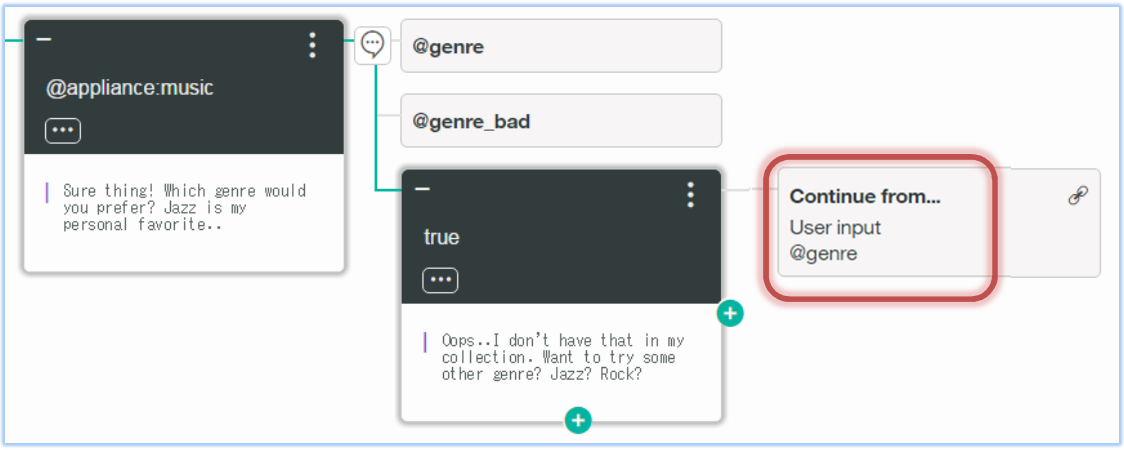
    `Condition:` **`true`**
    `Watson says:` **`Oops..I don't have that in my collection. Want to try some other genre? Jazz? Rock?`**

85. Select the [true] node you just created, click the three dots in the right top of the dialog rectangle and select **Continue from**...

86. Select the speech bubble "User input", connecting the [@appliance:music] node with the [true] node



This creates a new **continue from** statement from the [true] node



87. Test your changes:

```
Type turn on music
Watson> I got a request for appliance.
Watson> Sure thing! Which genre would you prefer? Jazz is my
personal favorite.
Type abc
Watson> Oops..I don't have that in my collection. Want to try some
other genre? Jazz? Rock?
Type Rock
Watson> Great choice! Playing some rock music for you.
```

Now the system repeats the prompt for genre until the user enters a music genre that is included in the *@genre* or *@genre_bad* entities.

Later in this lab exercise, you will add a mechanism for showing this reprompt only once by using context variables.

1.11.3 Continue from - Targeted response

The conversation flow author can separately specify output messages for any dialog node. However, putting the same output message into multiple places complicates workspace maintenance.

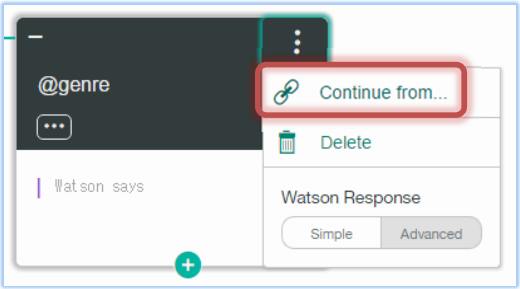
Using **Continue from** for targeting a response, helps avoid this and allows the re-use of output from another node.

88. Create a sibling for the [#turn_on] node and enter:

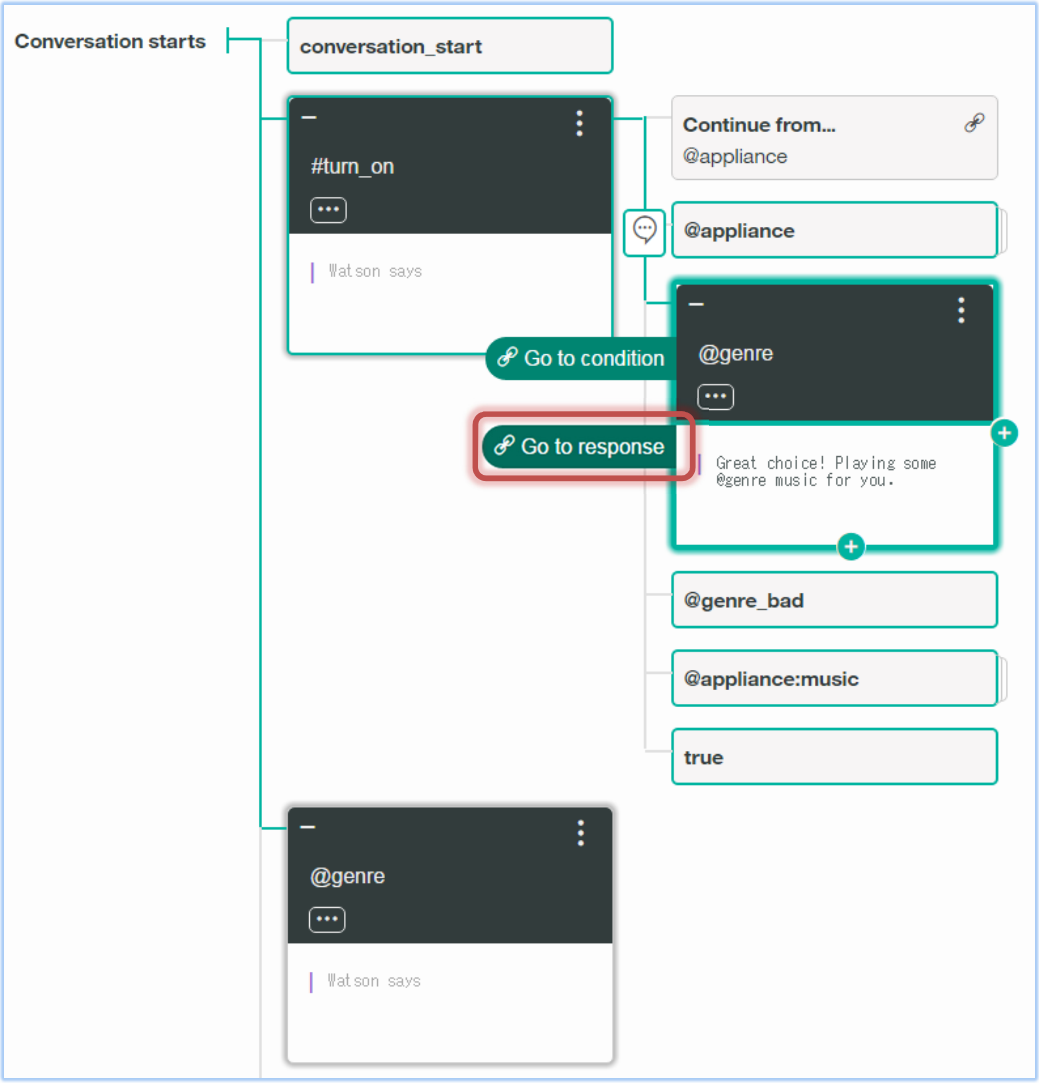
    `Condition:` **`@genre`**

    `Watson says:` (leave blank)

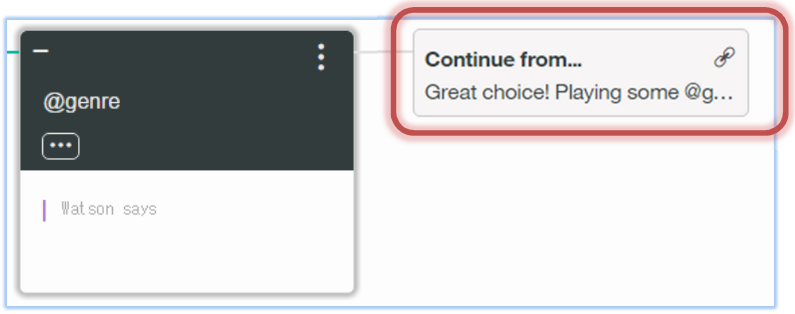
89. In the [@genre] node you just created, click three dots and select **Continue from**...



90. Select the other [@genre] node that is a child of the [#turn_on] node, and choose **Go to response**.



This creates a new **continue from** statement as part of the top level [@genre] node.



91. Test your changes:

```
You> rock please
Watson> Great choice! Playing some rock music for you.
```

1.12 Using context variables

When authoring advanced conversation flows, you may need to store status information about the current conversation, which can be used as context. You can store information by editing the context part of the dialog node definition in an advanced editor.

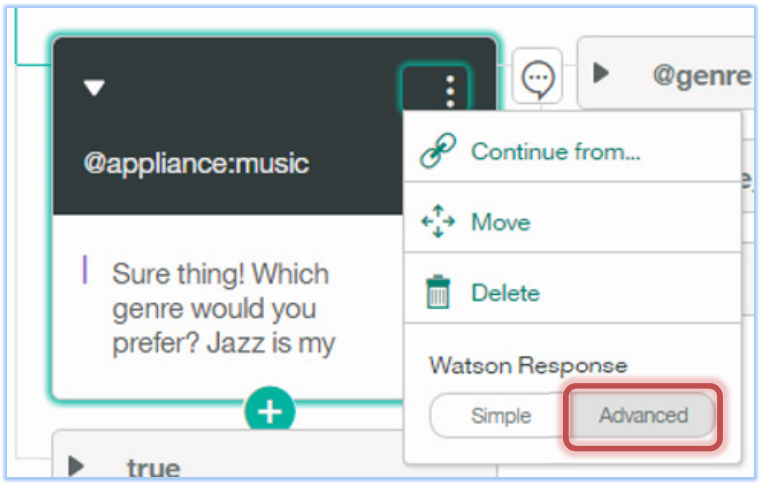
In this section, you will store the context related information and enhance the conversation behavior using context variables.

1.12.1 Boolean context variables

Start by adding a mechanism for reprompting similarly to the "turn on music" request. You want the system to reprompt when the user input contains a *#turn_on* intent and *@appliance:music* entity. The system reprompts until the user enters a valid music genre. At that point, add a Boolean context variable as a flag to prevent further reprompts.

92. Select the [@appliance:music] node under the [#turn_on] > [@appliance] nodes hierarchy

93. Click the three dots in [@appliance:music] and select **Advanced** of **Watson Response**



The advanced editor allows you to add your context variable definitions in JSON object format.

94. In the advanced editor add the following object definition (in bolt **red** text). Make sure you include the comma:
(To simplify this exercise, copy and paste the cod from this document into your tool)

```
{
  "output": {
    "text": "Sure thing! Which genre would you prefer? Jazz is my
personal favorite.."
  },
  "context": {
    "reprompt": true
  }
}
```

95. Select the [true] child node under the [@appliance:music] node you just modified and change the condition to:

    `Condition: $reprompt`

96. Select **$reprompt (create new condition)** when prompted.

    Note: $ refers to a context variable. When the reprompt context variable is true in the parent node, this condition will evaluate as true as well.

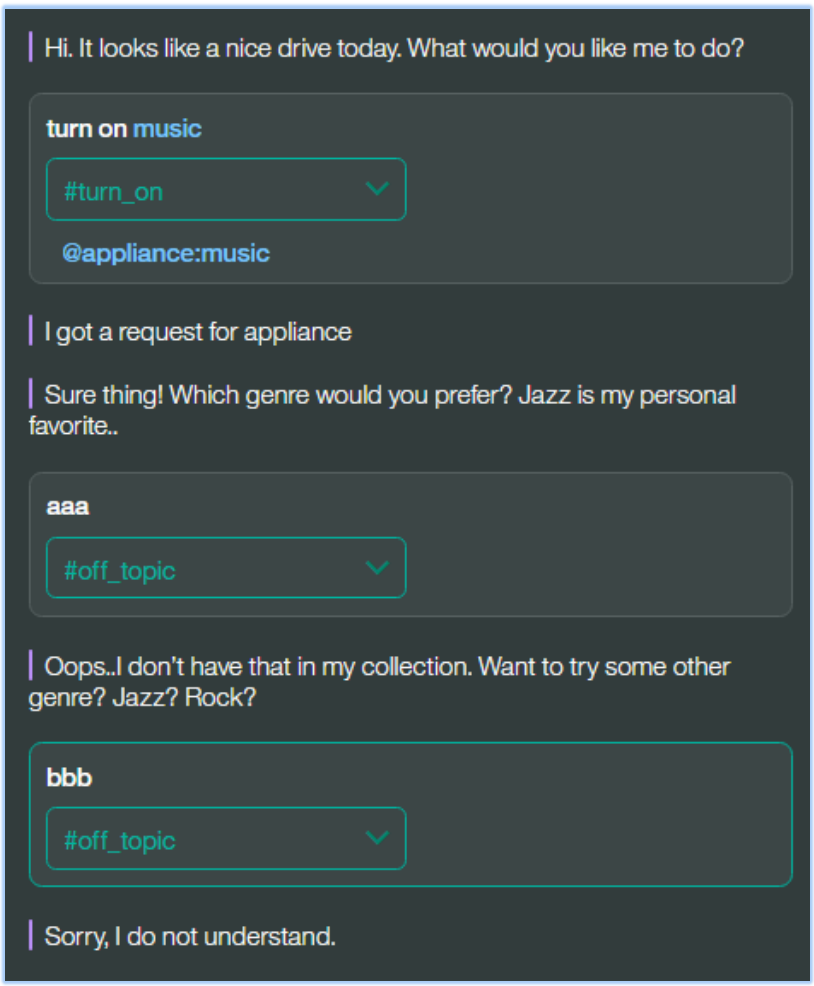
97. Click the three dots in the [$reprompt] node and select **Watson Response** > **Advanced**

98. In the advanced editor, add the following object definition:

```
{
  "output": {
    "text": "Oops..I don't have that in my collection. Want to try
some other genre? Jazz? Rock?"
  },
  "context": {
    "reprompt": false
  }
}
```

    Your dialog flow should now look like this:

99. Test your implementation with **Try it** out window.



1.13 Numeric context variable – Response Variations

You will now implement logic into the [Anything else] node to provide response variations.

When a conversation arrives at the [Anything else] node multiple times, you can assume that the user is confused about the preceding conversation.

In that case, it will help to change the wording of the response, or delegate the conversation to a human operator.

This can be done by adding a counter for arriving at the [Anything else] node.

This counter is a numeric context variable used to change the response based on different numeric values.

100. Select the [conversation_start] node and select **Watson Response** > **Advanced**

101. In the editor add the following object definition:

```
{
  "output": {
    "text": "Hi. It looks like a nice drive today. What would you
like me to do?"
  },
  "context": {
    "defaultCounter": 0
  }
}
```

102. Access the **Advanced** editor at the top level [Anything else] node.

103. Replace the entire text with this:

```
{
  "output": {
    "text": ""
  },
  "context": {
    "defaultCounter": "<?$defaultCounter+1?>"
  }
}
```

This code provides an empty response text to output and increment value of the **defaultCounter** variable. When the [Anything_else] node is evaluated initially, the **defaultCounter** is 0, as defined in the [conversation_start] node. Each time the conversation system reaches the [Anything else] node, the value is incremented.

You will now create two child nodes to provide a response for the first two times the conversation reaches the node, and another response for the third time the node is reached.

104. Select **Continue from** for the [Anything else] node.

105. Create a child node for the [Anything else] node select **Go to condition**.

106. Then enter the following condition:

```
Condition: $defaultCounter>2
```

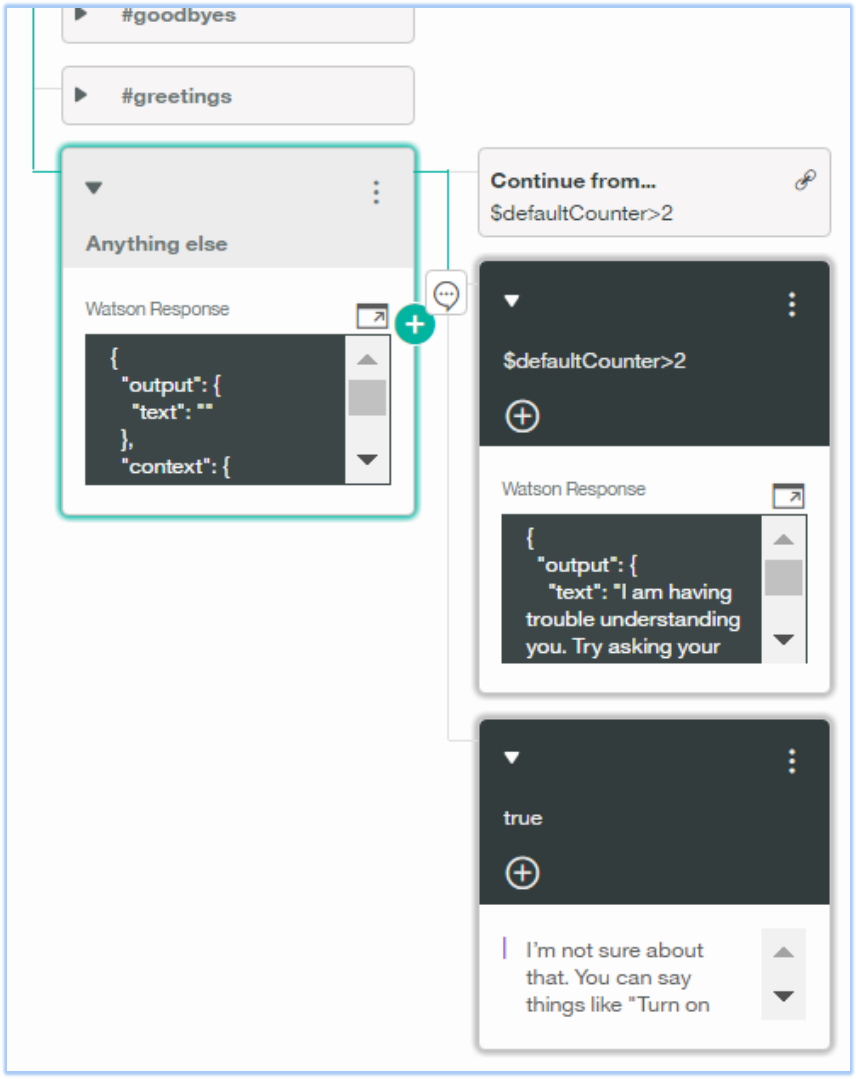
107. In the advanced editor, enter:

```
{
  "output": {
    "text": "I am having trouble understanding you. Try asking your
question in a different way."
  },
  "context": {
    "defaultCounter": 0
  }
}
```

108. Create a sibeling node for the [$defaultCounter>2] node and enter:

```
Condition: true
Watson says: I'm not sure about that. You can say things like "Turn
on my lights" or "Play some music."
```

Your flow should now look like this:



109. Test your implementation.

```
You> aaa
Watson> I'm not sure about that. You can say things like "Turn on my
lights" or "Play some music."
You> bbb
Watson> I'm not sure about that. You can say things like "Turn on my
lights" or "Play some music."
You> ccc
Watson> I am having trouble understanding you. Try asking your
question in a different way.
You> ddd
Watson> I'm not sure about that. You can say things like "Turn on my
lights" or "Play some music."
```

Recognize that the answer was different the third time (ccc) Watson did not understand.

1.14 Complex condition statement

In this section, you will define a complex expression in a dialog condition.

1.14.1 Use of logical expression

Your car dashboard application may handle some appliances that have the capability to adjust volume, or level, such as the air conditioner and the radio. However, some appliances do not have that capability so you will implement a conversation flow for receiving *#turn_up* or *#turn_down* intents and provide the appropriate response depending on the target appliance capability.

110. Create a new sibling for the **#turn_on** node and enter:

```
Condition: #turn_up
Watson says: (leave blank)
```

111. Create two child nodes under the [#turn_up] node and enter:

1st node:

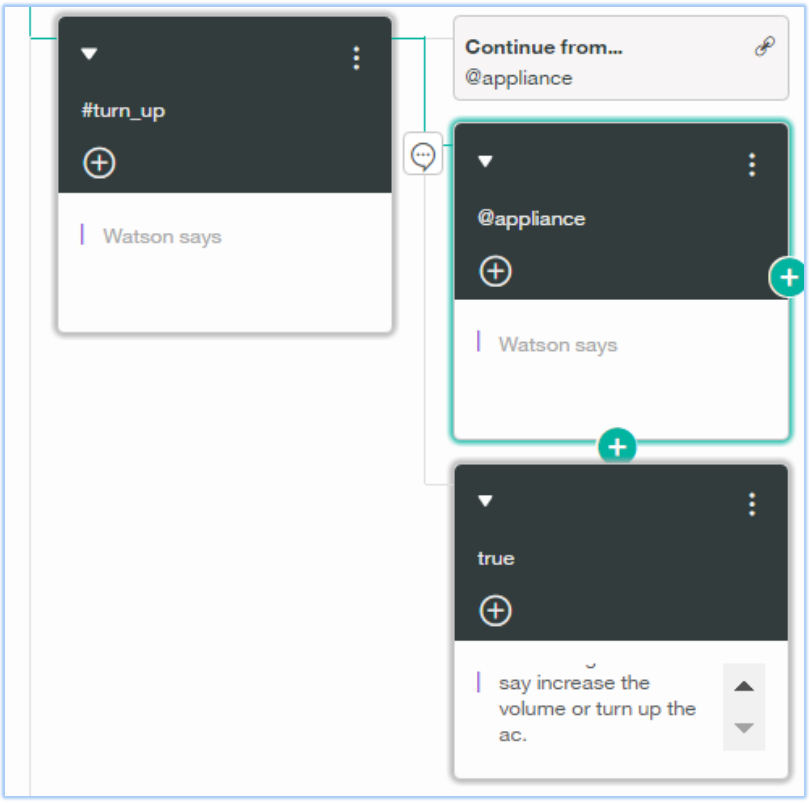
```
Condition: @appliance
Watson says: (leave blank)
```

2nd node:

```
Condition: true
Watson says: I understand you want me to turn up something. You can
say increase the volume or turn up the ac.
```

112. Add **Continue from** to the [#turn_up] node and connect to the condition of the [@appliance] node you just created.

Now flow should look like this:



The assumption is that the target appliance for *#turn_up* and *#turn_down* intents are the air conditioner, heater, music, volume, or fan of the *@appliance* entities.

Now add a dialog nodes to check if a valid appliance is specified along with *#turn_up*.

113. Create a new child node for [@appliance].

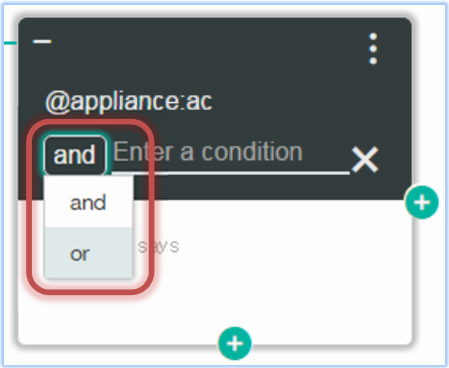
You want to specify multiple conditions of the following table following the procedure outlined:

	@appliance:ac
or	@appliance:heater
or	@appliance:music
or	@appliance:volume
or	@appliance:fan

114. Enter *@appliance:ac* in the condition field.

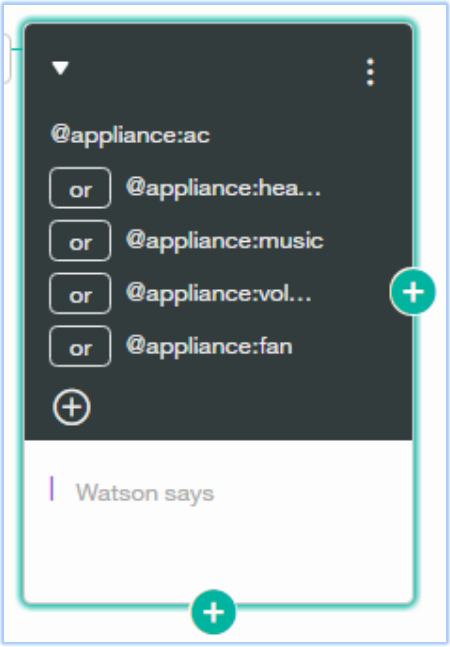
115. Click the + sign displayed in the Condition field to show **and**

116. Click the **and** and change it to **or**



117. Enter **@appliance:heater** in the second condition field.

118. Add additional applieances until your node looks like this:



119. On the same dialog node, enter:

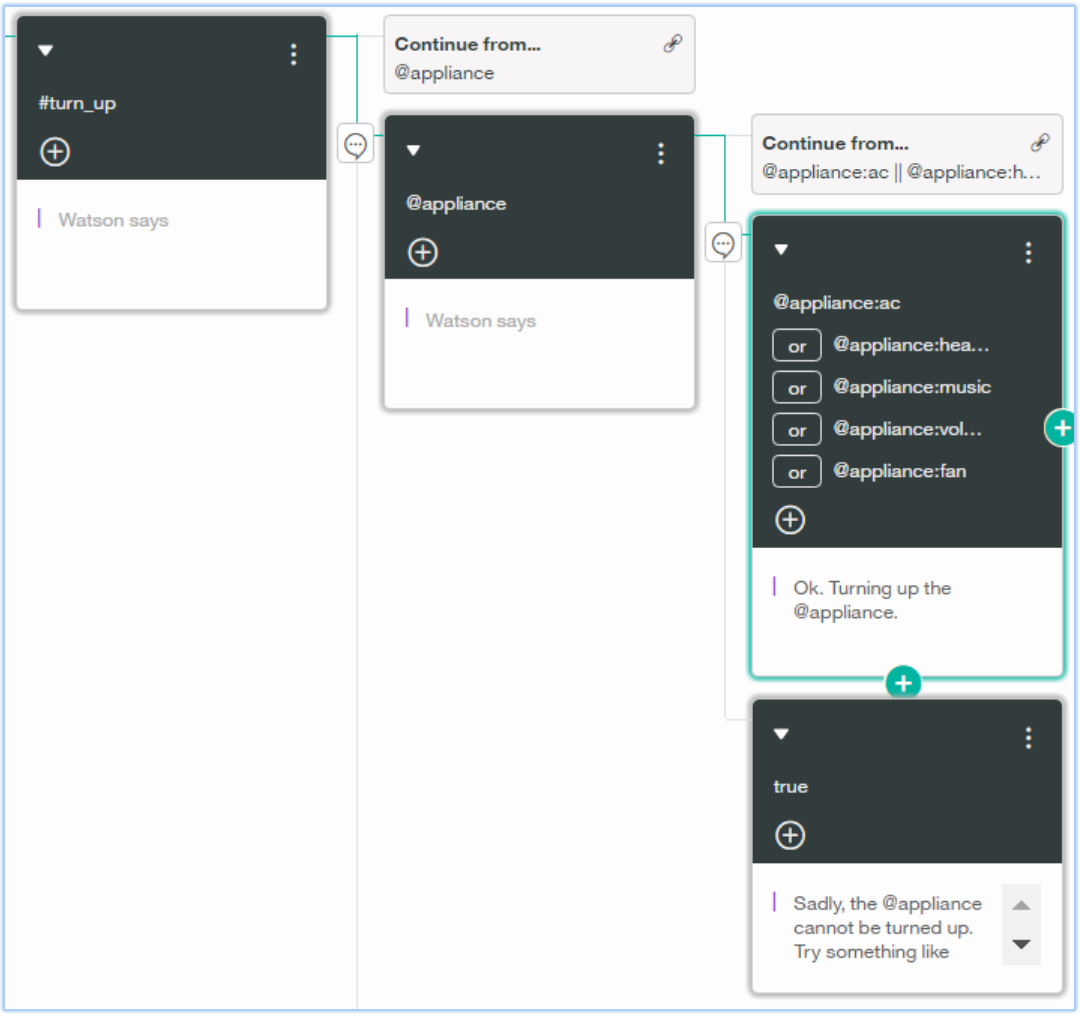
```
Watson says: Ok. Turning up the @appliance.
```

120. Create a sibling node and enter:

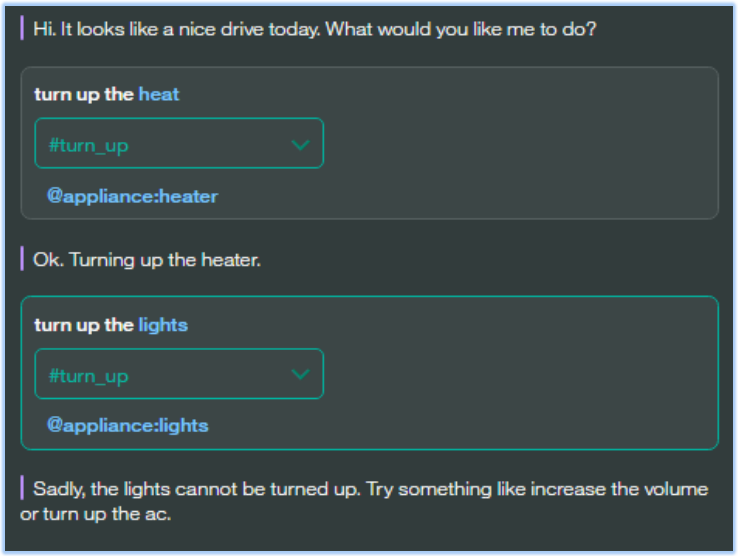
```
Condition: true
Watson says: Sadly, the @appliance cannot be turned up. Try
something like increase the volume or turn up the ac.
```

121. **Continue from** the [@appliance] node to the condition of the [@appliance:ac OR ...] node.

