Watson Assistant

Getting started tutorial

## **Before you begin**

You'll need a service instance to start.

If you created a project with the Watson Assistant service, you're all set with these prerequisites. Go to Step 1.

1. Go to the Watson Developer Console [Services](https://console.eu-gb.bluemix.net/developer/watson/services) page.
2. Select Watson Assistant, click **Add Services**, and either sign up for a free IBM Cloud account or log in.
3. Change the project name to watson-assistant-tutorial, and then click **Create Project**.

If you use IBM Cloud Dedicated, create your service instance from the [Watson Assistant](https://console.eu-gb.bluemix.net/catalog/services/conversation/) page in the Catalog.

## **Step 1: Launch the tool**

After you create a project that includes the Watson Assistant service, you'll land on the project details page. Launch the Watson Assistant tool from here.

Click **Launch Tool** for Watson Assistant under **Services**.

If you're prompted to log into the tool, provide your IBM Cloud credentials.

If you're not at a project details page for the Watson Assistant service, go to the Watson Developer Console [Projects](https://console.eu-gb.bluemix.net/developer/watson/projects) page and select the project.

IBM Cloud Dedicated: Select your service instance from the Dashboard to launch the tooling.

## **Step 2: Create a workspace**

Your first step in the Watson Assistant tool is to create a workspace.

A [workspace](https://console.bluemix.net/docs/services/conversation/configure-workspace.html) is a container for the artifacts that define the conversation flow.

1. In the Watson Assistant tool, click **Create**.
2. Give your workspace the name Watson Assistant tutorial. If the dialog you plan to build will use a language other than English, then choose the appropriate language from the list. Click **Create**. Youʼll land on the**Intents** tab of your new workspace.

## **Step 3: Create intents**

An [intent](https://console.bluemix.net/docs/services/conversation/intents.html) represents the purpose of a user's input. You can think of intents as the actions your users might want to perform with your application.

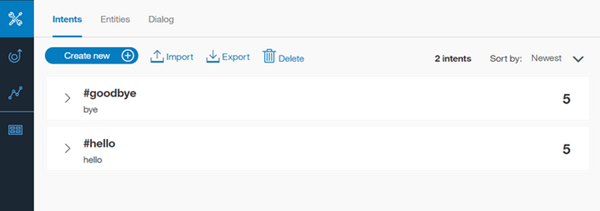
For this example, we're going to keep things simple and define only two intents: one for saying hello, and one for saying goodbye.

1. Make sure you're on the Intents tab. (You should already be there, if you just created the workspace.)
2. Click **Add intent**.
3. Name the intent hello, and then click **Create intent**.
4. Type hello into the **Add user example** field, and then press **Enter**.

Examples tell the Watson Assistant service what kinds of user input you want to match to the intent. The more examples you provide, the more accurate the service can be at recognizing user intents.

1. Add four more examples:
   * good morning
   * greetings
   * hi
   * howdy
2. Click the **Close** Close arrow icon to finish creating the #hello intent.
3. Create another intent named #goodbye with these five examples:
   * bye
   * farewell
   * goodbye
   * I'm done
   * see you later

You've created two intents, #hello and #goodbye, and provided example user input to train Watson to recognize these intents in your users' input.



## **Step 4: Add intents from a content catalog**

Add training data that was built by IBM to your workspace by adding intents from a content catalog. In particular, you will give your assistant access to the eCommerce content catalog so your dialog can address user requests to complete common online transactions.

1. In the Watson Assistant tool, click the **Content Catalog** tab.
2. Find **eCommerce** in the list, and then click **Add to workspace**.
3. Open the **Intents** tab to review the intents and associated example utterances that were added to your training data. You can recognize them because each intent name begins with the prefix #eCommerce\_. You will add the #eCommerce\_Cancel\_Product\_Order intent to your dialog in a later step.

You have successfully supplemented your training data with prebuilt content provided by IBM.

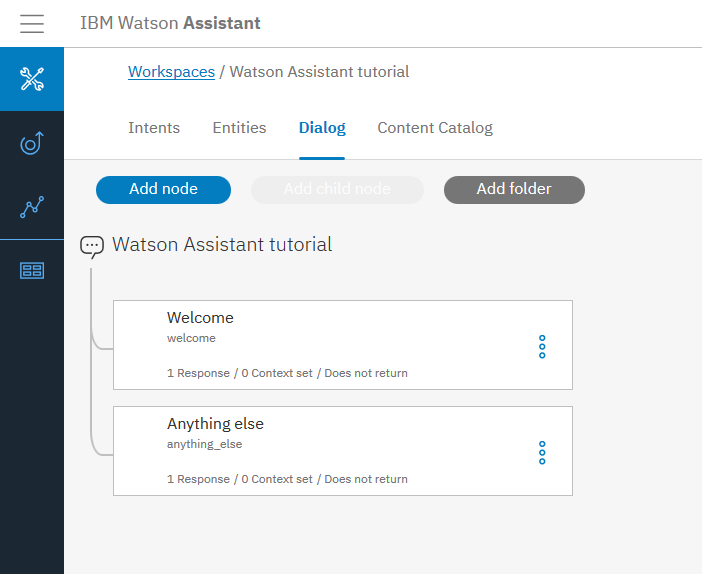
## **Step 5: Build a dialog**

A [dialog](https://console.bluemix.net/docs/services/conversation/dialog-build.html) defines the flow of your conversation in the form of a logic tree. Each node of the tree has a condition that triggers it, based on user input.

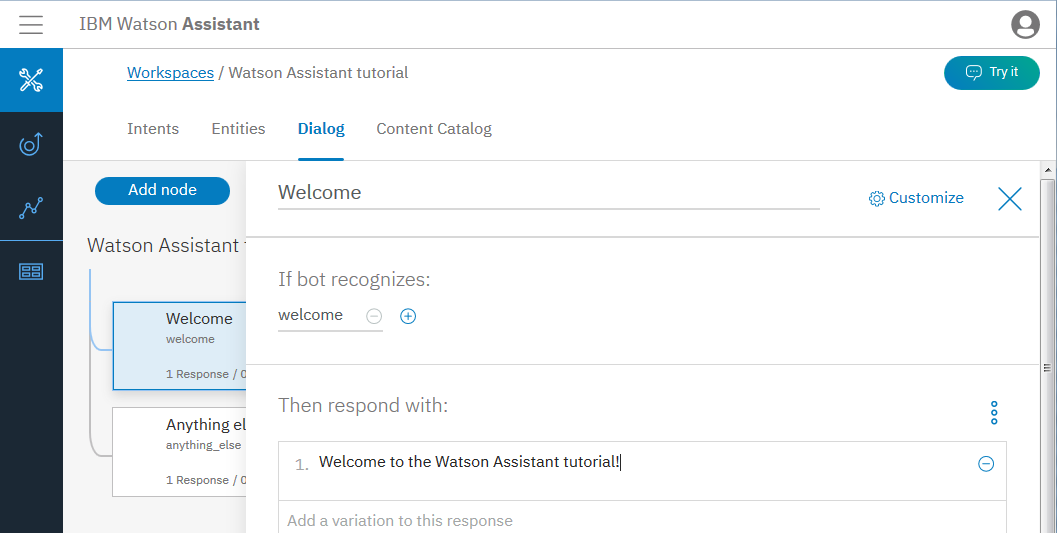
We'll create a simple dialog that handles our #hello and #goodbye intents, each with a single node.

### Adding a start node

1. In the Watson Assistant tool, click the **Dialog** tab.
2. Click **Create**. You'll see two nodes:
   * **Welcome**: Contains a greeting that is displayed to your users when they first engage with the bot.
   * **Anything else**: Contains phrases that are used to reply to users when their input is not recognized.



1. Click the **Welcome** node to open it in the edit view.
2. Replace the default response with the text, Welcome to the Watson Assistant tutorial!.



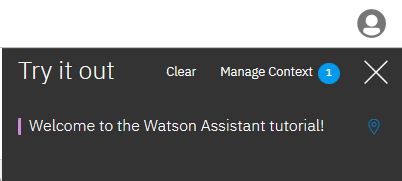
1. Click Close to close the edit view.

You created a dialog node that is triggered by the welcome condition, which is a special condition that indicates that the user has started a new conversation. Your node specifies that when a new conversation starts, the system should respond with the welcome message.

### Testing the start node

You can test your dialog at any time to verify the dialog. Let's test it now.

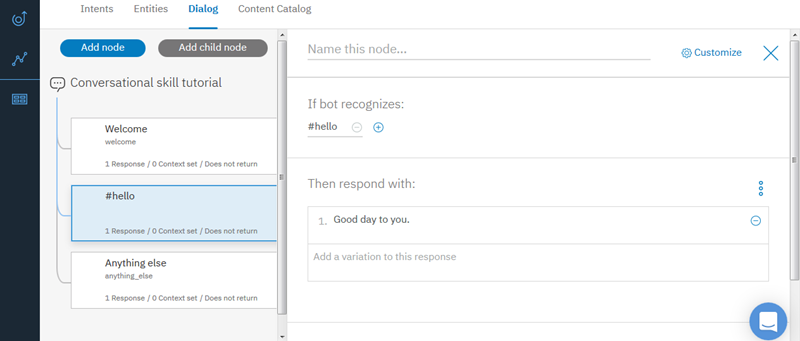
* Click the Ask Watson icon to open the "Try it out" pane. You should see your welcome message.



### Adding nodes to handle intents

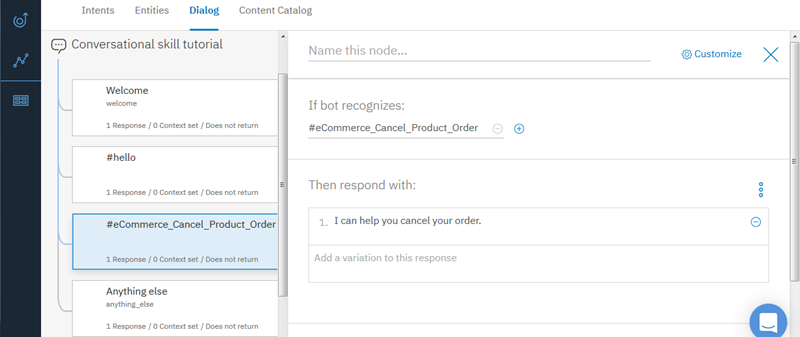
Now let's add nodes to handle our intents between the Welcome node and the Anything else node.

1. Click the More icon More options on the **Welcome** node, and then select **Add node below**.
2. Type #hello in the **Enter a condition** field of this node. Then select the **#hello** option.
3. Add the response, Good day to you.
4. Click Close to close the edit view.

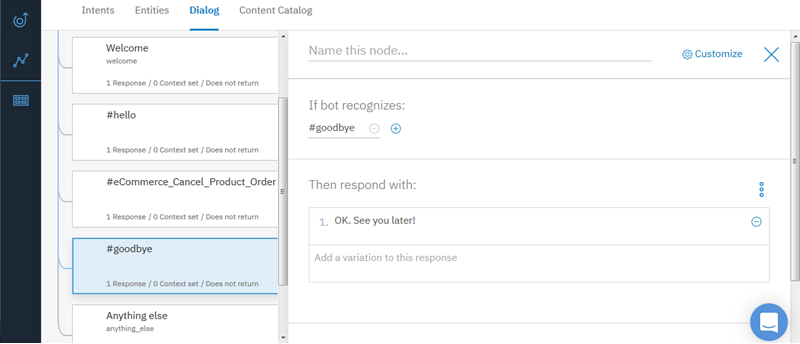


1. Click the More icon More options on this node, and then select **Add node below** to create a peer node. In the peer node, specify #eCommerce\_Cancel\_Product\_Order as the condition.
2. Add the following text as the response.

I can help you cancel your order.



1. Click the More icon More options on this node, and then select **Add node below** to create another peer node. In the peer node, specify #goodbye as the condition, and OK. See you later! as the response.



### Testing intent recognition

You built a simple dialog to recognize and respond to both hello and goodbye inputs. Let's see how well it works.

1. Click the Ask Watson icon to open the "Try it out" pane. There's that reassuring welcome message.
2. At the bottom of the pane, type Hello and press Enter. The output indicates that the #hello intent was recognized, and the appropriate response (Good day to you.) appears.
3. Try the following input:
   * bye
   * howdy
   * see ya
   * good morning
   * sayonara
4. Enter I want to cancel an order I placed. and press Enter. The output indicates that the #eCommerce\_Cancel\_Product\_Order intent was recognized, and the response that you added for it is displayed.

**Note**: In a dialog that is used by an assistant in production, you would likely add more child nodes that collect the order number and any other necessary information from the user, and then make a programmatic call to your order tracking backend service to cancel the order on the user's behalf.

Watson can recognize your intents even when your input doesn't exactly match the examples you included. The dialog uses intents to identify the purpose of the user's input regardless of the precise wording used, and then responds in the way you specify.

### Result of building a dialog

That's it. You created a simple conversation with three intents and a dialog to recognize them.

## **Step 6: Review the sample workspace**

Open the sample workspace to see intents similar to the ones you just created plus many more, and see how they are used in a complex dialog.

1. Go back to the Workspaces page. You can click the Back to workspaces button button from the navigation menu.
2. On the **Car Dashboard - Sample** workspace tile, click the **Edit sample** button.

# Configuring a Watson Assistant workspace

The natural-language processing for the Watson Assistant service happens inside a workspace, which is a container for all of the artifacts that define the conversation flow for an application.

A single Watson Assistant service instance can contain multiple workspaces. A workspace contains the following types of artifacts:

* [**Intents**](https://console.bluemix.net/docs/services/conversation/intents.html): An intent represents the purpose of a user's input, such as a question about business locations or a bill payment. You define an intent for each type of user request you want your application to support. In the tool, the name of an intent is always prefixed with the # character. To train the workspace to recognize your intents, you supply lots of examples of user input and indicate which intents they map to.
* [**Entities**](https://console.bluemix.net/docs/services/conversation/entities.html); An entity represents a term or object that is relevant to your intents and that provides a specific context for an intent. For example, an entity might represent a city where the user wants to find a business location, or the amount of a bill payment. In the tool, the name of an entity is always prefixed with the @ character. To train the workspace to recognize your entities, you list the possible values for each entity and synonyms that users might enter.
* [**Dialog**](https://console.bluemix.net/docs/services/conversation/dialog-build.html): A dialog is a branching conversation flow that defines how your application responds when it recognizes the defined intents and entities. You use the dialog builder in the tool to create conversations with users, providing responses based on the intents and entities that you recognize in their input.

The content catalog contains prebuilt common intents and entities that you can add to your application rather than building your own. For example, most applications require a greeting intent that starts a dialog with the user. You can add the **General** content catalog to add a group of intents that recognize and respond to utterances that are commonly used to start and end a conversation, among other things.

As you add information, the workspace uses this unique data to build a machine learning model that can recognize these and similar user inputs. Each time you add or change the training data, the training process is triggered to ensure that the underlying model stays up-to-date as your customer needs and the topics they want to discuss change.

## **Workspace limits**

The number of workspaces you can create in a single service instance depends on your Watson Assistant service plan. The sample workspace does not count toward your workspace limit unless you edit or duplicate the sample:

| **Service plan** | **Workspaces per service instance** |
| --- | --- |
| Standard/Premium | 20 |
| Lite\* | 5 |

\*After 30 days of inactivity, an unused Lite workspace might be deleted to free up space.

## **Creating workspaces**

You can create a workspace from scratch, use the provided sample workspace, or import a workspace from a JSON file. You can also duplicate an existing workspace within the same service instance.

You use the Watson Assistant tool to create workspaces. Follow these steps to create a workspace:

1. If the "Service Details" page is not already open, click your Watson Assistant service instance from the console. (When you create a service instance, the "Service Details" page is displayed.)
2. Click **Launch tool**.
3. Do one of the following things:
   * To create a workspace from scratch, click **Create**.
   * To use the sample as a starting point for your own workspace, edit the sample workspace. If you want to use it for multiple workspaces, then duplicate it instead.
   * To import a workspace from a JSON file, click the Import workspace icon, and select the JSON file you want to import from.

**Important:**

* + - The imported JSON file must use UTF-8 encoding.
    - The maximum size for a workspace JSON file is 10MB. If you need to import a larger workspace, consider importing the intents and entities separately after you have imported the workspace. (You can also import larger workspaces using the REST API. For more information, see the [API Reference](https://www.ibm.com/watson/developercloud/conversation/api/v1/#create_workspace).)
    - The JSON cannot contain tabs, newlines, or carriage returns.

Specify the data you want to include:

* + - Select **Everything (Intents, Entities, and Dialog)** if you want to import a complete copy of the exported workspace, including the dialog.
    - Select **Intents and Entities** if you want to use the intents and entities from the exported workspace, but you plan to build a new dialog.

1. Specify the details for the new workspace:
   * **Name**: A name no more than 64 characters in length. This value is required.
   * **Description**: A description no more than 128 characters in length.
   * **Language**: The language of the user input the workspace will be trained to understand.

After you create the workspace, it appears as a tile on the Workspaces page.

## **Exporting and copying workspaces**

You can use the Workspaces page to export workspaces and to copy a workspace to a new workspace.

Exporting a workspace saves a copy of all workspace data in a JSON file. The workspace data includes intents and entities that you defined as part of the training data. It also includes counter examples that are created when you identify inaccurate classifications by marking them as irrelevant. Use this option if you want to import the workspace into a different service instance or save a copy of the workspace data offline. When you import a workspace, you can choose to import only the intents and entities, which can be useful if you want to build a new dialog using the same training data.

Copying a workspace makes a complete copy of the workspace within the same service instance. Use this option if you want to adapt an existing workspace while preserving the original version.

* To export an existing workspace to a JSON file, click the menu icon in the workspace tile and then select **Download as JSON**.
* To create a copy of an existing workspace within the same service instance, click the menu icon in the workspace tile and then select **Duplicate**.

Specify the name, description, and language for the new workspace. All data (including intents, entities, and dialog) is included in the copy.

## **Sharing the workspace with team members**

After you create the service instance, you can give other people access to it. Together, you can define the training data and build the dialog.

**Important**: Only one person can edit an intent, entity, or a dialog node at a time. If multiple people work on the same item at the same time, then the changes made by the person who saves their changes last are the only changes applied. Changes that are made during the same time frame by someone else and are saved first are not retained. Coordinate the updates that you plan to make with your team members to prevent anyone from losing their work.

To share a workspace with other people, you must give them developer access to the service instance that hosts the workspace. If the instance you share hosts other workspaces that you do not want others to edit, then be sure to make that clear to anyone you invite.

1. Go to the Watson Developer Console [Projects](https://console.eu-gb.bluemix.net/developer/watson/projects) page, and log in. Click **Manage > Account > Users** from the menu.
2. Click **Invite users**, and then enter the email addresses of the people on your team to whom you want to give access.
3. In the Cloud Foundry access section, choose your organization from the Organization list.

The Organization roles field is automatically filled with Auditor. You can keep the default value in the field.

1. Optionally, limit the access you are granting to the workspaces in a single region and space.
2. In the Space roles field, choose **Developer**.

See [Cloud Foundry roles](https://console.bluemix.net/docs/iam/cfaccess.html#cfroles) for more information about the roles.

1. Click **Invite users**.

When the people you invite next log in to IBM Cloud, your organization will be included in their list of accounts.

With more people contributing to workspace development, unintended changes can occur, including workspace deletions. Consider creating backup copies of your workspace on a regular basis, so you can roll back to an earlier version if necessary. To create a backup, simply export the workspace as a JSON file.

# Planning your intents and entities

To plan the *intents* for your application, you need to consider what your customers might want to do, and what you want your application to be able to handle. Choosing the correct intent for a user's input is the first step in providing a useful response. The intents you identify for your application will determine the dialog flows you need to create; they also might determine which back-end systems your application needs to integrate with in order to complete customer requests (such as customer databases or payment-processing systems).

An *entity* represents a term or object in the user's input that provides clarification or specific context for a particular intent. If intents represent verbs (something a user wants to do), entities represent nouns (such as the object of, or the context for, an action). Entities make it possible for a single intent to represent multiple specific actions. An entity defines a class of objects, with specific values representing possible objects in that class.

Essentially, if you want to capture a request, or perform an action, use an intent. If you want to capture information that may affect how you respond to a request or action, use an entity.

As an example, suppose you want to create a cognitive car dashboard application that enables users to turn accessories on or off:

1. Gather as many actual customer questions, commands, or other inputs as possible. Using input from real users gives a better picture of the expected input than having experts create lists of possible utterances. Remember that customers might phrase the same kind of request in many different ways. For example, the following example utterances all represent requests to turn something off:
   * Headlights off
   * Turn the radio off
   * Stop the air conditioner
2. Once you have a list of examples, sort them into categories based on the capabilities you want your application to support; these categories represent the intents you will define, while the examples will help your application to identify those intents in new input.

Do not make your intents too similar. Similar intents can be difficult for the Watson Assistant service to distinguish. If you find that you have several intents that are close in meaning, consider whether you could combine them into a single intent, and then use entities to provide multiple possible responses for that intent.

Since the previous examples all represent the same intent (to turn something off), you might call this category #turn\_off. Remember that a customer's input does not have to be an exact match for any of your examples; intents are recognized using natural-language processing.

1. You would then use an entity to represent what the user wants to turn off. To do this, you might create an entity called @accessory, and give it the following possible values:
   * headlights
   * radio
   * air conditioner

For each value, you can also specify synonyms. For example, you might specify AC and cooling as synonyms for air conditioner; all three strings are treated as the same value. Listing multiple ways that users might refer to the same value helps improve your application's accuracy.

1. When the user's input is received, the conversation recognizes both intents and entities. Your dialog flow can then use these to provide the best answer. You should only create an entity for something that matters in terms of changing the way the application responds to an intent. In addition to the entities you define, the service provides a set of system entities that are already defined and available for you to use. For more information, see [Enabling system entities](https://console.bluemix.net/docs/services/conversation/entities.html#enable_system_entities).
2. Continue to refine your intents, entities, and examples as needed. Do not think of your set of intents and entities as finished products. It is likely that when you [build your dialogs](https://console.bluemix.net/docs/services/conversation/dialog-build.html), you will identify additional intents and entities that you need to add. You can also continue to gather input from new customers, and use it to add new examples; this iterative process improves your application's ability to recognize intents and entities accurately.

# Defining intents

**ntents** are purposes or goals expressed in a customer's input, such as answering a question or processing a bill payment. By recognizing the intent expressed in a customer's input, the Watson Assistant service can choose the correct dialog flow for responding to it.

## **Intent limits**

The number of intents and examples you can create depends on your Watson Assistant service plan:

| **Service plan** | **Intents per workspace** | **Examples per workspace** |
| --- | --- | --- |
| Standard/Premium | 2,000 | 25,000 |
| Lite | 100 | 25,000 |

## **Creating intents**

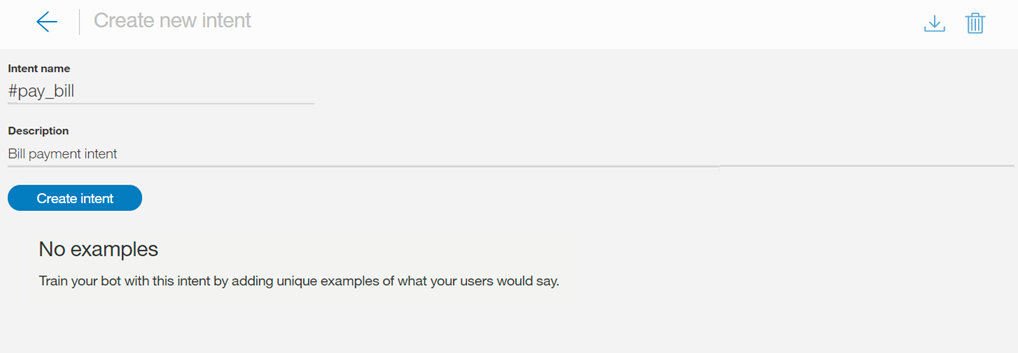
Use the Watson Assistant tool to create intents.

1. In the Watson Assistant tool, open your workspace and then select the **Intents** tab in the navigation bar. If **Intents** is not visible, use the Menu menu to open the page.
2. Select **Create new**.
3. In the **Intent name** field, type a name for the intent.
   * The intent name can contain letters (in Unicode), numbers, underscores, hyphens, and periods.
   * The name cannot consist of .. or any other string of only periods.
   * Intent names cannot contain spaces and must not exceed 128 characters. The following are examples of intent names:
     + #weather\_conditions
     + #pay\_bill
     + #escalate\_to\_agent

The tooling automatically includes the # character in the intent names, so you do not have to add one.

Add a description of the intent in the **Description** field.

1. Select **Create intent** to save your intent name.



1. Next, in the **Add user examples** field, type the text of a user example for the intent. An example can be any string up to 1024 characters in length. The following might be examples for the #pay\_bill intent:
   * I need to pay my bill.
   * Pay my account balance
   * make a payment

**Referencing entities and synonyms as intent examples**

If you have defined, or plan to define, entities that correspond to this intent, refer to the entities, or their associated synonyms, in some of the examples. Doing so helps to establish a relationship between the intent and entities.



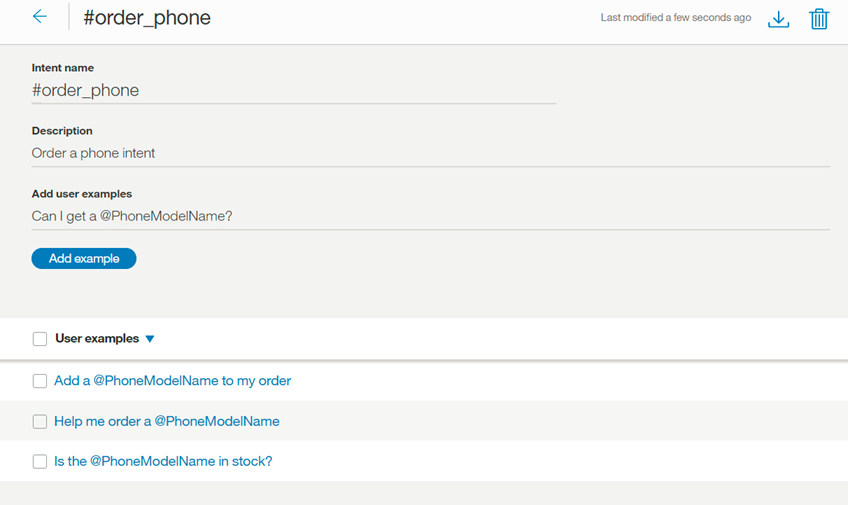
Important:

* + Intent example data should be representative and typical of data that end users will provide. Examples can be collected from actual user data, or from people who are experts in your specific field. The representative and accurate nature of the data is important.
  + Both training and test data (for evaluation purposes) should reflect the distribution of intents in real usage. Generally, more frequent intents have relatively more examples, and better response coverage.
  + You can include punctuation in the example text, as long as it appears naturally. If you believe that some users will express their intents with examples that include punctuation, and some users will not, include both versions. Generally, the more coverage for various patterns, the better the response.

**Directly referencing an @Entity as an intent example**

You may also choose to directly reference entities in your intent examples. For instance, say you have an entity called @PhoneModelName, which contains values Galaxy S8, Moto Z2, LG G6, and Google Pixel 2. When you create an intent, for example #order\_phone, you could then provide training data as follows:

* + Can I get a @PhoneModelName?
  + Help me order a @PhoneModelName.
  + Is the @PhoneModelName in stock?
  + Add a @PhoneModelName to my order.



**Note**: Currently, you can only directly reference closed entities that you define (pattern values will be ignored). You cannot use [system entities](https://console.bluemix.net/docs/services/conversation/system-entities.html).

If you choose to reference an entity as an intent example (for example, @PhoneModelName) anywhere in your training data it cancels out the value of using a direct reference (for example, Galaxy S8) in an intent example anywhere else. All intents will then use the entity-as-an-intent-example approach; you cannot select this approach for a specific intent only.

In practice, this means that if you have previously trained most of your intents based on direct references (Galaxy S8), and you now use entity references (@PhoneModelName) for just one intent, that would impact all your previous training. For example, if you also have an entity @ServiceCarrier, with values Verizon, AT&T, and Sprint, and you add a reference to @PhoneModelName in one utterance of the intent training, the intent classifier will treat literal mentions (Verizon, AT&T, and Sprint) in the intent training differently. If you do choose to use @Entity references, you need to be careful to replace all previous direct references with @Entityreferences.

**Note**: Defining one example intent with an @Entity that has 10 values defined for it **does not** equate to specifying that example intent 10 times. The Watson Assistant service does not give that much weight to that one example intent syntax.

**Important**: Intent names and example text can be exposed in URLs when an application interacts with the service. Do not include sensitive or personal information in these artifacts.

Press **Add example** to save the example.

1. Repeat the same process to add more examples. You can tab between each example. Provide at least 5 examples for each intent. The more examples you provide, the more accurate your application can be.
2. When you have finished adding examples, select Close arrow to finish creating the intent.

### Results

The intent you created is added to the Intents tab, and the system begins to train itself on the new data.

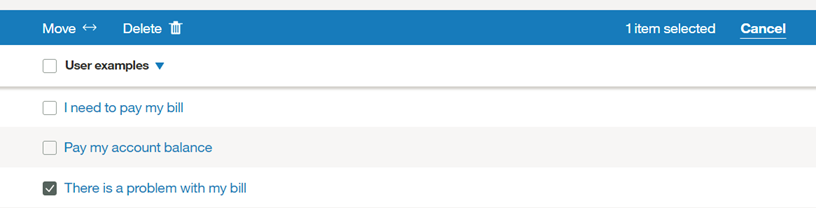
## **Editing intents**

You can select any intent in the list to open it for editing. You can make the following changes:

* Rename the intent.
* Delete the intent.
* Add, edit, or delete examples.
* Move an example to a different intent.

You can tab from the intent name to each example, editing the examples if you choose.

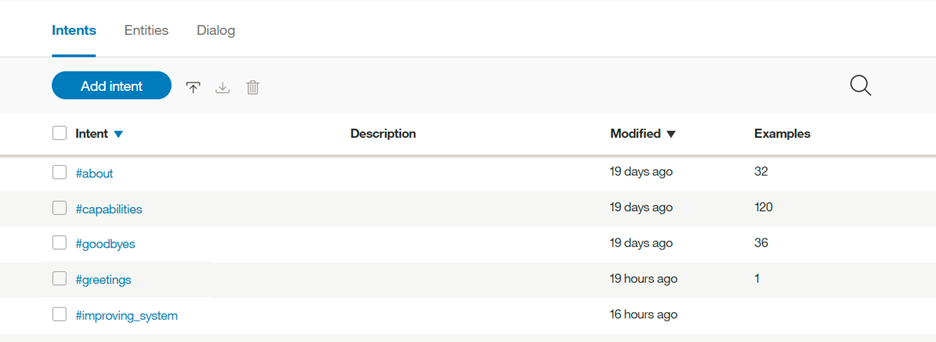
To move or delete an example, select the example by selecting the check box and then select **Move** or **Delete**.



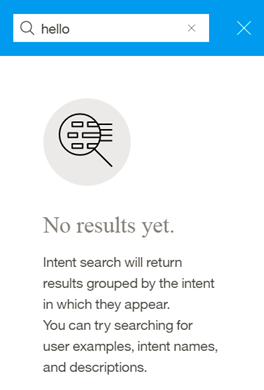
## **Searching intents**

Use the Search feature to find user examples, intent names, and descriptions.

1. Select the **Intents** tab in the navigation bar.



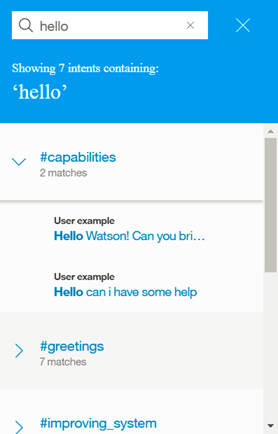
1. Select the Search icon: Search icon
2. Enter a search term or phrase.



**Note**: The first time you search, an index is created; you may see a message to wait while your contents are being indexed.

### Results

Intents containing your search term, with corresponding examples, are shown. Select any result to open it for editing.



## **Importing intents and examples**

If you have a large number of intents and examples, you might find it easier to import them from a comma-separated value (CSV) file than to define them one by one in the Watson Assistant tool.

