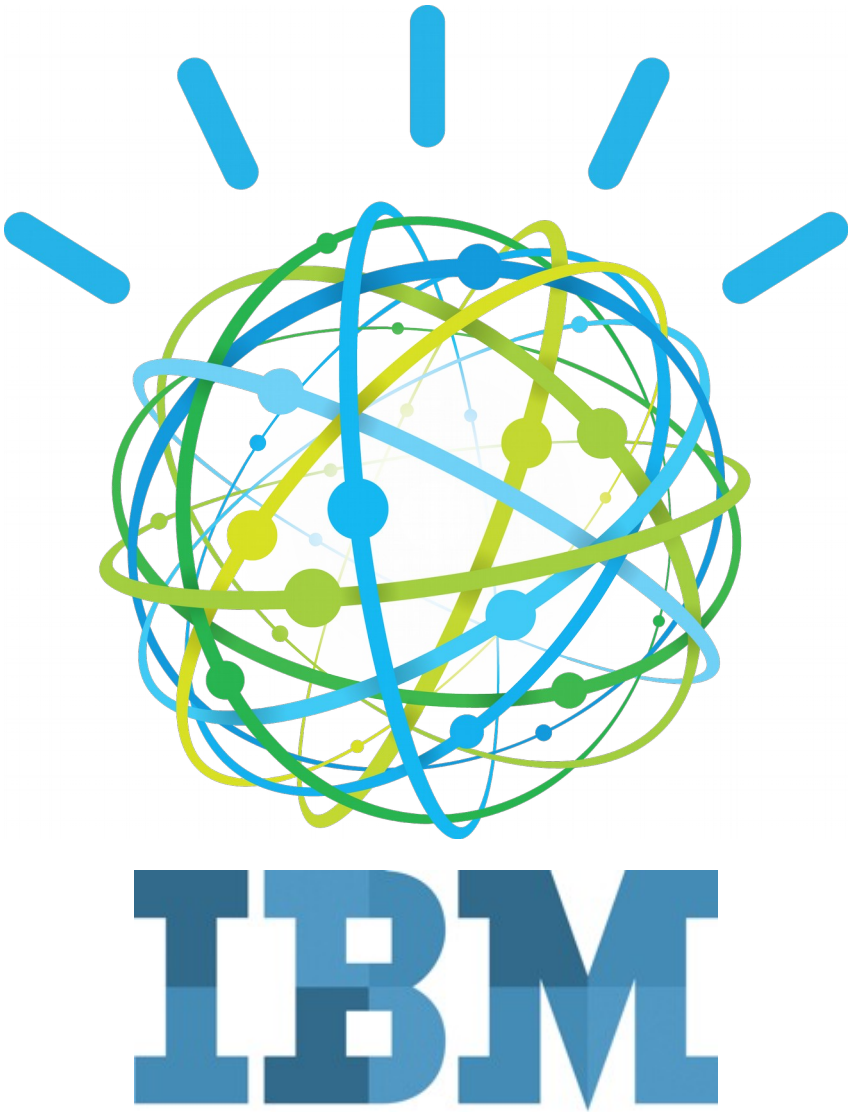
IBM Watson Assistant– Node.js

Cognitive Solutions Application Development

IBM Global Business Partners
Duration: 90 minutes
Updated: May 16, 2019
Klaus-Peter Schlotter
kps@de.ibm.com



Version 3.0

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

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

http://console.ng.bluemix.net

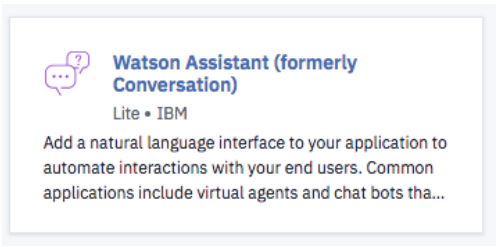
Note: The IBM Cloud UI is currently moving from **bluemix.net** to **cloud.ibm.com** https://cloud.ibm.com. But in this document the screenshots show bluemix.net.

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.

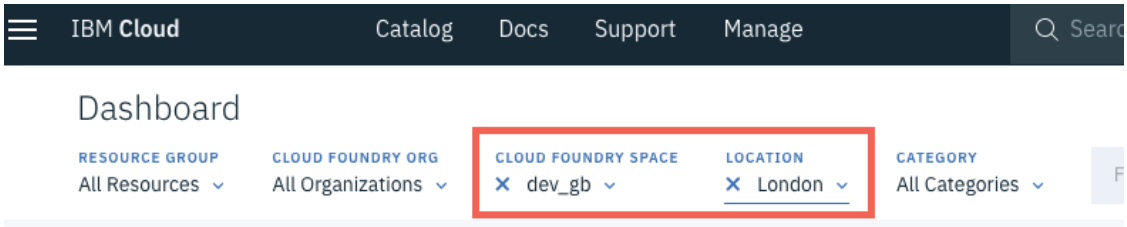


Step 6 Review the details for this service. At the top, there will be a description of the service. At the bottom, you can review the pricing plans. The Lite plan for this service provides no cost monthly allowances for workspaces, intents, and API calls. Enjoy your demo!

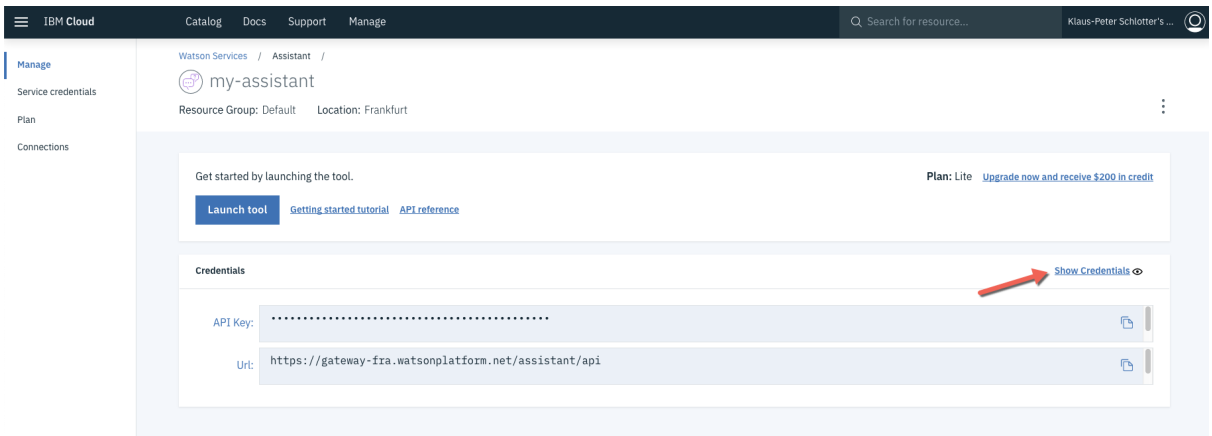
Step 7 Enter the information for your service, then **click** Create at the bottom.

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

Note: Use the region where you have your Cloud Foundry Space defined



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.

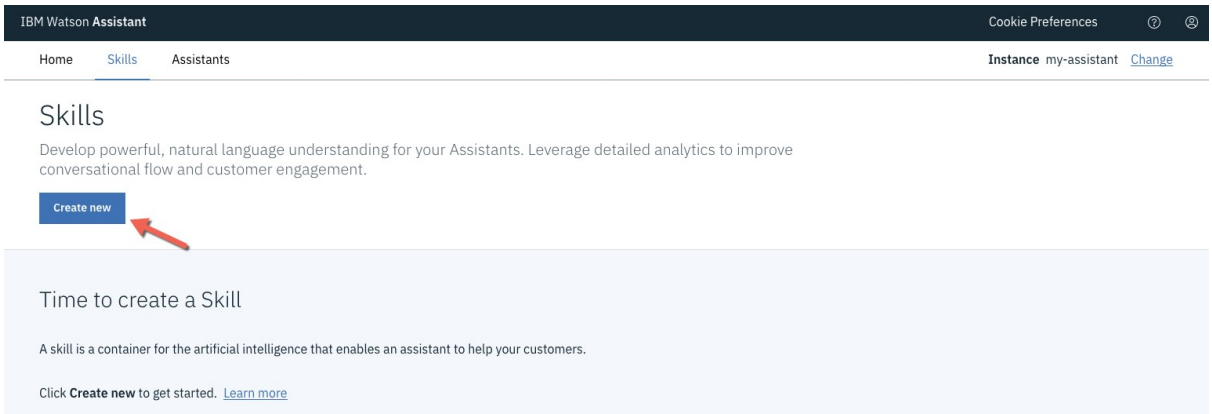
The resulting Skill is also available as JSON backup file on GitHub. In a Terminal you can download it with the following command:
`wget https://raw.githubusercontent.com/iic-dach/csadConversation/lab3_csadConversation/Lab3_skill-CSAD-Demo.json`
On the *Add Dialog Skill* page, on the *Import skill* tab you can import this JSON file..

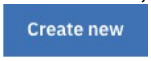
Step 10 In this guide, you will build a Conversation simple app to demonstrate the Natural Language Processing capabilities of IBM Watson in a simple chat interface. You will also learn how to identify a user's intent and then perform an action from a third-party application.

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

Step 12 The IBM Watson Assistant opens on the *Home* tab. Here you can learn about the basics of the Watson Assistant artifacts.

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

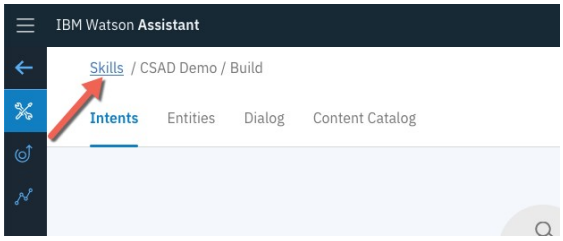


Step 14 First, you will need to create a new workspace. Workspaces enable you to maintain separate intents, user examples, entities, and dialogs for each application. **Click** 

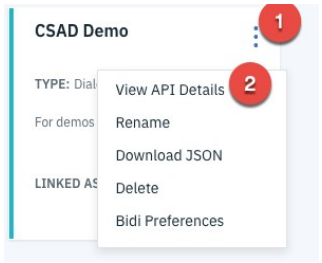
Step 15 On the *Add Dialog Skill* page, on the *Create skill* tab, enter the following values and click **Create**.