Now your dialog nodes should look like this:



122. Test your changes



123. Optionally implement a similar flow for **#turn_down** intent.

1.15 Using global variables

Conversation service provides an advanced method for handling contextual data through global variables and generating output messages and conditions. This section illustrates how to access global variable and manipulate their value.

You will implement a conversation flow to answer when user asked something about rain. This use case will later be implemented in subsequent labs as an expansion to the car dashboard.

124. Create a new top level node just after the [@genre] node and enter:

```
Condition: #weather
Watson says: (leave blank)
```

125. Created a **Continue from** on the node

126. Create a new child node under the **#weather** node and connect to **Go to condition**

127. Specify the following setting on the dialog node you just created:

```
Condition: input_text.contains('rain')
Watson says: This is a short shower, it should only last a minute.
```

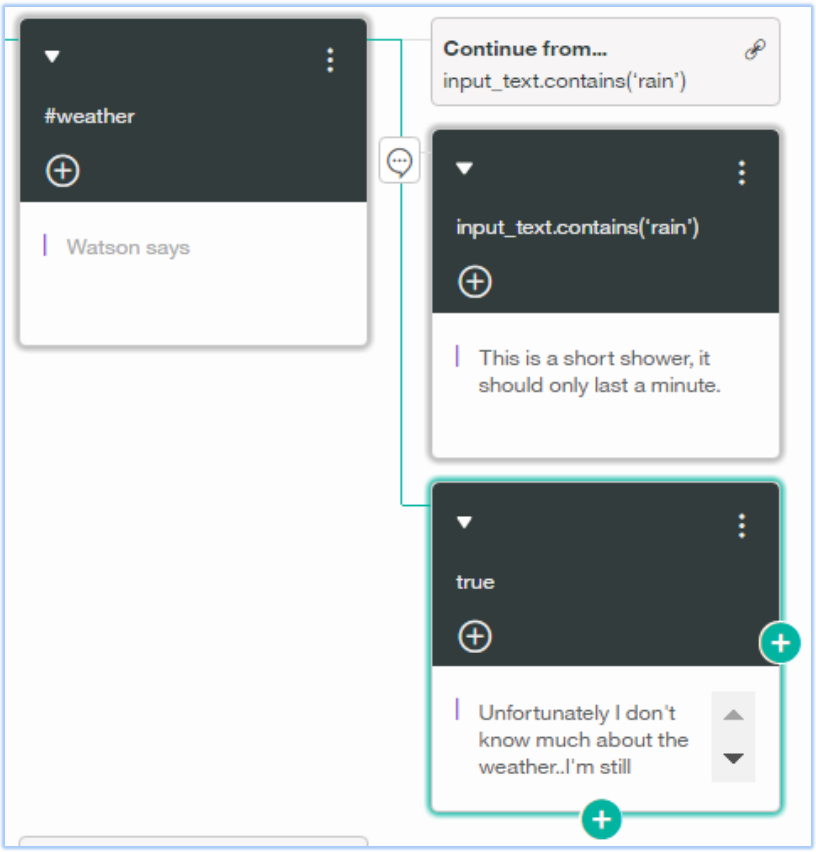
You can use global variables to access contextual data. In this example, the **input_text** variable contains the text that the user has entered in this node condition. The user input text is also accessible through the **text** object value of the **input** variable like **input.text**.

If the user says something about the weather and mentions the word “rain", this node will trigger the response entered in the **Watson says** field. You will also add another dialog in case the user did not talk about "rain".

128. Create a sibling node for the [input_text.contains('rain')] node and enter:

```
Condition: true
Watson says: Unfortunately I don't know much about the weather..I'm
still learning.
```

Your flow should look like this:



129. Test your changes

```
You> Will it rain?
Watson> This is a short shower, it should only last a minute.
You> How about weather?
Watson> Unfortunately I don't know much about the weather..I'm still
learning.
```

1.16 Check classified result and take over a transaction to the caller app

You will add a dialog flow to handle out-of-scope request from the user. The dashboard application is designed to accept car driver requests and control equipment on the vehicle. A request, categorized as out-of-scope, cannot be handled by this car dashboard system and should be processed by a separate system outside of the Conversation service and in the application layer.

In the conversation flow you are implementing, you will verify the confidence of intent classification, meaning whether a request is classified into *#out_of_scope* with a confidence value higher than a specific threshold. If so, the system will respond with a signal that it should be handled by another system.

130. Create new top level node just after [#goodbyes] and enter:

```
Condition: #out_of_scope
Watson says: (leave blank)
```

131. Select the [#out_of_scope] node and create a new **Continue from...**

132. Create a child node and select **Go to condition**

133. Enter a new condition:

```
Condition: intents[0].confidence>0.5
```

134. In the advanced editor, specify:

```
{
  "output": {
    "text": "That question is out of scope for this application,
take a look at the Conversation Enhanced application to handle
questions like these."
  },
  "context": {
    "callRetrieveAndRank": true
  }
}
```

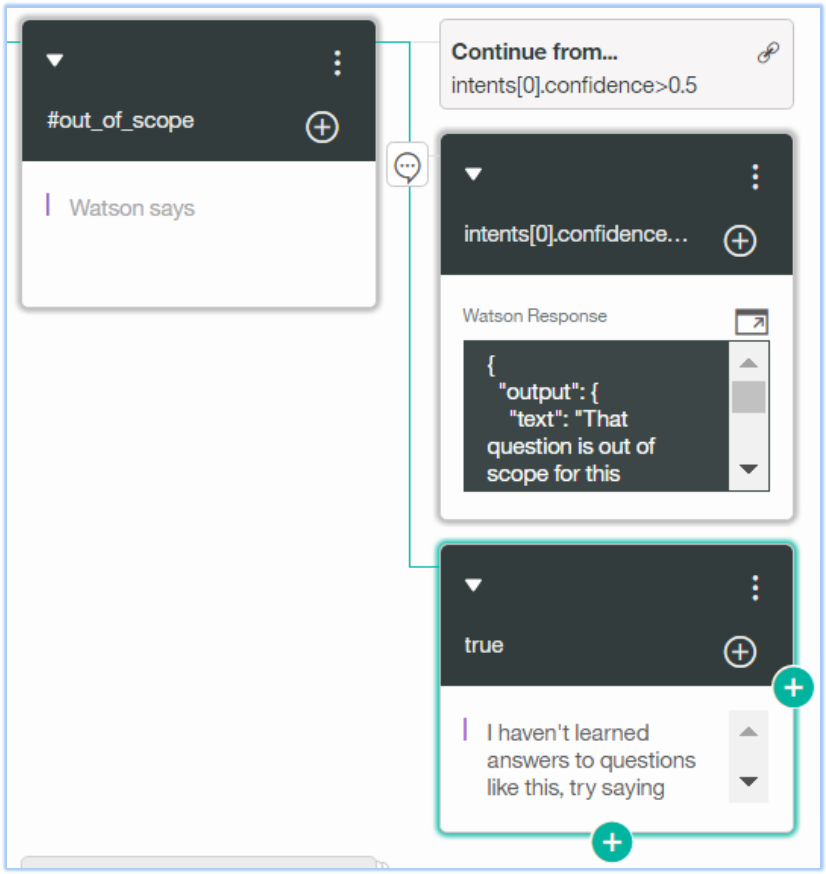
The classified intents can be access through global variable *intents* as array of intents. The first item of the array contains the top classified intent result and you can access the confidence value with *intents[0].confidence>0.5*. In this condition, the dialog system responds with a message and custom flag of a **callRetrieveAndRank** context variable when the input text is classified as *#out_of_scope* with 50%, or higher confidence value.

Add an alternative node for when the input text does not match the first node.

135. Create a sibling node for the [input_text.contains('rain')] node and enter:

```
Condition: true
Watson says: I haven't learned answers to questions like this, try
saying things like "Turn on my lights" or "Play some music."
```

Your dialog flow should now look like this:



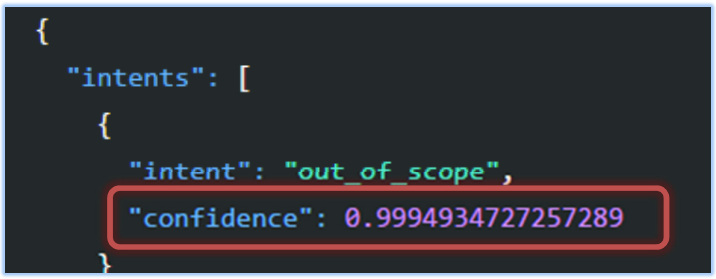
136. Test your changes

```
You> How do I adjust the mirror?
Watson> That question is out of scope for this application, take a
look at the Conversation Enhanced application to handle questions
like these
```

The Conversation Tool does not have a capability to show detailed response data from Conversation /message API call. You can see the detail request and response data on the Conversation simple app demo.

137. Open Conversation simple app at the following URL:
http://conversation-simple.mybluemix.net/

138. Type in a #out_of_scope request and see the request and response data.



The response data is show under **Watson understands** and it contains **callRetrieveAndRank** context variable.

Note: The source code for the Conversation simple app is available in the following GitHub repository. You can deploy the code on Bluemix and try it with your Conversation workspace.

https://github.com/watson-developer-cloud/conversation-simple

1.17 Complex conversation flow

In this section, you will create a more complex dialog flows to provide an improved experience for the user.

139. Create a new top level node just after the [@genre] node and specify the following setting:

```
Condition: #local_amenity
Watson says: (leave blank)
```

140. Select **Continue from**

141. Create a new child node and select **Go to condition**

142. Enter:

```
Condition: @amenity
Watson says: (leave blank)
```

If type of amenity (*@amenity*) is specified at *#local_amenity* intent, we will signal to the application to search and show the preferred amenity near the current location and ask the user to select an option number from the list.

In this lab exercise, we assume that the Car Dashboard application has capability to show 3 types of amenities as gas station, restaurant and restroom.

143. Select the [@amenity] dialog and create a new **Continue from...**

144. Create new child node and connect with condition

145. In the advanced editor, enter following code:

```
Condition: @amenity:gas
Watson Response:
      {
        "output": {
          "text": "There are a few gas stations nearby. Which one
      would you like to drive to?"
        },
        "context": {
          "reprompt": true
        }
      }
```

Here you set a reprompt context variable because you will get user input for the options and want to repeat the prompt until a valid option is being selected.

146. Create three child nodes under the [@amenity:gas] node and enter the following settings:

1st node:

```
Condition: @option
Watson says: Sure! Navigating to the @option gas station.
```

2nd node:

```
Condition: @amenity:restaurant OR @amenity:restroom
Watson says: (leave blank)
```

The 2nd node is created in case the user wants to change the amenity type from the previous request. When the conversation matched this node, it will transfer the conversation flow to continue from to the other dialog tree for handling @amenity:resurant, which is not created yet. You will add the continue from for this dialog node later.

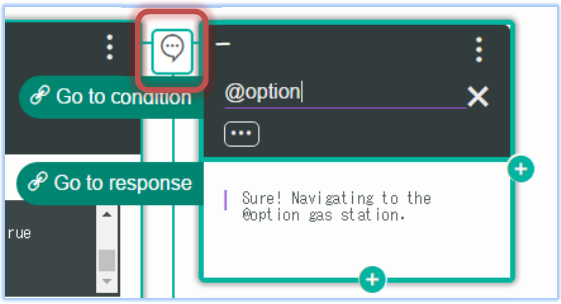
3rd node:

```
Condition: $reprompt
Watson Response:
      {
        "output": {
          "text": "Which option would you like? You can say first,
      third, nearest and so on.."
        },
        "context": {
          "reprompt": false
        }
      }
```

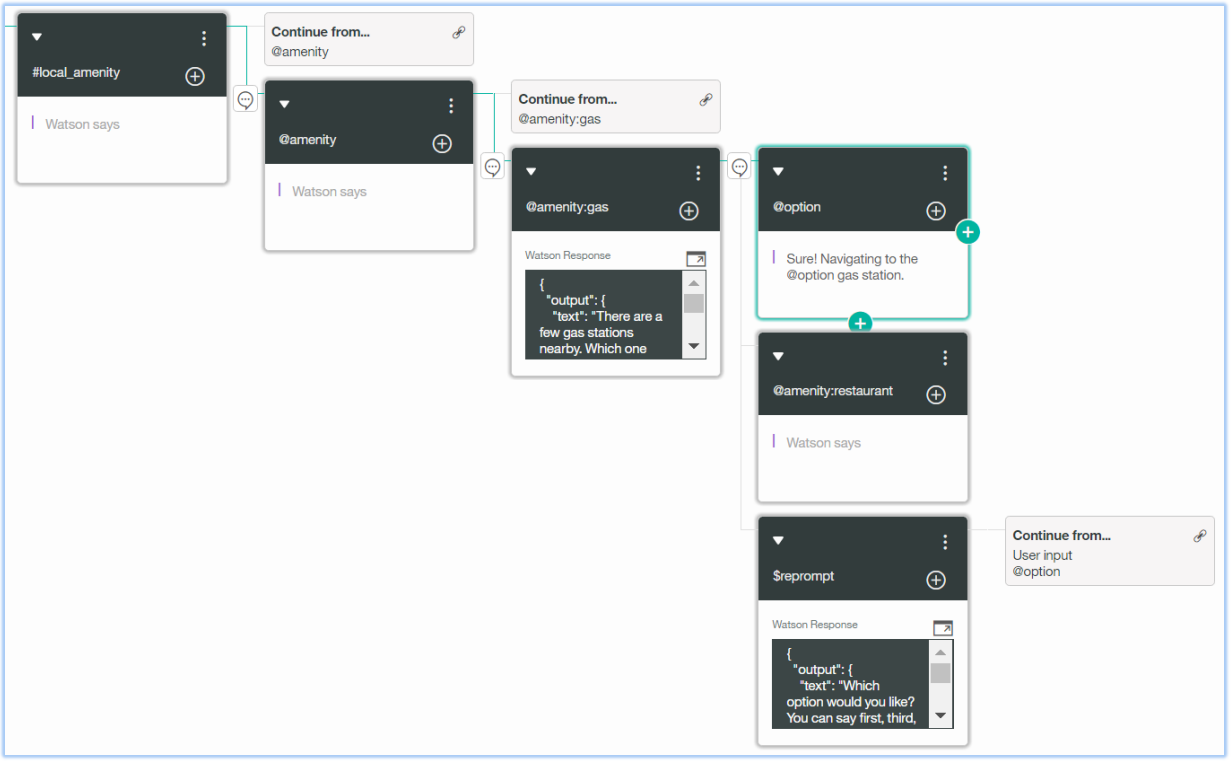
The 3rd node is used to reprompt the question when the user input was not valid. The *reprompt* context variable is set to 'false' because you do not want to loop here. This will transfer the conversation to the previous user input.

147. Select the [$reprompt] node and create new **continue from**

148. Click the [@option] node which is a child of the [@amenity:gas] node, and select **user input**



Now the conversation tree starting from the [#local_amenity] dialog node looks like this:



Add additional conversation tree branches for *@amenity:restaurant* and *@amenity:restroom* as sibling nodes for [@amenity:gas]. The conversation tree starting from [#local_amenity] node will have the following structure:

`[#local_amenity] node`
`Condition:` **`#local_amenity`**
`Watson says: (none)`	

	Continue from condition of [@amenity] node

[@amenity] node
Condition: **@amenity**
Watson says: (none)

Continue from condition of [@amenity:gas] node

[@amenity:gas] node
Condition: **@amenity:gas**
Watson Response:

```
{
  "output": { "text": "There are a few gas stations nearby. Which one would you like to drive to?"},
  "context": {"reprompt": true}
}
```

[@option] node
Condition: **@option**
Watson Response:

```
{
  "output": {"text": "Sure! Navigating to the @option gas station." },
  "context": {"reprompt": false}
}
```

[@amenity:restaurant OR @amenity:restroom] dialog node
Condition: **@amenity:restaurant** OR **@amenity:restroom**
Watson says: (None)

Continue from condition of [@amenity:restaurant] node

[$reprompt] node
Condition: **$reprompt**
Watson Response:

```
{
  "output": {"text": "Which option would you like? You can say first, third, nearest and so on.." },
  "context": {"reprompt": false}
}
```

Continue from user input of [@option] dialog node

[@amenity:restaurant] node
Condition: **@amenity:restaurant**
Watson Response:

```
{
  "output": { "text": "Of course. Do you have a specific cuisine in mind?"},
  "context": {"reprompt": true}
}
```

[@cuisine] node
Condition: **@cuisine**
Watson Response:

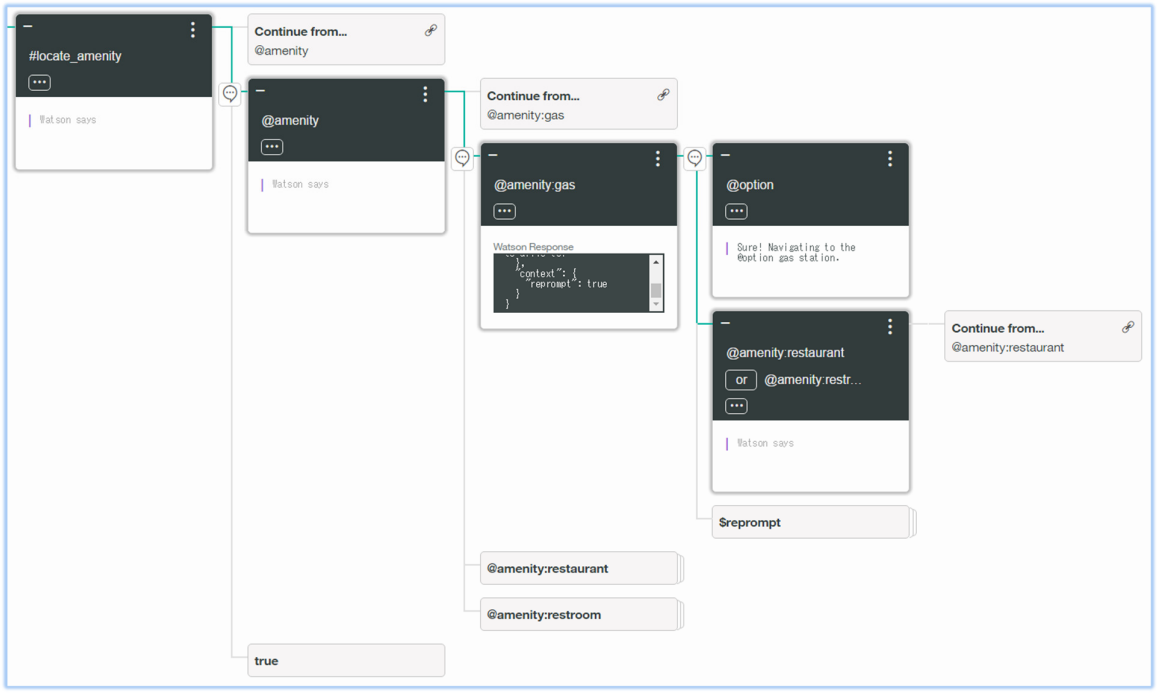
```
{
  "output": {"text": "Super! I've found some options for you. Which one do you like?" },
  "context": {"reprompt": true}
}
```

[@option] node
Condition: **@option**
Watson: says: **Sure thing. That place gets great reviews. You'll be there soon.**

[$reprompt] node

				Condition: **$reprompt** Watson Response: **{** **"output": {"text": "Which option would you like? You can say first, third, nearest and so on.." },** **"context": {"reprompt": false}** **}**
				Continue from user input of [@option] dialog node
			[@cuisine_bad] node Condition: **@cuisine_bad** Watson says: **It doesn't look like there are any restaurants that serve @cuisine_bad around you. Try saying things like burgers or tacos.**	
			[@amenity:gas OR @amenity:restroom] node Condition: **@amenity:gas** OR **@amenity:restroom** Watson says: (None)	
				Continue from condition of [@amenity:gas] node
			[$reprompt] node Condition: **$reprompt** Watson Response: **{** **"output": {"text": "I see a few nearby restaurants serving tacos, burgers, seafood and pasta. You can select one of these.."},** **"context": {"reprompt": false}** **}**	
				Continue from user input of [@cuisine] node
		[@amenity:restroom] node Condition: **@amenity:restroom** Watson says: **I've found the closest restroom stops. Which one would you like to drive to?**		
			[@option] node Condition: **@option** Watson says: **Ok. Navigating to @option restroom. We should get there quickly.**	
			[@amenity:gas OR @amenity:restaurant] node Condition: **@amenity:gas** OR **@amenity:restaurant** Watson says: (None)	
				Continue from condition of [@amenity:gas] dialog node
			[$reprompt] node Condition: **$reprompt** Watson Response: **{** **"output": {"text": "Which option would you like? You can say first, third, nearest and so on.." },** **"context": {"reprompt": false}** **}**	
				Continue from user input of [@option] dialog node
	[true] node Condition: **true** Watson says: **I understand you want me to locate an amenity. I can find restaurants, gas stations and restrooms nearby.**			

The conversation tree starting from **#local_amenity** dialog node looks like this now:



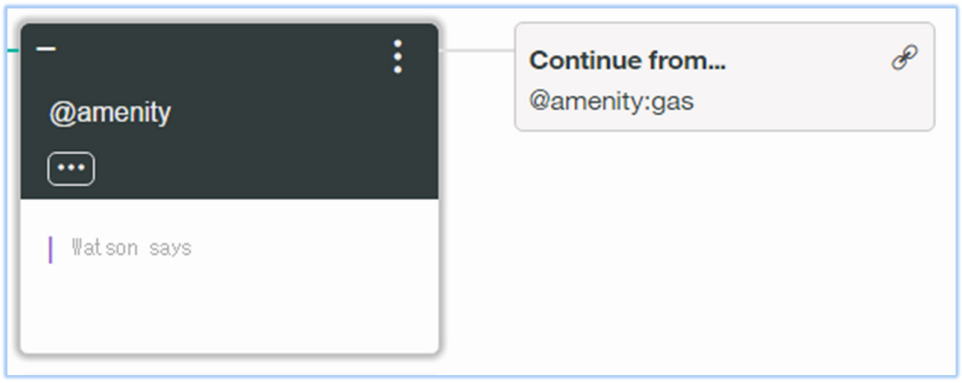
Finally, add another root node for requesting amenity location when *@amenity* entity is specified without *#local_amenity* intent.

149. Select the [#local_amenity] node, create a sibling node and enter:

```
Condition: @amenity
Watson says: (leave blank)
```

150. Created a new **continue from**

151. Select the [@amenity:gas] node under the [#local_amenity] > [@amenity] node branch and select **Go to condition**.



Now this **continue from** statement will continue from the condition of the [@amenity:gas] node and search matching node to the [@amenity:restaurant] and the [@amenity:restroom] nodes until it finds a matching node.

152. Test your final solution:

```
You> fuel
Watson> There are a few gas stations nearby. Which one would you
like to drive to?
You> first
Watson> Sure! Navigating to the first gas station.
You> meal
Watson> Of course. Do you have a specific cuisine in mind?
You> tacos
Watson> Super! I've found some options for you. Which one do you
like?
You> second
Watson> Sure thing. That place gets great reviews. You'll be there
soon.
```

If you had any problems during this exercise and would like to see the final solution, create a new application and load the JSON file: **\Lab1\car_workspace.json**

If you are now interested in how to integrate your service into an application and run it on Bluemix, continue with Lab 5.

2 Lab 2: Developing a Chatbot Using Conversation Service

Now that you have created the conversation bot, you can bind that service in your application.

During the first step of this exercise, you will run your app locally, using your conversation flow you just created. Then you publish the app to Bluemix to be available globally.

As a final step in this lab, you will extend the app to include an external service for retrieving weather information that can be accessed from within the car dashboard.

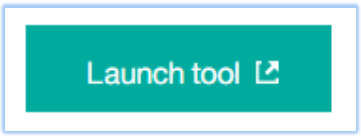
2.1 Preparation

1. Copy the zip file from github onto your machine and unzip it into \LabFiles\Lab2\
2. In a terminal window, navigate to **\LabFiles\Lab2**
3. Issue command `npm install`

2.2 Loading the Workspace

For this lab, you will load an already completed workspace to ensure that you can continue with the exercises, independently of whether you completed lab 1.

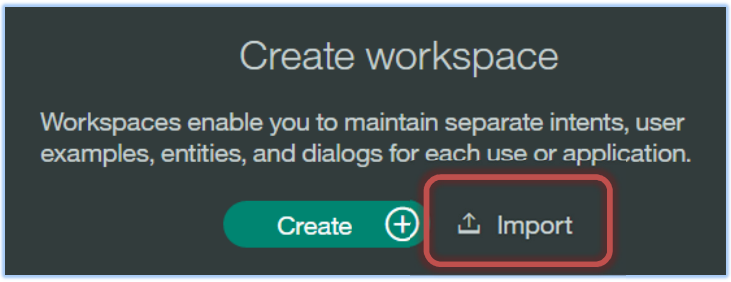
4. Log into Bluemix by entering: https://console.ng.bluemix.net into your browser (Chrome is preferred).
5. Access your conversation service from the previous exercise and click **Manage** in the upper left corner of the service
   (If you do not have a conversation service, follow the instructions in section 1.4 of this lab guide)
6. Start the Conversation Service tooling by clicking the the **Launch Tool** button



7. If you are prompted to log in, enter your Bluemix credentials

   A new window will prompt you to create a workspace.

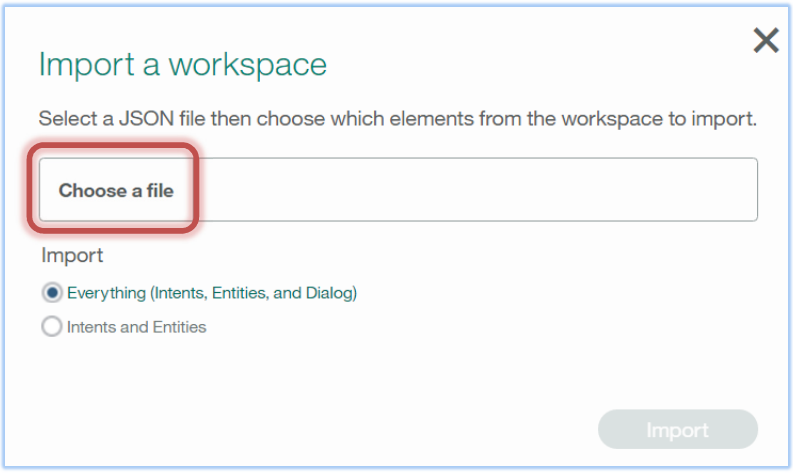
A workspace is a container for a conversation (dialog) flow along with the training data that the service uses.



For this exercise, you will use an already completed conversation flow 'workspace', which you will import as a .json file.

