IBM Watson Assistant/Speech – Node.js

Cognitive Solutions Application Development

IBM Global Business Partners
Duration: 180 minutes
Updated: February 10, 2020
Klaus-Peter Schlotter
kps@de.ibm.com

Version 5.0 (Watson library V5.x)

Uses the Assistant API V2

Table of Contents

Overview

The IBM Watson Developer Cloud (WDC) offers a variety of services for developing cognitive applications. Each Watson service provides a Representational State Transfer (REST) Application Programming Interface (API) for interacting with the service. Software Development Kits (SDKs) are also available and provide high-level wrappers for the underlying REST API. Using these SDKs will allow you to speed up and simplify the process of integrating cognitive services into your applications.

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

Examples of where Watson Assistant could be used include:

- Add a chat bot to your website that automatically responds to customers' questions
- Build messaging platform chat bots that interact instantly with channel users
- Allow customers to control your mobile app using natural language virtual agents
- And more!

Objectives

- Learn how to provision a Watson Assistant service and utilize the web tool interface
- Learn how to train your chat bot to answer common questions
- Learn how to utilize the Watson Assistant service APIs in Node.js
- Connect the Watson Assistant to Watson Speech-To-Text and Text-To-Speech services

Prerequisites

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

- Guide – Getting Started with IBM Watson APIs & SDKs
- Create a IBM Cloud account

You need to have a workstation with the following programs installed:

1. Node.js
2. Express – A Node.js framework

```
1. npm install -g express-generator
```

Section 1: Create a Conversation Dialog in IBM Cloud

Create a Conversation Service in IBM Cloud

IBM Cloud offers services, or cloud extensions, that provide additional functionality that is ready to use by your application's running code.

You have two options for working with applications and services in IBM Cloud. You can use the IBM Cloud web user interface or the Cloud Foundry command-line interface. (See Lab 01 on how to use the Cloud Foundry CLI).

Note: In this lab we use the IBM Cloud web UI.

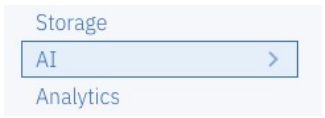
Step 1 In a web browser, navigate to the following URL

https://cloud.ibm.com.

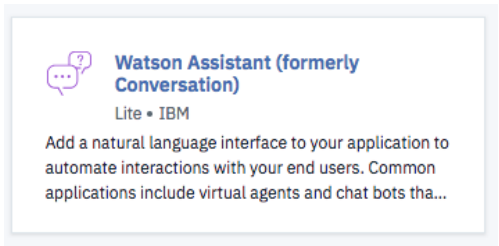
Step 2 Log in with your IBM Cloud credentials. This should be your IBMid.

Step 3 You should start on your dashboard which shows a list of your applications and services. Scroll down to the All Services section and click **Create Resource**.

Step 4 On the left, under Services, **click** on *AI* to filter the list and only show the cognitive services.



Step 5 Click on the Watson Assistant service.

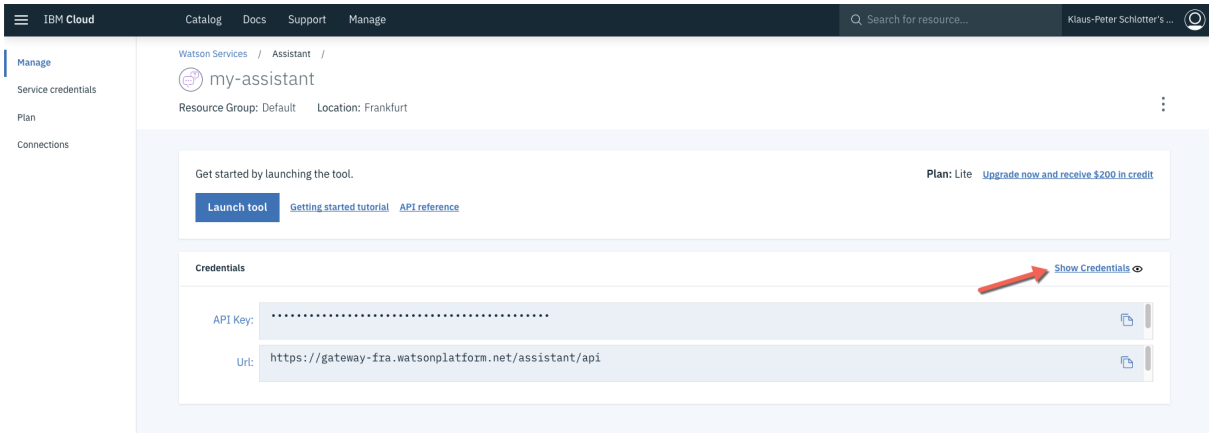


Step 6 Review the details for this service. At the top, there will be a description of the service. At the bottom, you can review the pricing plans. The Lite plan for this service provides no cost monthly allowances for workspaces, intents, and API calls. Enjoy your demo!

Step 7 Enter the information for your service, then **click** Create.

Field	Color property
Service name	my-assistant
Selected Plan	Lite
Chose a region/location to deploy	<yourRegion>
Select a resource group	Default

Step 8 IBM Cloud has created a new service instance.



Step 9 In the *Credentials* section **click** Show Credentials. You should see the *API Key* for your service. Later in this exercise, you will enter this value into a JSON configuration file for your Node.js application. Feel free to copy them to your clipboard, to a text file, or just return to this section of the IBM Cloud web interface when the credentials are needed.

Create a Watson Assistant Skill

Before using the Assistant instance, you will need to train it with the intents, entities, and/or dialog nodes relevant to your application's use case. You will create these items in a Skill in the following steps. A Skill is a container for the artifacts that define the behavior of your service instance.

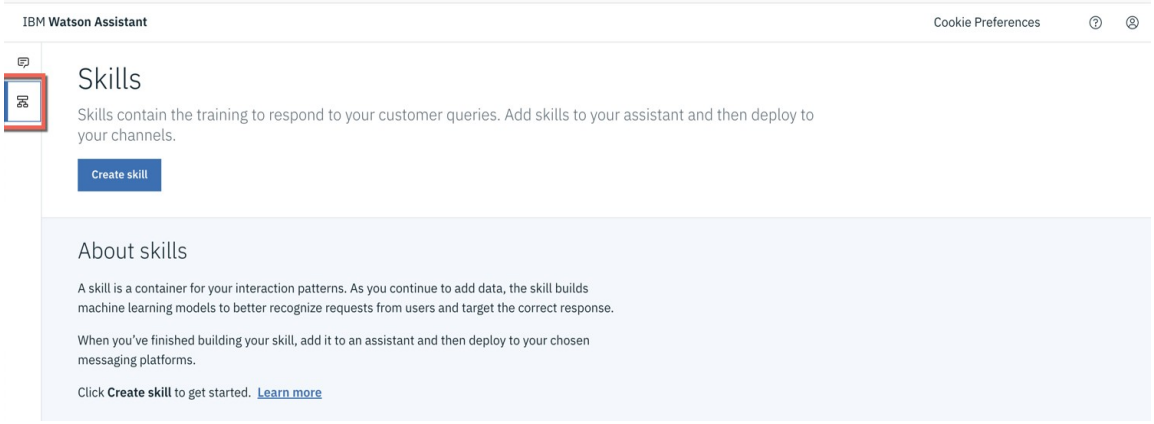
The resulting Skill is also available as JSON backup file on GitHub. In a Terminal you can download it with the following command:
`wget https://raw.githubusercontent.com/iic-dach/speech-assistant/Lab3_skill-CSAD-Demo.json`
On the *Add Dialog Skill* page, on the *Import skill* tab you can import this JSON file..

Step 10 In this guide, you will build a Conversation simple app to demonstrate the Natural Language Processing capabilities of IBM Watson in a simple chat interface. You will also learn how to identify a user's intent and then perform an action from a third-party application.

Step 11 Click the **Launch tool** button.

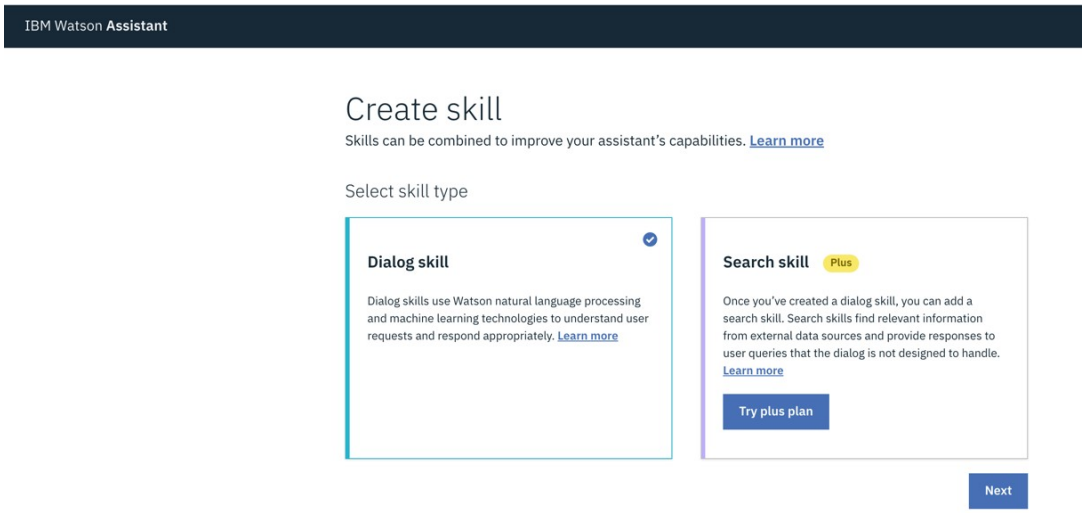
Step 12 The IBM Watson Assistant opens on the *Assistants* tab.

Step 13 Open the *Skills* tab. Here you can create new Skills.



Step 14 First, you will need to create a new workspace. Workspaces enable you to maintain separate intents, user examples, entities, and dialogs for each application. **Click** Create skill

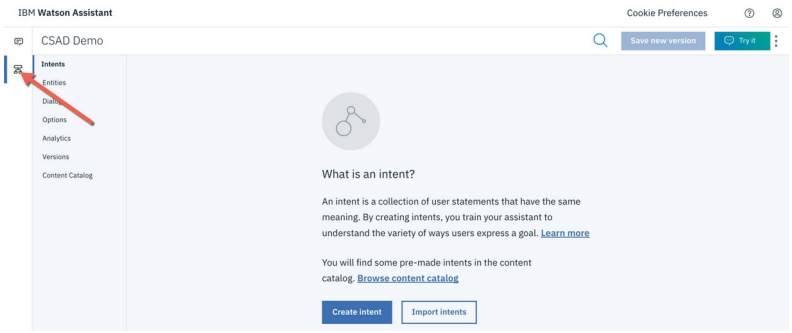
Step 15 On the *Create skill* page **select** *Dialog skill* and **click** Next



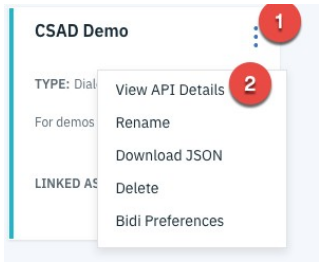
Step 16 On the *Create Dialog Skill* page, on the *Create skill* tab, **enter** the following values and click Create dialog skill.

Field	Value
Name	CSAD Demo
Description	For demos only
Language	We use **English(U.S.)** for this demo

Step 17 Once the Skill has been created, you will be redirected to it. However, before proceeding, you will need to know how to identify the Skill so that it can be referenced by future applications. At the left, **click** Skills.



Step 18 On the tile for your new Skill, click **Options** → **View API Details**.



Step 19 Locate the Workspace ID. You will need this value in future steps when creating a JSON configuration file for your demo application. Feel free to copy the value or just return to this section of the Conversation tooling web interface when the ID is needed.

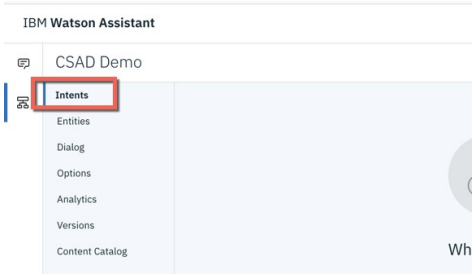
Step 20 In the upper right corner of this page, **click** ✕.

Step 21 **Click** on the Skill tile to be taken back to the new Skill.

Create Intents

Before using the new conversation, you will need to train it with the intents, entities, and/or dialog nodes relevant to your application's use case. An intent is the purpose or goal of a user's input.

Step 22 First, you will need to define some intents to help control the flow of your dialog. An intent is the purpose or goal of a user's input. In other words, Watson will use natural language processing to recognize the intent of the user's question/statement to help it select the corresponding dialog branch for your automated conversation. If not already there, **click** the *Intents* tab at the left of your workspace.



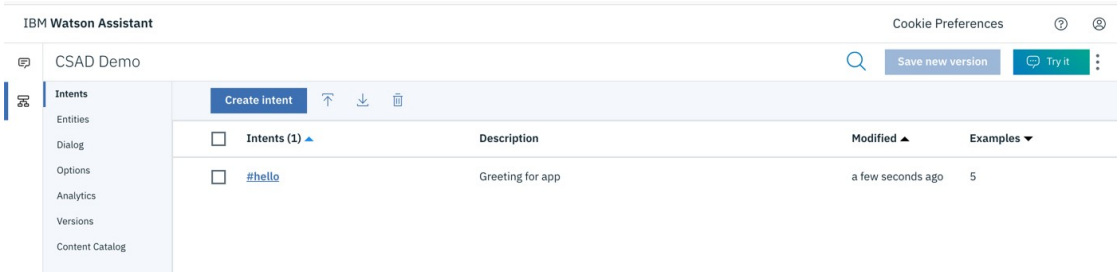
Step 23 **Click** Create intent enter the following values and **click** Create intent

Field	Value
Intent name	hello
Description	Greeting for app

Step 24 The user examples are phrases that will help Watson recognize the new intent. (Enter multiple examples by **pressing** "Enter" or by **clicking** the *Add example*). When finished, **click** ← at the top of the page.

Field	Value
User example	Good morning Greetings Hello Hi Howdy

Step 25 You should see your new intent listed on the Intents page. The number you see will indicate the total number of user examples that exist for that intent.



Step 26 Repeat the previous steps to create a new *intent* that helps the user end the conversation.

Field	Value
Intent name	goodbye
Description	Goodbye
User example	Bye Goodbye Farewell I am leaving See you later

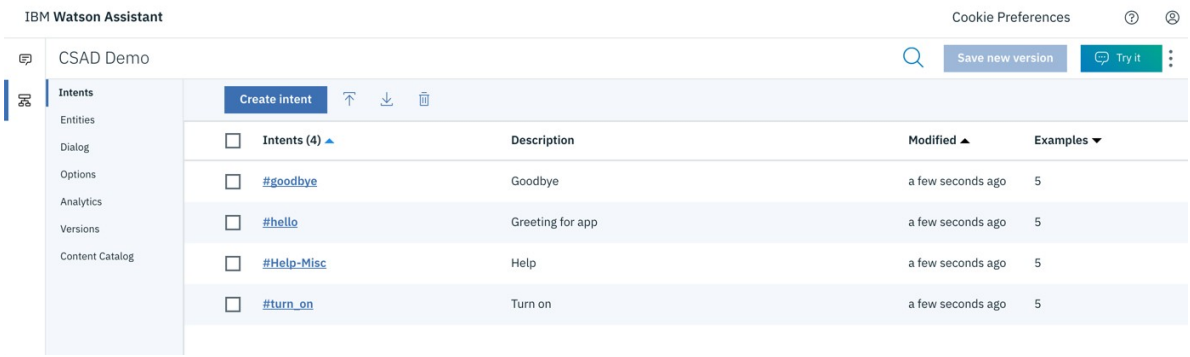
Step 27 Repeat the previous steps to create a new *intent* that helps the user ask for help. In the programming section of this guide, you will learn how to identify the user's intent (in a third-party application) so that you can perform the requested action.

Field	Value
Intent name	Help-Misc
Description	Help
User example	I have a request I would like some assistance I need information I have a problem I need help

Step 28 Repeat the previous steps to create a new *intent* that helps the user issue commands to turn on a device. In this example, you will assume the user is interacting with a home/business automation system. The purpose of this intent is to show you (in the next section) how to associate *entities* with an *intent* when building your dialog tree. Additionally, this intent demonstrates that the Conversation service can be used for more than just chat bots. It can be used to provide a natural language interface to any type of system!

Field	Value
Intent name	turn_on
Description	Turn on
User example	Arm the security system Lock the doors I need lights Turn on the lights Start recording

Step 29 At this point, you have defined some intents for the application along with the example utterances that will help train Watson to recognize and control the conversation flow.



Create Entities

Before using the new Assistant, you will need to train it with the intents, entities, and/or dialog nodes relevant to your application's use case. An entity is the portion of the user's input that you can use to provide a different response or action to an intent.

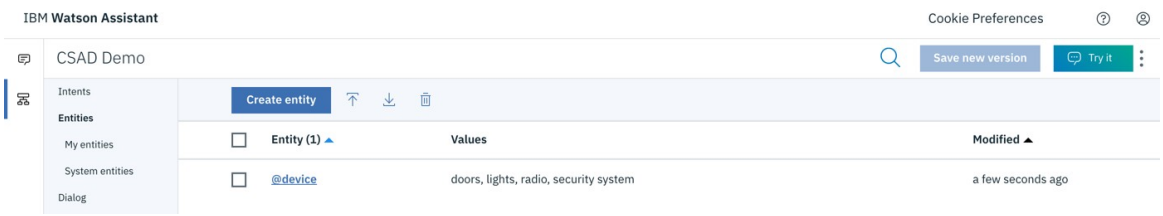
Step 30 Next, you will need to create some *entities*. An *entity* is the portion of the user's input that you can use to provide a different response or action to an *intent*. These entities can be used to help clarify a user's questions/phrases. You should only create *entities* for things that matter and might alter the way a bot responds to an *intent*. If not already there, **click** the *Entities* tab at the left of your Skill.

Step 31 On the *Entities* tab, click Create entity. In the dialog window, enter the following information. In this example, the **#turn_on** *intent* will indicate that the user wants to turn on a car device. So, you will need to create a new *entity* representing a device. **Enter** *device* and **click** Create entity. You will then provide *values* (and possibly synonyms) for the various types of devices that can be turned on. (Enter multiple *examples* by **pressing** "Enter" or by **clicking** the plus sign at the end of the line.)

Click Add value. When finished, **click** ← at the top of the page.

Field	Value	Synonym
Entity name	device	
Value	security system	alarm
	lights	bulb, lamp
	doors	locks, gates
	radio	car radio

Step 32 You should see your new *entity* listed on the *Entities* page.



Step 33 Repeat the previous steps to create the following new *entity* for **lights** controlled by the system.

Field	Value	Synonym
Entity name	lights	
Value	fog lamp	fog light
	high beam	full beam, main beam, brights
	low beam	headlights, passing lights, dim light
	rear fog lamp	rear fog light

Step 34 Repeat the previous steps to create the following new *entity* to request the current time.

> **Note:** You can get the time by using one of the predefined system entities. In this exercise, go ahead and enter it manually so it can be used to demonstrate future concepts.

Field	Value	Synonym
Entity name	help	
Value	time	clock, hour, minute, second

Step 35 At this point, you have defined *intents* and the associated *entities* to help Watson determine the appropriate response to a user's natural language input. Your application will be able to:

- Respond to a request to turn on specific devices
- If a user turns on lights, provide additional choices for light locations

Create Dialogs

> Before using the new conversation, you will need to train it with the intents, entities, and/or dialog nodes relevant to your application's use case. A dialog uses the intent and entity that have been identified, plus context from the application, to interact with the user and provide a response.

Step 36 Next, you will need to create some *dialogs*. A dialog is a conversational set of nodes that are contained in a workspace. Together, each set of nodes creates the overall dialog tree. Every branch on this tree is a different part of the conversation with a user. If not already there, **click** the *Dialog* tab at the left of your Skill.