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

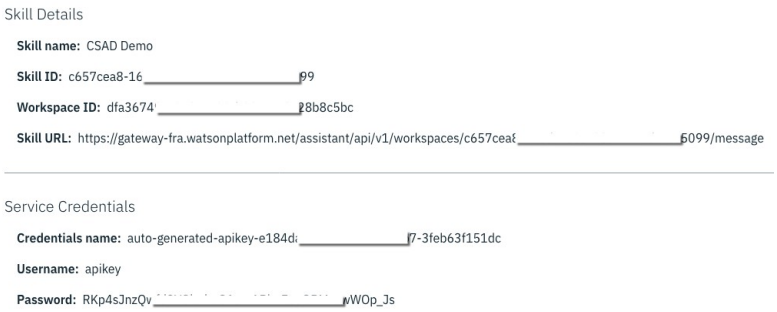
Step 16 Once the Skill has been created, you will be redirected to it. However, before proceeding, you will need to know how to identify the Skill so that it can be referenced by future applications. In the menu on the left, **click** Skills.



Step 17 On the tile for your new Skill, click **Options** → **View API Details**.



Step 18 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.



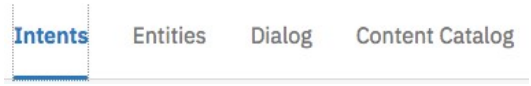
Step 19 In the upper right corner of this page, **click** ✕.

Step 20 **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 21 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 top of your workspace.



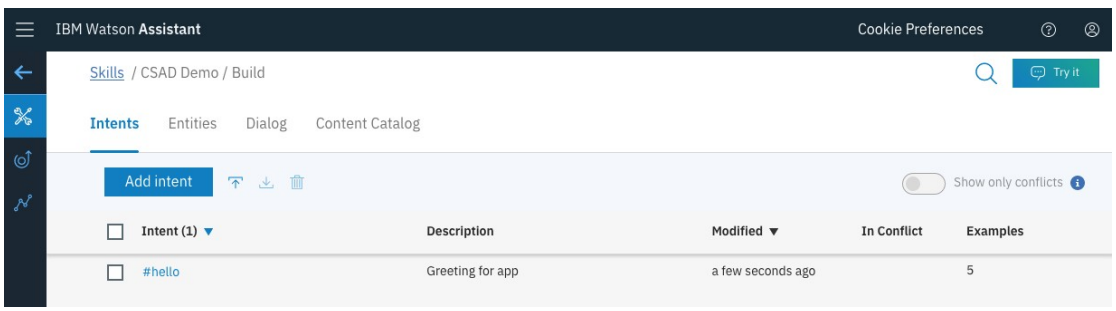
Step 22 **Click** Add intent enter the following values and **click** Create intent

Field	Value
Intent name	hello
Description	Greeting for app

Step 23 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 24 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 25 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

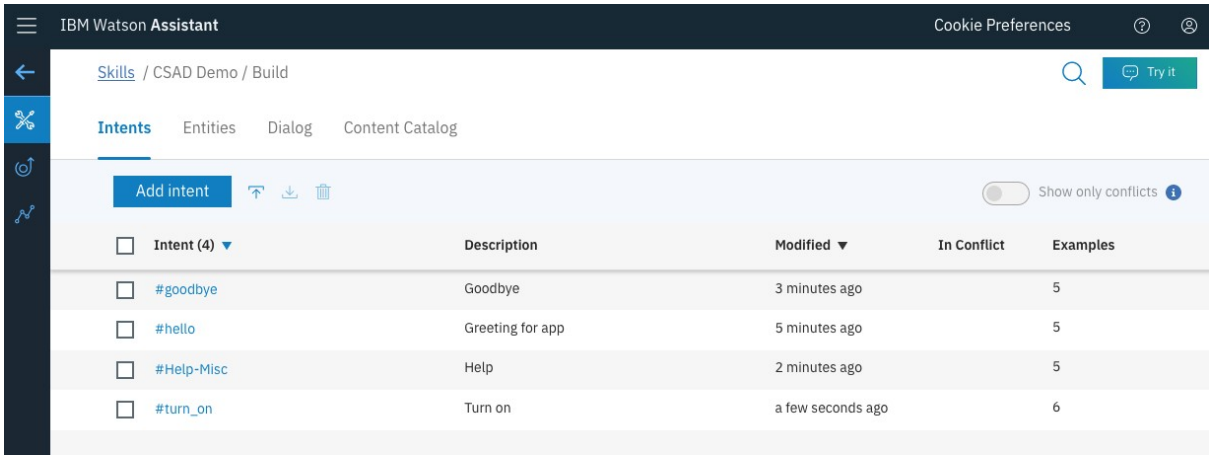
Step 26 Repeat the previous steps to create a new *intent* that helps the user ask for help. In the programming section of this guide, you will learn how to identify the user's intent (in a third-party application) so that you can perform the requested action.

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 27 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 headlights

Step 28 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.

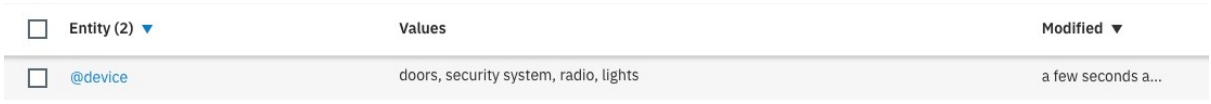
Step 29 Next, you will need to create some *entities*. An *entity* is the portion of the user's input that you can use to provide a different response or action to an *intent*. These entities can be used to help clarify a user's questions/phrases. You should only create *entities* for things that matter and might alter the way a bot responds to an *intent*. If not already there, **click** the *Entities* tab at the top of your Skill.

Step 30 On the *Entities* tab, click Add 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 31 You should see your new *entity* listed on the *Entities* page.



Step 32 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 33 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 34 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 35 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 that can be had with a user. If not already there, **click** the *Dialog* tab at the top of your Skill.

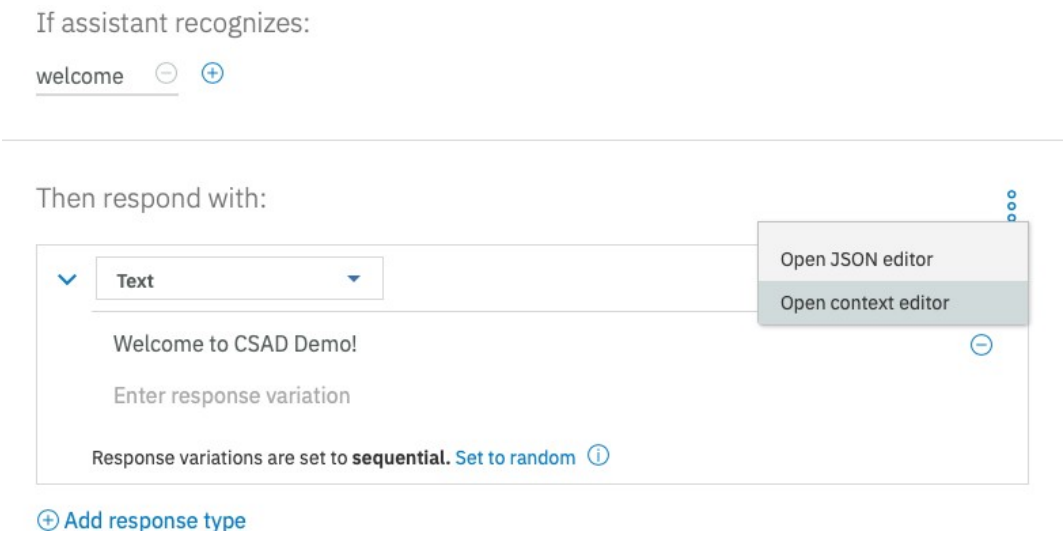
Step 36 Review the documentation for creating dialog nodes and for defining conditions.

Step 37 On the Dialog tab, **click** Create ⊕. Two default nodes will be 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 38 **Click** on the *Welcome* node to update the following properties

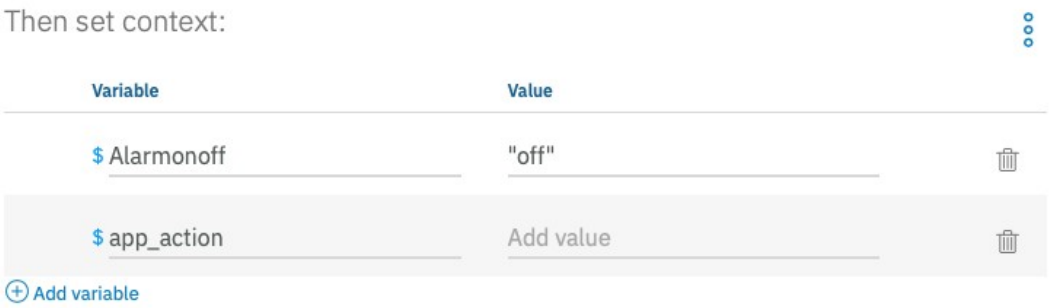
Field	Value
Name	Welcome *(should be the default)*
If bot recognizes:	welcome *(should be the default)*
Then respond with:	Welcome to the CSAD Demo!

Step 39 **Click** ⋮ then **click** *Open context editor*.



Step 40 Add two variables and **click** ✕ to close the properties view of the node.

Variable	Value
Alarmonoff	off
app_action	



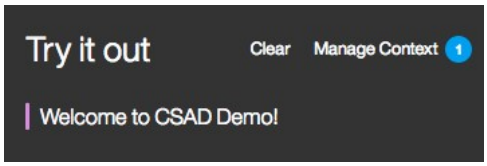
Step 41 Expand the *Anything else* node and review it's default values. **Close** the view.