2.2.1 Creating a new workspace by importing a Conversation Flow

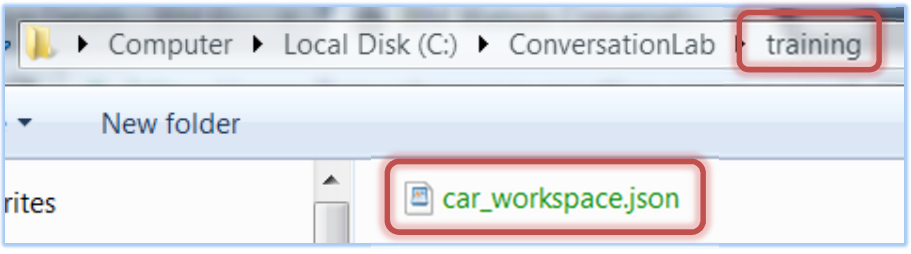
8. Select **Import** on the 'Create workspace' page



9. Select Choose a file

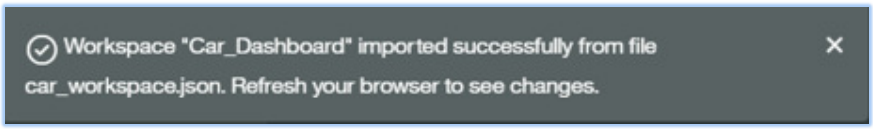
10. Navigate to **\LabFiles\Lab2\training** folder

11. Select the car_workspace.json file

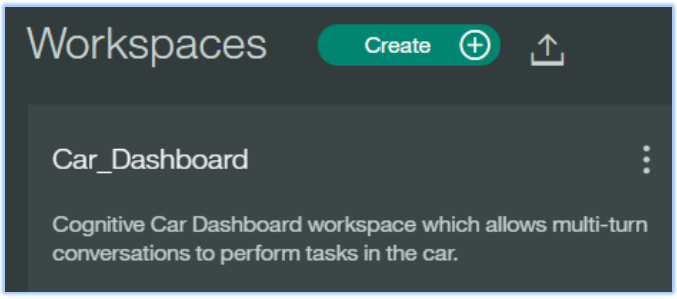


12. Click **Import**

    You should see a message indicating that the workspace was successfully imported



You will also see your new workspace **Car_Dashboard**



Your service is now available and can be bound (connected) to any app.

For this initial part of the exercise, you will be using a local web app that runs on your machine.

In order for that application to access your service, you must specify an .env system file and several credentials and IDs to access the service.

2.2.2 Creating the .env File

For your Web app to successfully connect to your service, you must provide three parameters to the code:

- conversation workspace ID
- username
- password

Remark: The .env file is a hidden system file that you can create using Sublime or Notepad++. There are ways to unhide system files if you want to view, or edit it (google is your friend ☺).

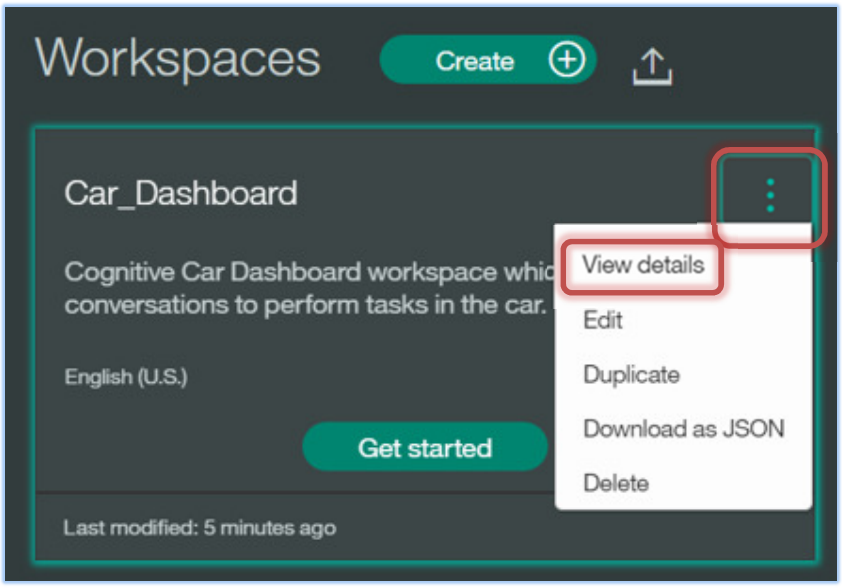
13. Open Sublime (Mac) or note++ (PC) to create the .env file

14. Type or copy and paste the following information into that file:

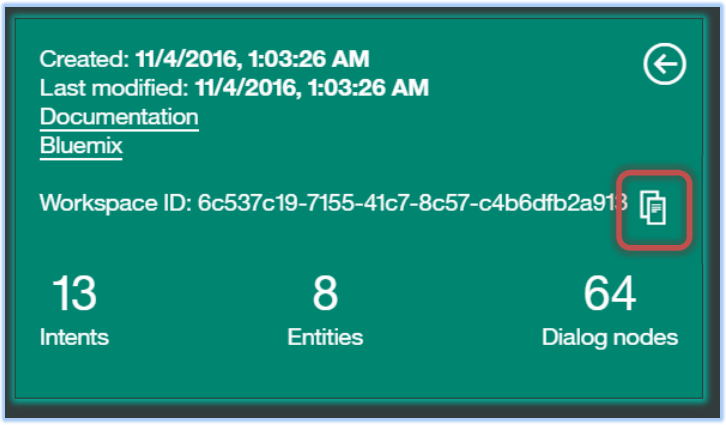
```
#environment variables
WORKSPACE_ID=
CONVERSATION_USERNAME=
CONVERSATION_PASSWORD=
```

2.2.3 Obtain your workspace ID

15. Obtain the workspace ID by clicking the three dots in the Workspace box and selecting **View Details**.



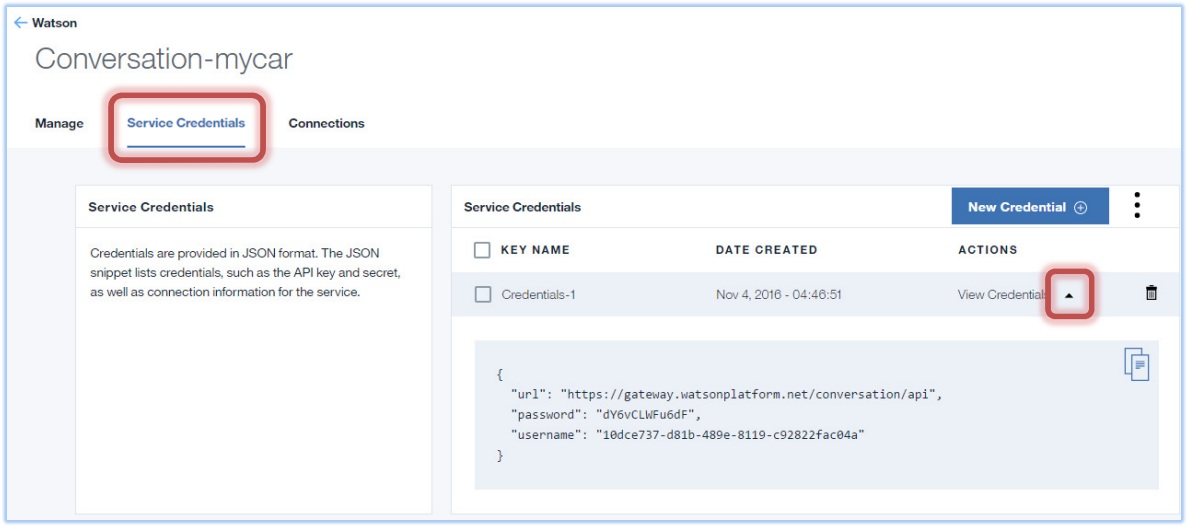
The box flips around and displays the workspace ID



16. Copy the ID by selecting the **copy** symbol, displayed at the end of the Workspace ID string
17. Paste the Workspace ID into the respective field in your editor

2.2.4 Obtain the credentials of your Conversation service

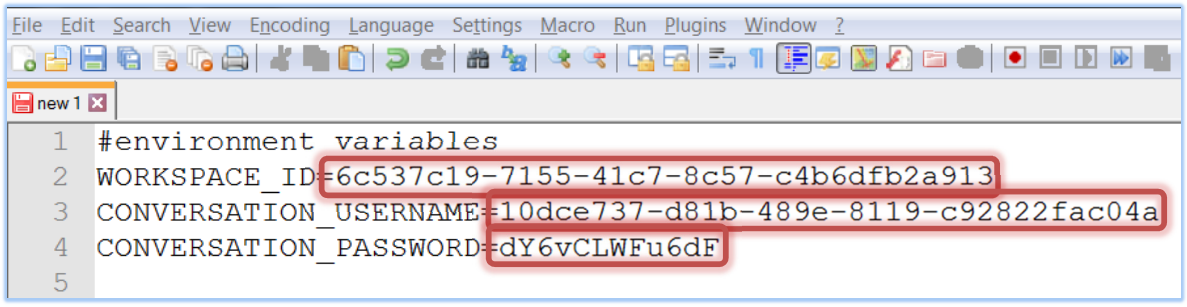
18. To view your service credentials, access Bluemix page, showing your service
19. If no longer open, click the service and select **Service Credentials**



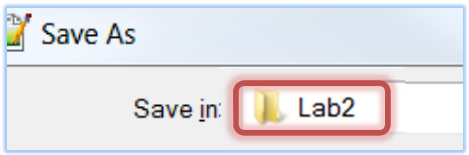
20. Click the little down arrow under **ACTIONS > View Credentials** to show the credentials

21. Copy and paste the username and password into the respective fields within your text editor (.env file)
Do not include the quotation marks ""

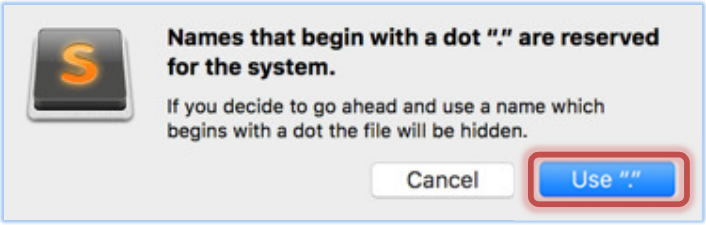
Your file should now look like this (with different values of course):



22. Save the file as **.env** in your \LabFiles\Lab2\ directory
Notice that there is a dot in front of the file name, which makes this a hidden system file.



23. If prompted (on Mac), click **Use "."**

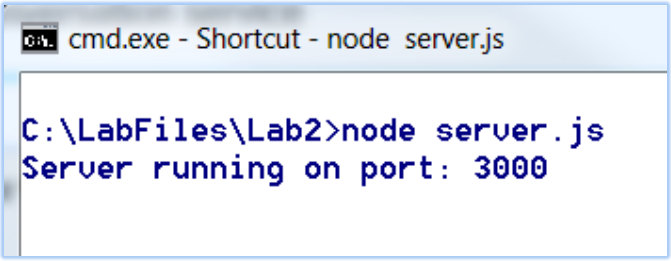


2.2.5 Start your Web Application

24. Return to the terminal window and run the command:

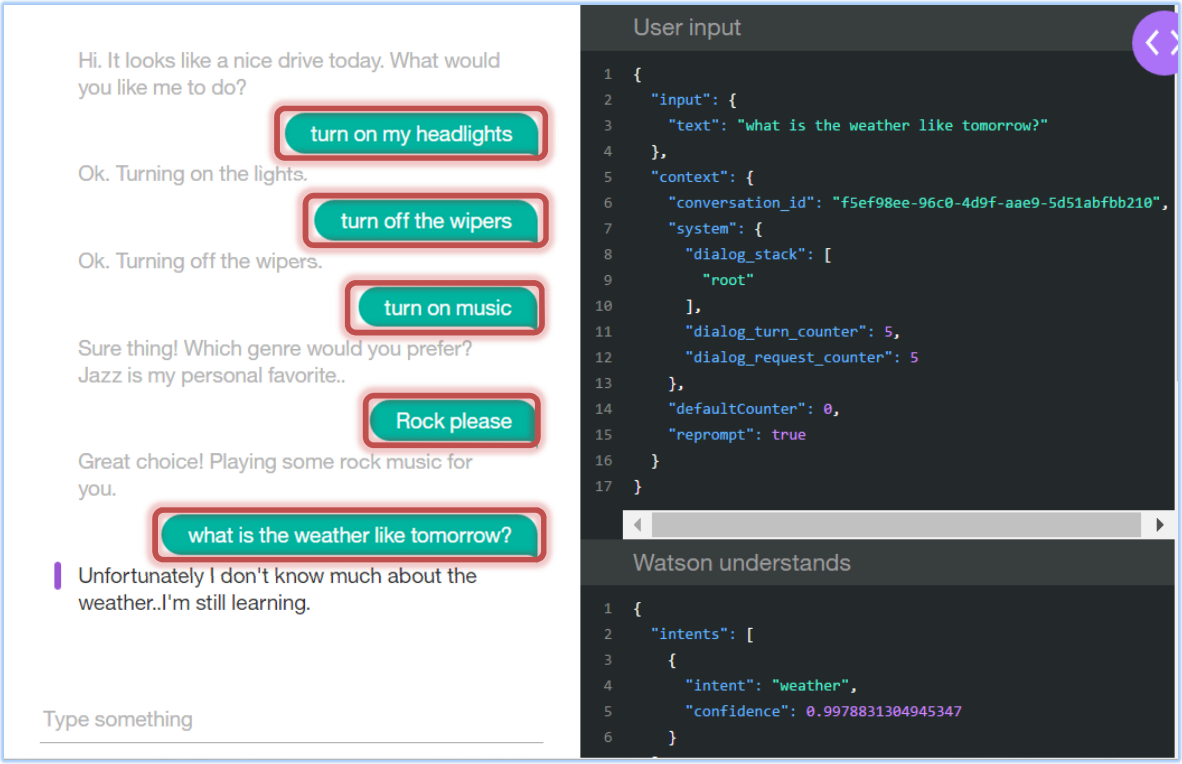
```
node server.js
```

Your web app is now running and can be accessed locally from a browser, using port 3000.



25. Open a new browser tab and enter **localhost:3000** as the URL

26. Test your application by running through a conversation as shown in the screen capture below:



Notice that the service does not yet know how to answer your question about the weather. However, it does recognize with very high confidence that you were asking about the weather (your question intent).

As part of this exercise, you will modify your dialog to present an answer based on where you are located.

You can obtain the actual weather forecast for where you are when you ask the question.

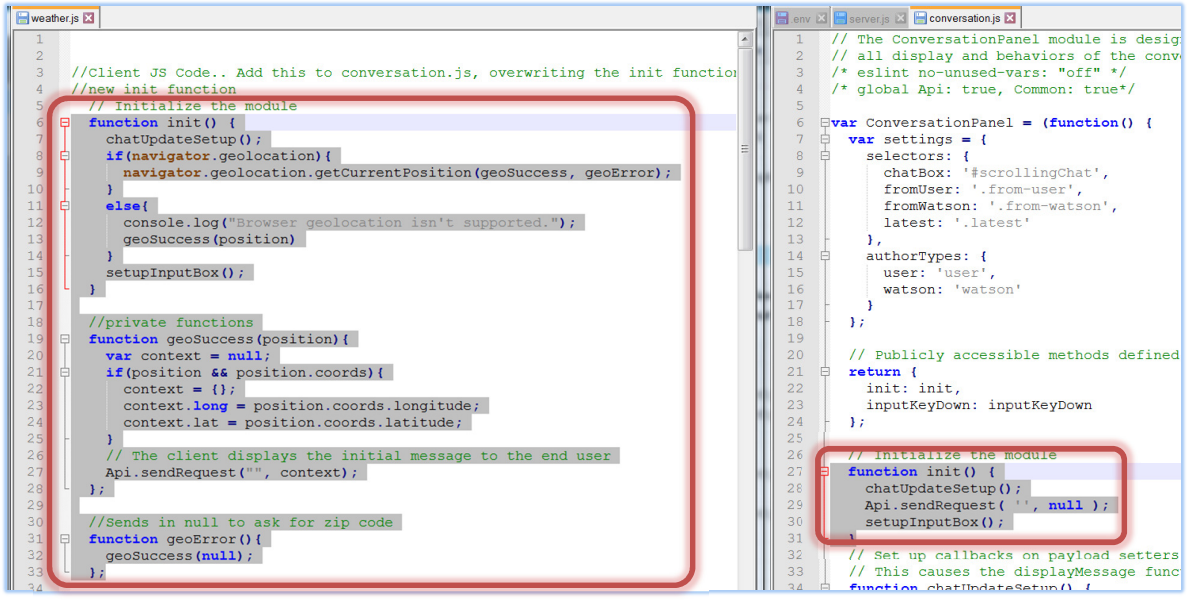
2.3 Binding the Conversation service with an external API

For this next section, you are going to use the geo-location capabilities of your browser to check the actual weather forecast. To do so, you will need to include the API key from the Weather Company web site that you obtained during the Prerequisites section of this lab guide.

27. From your \Lab2 directory, navigate to the **\AlreadyEditedFiles** folder and open the **weather.js** file.
    It includes two distinct sections:
    - The initialization function (line 6 through line 33) is used in the conversation.js file
    - The remainder of the code (line 53 through line 103) is used in the app.js code. The app.js file also requires two additional edits to work.

28. Navigate to the **\public\js** folder and open the **conversation.js** file.

29. Copy the **init()** function – rows 6 to 33 – from the **weather.js** file and replace the **init()** function in the **conversation.js** file – rows 27 to 31.



30. Save the **conversation.js** file and close it

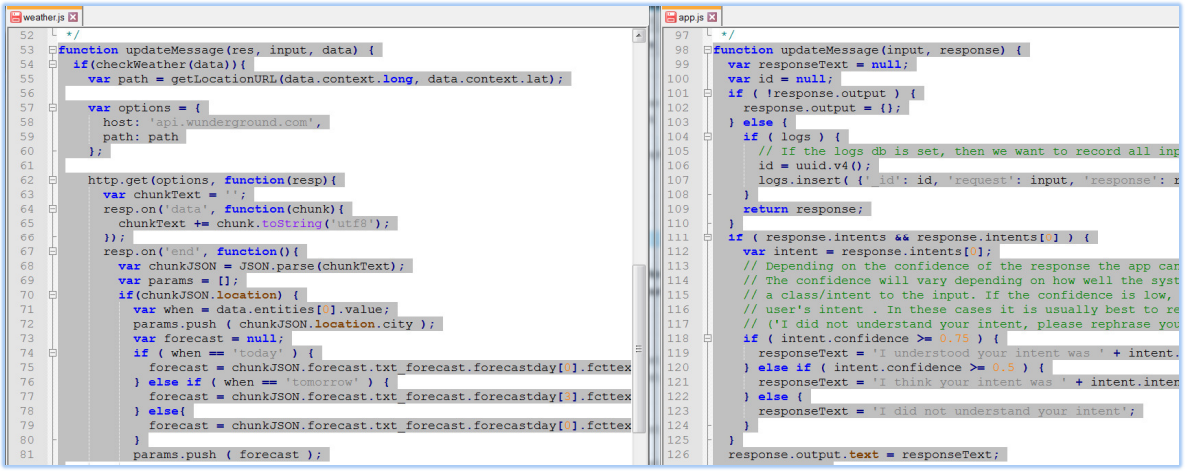
31. From your \Lab2\ folder, open the **app.js** file in your editor

32. Add the following variable to your other variables at the top of the **app.js** file

    `var http = require('http');` (line 24 will do nicely)

```
21  var express = require( 'express'
22  var bodyParser = require( 'body-p
23  var watson = require( 'watson-dev
24  var http = require('http');
25  // The following requires are nee
26  var uuid = require( 'uuid' );
27  var vcapServices = require( 'vcap
```

33. Copy lines 53 to the end (line 119) from the **weather.js** file and replace lines 99 through 134 (the updateMessage function) in the **app.js** file.
    (If the line numbers differ, search for `updateMessage` (Cntrl+F))



34. Locate **var key** = at the end of your file

```
162
163  var key = //"add your key in between the double quotes"; (and the semi colon at the end)
164
```

35. Include your! API key from the Weather Company as described

```
163  var key = "68d92ad5a902c901";
```

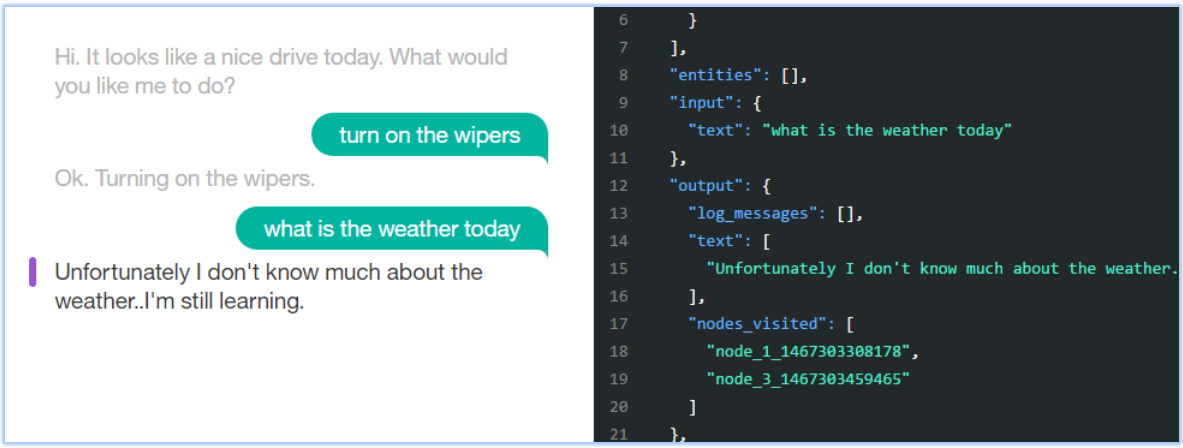
36. Save the app.js file in your \Lab2 directory

37. If your server is still running, stop it through Ctrl+C from the command line

38. Restart it by issuing the **node server.js** command

39. Ask the following questions:

    Turn on my wipers
    What is the weather *like today?*



Interestingly, all this work had very little impact on the result. Conversation service still understands that you were asking about the weather, but it is not able to provide a response.

As your next step, you need to incorporate your changes in the existing dialog structure.

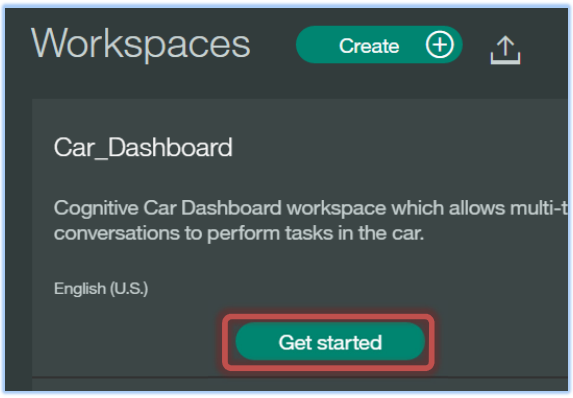
2.4 Updating Dialog

Working with Intents, Entities and Dialog is entirely separate discipline. If you are not yet familiar with this, you should go through Lab1 in this same lab guide.

For the purposes of this lab, you will edit the Entities and Dialog that you imported earlier into your Car_Dashboard workspace.

40. Go to the **Car_Dashboard** workspace

    You may need to click the left arrow to get back to the front page of the workspace card

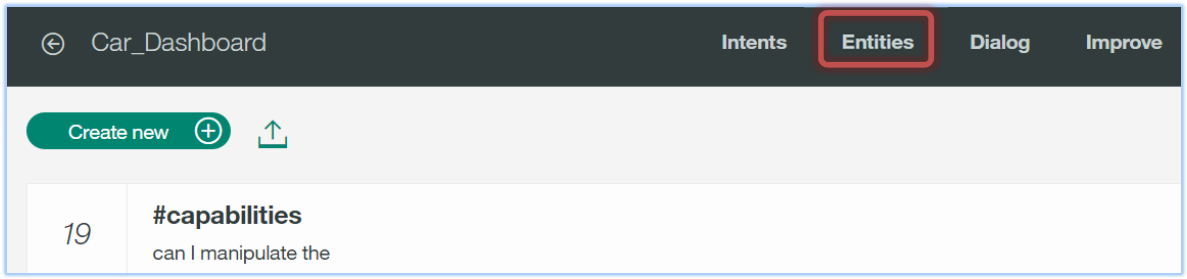


41. Click **Get started**

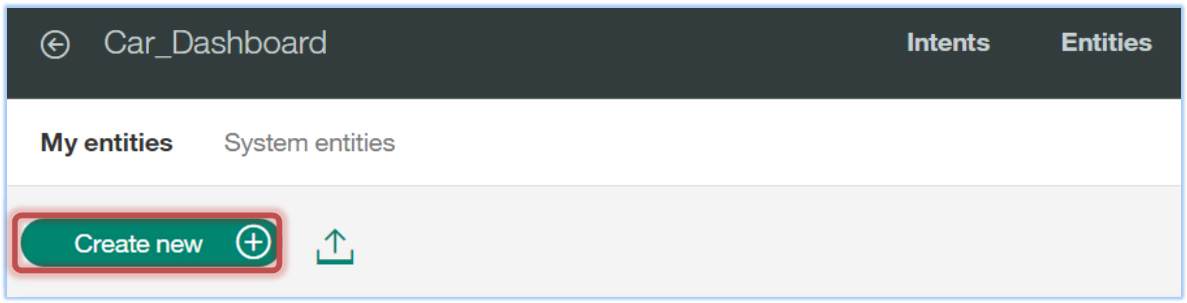
    You will see all the intents (#sign) that have been loaded into the workspace

42. Locate the one that says **#weather**.

    This intent is the reason that the service recognized our input as being weather related.



43. Click the **Entity** menu to identify when a person is asking about the weather



44. To add a new entity, click the plus sign and add an Entity:

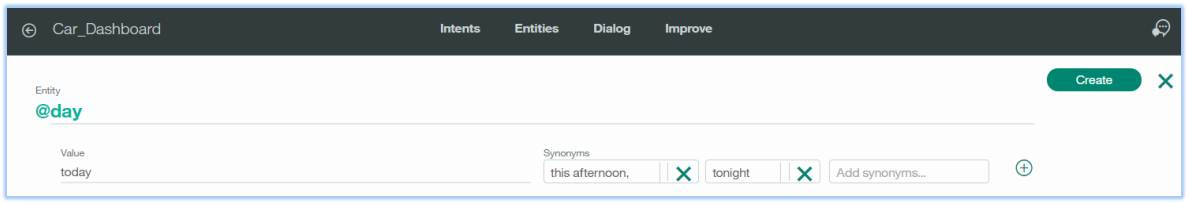
    `day` (case matters, use lower case)

45. Add the value:

    `today` (case matters, use lower case)

46. Add synonyms such as:

    `this afternoon` and **`tonight`** (case does not matter here, pick your own synonyms)

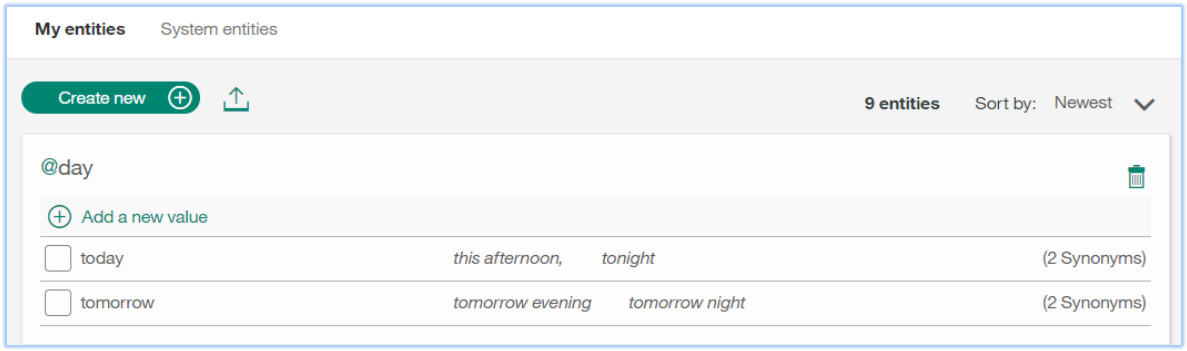


47. Click the **plus** sign and add another entity value of **`tomorrow`**

48. Add synonyms such as **`tomorrow evening`** and **`tomorrow night`**

49. Click **Create**

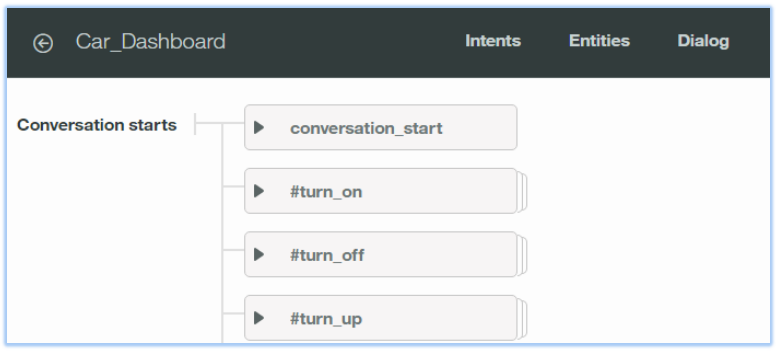
    Note that **#** signifies Intent and **@** signifies Entity.
    The result should look like this:

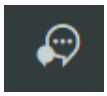


2.4.1 Update the Dialog Flow

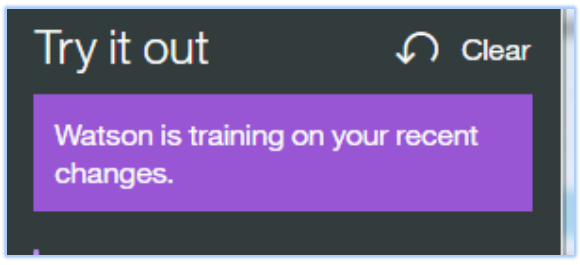
Now that you have intent and entity defined to deal with weather and the day the user is asking about, include those entries in your dialog.