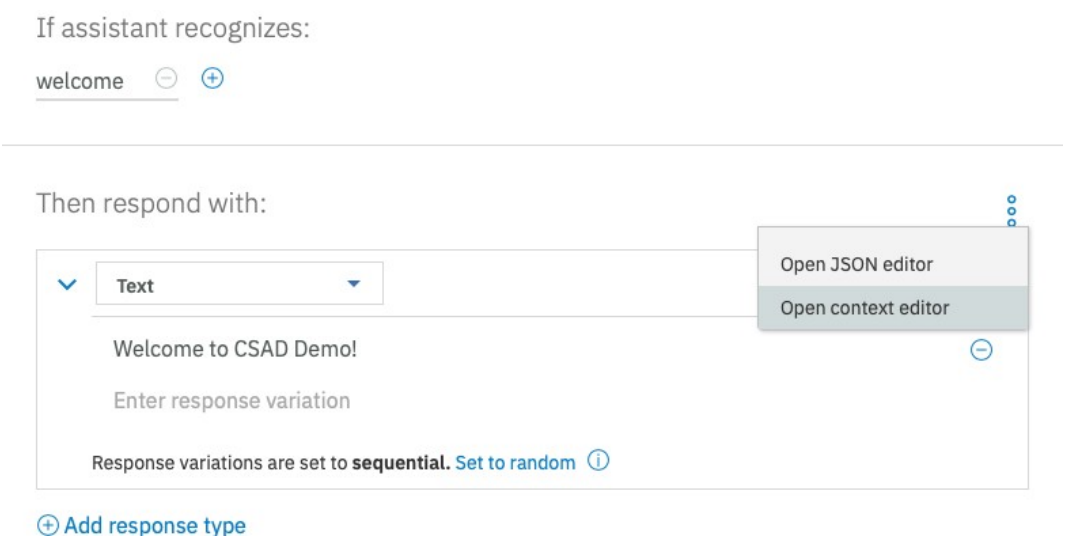
Step 37 Review the documentation for creating dialog nodes and for defining conditions.

Step 38 On the Dialog tab, two default nodes are created for you named *Welcome* and *Anything* else. The *Welcome* node is the starting point for this application's conversation. That is, if an API query is made without a context defined, this node will be returned. The *Anything else* node will be used if the user input does not match any of the defined nodes.

Step 39 **Click** on the *Welcome* node to update the following properties

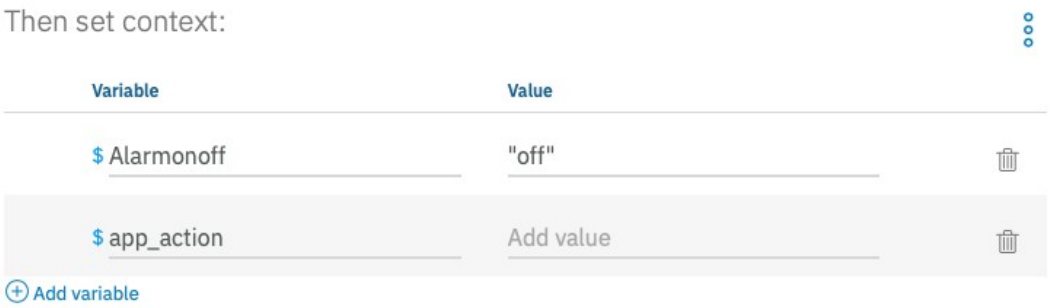
Field	Value
Name	Welcome *(should be the default)*
If bot recognizes:	welcome *(should be the default)*
Then respond with:	Welcome to the CSAD Demo!

Step 40 **Click** ⋮ then **click** *Open context editor*.



Step 41 Add two variables and **click** Customize ⚙ [×] to close the properties view of the node.

Variable	Value
Alarmonoff	off
app_action	



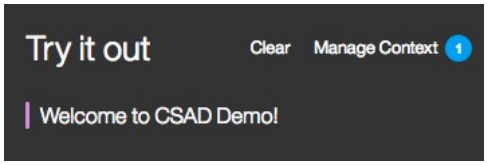
Step 42 Expand the *Anything else* node and review it's default values. **Close** the view by **clicking** × .

Field	Value
Name	Anything else (should be the default)
If bot recognizes:	anything_else (should be the default)
Then respond with:	1. I didn't understand. You can try rephrasing 2. Can you reword your statement? I'm not understanding. 3. I didn't get your meaning. (all should be default)
Response variations are sequential.* (Set to random)	

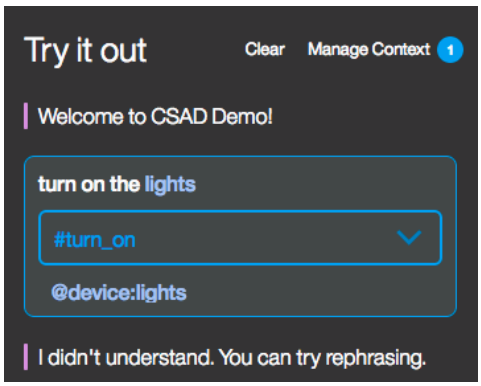
* Sequential means 1st response presented at first hit of anything_else node, and so on. Random means any of the responses is presented randomly.

Step 43 Now it's time to test the conversation.
In the upper right corner, **click** Try it

Step 44 A test user interface will immediately launch and, based on the *Welcome* node, provides a greeting to the end user. (You may see a message that Watson is being trained.)



Step 45 Since you have not yet defined any other dialog nodes (associated with your *intents* and *entities*), everything typed by the user will get routed to the *Anything else* node. F.e **type** "*turn on the lights"*.



Although we have defined this phrase in intents and entities, the system does not recognize them because we have not yet defined a node to catch them the bot does not yet understand (*Anything else* node).

Step 46 Did you notice the drop-down menu that appeared for your invalid input? You can optionally assign this phrase to an existing intent (or verify the correct intent was used). You can use this functionality in the future to keep Watson trained on new user inputs and to ensure the correct response is returned. Cool! For now, just proceed to the next step.

Step 47 **Click** in the top right corner to close the chat pane. Proceed to the next section.

Build your CSAD Assistant dialog

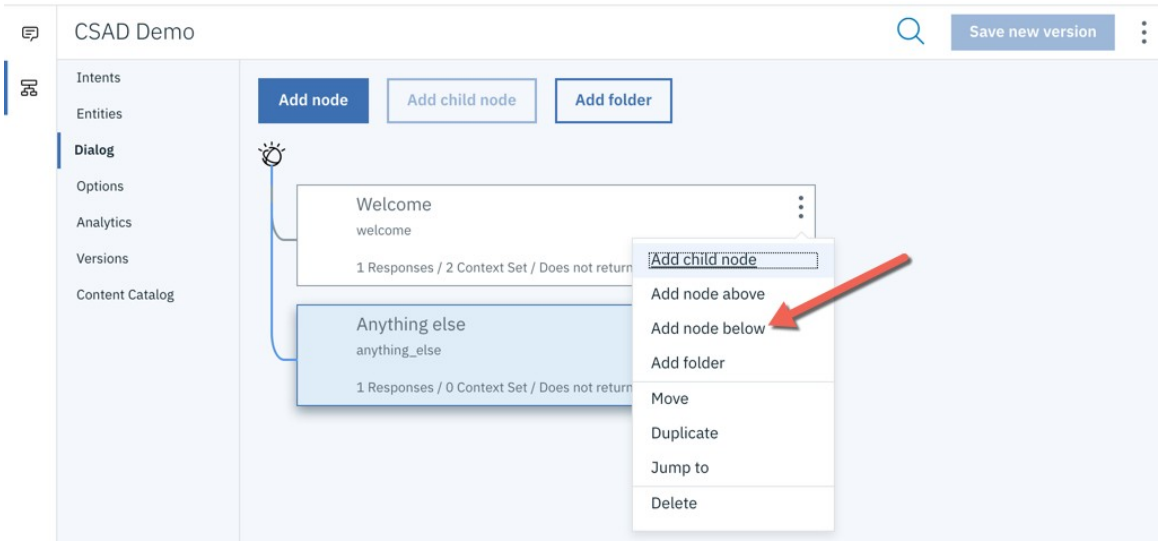
In this section, you will continue building your demo bot utilizing the intents, entities, and dialog nodes that you created in previous steps. You will do this entirely with the web interface (no programming or text/XML file hacking required!)

Step 48 You should create a dialog branch for each of the intents you identified as well as the start and end of the conversation. Determining the most efficient order in which to check conditions is an important skill in building dialog trees. If you find a branch is becoming very complex, check the conditions to see whether you can simplify your dialog by reordering them.

Note: It's often best to process the most specific conditions first.

Step 49 **Click** the menu on the *Welcome* node and then **click** *Add node below*.

In this step you are creating a new branch in your dialog tree that represents an alternative conversation.



Step 50 In this new node, enter the following values. By setting the condition to an *intent*, you are indicating that this node will be triggered by any input that matches the specified *intent*. Then **click** × to close the dialog.

Field	Value
Name this node…	Hello
If bot recognizes:	#hello
Then respond with:	Hi! What can I do for you?

Step 51 **Click** the menu ⋮ on the *Hello* node and then **click** *Add node below*, with the following values. Then **click** × to close the dialog.

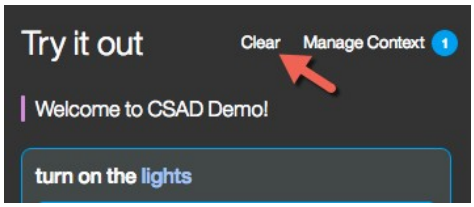
Field	Value
Name this node…	Goodbye
If bot recognizes:	#goodbye
Then respond with:	Until our next meeting.

Step 52 Using the same steps (Step 40 ff) as before, test the conversation by typing the following chat line(s):

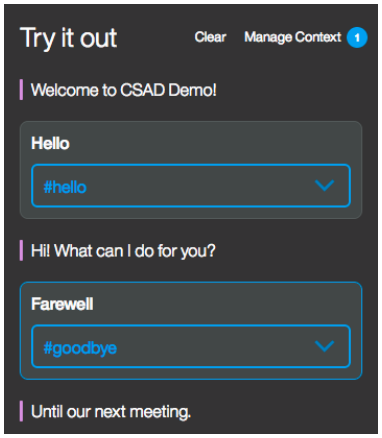
```
Hello
```

```
Farewell
```

You can clear previous tests by clicking **Clear** at the top of the dialog.

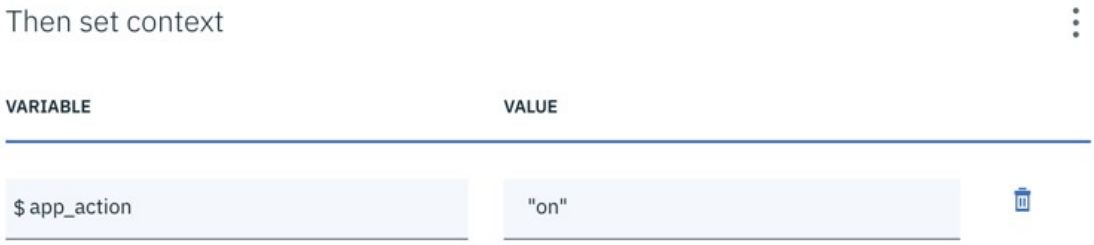


You should see the appropriate result:



Step 53 Next, you should create a new node below **Hello** (*Add node below*) for the **#turn_on** *intent*. As you'll recall, you have multiple devices that you might want to turn on. In earlier steps, you documented these devices using a new **@device** *entity*. This dialog branch will require multiple nodes to represent each of those devices. In this new node, enter the following values. In this example, the dialog branch will need additional information to determine which device needs to be turned on. So, leave the "Responses" field blank. **Click** × to close.

Field	Value
Name this node…	Turn on
If bot recognizes:	#turn_on
Then respond with:	
In the Context Editor (click ⋮)	
app_action	on



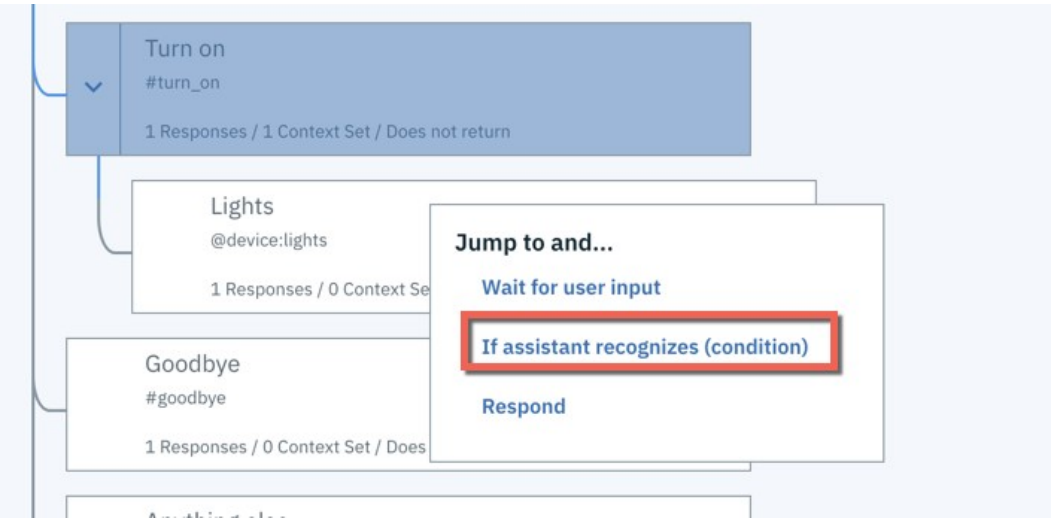
Step 54 **Click** the menu ⋮ on the *Turn On* node and **click** *Add child node*.

Step 55 In this new node, enter the following values. In this example, the only way this node will be reached is if the user specifies the **@device** *entity* "lights" (or one of its synonyms). Then **click** × to close the dialog.

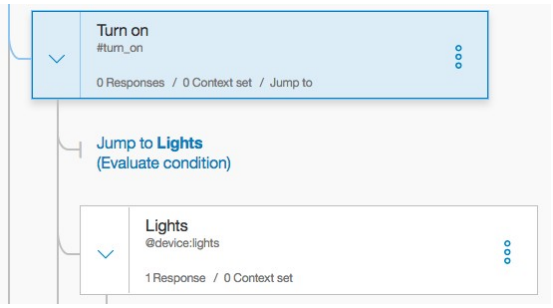
Field	Value
Name this node…	Lights
If bot recognizes:	@device:lights
Then respond with:	OK! Which light would you like to turn on?

Step 56 In this scenario, you want to automatically move from the *Turn On* node to the *Lights* node without waiting for additional user input.

a) **Click** ⋮ on the *Turn on* node and **select** *Jump to.* Now the Turn on node is selected. **Click** the *Lights* node and **select** I*f assistant recognizes (condition)*.



b) If bot recognizes (condition) is **lights**.



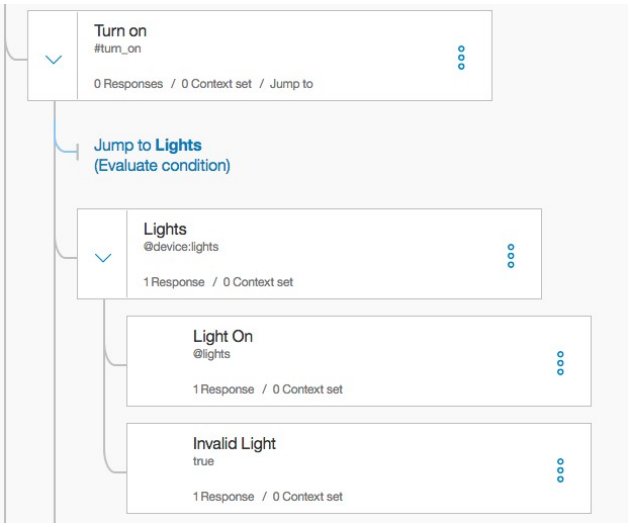
Step 57 At this point, Watson will ask you which lights to turn on. As you will recall, in earlier steps you created a @lights entity to define the available lights in the system. **Click** the menu ⋮ on the *Lights* node, **click** *Add child node*. In this new node, enter the following values. By setting the condition to an entity, you are indicating that this node will be triggered by any input that matches the specified entity.

Field	Value
Name this node…	Light On
If bot recognizes:	@lights
Then respond with:	OK! Turning on @lights light.

Step 58 Next, you will want Watson to respond if it does not recognize a light, or entity, provided by the user. **Click** the menu ⋮ on the **Light On** node and **click** *Add node below* to create a new peer node. In this new node, enter the following values.

> **Note:** The *true* keyword in *If bot recognizes* always executes this node when reached. That means, the conditions of the siblings above (Light On) are not recognized!

Now your tree for the lights should look like the following:



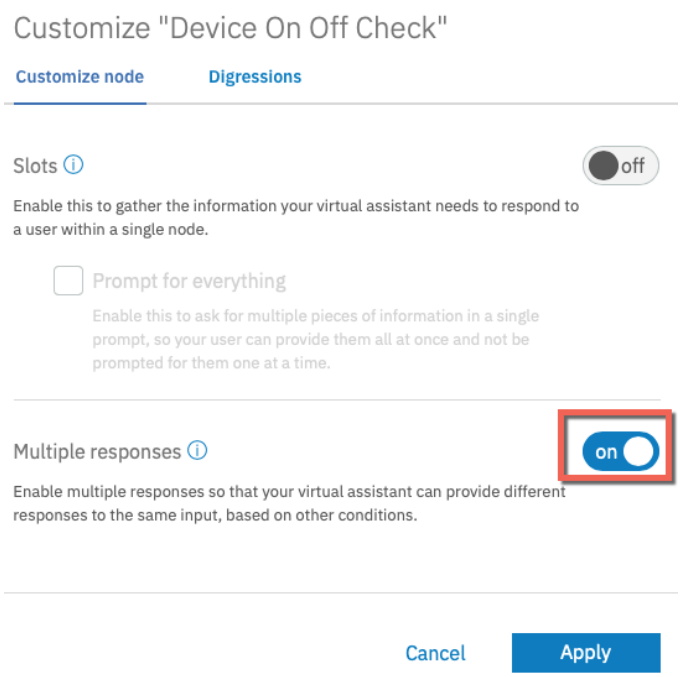
Step 59 You will need to add nodes to deal with the other devices that can be turned on. As you'll recall, you defined those devices with the **@device** *entity*. The **Turn On** node automatically jumps to the Lights node. So, **click** the menu ⋮ on the Lights node and **click** *Add node below* to create a new peer node. **Enter** the following values.