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)	

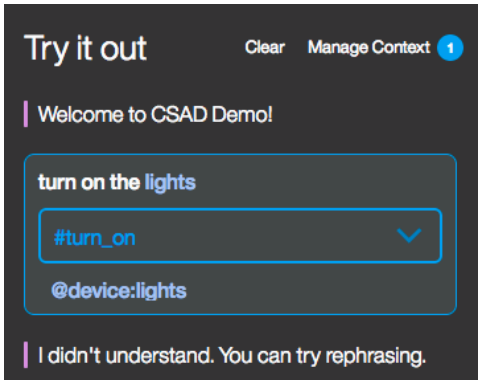
* Sequential means 1st response presented at first hit of anything_else node, and so on. Random means any of the responses is presented randomly.

Step 42 Now it's time to test the conversation.
In the upper right corner, **click** 

Step 43 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 44 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 recognize them but because we have not yet defined a node to catch them the bot does not yet understand (anything_else node).

Step 45 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 46 **Click** in the top right corner to close the chat pane. Proceed to the next section.

Build your conversation dialog

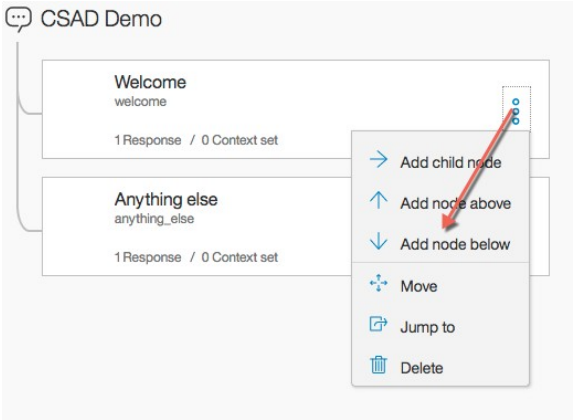
In this section, you will continue building your demo bot utilizing the intents, entities, and dialog nodes that you created in previous steps. You will do this entirely with the web interface (no programming or text/XML file hacking required!)

Step 47 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 48 **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 49 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 50 **Click** the menu ⋮ on the *Hello* node and then **click** *Add node below*, with the following values. Then **click** ✕ to close the dialog.

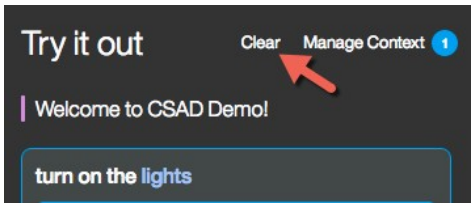
Field	Value
Name this node…	Goodbye
If bot recognizes:	#goodbye
Then respond with:	Until our next meeting.

Step 51 Using the same steps (Step 40 ff) as before, test the conversation by typing the following chat line(s):

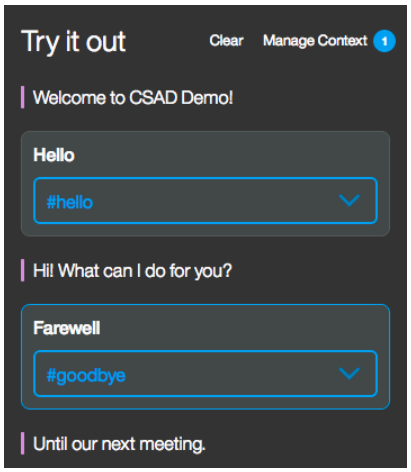
```
Hello
```

```
Farewell
```

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



You should see the appropriate result:



Step 52 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.

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

Step 53 **Click** the menu ⋮ on the *Turn On* node and **click** *Add child node*.

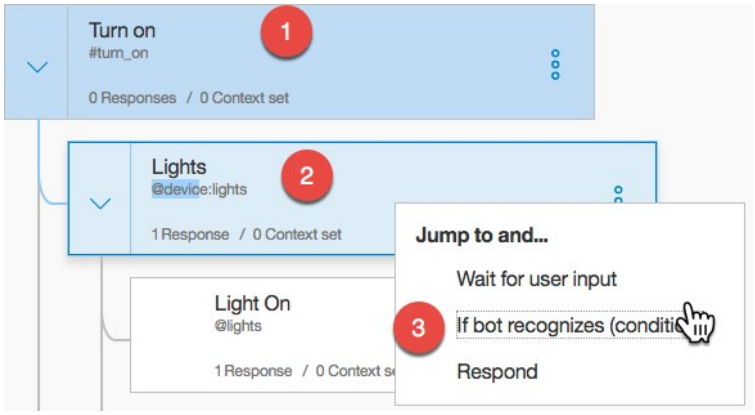
Step 54 In this new node, enter the following values. In this example, the only way this node will be reached is if the user specifies the **@device** *entity* "lights" (or one of its synonyms). Then **click** ✕ to close the dialog.

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

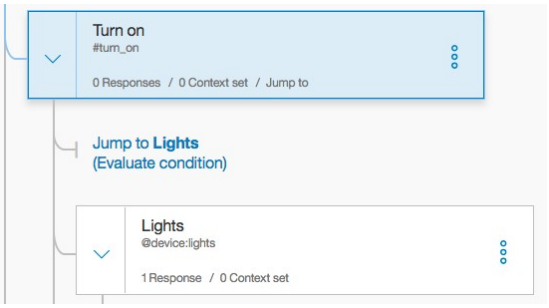
Step 55 In this scenario, you will want to automatically move from the *Turn On* node to the *Lights* node without waiting for additional user input.

a) Click the **Turn on** node and at the bottom **select** *Jump to* as the *And then* condition.

b) Now select the **Lights** node as the destination node.



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



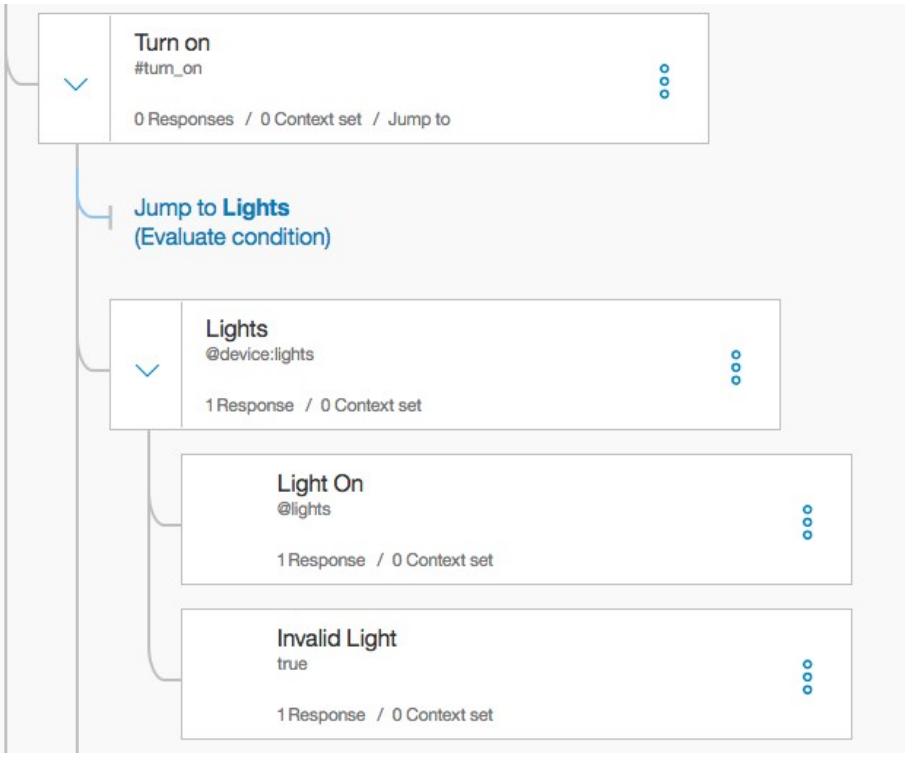
Step 56 At this point, Watson will ask you which lights to turn on. As you will recall, in earlier steps you created a @lights entity to define the available lights in the system. **Click** the menu ⋮ on the Lights node, **click** *Add child node* to create a node. In this new node, enter the following 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 57 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.

Field	Value
Name this node…	Invalid light
If bot recognizes:	true
Then respond with:	I'm sorry, I can only turn on some car related lights.

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



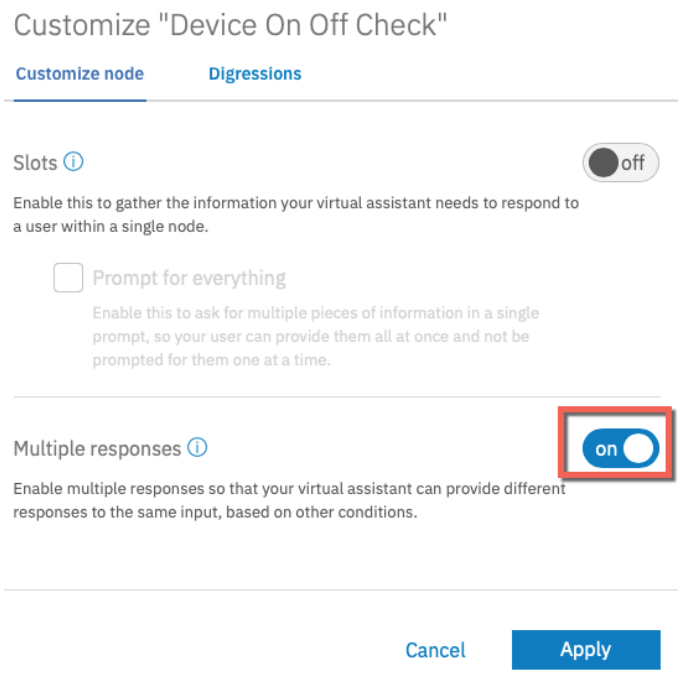
Step 58 You will need to add nodes to deal with the other devices that can be turned on. As you'll recall, you defined those devices with the **@device** *entity*. The **Turn On** node automatically jumps to the Lights node. So, **click** the menu ⋮ on the Lights node and **click** *Add node below* to create a new peer node. **Enter** the following values.