50. Click the **Dialog** menu



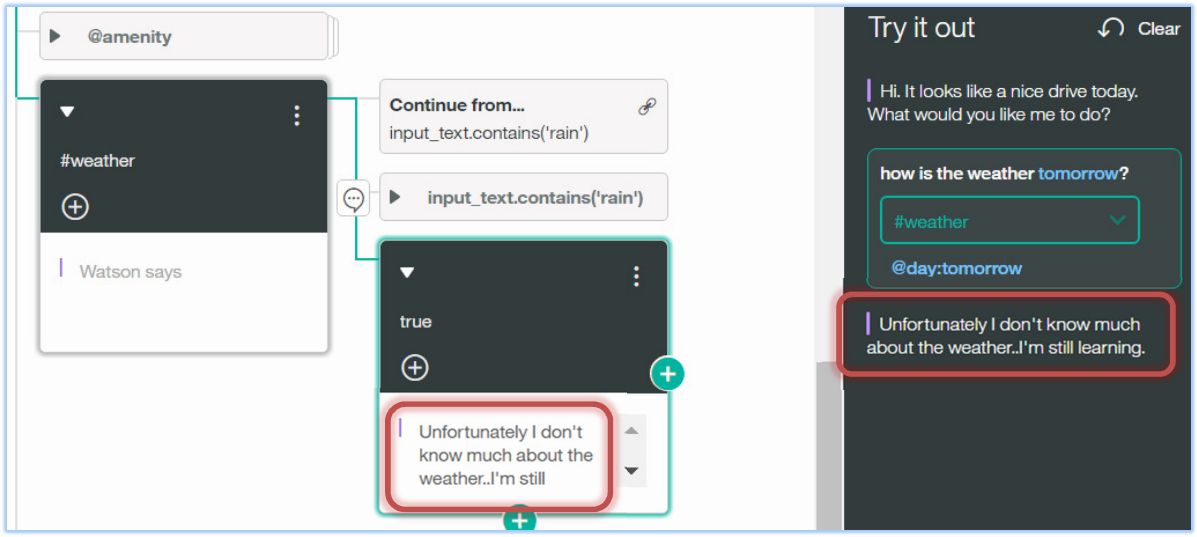
51. Click the chat icon  in the top right corner of the page to do a trial run.

    If a purple message appears that Watson is training on your recent changes, give it some time for that to finish (a green message will appear, indicating that training is complete).



52. Ask the system "`how is the weather tomorrow?`"

    The result should appear like this:

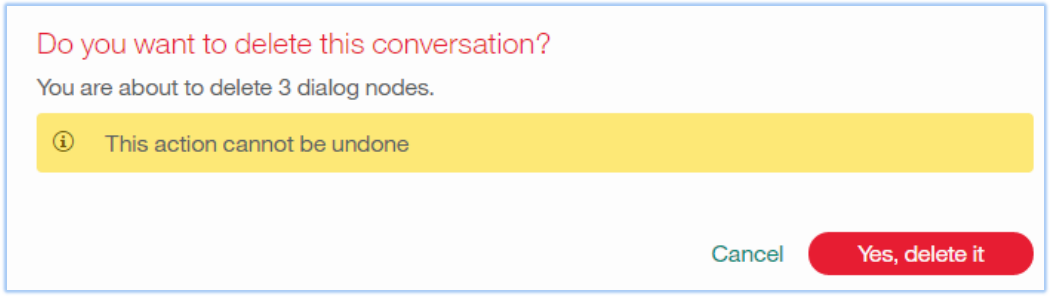


Notice in the chat flow on the left where the response came from.

53. Close the chat window with  and click the **#weather** Dialog node

54. Click the three dots in the **#weather** node and delete it

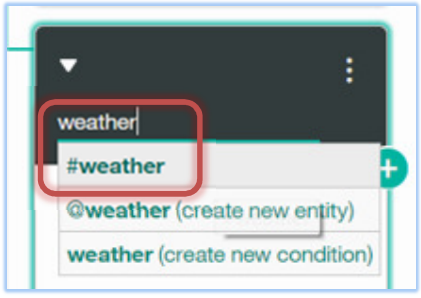
55. Confirm the warning notice that you are about to delete 3 nodes



56. Click Yes, delete it

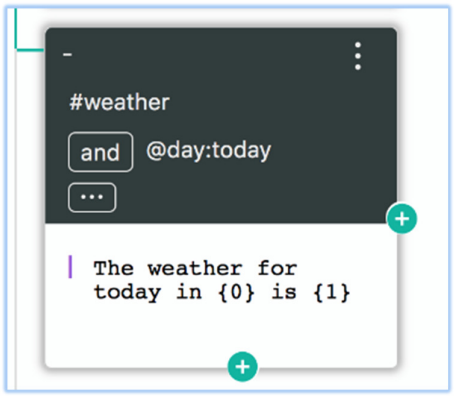
57. Click the amenity node (or any top level node) and click the plus sign in the bottom of the node to add another 'sibling' top level node

58. Type **weather** and then select the **#weather** for Intent



59. Click the **plus** inside the condition field and specify the condition as **@day:today**

60. Add the dialog: `The weather for today in {0} is {1}`



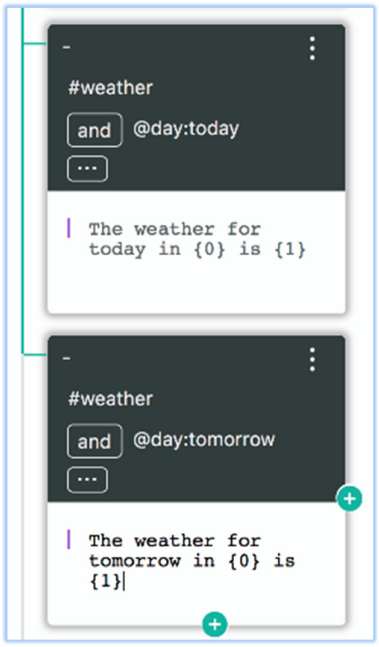
This node now means that if the condition (Intent: *weather* and Entity: *today*) is true then the dialog code gets executed.

What you are doing here is dealing with "open entities". In general, all entities are exactly defined and must be matched in order for the conversation service to return a correct response. By using variables {0} and (1), you can now use context from other APIs to create a custom response.

61. Click the plus sign underneath the dialog node you just created and add another node: **#weather**

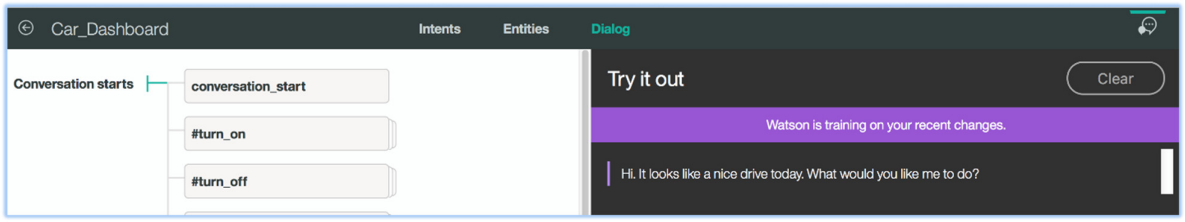
62. Add the condition of tomorrow: **@day:tomorrow**

63. Include the same script: `The weather for tomorrow in {0} is {1}`



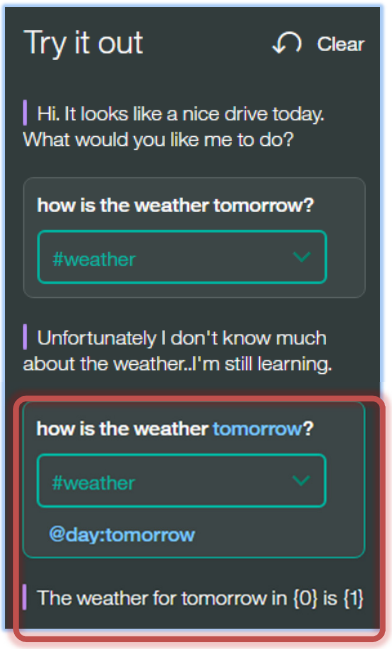
64. Select the **Ask Watson** icon

    Keep an eye on the message where it says Watson is training on your recent changes.



    It will turn green and state that training has finished and disappears shortly thereafter.

65. Test the conversation again asking the same question "`how is the weather tomorrow?`"



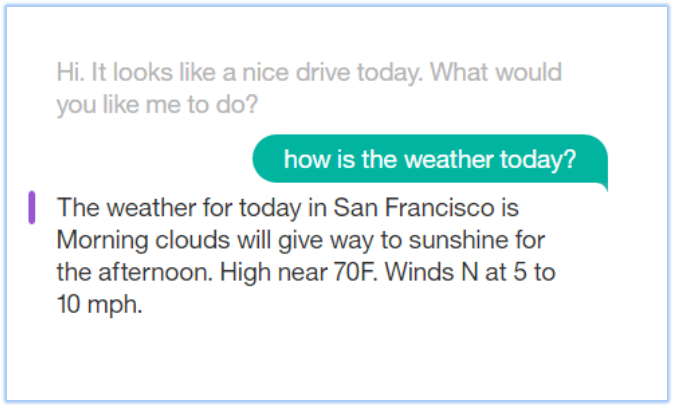
You can see that the formula from your dialog is being executed.

2.5 Test your application

66. To test your application locally, start your web app the same way you did earlier

67. Access your app through your browser

68. Enter the same question: `how is the weather today?`



If your answer does not reflect your current location, you can make the following changes (temporarily) in Chrome:

Manage location sharing

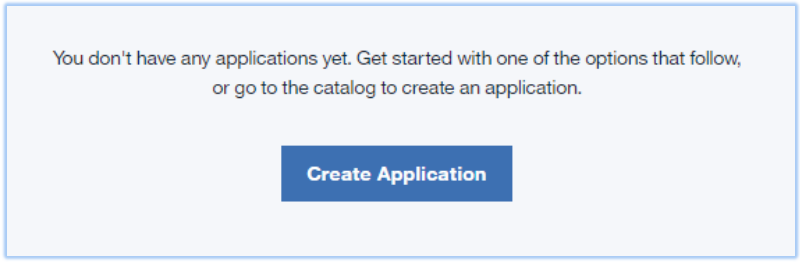
1. On your computer, open Chrome.
2. At the top right, click More Settings.
3. At the bottom, click Show advanced settings.
4. In the "Privacy" section, click Content settings.
5. In the dialog that appears, scroll down to the "Location" section.

2.6 Deploying your app to Bluemix

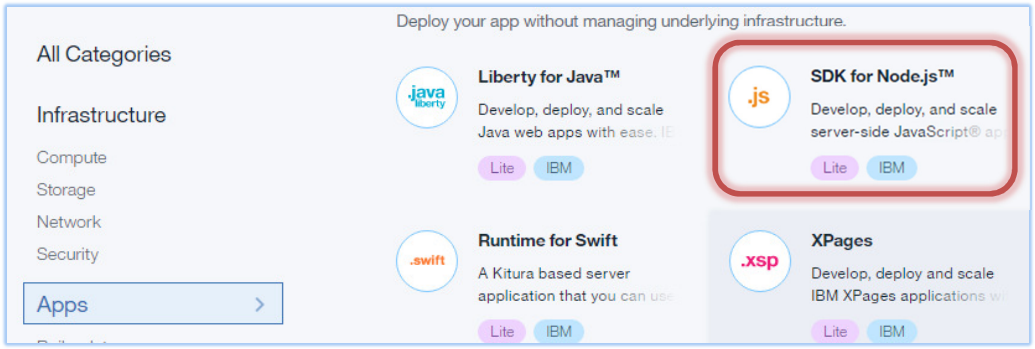
Now that your app is working locally, you may want to upload it onto Bluemix, so that everyone else can access your great work as well.

An easy way to deploy your app is by including an app name in the **manifest.yml** file and then using the **cf push** command to deploy the app to Bluemix.
In this guide, you will instead make greater use of Bluemix capabilities in ensuring that your app is deployed properly.

69. Go back to Bluemix and navigate to the **Dashboard**

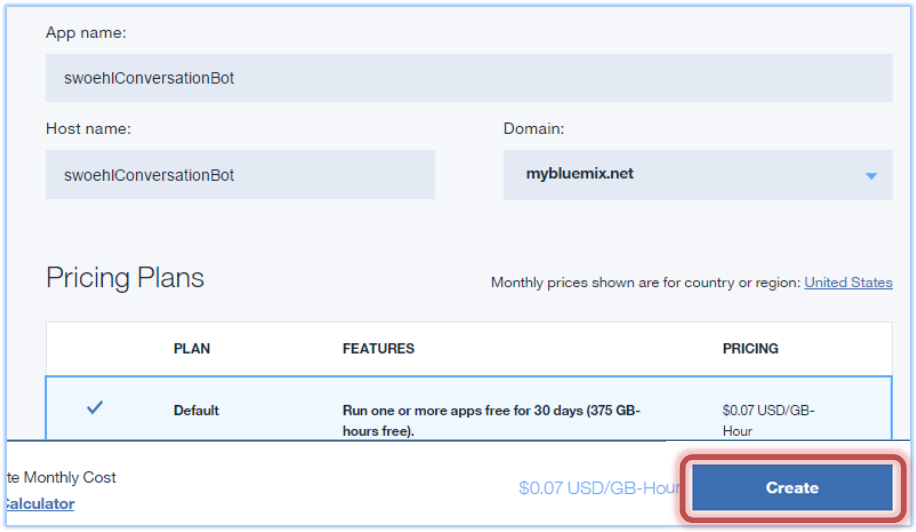


70. Click Create Application



71. Under Apps > Cloud Foundry Apps, select the SDK for Node.js

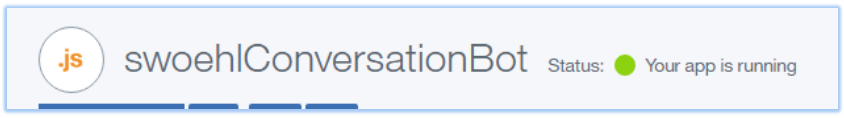
72. Give your application a unique name (i.e. <e-mail>ConversationBot)



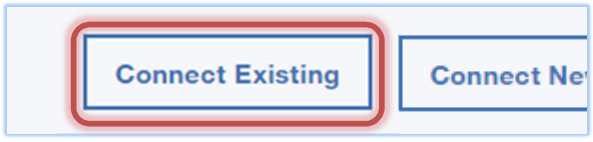
73. Click **Create**

    Wait until your app is fully staged and started. This can take a few minutes.

74. Select **Logs** on the left panel to observe progress and of course potential errors.

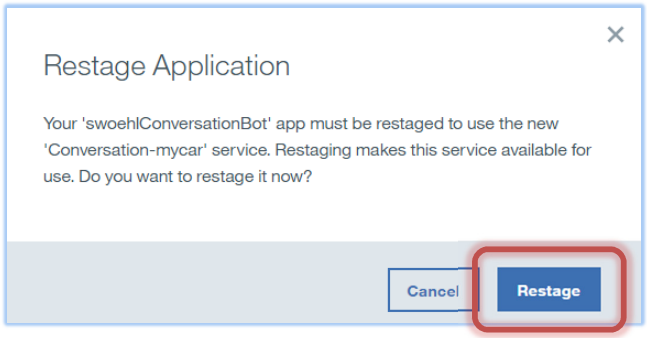


75. Once your App is running, click the **Connections** link in the left panel.



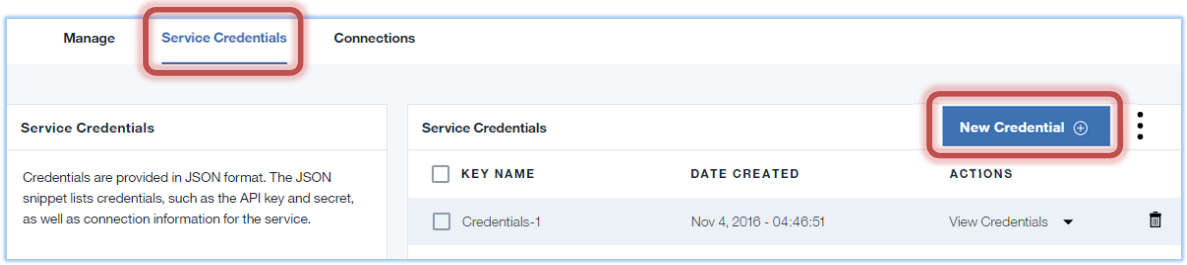
76. Click **Connect Existing** to bind an existing service to your new app

77. Select the conversation service you created at the beginning of this workshop and click **Connect**



78. To restage your app, click **Restage**

79. Navigate to the service

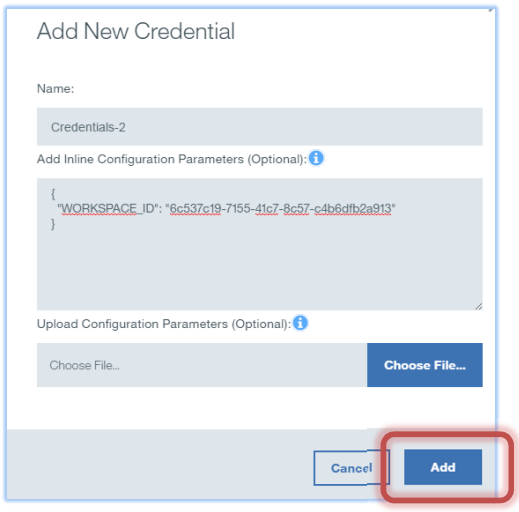


80. Click the **Service Credentials** link at the top of the page

81. Click New Credentials.

82. For name, accept default (Credentials-2) and for Add Inline Configuration Parameters (Optional), copy and paste the WORKSPACE_ID from the Conversation tooling UI, using JSON syntax:

```
{
  "WORKSPACE_ID": "12345678901234567890123 4"  (← include your ID here)
}
```



83. Click **Add**

84. Navigate back to your newly created application

85. Click the Getting **Started** link from the left panel

86. Click DOWNLOAD STARTER CODE

87. Extract the contents of the downloaded code into a temporary folder

88. Copy the **manifest.yml** file that has an app name and the user defined variable value, replacing the manifest.yml file in your working directory

89. To push the modified app back to Bluemix, access \Lab2 folder from the terminal console and type `cf push`

    Allow enough time for the app local to your machine to upload to Bluemix.

Your app is now running live in Bluemix. Send it to everyone ☺

2.7 Overview of the steps required to complete the workshop

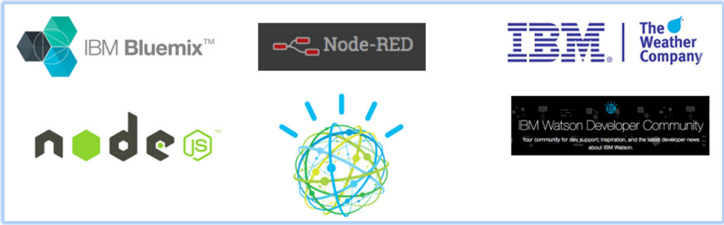
This table summarizes the steps that you need to complete to stand up an instance of the Watson Conversation service and integrate it an eternal service.

Step	Where do I go?	What do I do?
1	Github: https://github.com/apischdo/Bluemix-workshop-assets	Download the ConversationMaster.zip (you can also download everything that you see there). Alternatively, can also use the following command from a command prompt or your terminal: git clone https://github.com/apischdo/Bluemix-workshop-assets.git
2	Your machine, the terminal window	Run the command `npm install` from the same directory where you extracted or cloned the package
3	Bluemix	Create a Conversation service and launch the tooling
4	Conversation service tooling	Import the `car_workspace.js` file
6	Your machine: create a new system internal file	This `.env` file will contain the conversation ID and the username/password for the Conversation service
7	Your machine, the terminal window	From the same directory, run the `node server.js` command
8	Your machine: use the provided weather.js file	Update the conversation.js and the app.js files. You include the API from the Weather Company in this step.
9	Your machine	From the same directory, run the `node server.js` command to reflect your changes.
10	Bluemix:	Specify custom credentials
11	Your machine	Run the `cf push` command
12	Bluemix	Invoke the Routes URL and test your app on Bluemix

3 Lab 3: Creating an Application with STT, TTS and Weather

We all lead busy lives, so it can be a challenge to find timely and relevant information. But what if you had a personal assistant to deliver this for you? In this lab, you will build your own cognitive cloud app powered by IBM Watson to deliver news and weather customized for your preferences and location. You will build your own Node.js application deployed to IBM Bluemix and use services from the Watson Alchemy News and the Weather Company. News articles are analyzed for category and sentiment and a real-time weather forecast is delivered for your location. You will interact with your assistant using Watson speech-to-text and text-to-speech services. In the process of doing all this, you will also learn more about Node-RED, Node.js and Bluemix, as well as some of the services that Bluemix has to offer.

3.1 Technology Overview



3.2 Node-RED

Node-RED is an open-source visual tool, originated at IBM and now owned by the JS Foundation, that allows you to wire together IoT devices, API's and online services in interesting ways to build applications. A payload of data in JSON format flows from left to right and travels through a series of nodes that are connected by wires. Something triggers the flow -- could be an IoT device, or a twitter stream, for example, and then the nodes after it can analyze the data and decide on an action and finally an output node will send data somewhere (for example, to a database, or an alerting system). This is an overview of how Node-RED works -- You can learn more about Node-RED at nodered.org.

For the application you will be building today, your Node-RED flow will be triggered by a REST call issued by a microphone application that uses Watson to record and translate your voice to text. That text is then sent to your Node-RED instance via REST call. Your Node-RED instance will be a hosted application on Bluemix, IBM's PaaS. We will provide the microphone application, which acts like a client, and you will use Node-RED to build the backend that services the incoming text request and returns a voice response, which will then be played by the client application on your laptop's speaker. Your application will utilize news, weather and Watson text-to-speech services to create the response.

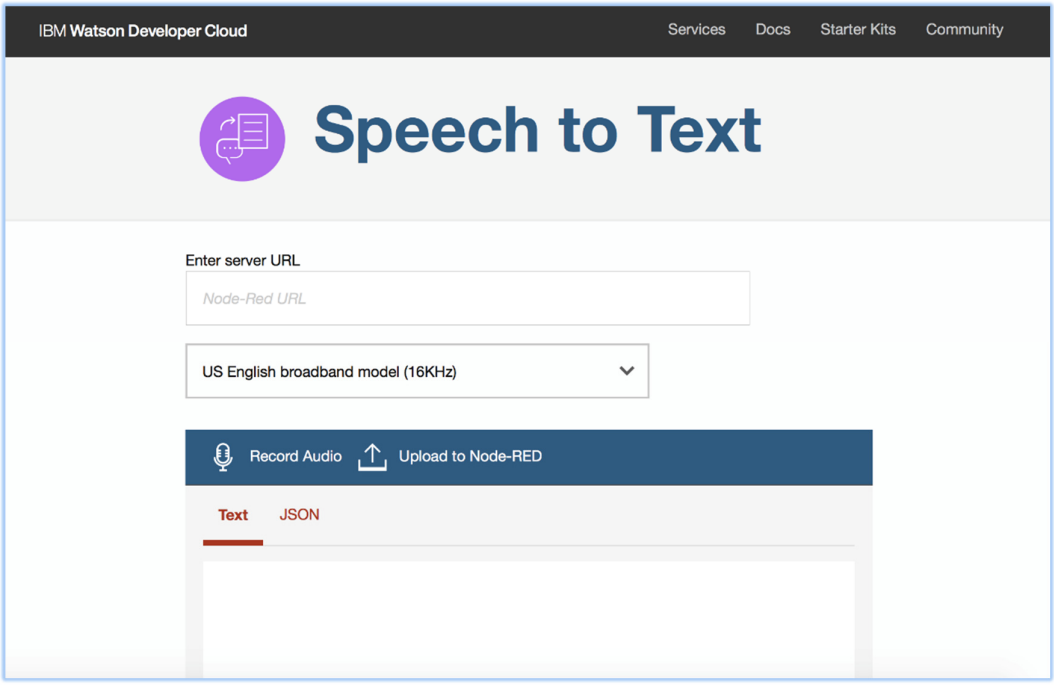
While Node-RED can run on many different hardware, including Raspberry Pi, in this lab, you will be working with a special version of Node-RED that is hosted on the cloud with Bluemix and has special nodes that can interface with Bluemix services including Watson services. If you are interested in learning more about how to build Node-RED applications that interface with Bluemix Watson services, this link has a variety of interesting starter-pack applications.

3.3 Watson speech-to-text and text-to-speech services

Watson speech-to-text service converts the human voice into the written word using machine intelligence to analyze the grammar and language structure to generate an accurate transcription in real time. The transcription of the incoming audio is sent back continuously and corrected as more speech is heard. In our demo app, you will see confidence scores from Watson assigned to each word along with alternative words that it was considering.

You can try it out in your favorite language at this link -- please use the Chrome browser:

https://wow-mic.mybluemix.net/



1. Click "Record Audio" and wait for it to go red before you speak, then click it again to stop recording. Don't worry about the Node-RED server URL yet. You will use that later in this lab.

The Watson text-to-speech service produces natural-sounding speech from input text with appropriate cadence and intonation and supports a variety of languages.

Learn more about Watson-powered STT and TTS at these links: http://www.ibm.com/watson/developercloud/speech-to-text.html https://www.ibm.com/watson/developercloud/text-to-speech.html

3.4 The Weather Company Data service

The Weather Company Data Service from Bluemix lets you integrate weather data from The Weather Company into your IBM Bluemix application for any specified geolocation. The service supports real-time, as well as forecast and historical weather data.

The Weather Company was acquired by IBM in 2015 and is the world's largest private weather enterprise, delivering up to 26 billion forecasts daily.

TWC powers top weather apps on all major mobile platforms, so you have likely used it before, "weather" you knew it or not.

3.5 Alchemy News service

The Alchemy Data News service indexes hundreds of thousands of articles and blogs every day, analyzes the text with machine learning algorithms, and makes those articles searchable across a variety of fields, including concept and sentiment. API documentation for the Alchemy News service is located here:

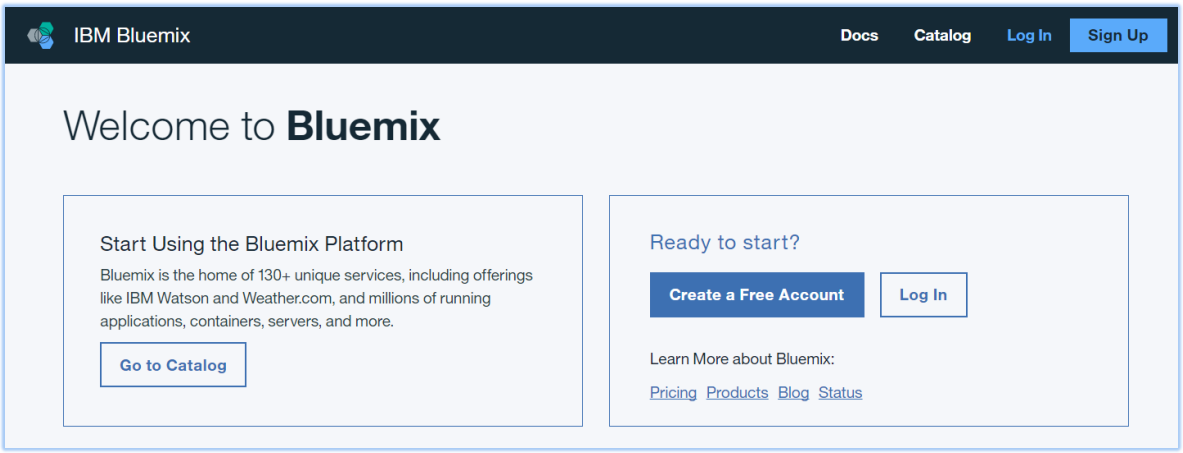
http://docs.alchemyapi.com/docs

3.6 Setup Instructions

Let's get started on building your application!