1. Collect the intents and examples into a CSV file, or export them from a spreadsheet to a CSV file. The required format for each line in the file is as follows:
2. <example>,<intent>

where <example> is the text of a user example, and <intent> is the name of the intent you want the example to match. For example:

Tell me the current weather conditions.,weather\_conditions

Is it raining?,weather\_conditions

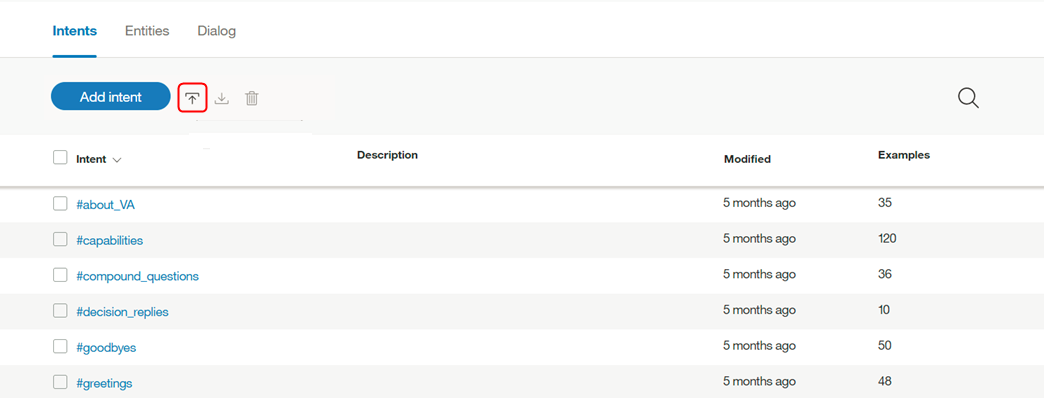
What's the temperature?,weather\_conditions

Where is your nearest location?,find\_location

Do you have a store in Raleigh?,find\_location

**Important:** Save the CSV file with UTF-8 encoding and no byte order mark (BOM).

1. In the Watson Assistant tool, open your workspace and then select the **Intents** tab in the navigation bar. If **Intents** is not visible, use the Menu menu to open the page.
2. Select the Import icon Import icon. Then, drag a file or browse to select a file from your computer. The file is validated and imported, and the system begins to train itself on the new data.



**Important:** The maximum CSV file size is 10MB. If your CSV file is larger, consider splitting it into multiple files and importing them separately.

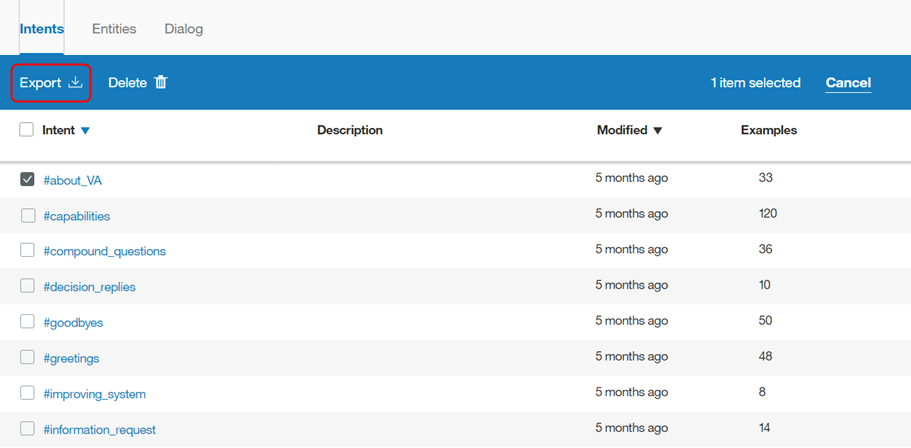
### Results

You can view the imported intents and the corresponding examples on the **Intents** tab. You might need to refresh the page in order to see the new intents and examples.

## **Exporting intents**

You can export a number of intents to a CSV file, so you can then import and reuse them for another Watson Assistant application.

1. On the Intents tab, select the intents you want from the list and choose Export.

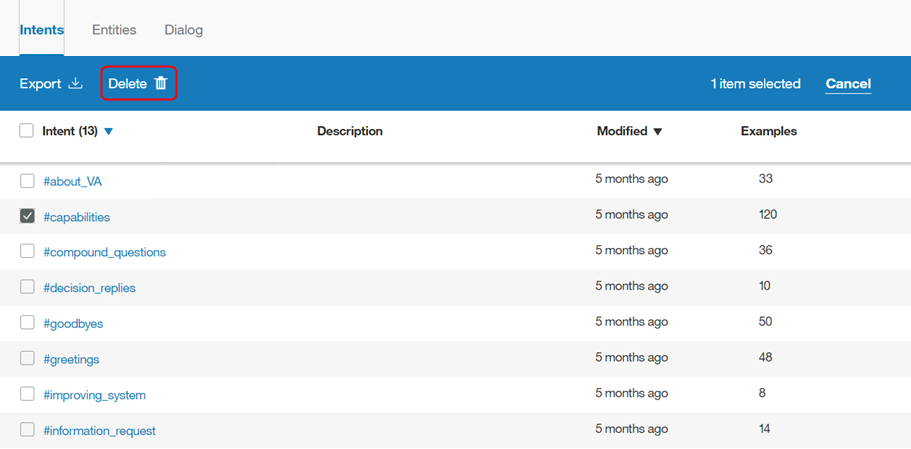


## **Deleting intents**

You can select a number of intents for deletion.

**IMPORTANT**: By deleting intents you are also deleting all associated examples, and these items cannot be retrieved later. All dialog nodes that reference these intents must be updated manually to no longer reference the deleted content.

1. On the Intents tab, select the intents you want from the list and choose Delete.

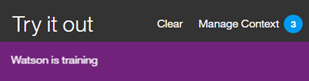


## **Testing your intents**

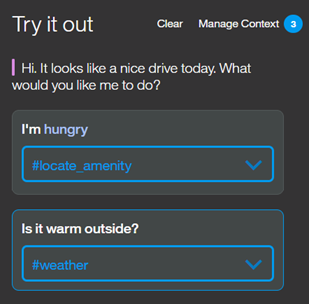
After you have finished creating new intents, you can test the system to see if it recognizes your intents as you expect.

1. In the Watson Assistant tool, select the Ask Watson icon.
2. In the Try it out pane, enter a question or other text string and press Enter to see which intent is recognized. If the wrong intent is recognized, you can improve your model by adding this text as an example to the correct intent.

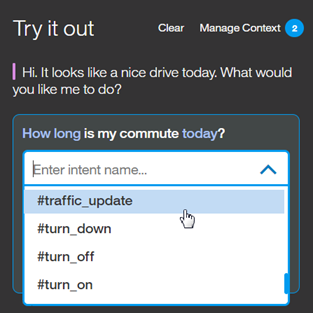
If you have recently made changes in your workspace, you might see a message indicating that the system is still retraining. If you see this message, wait until training completes before testing:



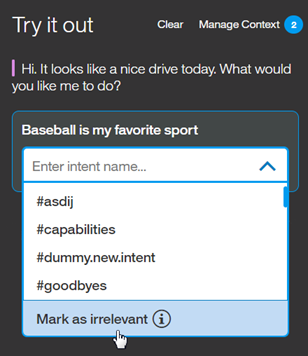
The response indicates which intent was recognized from your input.



1. If the system did not recognize the correct intent, you can correct it. To correct the recognized intent, select the displayed intent and then select the correct intent from the list. After your correction is submitted, the system automatically retrains itself to incorporate the new data.



1. If the input is unrelated to your application, you can indicate that. Select the displayed intent and choose **Mark as irrelevant**.



If your intents are not being correctly recognized, consider making the following kinds of changes:

* Add the unrecognized text as an example to the correct intent.
* Move existing examples from one intent to another.
* Consider whether your intents are too similar, and redefine them as appropriate.

## **Absolute scoring and Mark as irrelevant**

As of February 2017, there is a new algorithm for scoring intent confidence and returning intents. You can also mark inputs as irrelevant. These changes might require you to [upgrade to your workspace](https://console.bluemix.net/docs/services/conversation/upgrading.html).

### Absolute scoring

The Watson Assistant service now scores each intent’s confidence on its own, not in relation to other intents. This allows the flexibility to have multiple intents returned. It also means the system may not return an intent at all. If the top intent has low confidence that any intents relate to the user’s input (less than 0.2), it will still get included in the intents array output by the API, but conditioning on that #intent will return false. If you want to detect the case when no intents were detected with good confidence, you can condition on irrelevant.

As intent confidence scores change, your dialogs may need restructuring. For example, if you conditioned your dialog with an intent that now has low confidence, the system’s response will no longer be correct.

### Mark as irrelevant

Refer to [supported languages](https://console.bluemix.net/docs/services/conversation/lang-support.html) for the availability of this feature.

After you upgrade your workspace, you can [test input](https://console.bluemix.net/docs/services/conversation/intents.html#testing-your-intents) in the Try it out pane to see the changes. You can use **Mark as irrelevant** to indicate that the input is not related to your application.

If you have an intent, such as #off\_topic, for those inputs that are out of scope or off topic, delete the intent and test your workspace by marking the inputs as irrelevant.

**Important**: Intents that are marked as irrelevant are saved as counterexamples in the JSON workspace, and are included as part of the training data. Be sure that you want to make any changes.

* The inputs cannot be accessed or changed later in the tooling.
* The only way to remove the **Irrelevant** tag is to use the same input in the Try it out pane, and then change the intent.

# Defining entities

**Entities** represent a class of object or a data type that is relevant to a user's purpose. By recognizing the entities that are mentioned in the user's input, the Watson Assistant service can choose the specific actions to take to fulfill an intent.

## **Entity limits**

The number of entities, entity values, and synonyms that you can create depends on your Watson Assistant service plan:

| **Service plan** | **Entities per workspace** | **Entity values per workspace** | **Entity synonyms per workspace** |
| --- | --- | --- | --- |
| Standard/ Premium | 1000 | 100,000 | 100,000 |
| Lite | 25 | 100,000 | 100,000 |

System entities that you enable for use count toward your plan usage totals.

## **Creating entities**

Use the Watson Assistant tool to create entities.

1. In the Watson Assistant tool, open your workspace and then click the **Entities** tab. If **Entities** is not visible, use the Menu menu to open the page.
2. Click **Add entity**.

You can also click **Use System Entities** to select from a list of common entities, provided by IBM, that can be applied to any use case. See [Enabling system entities](https://console.bluemix.net/docs/services/conversation/entities.html#enable_system_entities) for more detail.

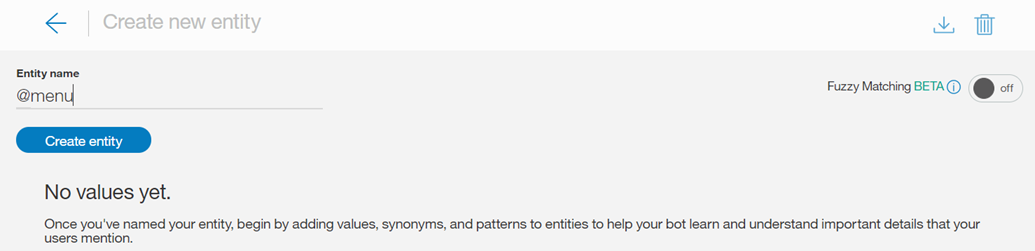
1. In the **Entity name** field, type a descriptive name for the entity.

The entity name can contain letters (in Unicode), numbers, underscores, and hyphens. For example:

* + @location
  + @menu\_item
  + @product

Don't include the @ character in the entity names when you create them in the Watson Assistant tool. Entity names can't contain spaces or be longer than 64 characters. And entity names can't begin with the string sys-, which is reserved for system entities.

1. Select **Create entity**.



1. In the **Value name** field, type the text of a possible value for the entity and hit the Enter key. An entity value can be any string up to 64 characters in length.

**Important:** Don't include sensitive or personal information in entity names or values. The names and values can be exposed in URLs in an app.

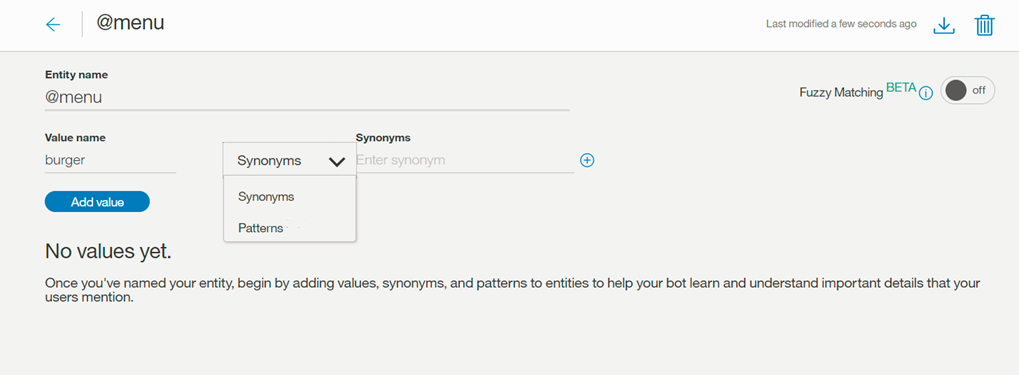
1. For **Fuzzy Matching**, click the button to select either on or off; fuzzy matching is off by default. This feature is available for languages noted in the [Supported languages](https://console.bluemix.net/docs/services/conversation/lang-support.html) topic.

You can turn on fuzzy matching to improve the ability of the service to recognize user input terms with syntax that is similar to the entity, but without requiring an exact match. There are three components to fuzzy matching - stemming, misspelling, and partial matching:

* + Stemming - The feature recognizes the stem form of entity values that have several grammatical forms. For example, the stem of 'bananas' would be 'banana', while the stem of 'running' would be 'run'.
  + Misspelling - The feature is able to map user input to the appropriate corresponding entity despite the presence of misspellings or slight syntactical differences. For example, if you define giraffe as a synonym for an animal entity, and the user input contains the terms giraffes or girafe, the fuzzy match is able to map the term to the animal entity correctly.
  + Partial match - With partial matching, the feature automatically suggests substring-based synonyms present in the user-defined entities, and assigns a lower confidence score as compared to the exact entity match.

**Note** - For English, fuzzy matching prevents the capturing of some common, valid English words as fuzzy matches for a given entity. This feature uses standard English dictionary words. You can also define an English entity value/synonym, and fuzzy matching will match only your defined entity value/synonym. For example, fuzzy matching may match the term unsure with insurance; but if you have unsure defined as a value/synonym for an entity like @option, then unsure will always be matched to @option, and not to insurance.

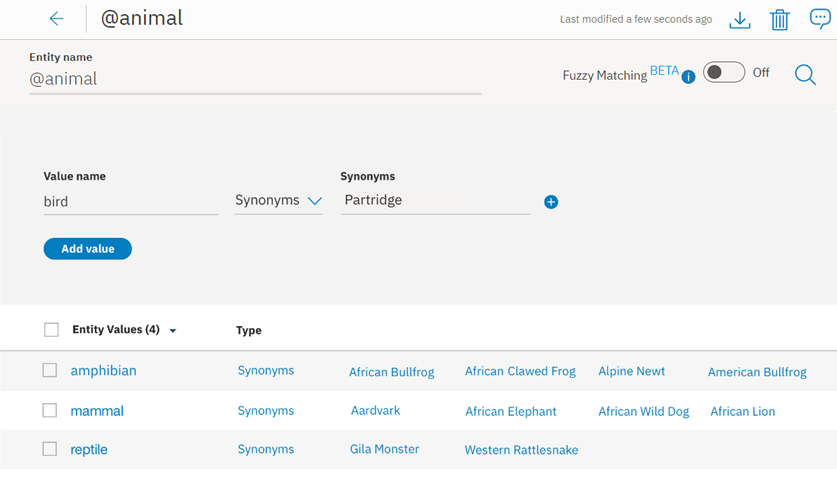
1. Once you have entered a value name, you can then add any synonyms, or define specific patterns, for that entity value by selecting either Synonyms or Patterns from the Type drop-down menu.



**Note:** You can add either synonyms or patterns for a single entity value, you cannot add both.

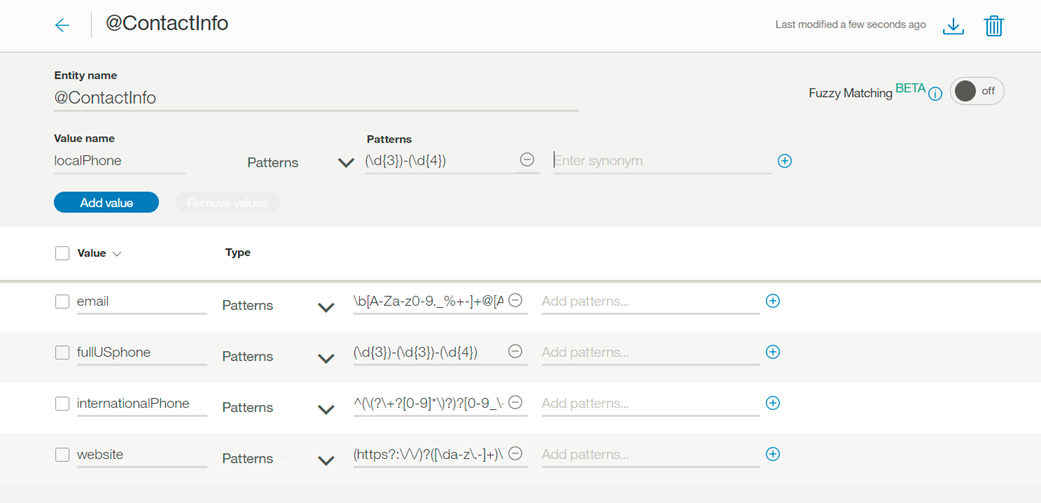
**Synonyms**

* + In the **Synonyms** field, type any synonym for the entity value. A synonym can be any string up to 64 characters in length.



**Patterns**

* + The **Patterns** field lets you define specific patterns for an entity value. A pattern **must** be entered as a regular expression in the field.
    - For each entity value, there can be a maximum of up to 5 patterns.
    - Each pattern (regular expression) is limited to 128 characters.



As in this example, for entity ContactInfo, the patterns for phone, email, and website values can be defined as follows:

* + - Phone
      * localPhone: (\d{3})-(\d{4}), e.g. 426-4968
      * fullUSphone: (\d{3})-(\d{3})-(\d{4}), e.g. 800-426-4968
      * internationalPhone: ^(\(?\+?[0-9]\*\)?)?[0-9\_\- \(\)]\*$, e.g., +44 1962 815000
    - email: \b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b, e.g. name@ibm.com
    - website: (https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]\*)\*\/?$, e.g. [https://www.ibm.com](https://www.ibm.com/)

Often when using pattern entities, it will be necessary to store the text that matches the pattern in a context variable (or action variable), from within your dialog tree.

Imagine a case where you are asking a user for their email address. The dialog node condition will contain a condition similar to @contactInfo:email. In order to assign the user-entered email as a context variable, the following syntax can be used to capture the pattern match within the dialog node's response section:

{

"context" : {

"email": "<? @contactInfo.literal ?>"

}

}

Capture groups - For regular expressions, any part of a pattern inside a pair of normal parentheses will be captured as a group. For example, the entity value fullUSphone contains three captured groups:

* + - (\d{3}) - US area code
    - (\d{3}) - Prefix
    - (\d{4}) - Line number

Grouping can be helpful if, for example, you wanted the Watson Assistant service to ask users for their phone number, and then use only the area code of their provided number in a response.

In order to assign the user-entered area code as a context variable, the following syntax can be used to capture the group match within the dialog node's response section:

{

"context" : {

"area\_code": "<? @fullUSphone.groups[1] ?>"

}

}

For additional information about using capture groups in dialog runtime, see [Storing pattern entity values in context variables](https://console.bluemix.net/docs/services/conversation/dialog-overview-context-groups.html).

The pattern matching engine employed by the Watson Assistant service has some syntax limitations, which are necessary in order to avoid performance concerns which can occur when using other regular expression engines.

* + - Entity patterns may not contain:
      * Positive repetitions (for example x\*+)
      * Backreferences (for example \g1)
      * Conditional branches (for example (?(cond)true))
    - When a pattern entity starts or ends with a Unicode character, and includes word boundaries, for example \bš\b, the pattern match does not match the word boundary correctly. In this example, for input š zkouška, the match returns Group 0: 6-7 š (š zkou***š***ka), instead of the correct Group 0: 0-1 š (***š*** zkouška).

The regular expression engine is loosely based on the Java regular expression engine. The Watson Assistant service will produce an error if you try to upload an unsupported pattern, either via the API or from within the Watson Assistant service Tooling UI.

1. Click **Add value** and repeat the process to add more entity values.
2. When you are finished adding entity values, select Close arrow to finish creating the entity.

### Results

The entity you created is added to the **Entities** tab, and the system begins to train itself on the new data.

## **Editing entities**

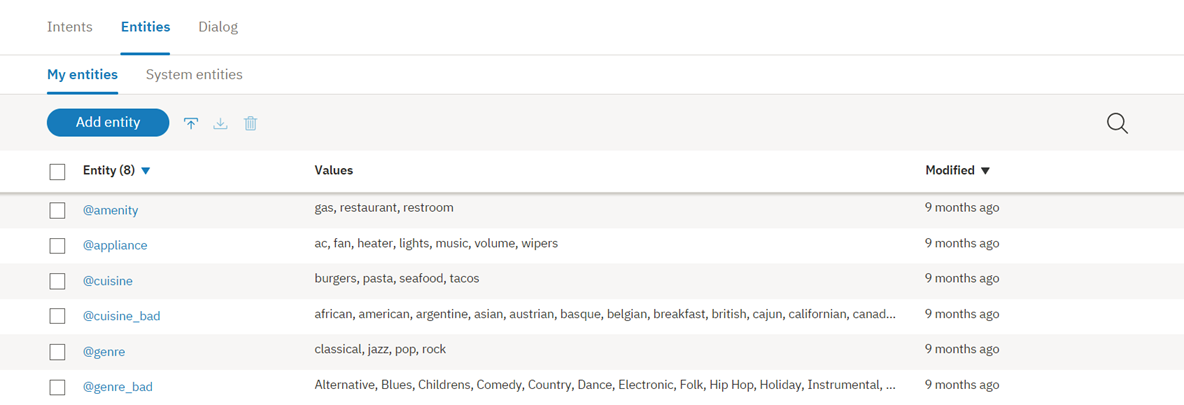
You can click any entity in the list to open it for editing. You can rename or delete entities, and you can add, edit, or delete values, synonyms, or patterns.

**Note**: If you change the entity type from synonym to pattern, or vice versa, the existing values are converted, but might not be useful as-is.

## **Searching entities**

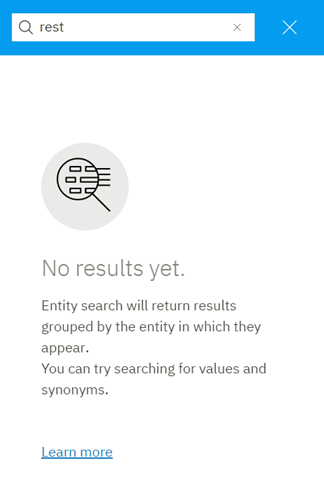
Use the Search feature to find entity values and synonyms.

1. Select the **Entities** tab in the navigation bar, then My Entities.



**Note**: System entities are not searchable.

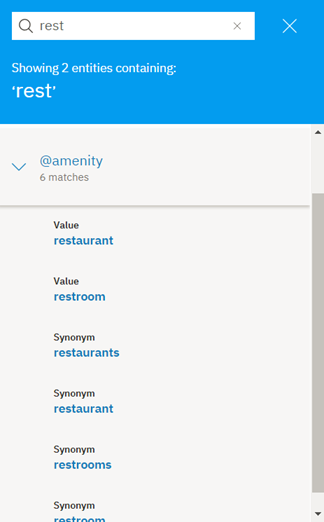
1. Select the Search icon: Search icon
2. Enter a search term or phrase.



**Note**: The first time you search, an index is created; you may see a message to wait while your contents are being indexed.

### Results

Entities containing your search term, with corresponding examples, are shown. Select any result to open it for editing.



## **Importing entities**

If you have a large number of entities, you might find it easier to import them from a comma-separated value (CSV) file than to define them one by one in the Watson Assistant tool.

1. Collect the entities into a CSV file, or export them from a spreadsheet to a CSV file. The required format for each line in the file is as follows:
2. <entity>,<value>,<synonyms>

where <entity> is the name of an entity, <value> is a value for the entity, and <synonyms> is a comma-separated list of synonyms for that value.

weekday,Monday,Mon

weekday,Tuesday,Tue,Tues

weekday,Wednesday,Wed

weekday,Thursday,Thur,Thu,Thurs

weekday,Friday,Fri

weekday,Saturday,Sat

weekday,Sunday,Sun

month,January,Jan

month,February,Feb

month,March,Mar

month,April,Apr

month,May

Importing a CSV file also supports patterns. Any string wrapped with / will be considered a pattern (as opposed to a synonym).

ContactInfo,localPhone,/(\d{3})-(\d{4})/

ContactInfo,fullUSphone,/(\d{3})-(\d{3})-(\d{4})/

ContactInfo,internationalPhone,/^(\(?\+?[0-9]\*\)?)?[0-9\_\- \(\)]\*$/

ContactInfo,email,/\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b/

ContactInfo,website,/(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]\*)\*\/?$/

Save the CSV file with UTF-8 encoding and no byte order mark (BOM). The maximum CSV file size is 10MB. If your CSV file is larger, consider splitting it into multiple files and importing them separately. In the Watson Assistant tool, open your workspace and then click the **Entities** tab.

1. Click Import and then drag a file, or browse to select a file from your computer. The file is validated and imported, and the system begins to train itself on the new data.

### Results

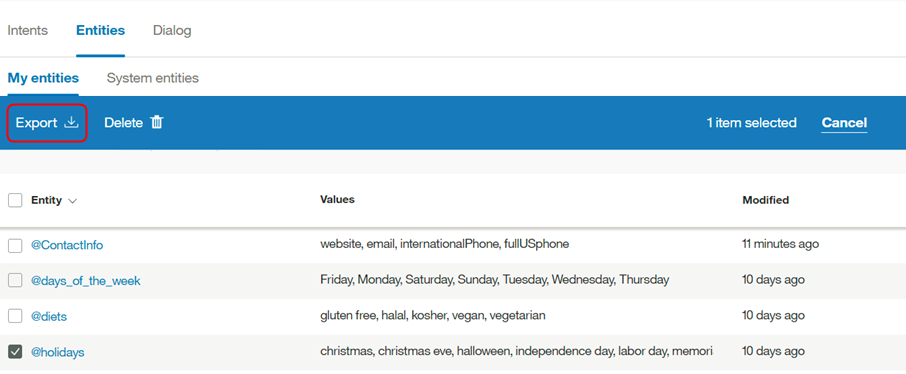
You can view the imported entities on the Entities tab. You might need to refresh the page to see the new entities.

## **Exporting entities**

You can export a number of entities to a CSV file, so you can then import and reuse them for another Watson Assistant application.

Exporting a CSV file supports patterns. Any string wrapped with / will be considered a pattern (as opposed to a synonym).

1. Select the entities you want, then select **Export**.

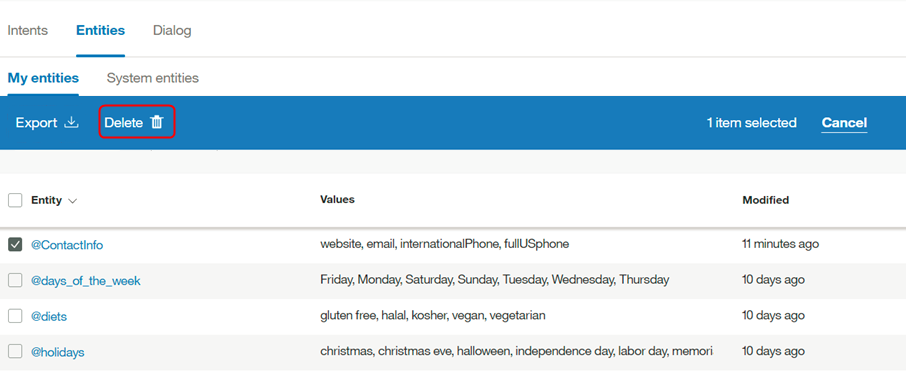


## **Deleting entities**

You can select a number of entities for deletion.

**IMPORTANT**: By deleting entities you are also deleting all associated values, synonyms, or patterns, and these items cannot be retrieved later. All dialog nodes that reference these entities or values must be updated manually to no longer reference the deleted content.

1. Select the entities you want, then select **Delete**.



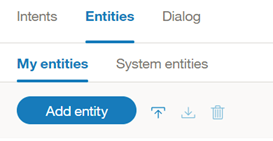
## **Enabling system entities**

The Watson Assistant service provides a number of system entities, which are common entities that you can use for any application. Enabling a system entity makes it possible to quickly populate your workspace with training data that is common to many use cases.

System entities can be used to recognize a broad range of values for the object types they represent. For example, the @sys-number system entity matches any numerical value, including whole numbers, decimal fractions, or even numbers written out as words.

System entities are centrally maintained, so any updates are available automatically. You cannot modify system entities.

1. On the Entities tab, click **System entities**.



1. Browse through the list of system entities to choose the ones that are useful for your application.
   * To see more information about a system entity, including examples of matching input, click the entity in the list.
   * For details about the available system entities, see [System entities](https://console.bluemix.net/docs/services/conversation/system-entities.html).
2. Click the toggle switch next to a system entity to enable or disable it.

### Results

After you enable system entities, the Watson Assistant service begins retraining. After training is complete, you can use the entities.

# Using content catalogs

**Content Catalogs** provide an easy way to add common intents to your Watson Assistant service workspace.

**IMPORTANT**: Content Catalog intents are meant to provide a starting point, and not meant to be fully built-out for production use. It is recommended that you review and expand on these intents, to make them better suited to how your application will use them.

## **Adding a content catalog to your workspace**

Use the Watson Assistant tool to add content catalogs.

1. In the Watson Assistant tool, open your workspace and then select the **Content Catalog** tab in the navigation bar.
2. Select a content catalog, such as Banking, to see the intents that are provided with it.

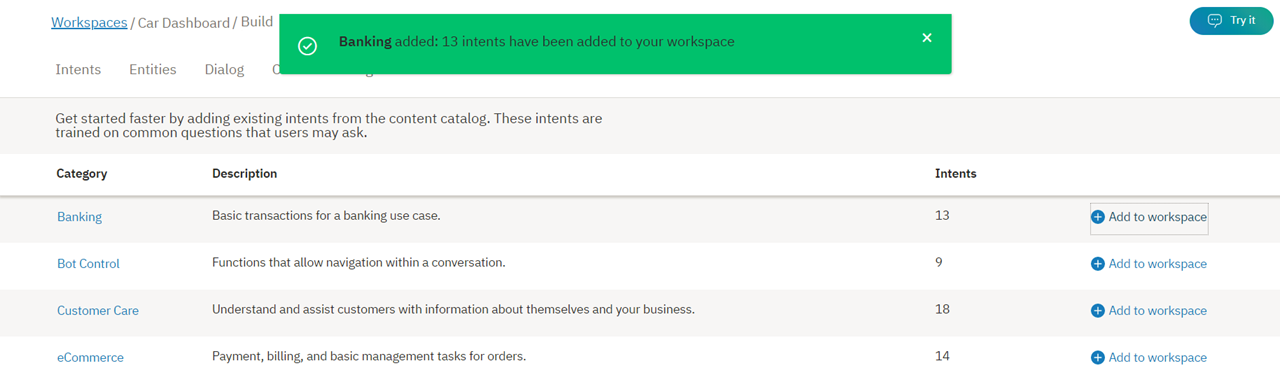


You will see information about the intents that are defined in the Banking category.