Field	Value
Name this node…	Device
If bot recognizes:	@device
Then respond with:	

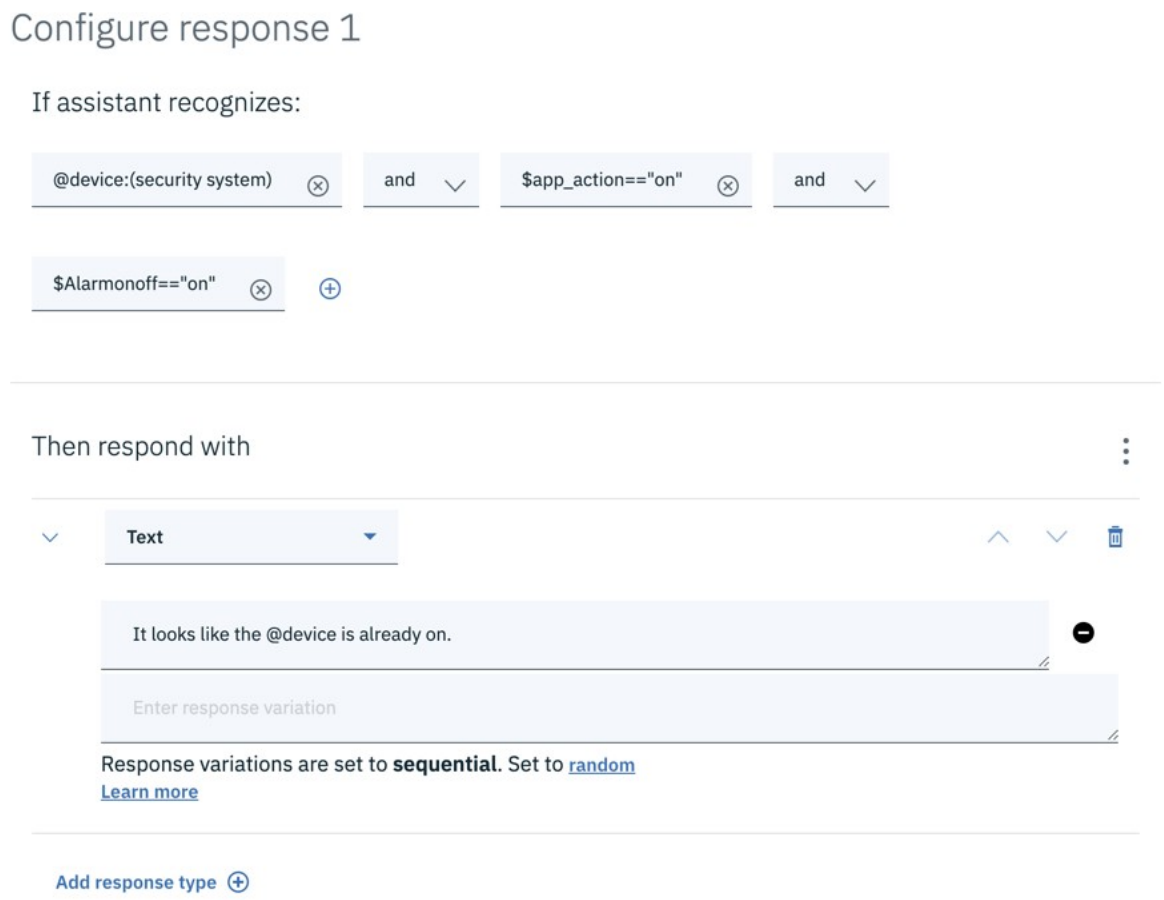
Step 60 On the *Device* node **click** ⋮ and **click** *Add a child node*. Add the following values

Field	Value
Name this node…	Device On Off Check
If bot recognizes:	true

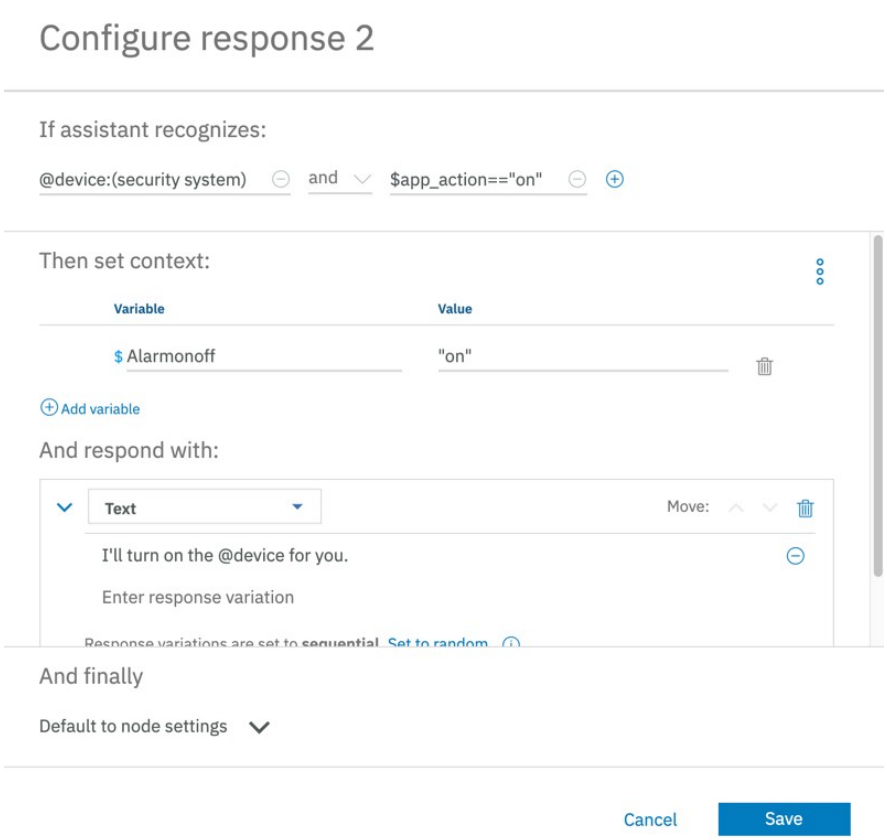
Step 61 **Click** Customize and **enable** *Multiple responses*. The click Apply.



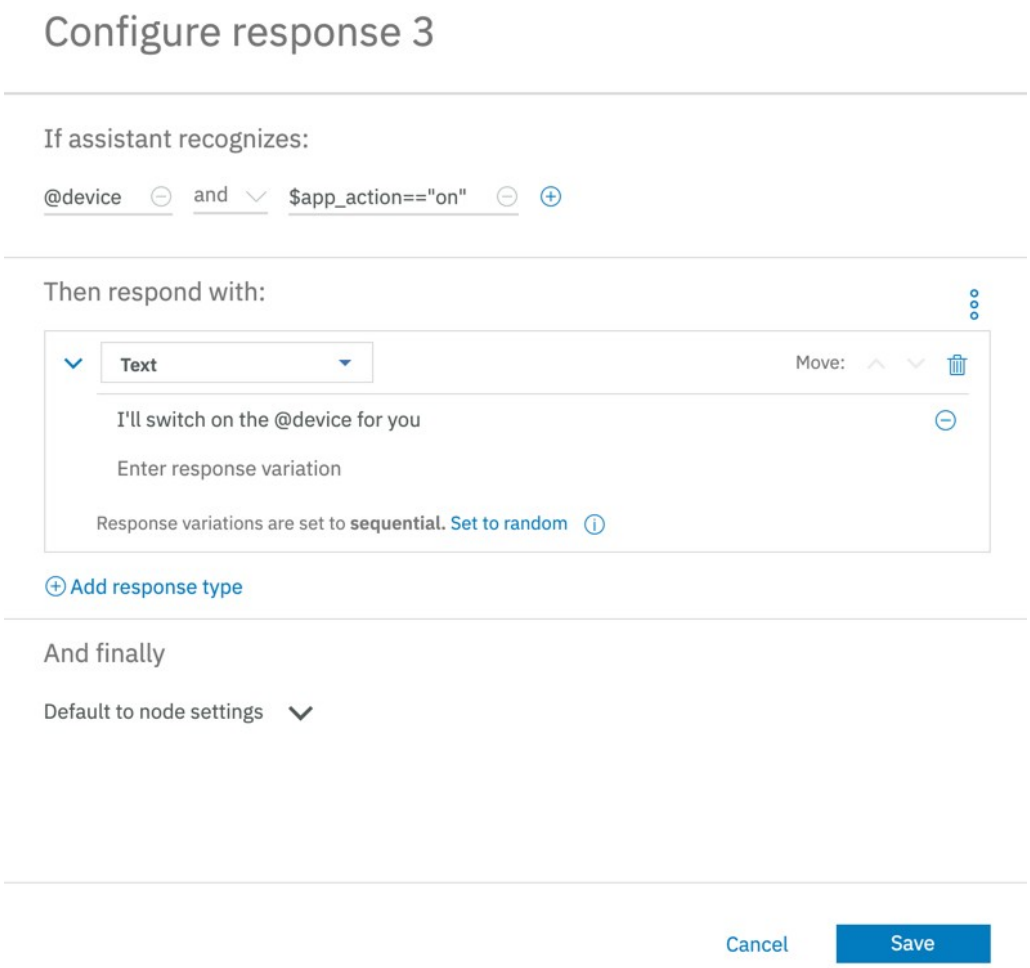
Step 62 Now add the following three answers in *Then respond with*: Use the ⚙ to customize and enter the values interactively and **click** Save. See the screenshots of each response:



If assistant recognizes:
@Device:(security system) AND $app_action==”on” AND $Alarmonoff==”on”
Then respond with:
It looks like the @device is already on.

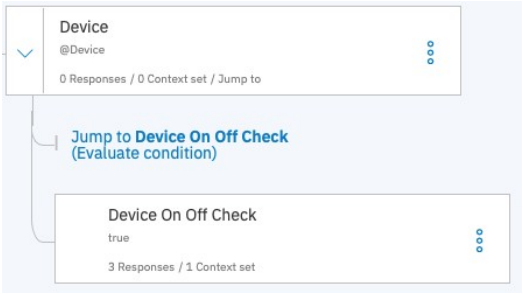


If assistant recognizes:
@Device:(security system) AND $app_action==”on”
Then set context (open the context editor)
$Alarmonoff “on”
And respond with:
I’ll turn on the @device for you.



If assistant recognizes:
@Device AND $app_action==”on”
Then respond with:
I’ll switch on the @device for you.

Step 63 Now **click** ⋮ on the *Device* node and **click** *Jump to*, then **select** the *Device On Off Check (If recognizes condition)*.



Step 64 Next, you will want Watson to respond if it does not recognize a device, or entity, provided by the user. **Click** the menu ⋮ of the *Device* node and **click** *Add node below* to create another peer node with the following values:

Field	Value
Name this node…	Invalid Device
If bot recognizes:	true
Then respond with:	I'm sorry, I don't know how to do that. I can turn on lights, radio, or security systems.

Step 65 Using the same steps as before, **click** Try it to test the conversation by **typing** the following chat line(s):

```
Hello

Arm the security system

Arm the security system          → Should be already on

Turn on the lights

The headlights

Turn on the lights

fog lamp
```

Note: You could add a context variable for each device/light and control the on/off behavior as for the *security system*.

Step 66 Finally, you will want to enable the Conversation service to identify your **#Help-Misc** *intent*. You will make this a part of the overall "Help" system for the bot. This intent will be used in the upcoming programming exercises to show you how to perform specific application actions based on a detected *intent* and *entity*. **Click** the menu ⋮ on the *Turn On* node and **click** *Add node below* to create a new peer node. Enter the following values.

Field	Value
Name this node…	Help
If bot recognizes:	#Help-Misc
Then respond with:	How can I help you?

Step 67 **Click** *Add child node* on the menu of the Help node.. In this new node, enter the following values. In this example, the only way this node will be reached is if the user specifies the @help:time *entity* (or one of its synonyms).

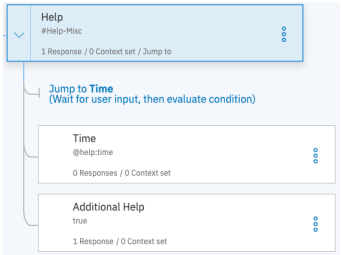
Note: You will provide your own custom response to the query in the programming exercises.

Field	Value
Name this node…	Time
If bot recognizes:	@help:time
The respond with:	The time will be provided by the server created later in this tutorial.

Step 68 **Click** *Add node below* on the menu of the *Time* node. In this new node, enter the following values. By setting the condition to an entity, you are indicating that this node will be triggered by any input that matches the specified entity.

Field	Value
Name this node…	Additional Help
If bot recognizes:	true
Then respond with:	I'm sorry, I can't help with that. For additional help, please call 555-2368.

Step 69 **Click** ⋮ on the *Help* node and **select** *Jump to*. **Click** the *Time* node and **select** *Jump to and… Wait for user input*.



Step 70 This is already an epic conversation, but feel free to experiment with more functionality:

- Define entities for additional devices and lights
- Add more synonyms for entities
- Add new intents, such as #turn_off to turn off devices

(Optional:) Create a Slots Dialog to book a Restaurant Table

Step 71 Add a slots dialog to book a table in a restaurant

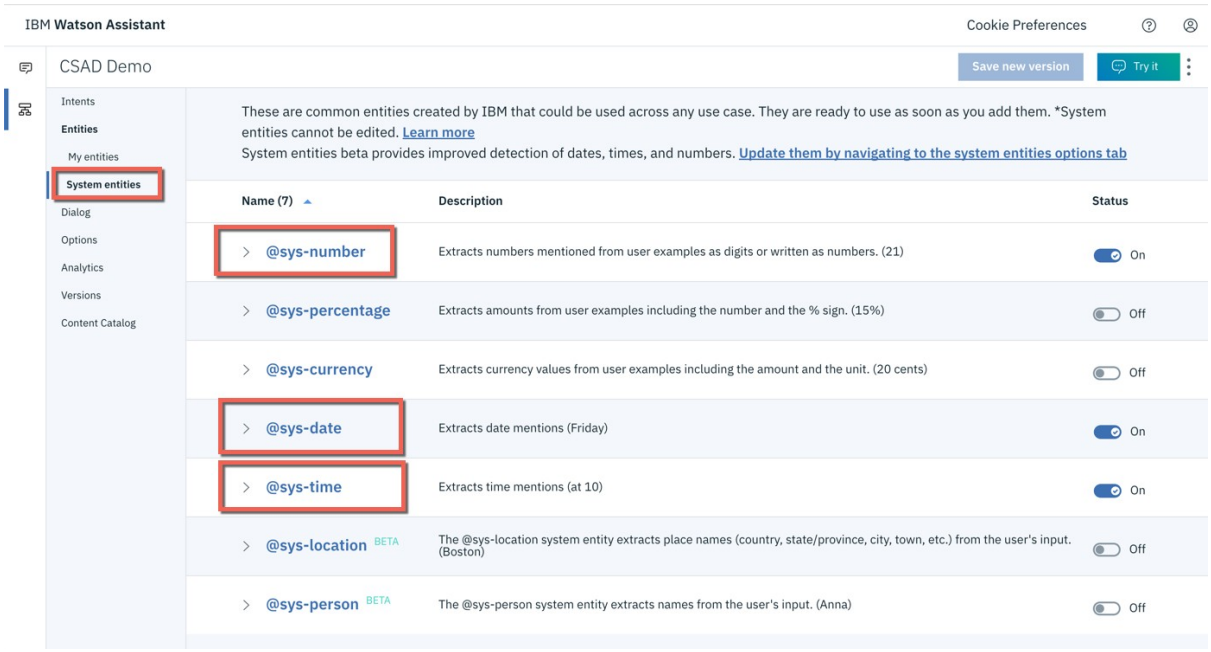
a) Add an Intent with the following values (see Step 21 ff):

Field	Value
Intent name	book_table
Description	Book a table in one of the restaurants
User example	I'd like to make a reservation I want to reserve a table for dinner Can 3 of us get a table for lunch? Do you have openings for next Wednesday at 7? Is there availability for 4 on Tuesday night? I'd like to come in for brunch tomorrow Can I reserve a table?

b) Add an Entity for Locations (See Step 29 ff)

Field	Value	Synonym
Entity name	locations	
Value	First Street	first, 1st
Value	Main Street	Main

c) Enable the System entities @sys-date, @sys-number, @sys-time



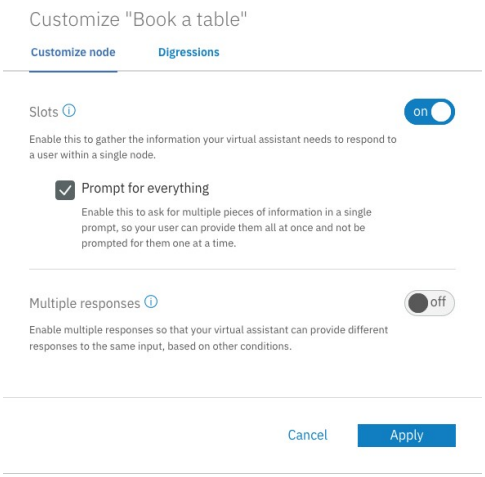
d) Add a dialog node for #book_table (Step 48 ff)

Click the menu ⋮ on the *Turn On* node and **click** *Add node below* to create a new peer node. Enter the following values.

Field	Value
Name this node…	Book a Table
If bot recognizes:	#book_table

e) Click ⚙ Customize at the top of dialog node definition panel.

f) **Enable** *Slots* and **select** *Prompt for everything*. The click Apply



g) Now you can enter the slots (Then check for:) ⊕ Add slot for more lines.

Check for	Save it as	If not present ask
@locations	$locations	Which store you want to got to? First or Main?
@sys-date	$date	What day you want to come in?
@sys-time	$time	What time did you want to arrive?
@sys-number	$number	How many people in your party?

h) In the field *If no slots are pre-filled, ask this first*: **enter**

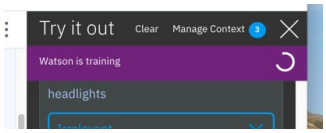
```
I need some more information to continue. I will need
the location, date, time, and number of people
```

i) In the field *Then respond with*: enter

```
Great, I have a table of $number people, on $date at
$time at our $locations store.
```

j) You can try (see Step 40 ff) this with a view sample inputs. Always Clear

If you see indication for “Watson is training”, wait until completion:

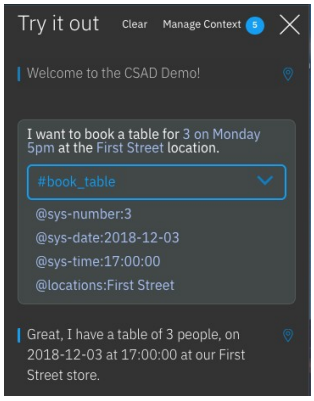



```
I want to book a table for 3 on Monday 5pm at the First Street location.
```

```
I want to book a table *clear previous dialog
```

```
I want a table for 3 please *clear previous dialog
```

The result should look something like this:



(Optional:) Backend Integration with Cloud Functions

(When Optional Step 70 has been done!)

In section 4 we will build a node.js server task to use the Watson Assistant skill built in Section 1. Watson Assistant also allows to call Cloud Functions (serverless computing) from within the context of a node definition.

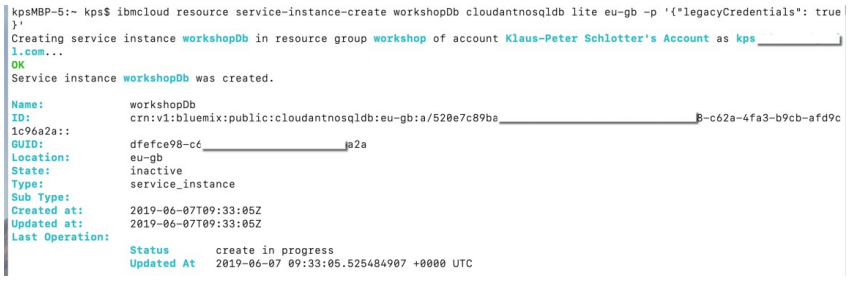
To show such an integration we store the restaurant table booking dialog result (Step 71) in a Cloudant database using a Cloud Function. This can be achieved without programming.

Step 72 Make sure you are logged in to your with the ibmcloud cli api to your account an respective region.

Step 73 In a terminal create a Cloudant service with the following command:

```
ibmcloud resource service-instance-create workshopDb cloudantnosqldb lite <your
region> -p '{"legacyCredentials": true}'

valid regions: eu-gb, eu-de, us-south, etc
```

> **Note:** Legacy credentials provide a password attribute for default behaviour. If you have an existing Cloudant service without legacy credentials see Step 85

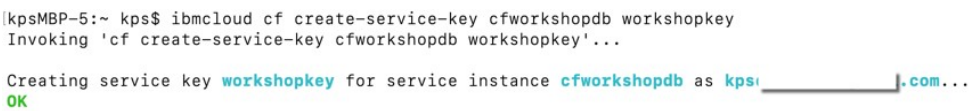


Step 74 Create a Cloud Foundry alias of your Cloudant services

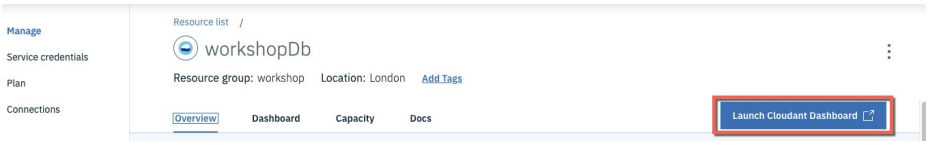
```
ibmcloud resource service-alias-create cfworkshopdb --instance-name workshopDb
```

Step 75 Create Credentials for this service with the following command

```
ibmcloud cf create-service-key cfworkshopdb workshopkey
```



Step 76 In the cloud console open the Dashboard of your Cloudant service



Step 77 At to top **click** Create Database . **Enter** a name f.e. *reservations* and **click** Create

Step 78 In a Terminal install the IBM Cloud Functions CLI plugin

```
ibmcloud plugin install Cloud-Functions -r bluemix
```

Step 79 With the following command you can automatically create Cloud Functions from certain IBM Services such as Cloudant, Weather Service, Watson Services to name a view.