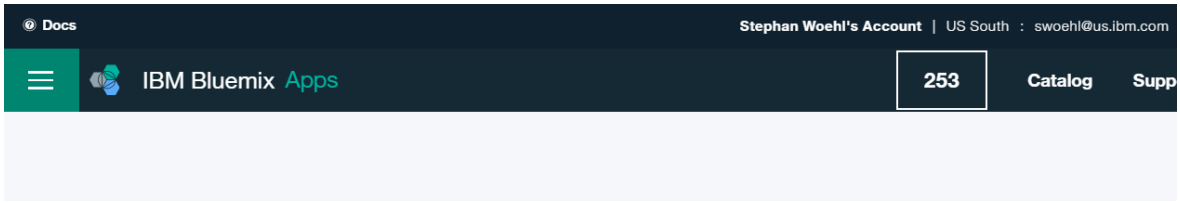
2. Using a Chrome browser, go to Bluemix at https://console.ng.bluemix.net
(.ng stands for North America with the Bluemix public datacenter in Dallas, TX)

3. If you have not yet done so, create a new account using your e-mail address.
(Feel free to create a separate account using a personal e-mail address for this lab)



→ You will be asked to select a unique name for your organization and space where you would like to host the application that you will build today.

Log into your Bluemix account

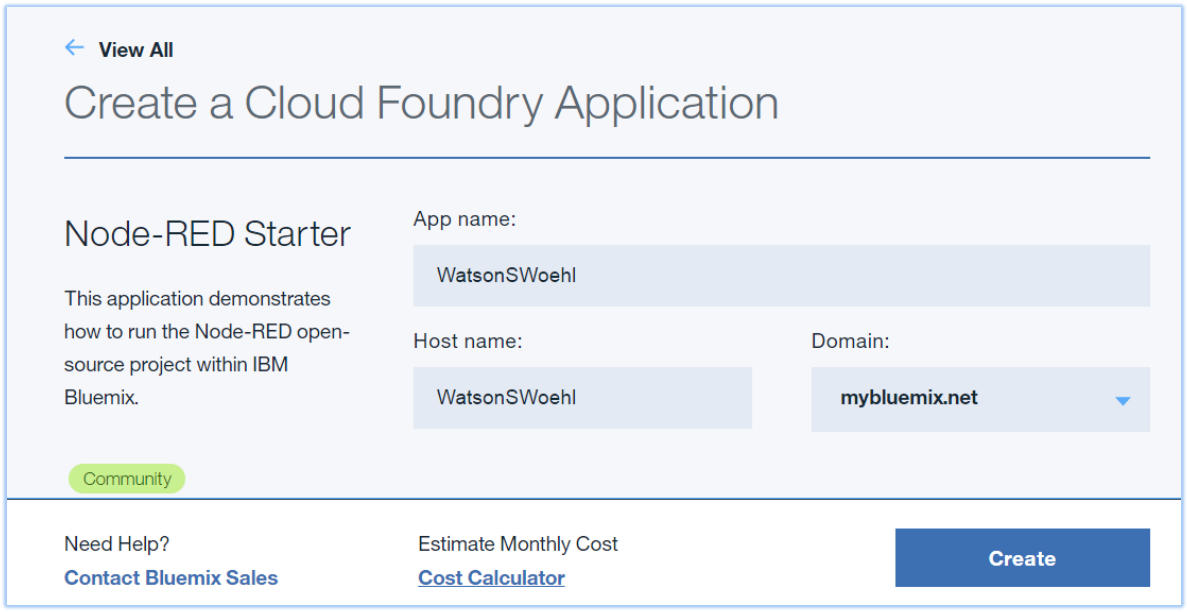


4. Select **Boilerplates** in the left menu to display available boilerplate code.

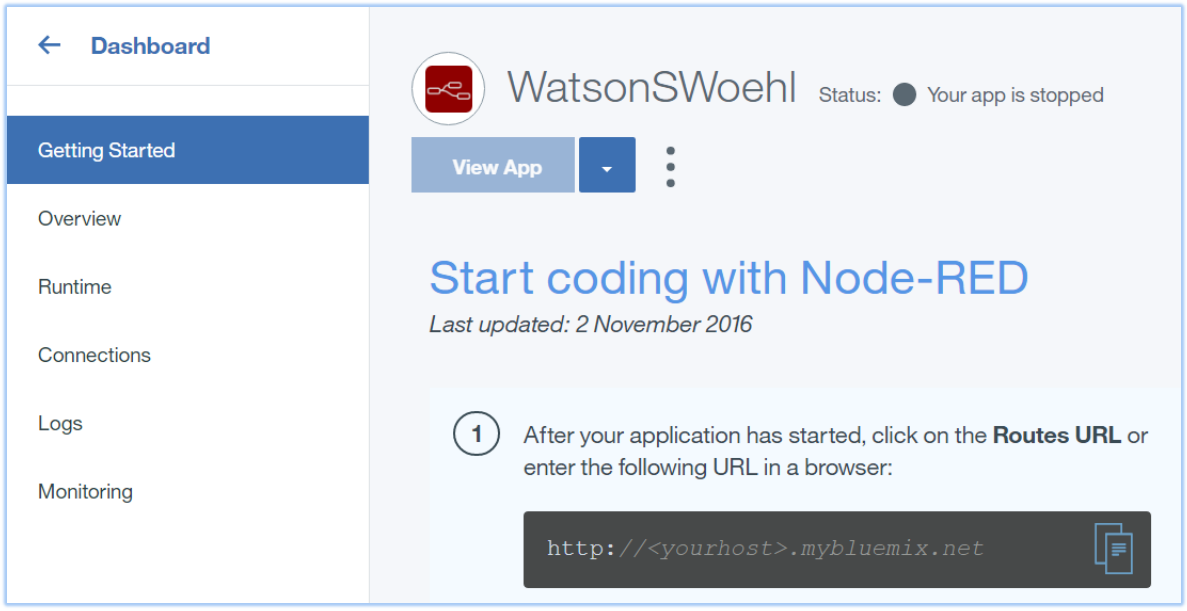
5. Scroll down and select **Node-RED Starter**

6. Give your application a unique name, such as ConversationLabWatson<e-mail_name> where <e-mail_name> is your short name before the @ sign

7. Click **Create**

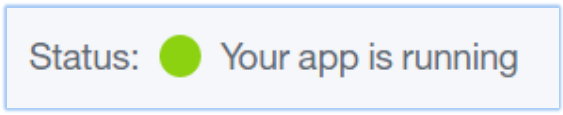


→ Your application is being created and starts running. It will have Cloudant database by default to store node red metadata.

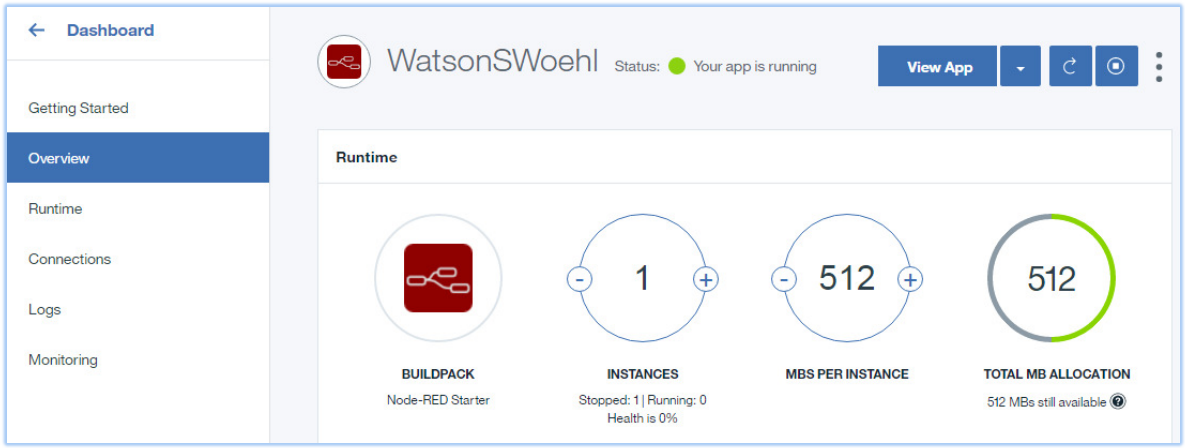


It may take several minutes for your app to stage and start. You can monitor progress in **Logs**.

Be patient until it shows the "Your App is running" indicator.



8. Click **Overview** in the left panel to see the detailed view of your application.

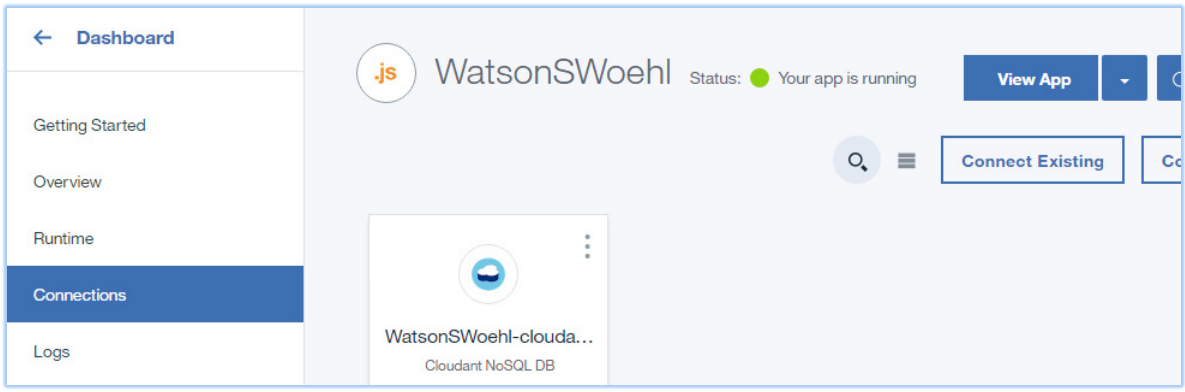


9. Click the **Connections** entry in the navigator

10. Add two more services (**Weather Company Data** and **Text to Speech**)
→ Both can be found under **Catalog** and by typing part of the service name

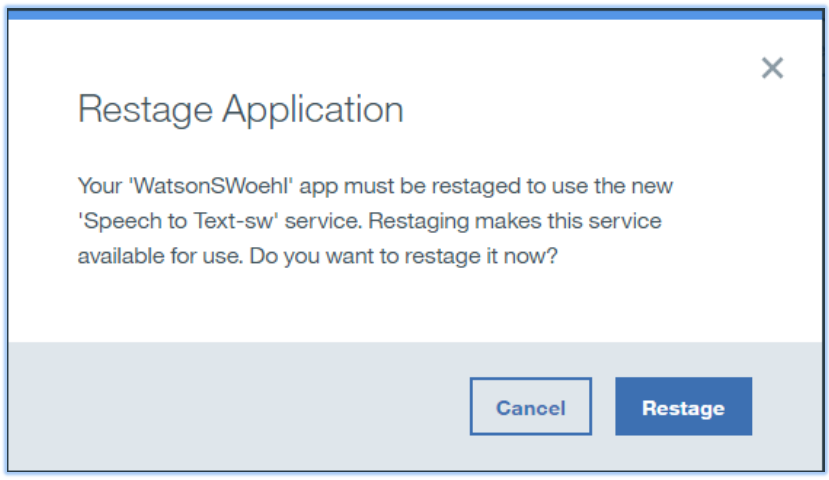
Next, you need to connect your services to your application. You can either do this from within each service by accessing the Connections link and selecting the application, or from within the application by connecting Existing Services.

11. Access the Dashboard and click your application.
12. Click **Connections** in the navigator.

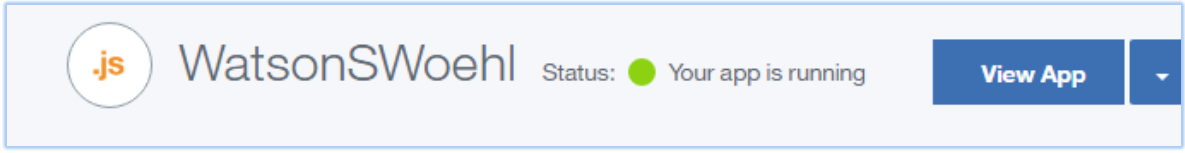


13. Click Connect Existing
14. Select each service and click **Connect**

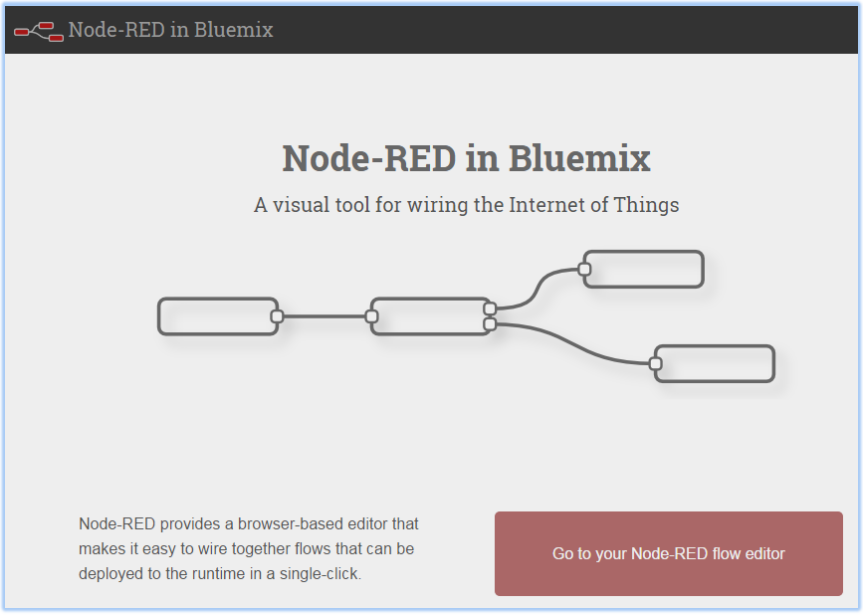
    → Each time you connect a service, you are asked to restage the application



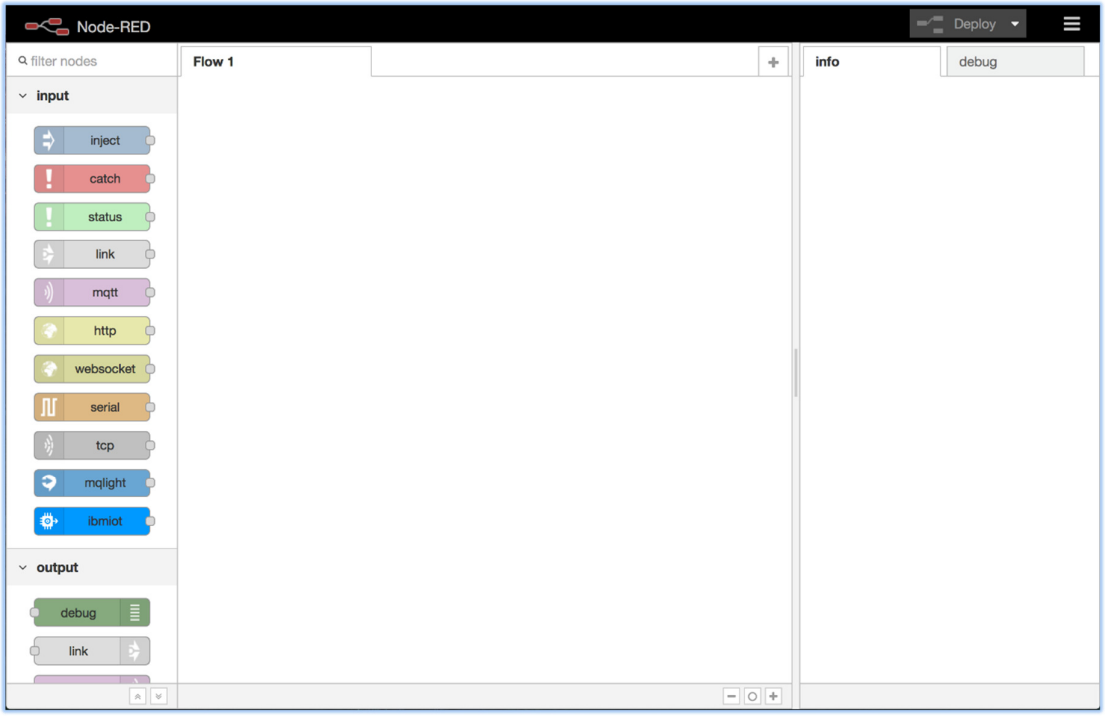
15. Click **Restage**
16. Make sure your application is running and click **View App**



→ You will be taken to **Node-RED**



17. Click the red **Go to your Node-RED flow editor** button to open the Node-red editor.

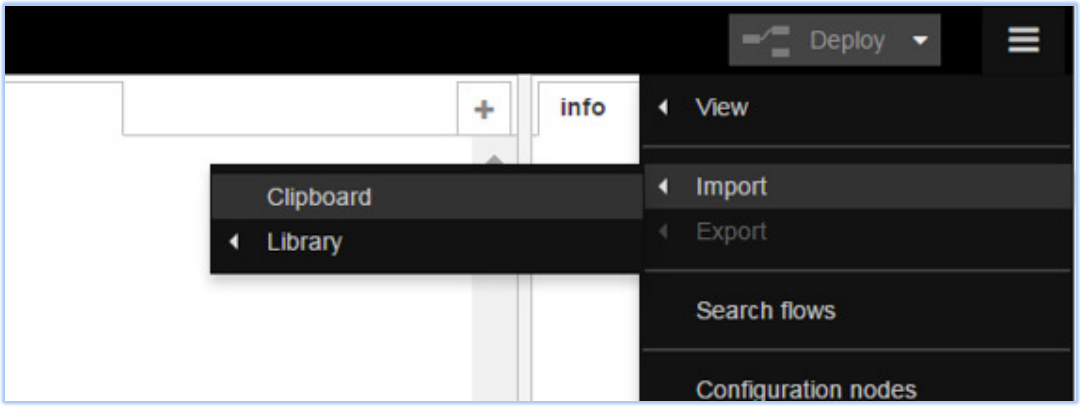


Copy the flows from the github repository and import them into your Node-RED editor as shown below.

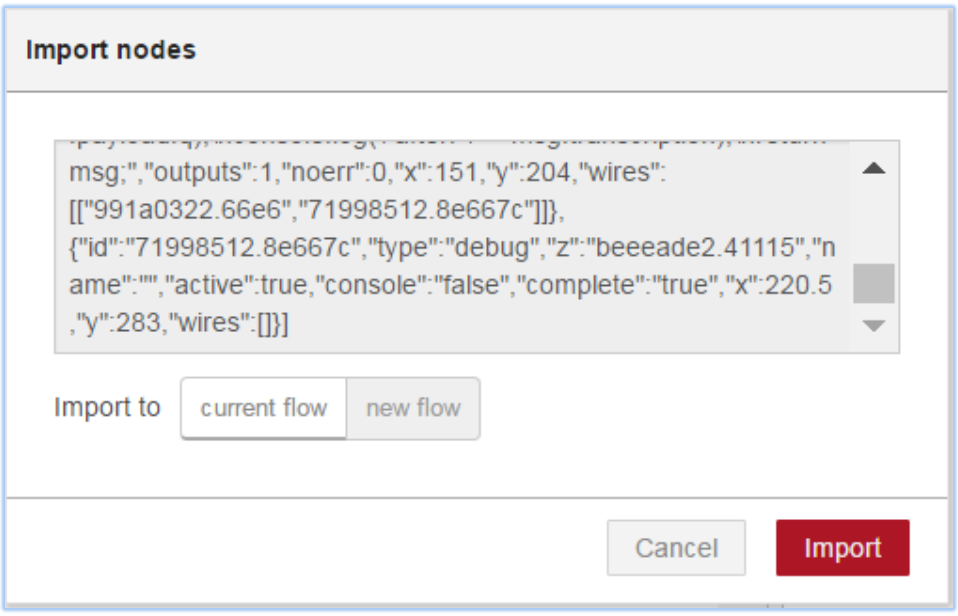
Ensure that you import each flow into a different tab

18. Start by selecting the three horizontal bars in the upper right corner of the window.

19. Select **Import** > **Clipboard**



20. Copy and paste the code in Main Flow into the first Flow tab

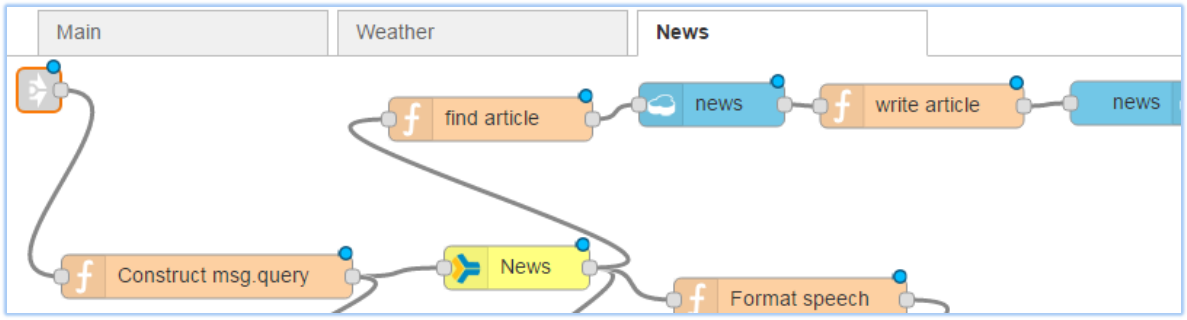


21. Click **Import**

22. Click the **+** sign to create a new tab and import Weather Flow

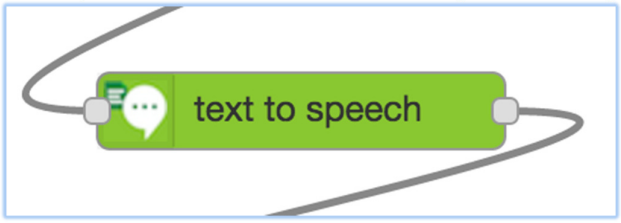
23. Create a third tab and import News Flow

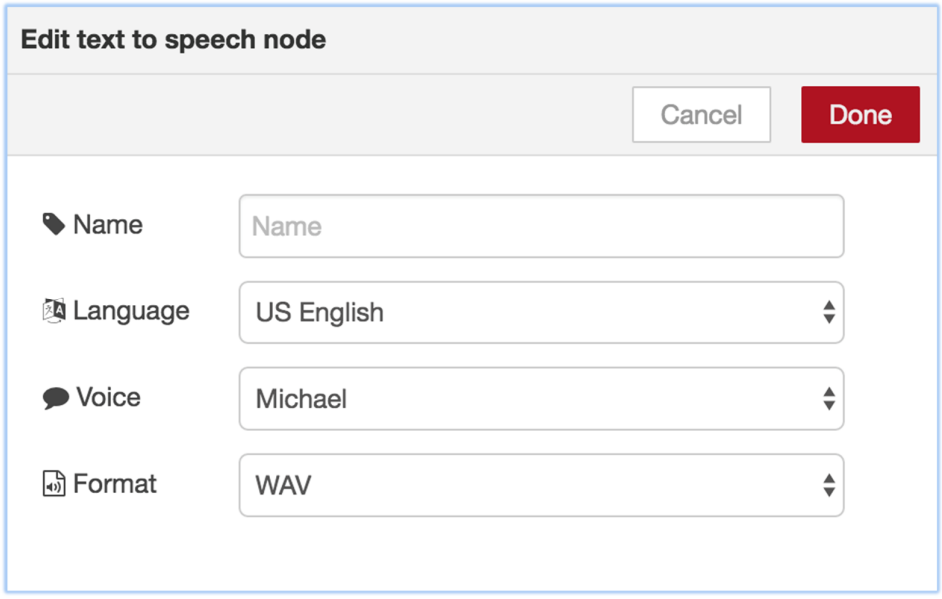
24. Double click each tab header and name them “main”, “weather” and “news” respectively.



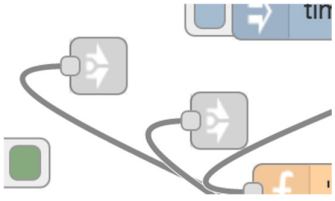
3.6.1 Nodes in Node-RED Editor

25. Click the "text to speech" node and select "US English" on language dropdown.





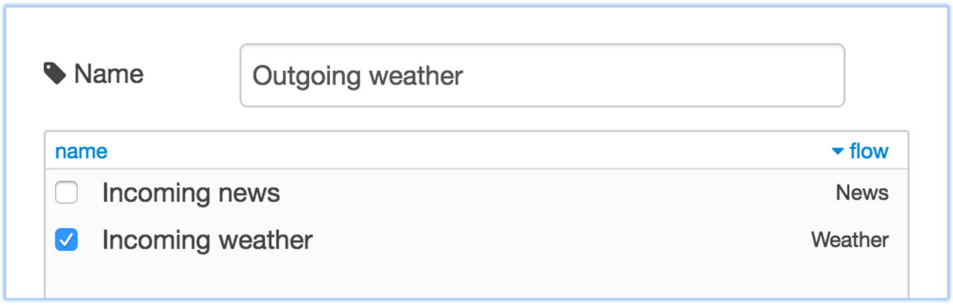
26. The "name" field can be left blank. Finally, click on the each link node on the main tab.



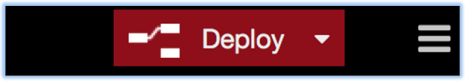
27. Connect to the proper receive node.

    i. Outgoing weather should connect to Incoming weather and

ii. `Outgoing news should connect to Incoming news`



28. Click on **Deploy**



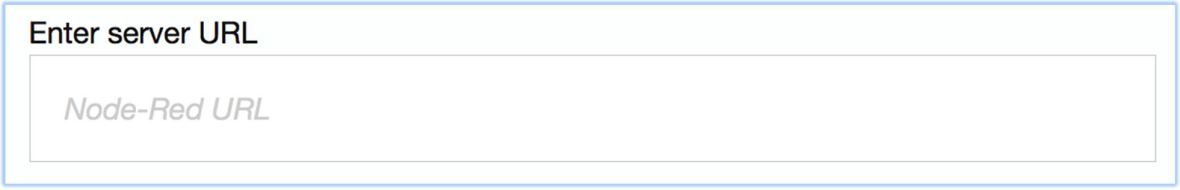
Speech to text Microphone application

- We are using another Bluemix application to record our message and send it to our node-red app. Check the link below.

https://wow-mic.mybluemix.net/

- Note that if you have time / interest, you can also clone this code yourself and host it on Bluemix (you will also need to create and connect the Watson "speech to text" service to it): https://github.com/CDSLab/WOW-mic

To use the microphone client app, simply enter your Node-RED application link in the text box shown, as: <--host name-->.mybluemix.net



Example: watsonkg.mybluemix.net Click on **Record Audio** to record your message eg.

```
What is the current weather in Chicago.
```

Stop recording and upload it to **Node-RED** by clicking on the "Upload to Node-RED" button.

If the URL you entered is wrong, an error message will pop up like below:

If everything works, you will receive a voice response (turn up your speakers).