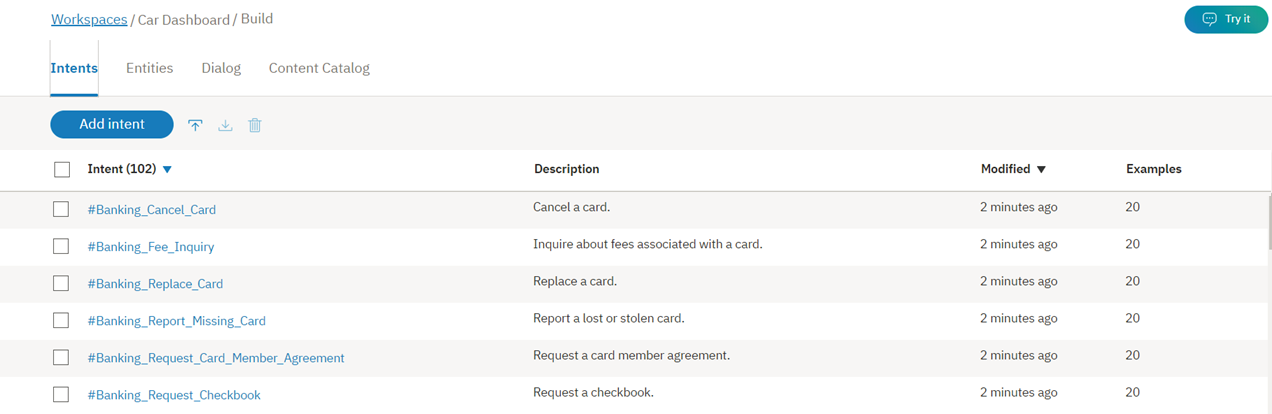


Intents that are added from a content catalog are distinguishable from other intents by their naming convention; in this case, #Banking\_ . . .

1. Select Close arrow to return to the **Content Catalog** tab.
2. Next, add the Banking content catalog to your workspace by clicking the Add to workspace button. You will see a message indicating that the Banking intents have been added to your workspace.



1. Now, select the **Intents** tab, and verify that the Banking intents have been added to your workspace.



### Results

The intents from the Banking content catalog have been added to the **Intents** tab of your workspace, and the system begins to train itself on the new data.

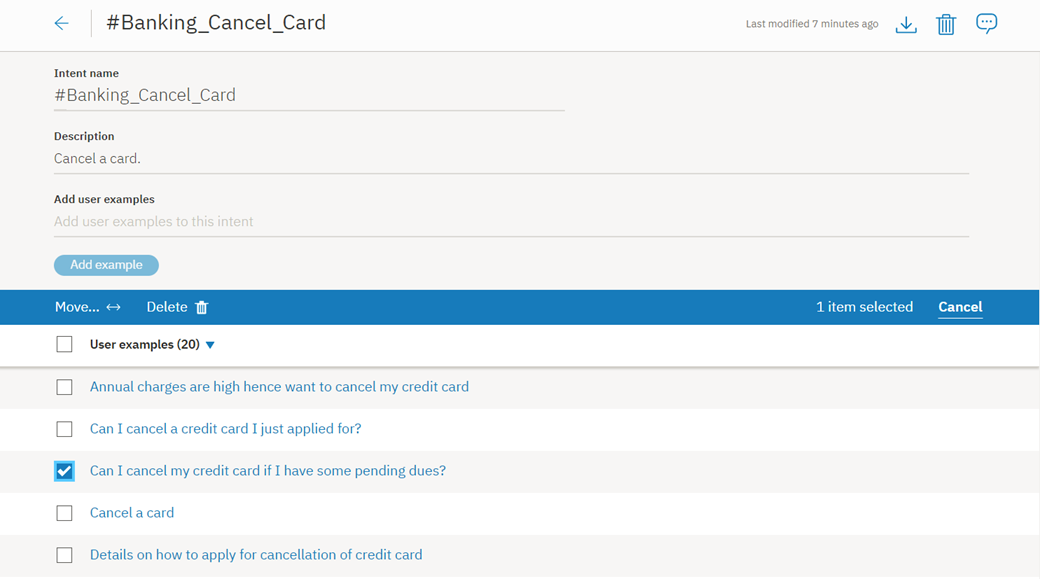
## **Editing content catalog examples**

Like any other intent, once the Banking content catalog intents have been added to your workspace, you can make the following changes:

* Rename the intent.
* Delete the intent.
* Add, edit, or delete examples.
* Move an example to a different intent.

You can tab from the intent name to each example, editing the examples if you choose.

To move or delete an example, select the example by selecting the check box and then select **Move** or **Delete**.



# Dialog overview

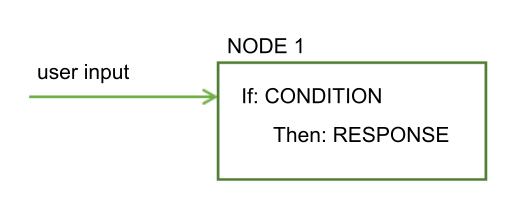
The dialog uses the intents and entities that are identified in the user's input, plus context from the application, to interact with the user and ultimately provide a useful response.

The response might be the answer to a question such as Where can I get some gas? or the execution of a command, such as turning on the radio. The intent and entity might be enough information to identify the correct response, or the dialog might ask the user for more input that is needed to respond correctly. For example, if a user asks, Where can I get some food? you might want to clarify whether they want a restaurant or a grocery store, to dine in or take out, and so on. You can ask for more details in a text response and create one or more child nodes to process the new input.

The dialog is represented graphically in the Watson Assistant tool as a tree. Create a branch to process each intent that you want your conversation to handle. A branch is composed of multiple nodes.

## **Dialog nodes**

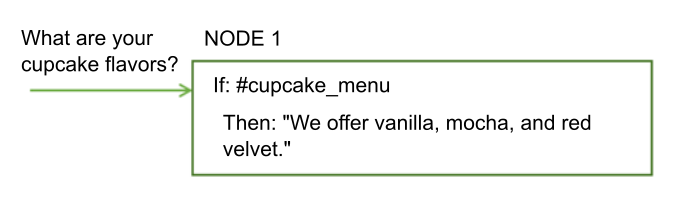
Each dialog node contains, at a minimum, a condition and a response.



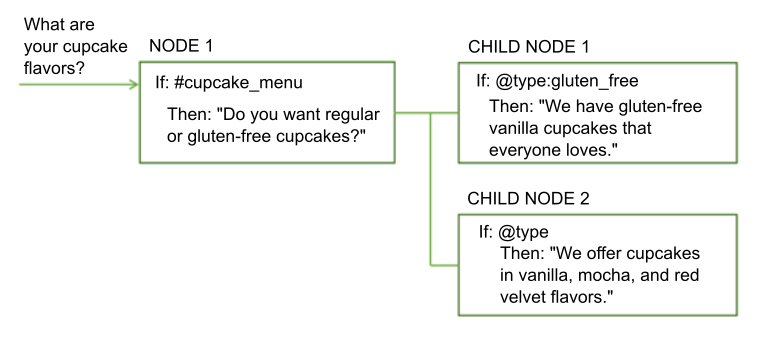
* Condition: Specifies the information that must be present in the user input for this node in the dialog to be triggered. The information might be a specific intent, an entity value, or a context variable value. See [Conditions](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#conditions)for more information.
* Response: The utterance that the service uses to respond to the user. The response can also be configured to trigger programmatic actions. See [Responses](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#responses) for more information.

You can think of the node as having an if/then construction: if this condition is true, then return this response.

For example, the following node is triggered if the natural language processing function of the service determines that the user input contains the #cupcake-menu intent. As a result of the node being triggered, the service responds with an appropriate answer.

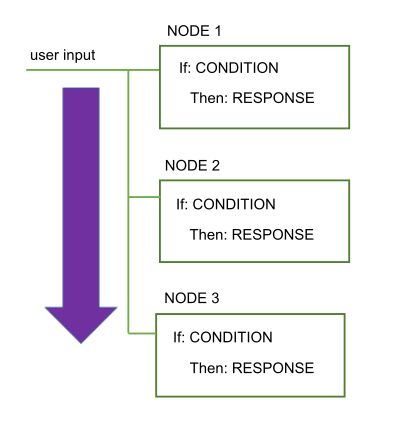


A single node with one condition and response can handle simple user requests. But, more often than not, users have more sophisticated questions or want help with more complex tasks. You can add child nodes that ask the user to provide any additional information that the service needs.



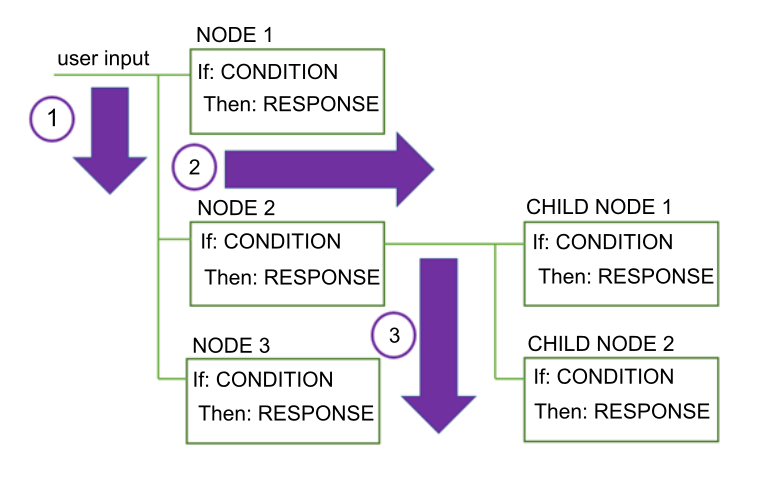
## **Dialog flow**

The dialog that you create is processed by the service from the first node in the tree to the last.



As it travels down the tree, if the service finds a condition that is met, it triggers that node. It then moves along the triggered node to check the user input against any child node conditions. As it checks the child nodes it moves again from the first child node to the last.

The services continues to work its way through the dialog tree from first to last node, along each triggered node, then from first to last child node, and along each triggered child node until it reaches the last node in the branch it is following.



When you start to build the dialog, you must determine the branches to include, and where to place them. The order of the branches is important because nodes are evaluated from first to last. The first root node whose condition matches the input is used; any nodes that come later in the tree are not triggered.

When the service reaches the end of a branch, or cannot find a condition that evaluates to true from the current set of child nodes it is evaluating, it jumps back out to the base of the tree. And once again, the service processes the root nodes from first to the last. If none of the conditions evaluates to true, then the response from the last node in the tree, which typically has a special anything\_else condition that always evaluates to true, is returned.

You can disrupt the standard first-to-last flow by customizing what happens after a node is processed. For example, you can configure a node to jump directly to another node after it is processed, even if the other node is positioned earlier in the tree. See [Defining what to do next](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#jump-to) for more details.

How you configure digression settings for each node can also impact how users move through the nodes at run time. If you enable digressions away from most nodes, users can jump from one node to another and back again more easily. See [Digressions](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#digressions) for more information.

## **Conditions**

A node condition determines whether that node is used in the conversation. Response conditions determine which response to display to a user.

* [Condition artifacts](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#condition-artifacts)
* [Condition syntax details](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#condition-syntax)
* [Condition usage tips](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#condition-tips)
* [Special conditions](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#special-conditions)

### Condition artifacts

You can use one or more of the following artifacts in any combination to define a condition:

* **Context variable**: The node is used if the context variable expression that you specify is true. Use the syntax, $variable\_name:value or $variable\_name == 'value'. For example, $city:Boston checks whether the $city context variable contains the value, Boston. If so, the node or response is processed.

Do not define a node or response condition based on the value of a context variable in the same dialog node in which you set the context variable value.

For more information about context variables, see [Context variables](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context).

* **Entity**: The node is used when any value or synonym for the entity is recognized in the user input. Use the syntax, @entity\_name. For example, @city checks whether any of the city names that are defined for the @city entity were detected in the user input. If so, the node or response is processed.

Be sure to create a peer node to handle the case where none of the entity's values or synonyms are recognized.

For more information about entities, see [Defining entities](https://console.bluemix.net/docs/services/conversation/entities.html).

* **Entity value**: The node is used if the entity value is detected in the user input. Use the syntax, @entity\_name:value and specify a defined value for the entity, not a synonym. For example: @city:Boston checks whether the specific city name, Boston, was detected in the user input.

If the entity is a pattern entity with capture groups, then you can check for a certain group value match. For example, you can use the syntax: @us\_phone.groups[1] == '617' See [Storing pattern entity values in context variables](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context-pattern-entities) for more information.

If you check for the presence of the entity, without specifying a particular value for it, in a peer node, be sure to position this node (which checks for a particular entity value) before the peer node that checks only for the presence of the entity. Otherwise, this node will never be evaluated.

* **Intent**: The simplest condition is a single intent. The node is used if the user's input maps to that intent. Use the syntax, #intent\_name. For example, #weather checks if the intent detected in the user input is weather. If so, the node is processed.

For more information about intents, see [Defining intents](https://console.bluemix.net/docs/services/conversation/intents.html).

* **Special condition**: Conditions that are provided with the service that you can use to perform common dialog functions. See the **Special conditions** table in the next section for details.

### Special conditions

| **Condition syntax** | **Description** |
| --- | --- |
| anything\_else | You can use this condition at the end of a dialog, to be processed when the user input does not match any other dialog nodes. The **Anything else** node is triggered by this condition. |
| conversation\_start | Like **welcome**, this condition is evaluated as true during the first dialog turn. Unlike **welcome**, it is true whether or not the initial request from the application contains user input. A node with the **conversation\_start** condition can be used to initialize context variables or perform other tasks at the beginning of the dialog. |
| false | This condition is always evaluated to false. You might use this at the start of a branch that is under development, to prevent it from being used, or as the condition for a node that provides a common function and is used only as the target of a **Jump to**action. |
| irrelevant | This condition will evaluate to true if the user’s input is determined to be irrelevant by the Watson Assistant service. |
| true | This condition is always evaluated to true. You can use it at the end of a list of nodes or responses to catch any responses that did not match any of the previous conditions. |
| welcome | This condition is evaluated as true during the first dialog turn (when the conversation starts), only if the initial request from the application does not contain any user input. It is evaluated as false in all subsequent dialog turns. The **Welcome** node is triggered by this condition. Typically, a node with this condition is used to greet the user, for example, to display a message such as Welcome to our Pizza ordering app. |
| Special conditions | |

### Condition syntax details

Use one of these syntax options to create valid expressions in conditions:

* Shorthand notations to refer to intents, entities, and context variables. See [Accessing and evaluating objects](https://console.bluemix.net/docs/services/conversation/expression-language.html).
* Spring Expression (SpEL) language, which is an expression language that supports querying and manipulating an object graph at runtime. See [Spring Expression Language (SpEL) language](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html) for more information.

You can use regular expressions to check for values to condition against. To find a matching string, for example, you can use the String.find method. See [Methods](https://console.bluemix.net/docs/services/conversation/dialog-methods.html) for more details.

### Condition usage tips

* **Checking for values with special characters**: If you want to check whether an entity or context variable contains a value, and the value includes a special character, such as an apostrophe ('), then you must surround the value that you want to check with parentheses. For example, to check if an entity or context variable contains the name O'Reilly, you must surround the name with parentheses.

@person:(O'Reilly) and $person:(O'Reilly)

The service converts these shorthand references into these full SpEL expressions:

entities['person']?.contains('O''Reilly') and context['person'] == 'O''Reilly'

**Note**: SpEL uses a second apostrophe to escape the single apostrophe in the name.

* **Checking for multiple values**: If you want to check for more than one value, you can create a condition that uses OR operators (||) to list multiple values in the condition. For example, to define a condition that is true if the context variable $state contains the abbreviations for Massachusetts, Maine, or New Hampshire, you can use this expression:

$state:MA || $state:ME || $state:NH

* **Checking for number values**: When using numeric variables, make sure the variables have values. If a variable does not have a value, it is treated as having a null value (0) in a numeric comparison.

For example, if you check the value of a variable with the condition @price < 100, and the @price entity is null, then the condition is evaluated as true because 0 is less than 100, even though the price was never set. To prevent the checking of null variables, use a condition such as @price AND @price < 100. If @pricehas no value, then this condition correctly returns false.

* **Checking for intents with a specific intent name pattern**: You can use a condition that looks for intents that match a pattern. For example, to find any detected intents with intent names that start with 'User\_', you can use a syntax like this in the condition:

intents[0].intent.startsWith("User\_")

However, when you do so, all of the detected intents are considered, even those with a confidence lower than 0.2. Also check that intents which are considered irrelevant by Watson based on their confidence score are not returned. To do so, change the condition as follows:

!irrelevant && intents[0].intent.startsWith("User\_")

* **How fuzzy matching impacts entity recognition**: If you use an entity as the condition and fuzzy matching is enabled, then @entity\_name evaluates to true only if the confidence of the match is greater than 30%. That is, only if @entity\_name.confidence > .3.
* **Handling multiple entities in input**: If you want to evaluate only the value of the first detected instance of an entity type, you can use the syntax @entity == 'specific-value' instead of the @entity:(specific-value) format.

For example, when you use @appliance == 'air conditioner', you are evaluating only the value of the first detected @appliance entity. But, using @appliance:(air conditioner) gets expanded to entity['appliance'].contains('air conditioner'), which matches whenever there is at least one @appliance entity of value 'air conditioner' detected in the user input.

## **Responses**

The dialog response defines how to reply to the user.

You can reply with one of these response types:

* [Simple text response](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#simple-text)
* [Conditional responses](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#multiple)
* [Complex response](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#complex)

### Simple text response

If you want to provide a text response, simply enter the text that you want the service to display to the user.



To include a context variable value in the response, use the syntax $variable-name to specify it. See [Context variables](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context) for more information.

Hello $user

If you include an email address in the response, you must escape the at symbol (@) with a backslash (\). For example, Send us your feedback at feedback\@example.com. Likewise, if you include a number sign (#) in the response, you must escape it. For example, We are the \#1 seller of lobster rolls in Maine. Entity names begin with @ and intent names begin with #. Escaping these symbols prevents the service from misreading the response text.

#### Adding variety

If your users return to your conversation service frequently, they might be bored to hear the same greetings and responses every time. You can add variations to your responses so that your conversation can respond to the same condition in different ways.

In this example, the answer that the service provides in response to questions about store locations differs from one interaction to the next:



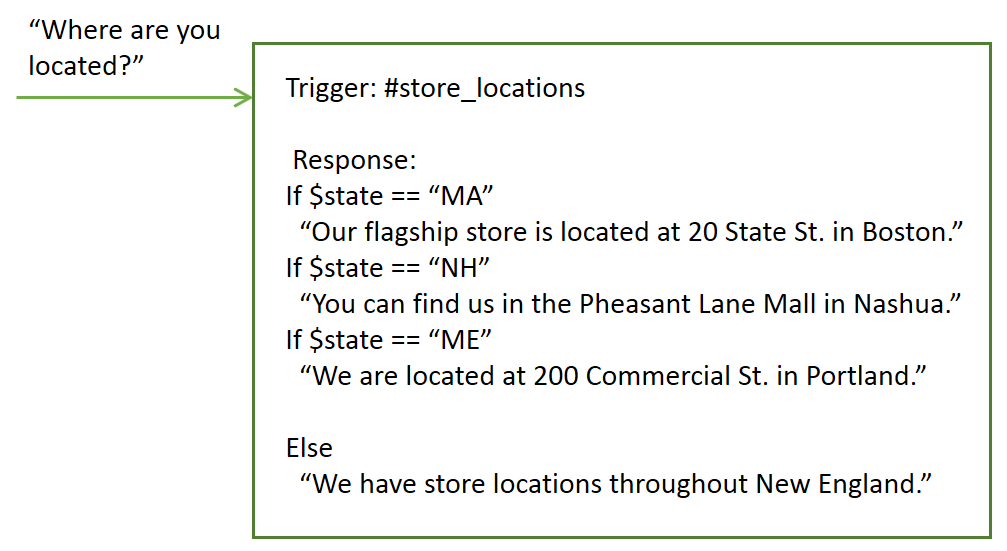
You can choose to rotate through the response variations sequentially or in random order. By default, responses are rotated sequentially, as if they were chosen from an ordered list.

### Conditional responses

A single dialog node can provide different responses, each one triggered by a different condition. Use this approach to address multiple scenarios in a single node.

The node still has a main condition, which is the condition for using the node and processing the conditions and responses that it contains.

In this example, the service uses information that it collected earlier about the user's location to tailor its response, and provide information about the store nearest the user. See [Context variables](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context) for more information about how to store information collected from the user.



This single node now provides the equivalent function of four separate nodes.

To add conditional responses to a node, complete the following steps:

1. Click **Customize**, and then click the **Multiple responses** toggle to turn it **On**.

The node response section changes to show a pair of condition and response fields. You can add a condition and a response into them.

1. To customize a response further, click the **Edit response** Edit response icon next to the response.

You must open the response for editing to change the value of a context variable when the response is triggered, for example. You update or add context values for each individual conditional response; there is no common JSON editor that shows you the context information for all of the responses at once.

1. Click **Add response** to add another conditional response.

The conditions within a node are evaluated in order, just as nodes are. Be sure that your conditional responses are listed in the correct order. If you need to change the order, select a condition and response pair and move it up or down in the list using the arrows that are displayed. A **Jump to** action that is configured for the node is not processed until after all of the conditional responses are processed.

### A complex response

To specify a more complex response, you can use the JSON editor to specify the response in the "output":{}property. The output.text field is always stored as a JSON array. If you try to save it in any other format, your text is replaced with an empty JSON array.

To specify more than one statement that you want to display on separate lines, define the output as a JSON array.

{

"output": {

"text": ["Hello there.", "How are you?"]

}

}

The first sentence is displayed on one line, and the second sentence is displayed as a new line after it.

To implement more complex behavior, you can define the output text as a complex JSON object. For example, you can use a complex object in the JSON output to mimic the behavior of adding response variations to the node. You can include the following properties in the complex object:

* **values**: A JSON array of strings that holds multiple versions of output text that this dialog node can return. The order in which values in the array are returned depends on the attribute selection\_policy.
* **selection\_policy**: The following values are valid:
  + **random**: The system randomly selects output text from the values array and does not repeat them consecutively. For example, consider output.text that contains three values. For the first three times, a random value is selected but not repeated another time. After all the output values are given, the system randomly selects another value and repeats the process.
  + {
  + "output":{
  + "text":{
  + "values":["Hello.","Hi.","Howdy!"],
  + "selection\_policy":"random"
  + }
  + }
  + }

The system returns one greeting from these three options that it picks at random. The next time the response is triggered, another greeting from the list is displayed. The greeting is again chosen at random, except the previously used greeting is intentionally not repeated.

* + **sequential**: The system returns the first output text the first time the dialog node is triggered, the second output text the second time the node is triggered, and so on.
  + {
  + "output":{
  + "text":{
  + "values":["Hello.", "Hi.", "Howdy!"],
  + "selection\_policy":"sequential"
  + }
  + }
  + }
* **append**: Specifies whether to append a value to an array or overwrite the values in the array with the new value or values. When set to false, the output collected in previously executed dialog nodes is overwritten by the text value specified in this particular node.
* {
* "output":{
* "text":{
* "values": ["Hello."],
* "append":false
* }
* }
* }

In this case, all other output text is overwritten by this output text.

The default behavior assumes selection\_policy = random and append = true. When the values array contains more than one item, the output text is randomly selected from its elements.

## **Defining what to do next**

After making the specified response, you can instruct the service to do one of the following things:

* **Wait for user input**: The service waits for the user to provide new input that the response elicits. For example, the response might ask the user a yes or no question. The dialog will not progress until the user provides more input.
* **Skip user input**: Use this option when you want to bypass waiting for user input and go directly to the first child node of the current node instead.

**Note**: The current node must have at least one child node for this option to be available.

* **Jump to another dialog node**: Use this option when you want the conversation to go directly to an entirely different dialog node. You can use a Jump to action to route the flow to a common dialog node from multiple locations in the tree, for example.

**Note**: The target node that you want to jump to must exist before you can configure the jump to action to use it.

### Configuring the Jump to action

If you choose to jump to another node, specify when the target node is processed by choosing one of the following options:

* **Condition**: If the statement targets the condition section of the selected dialog node, the service checks first whether the condition of the targeted node evaluates to true.
  + If the condition evaluates to true, the system processes the target node immediately.
  + If the condition does not evaluate to true, the system moves to the next sibling node of the target node to evaluate its condition, and repeats this process until it finds a dialog node with a condition that evaluates to true.
  + If the system processes all the siblings and none of the conditions evaluate to true, the basic fallback strategy is used, and the dialog evaluates the nodes at the base level of the dialog tree.

Targeting the condition is useful for chaining the conditions of dialog nodes. For example, you might want to first check whether the input contains an intent, such as #turn\_on, and if it does, you might want to check whether the input contains entities, such as @lights, @radio, or @wipers. Chaining conditions helps to structure larger dialog trees.

* **Response**: If the statement targets the response section of the selected dialog node, it is run immediately. That is, the system does not evaluate the condition of the selected dialog node; it processes the response of the selected dialog node immediately.

Targeting the response is useful for chaining several dialog nodes together. The response is processed as if the condition of this dialog node is true. If the selected dialog node has another **Jump to** action, that action is run immediately, too.

* **Wait for user input**: Waits for new input from the user, and then begins to process it from the node that you jump to. This option is useful if the source node asks a question, for example, and you want to jump to a separate node to process the user's answer to the question.

## **More information**

For information about the expression language used by dialog, plus methods, system entities, and other useful details, see the **Reference** section in the navigation pane.

# How the dialog is processed

Understand how your dialog is processed when a person interacts with your instance of the deployed Watson Assistant service at run time.

## **Anatomy of a dialog call**

Each user utterance is passed to the dialog as a /message API call. This includes utterances that users make in reply to prompts from the dialog that ask them for more information. Some subscription plans include a set number of API calls, so it helps to understand what constitutes a call. A single /message API call is equivalent to a single dialog turn, which consists of an input from the user and a corresponding response from the dialog.

The body of the /message API call request and response includes the following objects:

* context: Contains variables that are meant to be persisted. To pass information from one call to the next, the application developer must pass the previous API call's response context in with each subsequent API call. For example, the dialog can collect the user's name and then refer to the user by name in subsequent nodes.
* {
* "context" : {
* "user\_name" : "<? @sys-person.literal ?>"
* }

See [Retaining information across dialog turns](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context) for more information.

* input: The string of text that was submitted by the user. The text string can contain up to 2,048 characters.
* {
* "input" : {
* "text" : "Where's your nearest store?"
* }
* output: The dialog response to display to the user. You can use this section to define objects, such as variables, that are not meant to be persisted. For example, if you want to permanently delete a context variable named temp that you defined elsewhere in the dialog, you can use the following expression to do so.
* {
* "output": {
* "text" : {},
* "deleted\_variable" : "<? context.remove('temp') ?>"

See [A complex response](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#complex) for more information about the output object.

You can learn more about the /message API call from the [API reference](https://www.ibm.com/watson/developercloud/conversation/api/v1/).

## **Retaining information across dialog turns**

The dialog is stateless, meaning that it does not retain information from one interaction with the user to the next. It is the responsibility of the application developer to maintain any continuing information that the application needs. The application must look for, and store the context object in the message API response, and pass it in the context object with the next /message API request that is made as part of the conversation flow.

The simplest way to retain the information is to store the entire context object in memory in the client application - a web browser, for example. As an application becomes more complex, or if it needs to pass and store personally identifiable information, then you can store and retrieve the information from a database.

The application can pass information to the dialog, and the dialog can update this information and pass it back to the application, or to a subsequent node. The dialog does so by using context variables.

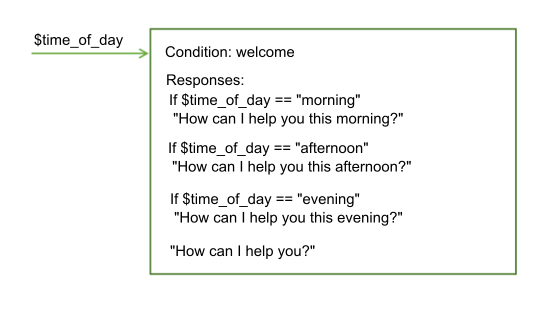
A context variable is a variable that you define in a node, and optionally specify a default value for. Other nodes or application logic can subsequently set or change the value of the context variable.

You can condition against context variable values by referencing a context variable from a dialog node condition to determine whether to execute a node. And you can reference a context variable from dialog node response conditions to show different reponses depending on a value provided by an external service or by the user.

### Passing context from the application

Pass information from the application to the dialog by setting a context variable and passing the context variable to the dialog.

For example, your application can set a $time\_of\_day context variable, and pass it to the dialog which can use the information to tailor the greeting it displays to the user.

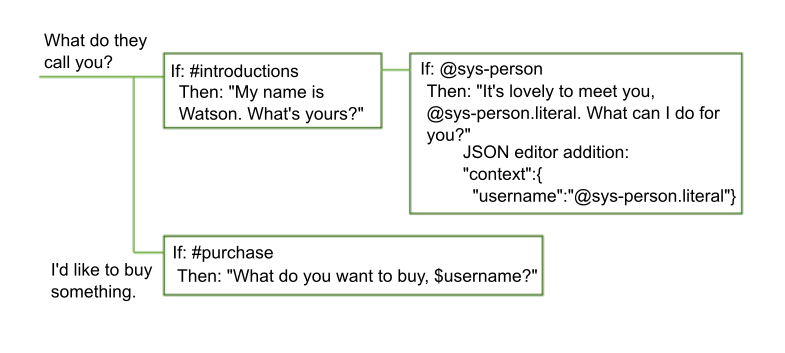


In this example, the dialog knows that the application sets the variable to one of these values: morning, afternoon, or evening. It can check for each value, and depending on which value is present, return the appropriate greeting. If the variable is not passed or has a value that does not match one of the expected values, then a more generic greeting is displayed to the user.

### Passing context from node to node

The dialog can also add context variables to pass information from one node to another or to update the values of context variables. As the dialog asks for and gets information from the user, it can keep track of the information and reference it later in the conversation.

For example, in one node you might ask users for their name, and in a later node address them by name.



In this example, the system entity @sys-person is used to extract the user's name from the input if the user provides one. In the JSON editor, the username context variable is defined and set to the @sys-person value. In a subsequent node, the $username context variable is included in the response to address the user by name.

## **Defining a context variable**

Define a context variable by defining a name and value pair for the variable in one of the following editors:

* **Context editor**: Shows a **Variable** field and a corresponding **Value** field in the node edit view that you can fill with the context variable name and value information.

**Note**: These fields are displayed automatically in nodes that you add. For nodes that were created with an earlier version of the service, you must open the context editor for the fields to be added.

* **JSON editor**: When opened, it provides a view into the underlying JSON content that is passed with the /message API request that is sent to the Watson Assistant service. You can define context variables by adding name and value pairs to the "context":{} section of the JSON body.

The name and value pair must meet these requirements:

* The name can contain any upper- and lowercase alphabetic characters, numeric characters (0-9), and underscores.

**Note**: You can include other characters, such as periods and hyphens, in the name. However, if you do, then you must use one of the following approaches every time you subsequently reference the variable:

* + **context['variable-name']**

The full SpEL expression syntax.

* + **$(variable-name)**

Shorthand syntax with the variable name enclosed in parentheses. See [Accessing and evaluating objects](https://console.bluemix.net/docs/services/conversation/expression-language.html#shorthand-syntax-for-context-variables)for more details.

* The value can be any supported JSON type, such as a simple string variable, a number, or a JSON array. When you define the context variable using the JSON editor, you can specify a JSON object as the value also.

The following table shows how to define name and value pairs in context variable editor fields:

| **Variable** | **Value** |
| --- | --- |
| dessert | cake |
| toppings\_array | ["onion","olives"] |
| age | 18 |

The following JSON sample defines values for the $dessert string, $toppings\_array array, and $age number context variables:

{

"context": {

"dessert": "cake",

"toppings\_array": ["onion", "olives"],

"age": 18

}

}

To define a context variable, complete the following steps:

1. Define the context variable in the section of the node that represents the time at which you want the variable to be set during dialog node evaluation.

**Note**: Any existing context variable values that are defined for this node are displayed in a set of corresponding **Variable** and **Value** fields. If you do not want them to be displayed in the edit view of the node, you must close the context editor. You can close the editor from the same menu that is used to open it; the following steps describe how to access the menu.

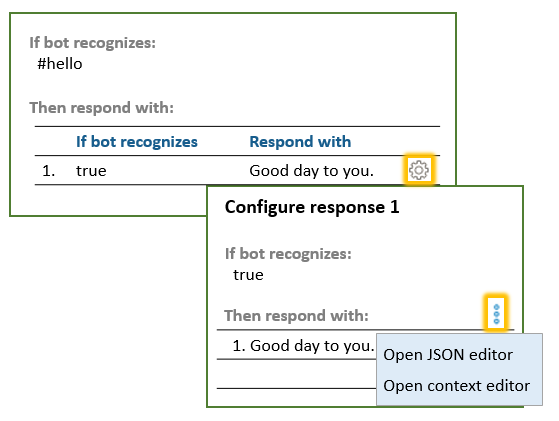
* + To add a context variable that is set or changed after the node response is processed, add the context variable into the response section.