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

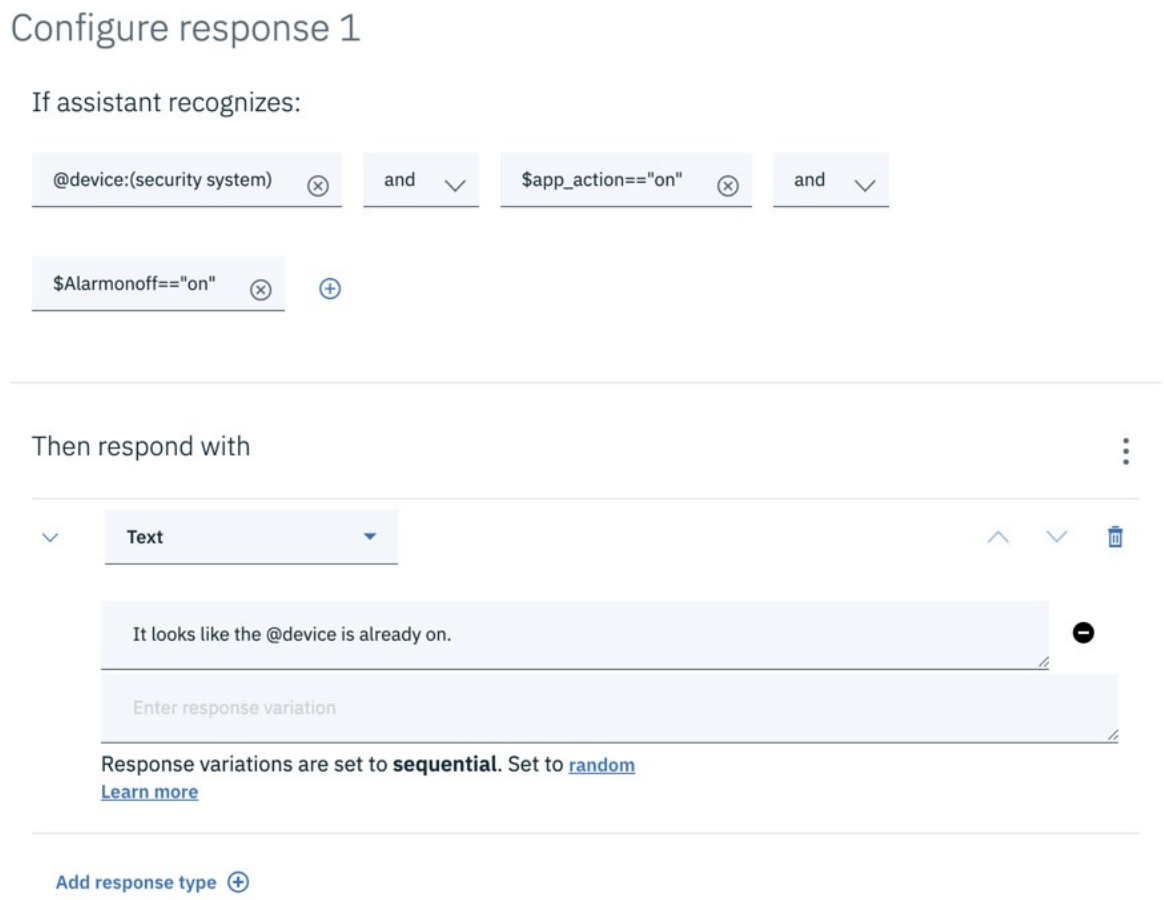
Step 59 On the Device **click** ⋮ and **click** *Add a child node*. Add the following values

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

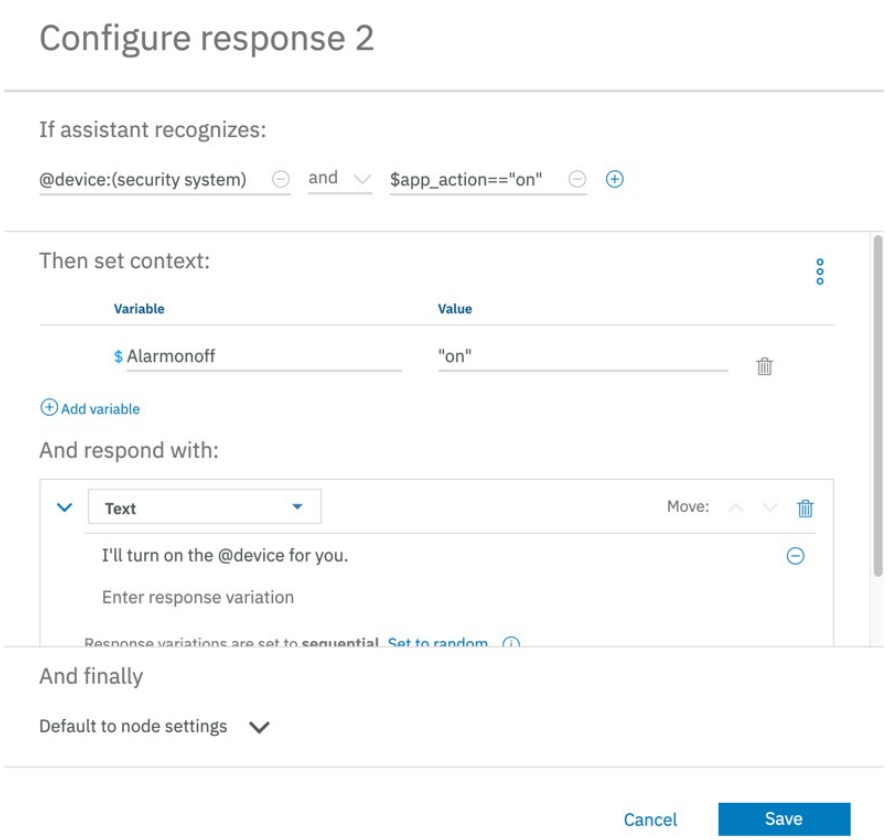
Step 60 **Click** Customize and **enable** *Multiple responses*. The click Apply.



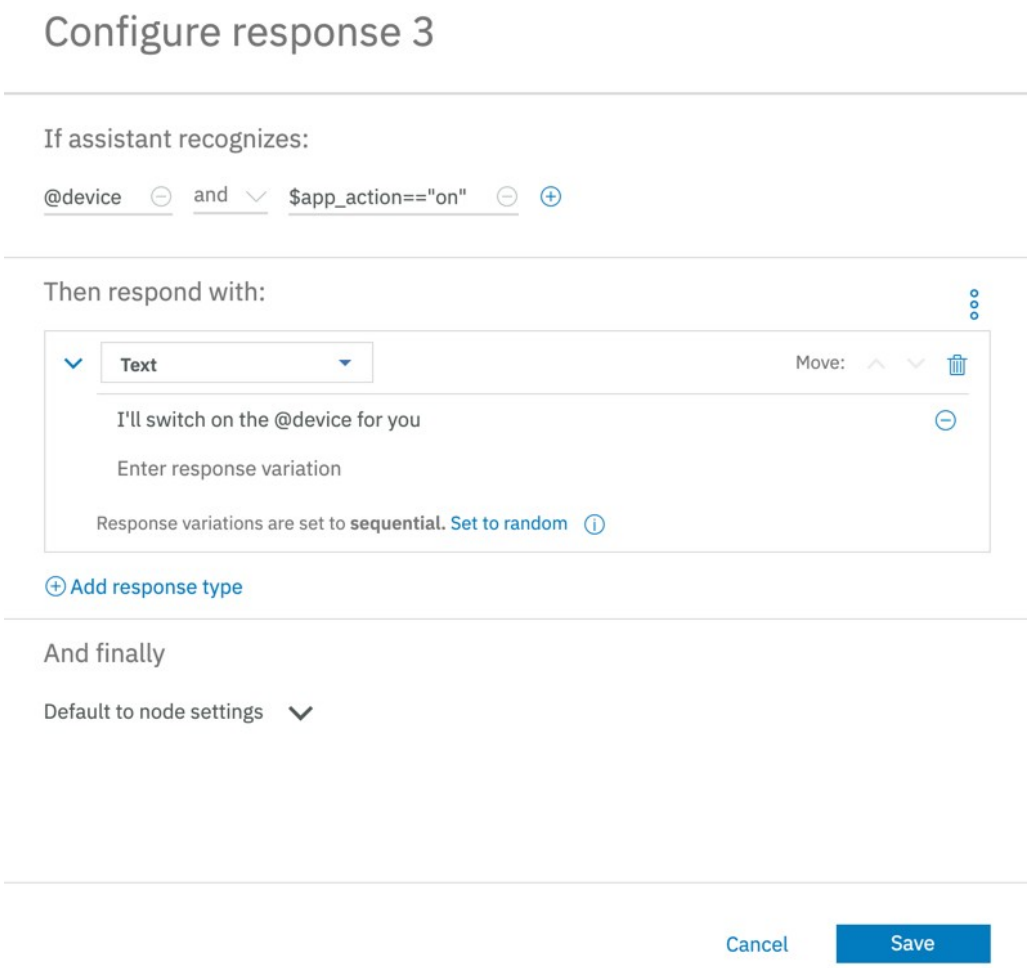
Step 61 Now add the following three answers in *Then respond with*: Use the ⚙ to customize and enter the values interactively and **click** Save. See the screenshots of each response:



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

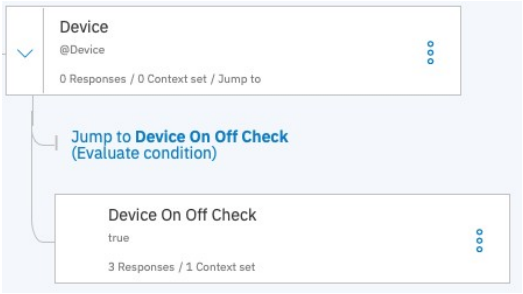


If assistant recognizes:
@Device:(security system) AND $app_action==”on”
Then set context (open the context editor)
$Alarmonoff “on”
And respond with:
I’ll turn on the @device for you.



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

Step 62 Now **click** ⋮ on the *Device* node and **click** *Jump to*, then **select** the *Device On Off Check*.