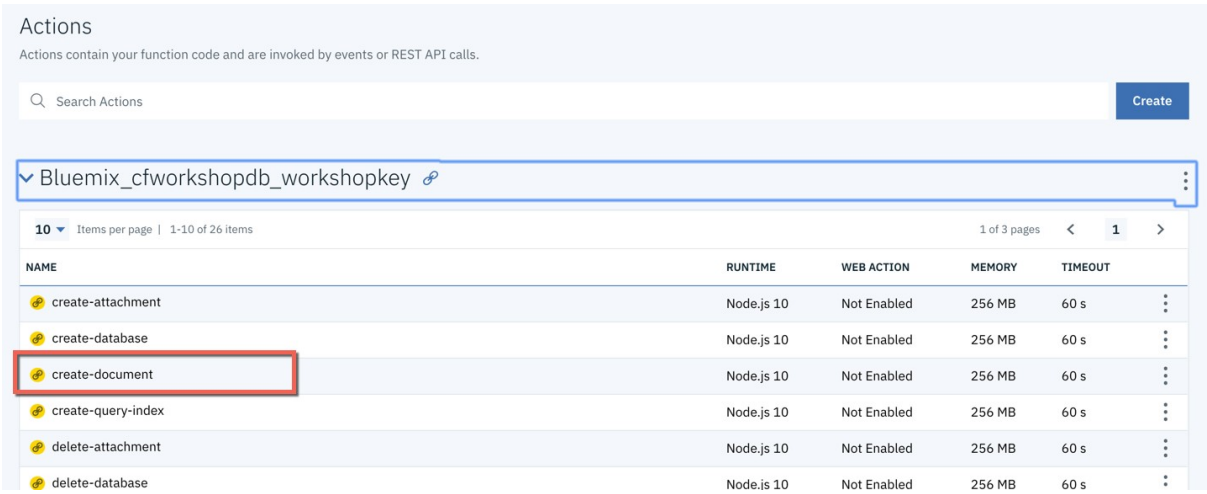
```
ibmcloud wsk package refresh
```

```
kpsMBP-5:~ kps$ ibmcloud wsk package refresh
'_' refreshed successfully
created bindings:
Bluemix_cfworkshopdb_workshopkey
updated bindings:
deleted bindings:
Bluemix_workshopDb_workshopkey
```

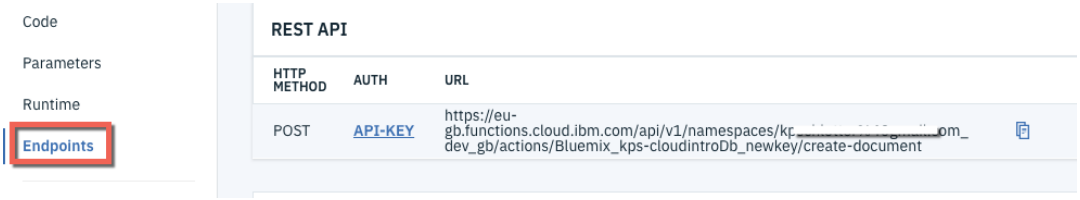
Step 80 In your cloud console goto Functions. Then **click** *Actions* in the left menu. You should see the functions that the previous command has created from some deployed services.

> **Note:** Make sure you have selected your ORG and SPACE at the top that you have a connection to from the Command Line Interface.

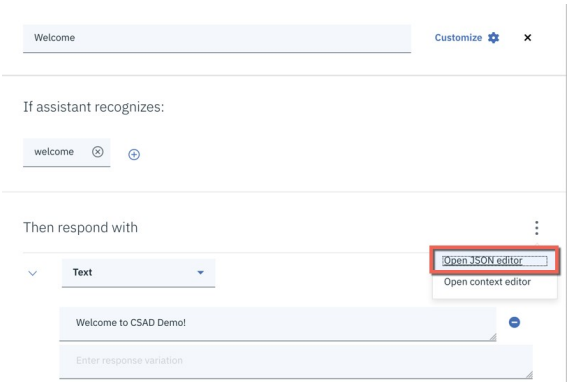
Step 81 We are interested in the *create-document* function of the Cloudant service. Click ⋮ → Manage Action



Step 82 **Select** *Endpoints* and **copy** the *URL* and the *API-KEY* for later use in our Assistant Skill. (click API-KEY to get the value).



Step 83 In your Assistant Skill **open** the *Welcome* node and the *Open JSON editor*. Replace the content with the following:

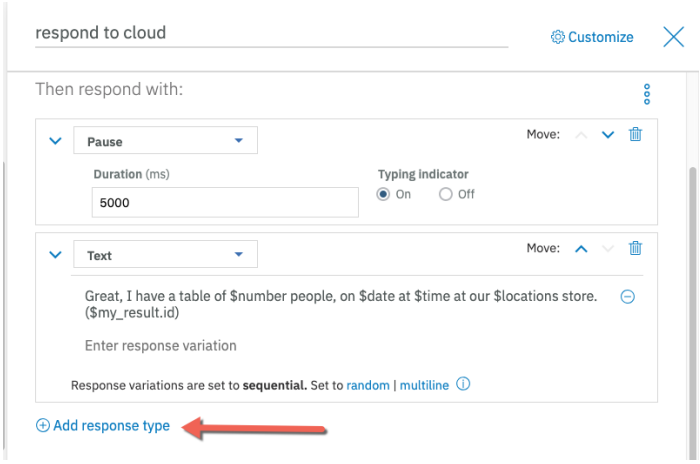


There is a codesnippets.txt file that contains all the code snippets of this lab section.

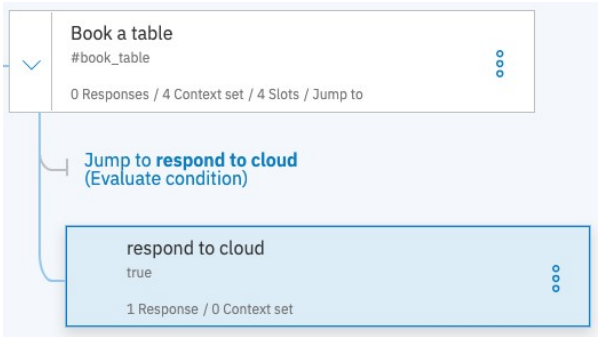
```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Welcome to CSAD Demo!"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  },
  "context": {
    "private": {
      "my_credentials": {
        "api_key": "<yourCloudFunctionsAPIKey>"
      }
    },
    "Alarmonoff": "off",
    "app_action": ""
  }
}
```

Step 84 **Click** ⋮ on the *Book a table* node and **select** *Add child node*.

Field	Value
Name this node…	Respond to cloud
If bot recognizes:	true
Then respond with (Pause)	5000
Then respond with (Text) (click + to add an response type)	Great, I have a table of $number people, on $date at $time at our $locations store. ($my_result.id)
And finally	Wait for user input



Step 85 **Click** ⋮ on the *Book a table* node and **select** *Jump to (recognizes condition)*. **Select** the *Respond to cloud node*.



Step 86 **Open** the *Book a table* node and on ⋮ *Open JSON editor* and **replace** the content with the following:

```
{
  "output": {
    "text": {
      "values": [],
      "selection_policy": "sequential"
    }
  },
  "actions": [
    {
        "name": "/kpschxxxx.com_dev_gb/actions/Bluemix_kps-cloudin-
troDb_newkey/create-document",
      "type": "server",
      "parameters": {
        "doc": {
          "date": "$date",
          "time": "$time",
          "number": "$number",
          "locations": "$locations"
        },
        "dbname": "reservations"
      },
      "credentials": "$private.my_credentials",
      "result_variable": "$my_result"
    }
  ]
}
```

Actions name is the part of the URL from Step 80 after .../namespaces.

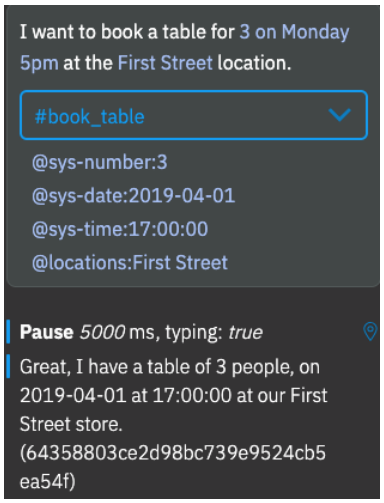
Note: In case you get an Error Code 400 you do not have legacy credentials, containing *password* and *port*, for your Cloudant service. Add the following two paramters from your Cloudant credentials to the code above:

```
"dbname": "reservations",
"iamApiKey": "<apikey from your Cloudant credentials>",
"url": "<url from your Cloudant credentials>"
```

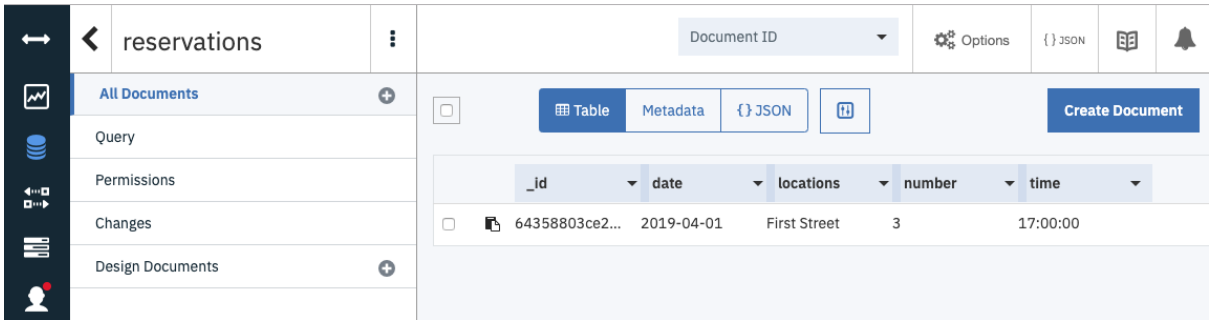
Step 87 Try the dialog with the following sentence:

```
I want to book a table for 3 on Monday 5pm at the First Street location.
```

An you should see a result like this:



Step 88 Go back to your Cloudant dashboard. There you should see the document that was created.

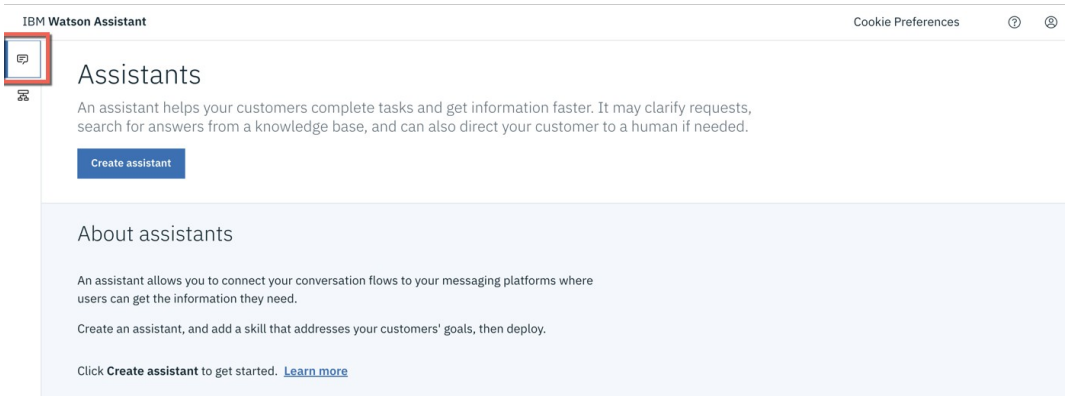


Section 2: Create an Assistant from our Skill

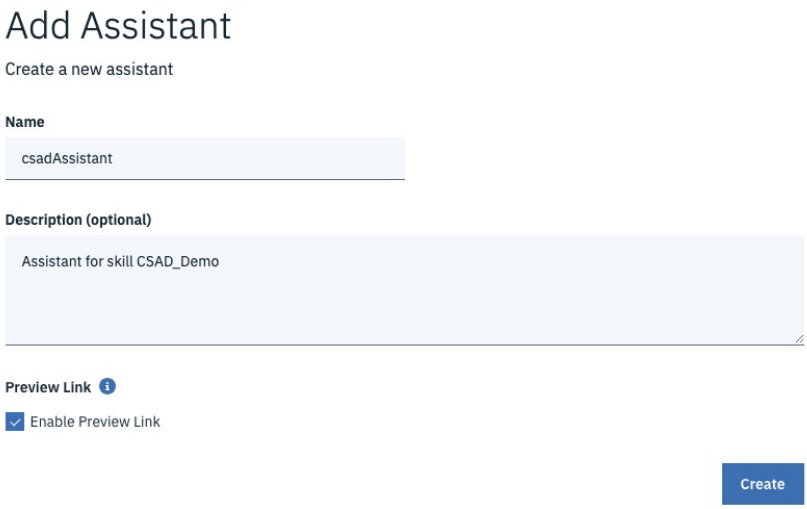
IBM Watson Assistant has a V2 API that provides a session context and therefore the context has not to be passed with any client request.

This is now also the API version used in the following Node.js application.

Step 89 Go to the Assistants section.



Step 90 **Click** Create assistant to create a new Assistant, **enter** name and description then **click** Create assistant.



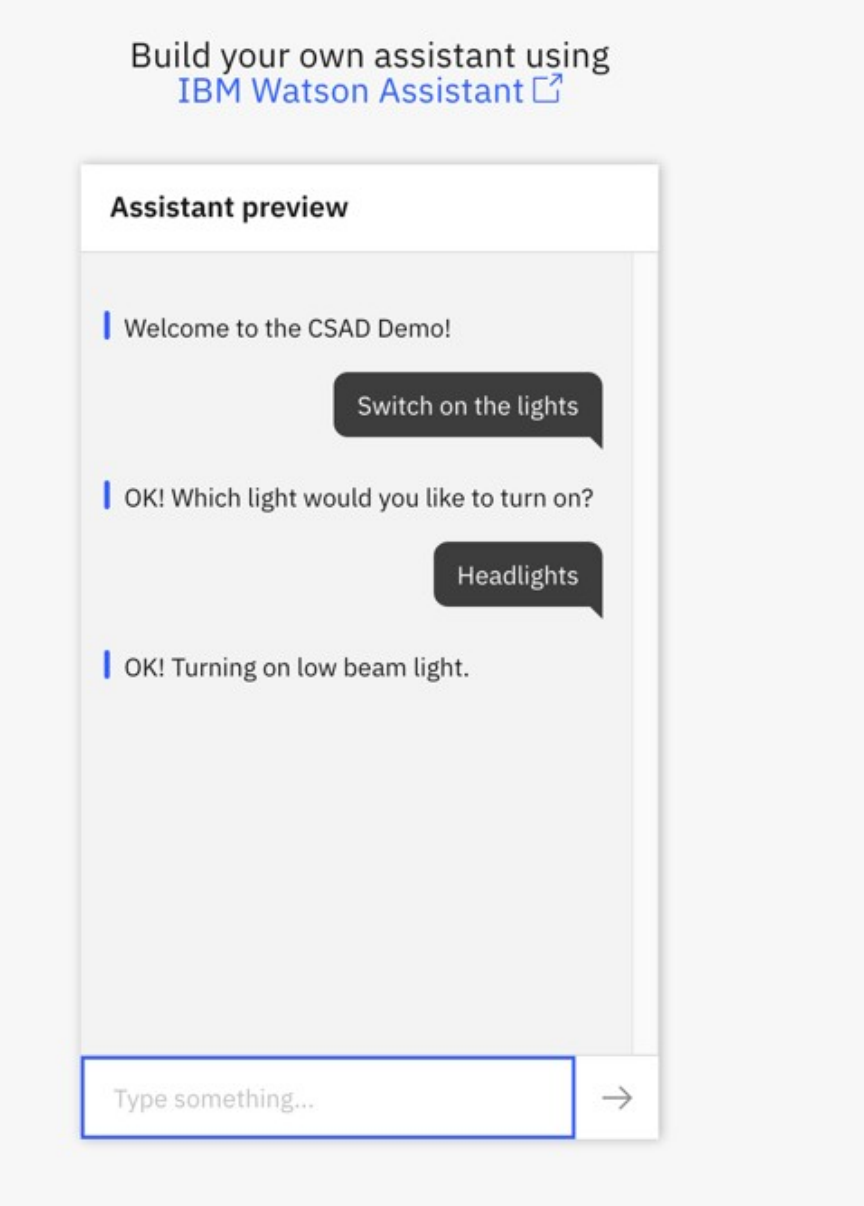
Step 91 Click Add dialog skill. and **select** our *CSAD Demo* Skill. With the Integrations function you can now directly integrate this Assistant into Facebook, Slack and other applications.

This is out of scopt of this introductory demo!!

Step 92 **Click** Preview Link to make this Assistant available on the internet. **Enter** a name and click Save Changes.

Step 93 **Copy** the link and **open** it in a browser.

Step 94 It should look like the following:

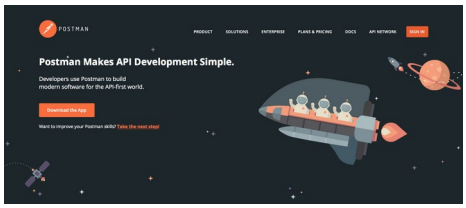


Step 95 Try some of your dialogs.

Section 3:(Optional) Test Watson Assistant (API V1) using Postman (REST Client)

The Watson Services on IBM Cloud have a REST (Representational State Transfer) interface and therefore they can be easily tested with a REST client like Postman. You can find the API reference in the Documentation of each service.

Step 96 Postman is available as an application for MacOS, Windows, and Linux. https://www.getpostman.com



Step 97 In the API reference of the IBM Cloud Watson Assistant service you can find an overview of the available API calls. The basic command is to list the your workspaces.

In Postman add the following **GET** request:

Url: https://<host of your service>/assistant/api/v1/workspaces?version=2019-02-28

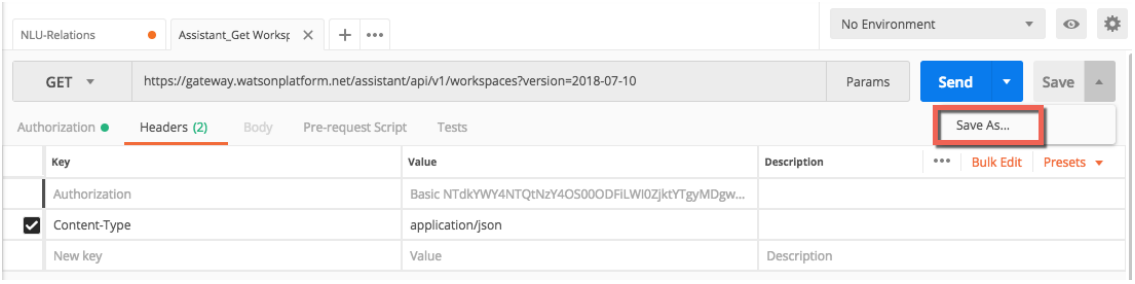
See your service credentials for the Url at your region. f.e. gateway-fra for Frankfurt.

Authorization Type: `Basic Auth` – Your credentials from Step 9

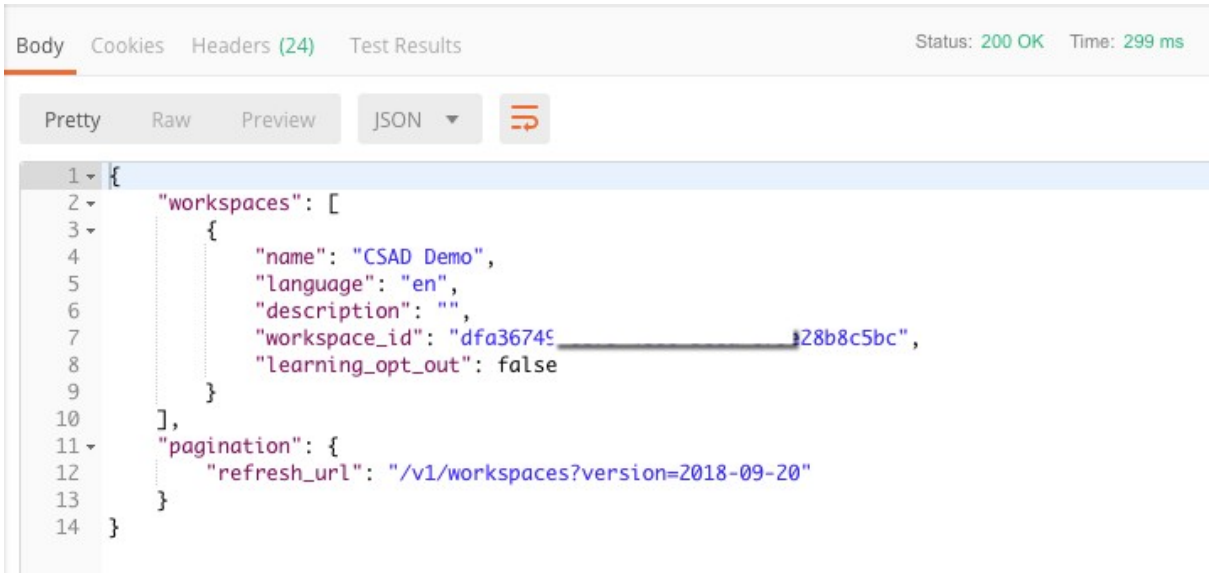
Note: The services in the IBM Cloud platform are currently transferred to an IAM authentication. So if your service was created with an **apikey** instead of **userid** and **password**, you specify the string **apikey** as userid and the **generated apikey** as password.