Click the **Options** Advanced response icon that is associated with the response, and then choose an editor by selecting one of the following options:

* + - **Open JSON editor**
    - **Open context editor**

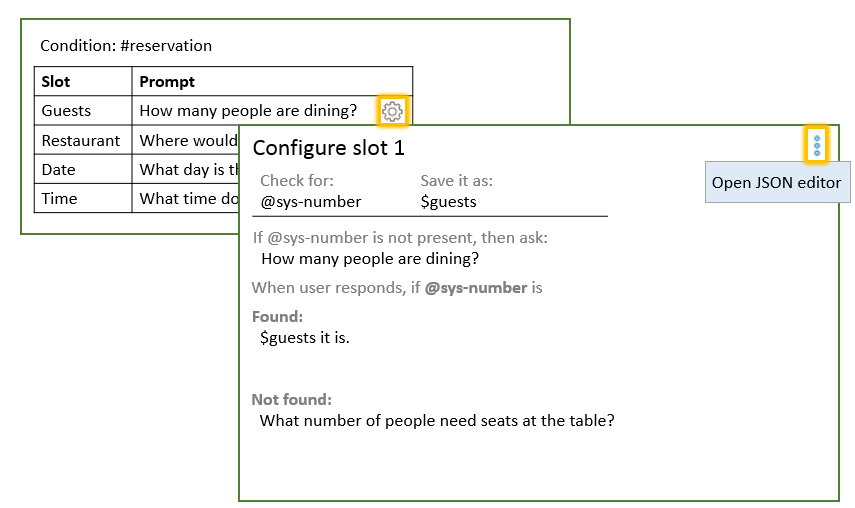


If the **Multiple responses** setting is **On** for the node, then you must click the **Edit response** Edit response icon first.

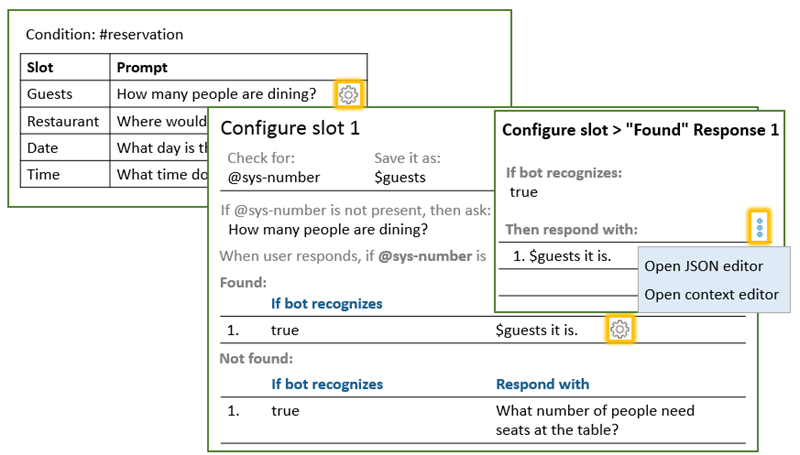


* + To add a context variable that is set or updated after a slot condition is met, click the **Edit slot** Edit response icon. From the **Options** Advanced response menu in the Configure slot view header, click **Open JSON editor**. (For more information about slots, see [Gathering information with slots](https://console.bluemix.net/docs/services/conversation/dialog-slots.html).)

**Note**: There is currently no way to use the context editor to define context variables that are set during this phase of dialog node evaluation.



* + To add a context variable that is processed after a response condition for a slot is met, click the **Edit slot** Edit response icon. Click the **Options** Advanced response icon, and then select **Enable conditional responses**. Click the **Edit response** Edit response icon next to the response with which you want to associate the context variable. Click the **Options** Advanced responseicon in the response section, and then choose an editor by selecting one of the following options:
    - **Open JSON editor**
    - **Open context editor**



1. To define the context variable in the context editor, add the variable name and value pair to the **Variable** and **Value** fields.
2. To define the context variable in the JSON editor, complete these additional steps:
   * Add a "context":{} block if one is not present.
   * {
   * "context":{},
   * "output":{}
   * }
   * In the context block, add a name and value pair for each context variable that you want to define.
   * {
   * "context":{
   * "name": "value"
   * },
   * "output": {}
   * }

In this example, a variable named new\_variable is added to a context block that already contains a variable.

{

"context":{

"existing\_variable": "value",

"new\_variable":"value"

}

}

To subsequently reference the context variable, use the syntax $name where name is the name of the context variable that you defined. For example, $new\_variable.

## **Common context variable tasks**

To store the entire string that was provided by the user as input, use input.text:

| **Variable** | **Value** |
| --- | --- |
| repeat | <?input.text?> |

{

"context": {

"repeat": "<?input.text?>"

}

}

To store the value of an entity in a context variable, use this syntax:

| **Variable** | **Value** |
| --- | --- |
| place | @place |

{

"context": {

"place": "@place"

}

}

You can add a JSON object to a context variable using either editor. The following expression defines a full\_name object that contains a set of first and last values, which together form a person's full name.

| **Variable** | **Value** |
| --- | --- |
| full\_name | { "first":"Paul", "last":"Smith" } |

{

"context": {

"full\_name": {

"first":"Paul",

"last":"Smith"

}

}

}

If you specify $full\_name.first in the response, Paul is displayed.

To store the value of a string that you extract from the user's input, you can include a SpEL expression that uses the extract method to apply a regular expression to the user input. The following expression extracts a number from the user input, and saves it to the $number context variable.

| **Variable** | **Value** |
| --- | --- |
| number | <?input.text.extract('[\d]+',0)?> |

{

"context": {

"number": "<?input.text.extract('[\\d]+',0)?>"

}

}

When you define a regular expression in the JSON editor, you must escape any back slashes that you use in the expression with another back slash (\\). You do not need to escape back slashes in regular expressions that you define using the context variable editor.

To store the value of a pattern entity, append .literal to the entity name. Using this syntax ensures that the exact span of text from user input that matched the specified pattern is stored in the variable.

| **Variable** | **Value** |
| --- | --- |
| email | @email.literal |

{

"context": {

"email": "<? @email.literal ?>"

}

}

## **Deleting a context variable**

To delete a context variable, set the variable to null.

{

"context": {

"order\_form": null

}

}

If you want to remove all trace of the context variable, you can use the JSONObject.remove(string) method to delete it from the context object. However, you must use a variable to perform the removal. Define the new variable in the message output so it will not be saved beyond the current call.

{

"output": {

"text" : {},

"deleted\_variable" : "<? context.remove('order\_form') ?>"

}

}

Alternatively you can delete the context variable in your application logic.

### Order of operation

The order in which you define the context variables does not determine the order in which they are evaluated by the service. The service evaluates the variables, which are defined as JSON name and value pairs, in random order. Do not set a value in the first context variable and expect to be able to use it in the second, because there is no guarantee that the first context variable in your list will be executed before the second one in your list. For example, do not use two context variables to implement logic that returns a random number between zero and some higher value that is passed to the node.

"context": {

"upper": "<? @sys-number.numeric\_value + 1?>",

"answer": "<? new Random().nextInt($upper) ?>"

}

Use a slightly more complex expression to avoid having to rely on the value of the $upper context variable being evaluated before the $answer context variable is evaluated.

"context": {

"answer": "<? new Random().nextInt(@sys-number.numeric\_value + 1) ?>"

}

### Storing pattern entity values

To store the value of a pattern entity in a context variable, append .literal to the entity name. Using this syntax ensures that the exact span of text from user input that matched the specified pattern is stored in the variable.

{

"context": {

"email": "<? @email.literal ?>"

}

}

To store the text from a single group in a pattern entity with groups defined, specify the array number of the group that you want to store. For example, assume that the entity pattern is defined as follows for the @phone\_number entity. (Remember, the parentheses denote pattern groups):

\b((958)|(555))-(\d{3})-(\d{4})\b

To store only the area code from the phone number that is specified in user input, you can use the following syntax:

{

"context": {

"area\_code": "<? @phone\_number.groups[1] ?>"

}

}

The groups are delimited by the regular expression that is used to define the group pattern. For example, if the user input that matches the pattern defined in the entity @phone\_number is: 958-234-3456, then the following groups are created:

| **Group number** | **Regex engine value** | **Dialog value** | **Explanation** |
| --- | --- | --- | --- |
| groups[0] | 958-234-3456 | 958-234-3456 | The first group is always the full matching string. |
| groups[1] | ((958)l(555)) | 958 | String that matches the regex for the first defined group, which is ((958)l(555)). |
| groups[2] | (958) | 958 | Match against the group that is included as the first operand in the OR expression ((958)l(555)) |
| groups[3] | (555) | null | No match against the group that is included as the second operand in the OR expression ((958)l(555)) |
| groups[4] | (\d{3}) | 234 | String that matches the regular expression that is defined for the group. |
| groups[5] | (\d{4}) | 3456 | String that matches the regular expression that is defined for the group. |
| Group details | | | |

To help you decipher which group number to use to capture the section of input you are interested in, you can extract information about all the groups at once. Use the following syntax to create a context variable that returns an array of all the grouped pattern entity matches:

{

"context": {

"array\_of\_matched\_groups": "<? @phone\_number.groups ?>"

}

}

Use the "Try it out" pane to enter some test phone number values. For the input 958-123-2345, this expression sets $array\_of\_matched\_groups to ["958-123-2345","958","958",null,"123","2345"].

You can then count each value in the array starting with 0 to get the group number for it.

| **Array element value** | **Array element number** |
| --- | --- |
| "958-123-2345" | 0 |
| "958" | 1 |
| "958" | 2 |
| null | 3 |
| "123" | 4 |
| "2345" | 5 |
| Array elements | |

It is easy to determine that, to capture the last four digits of the phone number, you need group #5, for example.

To return the JSONArray structure that is created to represent the grouped pattern entity, use the following syntax:

{

"context": {

"json\_matched\_groups": "<? @phone\_number.groups\_json ?>"

}

}

This expression sets $json\_matched\_groups to the following JSON array:

[

{"group": "group\_0","location": [0, 12]},

{"group": "group\_1","location": [0, 3]},

{"group": "group\_2","location": [0, 3]},

{"group": "group\_3"},

{"group": "group\_4","location": [4, 7]},

{"group": "group\_5","location": [8, 12]}

]

**Note**: location is a property of an entity that uses a zero-based character offset to indicate where the detected entity value begins and ends in the input text.

If you expect two phone numbers to be supplied in the input, then you can check for two phone numbers. If present, use the following syntax to capture the area code of the second number, for example.

{

"context": {

"second\_areacode": "<? entities['phone\_number'][1].groups[1] ?>"

}

}

If the input is I want to change my phone number from 958-234-3456 to 555-456-5678, then $second\_areacode equals 555.

## **Updating a context variable value**

If a node sets the value of a context variable that is already set, then the previous value is overwritten.

### Updating a complex JSON object

Previous values are overwritten for all JSON types except a JSON object. If the context variable is a complex type such as JSON object, a JSON merge procedure is used to update the variable. The merge operation adds any newly defined properties and overwrites any existing properties of the object.

In this example, a name context variable is defined as a complex object.

{

"context": {

"complex\_object": {

"user\_firstname" : "Paul",

"user\_lastname" : "Pan",

"has\_card" : false

}

}

}

A dialog node updates the context variable JSON object with the following values:

{

"complex\_object": {

"user\_firstname": "Peter",

"has\_card": true

}

}

The result is this context:

{

"complex\_object": {

"user\_firstname": "Peter",

"user\_lastname": "Pan",

"has\_card": true

}

}

See [Expression language methods](https://console.bluemix.net/docs/services/conversation/dialog-methods.html#objects) for more information about methods you can perform on objects.

### Updating arrays

If your dialog context data contains an array of values, you can update the array by appending values, removing a value, or replacing all the values.

Choose one of these actions to update the array. In each case, we see the array before the action, the action, and the array after the action has been applied.

* **Append**: To add values to the end of an array, use the append method.

For this Dialog runtime context:

{

"context": {

"toppings\_array": ["onion", "olives"]

}

}

Make this update:

{

"context": {

"toppings\_array": "<? $toppings\_array.append('ketchup', 'tomatoes') ?>"

}

}

Result:

{

"context": {

"toppings\_array": ["onion", "olives", "ketchup", "tomatoes"]

}

}

* **Remove**: To remove an element, use the remove method and specify its value or position in the array.
  + **Remove by value** removes an element from an array by its value.

For this Dialog runtime context:

{

"context": {

"toppings\_array": ["onion", "olives"]

}

}

Make this update:

{

"context": {

"toppings\_array": "<? $toppings\_array.removeValue('onion') ?>"

}

}

Result:

{

"context": {

"toppings\_array": ["olives"]

}

}

* + **Remove by position**: Removing an element from an array by its index position:

For this Dialog runtime context:

{

"context": {

"toppings\_array": ["onion", "olives"]

}

}

Make this update:

{

"context": {

"toppings\_array": "<? $toppings\_array.remove(0) ?>"

}

}

Result:

{

"context": {

"toppings\_array": ["olives"]

}

}

* **Overwrite**: To overwrite the values in an array, simply set the array to the new values:

For this Dialog runtime context:

```json

{

"context": {

"toppings\_array": ["onion", "olives"]

}

}

```

Make this update:

```json

{

"context": {

"toppings\_array": ["ketchup", "tomatoes"]

}

}

```

Result:

```json

{

"context": {

"toppings\_array": ["ketchup", "tomatoes"]

}

}

```

See [Expression language methods](https://console.bluemix.net/docs/services/conversation/dialog-methods.html#arrays) for more information about methods you can perform on arrays.

## **Digressions**

A digression occurs when a user is in the middle of a dialog flow that is designed to address one goal, and abruptly switches topics to initiate a dialog flow that is designed to address a different goal. The dialog has always supported the user's ability to change subjects. If none of the nodes in the dialog branch that is being processed match the goal of the user's latest input, the conversation goes back out to the tree to check the root node conditions for an appropriate match. The digression settings that are available per node give you the ability to tailor this behavior even more.

With digression settings, you can allow the conversation to return to the dialog flow that was interrupted when the digression occurred. For example, the user might be ordering a new phone, but switches topics to ask about tablets. Your dialog can answer the question about tablets, and then bring the user back to where they left off in the process of ordering a phone. Allowing digressions to occur and return gives your users more control over the flow of the conversation at run time. They can change topics, follow a dialog flow about the unrelated topic to its end, and then return to where they were before. The result is a dialog flow that more closely simulates a human-to-human conversation.

The animated image uses a mockup of the dialog tree user interface to illustrate the concept of a digression. It shows how a user interacts with dialog nodes that are configured to allow digressions that return to the dialog flow that was in progress. The user starts to provide the information required to make a dinner reservation. In the middle of filling slots in the #reservation node, the user asks a question about vegetarian menu options. The dialog answers the user's new question by finding a node that addresses it amongst the root nodes (a node that conditions on the #cuisine intent). It then returns to the conversation that was in progress by showing the prompt for the next empty slot from the original dialog node.

Watch this video to learn more.

* [Before you begin](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#prereqs)
* [Customizing digressions](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#enable-digressions)
* [Disabling digressions into a root node](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#disable-digressions)
* [Digression tutorial](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#digression-tutorial)
* [Design considerations](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#digression-design-considerations)

### Before you begin

As you test your overall dialog, decide when and where it makes sense to allow digressions and returns from digressions to occur. The following digression controls are applied to the nodes automatically. Only take action if you want to change this default behavior.

* Every root node in your dialog is configured to allow digressions to target them by default. Child nodes cannot be the target of a digression.
* Nodes with slots are configured to prevent digressions away. All other nodes are configured to allow digressions away. However, the conversation cannot digress away from a node under the following circumstances:
  + If any of the child nodes of the current node contain the anything\_else or true condition

These conditions are special in that they always evaluate to true. Because of their known behavior, they are often used in dialogs to force a parent node to evaluate a specific child node in succession. To prevent breaking existing dialog flow logic, digression are not allowed in this case. Before you can enable digressions away from such a node, you must change the child node's condition to something else.

* + If the node is configured to jump to another node or skip user input after it is processed

The final step section of a node specifies what should happen after the node is processed. When the dialog is configured to jump directly to another node, it is often to ensure that a specific sequence is followed. And when the node is configured to skip user input, it is equivalent to forcing the dialog to process the first child node after the current node in succession. To prevent breaking existing dialog flow logic, digressions are not allowed in either of these cases. Before you can enable digressions away from this node, you must change what is specified in the final step section.

### Customizing digressions

You do not define the start and end of a digression. The user is entirely in control of the digression flow at run time. You only specify how each node should or should not participate in a user-led digression. For each node, you configure whether:

* a digression can start from and leave the node
* a digression that starts elsewhere can target and enter the node
* a digression that starts elsewhere and enters the node must return to the interrupted dialog flow after the current dialog flow is completed

To change the digression behavior for an individual node, complete the following steps:

1. Click the node to open its edit view.
2. Click **Customize**, and then click the **Digressions** tab.

The configuration options differ depending on whether the node you are editing is a root node, a child node, a node with children, or a node with slots.

**Digressions away from this node**

If the circumstances listed earlier do not apply, then you can make the following choices:

* + **All node types**: Choose whether to allow users to digress away from the current node before they reach the end of the current dialog branch.
  + **All nodes that have children**: Choose whether you want the conversation to come back to the current node after a digression if the current node's response has already been displayed and its child nodes are incidental to the node's goal. Set the Allow return from digressions triggered after this node's responsetoggle to **No** to prevent the dialog from returning to the current node and continuing to process its branch.

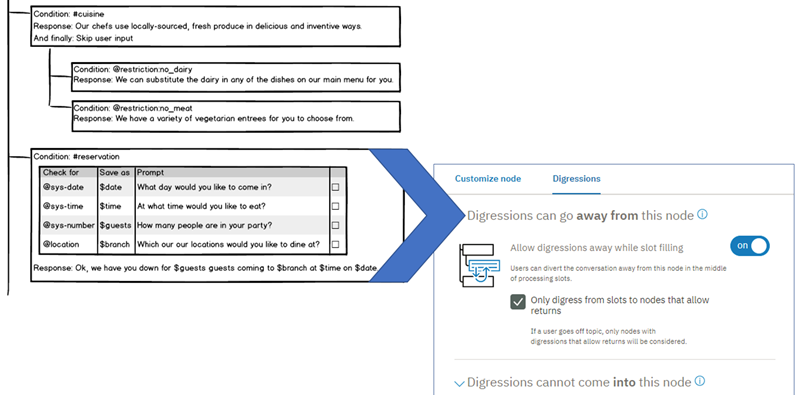
For example, if the user asks, Do you sell cupcakes? and the response, We offer cupcakes in a variety of flavors and sizes is displayed before the user changes subjects, you might not want the dialog to return to where it left off. Especially, if the child nodes only address possible follow-up questions from the user and can safely be ignored.

However, if the node relies on its child nodes to address the question, then you might want to force the conversation to return and continue processing the nodes in the current branch. For example, the initial response might be, We offer cupcakes in all shapes and sizes. Which menu do you want to see: gluten-free, dairy-free, or regular? If the user changes subjects at this point, you might want the dialog to return so the user can pick a menu type and get the information they wanted.

* + **Nodes with slots**: Choose whether you want to allow users to digress away from the node before all of the slots are filled. Set the Allow digressions away while slot filling toggle to **Yes** to enable digressions away.

If enabled, when the conversation returns from the digression, the prompt for the next unfilled slot is displayed to encourage the user to continue providing information. If disabled, then any inputs that the user submits which do not contain a value that can fill a slot are ignored. However, you can address unsolicited questions that you anticipate your users might ask while they interact with the node by defining slot handlers. See [Adding slots](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#add-slots) for more information.

The following image shows you how digressions away from the #reservation node with slots (shown in the earlier illustration) are configured.



* + **Nodes with slots**: Choose whether the user is only allowed to digress away if they will return to the current node by selecting the **Only digress from slots to nodes that allow returns** checkbox.

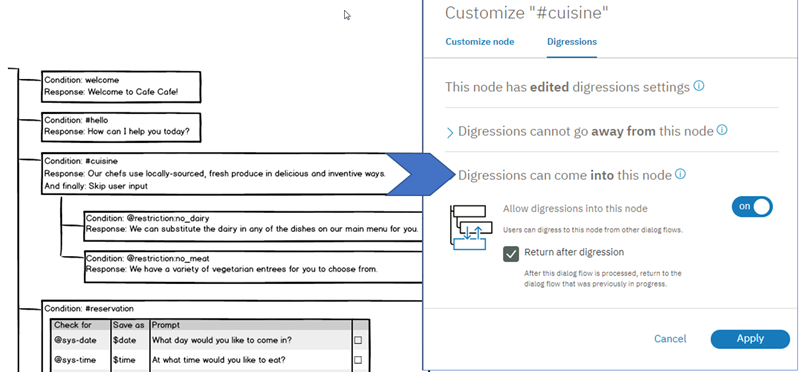
When selected, as the dialog looks for a node to answer the user's unrelated question, it ignores any root nodes that are not configured to return after the digression. Select this checkbox if you want to prevent users from being able to permanently leave the node before they have finished filling the required slots.

**Digressions into this node**

You can make the following choices about how digressions into a node behave:

* + Prevent users from being able to digress into the node. See [Disabling digressions into a root node](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#diable-digressions) for more details.
  + When digressions into the node are enabled, choose whether the dialog must go back to the dialog flow that it digressed away from. When selected, after the current node's branch is done being processed, the dialog flow goes back to the interrupted node. To make the dialog return afterwards, select **Return after digression**.

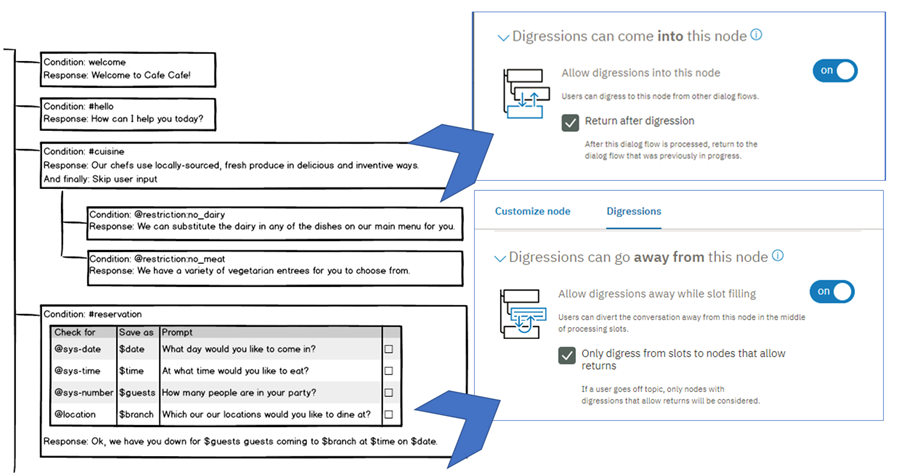
The following image shows you how digressions into the #cuisine node (shown in the earlier illustration) are configured.



1. Click **Apply**.
2. Use the "Try it out" pane to test the digression behavior.

Again, you cannot define the start and end of a digression. The user controls where and when digressions happen. You can only apply settings that determine how a single node participates in one. Because digressions are so unpredictable, it is hard to know how your configuration decisions will impact the overall conversation. To truly see the impact of the choices you made, you must test the dialog.

The #reservation and #cuisine nodes represent two dialog branches that can participate in a single user-directed digression. The digression settings that are configured for each individual node are what make this type of digression possible at run time.



### Disabling digressions into a root node

When a flow digresses into a root node, it follows the course of the dialog that is configured for that node. So, it might process a series of child nodes before it reaches the end of the node branch, and then, if configured to do so, goes back to the dialog flow that was interrupted. Through dialog testing, you might find that a root node is triggered too often, or at unexpected times, or that its dialog is too complex and leads the user too far off course to be a good candidate for a temporary digression. If you determine that you would rather not allow users to digress into it, you can configure the root node to not allow digressions in.

To disable digressions into a root node altogether, complete the following steps:

1. Click to open the root node that you want to edit.
2. Click **Customize**, and then click the **Digressions** tab.
3. Set the Allow digressions into this node toggle to **Off**.
4. Click **Apply**.

If you decide that you want to prevent digressions into several root nodes, but do not want to edit each one individually, you can add the nodes to a folder. From the Customize page of the folder, you can set the Allow digressions into this node toggle to **Off** to apply the configuration to all of the nodes at once. See [Organizing the dialog with folders](https://console.bluemix.net/docs/services/conversation/dialog-build.html#folders) for more information.

### Digression tutorial

Follow the [tutorial](https://console.bluemix.net/docs/services/conversation/tutorial-digressions.html) to import a workspace that has a set of nodes already defined. You can walk through some exercises that illustrate how digressions work.

### Design considerations

* **Avoid fallback node proliferation**: Many dialog designers include a node with a true or anything\_elsecondition at the end of every dialog branch as a way to prevent users from getting stuck in the branch. This design returns a generic message if the user input does not match anything that you anticipated and included a specific dialog node to address. However, users cannot digress away from dialog flows that use this approach.

Evaluate any branches that use this approach to determine whether it would be better to allow digressions away from the branch. If the user's input does not match anything you anticipated, it might find a match against an entirely different dialog flow in your tree. Rather than responding with a generic message, you can effectively put the rest of the dialog to work to try to address the user's input. And the root-level Anything else node can always respond to input that none of the other root nodes can address.

* **Reconsider jumps to a closing node**: Many dialogs are designed to ask a standard closing question, such as, Did I answer your question today? Users cannot digress away from nodes that are configured to jump to another node. So, if you configure all of your final branch nodes to jump to a common closing node, digressions cannot occur. Consider tracking user satisfaction through metrics or some other means.
* **Test possible digression chains**: If a user digresses away from the current node to another node that allows digressions away, the user could potentially digress away from that other node, and repeat this pattern one or more times again. If the starting node in the digression chain is configured to return after the digression, then the user will eventually be brought back to the current dialog node. In fact, any subsequent nodes in the chain that are configured not to return are excluded from being considered as digression targets. Test scenarios that digress multiple times to determine whether individual nodes function as expected.
* **Remember that the current node gets priority**: Remember that nodes outside the current flow are only considered as digression targets if the current flow cannot address the user input. It is even more important in a node with slots that allows digressions away, in particular, to make it clear to users what information is needed from them, and to add confirmation statements that are displayed after the user provides a value.

Any slot can be filled during the slot-filling process. So, a slot might capture user input unexpectedly. For example, you might have a node with slots that collects the information necessary to make a dinner reservation. One of the slots collects date information. While providing the reservation details, the user might ask, What's the weather meant to be tomorrow? You might have a root node that conditions on #forecast which could answer the user. However, because the user's input includes the word tomorrow and the reservation node with slots is being processed, the service assumes the user is providing or updating the reservation date instead. The current node always gets priority. If you define a clear confirmation statement, such as, Ok, setting the reservation date to tomorrow, the user is more likely to realize there was a miscommunication and correct it.

Conversely, while filling slots, if the user provides a value that is not expected by any of the slots, there is a chance it will match against a completely unrelated root node that the user never intended to digress to.

Be sure to do lots of testing as you configure the digression behavior.

* **When to use digressions instead of slot handlers**: For general questions that users might ask at any time, use a root node that allows digressions into it, processes the input, and then goes back to the flow that was in progress. For nodes with slots, try to anticipate the types of related questions users might want to ask while filling in the slots, and address them by adding handlers to the node.

For example, if the node with slots collects the information required to fill out an insurance claim, then you might want to add handlers that address common questions about insurance. However, for questions about how to get help, or your stores locations, or the history of your company, use a root level node.

# Creating a dialog

Use the Watson Assistant tool to create your dialog.

## **Dialog node limits**

The number of dialog nodes you can create depends on your service plan.

| **Service plan** | **Dialog nodes per workspace** |
| --- | --- |
| Standard/Premium | 100,000 |
| Lite | 25,000 |
| Service plan details | |

Tree depth limit: Service supports 2,000 dialog node descendants; tooling performs best with 20 or fewer.

## **Procedure**

To create a dialog, complete the following steps:

1. Open the **Build** page from the navigation bar, click the **Dialog** tab, and then click **Create**.

When you open the dialog builder for the first time, the following nodes are created for you:

* + **Welcome**: The first node. It contains a greeting that is displayed to your users when they first engage with the service. You can edit the greeting.
  + **Anything else**: The final node. It contains phrases that are used to reply to users when their input is not recognized. You can replace the responses that are provided or add more responses with a similar meaning to add variety to the conversation. You can also choose whether you want the service to return each response that is defined in turn or return them in random order.

1. To add more nodes to the dialog tree, click the **More** More icon icon on the **Welcome** node, and then select **Add node below**.
2. Enter a condition that, when met, triggers the service to process the node.

As you begin to define a condition, a box is displayed that shows you your options. You can enter one of the following characters, and then pick a value from the list of options that is displayed.

|  |  |
| --- | --- |
| **Character** | **Lists defined values for these artifact types** |
| # | intents |
| @ | entities |
| @{entity-name}: | {entity-name} values |
| $ | context-variables that you defined or referenced elsewhere in the dialog |
| Condition builder syntax | |

You can create a new intent, entity, entity value, or context variable by defining a new condition that uses it. If you create an artifact this way, be sure to go back and complete any other steps that are necessary for the artifact to be created completely, such as defining sample utterances for an intent.

To define a node that triggers based on more than one condition, enter one condition, and then click the plus sign (+) icon next to it. If you want to apply an OR operator to the multiple conditions instead of AND, click the andthat is displayed between the fields to change the operator type. AND operations are executed before OR operations, but you can change the order by using parentheses. For example: $isMember:true AND ($memberlevel:silver OR $memberlevel:gold)

The condition you define must be less than 500 characters in length.

For more information about how to test for values in conditions, see [Conditions](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#conditions).

1. **Optional**: If you want to collect multiple pieces of information from the user in this node, then click **Customize**and enable **Slots**. See [Gathering information with slots](https://console.bluemix.net/docs/services/conversation/dialog-slots.html) for more details.
2. Enter a response.
   * Add the text that you want the service to display to the user as a response.
   * If you want to define different responses based on certain conditions, then click **Customize** and enable **Multiple responses**.
   * For information about conditional responses or how to add variety to responses, see [Responses](https://console.bluemix.net/docs/services/conversation/dialog-overview.html##responses).
3. Specify what to do after the current node is processed. You can choose from the following options:
   * **Wait for user input**: The service pauses until new input is provided by the user.
   * **Skip user input**: The service jumps directly to the first child node. This option is only available if the current node has at least one child node.
   * **Jump to**: The service continues the dialog by processing the node you specify. You can choose whether the service should evaluate the target node's condition or skip directly to the target node's response. See [Configuring the Jump to action](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#jump-to-config) for more details.
4. **Optional**: Name the node.

The dialog node name can contain letters (in Unicode), numbers, spaces, underscores, hyphens, and periods.

Naming the node makes it easier for you to remember its purpose and to locate the node when it is minimized. If you don't provide a name, the node condition is used as the name.

1. To add more nodes, select a node in the tree, and then click the **More** More icon icon.
   * To create a peer node that is checked next if the condition for the existing node is not met, select **Add node below**.
   * To create a peer node that is checked before the condition for the existing node is checked, select **Add node above**.
   * To create a child node to the selected node, select **Add child node**. A child node is processed after its parent node.
   * To copy the current node, select **Duplicate**.

For more information about the order in which dialog nodes are processed, see [Dialog overview](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#dialog-flow).

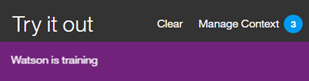
1. Test the dialog as you build it. See [Testing your dialog](https://console.bluemix.net/docs/services/conversation/dialog-build.html#test) for more information.

## **Testing your dialog**

As you make changes to your dialog, you can test it at any time to see how it responds to input.

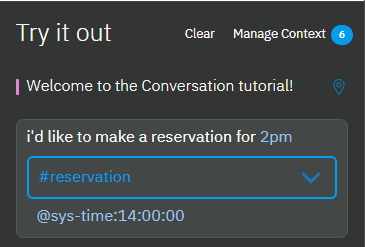
1. From the Dialog tab, click the Ask Watson icon.
2. In the chat pane, type some text and then press Enter.