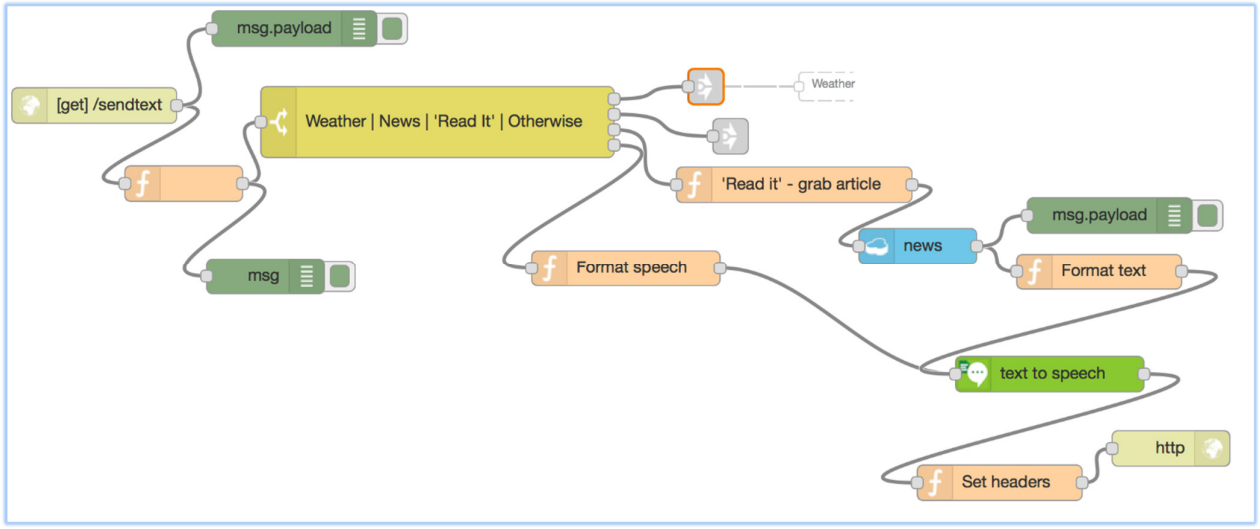
Questions currently supported are:

- What is the [current] weather in (location) ?
- What is the weather forecast for (location) ?
- Give me [positive | negative] news about (topic for title search)

Application Flow Details

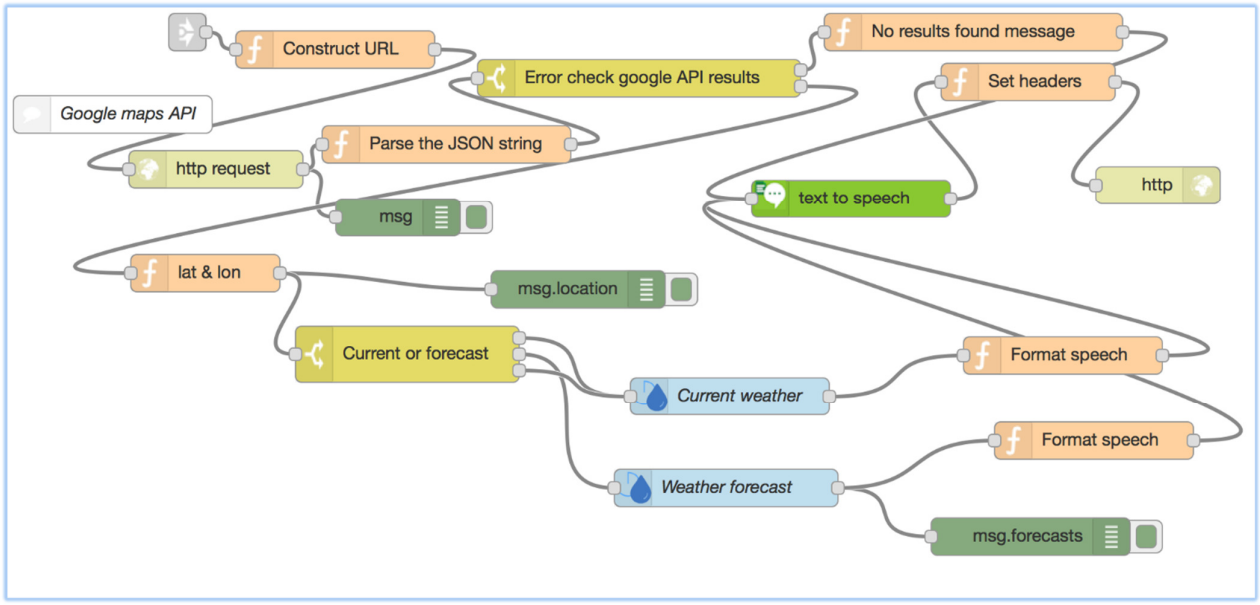
Now we will study each flow a little further to understand more fully how this application works.

Main Application Flow



Explanation: Text is passed in via a REST endpoint. Then we feed that into a function node which calls decodeURIComponent to remove the URL format of the text. That text is then fed into a switch statement which routes traffic to the news or weather flows based on if the input text contains "news" or "weather" keyword. If the text contains "read it", then the flow grabs the last requested news article from Cloudant and flows that into the text-to-speech node. If the text does not contain any recognized keywords, then pin 4 goes to a "format speech" node which formulates a catch-all response, "I don't understand what you meant".

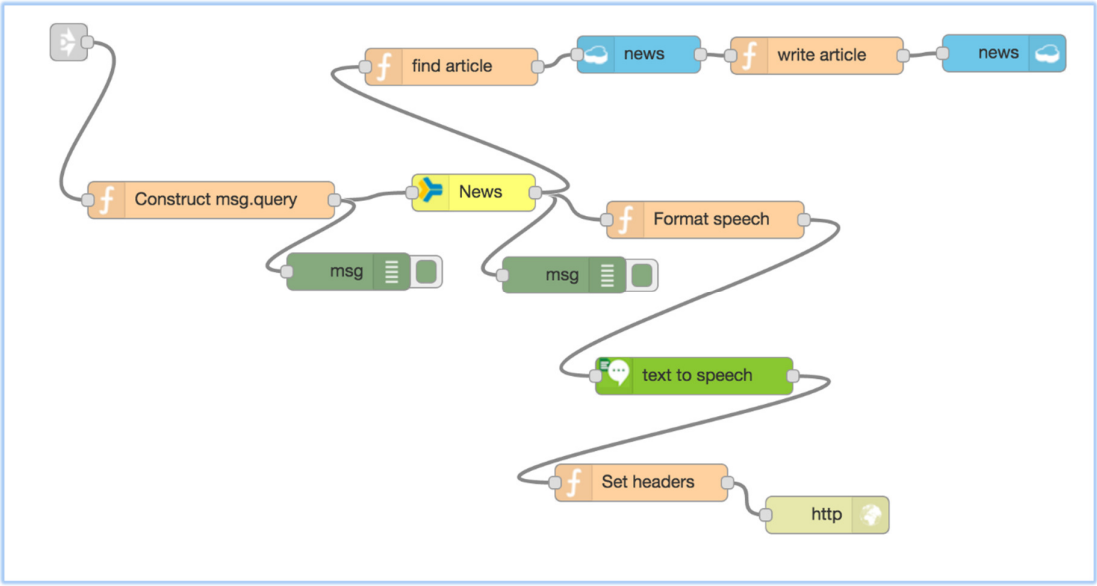
Weather Application flow



Explanation: This flow is triggered when the user asks for the weather. The first thing that needs to be determined is the precise location that is being requested. The first node in this flow, "Construct URL", is a function node that gets ready to call the Google maps API. Open it up (double-click) and you will see some javascript that first parses the incoming text and scrapes the word(s) after "in" or "for" - this is assumed to be the location requested. A string URL is created and passed as msg.url that will ask Google for more information about that location, including the latitude and longitude coordinates. That URL is then fed into the http request node which returns the response from google. The response is a string manifestation of JSON, so the next node will convert that string into parseable JSON.

Next we have some error checking. The "Error check google API results" checks if zero results are returned from Google -- if so, a response is generated for the assistant to say that the location is not recognized. But if results are returned, then a function node parses the JSON to extract the latitude and longitude coordinates. Then the text transcription is parsed further to determine if "current" or "forecast" is requested, and that goes to the appropriate weather node which is pre-configured to return either the current weather or the forecast. The JSON response from the weather node (which calls the weather service) is then parsed and wrapped inside an English response which then goes to the "text to speech" node which returns a wav in the payload. Finally an audio html tag is added to the response and sent back with the wav as the http response, which allows the browser client to play the audio.

News Application flow



Explanation: This flow gets triggered when the user asks for news. In "construct msg.query", incoming text is parsed to build a query to pass to the Alchemy News service via the News node. Anything after "about" is set as the title search field, and if you want news articles with "positive" or "negative" sentiment, that is configured as well.

More options passed to the News node include: `start=now-1d&end=now` which requests news articles from the last 24 hours and `rank=high` which requests popular articles from reputable sources. Debug nodes are placed before and after the call to the News node so you can see the input and outputs to the News service in the Debug panel.

The response received from the News node is parsed and formatted and sent to the text-to-speech node and then returned as audio in the http response.

You will also notice another wire flowing out of the News node and going to Cloudant. That flow writes the text from the news article and persists it to a Cloudant database incase the says "Read it!".

If you so wish, you can try to modify the query parameters passed into the news node, for example, you could change `start=node-1d` to `start=node-1h` for fresher news updates. Or you could play with the sentiment thresholds. Or there are many other options you could use to filter and search for articles, the API documentation is located at this link:

http://docs.alchemyapi.com/docs

I got everything working, now what ?

This is your application to modify and extend as you wish! We recommend that you experiment and start with small changes. For example, toggle the voice from male to female, or modify the text responses that are returned. Later maybe you could add a whole new category of question with a new flow and even hook it up to other Bluemix services. Your imagination is the limit!

4 Lab 4: Expanding the Application for Long Tail using Retrieve and Rank

4.1 Answer Retrieval

This repository contains a **Starter Kit** (SK) that is designed to show you how to create your own answer retrieval application for StackExchange, using the Retrieve and Rank (R&R) service, a cognitive API from the Watson Developer Cloud. Information retrieval applications enable users to search for content in specific information sources. Creating an answer retrieval system has historically been a very complex technique requiring lots of configuration and lots of expert tuning. This starter kit uses the Retrieve and Rank API to support the entire process of creating such a system, from uploading your data to evaluating results, including training your answer retrieval system.

Note:

Only after completing the steps defined below in table of contents, you will be able to deploy the application to Bluemix using the button below:

4.2 Table of Contents

4.3 How this app works

This starter kit uses Jupyter Notebook, a web application that allows you to create and share documents that contain code, visualizations, and explanatory text. (Jupyter Notebook was formely known as iPython Notebook)

Jupyter Notebook automatically executes specific sections of Python code that are embedded in a notebook, displaying the results of those commands in a highlighted section below each code block. The Jupyter notebooks in this SK show you the process of building a custom ranker for the data on Travel Stack Exchange.

This SK has three primary components:

Two Jupyter notebooks, which show you the process of building an answer retrieval system using the Watson Retrieve and Rank service.

These notebooks are:

- **Answer Retrieval** which shows how to create a basic SOLR collection and enhance it with a ranker
- **Custom Scorer**, which shows how to add custom features to your ranker Once you have completed the notebooks, you can launch a simple application that shows how the answer retrieval system performs. Specifically, the application compares SOLR, a common open-source Information Retrieval system, with Retrieve and Rank's ranker, which reranks search results to be in an order more salient to a particular user. It also shows how you can extend the basic ranker with custom features that consider domain-specific information.
- **Bash shell scripts** that enable you to train a ranker on data from any StackExchange question and answer site.
- **Python code** to help you extract other content from StackExchange and pre-process it for use with the Retrieve and Rank service.

Once you complete the notebooks and understand the format of data expected by the Retrieve and Rank API, you should be able to train Retrieve and Rank on any dataset.

4.4 Getting started

Before diving into the Jupyter notebooks, you should make sure you have all the prerequisites installed, and are familiar with the directory structure of the git repository that contains this SK.

4.4.1 Prerequisites

Ensure that you have the following prerequisites installed before using this SK:

- A Unix-based OS (or Cygwin)
- Git (see section 0 in the lab guide for installation instructions)
- Node.js (see section 0 in the lab guide for installation instructions)
- python (download version 2.7)
- Anaconda (download support for version 2.7)
  Installing this package also installs the Jupyter notebook package, which includes `iPython` (now referred to as `jupyter`)
- A bluemix account
- An instance of the Retrieve and Rank service

If you are using a Linux system, the git, anaconda, python, and node.js packages may be installable through your system's package manager.

4.4.2 Checking out the repository for this SK

Use `git` to clone the repository for this SK to your local machine. For example, using a command-line version of git, the command that you would execute is the following:

```
git clone git@github.com:watson-developer-cloud/answer-retrieval.git
```

4.4.3 Directory Structure of the repository

The directory that you created when cloning the git repository for this SK contains the following subdirectories:

`bin\` contains various bash and python scripts for interacting with the R&R API

`config\` contains a configuration that tells SOLR how the StackExchange data is structured.

`custom-scorer\` contains the code necessary to train scorers for R&R that use custom features

`data\` contains sample StackExchange data that is pre-processed for use by the Retrieve and Rank service. This data will be automatically uploaded to the Retrieve and Rank service as part of the Python code in the **Config** section of the **Answer Retrieval** notebook.

`notebooks\` contains the iPython notebooks

`static\` contains the static website assets, css, js, html

4.4.4 Installing dependencies for the application

29. Install the dependencies using pip:

```
pip install -r requirements.txt
pip install -r notebooks/requirements.txt
```

30. Create a `.env` using `.env.example` as example. You will need credentials for the Retrieve and Rank service.

31. Start the application:

```
python server.py
```

4.5 Running the notebooks

The Jupyter notebooks show you the process of creating an information retrieval system, step-by-step, automatically executing specified sections of Python code. We used Jupyter notebooks

because they encourage experimentation, which is an important part of developing any machine learning system.

You will need credentials in order to use R&R. These can be obtained after creating an account in (Bluemix)(http://bluemix.net) and creating an instance of the service in account. After these are done, you can click the "Service Credentials" entry in the left-hand navigation for that service in Bluemix to see your R&R Credentials.

Before starting the notebook, please add the username and password from the credentials for the instance of the Retrieve and Rank service that to created to the json file `credentials.json`. This file is located in the `config` directory of this SK's repository. This enables the notebooks to use these values throughout all of the code blocks in the notebook.

To start the notebooks, make sure you are in the root directory of your `git` checkout of the SK repository, and execute the command `jupyter notebook`. This will start the Jupyter notebook server, and open a browser window. Once the browser window is open, click on `notebooks`, and then open the notebook labeled `Answer- Retrieval.ipynb`. Follow the instructions in there to create your own ranker.

4.5.1 Using the Sample Data

The **Populate the Collection** section of the **Answer Retrieval** notebook loads sample data into the Solr collection that was created by previous code blocks in that Notebook. This sample data is located in the `config` directory of the repository for this SK.

4.5.2 Exploring with the UI

Important: The Sample Application UI will not have any rankers to display results for until you have stepped through the iPython notebooks.

Now that you have completed the iPython notebooks, you have 2 ways to search and compare your results of your experiments: basic Solr versus default ranker and basic Solr versus a ranker with custom scoring features. If you want to explore how these different rankers perform, you just have to modify a few things in your local environment properties (.env file).

- For using the UI with the default ranker, modify the SHOW_DEFAULT_RANKER property to "TRUE" and set RANKER_ID property to the default ranker id created at the end of the Answer Retrieval Notebook.
- For using the UI with the custom ranker, modify the SHOW_DEFAULT_RANKER property to "FALSE" and set RANKER_ID property to the custom ranker id created at the end of the Custom Scorer Notebook.

4.6 Using your own data

If you want to train rankers with data from other StackExchange sites, you first need to download the dumps. Once you have chosen a dump, you can use `bin/python/extract_stackexchange_dump.py` to convert it into a R&R-compatible

format. If you wish to use another data source, consult the Retrieve and Rank documentationn, which explains how R&R expects incoming data to be formatted.

4.7 Improving Relevance

It is often necessary to look for additional features in your dataset that can be used to provide information to the ranker that can instruct it on how to identify results that are more relevant to others. These are implemented as a set of custom features known as custom scorers. In the case of the Stack Exchange community support scenario that was used to collect the sample adata for this SK, you have access to metadata about each answer. Examples of this sort of metadata include the following:

- **User Reputation**: a rough measurement of how much the community trusts an expert author. Community users gain reputation points when a question that they have asked is voted up, an answer that they have made is voted up, an answer that they have made is accepted, and other criteria.
- **UpVotes** : the number of times that someone other than the expert has accepted an expert's answer as a pertinent response.
- **DownVotes**: the number of times that someone other than the expert has rejected an expert's answer as a pertinent response.
- **Number of Views**: the overall popularity of the related topic/question
- **Answer Accepted**: a Boolean that identifies whether the answer provided by the expert was accepted by the original author

Metadata such as the items in the preceding list can be used to create custom features that provide information to the ranker which it can use to enhance learning about the problem domain. If the metadata has a strong correlation to predicting relevance, you should see improvements to overall relevance metrics.

The custom features that you can create for a Retrieve and Rank implementation typically fall into 3 categories:

- Document
- Query
- Query and Document

You can write custom scorers using any of the following:

- DocumentScorer - A document scorer is a class whose input to the score method is a field or fields for a single Solr document. Consider a class called DocumentViews, which creates a score based on the number of views for a given topics and is represented a field in the Solr document.
- QueryScorer - A query scorer is a class whose input to the score method is a set of query params for a Solr query. Consider a class called IsQueryOnTopicScorer, which scores queries based on whether it thinks the underlying query text is on topic for the application domain.
- QueryDocumentScorer - A query-document scorer is a class whose input to the score method is 1) a set of query params for a Solr query, and 2) a field or fields for a Solr

document. Consider a class that scores the extent to which the "text" of a Solr document answers definitional questions. More specifically, the scorer will 1) identify if a query is asking for a definition and 2) if so, identify whether the document contains a likely definition or not.

The custom scorer notebook provided here as part of this starter kit provides access to a custom scoring framework allowing you to extract new features for the purposes of training a ranker.

4.8 Using Retrieve and Rank Custom Scorers

4.8.1 Description

This project enables the usage of custom features within the Retrieve & Rank service on Bluemix. This project was built in the Python programming language and uses the Flask micro-framework. To use this application, there is a Python script called `server.py`, which exposes two endpoints to be consumed by the application that uses it.

4.8.2 Application Architecture

The Flask server that is created within the script `server.py` is intended to run as a "sidecar" within the rest of the application; that is, the parent application will make REST API calls to this "sidecar" service rather than making direct calls to the deployed Retrieve & Rank service. The principal difference is that the Flask server will handle the integration/injection of custom features that have been registered.

Steps to get set up

There are 4 steps to set up the server to integrate custom features:

- Configure your environment. The python flask server is already setup to run and extract features from custom scorers. The custom scorers are packaged as wheel package and need to be installed whenever a new custom scorerer is created.
- Identify the custom scorers for your application. A custom scorer is a Python class that extracts a signal that is to be used for the ranker. There are 3 types of scorers supported:
    - "QueryDocument" Scorer - Extracts a signal/score based on both the contents of a query and the contents of a Solr Document
    - "Document" Scorer - Extracts a signal based on the contents of the Solr Document alone
    - "Query" Scorer - Extracts a signal based on the contents of the query alone

To make sure that the server has the most up to date scorers, go to the 'Custom Scorer' notebook and follow instructions to build and install the wheels package.

- Create a configuration file (see `config/features.json` as an example) to configure your application to consume these scorers. The configuration file must be a json file and must contain a "scorers" field, which is a list of individual scorer configurations. For each of the scorers that you identified in the previous step, the scorer configuration is a JSON object that must define the following:

- an 'init_args' json object, whose fields are the arguments to the constructor for the scorer
- a 'type' field, which should be either 'query', 'document' or 'query_document', depending on the type of scorer. The type is used to identify the package within the 'retrieve_and_rank_scorer' project that contains the appropriate scorer
- a 'module' field, which is the name of the python module which contains the scorer
- a 'class' field, which is the name of the scorer class For comparison, the `config/features.json` contains a single Document scorer, in the module document, with the class UpVoteScorer. This is to extract feature based on the positive votes that a post has received.

- Start the Flask server by running the command

```
python server.py
```

4.9 Privacy Notice

Sample web applications that include the cf_deployment_tracker package included here may be configured to track deployments to IBM Bluemix and other Cloud Foundry platforms. The following information is sent to a Deployment Trackerservice on each deployment:

- Python package version
- Python repository URL
- Application Name (application_name)
- Space ID (space_id)
- Application Version (application_version)
- Application URIs (application_uris)
- Labels of bound services
- Number of instances for each bound service and associated plan information

This data is collected from the `server.py` file in the sample application and the `VCAP_APPLICATION` and `VCAP_SERVICES`environment variables in IBM Bluemix and other Cloud Foundry platforms. This data is used by IBM to track metrics around deployments of sample applications to IBM Bluemix to measure the usefulness of our examples, so that we can continuously improve the content we offer to you. Only deployments of sample applications that include code to ping the Deployment Tracker service will be tracked.

5 R&R Separate Lab

THIS LAB IS INCOMPLETE AND UNTESTED AT THIS POINT (Nov 13 2016)

Source: node-red-labs/basic_examples/retrieve_and_rank/README.md

5.1 Overview

The IBM Watson™ Retrieve and Rank service combines two information retrieval components in a single service: the power of Apache Solr and a sophisticated machine learning capability. This combination provides users with more relevant results by automatically reranking them by using these machine learning algorithms.

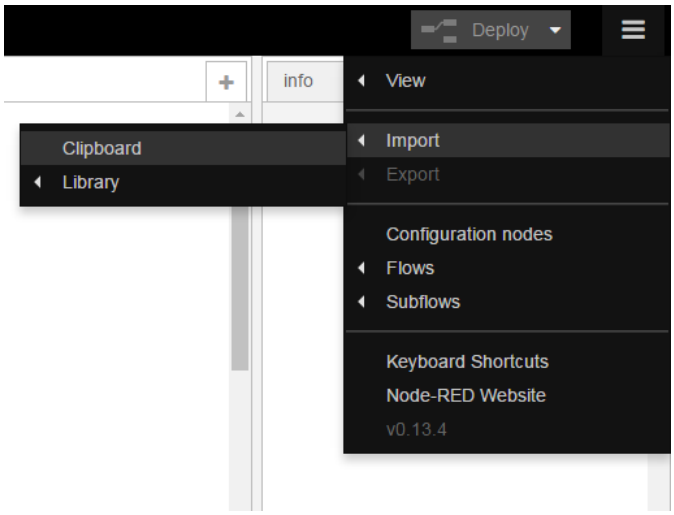
5.2 Installation of the Retrieve and Rank Nodes

Follow the instructions at watson contribution nodes to install the retrieve and rank nodes into your Bluemix instance of Node-RED.

5.3 Lab 1: Retrieve and Rank using the Cranfield data collection

To complete this tutorial, you use the publicly available test data that is called the Cranfield collection. The collection contains abstracts of aerodynamics journal articles, a set of questions about aerodynamics, and indicators of how relevant an article is to a question. To use your own data, please refer to the documentation.

This lab will use a given flow contained in this directory. Copy the contents of `lab.json` to the clipboard. In the Node-RED flow editor, import the flow as follows:

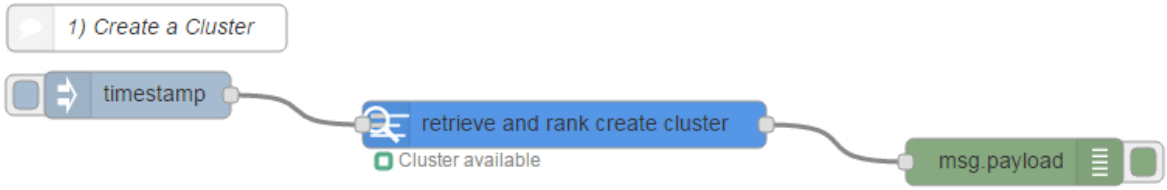


and paste in the contents of `lab.json`. Once imported, click "Deploy" in the top right corner. This flow sets up all the configuration nodes and searching facilities that are present in the Cranfield sample tutorial.

Before you proceed, make sure you have the Dropbox node on your palette and have configured it correctly. These instructions explain this step in more detail.

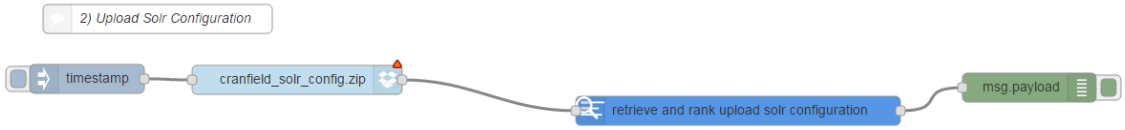
5.3.1 Create a cluster

The following set of nodes create a cluster within the rank and retrieve service.



Double click the node to select the size of the cluster and add a cluster name. For a simple cluster for testing, choose the "Free" option. Note that you can only create one free cluster per service. The cluster will take a minute or so to prepare and when it is available the status of the node will be updated.

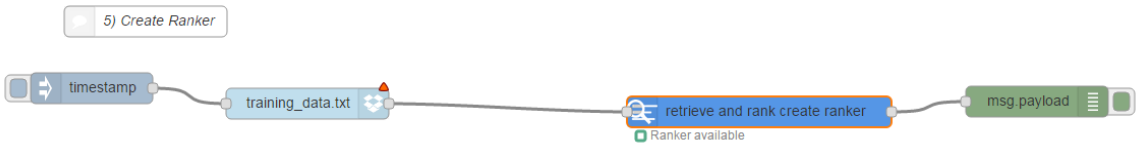
1. Upload Solr Configuration. The following set of nodes upload a .zip file for the given cluster consisting of the solr configuration. Double click the node to enter in the cluster_id given in the output from step 1) and a name for the configuration.



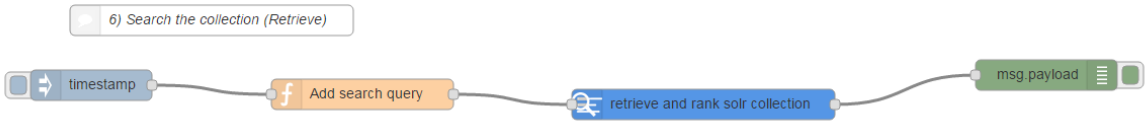
2. **Create Solr Collection**. The following set of nodes create a Solr Collection for the given cluster. Double click the node to enter in the `cluster_id` given in the output from step 1) and the configuration name specified in step 2).

3. **Index Documents**. The following set of nodes upload documents to the given collection for indexing. Double click the node to enter in the `cluster_id` given in the output from step 1) and the collection name specified in step 3).

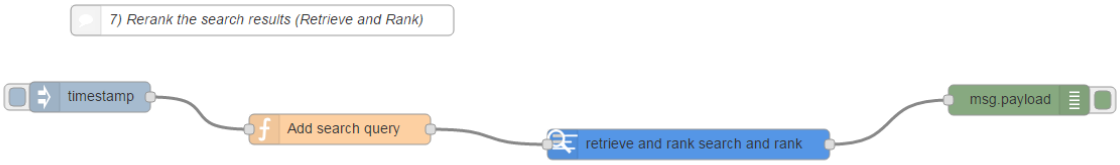
4. Create Ranker. The following set of nodes create a ranker to be used for ranking the documents. Double click the node to enter a name for the ranker. The ranker will take approximately 5 minutes to train. The status of the node will be updated once it is available.



5. Search the collection. Whilst you wait for the ranker to become available, you can perform queries on the collection (Retrieve). The following set of nodes perform this with a query string passed in on msg.payload.



6. **Rerank the search results**. Once the ranker is available, you can perform Retrieve queries on the collection to rank the documents. The following set of nodes perform this with a query string passed in on `msg.payload`. The `ranker_id` is specified in the node configuration panel.



6 Lab 5 – Use Alchemy Language to Expand Entities

Build Your Chatbot with Watson Conversation with Expanded Entities from Watson Alchemy Language

This section describes the commands and steps you need to execute to get the application up and running with Alchemy Language, Conversation, and Weather services. Because you have completed all steps in detail during previous exercises, the first section (5.1) covers this lab a s a quick guide.

6.1 Overview

In this lab, you will focus on the Watson Conversation service which offers developers a turn-key service for building bots and virtual agents. They have become very popular in various applications and platforms. You will also learn how to integrate the Watson Conversation service with another WDC service, Alchemy Language, to expand the list of entities that could be extracted from users' utterances.

6.2 Verifying Prerequisites

In this section, you will set up and run a simple app that leverages Watson Conversation. Before you start this section, please run the following commands to verify you have the required packages installed on your machine.

1. Open a Terminal window
2. Check that git is installed by typing `git --version`

   If this returns an error, then git is not installed. Please download and install git at https://git-scm.com/download/win
3. Check that node.js is installed by typing `node -v`

   If this returns an error, then node is not installed. Please download and install Node from nodejs.org
4. npm should be installed with nodejs, verify it is installed on your machine by typing `npm -v`
5. Check cf (cloud foundry command line interface) is installed by typing `cf -v`

   If this returns an error, then cf is not installed. Please download and install cf from the Bluemix application window

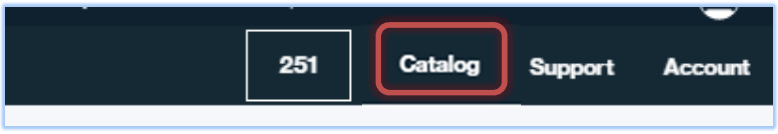
```
C:\LabFiles\Lab5>git --version
git version 2.10.2.windows.1

C:\LabFiles\Lab5>node -v
v6.9.1

C:\LabFiles\Lab5>npm -v
3.10.8

C:\LabFiles\Lab5>cf -v
cf version 6.22.2+a95e24c-2016-10-27
```

6.3 Creating a new Application

At this point, you should have a Bluemix account and be familiar with working with different services. If you need to perform any of the prerequisite work, review the prerequisites section at the beginning of this guide.