Headers: `Content-Type: application/json`

Note: You can save the request definition for further use by clicking **Save As.. Request Name**: `Conv_GetWorkspaces`, Collection: `Watson Services`.



Step 98 The result should look similar to this:



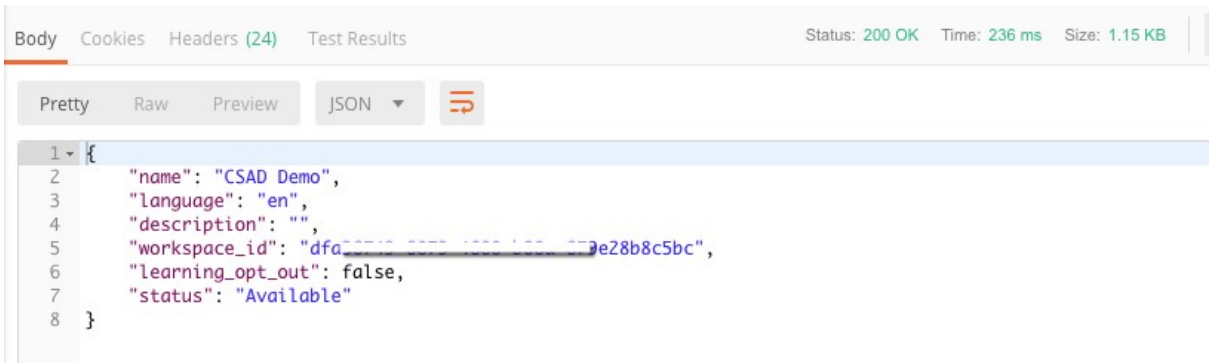
Step 99 **GET** the information from our **CSAD Demo** workspace.

Url: https://<host of your service>/assistant/api/v1/workspaces/<YOUR workspaceid from Step18>?version=2019-02-28

Authorization: as in Step 93

Headers: as in Step 93

Your result should look like this:



Step 100 Send (**POST** request) an initial message to start our conversation

Url: https://<host of your service>/assistant/api/v1/workspaces/<YOUR workspaceid from Step19>/message?version=2019-02-28

Authorization: as in Step 93

Headers: as in Step 93

Body: (raw)

```
{"input":
    {"text": ""},
    "context": {
        "conversation_id": "",
        "system": {
            "dialog_stack":[{"dialog_node":"root"}],
            "dialog_turn_counter": 0,
            "dialog_request_counter": 0
        }
    }
}
```

Your result should look like this:

Step 101 POST a request for #turn_on

Url: as in Step 93

Authorization: as in Step 93

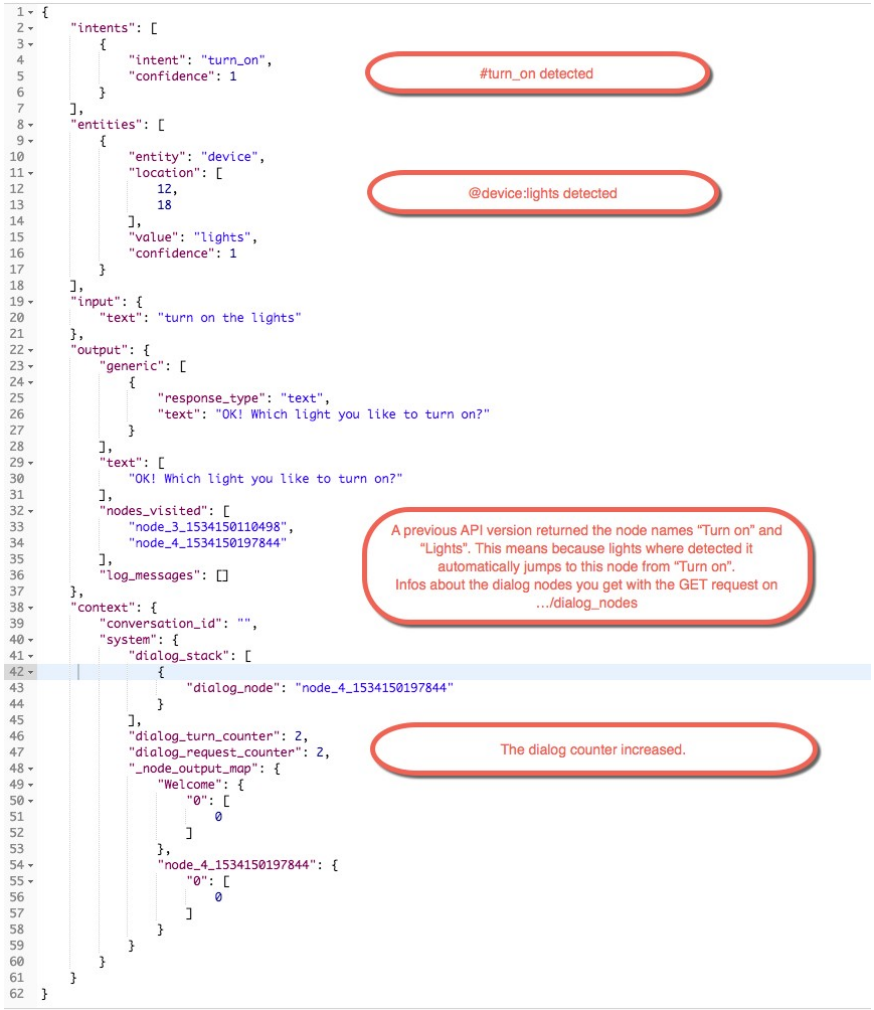
Headers: as in Step 93

Body: (raw) → Context from Step 96 Output.

```
{"input":
    {"text": "turn on the lights"},
    "context": {
        "conversation_id": "",
        "system": {
            "dialog_stack": [
                {
                    "dialog_node": "root"
                }
            ],
            "dialog_turn_counter": 1,
            "dialog_request_counter": 1,
            "_node_output_map": {
                "Welcome": [
                    0
                ]
            },
            "branch_exited": true,
            "branch_exited_reason": "completed"
        }
    }
}
```

Note: The context object is the memory of your conversation. For any further request always use the context object of the previous response. This enables the multi-session capability of the Watson Assistant service.

The output should look like this:




```
{
    "intents": [
        {
            "intent": "turn_on",
            "confidence": 1
        }
    ],
    "entities": [
        {
            "entity": "device",
            "location": [
                12,
                18
            ],
            "value": "lights",
            "confidence": 1
        }
    ],
    "input": {
        "text": "turn on the lights"
    },
    "output": {
        "generic": [
            {
                "response_type": "text",
                "text": "OK! Which light you like to turn on?"
            }
        ],
        "text": [
            "OK! Which light you like to turn on?"
        ],
        "nodes_visited": [
            "node_3_1534150110498",
            "node_4_1534150197844"
        ],
        "log_messages": []
    },
    "context": {
        "conversation_id": "",
        "system": {
            "dialog_stack": [
                {
                    "dialog_node": "node_4_1534150197844"
                }
            ],
            "dialog_turn_counter": 2,
            "dialog_request_counter": 2,
            "_node_output_map": {
                "Welcome": {
                    "0": [
                        0
                    ]
                },
                "node_4_1534150197844": {
                    "0": [
                        0
                    ]
                }
            }
        }
    }
}
```


With the following GET request in Postman

https://<your service host>/assistant/api/v1/workspaces/<your workspaceid>/dialog_nodes?version=2019-02-28

You can get the details of all the dialog nodes.

Step 102 You can proceed testing the service with other API calls.

Section 4: Create a Node.js Express application

Node.js is an open-source, cross-platform runtime environment for developing server-side web applications using JavaScript. Node.js has an event-driven architecture capable of asynchronous I/O utilizing callbacks. Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It's ability to facilitate rapid development of Node based web applications makes it the de facto framework for Node.js.

The completed lab can be cloned or downloaded from
`git clone https://github.com/iic-dach/speech-assistant.git`

There is a codesnippets.txt file that contains all the code snippets of this lab section.

Running the application locally

Step 103 In a Terminal window **create a folder** for your project f.e. `speech-assistant`

Step 104 Move into this folder and execute the following command

```
npm init
```

Specify a *name*, *description* and *author* when prompted, keep the other *defaults*. The result should be similar to the following in *package.json*:

```
{
  "name": "speech-assistant",
  "version": "1.0.0",
  "description": "IBM Watson Speech/Assistant",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "IBM Ecosystem Advocacy Group (EAG)",
  "license": "ISC"
}
```

Step 105 Adjust the package.json file with the **blue** content (see above):

Step 106 Open your project folder in Visual Studio code or you can start Visual Studio Code with the command "`code .`" from within the project folder. In the menu **select** *View → Integrated Terminal*

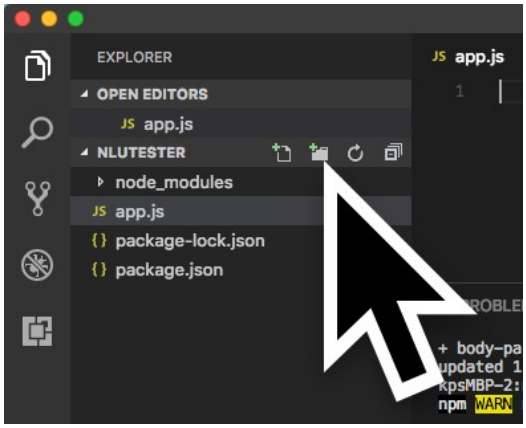
Step 107 In the *Intergrated Terminal* or another terminal execute the following commands to install the additional node modules needed for our project

```
npm install --save express ejs morgan body-parser ibm-watson
```

The modules will be stored in the node_modules folder and your *package.json* file should now have a dependencies section.

Step 108 **Create** *config.js* in the project root folder and enter your service credentials.

You can create files and folders for your project in the *Visual Studio Code*



Explorer on the left.

```
const config = {
  watson: {
    assistant: {
      username: "<yourServiceUsername>",
      password: "<yourServicePassword>",
      version: "2019-02-28",
      url: "yourServiceUrl",
      assistantId: "<yourAssistantId>"
    }
  }
};
module.exports = config;
```

```
const config = {
  watson: {
    assistant: {
      iam_apikey: "<yourApiKey>",
      version: "2019-02-28",
      url: "yourServiceUrl",
      assistantId: "<yourAssistantId>"
    }
  }
};
module.exports = config;
```

Depending on your service, use *username/password* or *apikey*.

Step 109 In the project folder *create* an *app.js* file.

Step 110 In app.js enter the following code to import the dependencies and instantiate the variables. **Save** and **close** the file.

```
const path = require('path');

const express = require('express');
const bodyParser = require('body-parser');
const logger = require('morgan');

const app = express();
app.use(bodyParser.json({ limit: '1mb'}));
app.use(express.static(path.join(__dirname, 'public')));
app.use(logger('dev'));

const port = process.env.PORT || 3000;
app.set('view engine', 'ejs');
app.set('views', 'views');

const watsonRoutes = require('./routes/watson');
app.use(watsonRoutes);

app.use(function (requst, response) {
  response.status(404).render("404");
});

app.listen(port, () => {
  console.log('Express app started on port ' + port);
})
```

Basic node.js setup with bodyParser for json format and the definitions of ejs as the template engine.

Step 111 In the project *root* **create** a *folder* named *routes* and in there **create** a *file* named *watson.js* with the following content: → (routes/watson.js). **Save** and **close** the file.

```
const express = require('express');

const watsonController = require('../controllers/watson');

const router = express.Router();

router.get('/', watsonController.getIndex);

router.post('/', watsonController.postMessage);

module.exports = router;
```

The router defines the urls we accept from the web with an appropriate function called that is defined in the controller file. Actually the getIndex() renders the Index page and postMessage() is a pure api function just returning json.

Step 112 In the project *root* **create** a *folder* named *controllers* and in there **create** a *file* named *watson.js* with the following content: → (controllers/watson.js)

```
const AssistantV2 = require('ibm-watson/assistant/v2');
const { IamAuthenticator } = require('ibm-watson/auth');
const config = require('../config');

const watsonAssistant = new AssistantV2({
  version: config.watson.assistant.version,
  authenticator: new IamAuthenticator({
    apikey: config.watson.assistant.iam_apikey
  }),
  url: config.watson.assistant.url
});

let assistantSessionId = null;

exports.getIndex = (req, res, next) => {
  watsonAssistant.createSession({
    assistantId: config.watson.assistant.assistantId
  })
  .then(result => {
    assistantSessionId = result.result.session_id,
    res.render('index');
  })
  .catch(err => {
    console.log(err);
  })
}

exports.postMessage = (req, res, next) => {
  let text = "";
  if (req.body.input) {
    text = req.body.input.text;
  }
  watsonAssistant.message({
    assistantId: config.watson.assistant.assistantId,
    sessionId: assistantSessionId,
    input: {
      'message_type': 'text',
      'text': text
    }
  })
  .then(assistantResult => {
    console.log(JSON.stringify(result, null, 2));
    res.json(assistantResult);  // just returns what is received from assistant
  })
  .catch(err => {
    res.status(404).json(err);  // after 5 min of inactivity the session is espired
  })
}
```

getIndex() creates a session with the assistant and returns the compiled index.ejs, see next steps, to the client. All client logic is defined as AJAX functions in the public/js/scripts.js file.

Note: The session object has a timeout defined in the assistant's settings. The lite version of the assistant has a maximum of 5 minutes. When the chat is inactive for more than five minutes you get an **error** in postMessage().

The postMessage() function just returns the information from the Assistant service to the client (see the console log), without any interaction. Our help request for the time will not yet be answered.

Step 113 In the root folder **create** a folder named *views*.

Note: The **ejs** view engine controls the layout with code indent. For a correct layout you have to adjust this manually or copy the code from the code snippets file.

Step 114 In the *views* folder **create** a *header.ejs* file with the following content:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>EAG Watson Assistant Lab</title>
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boot-
strap/4.1.3/css/bootstrap.min.css" crossorigin="anonymous">
<link rel='stylesheet' href='/stylesheets/styles.css' />
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/js/
bootstrap.min.js" crossorigin="anonymous"></script>
<script src="/js/scripts.js"></script>
</head>
<body onLoad="init()">
<div class="container" style="text-align: center"><body
class="container">
<div class="row"><div class="col-md-12"><p><h2>IBM EAG Watson As-
sistant/Speech Lab</h2></p></div></div>
```

Step 115 In the *views* folder **create** a *footer.ejs* file with the following content:

```
      <p><small>IBM Ecosystem Advocacy Group - 2010</small></p>
    </div>
  </body>
</html>
```

Step 116 In the *views* folder **create** a *404.ejs* file with the following content:

```
<% include header %>
<h2>404! Page not found.</h2>
<% include footer %>
```

Step 117 In the view folder create an index.ejs file with the following content:

```
<%- include ("header") %>
<p/>
<div class="row">
  <div class="col-md-3"></div>
  <div class="input-group col-md-6">
    <input type="text" id="text" name="text" class="form-control" placeholder="Enter
text sent to Watson">
    <span class="input-group-btn">
      <button class="btn btn-primary" onclick="sendMessage()" >Send</button>
    </span>
  </div>
</div>
<div class="row">
  <div class="col-md-3"></div>
  <div class="col-md-6 mt-2">
    <div><b>Conversation History:</b></div>
    <div id="history" class="form-control col text-left" ></div>
  </div>
</div>
<%- include ("footer") %>
```

Step 118 To add some styling to the webpage and additional javascripts **add a folder** named *public* to the project root folder.

In app.js this folder was defined to be static and is therefore available to the rendered html page.

Step 119 In this *public* folder **add** the following two folders:

`stylesheets` → public/stylesheets
`js` → public/js

Step 120 In the *stylesheets* folder **add a file** *styles.css* with the following content:

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}
a {
  color: #00B7FF;
}
.form-control {
  margin-right: 5px;
}
.wa-user {
  border-radius: 25px;
  background: #F0E68C;
  padding: 10px;
  width: 90%;
  text-align: right;
  margin-bottom: 5px;
  float: right;
}
.assistant {
  border-radius: 25px;
  background: #D3D3D3;
  padding: 10px;
  width: 90%;
  margin-bottom: 5px;
  float: left;
}
.histLabel {
  margin-bottom: 5px;
}
```

Step 121 In the *js* folder **add a file** *scripts.js* with the following content:
The context variable was used to handle Assistant API V1 requests.
(See previous versions of this document).

```
var context = {};
function updateChatLog(user, message) {
  if (message) {
    var div = document.createElement("div");
    if (user === 'Watson') {
      div.className = 'assistant';
    } else {
      div.className = 'wa-user';
    }
    div.innerHTML = "<b>" + user + ":</b> " + message + "<br/>";
    document.getElementById("history").appendChild(div);
    document.getElementById("text").value = "";
  }
}
function sendMessage() {
  var text = document.getElementById("text").value;
  updateChatLog("You", text);
  var payload = {};
  if (text) {
    payload.input = {"text": text};
  };
  if (context) {
    payload.context = context;
  };
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
      if (xhr.status == 200) {
        var json = JSON.parse(xhr.responseText);
        context = json.context;
        updateChatLog("Watson", json.result.output.generic[0].text);
      } else { // the session may be expired
        var json = JSON.parse(xhr.responseText);
       var errText = json.code + " - " + json.message + " - Reload page for new ses-
sion (timeout 5 min)";
        updateChatLog("Watson", errText);
      }
    }
  }
  xhr.open("POST", "./", true);
  xhr.setRequestHeader("Content-type", "application/json");
  xhr.send(JSON.stringify(payload));
}

function init() {
  document.getElementById("text").addEventListener("keydown", function(e) {
    if (!e) {
      var e = window.event;
    }
    if (e.keyCode == 13) {
      sendMessage();
    }
  }, false);
  sendMessage();
}
```

Step 122 Save all files.

Test the application

> Your demo application showcasing IBM Watson cognitive services is now ready! Take a deep breath and cross your fingers because it's time to test the application.

Step 123 Return to the terminal/command window. If the server is still running, enter **Ctrl-C** to kill the application so that your latest code changes will be loaded.

Step 124 Test the application on your local workstation with the following command(s).

```
npm start
```

Step 125 In a web browser, navigate to the following URL. If necessary, refresh the existing page to pick up the changes.

http://localhost:3000/

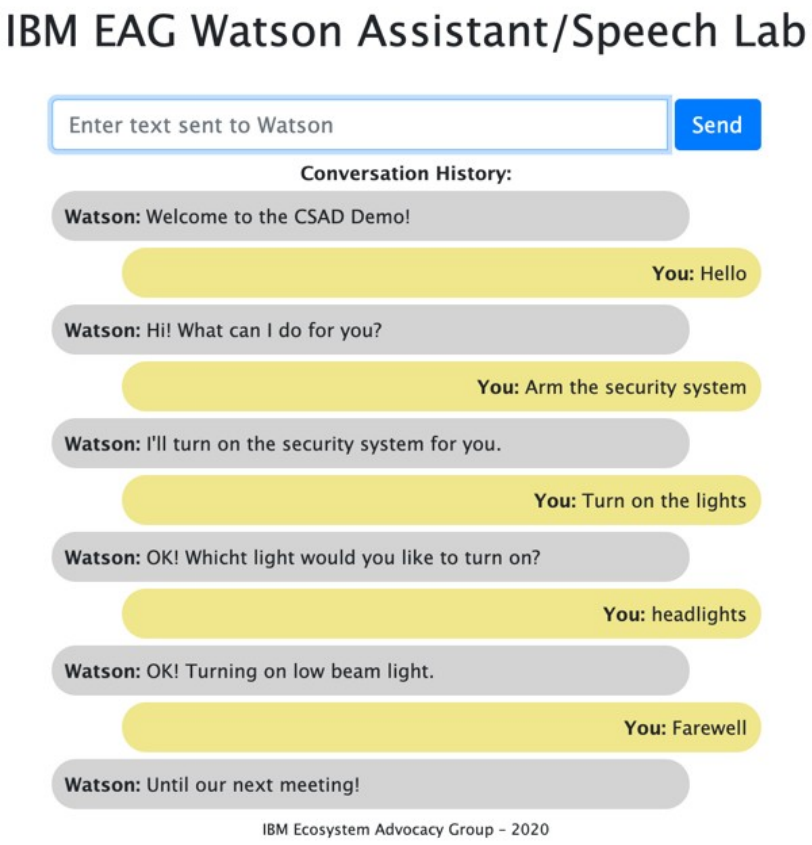
Step 126 You should see your IBM Watson Conversation application running! Notice that your client application has already contacted Watson and initiated a new conversation. As you'll recall, you can review the server console to see the JSON returned by the Conversation service.