Make sure the system has finished training on your most recent changes before you start to test the dialog. If the system is still training, a message is displayed in the Try it out pane:



1. Check the response to see if the dialog correctly interpreted your input and chose the appropriate response.

The chat window indicates what intents and entities were recognized in the input:



1. If you want to know which node in the dialog tree triggered a response, click the **Location** Location icon next to it. If you are not already in the Dialog tab, open it.

The source node is given focus and the route that the service traversed through the tree to get to it is highlighted. It remains highlighted until you perform another action, such as entering a new test input.

1. To check or set the value of a context variable, click the **Manage context** link.

Any context variables that you defined in the dialog are displayed.

In addition, a $timezone context variable is listed. The Try it out pane user interface gets user locale information from the web browser and uses it to set the $timezone context variable. This context variable makes it easier to deal with time references in test dialog exchanges. Consider doing something similar in your user application. If not specified, Greenwich Mean Time (GMT) is used.

You can add a variable and set its value to see how the dialog responds in the next test dialog turn. This capability is helpful if, for example, the dialog is set up to show different responses based on a context variable value that is provided by the user.

* 1. To add a context variable, specify the variable name, and press **Enter**.
  2. To define a default value for the context variable, find the context variable you added in the list, and then specify a value for it.

See [Context variables](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context) for more information.

1. Continue to interact with the dialog to see how the conversation flows through it.
   1. To find and resubmit a test utterance, you can press the Up key to cycle through your recent inputs.
   2. To remove prior test utterances from the chat pane and start over, click the **Clear** link. Not only are the test utterances and responses removed, but this action also clears the values of any context variables that were set as a result of your interactions with the dialog. Context variable values that you explicitly set or change are not cleared.

### What to do next

If you determine that the wrong intents or entities are being recognized, you might need to modify your intent or entity definitions.

If the correct intents and entities are being recognized, but the wrong nodes are being triggered in your dialog, make sure your conditions are written properly.

## **Searching your dialog**

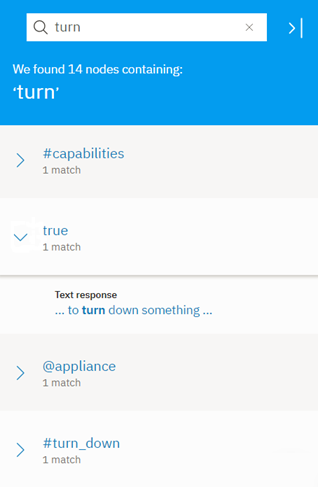
You can search the dialog to find one or more dialog nodes that mention a given word or phrase.

1. Select the Search icon: Search icon
2. Enter a search term or phrase.

**Note**: The first time you search, an index is created. You might be asked to wait while the text in your dialog nodes is indexed.

### Results

Nodes containing your search term, with corresponding examples, are shown. Select any result to open it for editing.



## **Finding a dialog node by its node ID**

You can search for a dialog node by its node ID. Enter the full node ID into the search field. You might want to find the dialog node that is associated with a known node ID for any of the following reasons:

* You are reviewing logs, and the log refers to a section of the dialog by its node ID.
* You want to map the node IDs listed in the nodes\_visited property of the API message output to nodes that you can see in your dialog tree.
* A dialog runtime error message informs you about a syntax error, and uses a node ID to identify the node you need to fix.

Another way to discover a node based on its node ID is by following these steps:

1. From the Dialog tab of the tooling, select any node in your dialog tree.
2. Close the edit view if it is open for the current node.
3. In your web browser's location field, a URL should display that has the following syntax:

https://watson-assistant.ng.bluemix.net/space/instance-id/workspaces/workspace-id/build/dialog#node=node-id

1. Edit the URL by replacing the current node-id value with the ID of the node you want to find, and then submit the new URL.
2. If necessary, highlight the edited URL again, and resubmit it.

The tooling refreshes, and shifts focus to the dialog node with the node ID that you specified. If the node ID is for a slot, a Found or Not found slot condition, a slot handler, or a conditional response, then the node in which the slot or conditional response is defined gets focus and the corresponding modal is displayed.

**Note**: If you still cannot find the node, you can export the workspace and use a JSON editor to search the workspace JSON file.

## **Copying a dialog node**

You can duplicate a node to create an exact copy of it as a peer node directly below it in the dialog tree. The copied node itself is given the same name as the original node, but with - copy*n* appended to it, where *n* is a number that starts with 1. If you duplicate the same node more than once, then the *n* in the name increments by one for each copy to help you distinguish the copies from one another. If the node has no name, it is given the name copy*n*.

When you duplicate a node that has child nodes, the child nodes are duplicated also. The copied child nodes have the exact same names as the original child nodes. The only way to distinguish a copied child node from an original child node is the copy reference in the parent node name.

1. On the node you want to copy, click the **More** More icon icon, and then select **Duplicate**.
2. Consider renaming the copied nodes or editing their conditions to make them distinct.

## **Moving a dialog node**

Each node that you create can be moved elsewhere in the dialog tree.

You might want to move a previously created node to another area of the flow to change the conversation. You can move nodes to become siblings or peers in another branch.

1. On the node you want to move, click the **More** More icon icon, and then select **Move**.
2. Select a target node that is located in the tree near where you want to move this node. Choose whether to place this node before or after the target node, or to make it a child of the target node.

## **Organizing the dialog with folders**

You can group dialog nodes together by adding them to a folder. There are lots of reasons to group nodes, including:

* To keep nodes that address a similar subject together to make them easier to find. For example, you might group nodes that address questions about user accounts in a User account folder and nodes that handle payment-related queries in a Payment folder.
* To group together a set of nodes that you want the dialog to process only if a certain condition is met. Use a condition, such as $isPlatinumMember, for example, to group together nodes that offer extra services that should only be processed if the current user is entitled to receive the extra services.
* To hide nodes from the runtime while you work on them. You can add the nodes to a folder with a falsecondition to prevent them from being processed.
* To apply the same configuration settings for digressing into a node to multiple root nodes at once. See [Digressions](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#digressions) for more information.

These characteristics of the folder impact how the nodes in a folder are processed:

* Condition: If specified, the service first evaluates the folder condition to determine whether to process the nodes within it.
* Customizations: Any configuration settings that you apply to the folder are inherited by the nodes in the folder. If you change the digression settings of the folder, for example, the changes are inherited by all the nodes in the folder.
* Tree hierarchy: Nodes in a folder are treated as root or child nodes based on whether the folder is added to the dialog tree at the root or child level. Any root level nodes that you add to a root level folder continue to function as root nodes; they do not become child nodes of the folder, for example. However, if you move root level nodes into a folder that is a child of another node, then the root nodes become children of that other node.

Folders have no impact on the order in which nodes are evaluated. Nodes continue to be processed from first to last. As the service travels down the tree, when it encounters a folder, if the folder condition is true, it immediately processes the first node in the folder, and continues down the tree in order from there. If a folder does not have a folder condition, it is transparent to the service.

### Adding a folder

To add a folder to a dialog tree, complete the following steps:

1. From the tree view of the **Dialog** tab, click **Add folder**.

The folder is added to the end of the dialog tree, just before the **Anything else** node. Unless an existing node in the tree is selected, in which case, it is added below the selected node.

If you want to add the folder elsewhere in the tree, from the node above the spot where you want to add it, click the **More** More icon icon, and then select **Add folder**.

You can add a folder below a child node within an existing dialog branch. To do so, click the **More** More icon icon on the child node, and then select **Add folder**.

The folder is opened in edit view.

1. **Optional**: Name the folder.
2. **Optional**: Define a condition for the folder.

If you do not specify a condition, true is used, meaning the nodes in the folder are always processed.

1. Add dialog nodes to the folder.
   * To add existing dialog nodes to the folder, you must move them to the folder one at a time.

On the node that you want to move, click the **More** More icon icon, select **Move**, and then click the folder. Select **To folder** as the move-to target.

As you move nodes, they are added at the start of the tree within the folder. Therefore, if you want to retain the order of a set of consecutive root dialog nodes, for example, move them starting with the last node first.

* + To add a new dialog node to the folder, click the **More** More icon icon on the folder, and then select **Add node to folder**.

The dialog node is added to the end of the dialog tree within the folder.

### Deleting a folder

You can delete either a folder alone or the folder and all of the dialog nodes in it.

To delete a folder, complete the following steps:

1. From the tree view of the **Dialog** tab, find the folder that you want to delete.
2. Click the **More** More icon icon on the folder, and then select **Delete**.
3. Do one of the following things:
   * To delete the folder only, and keep the dialog nodes that are in the folder, deselect the **Delete the nodes inside the folder** checkbox, and then click **Yes, delete it**.
   * To delete the folder and all of the dialog nodes in it, click **Yes, delete it**.

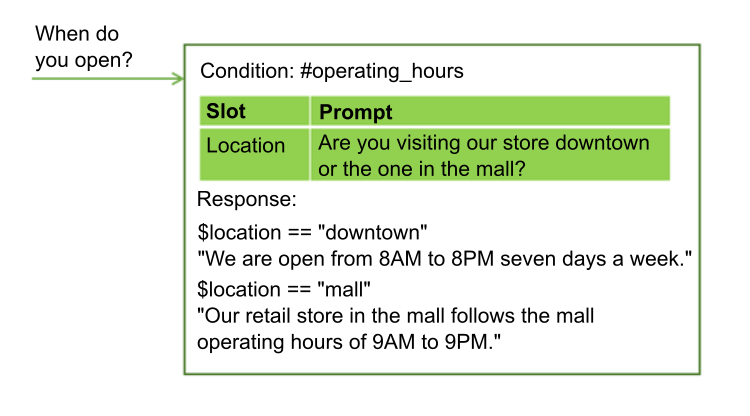
If you deleted the folder only, then the nodes that were in the folder are displayed in the dialog tree in the spot where the folder used be.

# Gathering information with slots

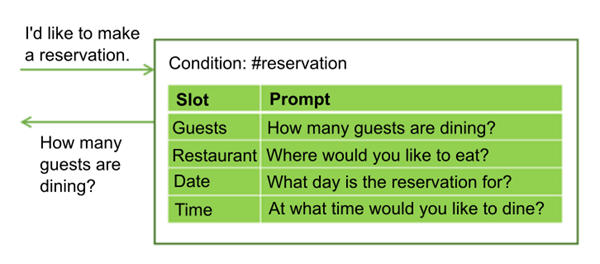
Add slots to a dialog node to gather multiple pieces of information from a user within that node. Slots collect information at the users' pace. Details the user provides upfront are saved, and the service asks only for the details they do not.

## **Why add slots?**

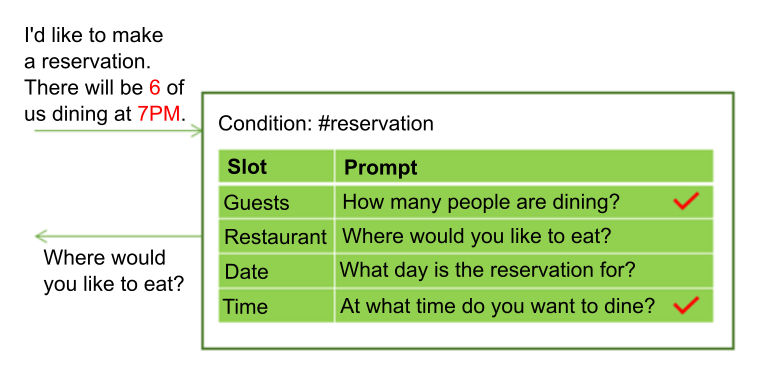
Use slots to get the information you need before you can respond accurately to the user. For example, if users ask about operating hours, but the hours differ by store location, you could ask a follow-up question about which store location they plan to visit before you answer. You can then add response conditions that take the provided location information into account.



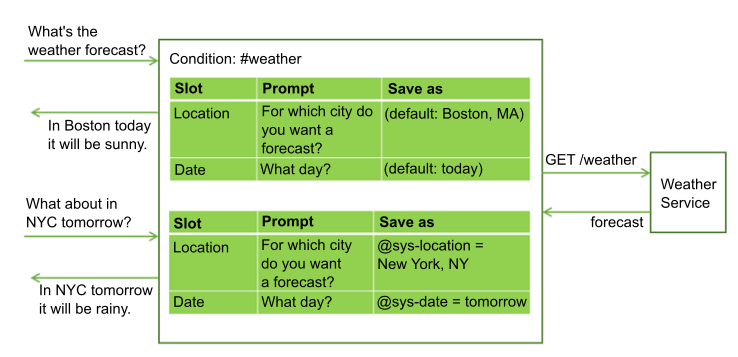
Slots can help you to collect multiple pieces of information that you need to complete a complex task for a user, such as making a dinner reservation.



The user might provide values for mutliple slots at once. For example, the input might include the information, There will be 6 of us dining at 7 PM. This one input contains two of the missing required values: the number of guests and time of the reservation. The service recognizes and stores both of them, each one in its corresponding slot. It then displays the prompt that is associated with the next empty slot.



Slots make it possible for the service to answer follow-up questions without having to re-establish the user's goal. For example, a user might ask for a weather forecast, then ask a follow-up question about weather in another location or on a different day. If you save the required forecast variables, such as location and day, in slots, then if a user asks a follow-up question with new variable values, you can overwrite the slot values with the new values provided, and give a response that reflects the new information. (For more information about how to call an external service from a dialog, see [Making programmatic calls from a dialog node](https://console.bluemix.net/docs/services/conversation/dialog-actions.html)).



Using slots produces a more natural dialog flow between the user and the service, and is easier for you to manage than trying to collect the information by using many separate nodes.

## **Adding slots**

1. Identify the units of information that you want to collect. For example, to order a pizza for someone, you might want to collect the following information:
   * Delivery time
   * Size
2. If you have not started to create a dialog, follow the instructions in [Creating a dialog](https://console.bluemix.net/docs/services/conversation/dialog-build.html) to create one.
3. From the dialog node edit view, click **Customize**, and then click the toggle next to **Slots** to turn it **On**.

**Note**: For more information about the **Prompt for everything** checkbox, see [Asking for everything at once](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-prompt-for-everything).

1. **Add a slot for each unit of required information**. For each slot, specify these details:
   * **Check for**: Identify the type of information you want to extract from the user's response to the slot prompt. In most cases, you check for entity values. In fact, the condition builder that is displayed suggests entities that you can check for. However, you can also check for an intent; just type the intent name into the field. You can use AND and OR operators here to define more complex conditions.

**Important**: The Check for value is first used as a condition, but then becomes the value of the context variable that you name in the Save as field. If you want to change how the value is saved (reformat it, for example), then add the expression that reformats the value directly to the **Check for** field.

For example, if the entity has regular expression patterns defined for it, then after adding the entity name, append .literal to it. After you choose @email from the list of defined entities, for example, edit the**Check for** field to contain @email.literal. By adding the .literal property, you indicate that you want to capture the exact text that was entered by the user and was identified as an email address based on its pattern. Make this syntax change directly in the **Check for** field.

**Warning** If you want to apply a complex expression to the value before you save the value, then you can open the JSON editor to define the complex SpEL expression. However, the complex expression that you define in the JSON editor will not be reflected in the **Check for** field when you exit the JSON editor. And if you click the **Check for** field to give the field focus at any time after you define the complex expression for the field, then the expression is removed.

Avoid checking for context variable values. Because the value you check for is also the value that is saved, when you use a context variable in the condition, it can lead to unexpected behavior when it gets used in the context. Do not try to use an optional slot to display a response only if a given context variable is set. If the variable is set, then the slot Found response that you define for the optional slot will be displayed along with the response that is returned by every other slot, over and over again.

* + **Save as**: Provide a name for the context variable in which to store the value of interest from the user's response to the slot prompt. Do not specify a context variable that is used earlier in the dialog, and therefor might have a value. It is only when the context variable for the slot is null that the prompt for the slot is displayed.
  + **Prompt**: Write a statement that elicits the piece of the information you need from the user. After displaying this prompt, the conversation pauses and the service waits for the user to respond.
  + If you want different follow-up statements to be shown based on whether the user provides the information you need in response to the initial slot prompt, you can edit the slot (by clicking the **Edit slot** Edit slot icon) and define the follow-up statements:
    - **Found**: Displayed after the user provides the expected information.
    - **Not found**: Displayed if the information provided by the user is not understood, or is not provided in the expected format. If the slot is filled successfully, or the user input is understood and handled by a slot handler, then this statement is never displayed.

For information about how to define conditions and associated actions for Found and Not found responses, see [Adding conditions to Found and Not found responses](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slot-handler-next-steps).

1. This table shows example slot values for a node that helps users place a pizza order by collecting two pieces of information, the pizza size and delivery time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Check for** | **Save as** | **Prompt** | **Follow-up if found** | **Follow-up if not found** |
| @size | $size | What size pizza would you like? | $size it is. | What size did you want? We have small, medium, and large. |
| @sys-time | $time | When do you need the pizza by? | For delivery by $time. | What time did you want it delivered? We need at least a half hour to prepare it. |
| Example slots for pizza order | | | | |

1. **Make a slot optional or disable it under certain conditions**. You can optionally configure a slot in these ways:
   * **Optional**: To make a slot optional, add a slot without a prompt. The service does not ask the user for the information, but it does look for the information in the user input, and saves the value if the user provides it. For example, you might add a slot that captures dietary restriction informations in case the user specifies any. However, you don't want to ask all users for dietary information since it is irrelevant in most cases.

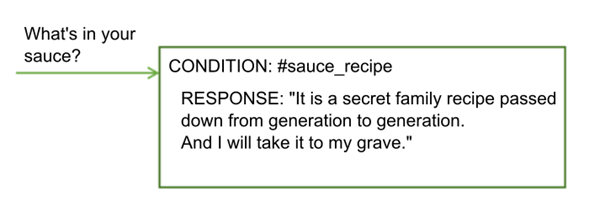
|  |  |  |
| --- | --- | --- |
| **Information** | **Check for** | **Save as** |
| Wheat restriction | @dietary | $dietary |
| Optional slot | | |

* + If you make a slot optional, only reference its context variable in the node-level response text if you can word it such that it makes sense even if no value is provided for the slot. For example, you might word a summary statement like this, I am ordering a $size $dietary pizza for delivery at $time. The resulting text makes sense whether the dietary restriction information, such as gluten-free or dairy-free, is provided or not. The result is either, I am ordering a large gluten-free pizza for delivery at 3:00PM. or I am ordering a large pizza for delivery at 3:00PM.
  + **Conditional**: If you want a slot to be enabled only under certain conditions, then you can add a condition to it. For example, if slot 1 asks for a meeting start time, slot 2 captures the meeting duration, and slot 3 captures the end time, then you might want to enable slot 3 (and ask for the meeting end time) only if a value for slot 2 is not provided. To make a slot conditional, edit the slot, and then from the **More** More icon menu, select **Enable condition**. Define the condition that must be met for the slot to be enabled.

You can condition on the value of a context variable from an earlier slot because the order in which the slots are listed is the order in which they are evaluated. However, only condition on a slot context variable that you can be confident will contain a value when this slot is evaluated. The earlier slot must be a required slot, for example.

1. **Keep users on track**. You can optionally define slot handlers that provide responses to questions users might ask during the interaction that are tangential to the purpose of the node.

For example, the user might ask about the tomato sauce recipe or where you get your ingredients. To handle such off-topic questions, click the **Manage handlers** link and add a condition and response for each anticipated question.



After responding to the off-topic question, the prompt associated with the current empty slot is displayed.

This condition is triggered if the user provides input that matches the slot handler conditions at any time during the dialog node flow up until the node-level response is displayed. See [Handling requests to exit a process](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-node-level-handler) for more ways to use the slot handler.

1. **Add a node-level response**. The node-level response is not executed until after all of the required slots are filled. You can add a response that summarizes the information you collected. For example, A $size pizza is scheduled for delivery at $time. Enjoy!

If you want to define different responses based on certain conditions, click **Customize**, and then click the **Multiple responses** toggle to turn it **On**. For information about conditional responses, see [Conditional responses](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#multiple).

1. **Add logic that resets the slot context variables**. As you collect answers from the user per slot, they are saved in context variables. You can use the context variables to pass the information to another node or to an application or external service for use. However, after passing the information, you must set the context variables to null to reset the node so it can start collecting information again. You cannot null the context variables within the current node because the service will not exit the node until the required slots are filled. Instead, consider using one of the following methods:
   * Add processing to the external application that nulls the variables.
   * Add a child node that nulls the variables.
   * Insert a parent node that nulls the variables, and then jumps to the node with slots.

Give it a try! Follow the step-by-step [tutorial](https://console.bluemix.net/docs/services/conversation/tutorial-slots.html).

## **Slots usage tips**

The following slot properties can help you check and set values in slot context variables.

| **Property name** | **Description** |
| --- | --- |
| all\_slots\_filled | Evaluates to true only if all of the context variables for all of the slots in the node have been set. See [Preventing a Found response from displaying when it is not needed](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-stifle-found-responses) for a usage example. |
| event.current\_value | Current value of the context variable for this slot. See [Replacing a slot context variable value](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-found-handler-event-properties) for a usage example for this property and the event.previous\_value property. |
| event.previous\_value | Previous value of the context variable for this slot. |
| has\_skipped\_slots | True if any of the slots or slot handlers that are configured with a next step option that skips slots was processed. See [Adding conditions to Found and Not found responses](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slot-handler-next-steps) for more information about next step options for slots and [Handling requests to exit a process](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-node-level-handler) for information about next step options for slot handlers. |
| slot\_in\_focus | Forces the slot condition to be applied to the current slot only. See [Getting confirmation](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-get-confirmation) for more details. |
| Slot properties | |

Consider using these approaches for handling common tasks.

* [Asking for everything at once](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-prompt-for-everything)
* [Capturing multiple values](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-multiple-entity-values)
* [Reformatting values](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-reformat-values)
* [Getting confirmation](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-get-confirmation)
* [Replacing a slot context variable value](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-found-handler-event-properties)
* [Avoiding number confusion](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-avoid-number-confusion)
* [Adding conditions to Found and Not found responses](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slot-handler-next-steps)
* [Moving on after multiple failed attempts](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-stop-trying-after-3)
* [Preventing a Found response from displaying when it is not needed](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-stifle-found-responses)
* [Handling requests to exit a process](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-node-level-handler)

### Asking for everything at once

Include an initial prompt for the whole node that clearly tells users which units of information you want them to provide. Displaying this prompt first gives users the opportunity to provide all the details at once and not have to wait to be prompted for each piece of information one at a time.

For example, when the node is triggered because a customer wants to order a pizza, you can respond with the preliminary prompt, I can take your pizza order. Tell me what size pizza you want and the time that you want it delivered.

If the user provides even one piece of this information in their initial request, then the prompt is not displayed. For example, the initial input might be, I want to order a large pizza. When the service analyzes the input, it recognizes large as the pizza size and fills the **Size** slot with the value provided. Because one of the slots is filled, it skips displaying the initial prompt to avoid asking for the pizza size information again. Instead, it displays the prompts for any remaining slots with missing information.

From the Customize pane where you enabled the Slots feature, select the **Prompt for everything** checkbox to enable the intial prompt. This setting adds the **If no slots are pre-filled, ask this first** field to the node, where you can specify the text that prompts the user for everything.

### Capturing multiple values

You can ask for a list of items and save them in one slot.

For example, you might want to ask users whether they want toppings on their pizza. To do so define an entity (@toppings), and the accepted values for it (pepperoni, cheese, mushroom, and so on). Add a slot that asks the user about toppings. Use the values property of the entity type to capture multiple values, if provided.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Check for** | **Save as** | **Prompt** | **Follow-up if found** | **Follow-up if not found** |
| @toppings.values | $toppings | Any toppings on that? | Great addition. | What toppings would you like? We offer ... |
| Multiple value slot | | | | |

To reference the user-specified toppings later, use the <? $entity-name.join(',') ?> syntax to list each item in the toppings array and separate the values with a comma. For example, I am ordering you a $size pizza with <? $toppings.join(',') ?> for delivery by $time.

### Reformatting values

Because you are asking for information from the user and need to reference their input in responses, consider reformatting the values so you can display them in a friendlier format.

For example, time values are saved in the hh:mm:ss format. You can use the JSON editor for the slot to reformat the time value as you save it so it uses the hour:minutes AM/PM format instead:

{

"context":{

"time": "<? @sys-time.reformatDateTime('h:mm a') ?>"

}

}

See [Methods to process values](https://console.bluemix.net/docs/services/conversation/dialog-methods.html) for other reformatting ideas.

### Getting confirmation

Add a slot after the others that asks the user to confirm that the information you have collected is accurate and complete. The slot can look for responses that match the #yes or #no intent.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Check for** | **Save as** | **Prompt** | **Follow-up if found** | **Follow-up if not found** |
| #yes || #no | $confirmation | I'm going to order you a $size pizza for delivery at $time. Should I go ahead? | Your pizza is on its way! | see Complex response |
| Confirmation slot | | | | |

**Complex response** Because users might include affirmative or negative statements at other times during the dialog (Oh yes, we want the pizza delivered at 5pm) or (no guests tonight, let's make it a small), use the slot\_in\_focusproperty to make it clear in the slot condition that you are looking for a Yes or No response to the prompt for this slot only.

(#yes || #no) && slot\_in\_focus

The slot\_in\_focus property always evaluates to a Boolean (true or false) value. Only include it in a condition for which you want a boolean result. Do not use it in slot conditions that checks for an entity type and then save the entity value, for example.

In the **Not found** prompt, clarify that you are expecting the user to provide a Yes or No answer.

{

"output":{

"text": {

"values": [

"Respond with Yes to indicate that you want the order to

be placed as-is, or No to indicate that you do not."

]

}

}

}

In the **Found** prompt, add a condition that checks for a No response (#no). When found, ask for the information all over again and reset the context variables that you saved earlier.

{

"conditions": "#no",

"output":{

"text": {

"values": [

"Let's try this again. Tell me what size pizza

you want and the time..."

]

}

},

"context":{

"size": null,

"time": null,

"confirmation": null

}

}

### Replacing a slot context variable value

If, at any time before the user exits a node with slots, the user provides a new value for a slot, then the new value is saved in the slot context variable, replacing the previously-specified value. Your dialog can acknowledge explicitly that this replacement has occurred by using special properties that are defined for the Found condition:

* event.previous\_value: Previous value of the context variable for this slot.
* event.current\_value: Current value of the context variable for this slot.

For example, your dialog asks for a destination city for a flight reservation. The user provides Paris. You set the $destination slot context variable to Paris. Then, the user says, Oh wait. I want to fly to Madrid instead.If you set up the Found condition as follows, then your dialog can handle this type of change gracefully.

When user responds, if @destination is found:

Condition: (event.previous\_value != null) &&

(event.previous\_value != event.current\_value)

Response: Ok, updating destination from

<? event.previous\_value ?> to <? event.current\_value ?>.

Response: Ok, destination is $destination.

This slot configuration enables your dialog to react to the user's change in destination by saying, Ok, updating the destination from Paris to Madrid.

### Avoiding number confusion

Some values that are provided by users can be identified as more than one entity type.

You might have two slots that store the same type of value, such as an arrival date and departure date, for example. Build logic into your slot conditions to distinguish such similar values from one another.

In addition, the service can recognize multiple entity types in a single user input. For example, when a user provides a currency, it is recognized as both a @sys-currency and @sys-number entity type. Do some testing in the Try it out pane to understand how the system will interpret different user inputs, and build logic into your conditions to prevent possible misinterpretations.

In logic that is unique to the slots feature, when two entities are recognized in a single user input, the one with the larger span is used. For example, if the user enters May 2, even though the Watson Assistant service recognizes both @sys-date (05022017) and @sys-number (2) entities in the text, only the entity with the longer span (@sys-date) is registered and applied to a slot.

### Adding conditions to Found and Not found responses

For each slot, you can use conditional responses with associated actions to help you extract the information you need from the user. To do so, follow these steps:

1. Click the **Edit slot** Edit slot icon for the slot to which you want to add conditional Found and Not found responses.
2. From the **More** More icon menu, select **Enable conditional responses**.
3. Enter the condition and the response to display if the condition is met.

**Found example**: The slot is expecting the time for a dinner reservation. You might use @sys-time in the Check forfield to capture it. To prevent an invalid time from being saved, you can add a conditional response that checks whether the time provided is before the restaurant's last seating time, for example. @sys-time.after('21:00:00') The corresponding response might be something like, Our last seating is at 9PM.

**Not found example**: The slot is expecting a @location entity that accepts a specific set of cities where the restaurant chain has restaurants. The Not found condition might check for @sys-location in case the user specifies a valid city, but one in which the chain has no sites. The corresponding response might be, We have no restaurants in that location.

1. If you want to customize what happens next if the condition is met, then click the **Edit response** Edit response icon.

For Found responses (that are displayed when the user provides a value that matches the value type specified in the Check for field), you can choose one of these actions to perform next:

* + **Move on (Default)**: Instructs the service to move on to the next empty slot after displaying the response. In the associated response, assure the user that their input was understood. For example, Ok. You want to schedule it for $date.
  + **Clear slot and prompt again**: If you are using an entity in the Check for field that could pick up the wrong value, add conditions that catch any likely misinterpretations, and use this action to clear the current slot value and prompt for the correct value.
  + **Skip to response**: If, when the condition you define is met, you no longer need to fill any of the remaining slots in this node, choose this action to skip the remaining slots and go directly to the node-level response next. For example, you could add a condition that checks whether the user's age is under 16. If so, you might skip the remaining slots which ask questions about the user's driving record.

For Not found responses (that are displayed when the user does not provide a valid value), you can choose one of these actions to perform:

* + **Wait for user input (Default)**: Pauses the conversation and the service waits for the user to respond. In the simplest case, the text you specify here can more explicitly state the type of information you need the user to provide. If you use this action with a conditional response, be sure to word the conditional response such that you clearly state what was wrong with the user's answer and what you expect them to provide instead.
  + **Prompt again**: After displaying the Not found response, the service repeats the slot prompt again and waits for the user to respond. If you use this action with a conditional response, the response can merely explain what was wrong about the answer the user provided. It does not need to reiterate the type of information you want the user to provide because the slot prompt typically explains that.

If you choose this option, consider adding at least one variation of the Not found response so that the user does not see the exact same text more than once. Take the opportunity to use different wording to explain to the user what information you need them to provide and in what format.

* + **Skip this slot**: Instructs the service to stop trying to fill the current slot, and instead, move on to the prompt for the next empty slot. This option is useful in a slot where you want to both make the slot optional and to display a prompt that asks the user for information. For example, you might have a @seating entity that captures restaurant seating preferences, such as outside, near the fireplace, private, and so on. You can add a slot that prompts the user with, Do you have any seating preferences? and checks for @seating.values. If a valid response is provided, it saves the preference information to $seating\_preferences. However, by choosing this action as the Not found response next step, you instruct the service to stop trying to fill this slot if the user does not provide a valid value for it.
  + **Skip to response**: If, when the condition you define is met, you no longer need to fill any of the remaining slots in this node, choose this action to skip the remaining slots and go directly to the node-level response next. For example, if after capturing the one-way flight information, the slot prompt is, Are you buying round trip tickets? the Not found condition can check for #No. If #No is found, use this option to skip the remaining slots that capture information about the return flight, and go straight to the node-level response instead.

Click **Back** to return to the edit view of the slot.

1. To add another conditional response, click **Add a response**, and then enter the condition and the response to display if the condition is met.

Be sure to add at least one response that will be displayed no matter what. You can leave the condition field blank for this catch all response. The service automatically populates the empty condition field with the truespecial condition.

1. Click **Save** to save your changes, close the edit view of the slot, and return to the edit view of the node.

### Moving on after multiple failed attempts

You can provide users with a way to exit a slot if they cannot answer it correctly after several attempts by using Not found conditional responses. In the catchall response, open the JSON editor to add a counter context variable that will keep track of the number of times the Not found response is returned. In an earlier node, be sure to set the initial counter context variable value to 0.

In this example, the service asks for the pizza size. It lets the user answer the question incorrectly 3 times before applying a size (medium) to the variable for the user. (You can include a confirmation slot where users can always correct the size when they are asked to confirm the order information.)

Check for: @size Save as: $size Not found catchall condition:

{

"output": {

"text": {

"values": [

"What size did you want? We have small, medium, and large."

],

"selection\_policy": "sequential"

}

},

"context": {

"counter": "<? context['counter'] + 1 ?>"

}

}

To respond differently after 3 attempts, add another Not found condition like this:

{

"conditions": "$counter > 1",

"output": {

"text": {

"values": [

"We will bring you a size medium pizza."