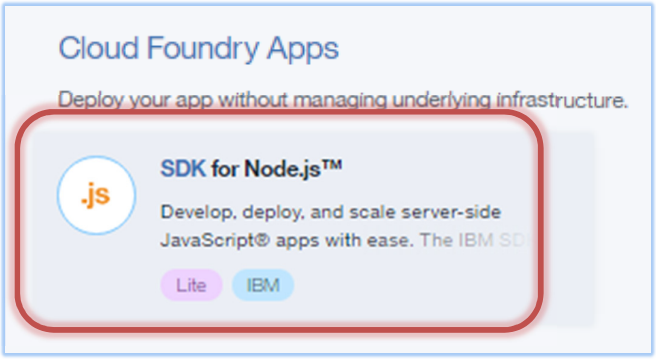
Bluemix offers a rich set of runtimes including Java, Swift, PHP, Ruby. In this lab you will focus on leveraging Node.js.

1. Log into Bluemix using the credentials you signed up with

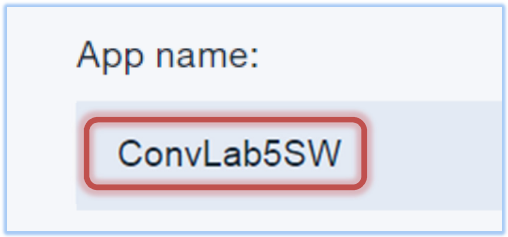


2. Click **Catalog** link from the dashboard

3. Scroll down through the catalog to get the Cloud Foundry Runtimes until you find **SDK for Node.js**



4. Click the **SDK for Node.js** runtime

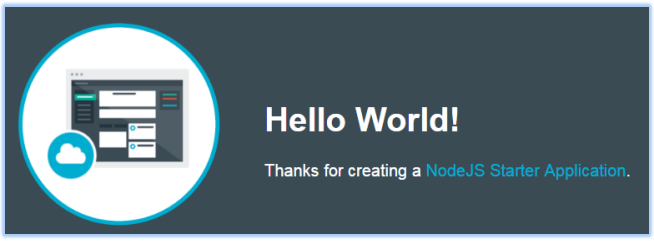


5. Provide a unique App name as shown in and click the **Create** button

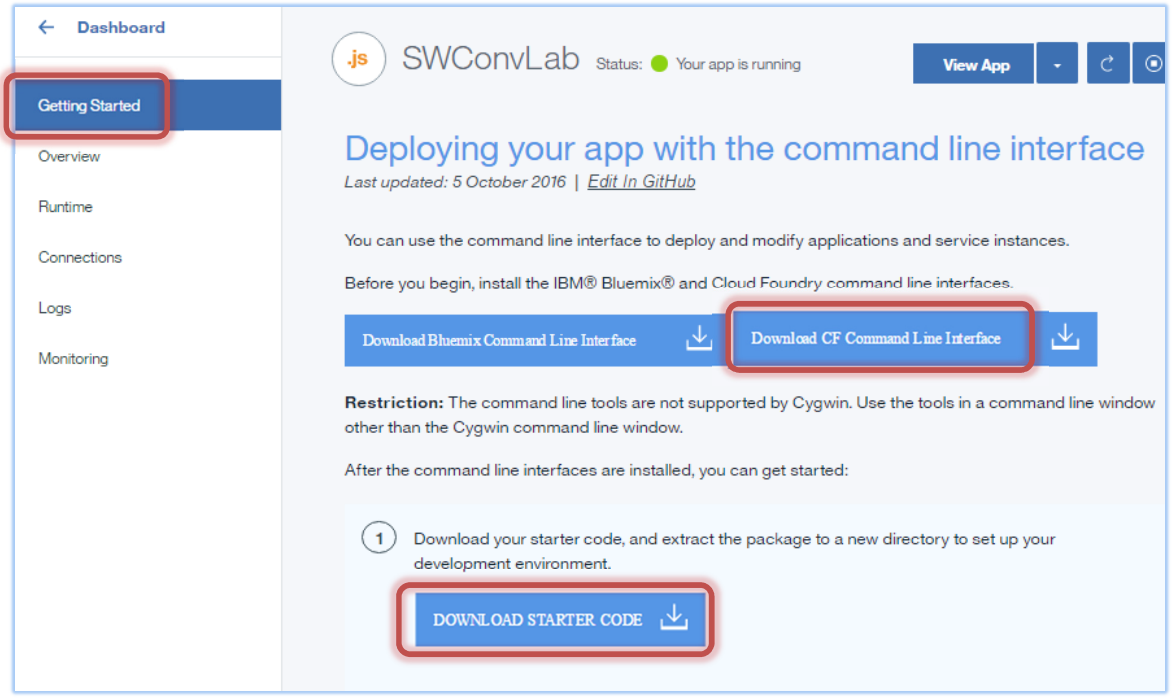
Wait until the application is started:

At this point, you have a create a simple Hello World Node.js app and are hosting it on your bluemix account.

6. To view your running app, point your preferred browser to http://<appname>.mybluemix.net (where <appname> is your app name): (http://**convlab5sw**.mybluemix.net/)



7. Go back to your Application and if necessary, select the **Getting Started** link in the top left corner of the window



As you can see, this is another link to the cf CLI download: **Download CF Command Line Interface**.

The page also contains a starter code through **Download Starter Code** to download code that you can edit and modify locally and then push back to Bluemix.

To do so, follow the instructions on the page to unzip the downloaded code, make changes and then push it back to Bluemix.

6.4 Downloading Dashboard Application

Now that you verified that all required tools are installed, you will set up and run the cognitive car dashboard sample application which is a simple application, very well suited for quickly getting started with Watson Conversation.

Start by downloading the code for the cognitive car dashboard application.

On your terminal window, run the following commands:

8. Enter `mkdir lab5`
9. Enter `cd lab5`
10. Enter `git clone https://github.com/watson-developer-cloud/conversation-simple.git/`
11. Enter `cd conversation-simple`
12. Enter `npm install` → installs node packages defined in package.json

13. Enter `copy .env.example .env` → You define service credentials in .env file

14. Open the .env file in notepad++

After creating your service, copy and paste the credentials for Conversation service

Define a Watson Conversation service and create a conversation flow that we can use in this application. Note that we need 3 parameters as defined in .env file, WORKSPACE_ID, CONVERSATION_USERNAME, and CONVERSATION_PASSWORD.

For the purposes of this lab, you can ignore the other parameters in .env

The CONVERSATION_USERNAME and CONVERSATION_PASSWORD are defined when a new conversation service is created. The WORKSPACE_ID, on the other hand, is defined when a conversation is built. To build a conversation using the Watson Conversation service, you need to define intents, entities, and the dialog flow to orchestrate the conversation based on extracted intents and entities from users' utterances as well as the context of the conversation. For details, please consult the official documentation of Watson Conversation service on Watson Developer Cloud site. You can also refer to this blog post on Watson Conversation with Alchemy Entity Extraction.

Next you create a Watson Conversation service and learn how to obtain the WORKSPACE_ID after defining a conversation.

6.5 Creating Watson Conversation service instance

(you can skip this if peformed in previous lab)

15. Log into Bluemix using your Bluemix credentials

16. Navigate to **Catalog**

17. Click **Watson** in the left navigation window. This will display the Watson services available.

18. Select the **Conversation service**

19. Provide a unique service name (or just use the default service name), **leave unbound** to keep the service unbound, and choose plan **Free** (this allows for 1000 api calls per month).

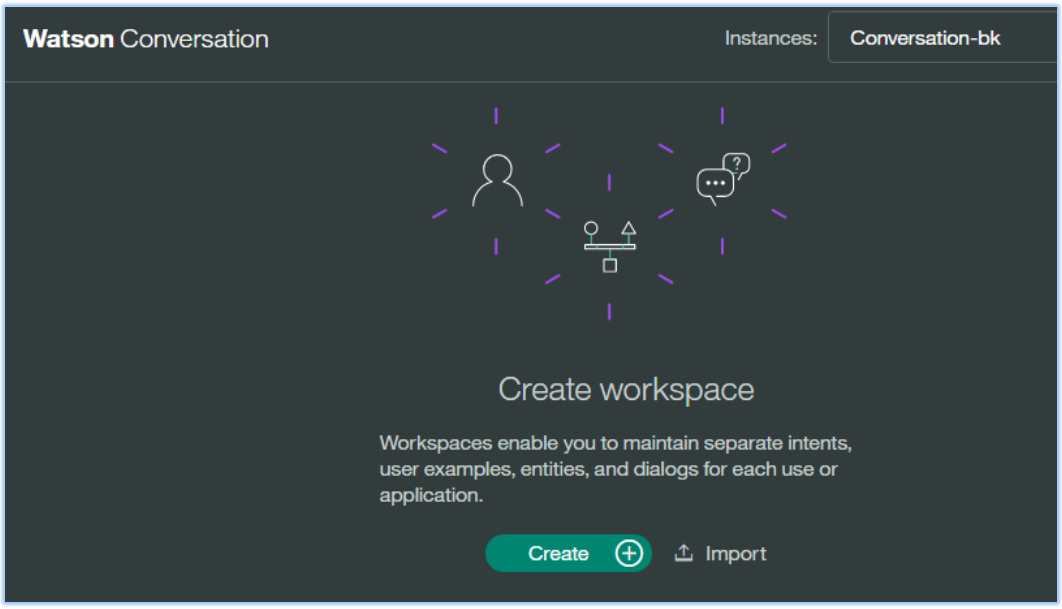
20. Click **Create**

    This creates a new instance of Watson Conversation service and provides credentials

6.5.1 Loading an existing workspace

(you can skip this if peformed in previous lab)

21. Click the **Launch tool** button in your **Manage** tab to start the conversation interface where you can define intents, entities, and the dialog for conversation.



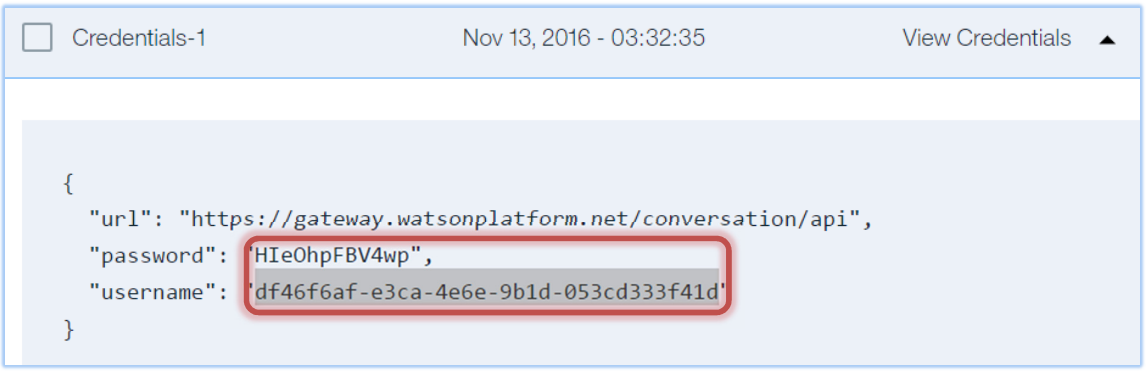
If no workspace has been defined, you can **Create** a new conversation and define intents, entities, and dialog, or **Import** a complete workspace.

22. Import the **car_workspace.json** workspace by clicking ***Import***

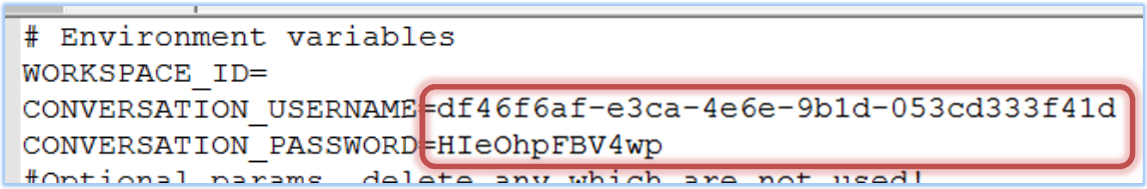
23. Choose the file, navigate to the directory where you cloned the github repository \Lab5\conversation-simple\training\car_workspace.json, and click **Import**

6.6 Entering Service Credentials in Application .env

24. Access the credentials by clicking **Service Credentials** tab



25. Copy the credentials (username and password) into the .env file:

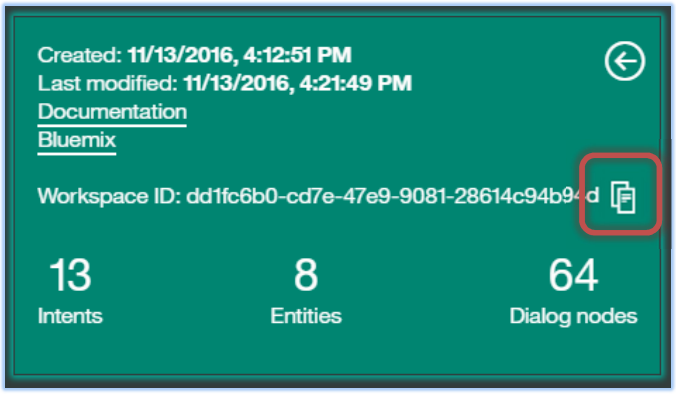


Now obtain the WORKSPACE_ID to populate it in your .env file.

26. Navigate back to your workspace (Manage >> Launch Tool)

27. Click the three dots on the workspace to open the menu and select **View details**

    This loads the details of the workspace including the workspace id

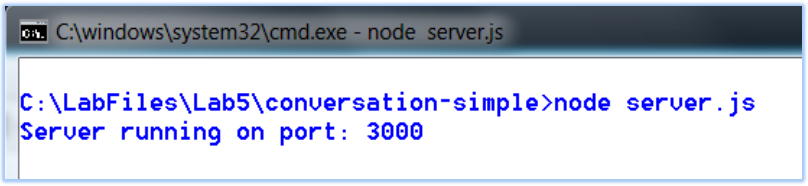


28. Copy the workspace id and set the variable in .env file of the conversation_simple application. WORKSPACE_ID = **workspace ID**

29. Save your .env file

6.7 Running the Application Locally

30. Run the application locally by typing `node server.js` from your terminal window



31. Point your browser to **localhost:3000** and you can interact with the app communicating with the cognitive car dashboard

6.8 Uploading the Application to Bluemix

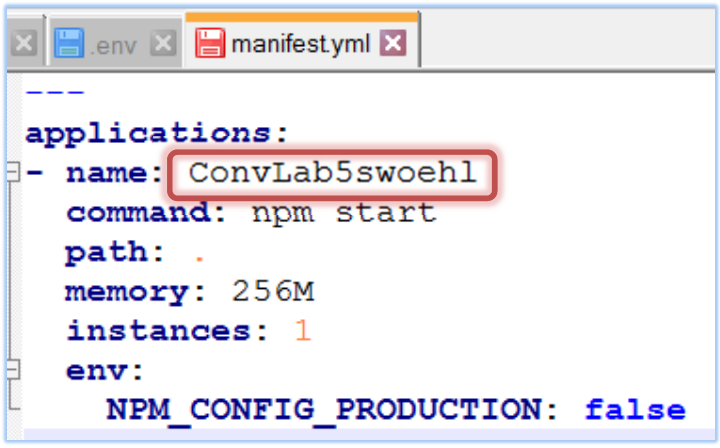
32. Now you can host the app on Bluemix. To do so, edit **manifest.yml** in the app's home directory

33. Delete the following lines:

```
declared-services:
      conversation-simple-demo-test1:
            label: conversation
            plan: free
services:
      - conversation-simple-demo-test1
```

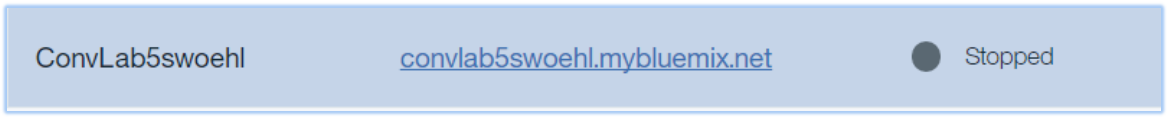
34. Change the name of the app to a unique name (use your initials as part of the name)

    `name: ConvLab5<email>` (where <email> is your email short name)



35. Push the app to bluemix by typing **`cf push`** on the command line in your terminal window

    This will take a few minutes after which you should get a message that your app is running. You can also observe progress in Bluemix.



36. Click the application and access Logs for more details

37. Point your browser to your app's hostname: http://ConvLab5swoehl.mybluemix.net and you can interact with the application

7 Integrating Alchemy Language for Entity Extraction

As you browse the conversation for the car dashboard workspace, note that there is a **#weather** intent defined to understand when the user is asking about the weather. However, the application currently doesn't offer a response. It simply says "**Unfortunately I don't know much about the weather. I'm still learning.**".

In this section, you will update the application to return weather information. To get the weather, you rely on the Weather Underground API. To use the weather underground api, you need to sign up for an apikey. Here is an example call using Weather Underground API.

http://api.wunderground.com/api/**apikey**/conditions/q/**state**/**city**.json

For example, to find weather conditions in San Francisco, CA:

http://api.wunderground.com/api/**apikey**/conditions/q/CA/San_Francisco.json

In addition to the weather underground apikey, note that you also need to pass in the city and state for the weather api to return the weather forecast for that city/state. To get the city/state from text the user inputs in their conversation, you have 2 options:

- Define two entities, city and state, in conversation and provide a list of all possible cities and states.
- Design your application to manage the extraction of relevant entities and pass those as context variables to the conversation service.

While option 1 is a possibility, it is clear that option 2 would be much easier and more scalable. Furthermore, while it is possible to identify and list all cities and states (for option 1), it may **not** be possible to always list all possible variations of a certain entity of interest in other applications (for example, organization). During the remainder of this lab, we will focus on option 2; specifically, leveraging Watson Alchemy Language service which is already pre-trained to extract 42 different entity types including cities and states.

7.1 Creating Watson Alchemy Language Service Instance

To create Alchemy Language service instance, repeat the same instructions you executed for Watson Conversation.

1. To create an Alchemy Language service, select **AlchemyAPI** from the Watson services catalog in Bluemix

2. When created, access the service credentials

```
{
  "url": "https://gateway-a.watsonplatform.net/calls",
  "note": "It may take up to 5 minutes for this key to become active",
  "apikey": "b7c89d3892c9d29846aed6953218c4d96098c60b"
}
```

3. Copy the **apikey** for Alchemy Language and save that for later use in the .env file for your application

7.2 Download Application

Execute the following steps to download sample code which includes the integration with the Alchemy Language service.

1. In your terminal, enter `mkdir lab6`
2. Enter `cd lab6`
3. Enter `git clone https://github.com/joe4k/conversation-simple.git`
4. Type `cd conversation-simple`
5. Enter `npm install` → installs node packages defined in package.json
6. Enter `copy .env.example .env` → Define service credentials in .env file
7. Edit the .env file and copy/paste the credentials for Conversation service that you obtained when you created the service.
8. Also copy/paste the credentials for Alchemy Language service in .env file.

   Your .env file should look something like this:

```
# Environment variables
WORKSPACE_ID=your_conversation_workspaceID
CONVERSATION_USERNAME=your_conversation_username
CONVERSATION_PASSWORD=your_conversation_password
ALCHEMY_LANGUAGE=your_alchemy_apikey ← insert apikey here
```

Browse the code briefly and note how the Alchemy Language service wrapper was created and how Alchemy Language service is called to extract ***City*** entity and defining it as a context variable for conversation service ***payload.context.appCity***.

```
57  // Create service wrapper for AlchemyAPI
58  var alchemy_language = watson.alchemy_language({
59    api_key: process.env.ALCHEMY_API_KEY || 'YOUR_API_KEY'
60  });
```

```
alchemy_language.entities(params, function(error, response) {
      if (error) {
        return res.status(error.code || 500).json(error);
      }
      if(response != null) {
        var entities = response.entities;
        var cityList = entities.map(function(entry) {
              if(entry.type == "City") {
               return(entry.text);
              }
        });
     cityList = cityList.filter(function(entry) {
       if(entry != null) {
        return(entry);
       }
     });
     if(cityList.length > 0) {
      payload.context.appCity = cityList[0];
     } else {
      delete payload.context.appCity;
     }
        var stateList = entities.map(function(entry) {
              if(entry.type == "StateOrCounty") {
               return(entry.text);
              }
        });

     stateList = stateList.filter(function(entry) {
       if(entry != null) {
        return(entry);
       }
     });
```

Next, you need to update our conversation flow to check if the context variable appCity is defined and if it is, return the weather for the extracted city and state.

Hint: If you have any difficulty in modifying conversation flow(next steps), you can import the ***car_workspace_alchemy*** workspace from the code you cloned from https://github.com/joe4k/conversation-simple.git

In your conversation service, ***Import*** workspace and choose file ***lab6/conversation-simple/training/car_workspace_alchemy.json***
NOTE: If you do import car_workspace_alchemy.json into a new workspace, you will need to edit the .env file to point to the new WORKSPACE_ID

To do so, in your conversation workspace, go the node where #weather intent is defined and add a child node (to its right) to check if $appCity and $appST are defined.

Note that these city/state entities are extracted by the application using Alchemy Language service and passed to your conversation service as context variables. If defined, add a response to the user

indicating you can provide the weather for given city/state. If not, you can add another node ***$nprompt*** to prompt the user to input city where they'd like the weather information.

Lastly you can add a ***true*** node for a default response if user doesn't provide valid city.

You are now ready to run the application and serve the weather conditions for the given city/state.

9. Run the application locally by typing `node server.js` on the terminal window.

10. Point your browser to **localhost:3000** and you can interact with the app communicating with the cognitive care dashboard.

11. As a last step, host the app on Bluemix. To do so, edit the **manifest.yml** file in the app's home directory. Change the name of the app to a unique name (use your initials as part of the name)

    name: ConvLab6<email>

```
applications:
- name: ConvLab6swoehl
  command: npm start
  path: .
  memory: 256M
  instances: 1
  env:
    NPM_CONFIG_PRODUCTION: false
```

12. Push the app to Bluemix through the `cf push` command
    This will take a few minutes after which you should get a message that your app is running.

```
details
#0   running   2016-11-13 07:12:36 PM   33.7%   121.5M of 256M   235.1M of 1G
```

13. Point your browser to your app's hostname: http://ConvLab6swoehl.mybluemix.net and you can interact with the application.

7.3 Conclusion

In this lab exercise, you integrated Watson Alchemy Language service with Watson Conversation service to extract entities of interest from users' input text (in this case, city and state) and pass that information to the conversation service as context variables so Conversation service can take the required action.

This should illustrate the flexibility of incorporating multiple service to implement your ideas.

We look forward to see what you will come up with on your own!

8 2-page Instructions for Lab 5 and 6 (for the impatient among you ☺)

Open a terminal window on your laptop and execute the following commands:

```
mkdir lab7
cd lab7
git clone https://github.com/joe4k/conversation-simple.git
cd conversation-simple
```

`npm install` → Installs node packages defined in package.json

`cp .env.example .env` (the command is `copy .env.example .env` in Windows)
→ we define service credentials in .env file

14. edit .env file to copy and paste the credentials for Alchemy Language, Conversation service and Weather API (you will create these next).

15. To create Alchemy Language and Conversation service credentials, execute the commands:

    `cf login` connects you to your bluemix account

    - API endpoint: https://api.ng.bluemix.net
    - username: your_bluemix_username
    - password: your_bluemix_password

    ```
    cf create-service conversation free lab7-conv-service
    ```

    - create conversation using free plan and call it wowlab-conv-service

    ```
    cf create-service-key lab7-conv-service svcKey
    ```

    ```
    cf service-key lab7-conv-service svcKey
    ```

    - returns username and password credentials for conversation service

16. Copy username and password to the .env file

    - CONVERSATION_USERNAME=username
    - CONVERSATION_PASSWORD=password

17. cf create-service alchemy_api free wowlab-alchemy-service

    - create alchemy language service using free plan and call it wowlab-alchemy-service

18. cf create-service-key wowlab-alchemy-service svcKey

19. cf service-key wowlab-alchemy-service svcKey

    - returns apikey for AlchemyLanguage service

20. Copy apikey to the .env file **ALCHEMY_API_KEY**=***apikey***

To get the weather, you use the Weather Underground API. To use the weather underground api, you need to sign up for an apikey. Once you get the key, edit .env file and copy the weather api key to .env file.

WEATHER_API_KEY=weatherapikey

The last piece of information you need is the WORKSPACE_ID. To get this, you need to create a workspace in our conversation service and build a conversation which involves defining intents, entities and building a dialog to orchestrate the interaction with the user. To do so:

21. Point your browser to http://bluemix.net
22. Log in with your Bluemix credentials
23. Find your conversation service with the name wowlab-conv-service. Click to open the page for that service.
24. Find the Launch button and click it to launch the tooling for the conversation service.
25. Click Import to import a json file which defines the conversation workspace.

    Choose file lab7/conversation-simple/training/car_workspace_alchemy.json

    This imports intents, entities, and the dialog for this conversation into a workspace called Car_Workspace_Alchemy.
26. Click the menu to View details
27. Copy Workspace ID, edit .env file and add workspace id

    ```
    WORKSPACE_ID=workspaceID
    ```

    Now you're ready to run the application. On the terminal command line.
28. Execute the command: `node server.js`
29. Point your browser to http://localhost:3000
30. Experiment with conversation application by asking questions like: "What is the weather in <city>"

Push your application to Bluemix:

31. `edit manifest.yml` and change name to a unique name (ConvLab5<email>)
32. enter `cf push`
33. Point your browser to http://ConvLab7<email>.mybluemix.net
34. Experiment with conversation application: Ask things like: "`What is the weather in Austin, TX`"

9 Car Dashboard Extension: Regular Expressions and System Entities

9.1 Introduction

This exercise builds on the car demo application. In the car scenario you are driving a cognitive enabled car. You ask in a natural way, the way you speak, for the car to do things for you, such as turn on wipers, play music and so forth.

It is strongly recommended that you watch this 14-minute video: https://youtu.be/ELwWhJGE2P8 and go through conversation service lab as a precursor to this exercise. In this exercise we will extend the car demo with an intent for "requesting maintenance" and then incorporate the dialog/entities necessary to gather the details of that maintenance request.

9.2 Pre-requisites

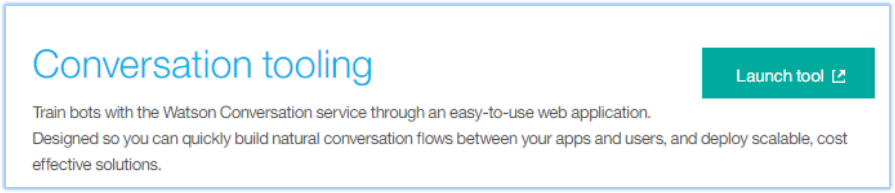
This exercise requires the following components:

- Bluemix and Watson Conversation Service
- Node runtime
  - Ensure you have node.js and npm installed.
  - Set up from https://nodejs.org/download/
- GitHub Repository
  - Clone the simple car application - https://github.com/watson-developer-cloud/conversation-simple

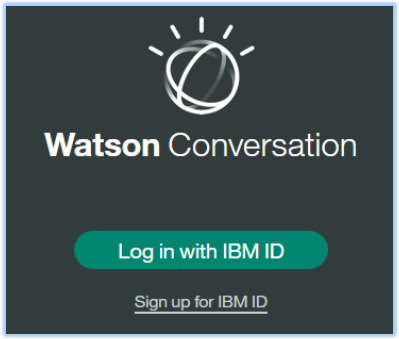
9.3 Scheduling Maintenance

To start this exercise, we will begin by importing the workspace JSON file from GitHub.