The session expires after 5 minutes of inactivity

Step 127 Enter the following messages and observe Watson's responses in the Conversation History. The responses will be the same as those you saw earlier when performing tests in the Conversation tool web application. The difference this time is that you are using the API to embed this conversation into an external application!

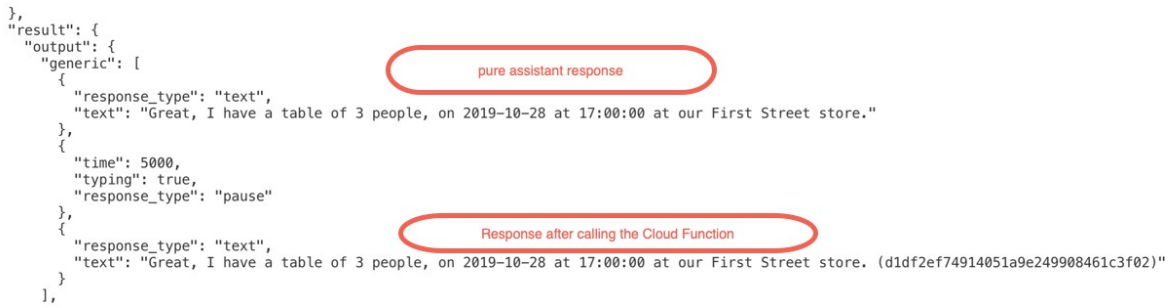
```
Hello

Arm the security system

Turn on the lights

The headlights

Farewell
```

Step 128 Feel free to experiment with other phrases to further explore the dialog branches that you created in earlier steps.



Step 129 At this point, you should review the Node.js server console and the developer tools of the browser to see the full JSON object that was returned. There's more data there than we displayed in this demo ☺

F.e try the book_table intent and see in the nodejs console what type of return messages you get from the assistant.



Step 130 The node.js application we have created so far does nothing more than receiving user input, send it to the Watson Assistant service and return the received text to the user. In the next section we can see how further back-end integration can be achieved.

This is basically the same functionality the preview link of the assistant provides.

Process user input to detect intents and perform app actions

IBM Watson Assistant is a service that uses natural language processing to detect the intent of a user's input and return a response. In simple chat apps, the response returned from dialog nodes can be sufficient. However, in most cases, you will need to perform some actions based off the user input, such as query a database, update sales records, etc. To do this, you will need a proxy application that communicates with the end user, your backend systems, and the IBM Watson Conversation service. In this section, you will learn how to detect intents from user input.

An alternative to this approach is using **Cloud Function** or **Web Hooks** for integration but for heavy duty tasks I prefer the technique showed here.

Step 131 The simple application (created in this guide) passes the user supplied input to the Watson Assistant service. Once Watson analyzes the input and returns a JSON response, that response is passed back to the client application. In this section, you will modify the code to detect specific intents so that you could perform various actions in your own system(s).

Step 132 Locate the following file and open it with a text/code editor.

`…/speech-assistant/controllers/watson.js`

Step 133 If you accurately followed the steps in this guide, lines 41 – 44 should contain the application logic that is executed after Watson returns a response. In the .then() block **Insert a blank line so that line 42 is now blank**. This will be the insertion point for your new code snippet. You should also **comment out line 17** since you will be using the server console to display updates from your custom app actions. Your code should now look as follows:

```
33    watsonAssistant.message({
34      assistantId: config.watson.assistant.assistantId,
35      sessionId: assistantSessionId,
36      input: {
37        'message_type': 'text',
38        'text': text
39      }
40    })
41    .then(result => {
42
43      // console.log(JSON.stringify(result, null, 2));
44      res.json(result);
45    })
46    .catch(err => {
47      console.log(err);
48    })
49  }
```


Step 134 At line 42, enter the following code to identify the intent and entity returned by Watson and perform some custom actions. You will also update the JSON response object to include a new message that should be sent to the end user with your custom information.

```
if (text === "") {
  return res.json(assistantResult);
}
console.log("Detected input: " + text);
if (assistantResult.result.output.intents.length > 0) {
  var intent = assistantResult.result.output.intents[0];
  console.log("Detected intent: " + intent.intent);
  console.log("Confidence: " + intent.confidence);
}
if (assistantResult.result.entities.length > 0) {
  var entity = assistantResult.result.output.entities[0];
  console.log("Detected entity: " + entity.entity);
  console.log("Value: " + entity.value);
  if ((entity.entity === 'help') && (entity.value === 'time')) {
    var msg = 'The current time is ' + new
Date().toLocaleTimeString();
    console.log(msg);
    assistantResult.result.output.generic[0].text = msg;
  }
}
```

Step 135 Refer to the following table for a description of the code:

Lines	Description
42 – 43	Return immediately when no input text
45	The user input that was passed to the Assistant service
46 – 50	If the Assistant service identified an intent from the input… 48-49: Get the *intent* and print its name and confidence level.
51 – 59	If the Assistant service identified an entity from the input… 52-54: Get the *entity* and print its name and value 55: Check for a specific entity so that your app can respond accordingly 56-57: Get and print the current system time 58: Modify the Watson output with your new custom app response
62	Send the modified JSON response to the client application

Step 136 On line 49, you obtain and print IBM Watson's confidence level. This value is intended to be an indicator of how well Watson was able to understand and respond to the user's current query. If this value is lower than you are comfortable with (let's say < 60%), you could abort the operation and ask the user for clarification. Watson will rarely provide an answer with a 100% confidence level.

Step 137 **Save** and close the file.

Step 138 Return to the terminal/command window. If the server is still running, enter **Ctrl-C** to kill the application so that your latest code changes will be loaded.

Step 139 Test the application on your local workstation with the following command(s).

```
npm start
```

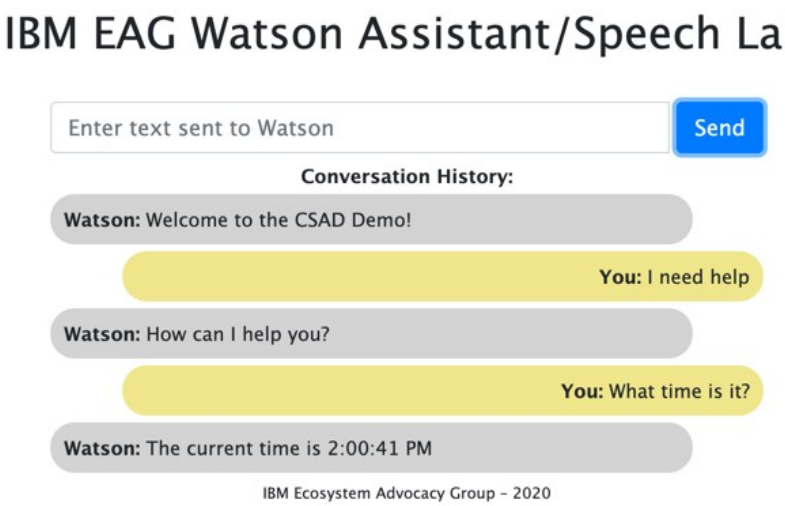
Step 140 You should see some output indicating the application is running. In a web browser, navigate to the following URL. If necessary, refresh the existing page to pick up the changes.

http://localhost:3000/

Step 141 Test the application using the following inputs. After typing each line, review the results on the server console to see your custom app actions based off specific *intents* and *entities*.

```
Hello

I need some help.

What time is it?
```

Step 142 In the screenshots below, the custom application will output messages to the server console based off the returned intent. For example:




```
Detected entity: help
Value: time
The current time is 2:00:41 PM
```


(Optional) Deploy the application to IBM Cloud

In this demo exercise, you have been working locally and running your Node.js server on your own workstation. You can very easily push your application to the IBM Cloud environment and make it publicly accessible to customers, partners, friends, and/or family!

Step 143 Create a manifest.yml file in the project folder with the following content: Under *services* use the name of you Conversation service created in **Step 7**. As name you should specify a unique name (or the *random-route* option below the name statement). Memory 256M if you use a *Lite* account

```
random-route: true
```

```
applications:
- name: xxx-speech-assistant
  path: .
  buildpack: sdk-for-nodejs
  command: node app.js
  memory: 256M
```

Step 144 **Save** the manifest.yml file.

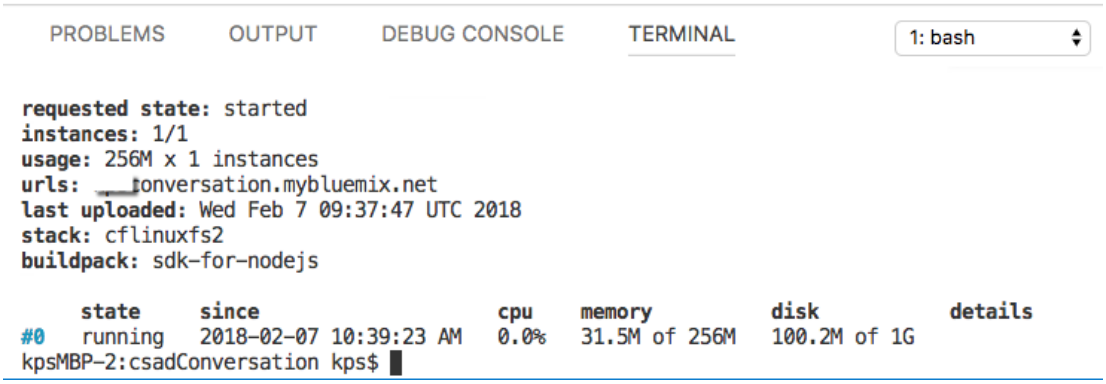
Step 145 Stop your running cloud applications that you do not exceed the memory limit of the IBM Cloud “Lite” account.

Step 146 Make sure you are logged in to your IBM Cloud account.
(**See Workstation Setup document Step 9**)

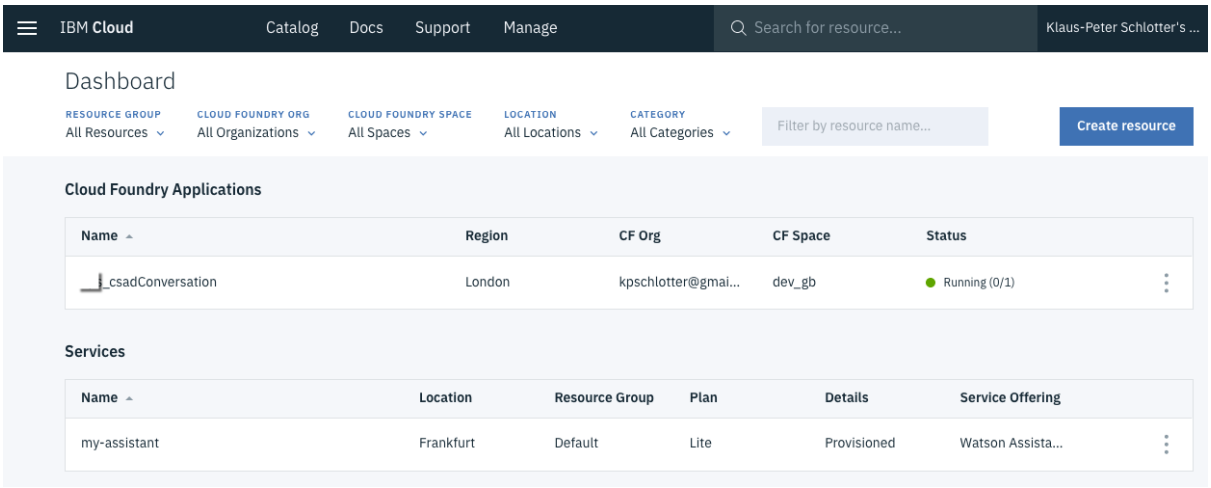
Step 147 In a terminal windows in the project folder execute the command

```
ibmcloud app push
```

After completion you should see a message that the application is running on the IBM Cloud.



Step 148 You will also see the application in your IBM Cloud console.



Enable Speech for the Watson Assistant

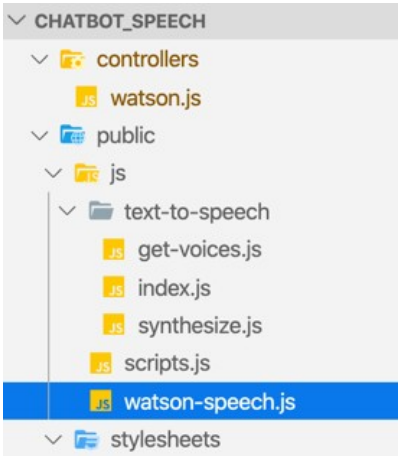
Step 149 Create a Watson Speech-To-Text and a Watson Text-To-Speech service as you did for the Watson Assistant in **Section 1 Step 1** – **Step 9**.

Step 150 Install the client helpers Watson-Speech with the following command in the root of the project.

```
npm install --save watson-speech
```

This plugin can be used directly in the server part (watson.js) but the client side has to be copied into the folder *public/js*.

Step 151 **Go** to the *node_modules/watson-speech/dist* folder and **copy** the file *watson-speech.js* **into** the *public/js* folder and also the complete *text-to-speech* folder.



Step 152 Add additional buttons to the index.ejs file. See the completed file:

```
<%- include ("header") %>
<p/>
  <div class="row">
    <div class="col-md-3"></div>
    <div class="col-md-6 d-flex justify-content-between buttonRow">
      <div class="form-check" style="float: right;text-align: right;">
        <input type="checkbox" class="form-check-input" id="checkSpeech"
checked>
        <label class="form-check-label" for="checkSpeech">Auto Text-To-
Speech</label>
      </div>
    </div>
  </div>
  <div class="row">
    <div class="col-md-3"></div>
    <div class="col-md-6 d-flex justify-content-between buttonRow">
      <button id="btnSpeech" class="btn btn-primary" onclick="getSpeech()"
>Speech Input</button>
      <button class="btn btn-primary text-right" onclick="sendMessage()"
>Send to Assistant</button>
    </div>
  </div>
  <div class="row">
    <div class="col-md-3"></div>
    <div class="input-group col-md-6">
      <input type="text" id="text" name="text" class="form-control"
placeholder="Enter text sent to Watson">
    </div>
  </div>
  <div class="row">
    <div class="col-md-3"></div>
    <div class="col-md-6 mt-2">
      <div class="histLabel"><b>Conversation History:</b></div>
      <div id="history" class="text-left" ></div>
    </div>
  </div>
<%- include ("footer") %>
```

If you do not want the checkbox to be selected, just remove the `checked` option.

Step 153 **Add** the following style to *styles.css*

```
.histLabel {
  margin-bottom: 5px;
}
.buttonRow{
margin-bottom: 5px;
}
```

Step 154 In header.ejs **add** the *Watson-speech* client script above script.js

```
<script src="/js/watson-speech.js"></script>
```

Step 155 In scripts.js add the following function to get the speech input at the bottom of the file: (called by the Speek button)

```
// Grab input from microphone and transform it to text
function getSpeech() {
  var btn = document.getElementById("btnSpeech");
  btn.className = 'btn btn-danger';
  fetch('/api/speech-to-text/token')
    .then(function(response) {
      return response.json();
    })
    .then(function (token) {
      var stream =
WatsonSpeech.SpeechToText.recognizeMicrophone(Object.assign(token, {
        model: 'en-US_BroadbandModel',
        outputElement: '#text' // CSS selector or DOM Element
      }));
      stream.on('data', function(data) {
        if(data.results[0] && data.results[0].final) {
          stream.stop();
          btn.className = 'btn btn-primary';
          console.log('stop listening.');
        }
      });
      stream.on('error', function(err) {
        console.log(err);
      });
    }).catch(function(error) {
      console.log(error);
    });
}
```

Step 156 In scripts.js add the following function to for text-to-speech at the bottom of the file: (Called when Watson replies).

```
  // Automatically speek, when there is an result from Watson Assistant
  function speak(message) {
    fetch('/api/text-to-speech/token')
      .then(function(response) {
        return response.json();
      })
      .then(function(token) {
        const audio = WatsonSpeech
         .TextToSpeech.synthesize(Object.assign(token, {
            text: message
         }));
         audio.onerror = function(err) {
           console.log('audio error: ', err);
         };
      });
  }
```

Step 157 In scripts.js after updateChatLog() call the speek() function when the checkbox is checked. On line 32 and line 38.

```
if (document.getElementById('checkSpeech').checked)
  speak(json.result.output.generic[0].text);
```

```
26  xhr.onreadystatechange = function() {
27    if (xhr.readyState == 4) {
28      if (xhr.status == 200) {
29        var json = JSON.parse(xhr.responseText);
30        context = json.context;
31        updateChatLog("Watson", json.result.output.generic[0].text);
32        if (document.getElementById('checkSpeech').checked)
33            speak(plainText);
34      } else { // the session may be expired
35        var json = JSON.parse(xhr.responseText);
36       var errText = json.code + " - " + json.message + " - Reload page for new session (timeout 5 m
37        updateChatLog("Watson", errText);
38        if (document.getElementById('checkSpeech').checked)
39            speak(plainText);
40      }
41    }
42  }
```

Step 158 In routes/watson.js add the following on line 11 and line 13 (before module.exports)

```
router.get('/api/speech-to-text/token', watsonController.getSttToken);

router.get('/api/text-to-speech/token', watsonController.getTtsToken);
```

Step 159 In controllers/watson.js on line 3 import the TokenManager

```
const { IamAuthenticator, IamTokenManager } = require('ibm-watson/auth');
```

Step 160 On line 28 add the following functions:

```
const sttAuthenticator = new IamTokenManager({
  apikey: config.watson.speechToText.iam_apikey
});

const ttsAuthenticator = new IamTokenManager({
  apikey: config.watson.textToSpeech.iam_apikey
});
```

Step 161 At the bottom of the file add the following two function

```
exports.getSttToken = (req, res, next) => {
  return sttAuthenticator
    .requestToken()
    .then(({ result }) => {
      res.json({ accessToken: result.access_token, url:
        config.watson.speechToText.url });
    })
    .catch(console.error);
}

exports.getTtsToken = (req, res, next) => {
  return ttsAuthenticator
    .requestToken()
    .then(({ result }) => {
      res.json({ accessToken: result.access_token, url:
       config.watson.textToSpeech.url });
    })
    .catch(console.error);
}
```

Step 162 Update the config.js with the TTS and STT credentials

```
var config = {
  watson: {
    assistant: {
      iam_apikey: "<yourApiKey>",
      version: "2019-02-28",
      url: "yourServiceUrl",
      assistantId: "<yourAssistantId>"
    },
    speechToText: {
      url: "<yourServiceUrl>",
      iam_apikey: "<yourApiKey>"
    },
    textToSpeech: {
      url: "<yourServiceUrl>",
      iam_apikey: "<yourApiKey>"
    }
  }
};
```

Step 163 Save all files and start the app with

```
npm start
```

You can input your text with speech and when the bot answers and the checkbox is checked, the reply is spoken as well.

Section 5: Create a Discovery Service Instance

More details on the Discovery Service you can see in the document ***Lab 05 Discovery***, where the following steps are copied from.

Step 164 Create a Discovery service instance as you did for the other Watson services.

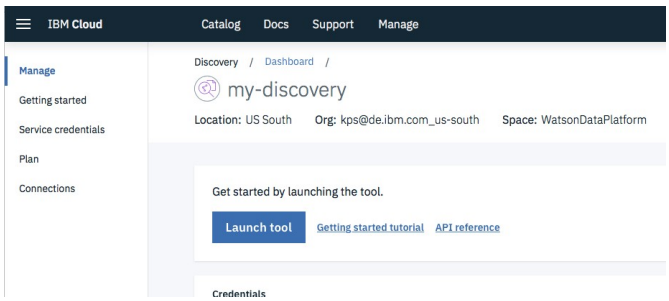
Section 6 Integrate Discovery into a Chatbot (API)

In this section we integrate the a Watson Discovery document collection via the Watson Discovery API.

Create a Watson Discovery Document Collection

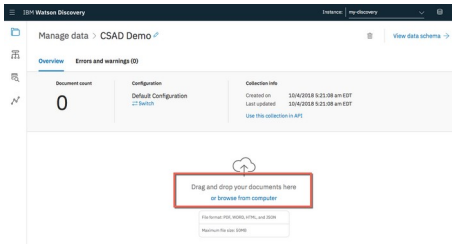
Step 165 Download the zip file manualdocs.zip from github (**click** the Download button) and extract it into a folder. It's a web manual for a Ford car.

Step 166 In your cloud console dashboard **click** on the "my-discovery" service created above and **click** the Launch tool button.