]

}

},

"context": {

"size": "medium"

}

This Not found condition is more precise than the Not found catchall condition, which defaults to true. Therefore, you must move this response so it comes before the original conditional response or it will never be triggered. Select the conditional response and use the up arrow to move it up.

### Preventing a Found response from displaying when it is not needed

If you specify Found responses for multiple slots, then if a user provides values for multiple slots at once, the Found response for at least one of the slots will be displayed. You probably want either the Found response for all of them or none of them to be returned.

To prevent Found responses from being displayed, you can do one of the following things to each Found response:

* Add a condition to the response that prevents it from being displayed if particular slots are filled. For example, you can add a condition, like !($size && $time), that prevents the response from being displayed if the $size and $time context variables are both provided.
* Add the !all\_slots\_filled condition to the response. This setting prevents the response from being displayed if all of the slots are filled. Do not use this approach if you are including a confirmation slot. The confirmation slot is also a slot, and you typically want to prevent the Found responses from being displayed before the confirmation slot itself is filled.

### Handling requests to exit a process

Add at least one slot handler that can recognize it when a user wants to exit the node.

For example, in a node that collects information to schedule a pet grooming appointment, you can add a slot handler that conditions on the #cancel intent, which recognizes utterances such as, Forget it. I changed my mind.

1. In the JSON editor for the handler, fill all of the slot context variables with dummy values to prevent the node from continuing to ask for any that are missing. And in the handler response, add a message such as, Ok, we'll stop there. No appointment will be scheduled.
2. Choose what action you want the service to take next from the following options:
   * **Prompt again (Default)**: Displays the prompt for the slot that the user was working with just before asking the off-topic question.
   * **Skip current slot**: Displays the prompt associated with the slot that comes after the slot that the user was working with just before asking the off-topic question. And the service makes not further attempts to fill the skipped slot.
   * **Skip to response**: Skips the prompts for all remaining empty slots including the slot the user was working with just before asking the off-topic question.
3. In the node-level response, add a condition that checks for a dummy value in one of the slot context variables. If found, show a final message such as, If you decide to make an appointment later, I'm here to help. If not found, it displays the standard summary message for the node, such as I am making a grooming appointment for your $animal at $time on $date.

Here's a sample of JSON that defines a slot handler for the pizza example. Note that, as described earlier, the context variables are all being set to dummy values. In fact, the $size context variable is being set to dummy. This $size value triggers the node-level response to show the appropriate message and exit the slots node.

{

"conditions": "#cancel",

"output": {

"text": {

"values": [

"Ok, we'll stop there. No pizza delivery will be scheduled."

],

"selection\_policy": "sequential"

}

},

"context": {

"time": "12:00:00",

"size": "dummy",

"confirmation":"true"

}

}

**Important**: Take into account the logic used in conditions that are evaluated before this condition so you can build distinct conditions. When a user input is received, the conditions are evaluated in the following order:

* Current slot level Found conditions.
* Slot handlers in the order they are listed.
* Current slot level Not found conditions.

Be careful about adding conditions that always evaluate to true (such as the special conditions, trueor anything\_else) as slot handlers. Per slot, if the slot handler evaluates to true, then the Not found condition is skipped entirely. So, using a slot handler that always evaluates to true effectively prevents the Not found condition for every slot from being evaluated.

For example, you groom all animals except cats. For the Animal slot, you might be tempted to use the following slot condition to prevent cat from being saved in the Animal slot:

Check for @animal && !@animal:cat, then save it as $animal.

And to let users know that you do not accept cats, you might specify the following value in the Not found condition of the Animal slot:

If @animal:cat then, "I'm sorry. We do not groom cats."

While logical, if you also define an #exit slot handler, then - given the order of condition evaluation - this Not found condition will likely never get triggered. Instead, you can use this slot condition:

Check for @animal, then save it as $animal.

And to deal with a possible cat response, add this value to the Found condition:

If @animal:cat then, "I'm sorry. We do not groom cats."

In the JSON editor for the Found condition, reset the value of the $animal context variable because it is currently set to cat and should not be.

{

"output":{

"text": {

"values": [

"I'm sorry. We do not groom cats."

]

}

},

"context":{

"animal": null

}

}

## **Slots examples**

To access JSON files that implement different common slot usage scenarios, go to the community [GitHub repo](https://github.com/watson-developer-cloud/community/tree/master/watson-assistant).

To explore an example, download one of the example JSON files, and then import it as a new workspace. From the Dialog tab, you can review the dialog nodes to see how slots were implemented to address different use cases.

# Making programmatic calls from a dialog node

Define actions that can make programmatic calls to external applications or services and get back a result as part of the processing that occurs within a dialog turn.

You can use an external service to validate information that you collected from the user, or perform calculations or string manipulations on the input which are too complex to be handled by using supported SpEL expressions and methods. Or you can interact with an external web service to get information, such as an air traffic service to check on a flight's expected arrival time or a weather service to get a forecast. You can even interact with an external application, such as a restaurant reservation site, to complete a simple transaction on the user's behalf.

When you define the programmatic call, you choose one of the following types:

* **client**: Defines a programmatic call in a standardized format that your external client application can use to perform the programmatic call or function, and return the result to the dialog. This type of call basically tells the dialog to pause here and let the client application go do something. The program that the client application runs can be anything that you choose. Just be sure to specify the call name and parameter details, as well as the error message variable name, according to the JSON formatting rules that are outlined later.
* **server**: Calls an Cloud Functions action directly, and returns the result to the dialog.

Currently, you can call a Cloud Functions action from Watson Assistant instances that are hosted in the US South or Germany regions.

**Note**: The Cloud Functions instance that is used is the one hosted in the same location (US South or Germany). Therefore, do not define an action in a Cloud Functions instance that is hosted in Germany if you plan to access it from a Watson Assistant service instance hosted in US South, for example.

**Important**: Only use this method to make a call to a Cloud Functions action that you know can return in **under 5 seconds**. The request to Cloud Functions times out if an individual service call takes longer than that. And if your dialog makes more than one call to an external service, the total amount of time allowed for the calls to complete is 7 seconds. If the first three calls complete in 2 seconds each, and the fourth takes more than 1 second, then the fourth call is stopped, and the error message for the call indicates that the call was not completed. For less efficient services that you need to call, manage the call through your client application and pass the information to the dialog as a separate step.

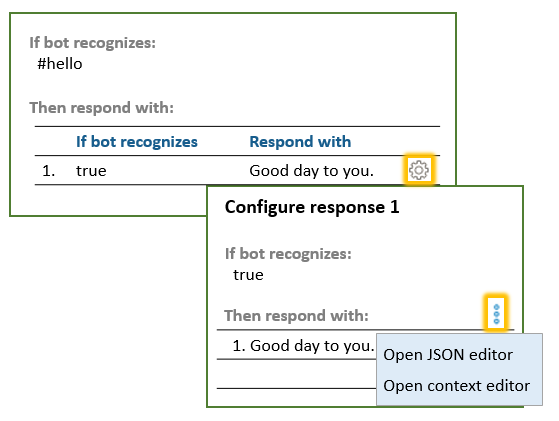
## **Procedure**

To make a programmatic call from a dialog node, complete the following steps:

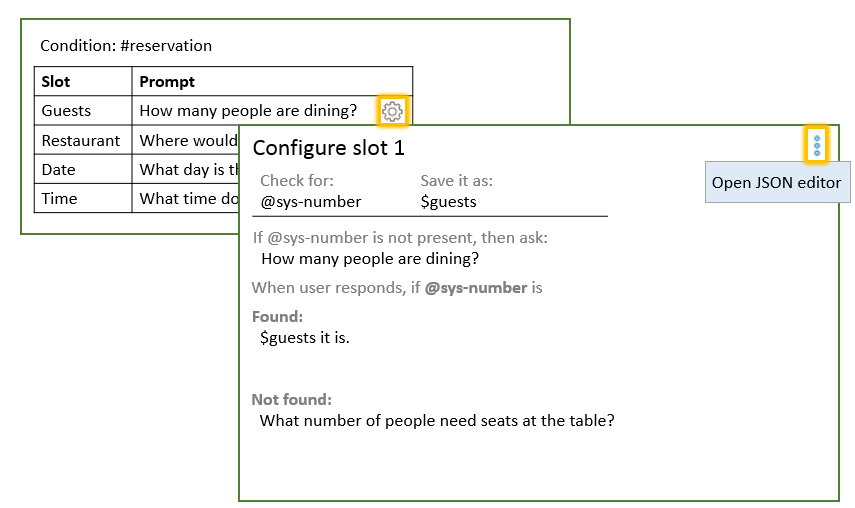
1. In the dialog node from which you want to make the programmatic call, open the JSON editor.
   * To make a programmatic call that is executed after the response for a node is evaluated, open the JSON editor for the node response.



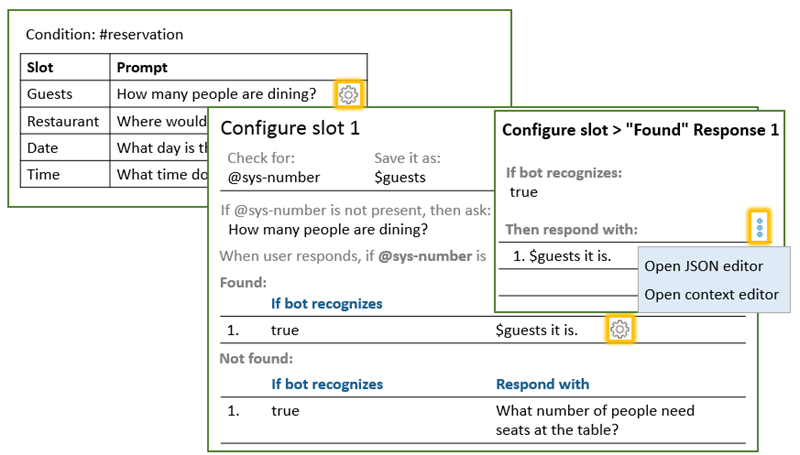
If the **Multiple responses** setting is **On** for the node, then you must click the **Edit response** Edit response icon before the **Options** Advanced response menu will be visible.



1. If you want to display or further process the response from the external service within the same dialog turn, then you must add a second node that does so, and jump to it from this node.
   * To make a call that can be used by an individual slot, click the **Edit slot** icon Edit slot icon for the slot, and then do one of the following things:
     + To make a programmatic call that is executed after the slot condition is evaluated to true, open the JSON editor that is associated with the slot condition.



* + - To make a programmatic call that is executed after the slot is successfully filled, open the JSON editor that is associated with the Found response. To do so, from the **Options** Options icon menu for the slot, click **Enable conditional responses**. For the Found response, click the **Edit response** Edit response icon. From the **Options** Options icon menu for the Found response, click **Open JSON editor**.



1. Use the following syntax to define the programmatic call.
2. {
3. "context": {
4. "variable\_name" : "variable\_value"
5. },
6. "actions": [
7. {
8. "name":"<actionName>",
9. "type":"client | server",
10. "parameters": {
11. "<parameter\_name>":"<parameter\_value>",
12. "<parameter\_name>":"<parameter\_value>"
13. },
14. "result\_variable": "<result\_variable\_name>",
15. "credentials": "<reference\_to\_credentials>"
16. }
17. ],
18. "output": {
19. "text": "response text"
20. }
21. }

The actions array specifies the programmatic calls to make from the dialog. It can define up to 5 separate programmatic calls. Specify the following name and value pairs in the JSON array:

* + <actionName>: Required. The name of the action or service to call. The name cannot be longer than 64 characters.
    - For client action types, specify a name in whatever syntax you want. The goal is to specify a name that your client application will recognize and know how to handle.

For example: calculateRate

* + - For Cloud Functions (server) action types, use this syntax to provide the fully qualified name of the action: /<namespace>/[<package-name>]/<action name>
      * If the action is part of a package, then the <package-name> information is required; otherwise it is not.
      * If you are calling a sequence of actions, then specify the <sequence name> in place of the <action name>.
      * The namespace for a user-defined action typically has the syntax: <myIBMCloudOrganizationID>\_<myIBMCloudSpace>. For example: /jdoeorg\_prod10/search flights
      * The actions that are provided with Cloud Functions often have the namespace: whisk.system, but you should always verify the namespace to be sure. For example: /whisk.system/weather/forecast
  + <type>: Indicates the type of call to make. Choose from the following types:
    - **client**: Sends a message response with programmatic call information in a standardized format that your external client application can use to perform the call or function, and get a result on behalf of the dialog. The JSON object in the response body specifies the service or function to call, any associated parameters to pass with the call, and how the result should be sent back.
    - **server**: Calls a Cloud Functions action (one or more) directly. You must define the action itself separately by using IBM® Cloud Functions. For more information, see [Creating a Cloud Functions action](https://console.bluemix.net/docs/services/conversation/dialog-actions.html#create-action) below.

Specifying the type is optional. The default value is client.

* + <action\_parameters>: Any parameters that are expected by the external program, specified as a JSON object. Parameters are only required if the external program requires them.
  + <result\_variable\_name>: The name to use to reference the JSON object that is returned by the external service or program. The result is added to the context section of the /message response. In other words, the result is stored as a context variable so it can be displayed in the node response or accessed by dialog nodes that are triggered later. Any existing value for the context variable is overwritten by the value that is returned by the action. You can specify the result\_variable\_name by using the following syntax:
    - my\_result
    - $my\_result

The name cannot be longer than 64 characters. The variable name cannot contain the following characters: parentheses (), square brackets ([]), a single quote ('), a quotation mark ("), or a back slash (\).

If you want to save the result to the output or input section of the /message response, then you can prepend one of the following location keywords to the result\_variable\_name:

* + - output.: Adds the result to the output section of the /message response. For example, output.my\_result.
    - input.: Adds the result to the input section of the /message response. For example, input.my\_result.

You can specify a context. location keyword prefix also. For example, context.my\_result. However, you do not need to because the result is added to the context by default.

You can include periods in the variable name to create a nested JSON object. For example, you can define these variables to capture results from two separate requests to a weather service for forecasts for today and tomorrow:

* + - context.weather.today
    - context.weather.tomorrow

The results (temp and rain parameter values) are stored in the context in this structure:

{

"weather": {

"today": {

"temp": "20",

"rain": "30"

},

"tomorrow": {

"temp": "23",

"rain": "80"

}

}

}

If multiple actions in a single JSON action array add the result of their programmatic call to the same context variable, then the order in which the context is updated matters:

* + - If you have a combination of server and client actions in the array, the service processes the server type actions first. As a result, the value that is calculated for the context variable by the last client type action in the array overwrites the value calculated for it by any server type actions.
    - Per action type, the order in which the actions are defined in the array determines the order in which the context variable's value is set. The context variable value returned by the last action in the array overwrites the values calculated by any other actions.
  + <reference\_to\_credentials>: The name of the object in which the Cloud Functions credentials are stored. Required for server actions only. These credentials are used to access the Cloud Functions instance on which the action runs. These are not your IBM Cloud credentials.

To discover the credentials, complete the following steps:

* + - Go to the [Cloud Functions API key](https://console.bluemix.net/openwhisk/learn/api-key) page.
      * If you have not yet created an account, do so.
      * If you are not logged in, log in.
    - Click the **Show Auth Key** icon Show Auth Key to show the credentials. The segment before the colon (:) is your user ID. The segment after the colon is your password.

**Attention**: Any charges that are incurred when the action runs are charged to the person who owns these credentials.

To protect the credentials, do not store them in the Watson Assistant workspace. Instead, pass them from the client application as part of context. You can prevent the information from being stored in Watson logs by nesting your context variable within the $private section of the message context. For example: $private.my\_credentials.

The credentials object that you define must contain parameters named user and password.

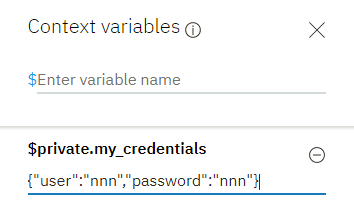
{

"user":"5tj3b41j-bf3j-5d92-24g9-4a7769ab12af",

"password":"y65gqSTSRzqE..."

}

While testing the dialog, you can temporarily set the $private.my\_credentials context variable with your real Cloud Functions username and password values by clicking **Manage context** from the "Try it out" pane in the tooling.



## **Creating a Cloud Functions action**

If you choose to define a server type programmatic call, then before you can call it from a dialog, you must create the action in IBM® Cloud Functions. If you are defining a client type programmatic call, then skip this procedure.

**Note:** Running a Cloud Functions action might incur a cost. For more information, see [Pricing](https://console.bluemix.net/openwhisk/learn/pricing). Cloud Functions does not distinguish between calls that are made from the "Try it out" pane during testing and calls that are made from an application in production.

To create a Cloud Functions action, complete the following steps:

1. Go to the [online Cloud Functions editor](https://console.ng.bluemix.net/openwhisk/create), where you can write your code directly in your browser.

There is also a [command line interface](https://console.bluemix.net/openwhisk/learn/cli) you can install that enables you to define an action using code you write locally.

1. Create a Cloud Functions action using one of the supported programming languages. See the [Cloud Functions documentation](https://console.ng.bluemix.net/docs/openwhisk/openwhisk_actions.html) for details.

Keep the following tips in mind:

* + Refer to the [example of a Cloud Functions action](https://console.ng.bluemix.net/docs/openwhisk/openwhisk_actions.html#openwhisk_apicall_action) to see how to call an external service.
  + To make a call to a Watson service, use the [Watson Developer Cloud SDK](https://github.com/watson-developer-cloud) for the language you want to use.
  + Make sure your Cloud Functions action accepts any input parameters as a JSON object, and returns any output as a JSON object.
  + If you are using Node.js to write your Cloud Functions action, make sure you use Promise for asynchronous processing. Also make sure you return the final result from the main function.

Alternatively, you can create a sequence of actions.

## **Handling errors**

If the Cloud Functions action encounters an error, the error message is returned to the dialog and is stored as a property of the response variable named cloud\_functions\_call\_error. The error might occur if your Cloud Functions action cannot get a response from an external service, or if the Cloud Function action fails, for example. If the Cloud Function credentials are not provided or are incorrect, an error is returned. This context variable is used for server actions only; in your client application, consider creating a similar object that captures error information and returns it to the dialog as a context variable.

You can condition the dialog node response to first check for errors. For example, you can ensure that the response that references a Cloud Functions action result is shown only if no errors were encountered by adding this expression to the response condition:

$forecast\_result.cloud\_functions\_call\_error == null

For a client type programmatic call, you can pass information about error processing by defining a context variable, such as action\_error. You can pass it back to the service as part of the result variable. Then, you can display a response only if no errors were encountered by defining a response condition like this:

$forecast\_result.action\_error == null

## **Client call example**

The following example shows what a call to an external weather service might look like. It is added to the JSON editor that is associated with the node response. By the time the node-level response is triggered, slots have collected and stored the date and location information from the user. This example assumes that the service that will be called has an endpoint named /weather, and that it takes location and date parameters, and returns a JSON object,{"forecast": "<value>"}.

{

"actions": [

{

"name": "MyWeatherFunction",

"type": "client",

"parameters": {

"date": "$date",

"location": "$location"

},

"result\_variable": "context.my\_forecast"

}

]

}

Normally, the service only returns to the client from a POST /message request when new user input is required, such as after executing a parent and before executing one of its child nodes. However, if you add a client action to a node, then after evaluation, the service always returns to the client so that the result of the action call can be returned. To prevent waiting for user input when it should not, such as for a node that is configured to jump directly to a child node, the service adds the following value to the message context:

{

"context": {

"skip\_user\_input": true

}

}

If you want the client to perform an action, but not get user input, then you can follow the same convention, and add the skip\_user\_input context variable to the parent node to communicate that to the client application.

Your client application should always check for the skip\_user\_input variable on context. If present, then it knows not to request new input from the user, but instead execute the action, add its result into the message, and pass it back to the service. The new POST message request should include the message returned by the previous POST message response (namely, the context, input, intents, entities, and optionally the output section) and, instead of the JSON object that defines the programmatic call to make, it should include the result that was returned from the programmatic call.

In a child node that you jump to after this node, add the response to show the user:

{

"output": {

"text": {

"values": [

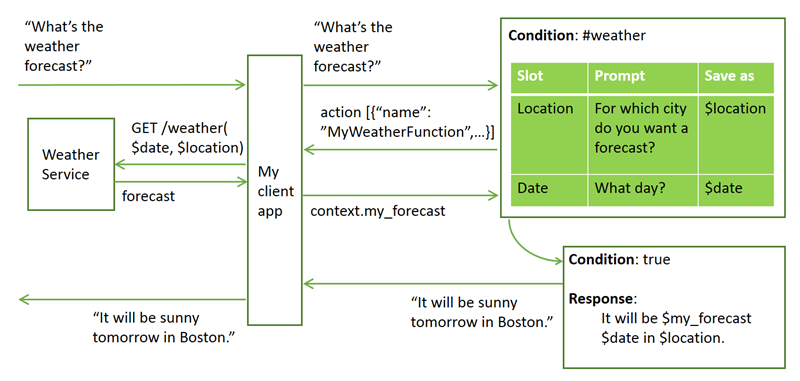
"It will be $my\_forecast $date.literal in $location.literal."

]

}

}

The following diagram illustrates how you can use a client call to get weather forecast information, and return it to the user.



## **Server call example**

The following example shows what a call to a Cloud Functions action might look like. This example shows how to use the Cloud Functions echo action that is defined in the [Utilities package](https://console.bluemix.net/docs/openwhisk/openwhisk_actions.html#openwhisk_create_action_sequence) provided with the service. The action takes a text string, and returns it.

{

"actions": [

{

"name": "/whisk.system/utils/echo",

"type":"server",

"parameters": {

"message": "<?input.text?>"

},

"result\_variable": "context.my\_input\_returned",

"credentials":"$private.my\_credentials"

}

]

}

The output of the Cloud Functions action, which is stored in the context.my\_input\_returned variable, can now be accessed by subsequent dialog nodes.

{

"output": {

"text": {

"values": [

"Your input was: $my\_input\_returned."

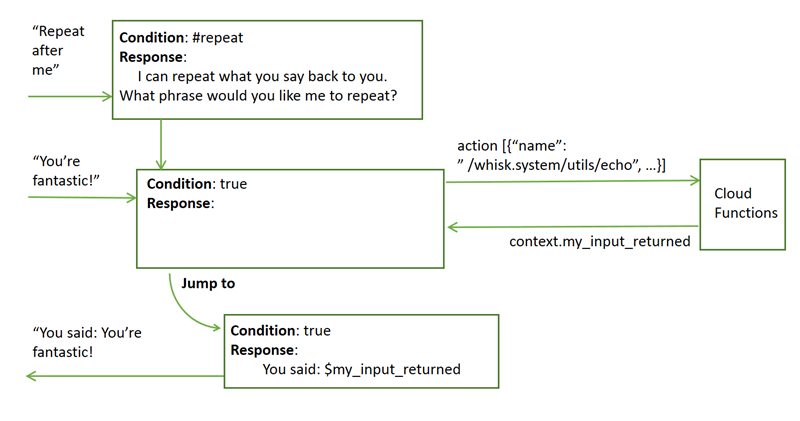
]

}

}

}

The following diagram illustrates how to call a Cloud Functions action with a simple example that calls the built-in Cloud Functions echo service. It asks the user for input and passes it to the echo service. The echo service returns the same text back, which is displayed to the user.



### Echo action example

To see a workspace with a dialog that is already set up to call the Cloud Functions built-in Echo action, complete the following steps:

1. Download the [CloudFunctionsEcho.json](https://github.com/watson-developer-cloud/community/raw/master/watson-assistant/cloud-functions-echo.json) file.
2. Import the JSON file as a new workspace.
3. Review the dialog to see how the call to the Echo action is specified.
4. From the "Try it out" pane, click **Manage context**, and then (temporarily) set the context variables to your Cloud Functions username and password.
5. $private.my\_credentials
6. {
7. "user":"<your-CF-instance-username>",
8. "password":"<your-CF-instance-password>"
9. }
10. Test the dialog by entering some input.

The service will use the Cloud Functions Echo action to repeat whatever you enter back to you.

## **Advanced server call example**

You can call multiple actions from within a single dialog flow. In fact, you can call up to five actions within one actions JSON object in a single dialog node. However, any server type actions that are defined in an actions JSON array are all processed in parallel. Therefore, you cannot call one server type action and pass the result from it to a second server type action in the same actions block. The best way to call server actions in a specific order is to use a Cloud Functions sequence. At runtime, this approach is faster because the dialog only has to make one external call to complete multiple actions. To use a sequence, just reference the sequence name instead of an action name in theactions block definition. Alternatively, you can call the first server type action from one node and jump to a child node that calls the next server type action.

If you define one actions array that has a mix of client type and server type actions, then when the dialog node is executed, the client actions are not sent to the client until after all the server type actions have been processed.

The following examples show what a call to a Cloud Functions action might look like.

This example shows how to use a Cloud Functions action to call an external service that takes a city name and returns the latitude and longitude coordinates of the provided location.

{

"actions": [

{

"name": "/jdoeorg\_prod/get coordinates",

"type":"server",

"parameters": {

"location": "$location"

},

"result\_variable": "context.my\_coordinates",

"credentials":"$private.my\_credentials"

}

]

}

The $my\_coordinates context variable saves the two values that are returned by the get coordinates service like this:

{

"lat":"42.3611",

"long":"-71.0571"

}

This example shows how to use the Cloud Functions forecast action that is defined in the [Weather package](https://console.bluemix.net/docs/openwhisk/openwhisk_weather.html#openwhisk_catalog_weather)provided with the Cloud Functions service. The action expects latitude and longitude coordinates, and a time period. It returns a JSON object with forecast information for the specified location over the specified time period. The coordinates, which are returned by the earlier action, are specified as $my\_coordinates.lat and $my\_coordinates.long.

{

"actions": [

{

"name": "/whisk.system/weather/forecast",

"type":"server",

"parameters": {

"latitude": "$my\_coordinates.lat",

"longitude": "$my\_coordinates.long",

"timePeriod": "$period",

"username": "$private.my\_weather\_service.username",

"password": "$private.my\_weather\_service.password"

},

"result\_variable": "context.forecasts",

"credentials":"$private.my\_credentials"

}

]

}

**Note**: A username and a password are listed as parameters. These are only present because this particular action requires them; together they define the credentials required by the external Weather service that the provided action calls on the back end. They are different from the IBM Cloud Function account credentials. Take steps to keep these credentials private as well.

The output of the Cloud Functions action, which is stored in the context.forecasts variable, can now be accessed by subsequent dialog nodes.

{

"output": {

"text": {

"values": [

"For the next $period in $location, you can expect $forecasts."

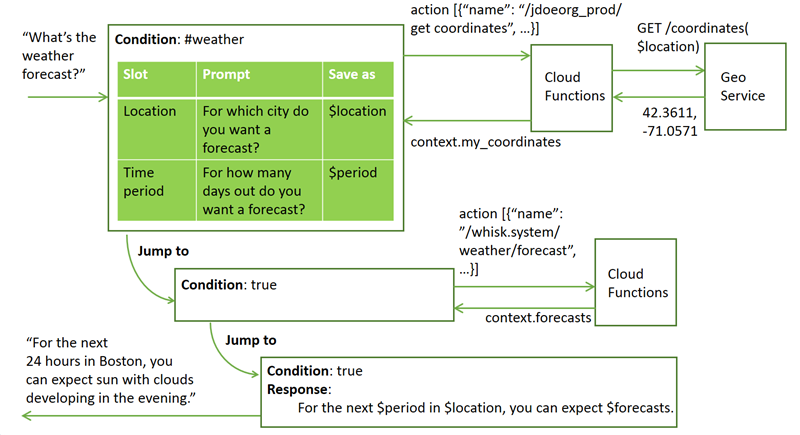
]

}

}

}

The following diagram illustrates a complex interaction. A user asks for a weather forecast. The slots in the dialog node prompt the user for location and time period information. To call the Cloud Functions forecast service, you must provide geographic coordinates. Therefore, the dialog first gets latitude and longitude details for the location provided by the user by sending a request to the Cloud Functions service, which passes it to an external service that returns the coordinate information. In a subsequent node, the dialog sends the newly obtained coordinate information along with the time period information it got from the user to the Cloud Functions built-in forecast service to get the forecast details. The dialog then answers the user's question by showing the forecast result that was provided by the Cloud Functions forecast service.



# Modifying a dialog using the API

The Watson Assistant REST API supports modifying your dialog programmatically, without using the Watson Assistant tool. You can use the /dialog\_nodes API to create, delete, or modify dialog nodes.

Remember that the dialog is a tree of interconnected nodes, and that it must conform to certain rules in order to be valid. This means that any change you make to a dialog node might have cascading effects on other nodes, or on the structure of your dialog. Before using the /dialog\_nodes API to modify your dialog, make sure you understand how your changes will affect the rest of the dialog. You can make a backup copy of the current dialog by exporting the workspace in which it resides. See [Exporting and copying workspaces](https://console.bluemix.net/docs/services/conversation/configure-workspace.html#exporting-and-copying-workspaces) for details.

A valid dialog always satisfies the following criteria:

* Each dialog node has a unique ID (the dialog\_node property).
* A child node is aware of its parent node (the parent property). However, a parent node is not aware of its children.
* A node is aware of its immediate previous sibling, if any (the previous\_sibling property). This means that all siblings that share the same parent form a linked list, with each node pointing to the previous node.
* Only one child of a given parent can be the first sibling (meaning that its previous\_sibling is null).
* A node cannot point to a previous sibling that is a child of a different parent.
* Two nodes cannot point to the same previous sibling.
* A node can specify another node that is to be executed next (the next\_step property).
* A node cannot be its own parent or its own sibling.
* A node must have a type property that contains one of the following values. If no type property is specified, then the type is standard.
  + event\_handler: A handler that is defined for a frame node or an individual slot node.

From the tooling, you can define a frame node handler by clicking the **Manage handlers** link from a node with slots. (The tooling user interface does not expose the slot-level event handler, but you can define one through the API.)

* + frame: A node with one or more child nodes of type slot. Any child slot nodes that are required must be filled before the service can exit the frame node.

The frame node type is represented as a node with slots in the tooling. The node that contains the slots is represented as a node of type=frame. It is the parent node to each slot, which is represented as a child node of type slot.

* + response\_condition: A conditional response.

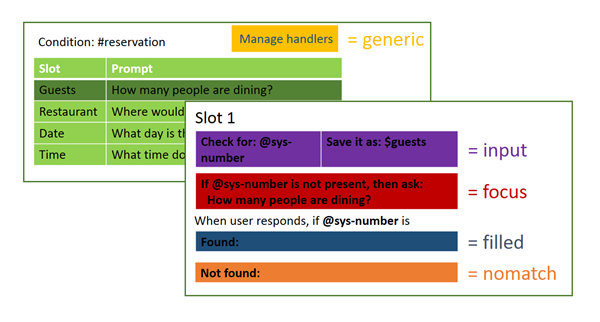
In the tooling, you can add one or more conditional responses to a node. Each conditional response that you define is represented in the underlying JSON as an individual node of type=response\_condition.

* + slot: A child node of a node of type frame.

This node type is represented in the tooling as being one of multiple slots added to a single node. That single node is represented in the JSON as a parent node of type frame.

* + standard: A typical dialog node. This is the default type.
* For nodes of type slot that have the same parent node, the sibling order (specified by the previous\_sibling property) reflects the order in which the slots will be processed.
* A node of type slot must have a parent node of type frame.
* A node of type frame must have at least one child node of type slot.
* A node of type response\_condition must have a parent node of type standard or frame.
* Nodes of type response\_condition and event\_handler cannot have children.
* A node of type event\_handler must also have an event\_name property that contains one of the following values to identify the type of node event:
  + filled: Defines what to do if the user provides a value that meets the condition specified in the Check for field of a slot, and the slot is filled in. A handler with this name is only present if a Found condition is defined for the slot.
  + focus: Defines the question to show to prompt the user to provide the information needed by the slot. A handler with this name is only present if the slot is required.
  + generic: Defines a condition to watch for that can address unrelated questions that users might ask while filling a slot or node with slots.
  + input: Updates the message context to include a context variable with the value that is collected from the user to fill the slot. A handler with this name must be present for each slot in the frame node.
  + nomatch: Defines what to do if the user's response to the slot prompt does not contain a valid value. A handler with this name is only present if a Not found condition is defined for the slot.

The following diagram illustrates where in the tooling user interface you define the code that is triggered for each named event.