Step 63 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 64 Using the same steps as before, 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
```


Step 65 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 66 **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).

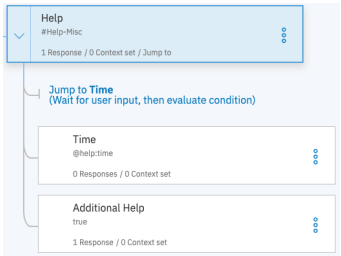
Note: You will provide your own custom response to the query in the programming exercises.

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 67 **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 68 **Click** ⋮ on the *Help* node and **select** *Jump to*. **Click** the *Time* node and **select** *Jump to and… Wait for user input*.



Step 69 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

Step 70 **(Optional)** Add a slots dialog to book a table in a restaurant

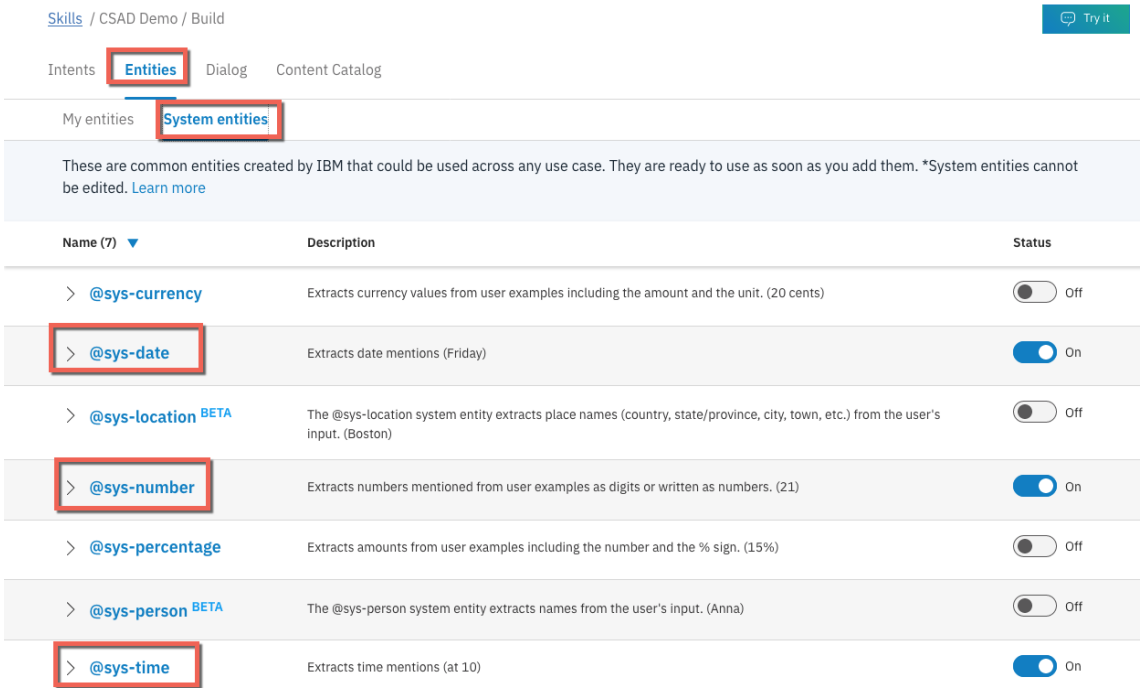
a) Add an Intent with the following values (see Step 21 ff):

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



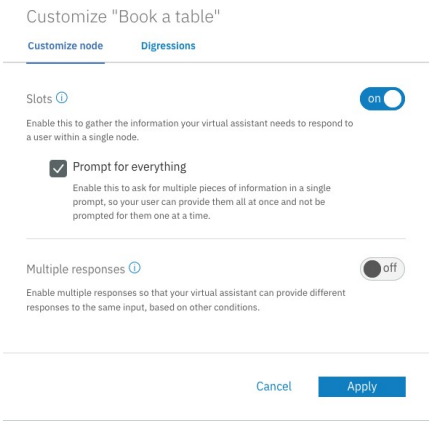
d) Add a dialog node for #book_table (Step 45 ff)

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 prefilled, 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.
```

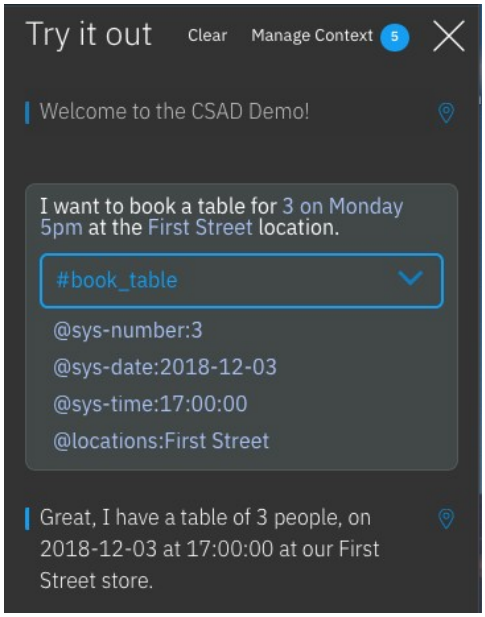
j) You can try (see Step 40 ff) this with a view sample inputs. Always Clear

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

```
I want to book a table
```

```
I want a table for 3 please
```

The result should look something like this:



Section 2: 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 (in Beta as of 20.03.2019) 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 69) in a Cloudant database using a Cloud Function. This can be achieved without programming.

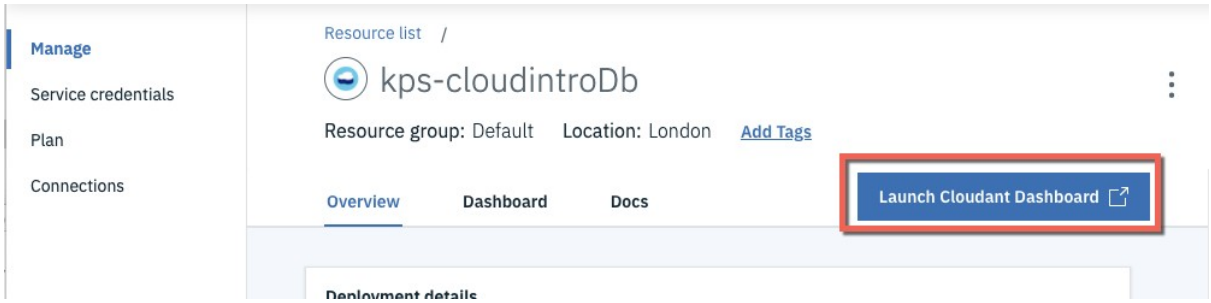
Step 71 In your Dashboard **click** Create resource, **select** the *Databases* category and **click** *Cloudant*.

Step 72 **Enter** a *Service name*, f.e workshopDb.

choose a region (select the region where you have connected with the ibm-cloud CLI). With the command `ibmcloud target` you can see your active definitions.

As authentication method **choose** *Use only IAM*.

Step 73 Launch the Cloudant Dashboard



Step 74 At to top **click** Create Database . **Enter** a name f.e. *reservations* and **click** Create

Step 75 In a Terminal install the IBM Cloud Functions CLI plugin

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

Step 76 Create a Cloud Foundry alias of your Cloudant services

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

Step 77 Create Credentials for this service with the following command

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

Alternativeley create this key in the Credentials section of your Cloudant instance created in Step 72)

Check the existance of this credential!

Step 78 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.

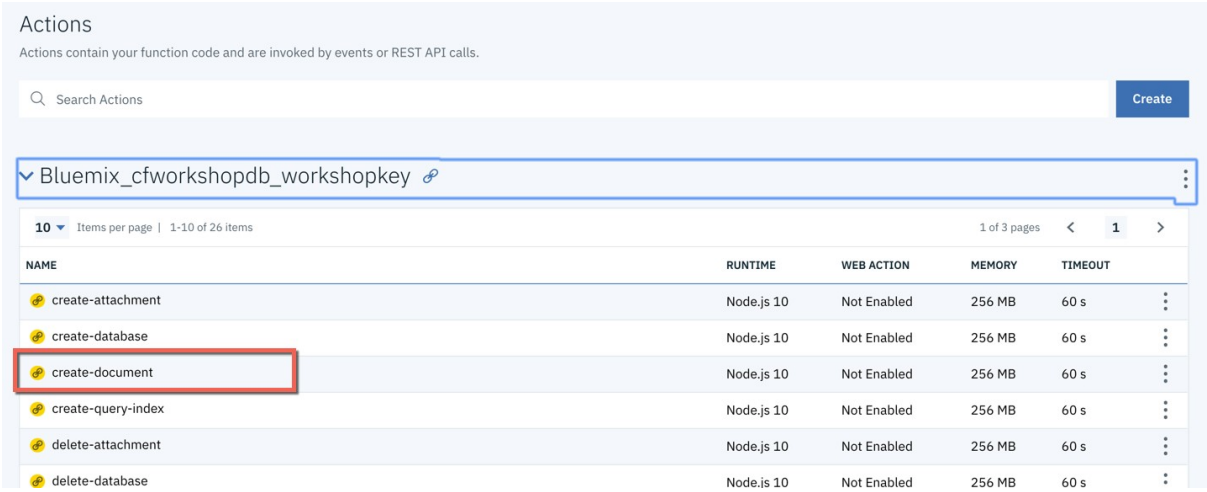
```
ibmcloud wsk package refresh
```

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

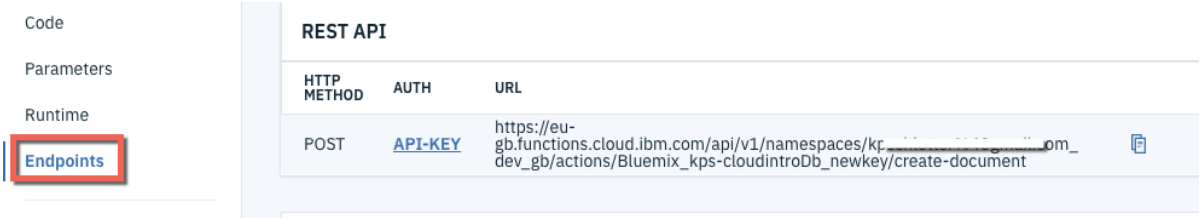
Step 79 In your cloud console goto Functions. Then **click** *Actions* in the left menu. You should see the functions that the previous command has created from some deployed services.

> **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 80 We are interested in the *create-document* function of the Cloudant service. Click ⋮ → Manage Action



Step 81 **Select** *Endpoints* and **copy** the *URL* and the *API-KEY* for later use in our Assistant Skill. (click API-KEY to get the value).