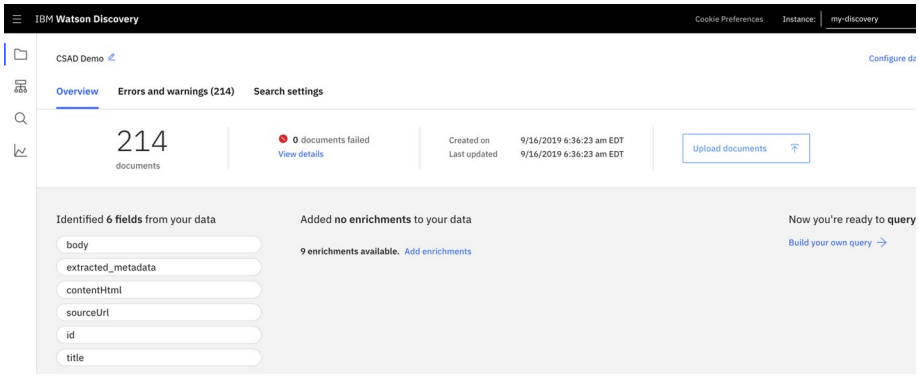


Step 167 On the *Manage data* page **click** Upload your own data .
As collection Name **enter** *CSAD Demo* and click Create.

Step 168 The new collection opens and you can either **browse** or **drag 'n' drop** all the documents extracted above (234 json files).

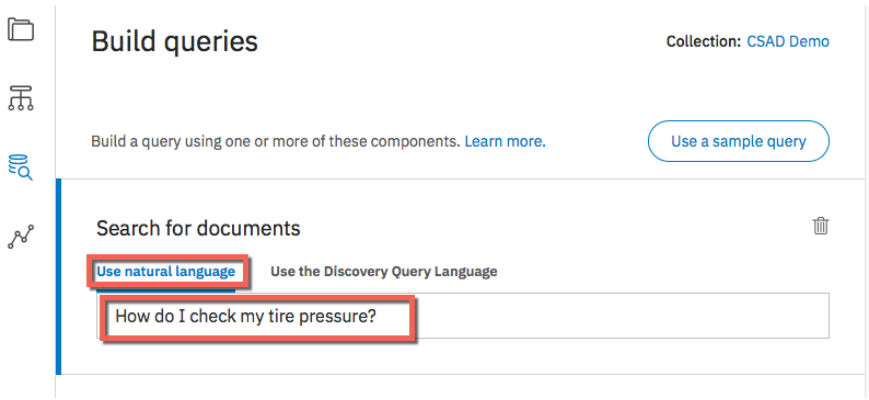


Step 169 The Discovery Service now indexes the documents. This may take a few minutes.



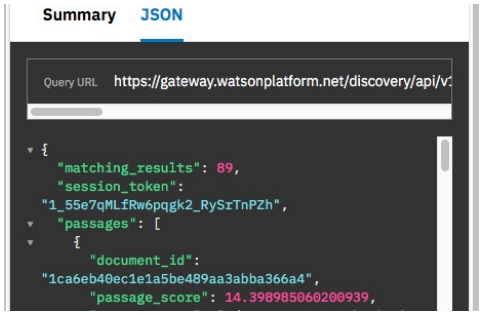
Step 170 On the left, click the button to open the *Build queries* page.

Step 171 **Enter** the following query: `How do I check my tire pressure?`



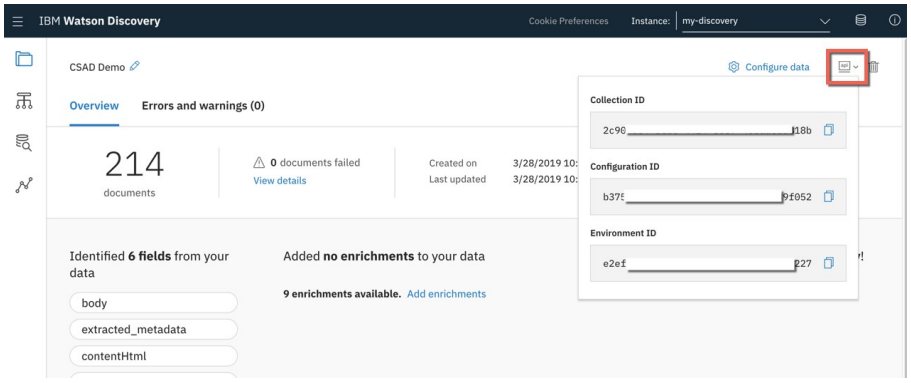
and press the Run query button.

On the left you can then see the results. Especially switch between the Summary and the JSON view of the results.



There is a lot more to explore here, but this is out of scope for this short tutorial. We just want to use the question submitted above later in our extended chatbot.

Step 172 On the top right of the *Overview* page **click** to show the ids you have to copy for later use by the application in addition to the service credentials.



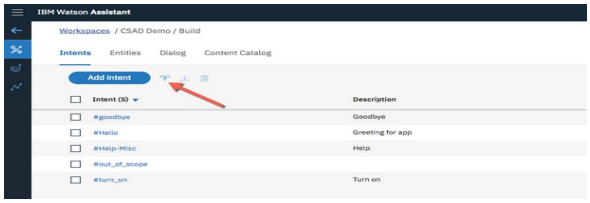
Update the Assistant Skill to integrate the Discovery Collection

Step 173 Download the out_of_scope_intents.csv from the Github repository to your project folder. Use the following in the terminal:

```
wget https://github.com/iic-dach/csadConversation/blob/master/resources/
Out_of_scope_intents.csv
```

Step 174 Go back to your Watson Assistant console that should still be open in the browser (Step 11)

Create a new intent *out_of_scope* and return to the list of intents. C**lick** the button to import the user examples from the file downloaded above.



131 examples will be imported.

Step 175 On the *Dialog* tab **click** the menu button on the *Turn on* node and **click** *Add node below*. **Name it** *Out of Scope*. In the field *If bot recognizes* **type** *#out_of_scope*.

Step 176 **Note:** Due to modularity we create the node in this location. Because this is also some kind of help question, it might be also appropriate, to insert this in the child tree of the *help* node.

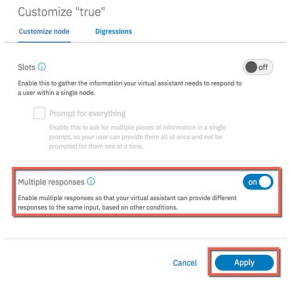
Step 177 Now **click** the menu button on the *#out_of_scope* node. And **click** *Add child node*.

Step 178 In this child node,**name** it *Check out of Scope,* in **If bot recognizes**, just **type** *true*.

Step 179 Now **click** the menu button on the *#out_of_scope* node. And **click** *Jump to (Recognizes Condition)*.

Step 180 **Click** the *Check out of Scope* node.

Step 181 **Click** Customize and **enable** *Multiple responses*. Then click Apply



Step 182 Now you can add more than one answer.

Add the following:

If bot recognizes	Respond with
intents[0].confidence>=0.94	That question is out of scope for this application, take a look at the Conversation Enhanced application to handle questions like these.
intents[0].confidence<0.94	Sorry I haven't learned answers to questions like this.

Step 183 Now click the Try it and **enter** the following question:

```
Hi, how do I check my tire pressure?
```

This returns the first answer because it matches an out_of_scope intent.

Step 184 Now enter another question:

```
How do I check my account balance?
```

This returns the second answer because it only matches an out_of_scope intent to some degree.

This is not a very good solution because we only check the natural language and we do not have any entities assigned here. It works when you enter

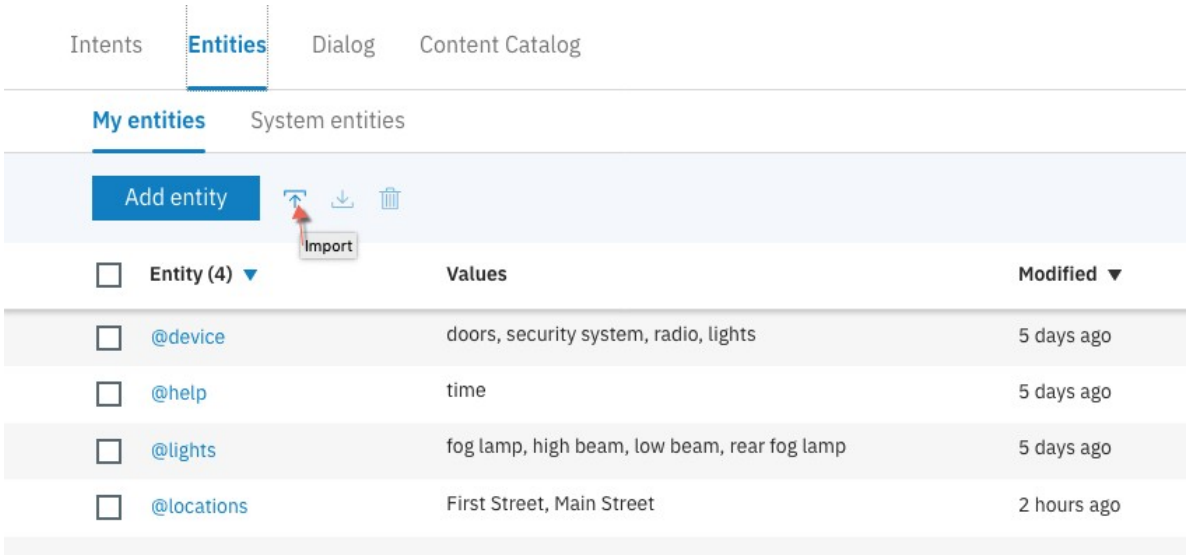
`How do I check my tire pressure` – Similar to `Hi, how do I check my tire pressure?`

`How do I check my blood pressure` Is very similar to the above but <94%

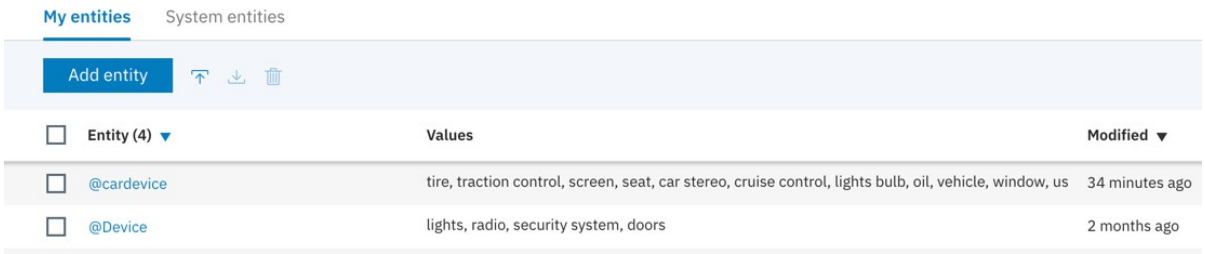
A solution would be just to add an entity @cardevice and add all the devices listed in the intent examples.

Step 185 In your Skill's Entities add an Entity by importing the cardevice_entity.csv CSV file.

```
wget https://github.com/iic-dach/csadConversation/blob/master/resources/
cardevice_entity.csv
```



This will add an Entity @cardevice



Step 186 Now change the *Check out of Scope* node (Step 183) to the following

If bot recognizes	Respond with
@cardevice	That question is out of scope for this application, take a look at the Conversation Enhanced application to handle questions like these.
true	Sorry I haven't learned answers to questions like this.

“true” always executes when the above condition is not met.

Step 187 The backend is now adjusted for integrating the Discovery service into our chatbot application.

Step 188 Update the application to integrate with the Discovery Service

Step 189 Open the cloned repository in Visual Studio Code or continue with the application completed in Lab 3.

Step 190 In *config.js* enter the credentials and IDs from your Discovery service.
(If you cloned the lab3 repository you make a copy of the *config_sample.js)*

You can cut and paste from the codesnippets.txt file.

```
var config = {
  watson: {
    assistant: {
      ...
    },
    discovery: {
      version: "2019-04-30",
      username: "<your discovery username>",
      password: "<your discovery password>",
      url: "<yourServiceUrl>"
    },
    discoveryEnv: {
      collectionId: "<your collectionId>",
      environmentId: "<your environmentId>"
    }
  }
};
module.exports = config;
```

```
var config = {
  watson: {
    assistant: {
      ...
    },
    speechToText: {
      ...
    },
    textToSpeech: {
      ...
    },
    discovery: {
      version: "2019-04-30",
      iam_apikey: "<yourApiKey>",
      url: "<yourServiceUrl>"
    },
    discoveryEnv: {
      collectionId: "<your collectionId>",
      environmentId: "<your environmentId>"
    }
  }
};
module.exports = config;
```

Depending on your service, use *username/password* or *apikey*.

Step 191 In controllers*/watson.js* **add** a blank line after line 1.

Step 192 In line 2 add the following – this imports the discovery api

```
const DiscoveryV1 = require('ibm-watson/discovery/v1');
```

Step 193 **Add** a blank line after line 13

Step 194 In line 14 add the following – this creates an instance of Discovery

```
const watsonDiscovery = new DiscoveryV1({
  version: config.watson.discovery.version,
  authenticator: new IamAuthenticator({
    apikey: config.watson.discovery.iam_apikey
  }),
  url: config.watson.discovery.url
});
```

Step 195 Replace postMessage() with the following: (use codesnippets.txt)

```
exports.postMessage = (req, res, next) => {
  let text = '';
  if (req.body.input) {
    text = req.body.input.text;
  }
  watsonAssistant.message({
    assistantId: config.watson.assistant.assistantId,
    sessionId: assistantSessionId,
    input: {
      'message_type': 'text',
      'text': text
    }
  })
  .then(assistantResult => {
    console.log(JSON.stringify(assistantResult, null, 2));
    if (text === '') {
      return res.json(assistantResult);
    }
    console.log("Detected input: " + text);
    if (assistantResult.result.output.intents.length > 0) {
      var intent = assistantResult.result.output.intents[0];
      console.log("Detected intent: " + intent.intent);
      console.log("Confidence: " + intent.confidence);
    }
    if (assistantResult.result.output.entities.length > 0) {
      var entity = assistantResult.result.output.entities[0];
      console.log("Detected entity: " + entity.entity);
      console.log("Value: " + entity.value);
      if ((entity.entity === 'help') && (entity.value === 'time')) {
        var msg = 'The current time is ' + new Date().toLocaleTimeString();
        console.log(msg);
        assistantResult.result.output.generic[0].text = msg;
      }
      if (intent != null && intent.intent === "out_of_scope"
        && assistantResult.result.output.entities.filter(val => val.entity ===
"cardevice").length > 0) {
          let discoveryParams = {
            'query': text,
            'environmentId': config.watson.discoveryEnv.environmentId,
            'collectionId': config.watson.discoveryEnv.collectionId,
            'passages': true,
            return: 'text, title, sourceUrl, passages'
          };
          watsonDiscovery.query(discoveryParams)
            .then(discoveryResult => {
              console.log(JSON.stringify(discoveryResult.result, null, 2));
              assistantResult.result.output.generic[0].text =
discoveryResult.result.passages[0].passage_text;
              return res.json(assistantResult);
            })
            .catch(err => {
              console.log('error:', err);
              return res.json(err);
            });
      } else {
        // console.log(JSON.stringify(assistantResult, null, 2));
        res.json(assistantResult);
      }
    } else {
      res.json(assistantResult);
    }
  })
  .catch(err => { // session expired -> invalid
    res.status(404).json(err);
  })
}
```

Here we have added a section for the out_of_scope event and entity cardevice.

Lines	Description
50	Successful return from Watson Assistant with assistantResult
56	Check that there is an intent
61	Check that there is an entity
65	Check for the help entity with a time value
70	Check for #out_of_scope with a @cardevice
79	Call the Discovery service with appropriate parameters
82	On a successful return set the first text passage as the Assistant return message
90	When no Discovery is called, return what's set as the return message.
94	Catch any remote call error (networking, etc.)
97	The assistant session in the lite plan expires after 5 minutes of inactivity

Step 196 The results of a discovery search often return code with html tags or variables. Therefore the tags have to be removed. **Add** the following function to the *scripts.js* in folder *public/js* before function sendMessage().

```
          function strip_html_tags(str) {
            if ((str===null) || (str===''))
              return false;
            else
              str = str.toString();
              return str.replace(/<[^>]*>/g, '');
          }
```

Step 197 Change the xht.onreadystetechange function to the following. This is now already prepared for the Watson Assistant Search Skill integration which does not return response_type TEXT but a response_type SEARCH.

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if (xhr.status == 200) {
      var json = JSON.parse(xhr.responseText);
      context = json.context;
      const genericRes = json.result.output.generic.filter(el => el.response_type === "text"
|| el.response_type === "search");
      if (genericRes[0].response_type === "text") {
        let plainText = strip_html_tags(genericRes[0].text).replace(/&.*;/, '');
        updateChatLog("Watson", plainText);
        console.log(plainText);
        if (document.getElementById('checkSpeech').checked)
          speak(plainText);
        } else {
          let title = "<b>" + json.result.output.generic[0].results[0].title + "</b><br/>";
          let passage =
title.concat(json.result.output.generic[0].results[0].body.substring(0, 100)).concat(' ...');
          updateChatLog("Watson", passage);
          if (document.getElementById('checkSpeech').checked)
            speak(passage);
          }
        } else { // the session may be expired
          var json = JSON.parse(xhr.responseText);
          var errText = json.code + " - " + json.message + " - Reload page for new session
(timeout 5 min)";
          updateChatLog("Watson", errText);
        }
      }
    }
```

Step 198 Save the file.

Step 199 In the terminal execute

```
npm install
```
– only if you have cloned the repo.

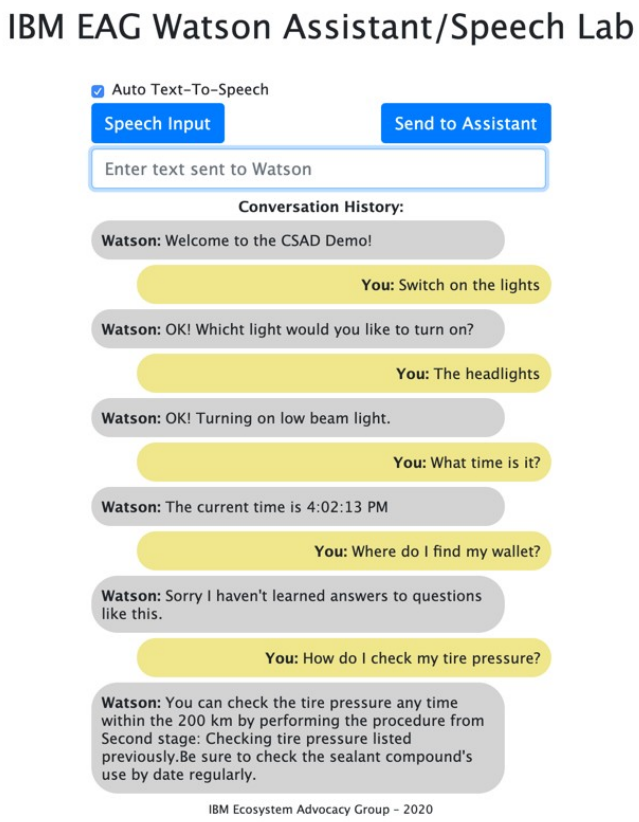
```
npm start
```

Step 200 In the browser open

http://localhost:3000

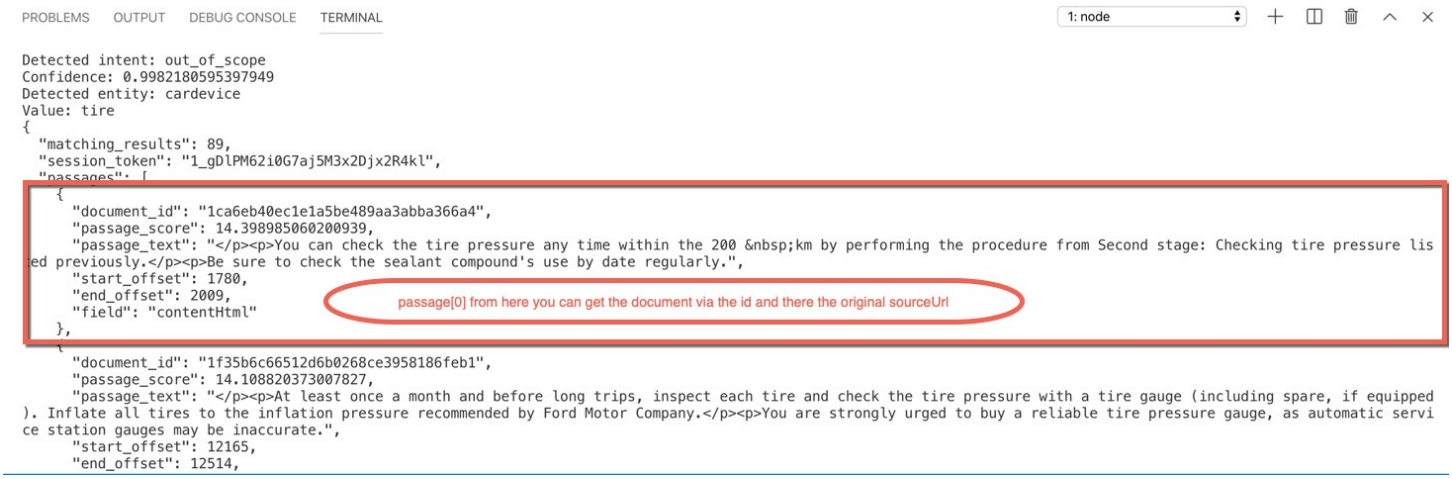
Step 201 Enter some statements into the Assistant input line:

```
Switch on the lights

The headlights

What time is it?