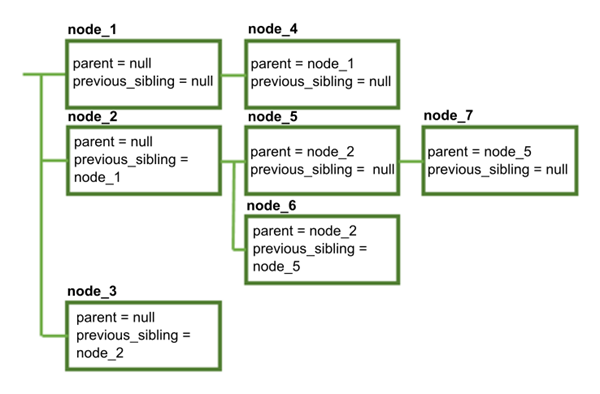
* A node of type event\_handler with an event\_name of generic can have a parent of type slot or frame.
* A node of type event\_handler with an event\_name of focus, input, filled, or nomatch must have a parent of type slot.
* If more than one event\_handler with the same event\_name is associated with the same parent node, then the order of the siblings reflects the order in which the event handlers will be executed.
* For event\_handler nodes with the same parent slot node, the order of execution is the same regardless of the placement of the node definitions. The events are triggered in this order by event\_name:
  + focus
  + input
  + filled
  + generic\*
  + nomatch

\*If an event\_handler with the event\_name generic is defined for this slot or for the parent frame, then it is executed between the filled and nomatch event\_handler nodes.

The following examples show how various modifications might cause cascading changes.

## **Creating a node**

Consider the following simple dialog tree:



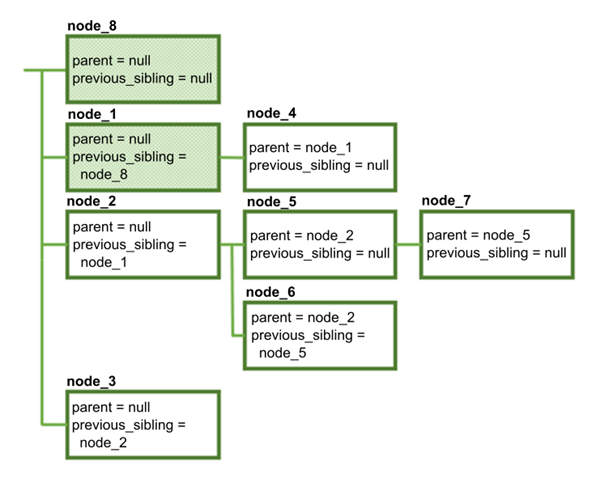
We can create a new node by making a POST request to /dialog\_nodes with the following body:

{

"dialog\_node": "node\_8"

}

The dialog now looks like this:



Because **node\_8** was created without specifying a value for parent or previous\_sibling, it is now the first node in the dialog. Note that in addition to creating **node\_8**, the service also modified **node\_1** so that its previous\_sibling property points to the new node.

You can create a node somewhere else in the dialog by specifying the parent and previous sibling:

{

"dialog\_node": "node\_9",

"parent": "node\_2",

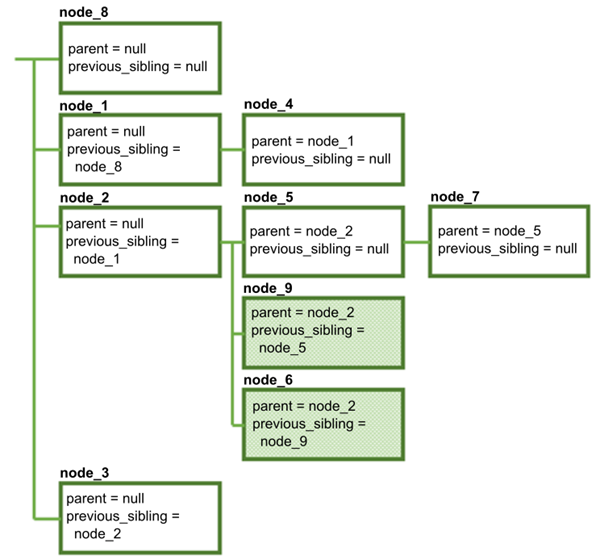
"previous\_sibling": "node\_5"

}

The values you specify for parent and previous\_node must be valid:

* Both values must refer to existing nodes.
* The specified parent must be the same as the parent of the previous sibling (or null, if the previous sibling has no parent).
* The parent cannot be a node of type response\_condition or event\_handler.

The resulting dialog looks like this:



In addition to creating **node\_9**, the service automatically updates the previous\_sibling property of node\_6 so that it points to the new node.

## **Moving a node to a different parent**

Let's move **node\_5** to a different parent by using the POST /dialog\_nodes/node\_5 method with the following body:

{

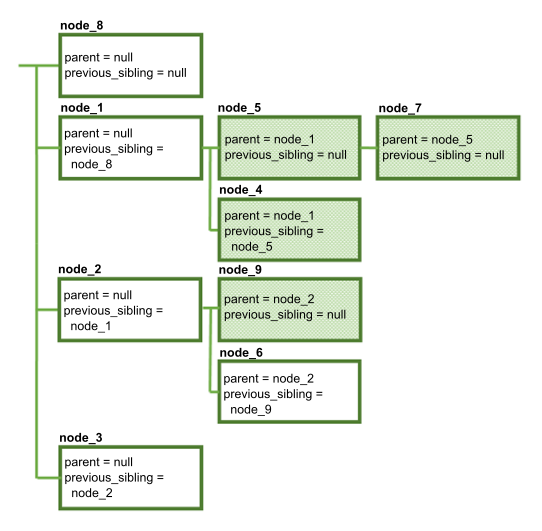
"parent": "node\_1"

}

The specified value for parent must be valid:

* It must refer to an existing node.
* It must not refer to the node being modified (a node cannot be its own parent).
* It must not refer to a descendant of the node being modified.
* It must not refer to a node of type response\_condition or event\_handler.

This results in the following changed structure:



Several things have happened here:

* When **node\_5** moved to its new parent, **node\_7** went with it (because the parent value for **node\_7** did not change). When you move a node, all descendants of that node stay with it.
* Because we did not specify a previous\_sibling value for **node\_5**, it is now the first sibling under **node\_1**.
* The previous\_sibling property of **node\_4** was updated to node\_5.
* The previous\_sibling property of **node\_9** was updated to null, because it is now the first sibling under **node\_2**.

## **Resequencing siblings**

Now let's make **node\_5** the second sibling instead of the first. We can do this by using the POST /dialog\_nodes/node\_5 method with the following body:

{

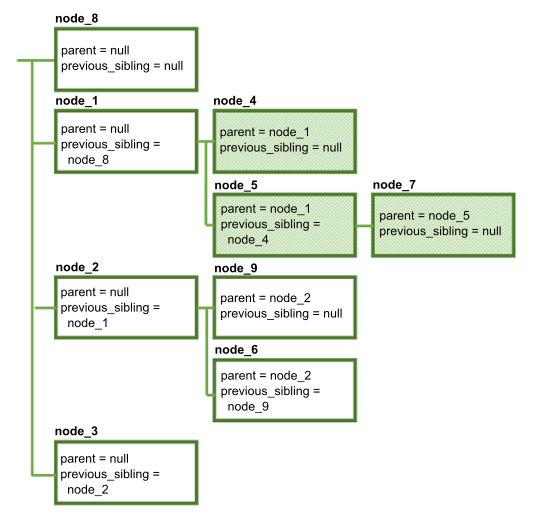
"previous\_sibling": "node\_4"

}

When you modify previous\_sibling, the new value must be valid:

* It must refer to an existing node
* It must not refer to the node being modified (a node cannot be its own sibling)
* It must refer to a child of the same parent (all siblings must have the same parent)

The structure changes as follows:

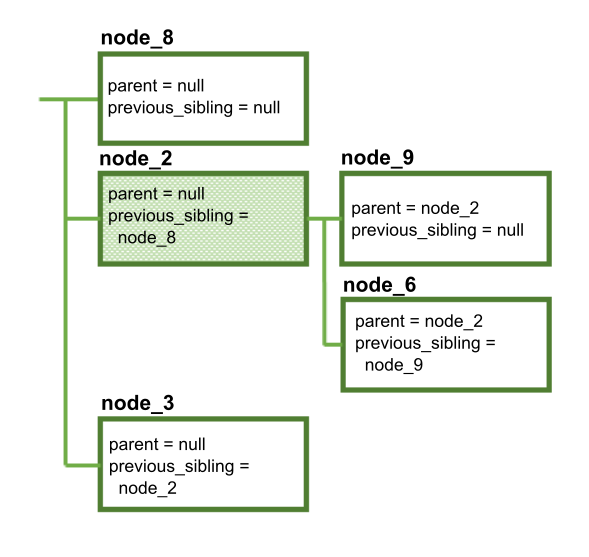


Note that once again, **node\_7** stays with its parent. In addition, **node\_4** is modified so that its previous\_sibling is null, because it is now the first sibling.

## **Deleting a node**

Now let's delete **node\_1**, using the DELETE /dialog\_nodes/node\_1 method.

The result is this:



Note that **node\_1**, **node\_4**, **node\_5**, and **node\_7** were all deleted. When you delete a node, all descendants of that node are deleted as well. Therefore, if you delete a root node, you are actually deleting an entire branch of the dialog tree. Any other references to the deleted node (such as next\_step references) are changed to null.

In addition, **node\_2** is updated to point to **node\_8** as its new previous sibling.

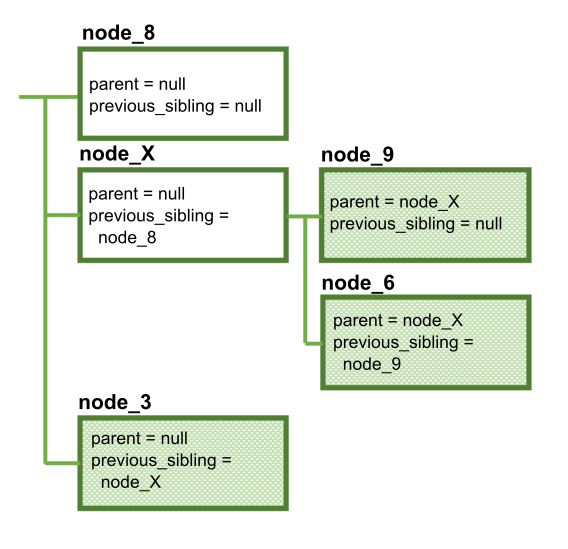
## **Renaming a node**

Finally, let's rename **node\_2** using the POST /dialog\_nodes/node\_2 method with the following body:

{

"dialog\_node": "node\_X"

}



The structure of the dialog has not changed, but once again multiple nodes were modified to reflect the changed name:

* The parent properties of **node\_9** and **node\_6**
* The previous\_sibling property of **node\_3**

Any other references to the deleted node (such as next\_step references) are also changed.

# Deployment overview

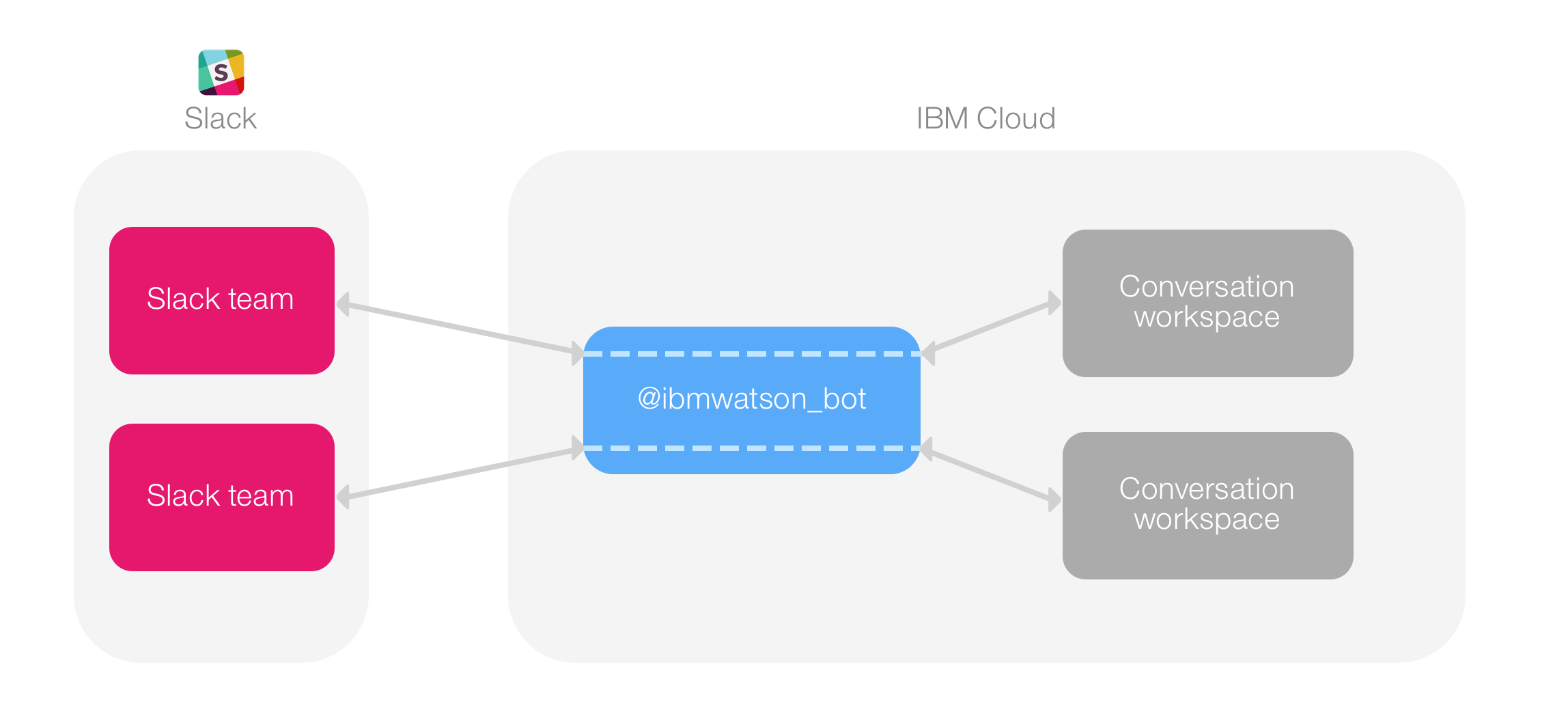
After you have configured your workspace and built a working dialog, you can deploy your application by connecting the workspace to the interface your customers will use. You can also connect your workspace to other services (including other Watson services). There are various ways you can do this.

* [**Testing in Slack**](https://console.bluemix.net/docs/services/conversation/test-deploy.html): You can use the test deployment tool to quickly deploy a chat bot in a Slack channel, using your workspace and a prebuilt Slack app. This option is the fastest and easiest way to test your workspace, but it has limitations.
* [**Deploying to a channel with the Watson Assistant connector**](https://console.bluemix.net/docs/services/conversation/conversation-connector.html): You can use the Watson Assistant connector to deploy a chat bot to Slack or Facebook Messenger, using your workspace and your own Slack or Facebook app.
* [**Building a client application**](https://console.bluemix.net/docs/services/conversation/develop-app.html): You can use the Watson SDK for the language of your choice to develop your own application front end. Your application can then use the Watson Assistant APIs to communicate with the service, while also integrating with third-party services and your own business systems.
* [**Building a bot with Botkit**](https://console.bluemix.net/docs/services/conversation/integrations.html): You can use the Botkit framework to integrate your workspace with social media and messaging channels.

# Testing in Slack

You can use the test deployment tool to integrate your Watson Assistant workspace into a Slack team as a bot user. Use this method if you want to quickly test using a Slack bot as the user interface for your workspace.

The test deployment tool uses the IBM® Cloud Functions service to deploy a prebuilt Slack application to your team as a bot user. This application handles communication with your Watson Assistant workspaces.

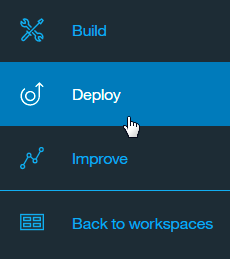


Note that the test deployment tool has some limitations:

* You cannot use this tool to publish an application for other teams to use.
* If you use this method to deploy more than one workspace to the same team, all of the workspaces will respond to the @ibmwatson\_bot user name. It is recommended that you use this tool to deploy only one workspace at a time to each Slack team.
* You must have permission to install apps to your Slack team. Check with your Slack administrator if you are not sure whether you have this permission.
* The prebuilt Slack application is for testing purposes only, and might not be available at all times.
* Because of Cloud Functions restrictions, this tool is currently available only for the IBM Cloud US South region.

To install your application as a bot user:

1. In the Watson Assistant tool, open the workspace you want to test in Slack.
2. Click the menu icon in the upper left corner, and then select **Deploy**. The Deploy Options page opens.



1. Under **Deploy with Cloud Functions**, click **Test in Slack** and follow the instructions.

Create Slack test button

## **Chatting with the bot**

After you complete the deployment process, you can use the @ibmwatson\_bot user name to interact with your Watson Assistant workspace, just as you would with any other Slack bot.

Keep in mind that the bot deployed to your team preserves state for each user within a particular channel. This means that any variables you store in the dialog context are maintained indefinitely, unless your dialog clears them.

If you need to be able to reset the conversation to a known starting state, you must do so within your dialog. Make sure that your dialog has a node that is executed at the end of the conversation or any other time you need to start over. Update the JSON for this node to reset all context variables to the appropriate starting values. (If necessary, you can use **Jump to** actions or a special "end conversation" intent to execute this node.)

For example, if your workspace uses a context variable called drink\_order to store a user's beverage selection, you can use the context.remove method to delete this variable when the conversation ends:

"context": {

"reset\_drink\_order": "<?context.remove('drink\_order')?>"

}

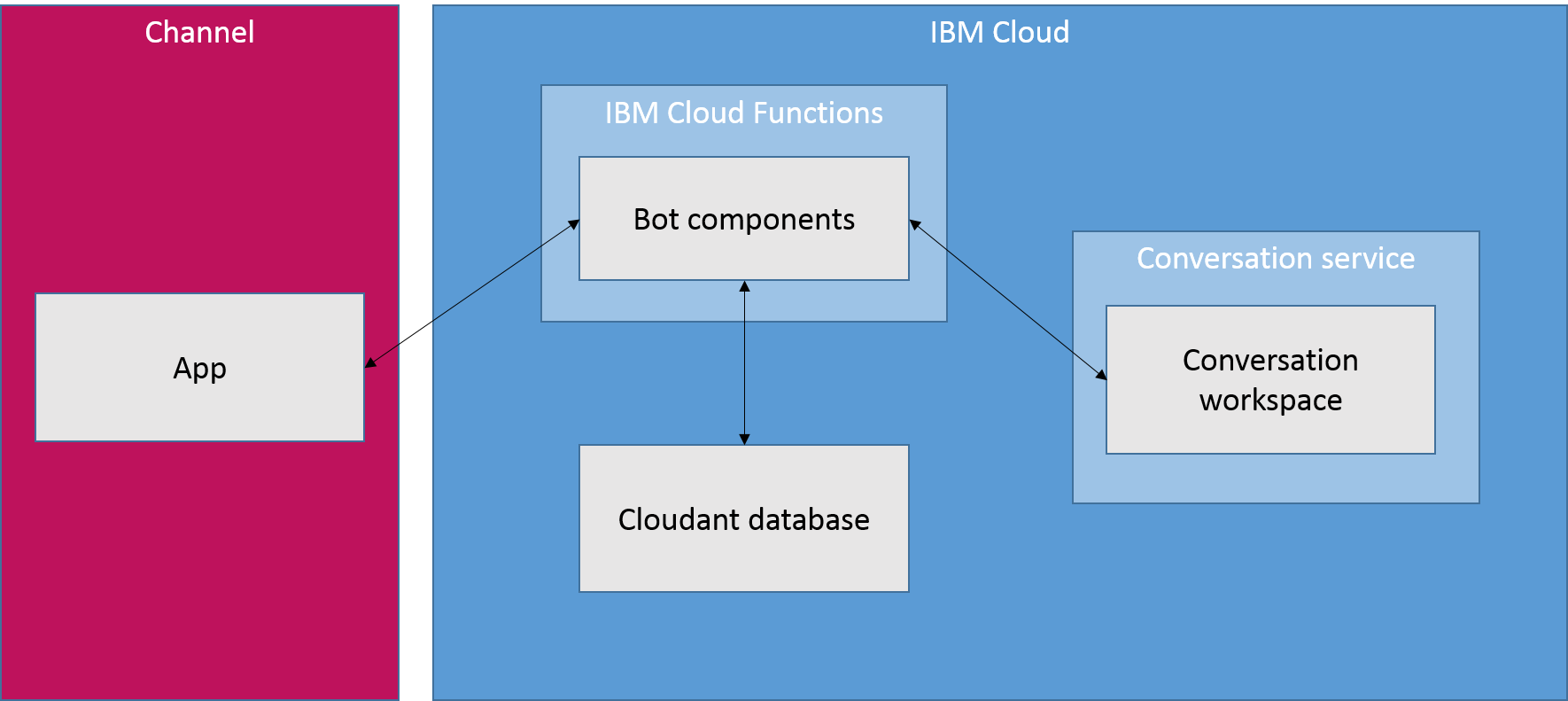
For more information about modifying context variable values, see [Updating a context variable value](https://console.bluemix.net/docs/services/conversation/dialog-overview.html#updating-a-context-variable-value).

**Note:** When you have finished testing your workspace, you can delete the test deployment by going back to the test deployment tool and clicking **Delete test**. Remember that you must also separately deauthorize the bot application in your Slack team.

# Deploying to a channel with the Conversation connector

After you have developed your workspace, you can use the Conversation connector to quickly connect it to a communication channel such as Slack or Facebook Messenger. The Conversation connector is a set of IBM® Cloud Functions components that mediate communication between your Watson Assistant workspace and a Slack or Facebook app you own, storing session data in a Cloudant database.

By connecting your workspace to a channel app, you can make it available as a chat bot that Slack or Facebook Messenger users can interact with. The responses from your dialog can include multimedia and interactive responses such as images and clickable buttons. (For more information about defining a multimedia response, see [Multimedia responses](https://console.bluemix.net/docs/services/conversation/dialog-multimedia.html).)



The Conversation connector has some limitations:

* You must create a Slack or Facebook app, or have administrative permissions to modify the app you want to use.
* Because of Cloud Functions restrictions, this tool is currently available only for the IBM Cloud US South region.

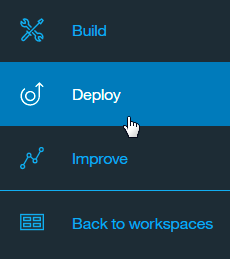
## **Deploying to Slack using the Watson Assistant tool**

The Watson Assistant tool provides a quick way to deploy a bot using the Conversation connector. The tool steps you through the process of gathering the required information to configure your channel app and the connector components, and then it automatically deploys the required components to the IBM Cloud.

**Note:** The Watson Assistant tool interface currently supports only Slack, but you can use an automated IBM® Cloud process to deploy for Facebook Messenger. For more information about deploying for Facebook Messenger, see the documentation in the Conversation connector [GitHub repository](https://github.com/watson-developer-cloud/conversation-connector/blob/master/channels/facebook/README.md).

To deploy a bot using the automated deployment tool:

1. In the Watson Assistant tool, open the workspace you want to deploy.
2. Click the menu icon in the upper left corner, and then select **Deploy**.



1. Under **Deploy with Cloud Functions**, click **Deploy to Slack**.
2. On the Slack page, click **Deploy to Slack app** and follow the instructions.

Deploy to Slack app button

## **Deploying manually**

Instead of using the automated tool to deploy your workspace as a bot, you can manually configure and deploy the required components to the IBM Cloud. There are several situations in which you might want to do this:

* **Partial redeployment**. You might want to redeploy some components of an existing deployment in order to change your configuration, correct problems, or apply a fix from a newer version.
* **Extending your bot with new capabilities**. You can modify the provided components to add new functions, such as preprocessing user input before sending it to the Watson Assistant workspace.
* **Deploying to a new channel**. If you want to deploy a bot to a channel other than Slack or Facebook Messenger, you can follow the pattern of the existing components to develop your own components.

The Conversation connector components are hosted in a public GitHub repository, which you can download or clone. For more information, see the documentation provided in the [repository](https://github.com/watson-developer-cloud/conversation-connector).

## **Chatting with the bot**

After you complete the deployment process, you can use the bot user name to interact with your Watson Assistant workspace, just as you would with any other Slack or Facebook Messenger bot.

Keep in mind that the Conversation connector preserves state separately for each user who is interacting with the bot. (In the case of Slack, a single user might have multiple unique conversations with the bot, one through direct messages and one in each channel.) This means that any variables you store in the dialog context are maintained indefinitely, unless you clear them.

If you need to be able to reset the conversation to a known starting state, you can do so within your dialog. Make sure that your dialog has a node that is executed at the end of the conversation or any other time you need to start over. Update the JSON for this node to reset all context variables to the appropriate starting values. (If necessary, you can use **Jump to** actions or a special "end conversation" intent to execute this node.)

For example, if your workspace uses a context variable called drink\_order to store a user's beverage selection, you can use the context.remove method to delete this variable when the conversation ends:

"context": {

"reset\_drink\_order": "<?context.remove('drink\_order')?>"

}

For more information about modifying context variable values, see [Updating a context variable value](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#context-update).

You can also clear the context, or make other changes to the behavior of the bot, by editing the deployed Cloud Functions actions. For more information, see documentation provided in the [GitHub repository](https://github.com/watson-developer-cloud/conversation-connector).

# Building a client application

So you have a working dialog. Now you want to develop the application that will interact with your users and communicate with the IBM Watson™ Assistant service.

You can view this tutorial for Node.js (Javascript), Python, or Java by clicking the language selector in the upper right. For details about all supported languages, refer to the Watson [SDKs](https://console.bluemix.net/docs/services/watson/getting-started-sdks.html#sdks).

## **Setting up the Watson Assistant service**

The example application we will create in this section implements several functions of a cognitive personal assistant. The application code will connect to a Watson Assistant workspace, where the cognitive processing (such as the detection of user intents) takes place.

Before continuing with this example, you need to set up the required Watson Assistant workspace:

1. Download the workspace [JSON file](https://watson-developer-cloud.github.io/doc-tutorial-downloads/conversation/conversation-simple-example.json).
2. [Import the workspace](https://console.bluemix.net/docs/services/conversation/configure-workspace.html#creating-workspaces) into an instance of the Watson Assistant service.

## **Getting service information**

To access the Watson Assistant service REST APIs, your application needs to be able to authenticate with IBM® Cloud and connect to the right Watson Assistant workspace. You'll need to copy the service credentials and workspace ID and paste them into your application code.

To access the service credentials and the workspace ID from your workspace, select the Menu menu, choose **Deploy**, and then go to the **Credentials** tab.

You can also access the service credentials from your IBM® Cloud dashboard.

## **Communicating with the Watson Assistant service**

Interacting with the Watson Assistant service is simple. Let's take a look at an example that connects to the service, sends a single message, and prints the output to the console:

// Example 1: sets up service wrapper, sends initial message, and

// receives response.

var AssistantV1 = require('watson-developer-cloud/assistant/v1');

// Set up Assistant service wrapper.

var service = new AssistantV1({

username: 'USERNAME', // replace with service username

password: 'PASSWORD', // replace with service password

version: '2018-02-16'

});

var workspace\_id = 'WORKSPACE\_ID'; // replace with workspace ID

// Start conversation with empty message.

service.message({

workspace\_id: workspace\_id

}, processResponse);

// Process the service response.

function processResponse(err, response) {

if (err) {

console.error(err); // something went wrong

return;

}

// Display the output from dialog, if any.

if (response.output.text.length != 0) {

console.log(response.output.text[0]);

}

}

The first step is to create a wrapper for the Watson Assistant service.

The wrapper is an object you will use to send input to, and receive output from, the service. When you create the service wrapper, specify the authentication credentials from the service key, as well as the version of the Watson Assistant API you are using.

In this Node.js example, the wrapper is an instance of AssistantV1, stored in the variable service. The Watson SDKs for other languages provide equivalent mechanisms for instantiating a service wrapper.

After creating the service wrapper, we use it to send a message to the Watson Assistant service. In this example, the message is empty; we just want to trigger the conversation\_start node in the dialog, so we don't need any input text.

Use the node <filename.js> command to run the example application.

**Note:** Make sure you have installed the Watson SDK for Node.js using npm install watson-developer-cloud.

Assuming everything works as expected, the Watson Assistant service returns the output from the dialog, which is then printed to the console:

Welcome to the Watson Assistant example!

This output tells us that we have successfully communicated with the Watson Assistant service and received the welcome message specified by the conversation\_start node in the dialog. Now we can add a user interface, making it possible to process user input.

## **Processing user input to detect intents**

To be able to process user input, we need to add a user interface to our application. For this example, we'll keep things simple and use standard input and output. We can use the Node.js prompt-sync module to do this. (You can install prompt-sync using npm install prompt-sync.)

// Example 2: adds user input and detects intents.

var prompt = require('prompt-sync')();

var AssistantV1 = require('watson-developer-cloud/assistant/v1');

// Set up Assistant service wrapper.

var service = new AssistantV1({

username: 'USERNAME', // replace with service username

password: 'PASSWORD', // replace with service password

version: '2018-02-16'

});

var workspace\_id = 'WORKSPACE\_ID'; // replace with workspace ID

// Start conversation with empty message.

service.message({

workspace\_id: workspace\_id

}, processResponse);

// Process the service response.

function processResponse(err, response) {

if (err) {

console.error(err); // something went wrong

return;

}

// If an intent was detected, log it out to the console.

if (response.intents.length > 0) {

console.log('Detected intent: #' + response.intents[0].intent);

}

// Display the output from dialog, if any.

if (response.output.text.length != 0) {

console.log(response.output.text[0]);

}

// Prompt for the next round of input.

var newMessageFromUser = prompt('>> ');

service.message({

workspace\_id: workspace\_id,

input: { text: newMessageFromUser }

}, processResponse)

}

This version of the application begins the same way as before: sending an empty message to the Watson Assistant service to start the conversation.

The processResponse() function now displays any intent detected by the dialog along with the output text, and then it then prompts for the next round of user input.

But something still isn't right:

Welcome to the Watson Assistant example!

>> hello

Detected intent: #hello

Welcome to the Watson Assistant example!

>> what time is it?

Detected intent: #time

Welcome to the Watson Assistant example!

>> goodbye

Detected intent: #goodbye

Welcome to the Watson Assistant example!

>>

The Watson Assistant service is detecting the correct intents, and yet every turn of the conversation returns the welcome message from the conversation\_start node (Welcome to the Watson Assistant example!).

This is happening because the Watson Assistant service is stateless; it is the responsibility of the application to maintain state information. Because we are not yet doing anything to maintain state, the Watson Assistant service sees every round of user input as the first turn of a new conversation, triggering the conversation\_start node.

## **Maintaining state**

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

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

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

// Example 3: maintains state.

var prompt = require('prompt-sync')();

var AssistantV1 = require('watson-developer-cloud/assistant/v1');

// Set up Assistant service wrapper.

var service = new AssistantV1({

username: 'USERNAME', // replace with service username

password: 'PASSWORD', // replace with service password

version: '2018-02-16'

});

var workspace\_id = 'WORKSPACE\_ID'; // replace with workspace ID

// Start conversation with empty message.

service.message({

workspace\_id: workspace\_id

}, processResponse);

// Process the service response.

function processResponse(err, response) {

if (err) {

console.error(err); // something went wrong

return;

}

// If an intent was detected, log it out to the console.

if (response.intents.length > 0) {

console.log('Detected intent: #' + response.intents[0].intent);

}

// Display the output from dialog, if any.

if (response.output.text.length != 0) {

console.log(response.output.text[0]);

}

// Prompt for the next round of input.

var newMessageFromUser = prompt('>> ');

// Send back the context to maintain state.

service.message({

workspace\_id: workspace\_id,

input: { text: newMessageFromUser },

context : response.context,

}, processResponse)

}

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

service.message({

input: { text: newMessageFromUser },

context : response.context,

}, processResponse)

This ensures that the context is maintained from one turn to the next, so the Watson Assistant service no longer thinks every turn is the first:

>> hello

Detected intent: #hello

Good day to you.

>> what time is it?