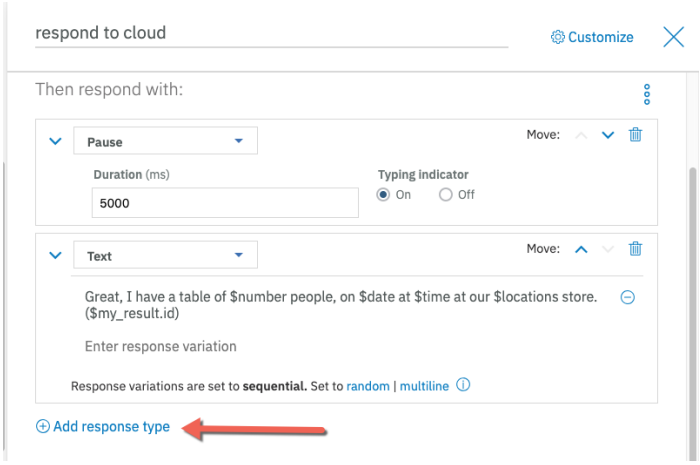
Step 82 In your Assistant Skill **open** the *Welcome* node and the *Open JSON editor*. Replace the content with the following:

There is a lab03_codesnippets.txt file that contains all the code snippets of this lab section.

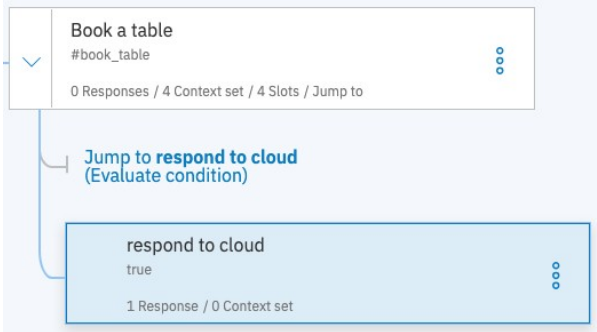
```
{
  "context": {
    "private": {
      "my_credentials": {
        "api_key": "<yourCloudFunctionsAPIKey>"
      }
    }
  },
  "output": {
    "text": {
      "values": [
        "Welcome to the CSAD Demo!"
      ],
      "selection_policy": "sequential"
    }
  }
}
```

Step 83 **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)	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 84 **Click** ⋮ on the *Book a table* node and **select** *Jump to*. **Select** the *Respond to cloud node*.



Step 85 **Open** the *Book a table* node and on ⋮ *Open JSON editor* and **replace** the content with the following:

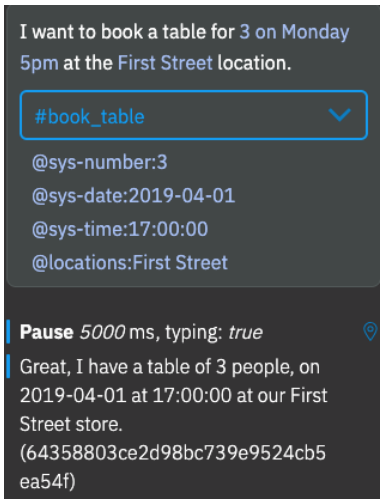
```
{
  "output": {
    "text": {
      "values": [],
      "selection_policy": "sequential"
    }
  },
  "actions": [
    {
        "name": "/kpschxxxx.com_dev_gb/actions/Bluemix_kps-cloudin-
troDb_newkey/create-document",
      "type": "server",
      "parameters": {
        "doc": {
          "date": "$date",
          "time": "$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 79 after .../namespaces.

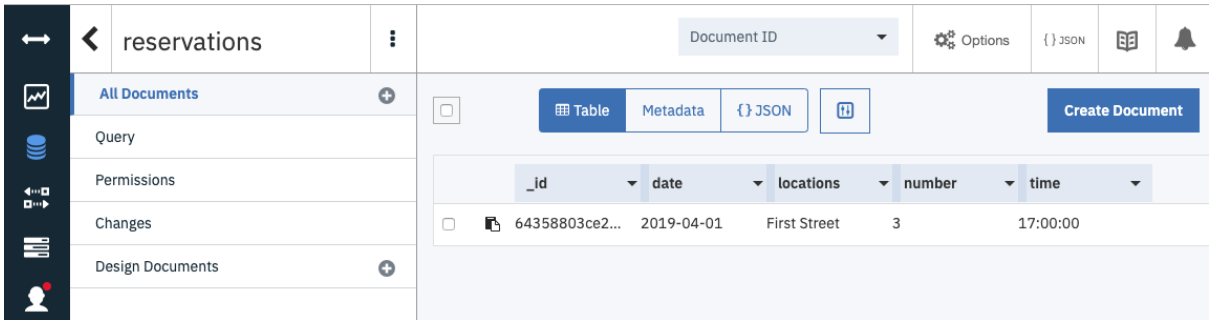
Step 86 Try the dialog with the following sentence:

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

An you should see a result like this:

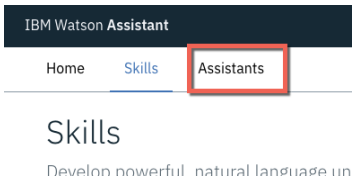


Step 87 Go back to your Cloudant dashboard. There you should see the document that was created.

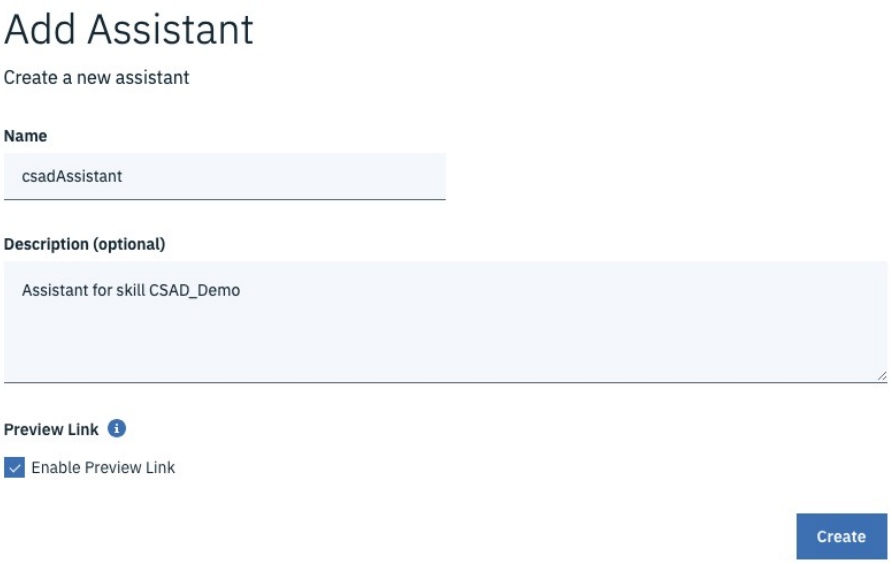


Create an Assistant from our Skill

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.

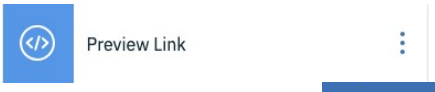


Step 88 **Click** Create new to create a new Assistant, **enter** name and description then **click** Create.



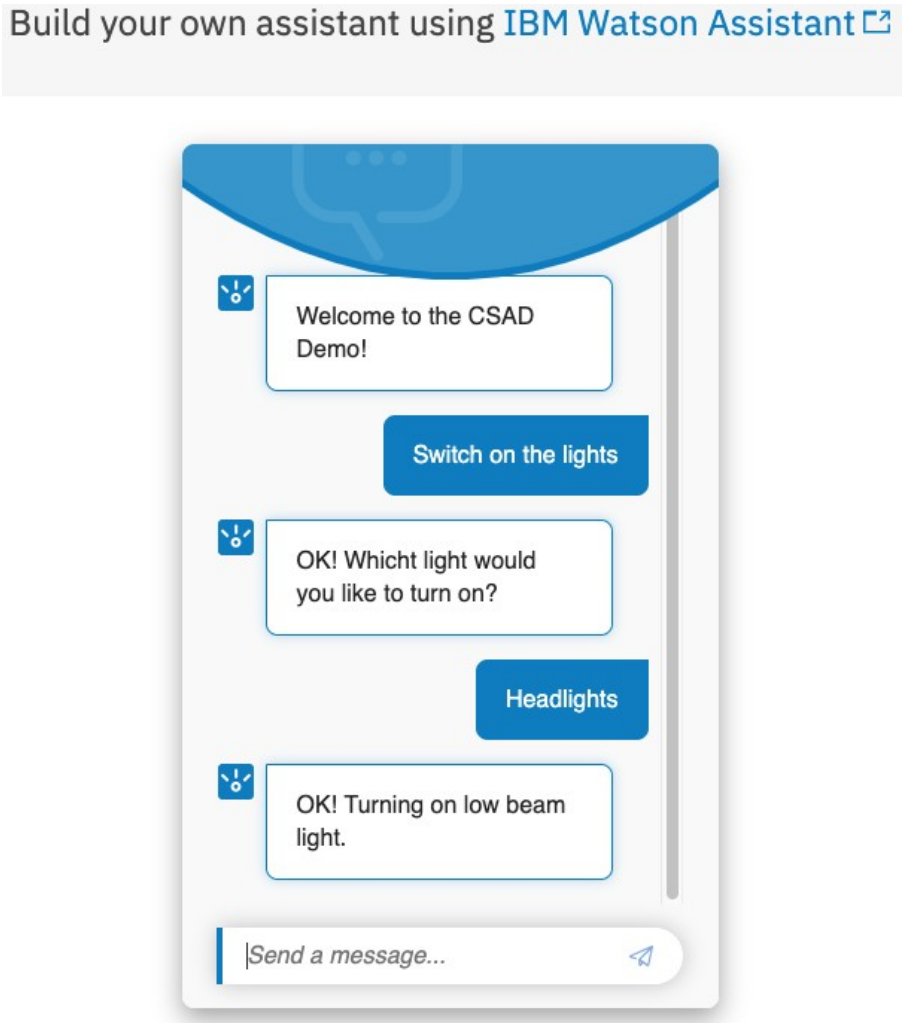
Step 89 Click Add dialog skill. and **select** our *CSAD Demo* Skill. With the Integrations function you can now directly integrate this Assistant into Facebook, Slack and other applications.

This is out of scopt of this introductory demo!!

Step 90 **Click** 

to make this Assistant available on the internet. **C** a name and click Save Changes. Copy the link and open it in a browser.

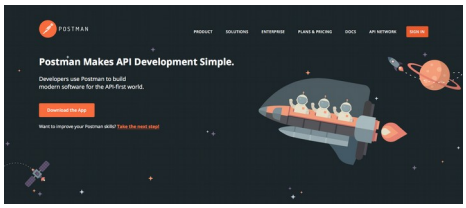
Step 91 It should look like the following:



Section 3: Test your Conversation 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 92 Postman is available as an application for MacOS, Windows, and Linux. https://www.getpostman.com



Step 93 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

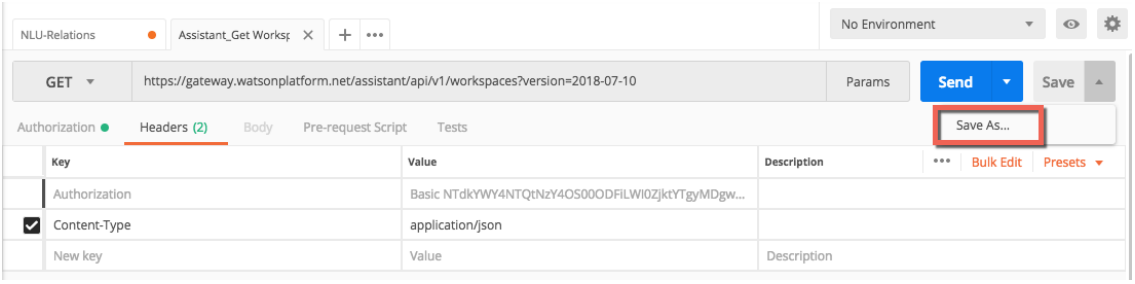
See your service credentials for the Url at your region. f.e. gateway-fra for Frankfurt.

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

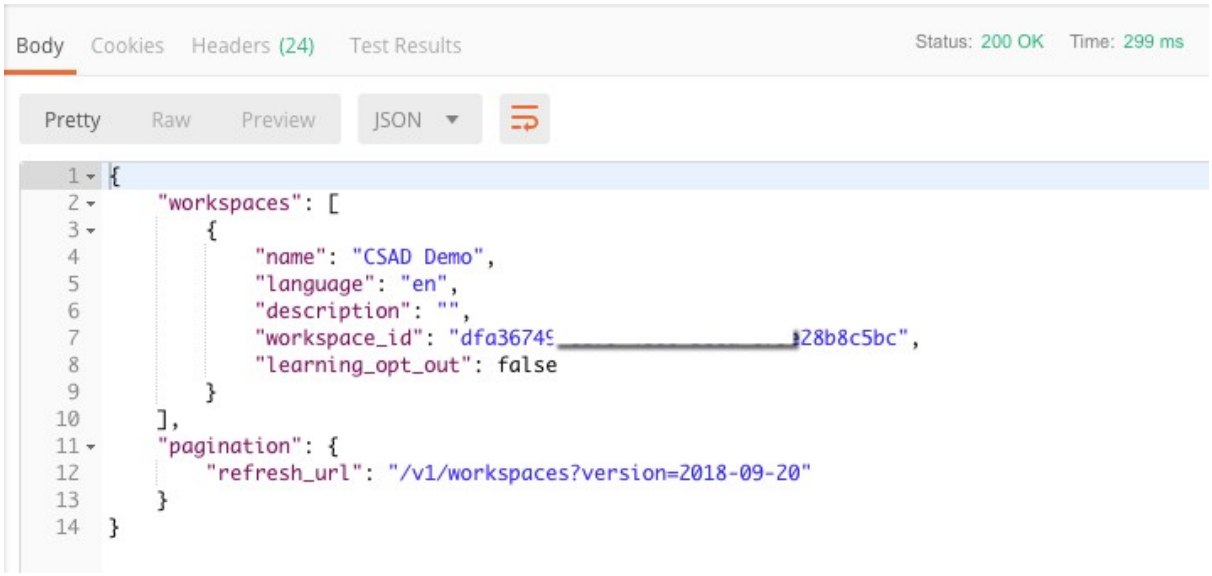
Note: The services in the IBM Cloud platform are currently transferred to an IAM authentication. So if your service was created with an **apikey** instead of **userid** and **password**, you specify the string **apikey** as userid and the **generated apikey** as password.

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 94 The result should look similar to this:



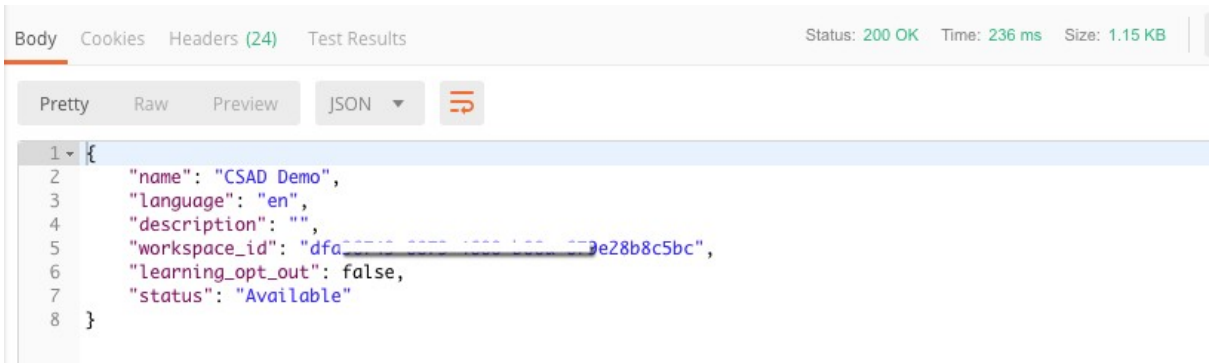
Step 95 **GET** the information from our **CSAD Demo** workspace.

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

Authorization: as in Step 87

Headers: as in Step 87

Your result should look like this:



Step 96 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 87

Headers: as in Step 87

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 97 POST a request for #turn_on

Url: as in Step 87

Authorization: as in Step 87

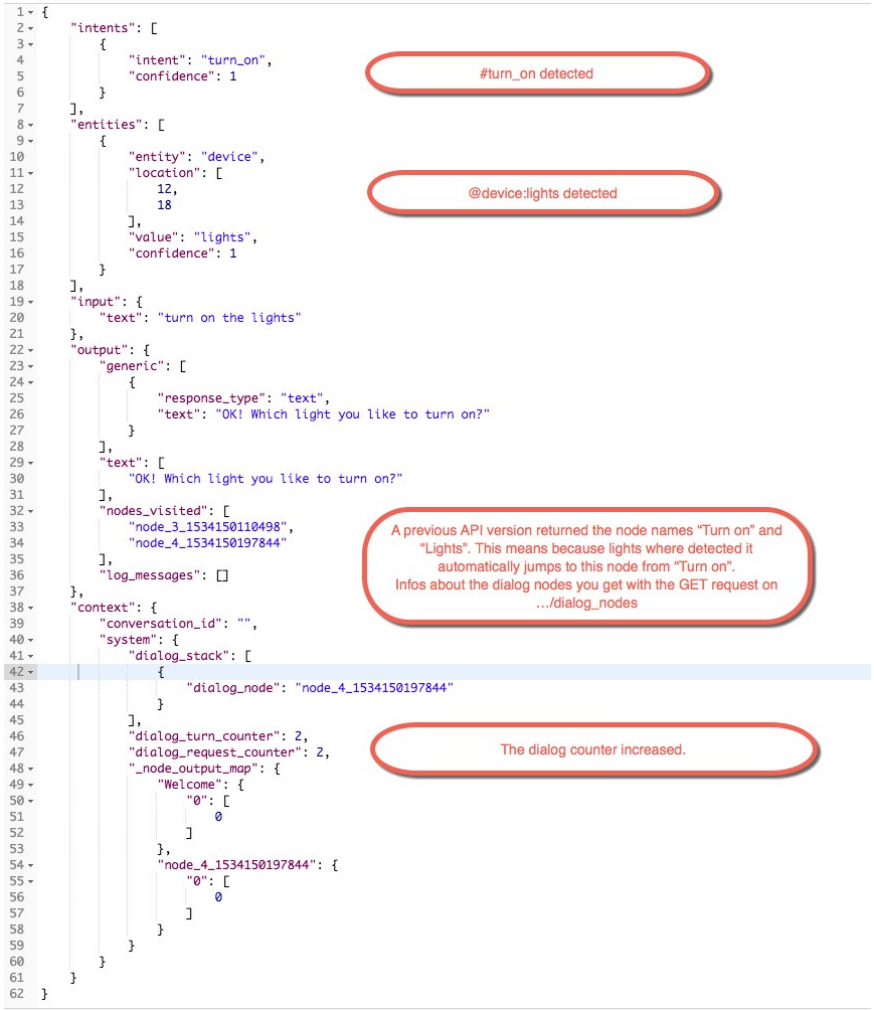
Headers: as in Step 87

Body: (raw) → Context from Step 87 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 98 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 -b lab3_csadConversation https://github.com/iic-dach/csadConversation.git`

There is a lab03_codesnippets.txt file that contains all the code snippets of this lab section.

Running the application locally

Server part – node.js

Step 99 In a Terminal window **create a folder** for your project f.e. *csadConversation*

Step 100 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": "csadconversation",
  "version": "1.0.0",
  "description": "A simple Watson Assistant example",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Klaus-Peter Schlotter",
  "license": "ISC"
}
```

Step 101 Add a start command to the package.json file (see above):

```
"start": "node app.js"
```


Step 102 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*

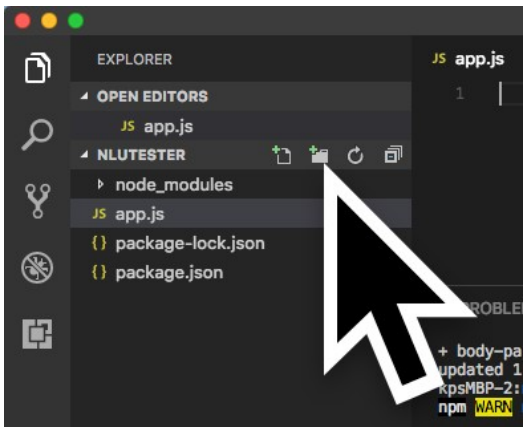
Step 103 In the *Intergrated Terminal* or another terminal execute the following commands to install the additional node modules needed for our project

```
npm install --save express ejs morgan body-parser watson-developer-cloud
```

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

Step 104 **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.