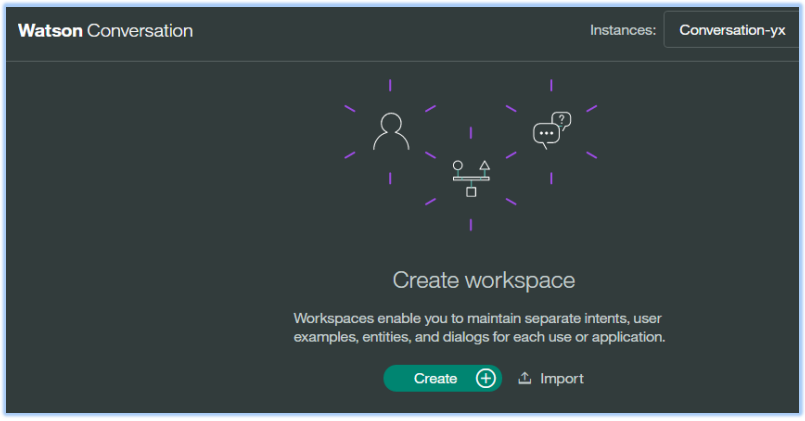
35. Go to Manage within the Conversation Service dashboard



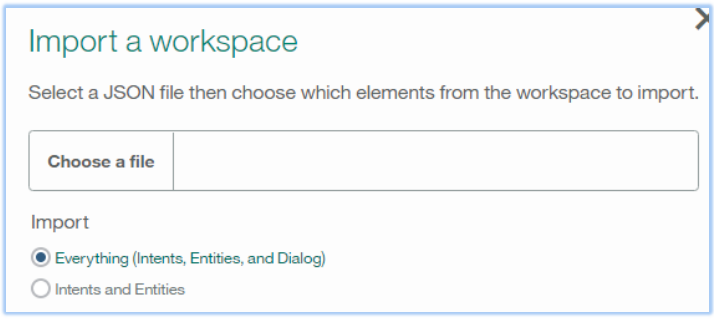
36. Click the Launch Tool button to launch the Conversation tool



37. If you are prompted to log in, click the Log in with IBM ID button. You should be logged-in automatically using the Bluemix credentials you entered earlier.



38. On the Create workspace page, select Import



39. Select Choose a file and then find/select the car_workspace.json file downloaded from the GitHub clone (under the training directory)

40. Click **Open** and then **Import**

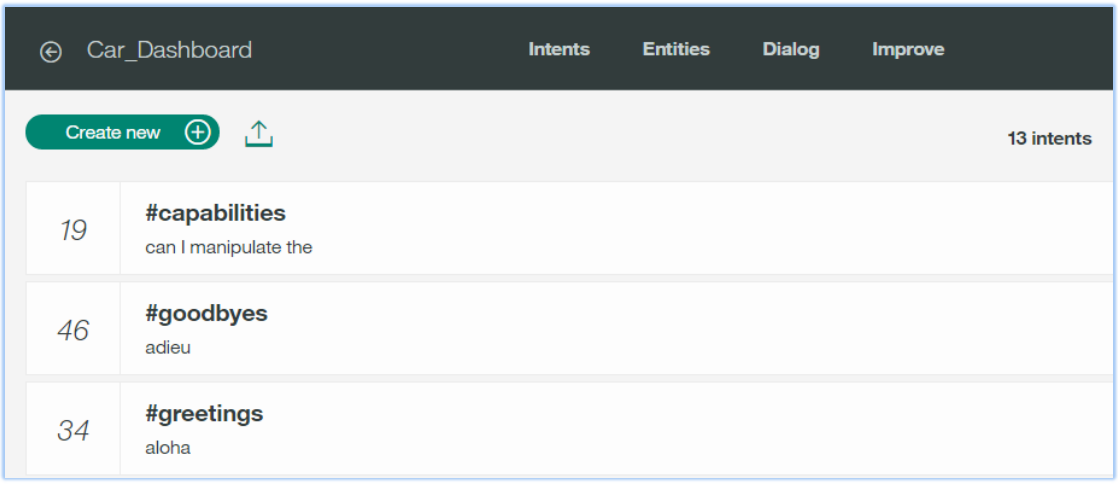
    The **Car_Dashboard** workspace will be displayed.

41. Click **Get Started** on the workspace

9.4 Create Intent

Now add the new intent to the service.

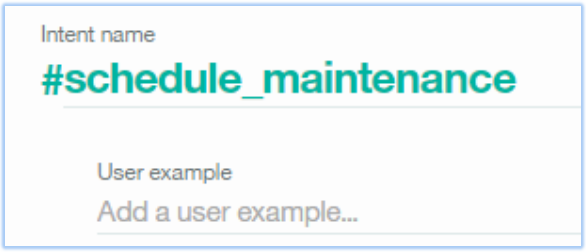
42. Start by clicking on the **Intents** tab in the workspace.



9.4.1 Add an Intent

43. Click **Create new**

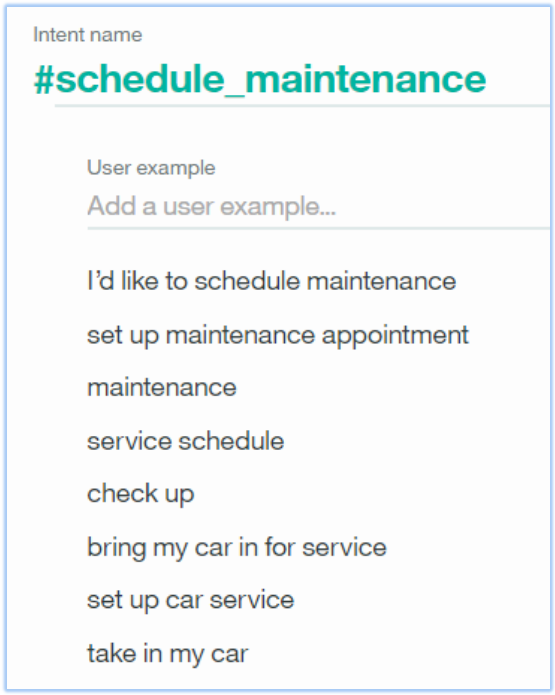
44. Give the intent a name, such as **schedule_maintenace**



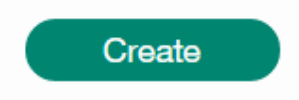
45. Add examples utterances for this new intent under User example

46. Start with the following samples (without quotes).
Either click the plus sign or use the enter key after each utterance :

```
"I'd like to schedule maintenance"
"set up maintenance appointment"
"maintenance"
"service schedule"
"check up"
"bring my car in for service"
"set up car service"
"take in my car"
```

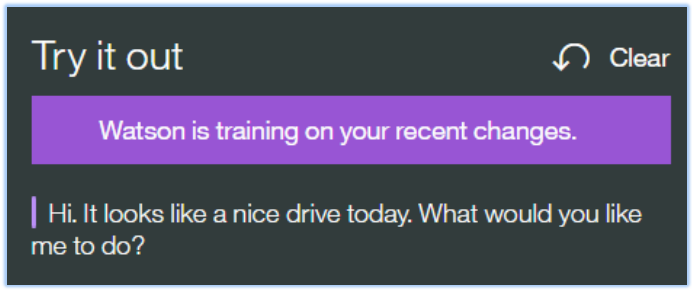


47. Click **Create**

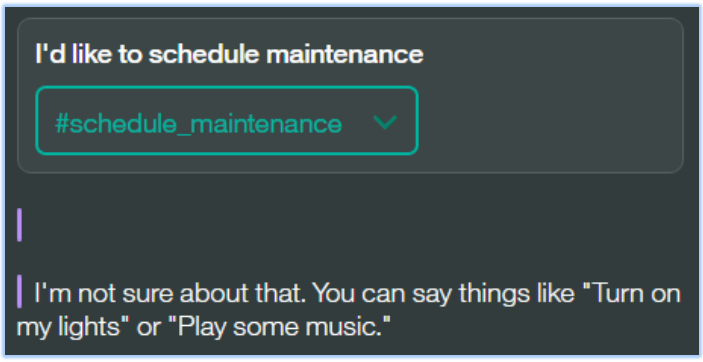


9.4.2 Test Intent

48. Open the try out panel (if there purple "training" banner is there, wait for it to complete training)



49. Test the new intent with some sample utterances to see if you get the **schedule_maintenance** intent



Recognize that conversation recognizes the correct intent, but it is not yet able to provide a response because you have not created a dialog to deal with the input.

9.5 Create Dialog

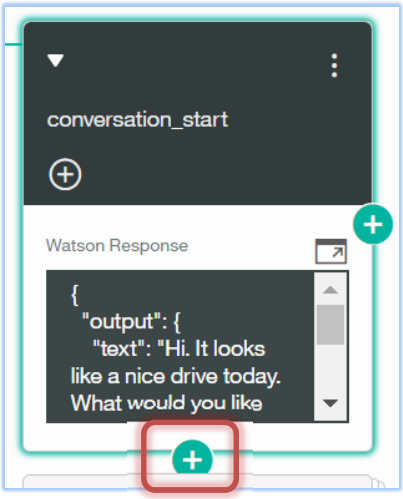
Next, capture some information from the end user that is necessary to schedule their maintenance appointment.

- The service date
- The number of miles on the car
- The phone number for a reminder calls.

50. Click the **Dialog** tab in the conversations UI

9.5.1 Add Dialog Nodes

51. Expand the **conversation_start** dialog node and click the plus sign on the bottom of the node to add a sibling

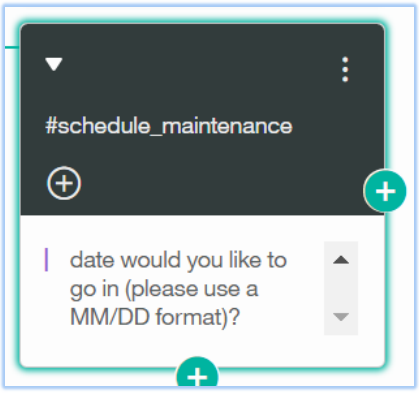


52. In the new node, add the condition for the **#schedule_maintenance** intent

53. Add an output statement asking the user for the date:

```
Ok. I can help you schedule your maintenance. What date would you
like to go in (please use a MM/DD format)?
```

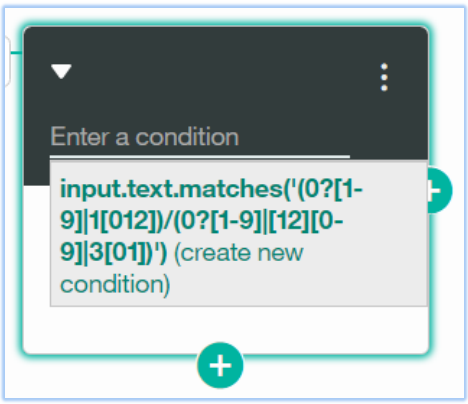
Note: Your are using a very simply date format in this exercise. → In the future this may be expanded.



54. Click the plus side on the right edge of your new node to create a new child node that will validate the date

55. In the input condition of the new node use a regular expression to ensure the user entered a date in the format of MM/DD (Month/Day). If the condition matches, store the date input in the context.

56. Use a regular expression such as the following:
(0?[1-9]|1[012])/ (0?[1-9]|[12][0-9]|3[01]) to match against the input.text [This expression allows for: 1-9, 01-09, 10, 11 or 12 as the month followed by a slash and 1-9, 01-09, 10-19, 20-29, or 30 or 31 as the date].
The full condition would be:
`input.text.matches('(0?[1-9]|1[012])/(0?[1-9]|[12][0-9]|3[01])')`



57. In the node Advanced Mode, store the input.text in a context variable called `maintenance_desired_date:`

58. Also supply an output prompt text, instructing the user to enter the approximate vehicle mileage. For example: "`Ok. Maintenance requested on <?input.text?>. Approximately how many miles does your vehicle have ?`"

59. Full response:

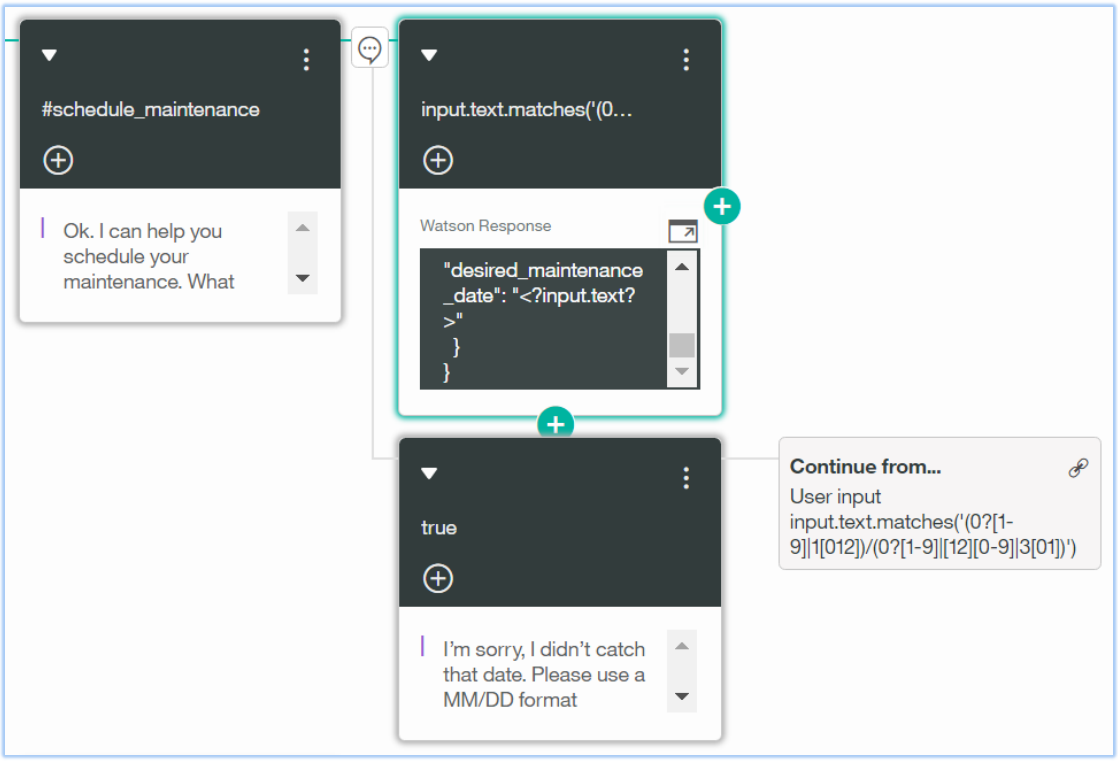
```
{
   "output":
    {
    "text":"Ok, Maintenance requested on <?input.text?>.
Approximately how many miles does your vehicle have?"
    },
    "context":
    {
    "desired_maintenance_date":"<?input.text?>"
    }
}
```

Advanced response


```
{
  "output":
   {
   "text":"Ok, Maintenance requested on <?input.text?>. Approximately how many miles does your vehicle have?"
   },
   "context":
   {
   "desired_maintenance_date":"<?input.text?>"
   }
}
```

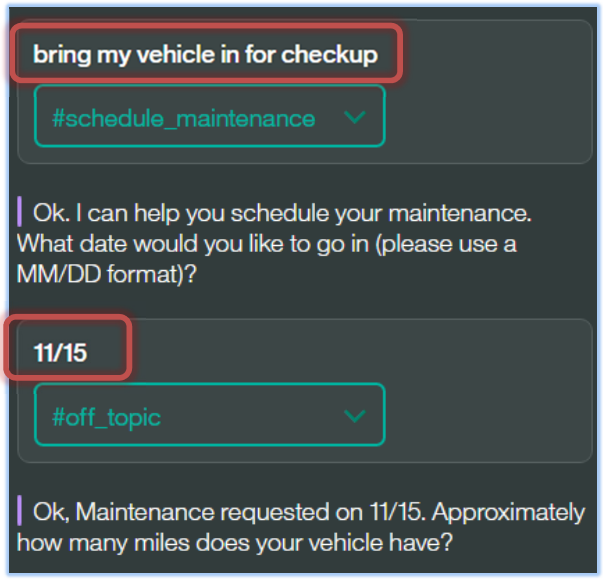

60. [optional] Add a true node in this part of the conversation for cases where the user entered date does not match the regexp. Advising them that the date was not valid and what the expected format is.
    Enter **condition** `true`
    Enter response:
    `I'm sorry, I didn't catch that date. Please use a MM/DD format (numeric)`

61. Then add a continue-from that points to the input node prior to the match node created.



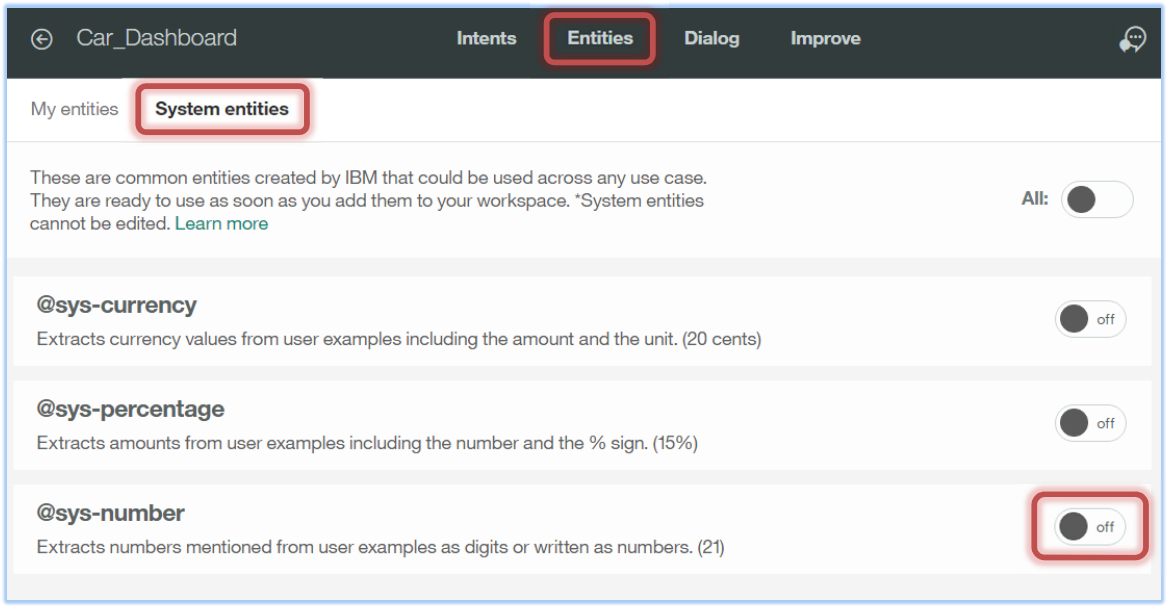
Test the new dialog nodes using the try it out panel

62. In the **try it out** panel, clear out any prior tests
63. User an utterance that should trigger the maintenance intent
64. When prompted, enter a date for the maintenance

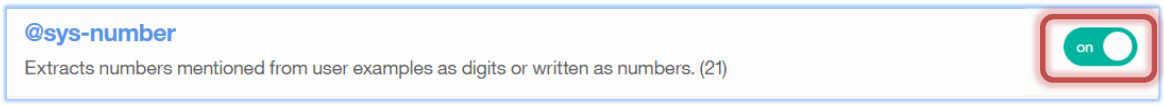


Next, you add the nodes to capture the vehicle mileage.
For this section ensure you have the @sys-number system entity turned on.

65. Under **Entities**, access **System entities**



66. Switch the system entity **@sys-number** to on



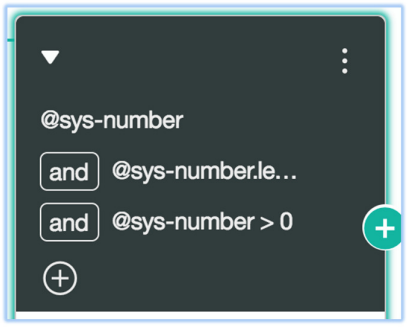
67. Go back to **Dialog**

68. Add a child node to your [date validation] node to create a new node that will capture the mileage

In the input condition of the new node use the number system entity to capture the vehicle miles. If the condition matches, store the date input in the context. Also validate that miles is a single non-negative number:

69. Add the following three conditions on the node:

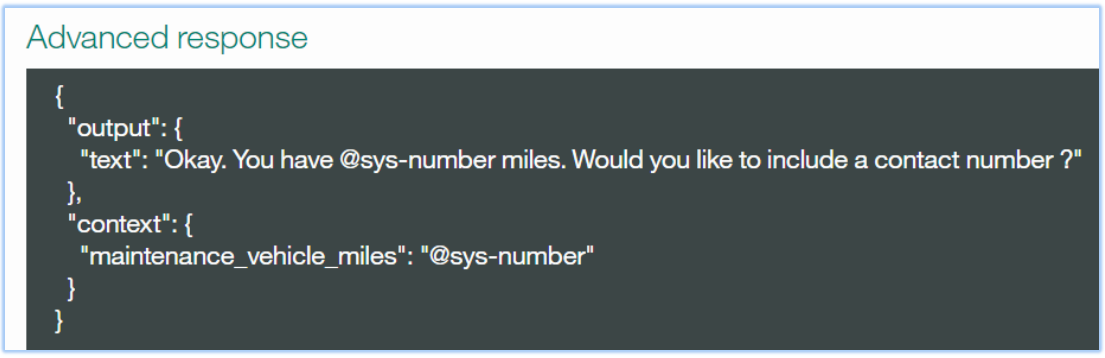
```
@sys-number AND @sys-number.length == 1 AND @sys-number > 0
```



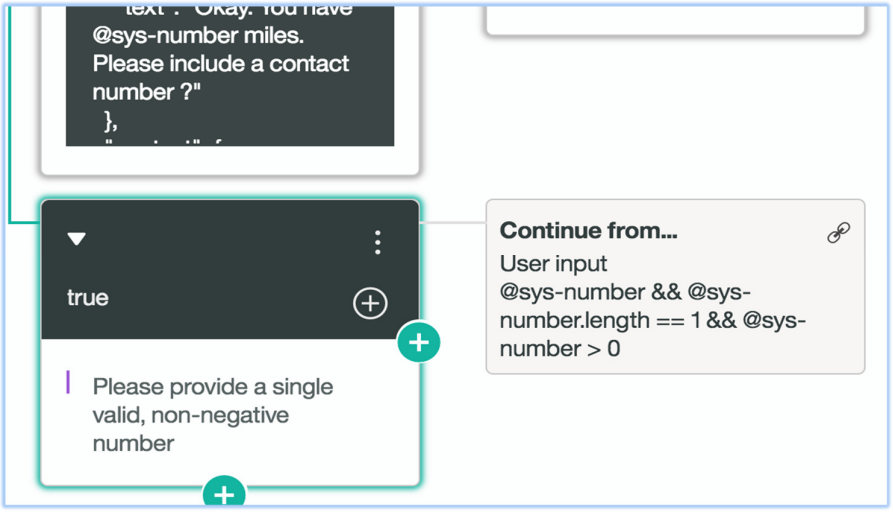
70. Switch the node to Advanced Mode and store the system entity in a context variable called "maintenance_vehicle_miles". Also supply an output prompt text instructing the user to enter a contact number (for example: `"Okay. You have @sys-number miles. Would you like to include a contact number?"`)

    Full code:

```
{
  "output": {
    "text": "Okay. You have @sys-number miles. Would you like to
include a contact number ?"
  },
  "context": {
    "maintenance_vehicle_miles": "@sys-number"
  }
}
```



71. Add a true node in this part of the conversation for cases where the user entered value does not match our three conditions above. Advising them that the mileage was not valid and what is expected. Then add a continue-from which points to input node prior to the mileage match node you created.



Test the new dialog:

72. Use an utterance that should trigger the maintenance intent

73. When prompted, enter a date for the maintenance

74. When prompted, enter a number (attempt different formats to test the entity and your validation rules, for example, “fifty thousand” , “-5”, etc

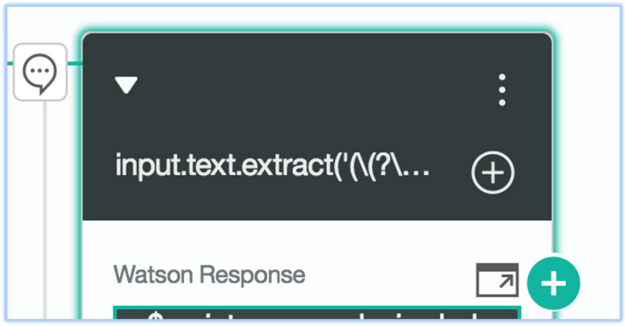
Finally, you will gather the users contact number for this scheduled maintenance. In this case, you will do it using extraction and regular expressions

75. Click the plus on the right edge of your vehicle mileage node to create a new node that will capture the contact details.

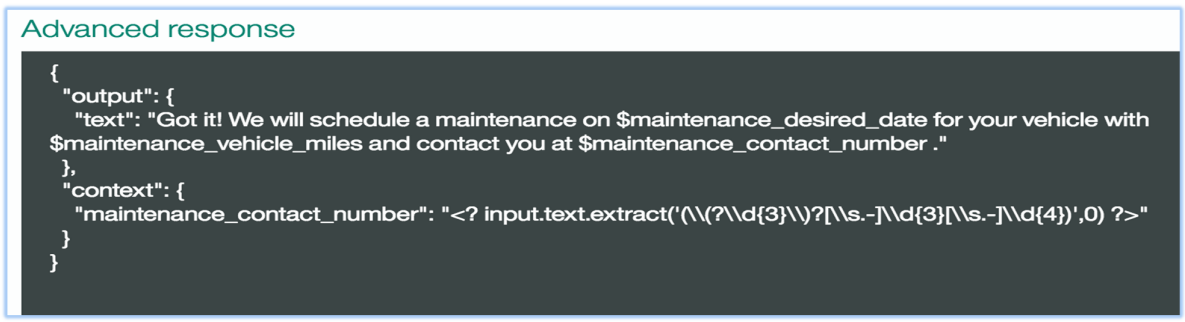
76. In the input condition of the new node use a regular expression to ensure the user entered a contact number in the format of 3 Digit Area Code – 7 Digit phone number. If the condition matches, store the contact number in the context.

    Use a regular expression to extract the phone number digits from the user input and validate that we received 5 digits:

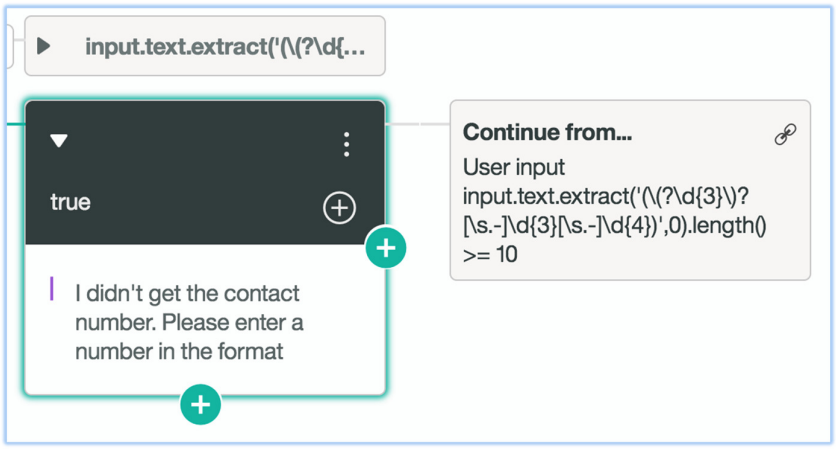
```
input.text.extract('(\(?\d{3}\)?[\s.-]\d{3}[\s.-]\d{4})',0).length()
>= 10
```



77. In Advanced mode, extract the number into a context variable called “maintenance_contact_number”.
    Supply an output prompt text informing the user of the details of this scheduled maintenance.
    (for example: `"Got it! We will schedule a maintenance on $maintenance_desired_date for your vehicle with $maintenance_vehicle_miles miles and contact you at $maintenance_contact_number ."`)



78. Add a true node in this part of the conversation for cases where the user entered text does not contain a number per the regexp. Advising them that the number was not valid and what the expected format is. Then add a continue-from that points to the input node prior to the match node created.



Test the new dialog nodes using the try it out panel.

79. Enter an utterance that should trigger the maintenance intent

80. When prompted, enter a date for the maintenance

81. When prompted, enter a number (attempt different formats to test the entity and your validation rules, for example, "`fifty thousand`", "`-5`", etc.)

82. When prompted, enter a phone number (again test different formats to ensure the extraction using regular expressions is working).

    That concludes this exercise.

9.6 Conclusion and Extensions

This exercise builds on the car demo application and hopefully demonstrates a couple of the uses for system entities and regular expressions in dialog flows. It is strongly recommended that the user attempt other regular expressions and the other system entities that are part of the service.

This lab does not include application layer integrations. It is presumed other validations and integrations would be accomplished in the application. For example, ensuring the selected maintenance date is available before continuing, or incorporating a call back mechanism (i.e SMS or phone via Twilio or others).