Where can I find my wallet?

How do I check my tire pressure?
```



The question for the tire pressure should return the same as in **Step 171**, the first passage.

In the server console you should see the JSON returned from Discovery:



Push your appliction to the IBM Cloud

See **Lab03 Step 127ff** on how to push your application to the IBM Cloud.

Section 7 Integrate Discovery into a Chatbot (Search Skill)

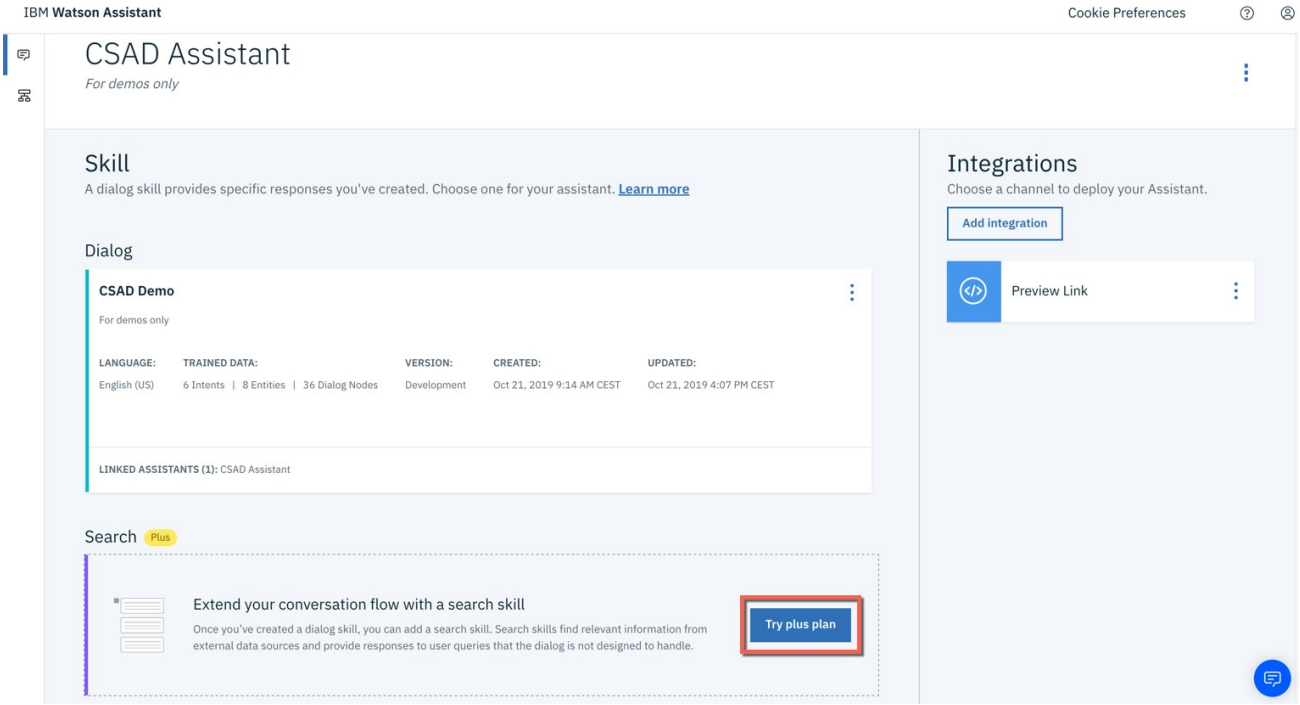
In the previous section we have seen in how to integrate the Watson Discovery service into a chatbot using the API.

In the Watson Assistant service there is now a *Search Skill* option available that allows to directly integrate a Discovery document collection into a chatbot.

To use this search skill we have to upgrade our Discovery service instance to the **Plus Trial** plan.

Create the Search Skill

Step 202 In the browser **open** your *Watson Assistant* console and **click** Try plus plan



Step 203 Then **click** Start free trial and **click** Agree to start the 30 day trial period.

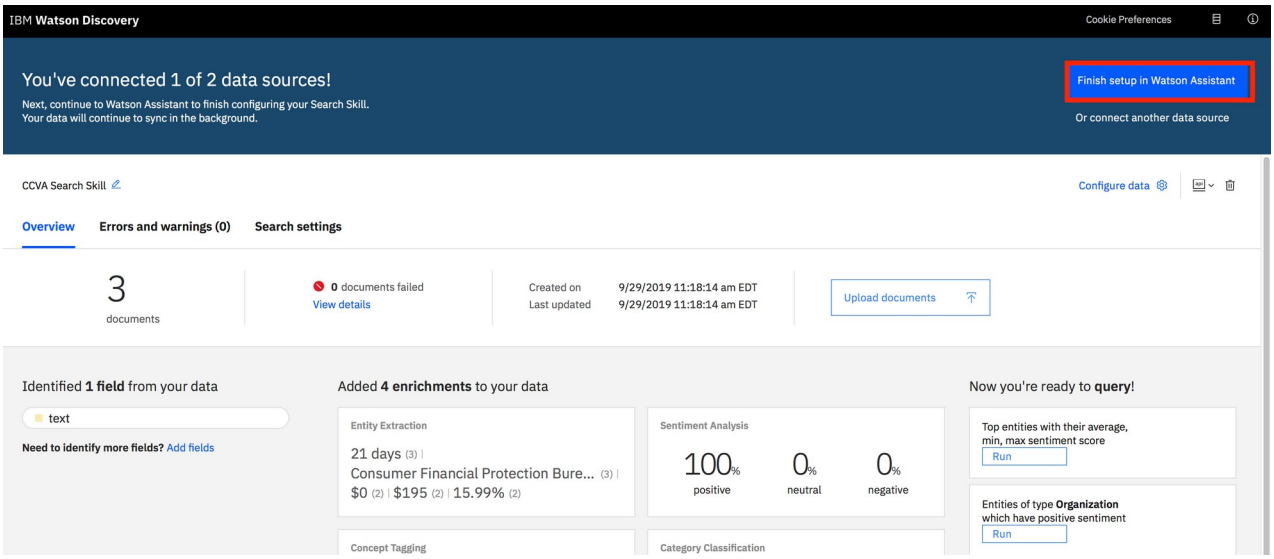
Step 204 After the upgrade **click** Add search skill.

Step 205 **Name it** f.e. `CSAD Search` and **click** *Continue*. Your previously created Discovery service will be displayed with the already defined collections.

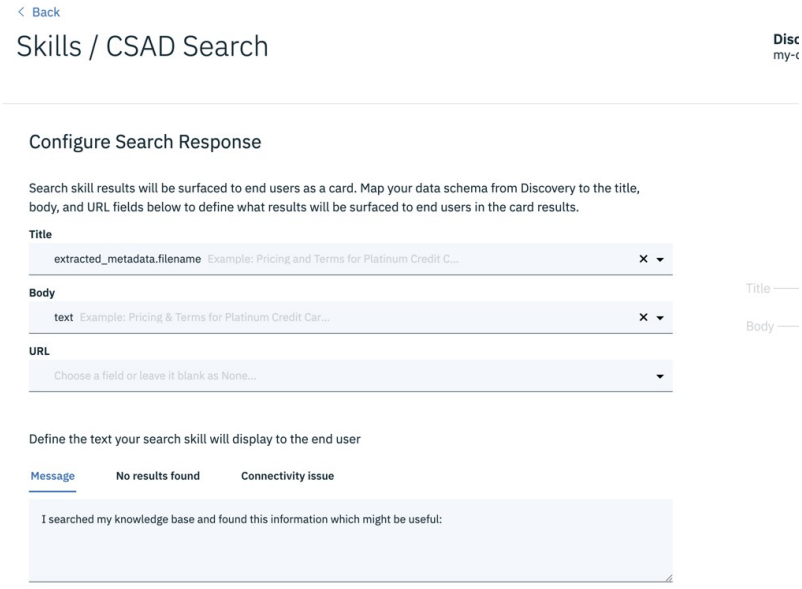
Step 206 **Click** Create new collection. Then **click** Let's get started. You cannot have more than two collection is your lite service.

Step 207 **Click** Upload documents. You can find these documents here or in the resources folder of the Github repository. **Name** your collection `CSAD Search`. Now upload the three PDF documents. **This may take a while!!**.

Step 208 Once the documents have been ingested, **click** *Finish setup in Watson Assistant*

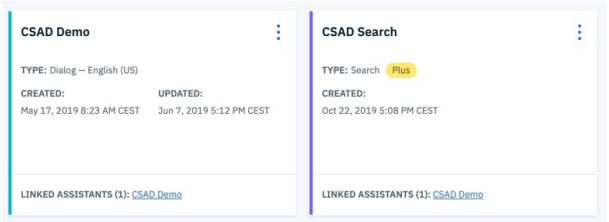


Step 209 Make sure the new collection is selected and **click** *Configure*.



For **Title** select the field: *extracted_metadata.filename*
For **Body**, select the filed: *text*.
For **url**, leave empty.

Then **click** *Create*.



Connect the Search Skill to a place in the dialog

Step 210 **Open** the *Intents* section of your Assistant skill

Step 211 **Click** Create intent enter the following values and **click** Create intent

Field	Value
Intent name	Credit_Card_Info
Description	Search Skill integration

Step 212 The user examples are phrases that will help Watson recognize the new intent. (Enter multiple examples by **pressing** "Enter" or by **clicking** the *Add example*). When finished, **click** ← at the top of the page.

Field	Value
User example	are there any international fees if i use my card in paris atm fees for my credit card fees for using my card abroad what are the apr fees for my silver credit card What are the fees for using an ATM in another country?

Step 213 **Open** the *Entities* section of you Assistant skill

Step 214 **Click** Create entity, **enter** `card_info` and **click** Create entity

Step 215 **Click** Add value. When finished, **click** ← at the top of the page.

Field	Value	Synonym
Entity name	card_info	
Value	APR	annual percentage rate
	cash advance	advanced cash, money advance
	international fees	International charges, foreign fees, foreign charges, fees while abroad, foreign exchange
	annual fee	yearly fee, annual membership, yearly membership, yearly cost, entry fee

Step 216 **Open** the *Dialog* section of your Assistant skill

Step 217 **Click** the menu ⋮ on the *Book a Table* node and then **click** *Add node below*, with the following values.

Field	Value
Name this node…	Information for a Card
If bot recognizes:	#Credit_Card_Info

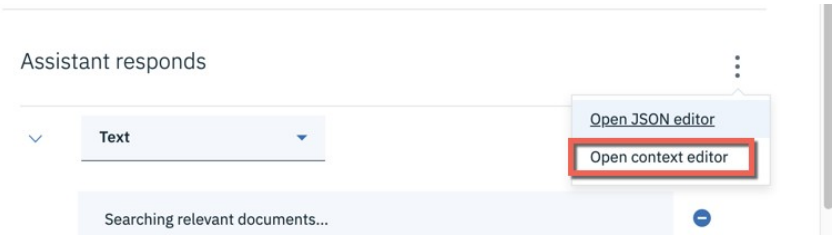
Step 218 **Click** Customize and **enable** *Multiple responses*. The click Apply

Step 219 Enter the following responses:

If Assistant recognizes	Respond with
@card_info:(annual fee)	Searching relevant documents…
@card_info:(international fees)	Searching relevant documents...
@card_info:(APR)	Searching relevant documents…
@card_info:(cash advance)	Searching relevant documents...
anything_else	Searching relevant documents...

Step 220 Now customize the response for response 1, click ⚙

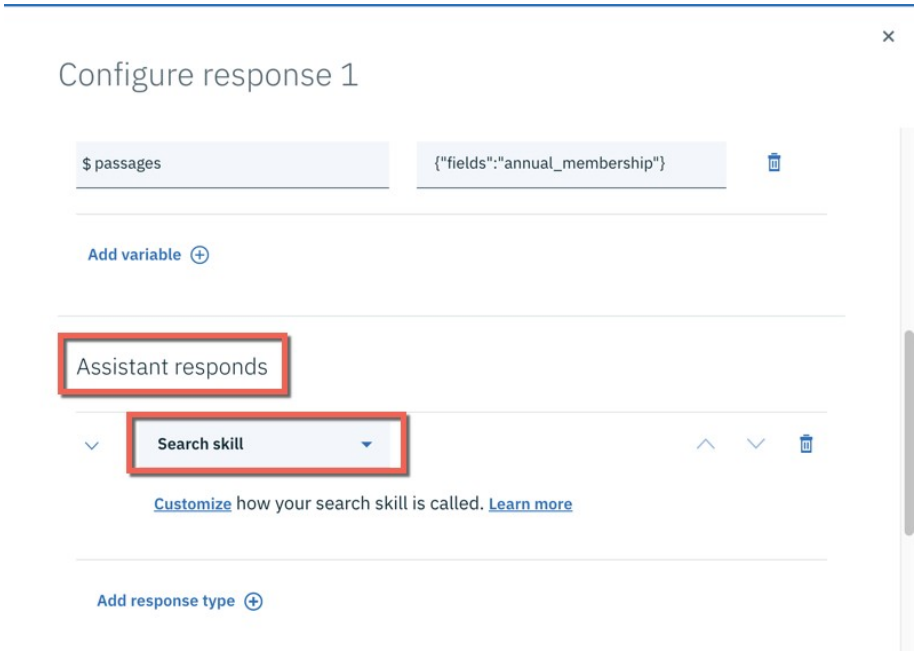
Step 221 Open the Context Editor



Step 222 Add two context variables

Variable	Value
actions	"WDS_SEARCH"
passages	{"fields":"annual_membership"}

Step 223 **Change** the *Assistant responds* option to *Search skill*. **Click** Save



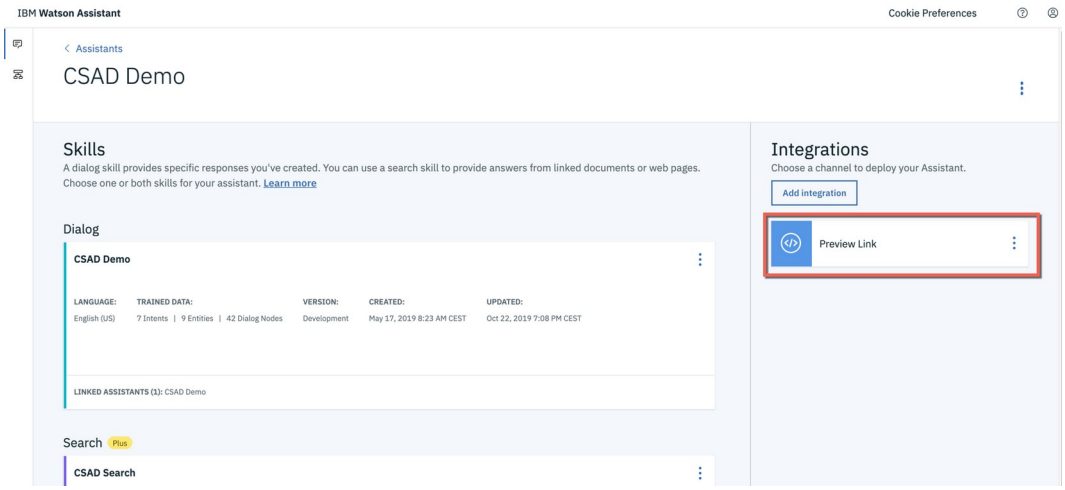
Step 224 Do the same for the second response, but for passages enter the following:

Variable	Value
actions	"WDS_SEARCH"
passages	{"fields":"international_fees"}

Step 225 Then **click** × to close the dialog.

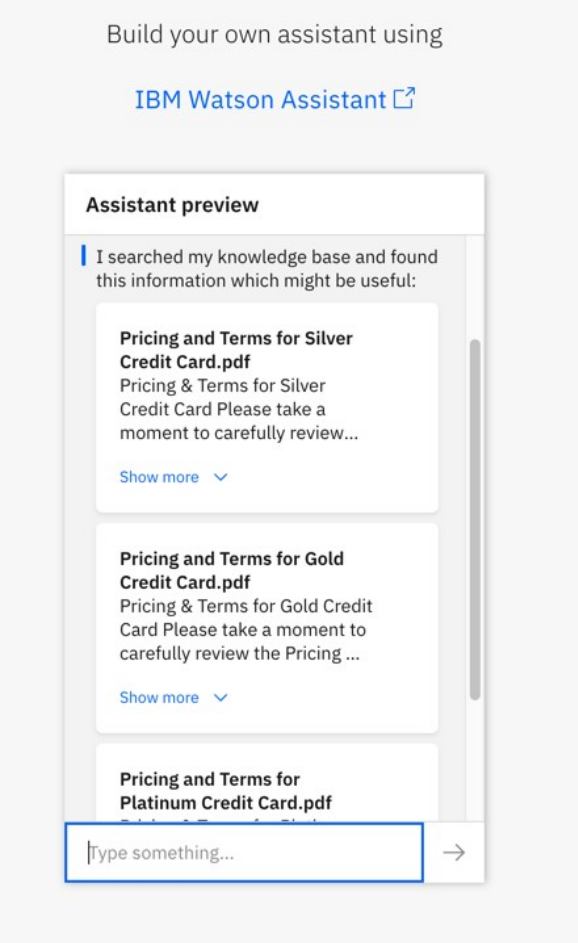
Step 226 All other responses just return "*Searching relevant documents...*"

Step 227 **Open** your Assistant and **click** the *Preview Link*. **Click** the *URL shown* to open the Assistant in the browser.

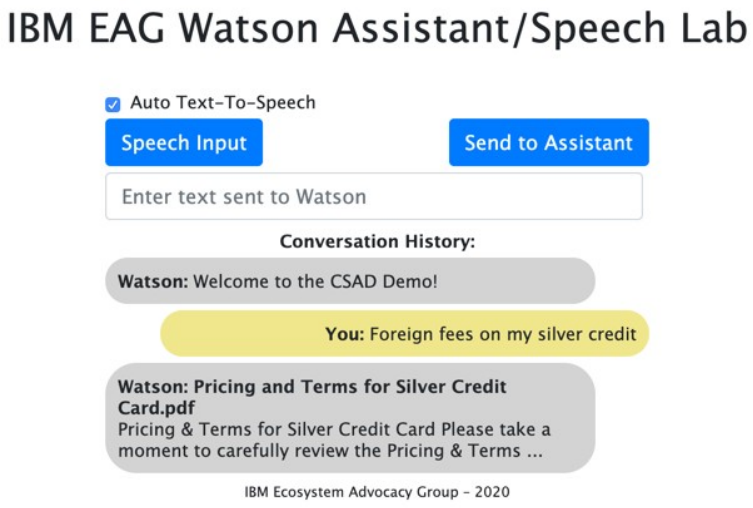


Step 228 Enter the following into your Assistant preview:

```
Foreign fees on my silver credit card
```



Step 229 Now you can test the search skill from your application
Enter `Foreign fees on my silver credit card` in the chat bot.
The first 50 characters will be displayed.



Tutorials using the Watson Discovery Services

Watson Discovery News Alerting

In this Code Pattern, we will build a Node.js web application that will use the Watson Discovery Service to access Watson Discovery News.

Watson Discovery News is a default data collection that is associated with the Watson Discovery Service. It is a dataset of primarily English language news sources that is updated continuously, with approximately 300,000 new articles and blogs added daily.

https://www.youtube.com/watch?v=zFl-2FybDdY&feature=youtu.be
https://github.com/IBM/watson-discovery-news-alerting
discovering news alerting (The Build Query tool):
https://www.youtube.com/watch?v=N-HaIpPGde0&feature=youtu.be