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

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

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

Step 105 In the project folder *create* an *app.js* file.

Step 106 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 107 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 108 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 AssistantV1 = require('watson-developer-cloud/assistant/v1');
const config = require('../config');

const watsonAssistant = new AssistantV1(config.watson.assistant);

exports.getIndex = (req, res, next) => {
  res.render('index');
}

exports.postMessage = (req, res, next) => {
  console.log('Text:' + req.body.input);
  const parameters = {
    'input': req.body.input,
    'context': req.body.context,
    'workspace_id': config.watson.assistant.workspace_id
  };
  assistantMessage(parameters)
    .then(response => {
      console.log(JSON.stringify(response, null, 2));
      res.json(response);
    })
    .catch(err => {
      console.log('error:', err);
    });
}

// convert assistant.message() to Promise
assistantMessage = (params) => {
  return new Promise((resolve, reject) => {
    watsonAssistant.message(params, (err, res) => {
      if (err) {
        reject(err);
      } else {
        resolve(res);
      }
    })
  });
}
```

getIndex() just 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.

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 be answered.

AssistantMessage() converts the watsonAssistant.message() to a Promise to get .then() .catch() processing instead of nested if() … else().

Step 109 In the root folder **create** a folder named *views*.

Step 110 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/bootstrap/4.0.0/css/
bootstrap.min.css" crossorigin="anonymous">
<link rel='stylesheet' href='/stylesheets/styles.css' />
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"
crossorigin="anonymous"></script>
<script src="/js/scripts.js"></script>
  </head>
<body class="container">
  <h2>
    IBM EAG Watson Assistant Lab
  </h2>
```

Step 111 In the *views* folder **create** a *footer.ejs* file with the following content:

```
  <p><small>IBM Ecosystem Advocacy Group - 2019</small></p>
</body>
</html>
```

Step 112 In the *views* folder **create** a *404.ejs* file with the following content:

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

Step 113 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 114 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 115 In this *public* folder **add** the following two folders:

`stylesheets` → public/stylesheets

`js` → public/js

Step 116 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;
}
```

Step 117 In the *js* folder **add a file** *scripts.js* with the following content:

```
var context = {};

function updateChatLog(user, message) {
  if (message) {
    var div = document.createElement("div");
    div.innerHTML = "<b>" + user + "</b>: " + message;
    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.output.text);
      }
    }
  }
  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 118 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 119 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.

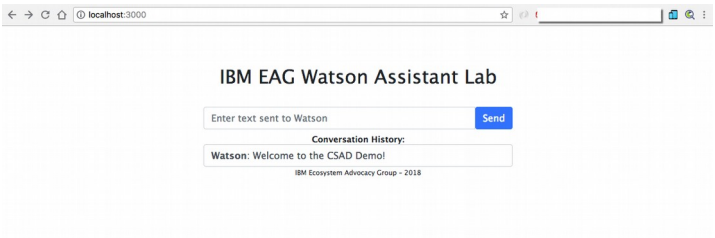
Step 120 Test the application on your local workstation with the following command(s).

```
npm start
```

Step 121 In a web browser, navigate to the following URL. If necessary, refresh the existing page to pick up the changes.

http://localhost:3000/

Step 122 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.



Step 123 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
```

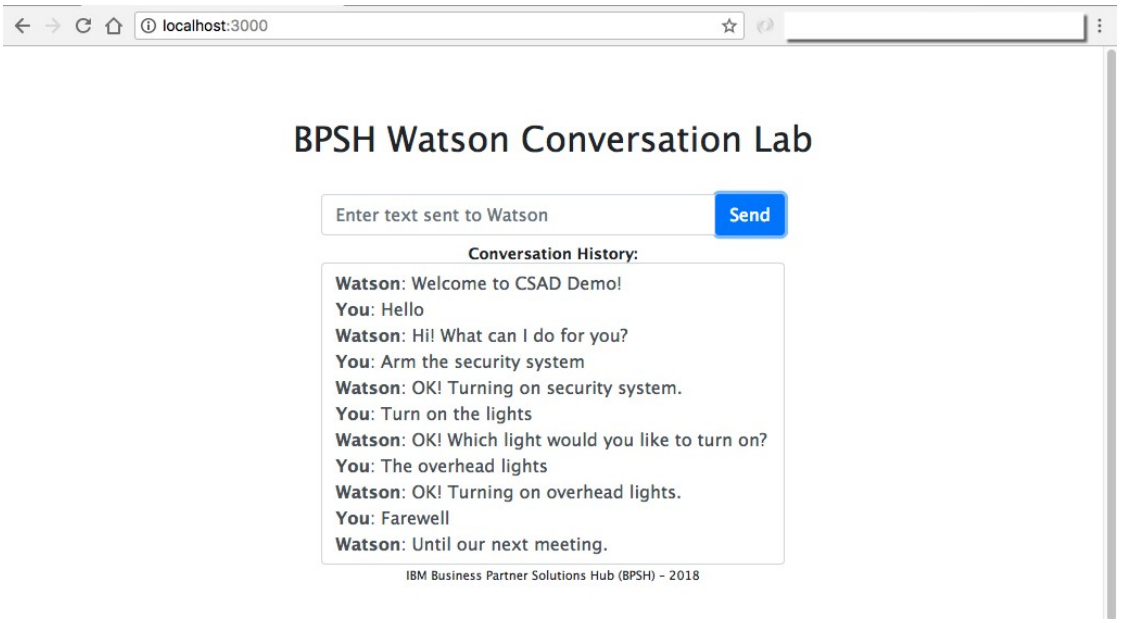
```
Arm the security system
```

```
Turn on the lights
```

```
The headlights
```

```
Farewell
```

Step 124 Feel free to experiment with other phrases to further explore the dialog branches that you created in earlier steps.



Step 125 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 ☺

Step 126 So far, 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 and also in Lab 5, we can see how further backend integration can be achieved.

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.

Step 127 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 128 Locate the following file and open it with a text/code editor.

`…/csadConversation/controllers/watson.js`

Step 129 If you accurately followed the steps in this guide, lines 19 – 20 should contain the application logic that is executed after Watson returns a response. In the .then() block **Insert a blank line so that line 19 is now blank**. This will be the insertion point for your new code snippet. You should also **comment out line 20** since you will be using the server console to display updates from your custom app actions. Your code should now look as follows:

```
17    assistantMessage(parameters)
18      .then(response => {
19
20  //      console.log(JSON.stringify(response, null, 2));
21        res.json(response);
22      })
23      .catch(err => {
24        console.log('error:', err);
25      });
26  }
27
```


Step 130 At line 19, 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 (!response.input.text) {
  return res.json(response);
}
console.log("Detected input: " + response.input.text);
if (response.intents.length > 0) {
  var intent = response.intents[0];
  console.log("Detected intent: " + intent.intent);
  console.log("Confidence: " + intent.confidence);
}
if (response.entities.length > 0) {
  var entity = response.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);
    response.output.text = msg;
  }
}
```

Step 131 Refer to the following table for a description of the code:

Lines	Description
19 – 20	Return immediately when no input text
22	The user input that was passed to the Conversation service
23 – 27	If the Conversation service identified an intent from the input… 23-25: Get the *intent* and print its name and confidence level.
28 – 37	If the Conversation service identified an entity from the input… 28-30: Get the *entity* and print its name and value 31: Check for a specific entity so that your app can respond accordingly 32-33: Get and print the current system time 34: Modify the Watson output with your new custom app response
39	Send the modified JSON response to the client application

Step 132 On line 26, 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 133 **Save** and close the file.

Step 134 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 135 Test the application on your local workstation with the following command(s).

```
npm start
```

Step 136 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 137 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 138 In the screenshots below, the custom application will output messages to the server console based off the returned intent. For example:

(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 139 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: csadConversation-xxx
  path: .
  buildpack: sdk-for-nodejs
  command: node app.js
  memory: 256M
```

Step 140 **Save** the manifest.yml file.

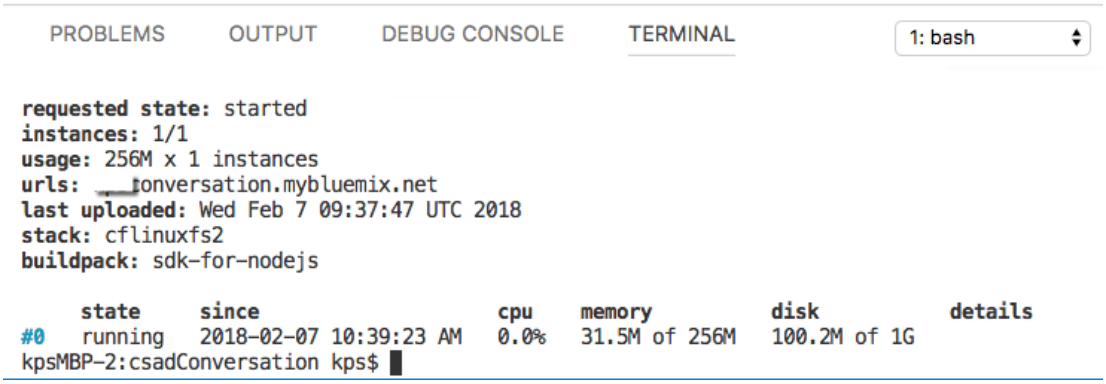
Step 141 Stop your running cloud applications that you do not exceed the memory limit of the IBM Cloud “Lite” account.

Step 142 Make sure you are logged in to your IBM Cloud account.
(**See Workstation Setup document Step 9**)

Step 143 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 144 You will also see the application in your IBM Cloud console.