Detected intent: #time

>> goodbye

Detected intent: #goodbye

OK! See you later.

>>

Now we're making progress! The Watson Assistant service is correctly recognizing our intents, and the dialog is returning the correct output text (where provided) for each intent.

However, nothing else is happening. When we ask for the time, we get no answer; and when we say goodbye, the conversation does not end. That's because those intents require additional actions to be taken by the app.

## **Implementing app actions**

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

These action flags are sent using the action property, which our dialog defines as part of the response JSON. When the dialog determines that the application needs to do something, it sets the value of action to the appropriate value, either display\_time or end\_conversation.

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

But in our example, we're using a simple key/value pair that supports a single action flag. Our application code needs to check the value of the action property in the response and then carry out any specified action. (This version also removes the display of detected intents, now that we're sure those are being correctly identified.)

// Example 4: implements app actions.

var prompt = require('prompt-sync')();

var AssistantV1 = require('watson-developer-cloud/assistant/v1');

// Set up Assistant service wrapper.

var service = new AssistantV1({

username: 'USERNAME', // replace with service username

password: 'PASSWORD', // replace with service password

version: '2018-02-16'

});

var workspace\_id = 'WORKSPACE\_ID'; // replace with workspace ID

// Start conversation with empty message.

service.message({

workspace\_id: workspace\_id

}, processResponse);

// Process the service response.

function processResponse(err, response) {

if (err) {

console.error(err); // something went wrong

return;

}

var endConversation = false;

// Check for action flags.

if (response.output.action === 'display\_time') {

// User asked what time it is, so we output the local system time.

console.log('The current time is ' + new Date().toLocaleTimeString() + '.');

} else if (response.output.action === 'end\_conversation') {

// User said goodbye, so we're done.

console.log(response.output.text[0]);

endConversation = true;

} else {

// Display the output from dialog, if any.

if (response.output.text.length != 0) {

console.log(response.output.text[0]);

}

}

// If we're not done, prompt for the next round of input.

if (!endConversation) {

var newMessageFromUser = prompt('>> ');

service.message({

workspace\_id: workspace\_id,

input: { text: newMessageFromUser },

// Send back the context to maintain state.

context : response.context,

}, processResponse)

}

}

The processResponse() function now checks the value of the action property of the output object received from the Watson Assistant service. If the value is either display\_time or end\_conversation, the application carries out the appropriate action.

Welcome to the Watson Assistant example!

>> hello

Good day to you.

>> what time is it?

The current time is 12:40:42 PM.

>> goodbye

OK! See you later.

Success! The application now uses the Watson Assistant service to identify the intents in natural-language input, displays the appropriate responses, and implements the requested client actions.

Of course, a real-world application would use a more sophisticated user interface, such as a web chat window. And it would implement more complex actions, possibly integrating with a customer database or other business systems. But the basic principles of how the application interacts with the Watson Assistant service would remain the same.

For some more complex examples, see [Sample apps](https://console.bluemix.net/docs/services/conversation/sample-applications.html).

# Building a bot with BotKit

If you want customers to be able to communicate with your workspace through social media and messaging channels, you can use the [Botkit](http://howdy.ai/botkit) middleware plugin.

This plugin enables developers to easily integrate a Watson Assistant workspace with multiple social channels, including Slack, Facebook, and Twilio. After these integrations are established, customers can have simultaneous, independent conversations with your Watson Assistant workspace through different channels.

To use the Botkit middleware plugin, clone or download the [GitHub repository](https://github.com/watson-developer-cloud/botkit-middleware).

# Tutorial: Building a complex dialog

In this tutorial, you will use the Watson Assistant service to create a dialog that helps users interact with a cognitive car dashboard.

## **Learning objectives**

By the time you finish the tutorial, you will understand how to:

* Define entities
* Plan a dialog
* Use node and response conditions in a dialog

### Duration

This tutorial will take approximately 2 to 3 hours to complete.

### Prerequisite

Before you begin, complete the [Getting Started tutorial](https://console.bluemix.net/docs/services/conversation/getting-started.html).

You will use the Watson Assistant tutorial workspace that you created, and add nodes to the simple dialog that you built as part of the getting started exercise.

## **Step 1: Add intents and examples**

Add an intent on the Intents tab. An intent is the purpose or goal expressed in user input.

1. On the Intents page of the Watson Assistant tutorial workspace, click **Add intent**.
2. Add the following intent name, and then click **Create intent**:
3. turn\_on

A # is prepended to the intent name you specify. The #turn\_on intent indicates that the user wants to turn on an appliance such as the radio, windshield wipers, or headlights.

1. In the **Add user example** field, type the following utterance, and then click **Add example**:
2. I need lights
3. Add these 5 more examples to help Watson recognize the #turn\_on intent.
4. Play some tunes
5. Turn on the radio
6. turn on
7. Air on please
8. Crank up the AC
9. Turn on the headlights
10. Click the **Close** Close arrow icon to finish adding the #turn\_on intent.

You now have three intents, the #turn\_on intent that you just added, and the #hello and #goodbye intents that were added in the Getting started tutorial that you completed as a prerequisite step. Each intent has a set of example utterances that help train Watson to recognize the intents in user input.

## **Step 2: Add entities**

An entity definition includes a set of entity values that can be used to trigger different responses. Each entity value can have multiple synonyms, which define different ways that the same value might be specified in user input.

Create entities that might occur in user input that has the #turn\_on intent to represent what the user wants to turn on.

1. Click the **Entities** tab to open the Entities page.
2. Click **Add entity**.
3. Add the following entity name, and then press Enter:
4. appliance

A @ is prepended to the entity name you specify. The @appliance entity represents an appliance in the car that a user might want to turn on.

1. Add the following value to the **Value name** field:
2. radio

The value represents a specific appliance that users might want to turn on.

1. Add other ways to specify the radio appliance entity in the **Synonyms** field. Press **Tab** to give the the field focus, and then enter the following synonyms. Press **Enter** after each synonym.
2. music
3. tunes
4. Click **Add value** to finish defining the radio value for the @appliance entity.
5. Add other types of appliances.
   * Value: headlights. Synonym: lights.
   * Value: air conditioning. Synonyms: air and AC.
6. Click the toggle to turn fuzzy matching **On** for the @appliance entity. This setting helps the service recognize references to entities in user input even when the entity is specified in a way that does not exactly match the syntax you use here.
7. Click the **Close** Close arrow icon to finish adding the @appliance entity.
8. Repeat Steps 2-8 to create the @genre entity with fuzzy matching on, and these values and synonyms:
   * Value: classical. Synonym: symphonic.
   * Value: rhythm and blues Synonym: r&b.
   * Value: rock. Synonym: rock & roll, rock and roll, and pop.

You defined two entities: @appliance (representing an appliance the bot can turn on) and @genre (representing a genre of music the user can choose to listen to).

When the user's input is received, the Watson Assistant service identifies both the intents and entities. You can now define a dialog that uses intents and entities to choose the correct response.

## **Step 3: Create a complex dialog**

In this complex dialog, you will create dialog branches that handle the #turn\_on intent you defined earlier.

### Add a root node for #turn\_on

Create a dialog branch to respond to the #turn\_on intent. Start by creating the root node:

1. Click the More icon More options on the **#hello** node, and then select **Add node below**.
2. Start typing #turn\_on in the condition field, and then select it from the list. This condition is triggered by any input that matches the #turn\_on intent.
3. Do not enter a response in this node. Click Close to close the node edit view.

### Scenarios

The dialog needs to determine which appliance the user wants to turn on. To handle this, create multiple responses based on additional conditions.

There are three possible scenarios, based on the intents and entities that you defined:

**Scenario 1**: The user wants to turn on the music, in which case the bot must ask for the genre.

**Scenario 2**: The user wants to turn on any other valid appliance, in which case the bot echos the name of the requested appliance in a message that indicates it is being turned on.

**Scenario 3**: The user does not specify a recognizable appliance name, in which case the bot must ask for clarification.

Add nodes that check these scenario conditions in this order so the dialog evaluates the most specific condition first.

### Address Scenario 1

Add nodes that address scenario 1, which is that the user wants to turn on the music. In response, the bot must ask for the music genre.

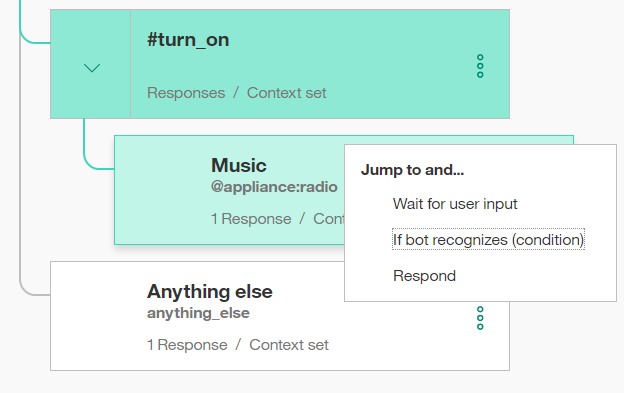
#### Add a child node that checks whether the appliance type is music

1. Click the More icon More options on the **#turn\_on** node, and select **Add child node**.
2. In the condition field, enter @appliance:radio. This condition is true if the value of the @appliance entity is radio or one of its synonyms, as defined on the Entities tab.
3. In the response field, enter What kind of music would you like to hear?
4. Name the node Music.
5. Click Close to close the node edit view.

#### Add a jump from the #turn\_on node to the Music node

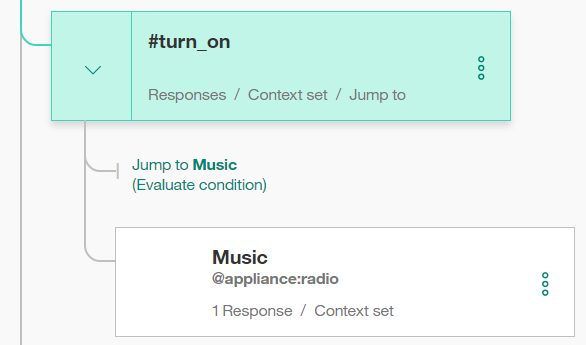
Jump directly from the #turn on node to the Music node without asking for any more user input. To do this, you can use a **Jump to** action.

1. Click the More icon More options on the **#turn\_on** node, and select **Jump to**.
2. Select the **Music** child node, and then select **If bot recognizes (condition)** to indicate that you want to process the condition of the Music node.



Note that you had to create the target node (the node to which you want to jump) before you added the **Jump to** action.

After you create the Jump to relationship, you see a new entry in the tree:



#### Add a child node that checks the music genre

Now add a node to process the type of music that the user requests.

1. Click the More icon More options on the **Music** node, and select **Add child node**. This child node is evaluated only after the user has responded to the question about the type of music they want to hear. Because we need a user input before this node, there is no need to use a **Jump to** action.
2. Add @genre to the condition field. This condition is true whenever a valid value for the @genre entity is detected.
3. Enter OK! Playing @genre. as the response. This response reiterates the genre value that the user provides.

#### Add a node that handles unrecognized genre types in user responses

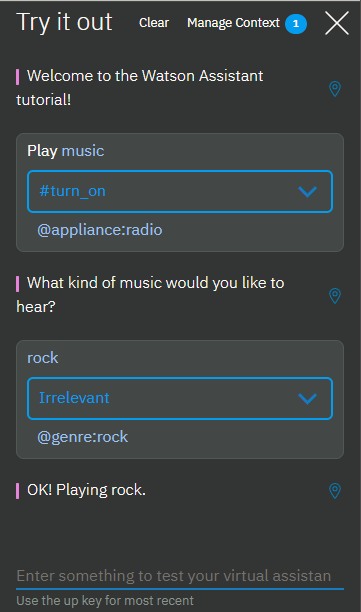
Add a node to respond when the user does not specify a recognized value for @genre.

1. Click the More icon More options on the @genre node, and select **Add node below** to create a peer node.
2. Enter true in the condition field. The true condition is a special condition. It specifies that if the dialog flow reaches this node, it should always evaluate as true. (If the user specifies a valid @genre value, this node will never be reached.)
3. Enter I'm sorry, I don't understand. I can play classical, rhythm and blues, or rock music. as the response.

That takes care of all the cases where the user asks to turn on the music.

#### Test the dialog for music

1. Select the Ask Watson icon to open the chat pane.
2. Type Play music. The bot recognizes the #turn\_on intent and the @appliance:music entity, and it responds by asking for a musical genre.
3. Type a valid @genre value (for example, rock). The bot recognizes the @genre entity and responds appropriately.



1. Type Play music again, but this time specify an invalid response for the genre. The bot responds that it does not understand.

### Address Scenario 2

We will add nodes that address scenario 2, which is that the user wants to turn on another valid appliance. In this case, the bot echos the name of the requested appliance in a message that indicates it is being turned on.

#### Add a child node that checks for any appliance

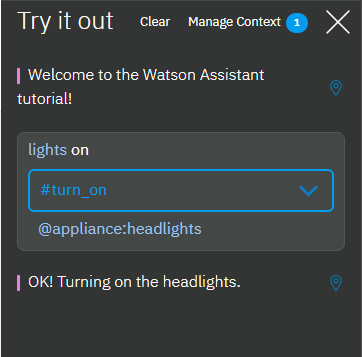
Add a node that is triggered when any other valid value for @appliance is provided by the user. For the other values of @appliance, the bot doesn't need to ask for any more input. It just returns a positive response.

1. Click the More icon More options on the **Music** node, and then select **Add node below** to create a peer node that is evaluated after the @appliance:music condition is evaluated.
2. Enter @appliance as the node condition. This condition is triggered if the user input includes any recognized value for the @appliance entity besides music.
3. Enter OK! Turning on the @appliance. as the response. This response reiterates the appliance value that the user provided.

#### Test the dialog with other appliances

1. Select the Ask Watson icon to open the chat pane.
2. Type lights on.

The bot recognizes the #turn\_on intent and the @appliance:headlights entity, and it responds with OK, turning on the headlights.



1. Type turn on the air.

The bot recognizes the #turn\_on intent and the @appliance:(air conditioning) entity, and it responds with OK, turning on the air conditioning.

1. Try variations on all of the supported commands based on the example utterances and entity synonyms you defined.

### Address Scenario 3

Now add a peer node that is triggered if the user does not specify a valid appliance type.

1. Click the More icon More options on the **@appliance** node, and then select **Add node below** to create a peer node that is evaluated after the @appliance condition is evaluated.
2. Enter true in the condition field. (If the user specifies a valid @appliance value, this node will never be reached.)
3. Enter I'm sorry, I'm not sure I understood you. I can turn on music, headlights, or air conditioning. as the response.

#### Test some more

1. Try more utterance variations to test the dialog.

If the bot fails to recognize the correct intent, you can retrain it directly from the chat window. Select the arrow next to the incorrect intent and choose the correct one from the list.

Optionally, you can review the **Car Dashboard - Sample** workspace to see this same use case fleshed out even more with a longer dialog and additional functionality.

1. Click the **Back to workspaces** button Shows Back to workspaces button in menu from the navigation menu.
2. On the **Car Dashboard - Sample** tile, click **Edit sample**.

## **Next steps**

Now that you have built and tested your workspace, you can deploy it by connecting it to a user interface. There are several ways you can do this.

### Test in Slack

You can use the test deployment tool to [deploy your workspace](https://console.bluemix.net/docs/services/conversation/test-deploy.html) as a chat bot in a Slack channel in just a few steps. This option is the fastest and easiest way to deploy your workspace for testing, but it has limitations.

### Build your own front-end application

You can use the Watson SDKs to [build your own](https://console.bluemix.net/docs/services/conversation/develop-app.html) front-end application that connects to your workspace using the Watson Assistant REST API.

### Deploy to social media or messaging channels

You can use the [Botkit framework](https://console.bluemix.net/docs/services/conversation/integrations.html) to build a bot application that you can integrate with social media and messaging channels such as Slack, Facebook, and Twilio.

# Tutorial: Adding a node with slots to a dialog

In this tutorial, you will add slots to a dialog node to collect multiple pieces of information from a user within a single node. The node you create will collect the information that is needed to make a restaurant reservation.

## **Learning objectives**

By the time you finish the tutorial, you will understand how to:

* Define the intents and entities that are needed by your dialog
* Add slots to a dialog node
* Test the node with slots

### Duration

This tutorial will take approximately 30 minutes to complete.

### Prerequisite

Before you begin, complete the [Getting Started tutorial](https://console.bluemix.net/docs/services/conversation/getting-started.html). You will use the Watson Assistant tutorial workspace that you created, and add nodes to the simple dialog that you built as part of the getting started exercise.

**Note**: You can also start with a new workspace if you want. Just be sure to create the workspace before you begin this tutorial.

## **Step 1: Add intents and examples**

Add an intent on the Intents tab. An intent is the purpose or goal that is expressed in user input. You will add a #reservation intent that recognizes user input that indicates that the user wants to make a restaurant reservation.

1. On the Intents page of the tutorial workspace, click **Add intent**.
2. Add the following intent name, and then click **Create intent**:
3. reservation

The #reservation intent is added. A number sign (#) is prepended to the intent name to label it as an intent. This naming convention helps you and others recognize the intent as an intent. It has no example user utterances associated with it yet.

1. In the **Add user examples** field, type the following utterance, and then click **Add example**:
2. i'd like to make a reservation
3. Add these additional examples to help Watson recognize the #reservation intent.
4. I want to reserve a table for dinner
5. Can 3 of us get a table for lunch?
6. do you have openings for next Wednesday at 7?
7. Is there availability for 4 on Tuesday night?
8. i'd like to come in for brunch tomorrow
9. can i reserve a table?
10. Click the **Close** Close arrow icon to finish adding the #reservation intent and its example utterances.

## **Step 2: Add entities**

An entity definition includes a set of entity values that represent vocabulary that is often used in the context of a given intent. By defining entities, you can help the service identify references in the user input that are related to intents of interest. In this step, you will enable system entities that can recognize references to time, date, and numbers.

1. Click **Entities** to open the Entities page.
2. Enable system entities that can recognize date, time, and number references in user input. Click the **System entities** tab, and then turn on these entities:
   * @sys-time
   * @sys-date
   * @sys-number

You have successfully enabled the @sys-date, @sys-time, and @sys-number system entities. Now you can use them in your dialog.

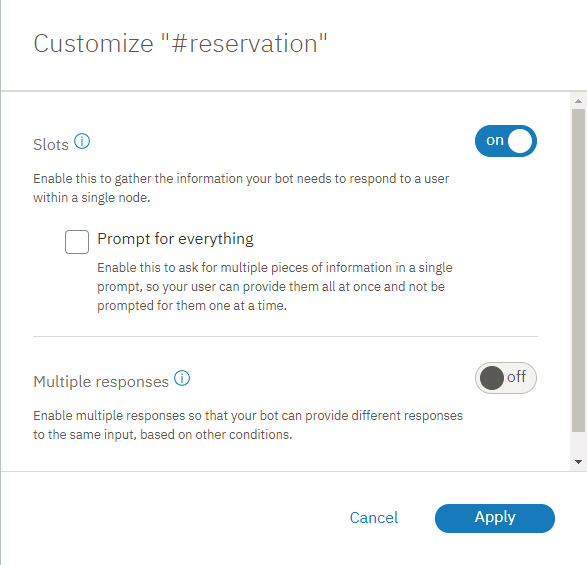
## **Step 3: Add a dialog node with slots**

A dialog node represents the start of a thread of dialog between the service and the user. It contains a condition that must be met for the node to be processed by the service. At a minimum, it also contains a response. For example, a node condition might look for the #hello intent in user input, and respond with, Hi. How can I help you? This example is the simplest form of a dialog node, one that contains a single condition and a single response. You can define complex dialogs by adding conditional responses to a single node, adding child nodes that prolong the exchange with the user, and much more. (If you want to learn more about complex dialogs, you can complete the [Building a complex dialog](https://console.bluemix.net/docs/services/conversation/tutorial.html) tutorial.)

The node that you will add in this step is one that contains slots. Slots provide a structured format through which you can ask for and save multiple pieces of information from a user within a single node. They are most useful when you have a specific task in mind and need key pieces of information from the user before you can perform it. See [Gathering information with slots](https://console.bluemix.net/docs/services/conversation/dialog-slots.html) for more information.

The node you add will collect the information required to make a reservation at a restaurant.

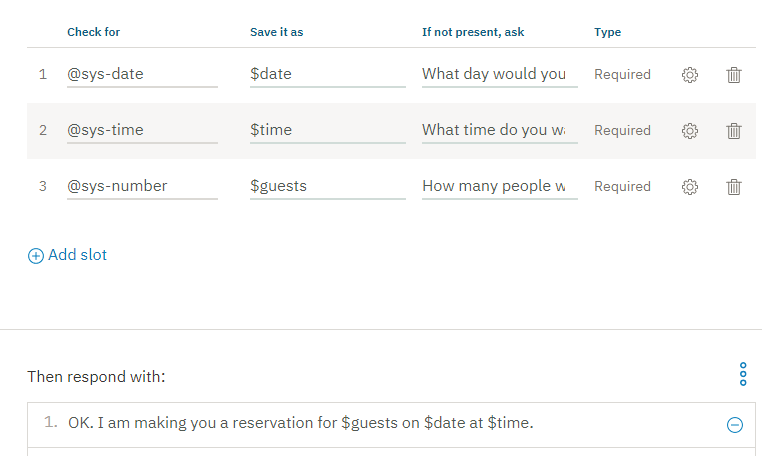
1. Click the **Dialogs** tab to open the dialog tree.
2. Click the More icon More options on the **#greeting** node, and then select **Add node below**.
3. Start typing #reservation in the condition field, and then select it from the list. This node will be evaluated if the user input matches the #reservation intent.
4. Click **Customize**, click the **Slots** toggle to turn it **on**, and then click **Apply**.



1. Define the following slots:

|  |  |  |
| --- | --- | --- |
| **Check for** | **Save it as** | **If not present, ask** |
| @sys-date | $date | What day would you like to come in? |
| @sys-time | $time | What time do you want the reservation to be made for? |
| @sys-number | $guests | How many people will be dining? |
| Slot details | | |

1. As the response, specify OK. I am making you a reservation for $guests on $date at $time.



1. Click Close to close the node edit view.

## **Step 4: Test the dialog**

1. Select the Ask Watson icon to open the chat pane.
2. Type i want to make a reservation.

The bot recognizes the #reservation intent, and it responds with the prompt for the first slot, What day would you like to come in?.

1. Type Friday.

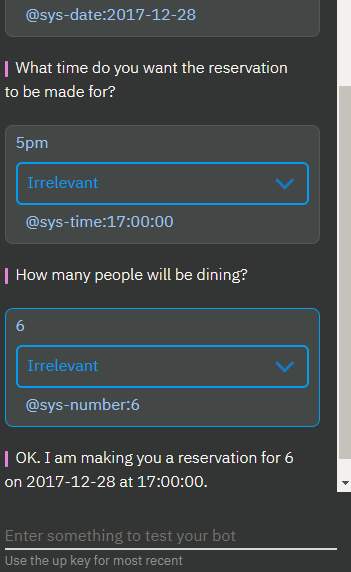
The bot recognizes the value, and uses it to fill the $date context variable for the first slot. It then shows the prompt for the next slot, What time do you want the reservation to be made for?

1. Type 5pm.

The bot recognizes the value, and uses it to fill the $time context variable for the second slot. It then shows the prompt for the next slot, How many people will be dining?

1. Type 6.

The bot recognizes the value, and uses it to fill the $guests context variable for the third slot. Now that all of the slots are filled, it shows the node response, OK. I am making you a reservation for 6 on 2017-12-29 at 17:00:00.



It worked! Congratulations. You have successfully created a node with slots.

## **Summary**

In this tutorial you created a node with slots that can capture the information necessary to reserve a table at a restaurant.

## **Next steps**

Improve the experience of users who interact with the node. Complete the follow-on tutorial, [Improving a node with slots](https://console.bluemix.net/docs/services/conversation/tutorial-slots-complex.html). It covers simple improvements, such as how to reformat the date (2017-12-28) and time (17:00:00) values that are returned by the system. It also covers more complex tasks, such as what to do if the user does not provide the type of value that your dialog expects for a slot.

In this tutorial, you will add slots to a dialog node to collect multiple pieces of information from a user within a single node. The node you create will collect the information that is needed to make a restaurant reservation.

## **Learning objectives**

By the time you finish the tutorial, you will understand how to:

* Define the intents and entities that are needed by your dialog
* Add slots to a dialog node
* Test the node with slots

### Duration

This tutorial will take approximately 30 minutes to complete.

### Prerequisite

Before you begin, complete the [Getting Started tutorial](https://console.bluemix.net/docs/services/conversation/getting-started.html). You will use the Watson Assistant tutorial workspace that you created, and add nodes to the simple dialog that you built as part of the getting started exercise.

**Note**: You can also start with a new workspace if you want. Just be sure to create the workspace before you begin this tutorial.

## **Step 1: Add intents and examples**

Add an intent on the Intents tab. An intent is the purpose or goal that is expressed in user input. You will add a #reservation intent that recognizes user input that indicates that the user wants to make a restaurant reservation.

1. On the Intents page of the tutorial workspace, click **Add intent**.
2. Add the following intent name, and then click **Create intent**:
3. reservation

The #reservation intent is added. A number sign (#) is prepended to the intent name to label it as an intent. This naming convention helps you and others recognize the intent as an intent. It has no example user utterances associated with it yet.

1. In the **Add user examples** field, type the following utterance, and then click **Add example**:
2. i'd like to make a reservation
3. Add these additional examples to help Watson recognize the #reservation intent.
4. I want to reserve a table for dinner
5. Can 3 of us get a table for lunch?
6. do you have openings for next Wednesday at 7?
7. Is there availability for 4 on Tuesday night?
8. i'd like to come in for brunch tomorrow
9. can i reserve a table?
10. Click the **Close** Close arrow icon to finish adding the #reservation intent and its example utterances.

## **Step 2: Add entities**

An entity definition includes a set of entity values that represent vocabulary that is often used in the context of a given intent. By defining entities, you can help the service identify references in the user input that are related to intents of interest. In this step, you will enable system entities that can recognize references to time, date, and numbers.

1. Click **Entities** to open the Entities page.
2. Enable system entities that can recognize date, time, and number references in user input. Click the **System entities** tab, and then turn on these entities:
   * @sys-time
   * @sys-date
   * @sys-number

You have successfully enabled the @sys-date, @sys-time, and @sys-number system entities. Now you can use them in your dialog.

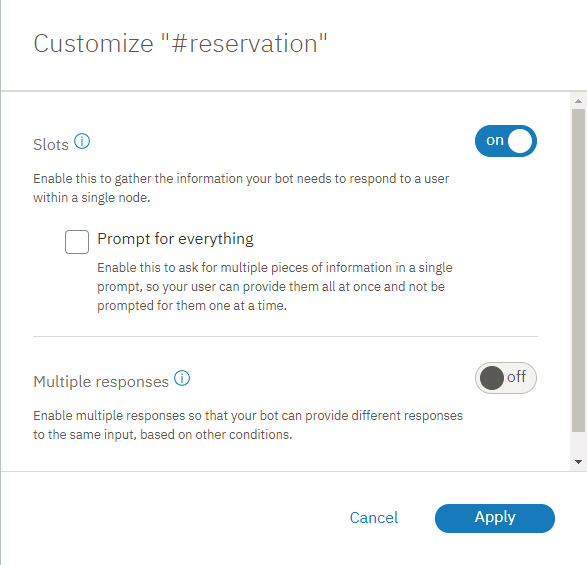
## **Step 3: Add a dialog node with slots**

A dialog node represents the start of a thread of dialog between the service and the user. It contains a condition that must be met for the node to be processed by the service. At a minimum, it also contains a response. For example, a node condition might look for the #hello intent in user input, and respond with, Hi. How can I help you? This example is the simplest form of a dialog node, one that contains a single condition and a single response. You can define complex dialogs by adding conditional responses to a single node, adding child nodes that prolong the exchange with the user, and much more. (If you want to learn more about complex dialogs, you can complete the [Building a complex dialog](https://console.bluemix.net/docs/services/conversation/tutorial.html) tutorial.)

The node that you will add in this step is one that contains slots. Slots provide a structured format through which you can ask for and save multiple pieces of information from a user within a single node. They are most useful when you have a specific task in mind and need key pieces of information from the user before you can perform it. See [Gathering information with slots](https://console.bluemix.net/docs/services/conversation/dialog-slots.html) for more information.

The node you add will collect the information required to make a reservation at a restaurant.

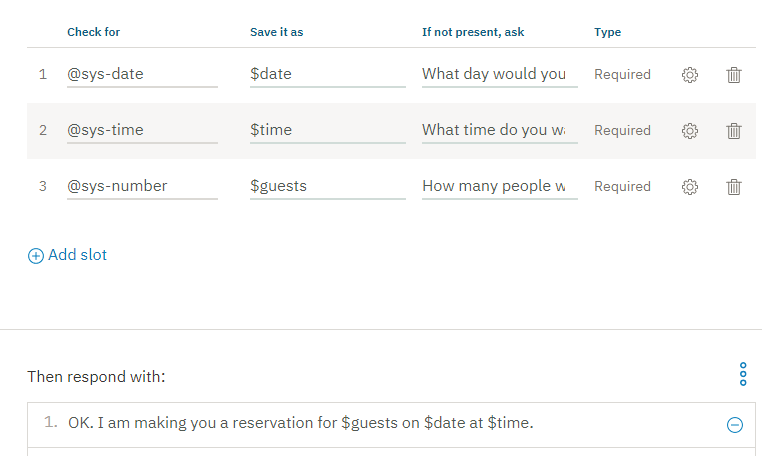
1. Click the **Dialogs** tab to open the dialog tree.
2. Click the More icon More options on the **#greeting** node, and then select **Add node below**.
3. Start typing #reservation in the condition field, and then select it from the list. This node will be evaluated if the user input matches the #reservation intent.
4. Click **Customize**, click the **Slots** toggle to turn it **on**, and then click **Apply**.



1. Define the following slots:

|  |  |  |
| --- | --- | --- |
| **Check for** | **Save it as** | **If not present, ask** |
| @sys-date | $date | What day would you like to come in? |
| @sys-time | $time | What time do you want the reservation to be made for? |
| @sys-number | $guests | How many people will be dining? |
| Slot details | | |

1. As the response, specify OK. I am making you a reservation for $guests on $date at $time.



1. Click Close to close the node edit view.

## **Step 4: Test the dialog**

1. Select the Ask Watson icon to open the chat pane.
2. Type i want to make a reservation.

The bot recognizes the #reservation intent, and it responds with the prompt for the first slot, What day would you like to come in?.

1. Type Friday.

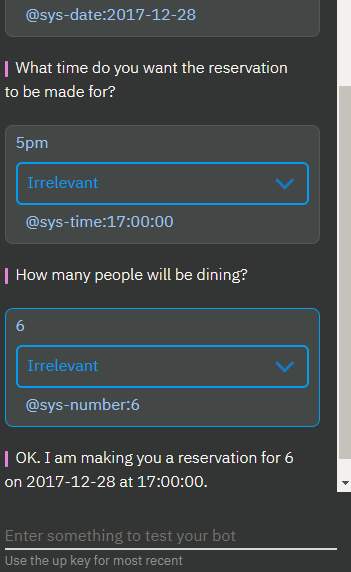
The bot recognizes the value, and uses it to fill the $date context variable for the first slot. It then shows the prompt for the next slot, What time do you want the reservation to be made for?

1. Type 5pm.

The bot recognizes the value, and uses it to fill the $time context variable for the second slot. It then shows the prompt for the next slot, How many people will be dining?

1. Type 6.

The bot recognizes the value, and uses it to fill the $guests context variable for the third slot. Now that all of the slots are filled, it shows the node response, OK. I am making you a reservation for 6 on 2017-12-29 at 17:00:00.



It worked! Congratulations. You have successfully created a node with slots.

## **Summary**

In this tutorial you created a node with slots that can capture the information necessary to reserve a table at a restaurant.

## **Next steps**

Improve the experience of users who interact with the node. Complete the follow-on tutorial, [Improving a node with slots](https://console.bluemix.net/docs/services/conversation/tutorial-slots-complex.html). It covers simple improvements, such as how to reformat the date (2017-12-28) and time (17:00:00) values that are returned by the system. It also covers more complex tasks, such as what to do if the user does not provide the type of value that your dialog expects for a slot.

# Tutorial: Improving a node with slots

In this tutorial, you will enhance a simple node with slots that collects the information necessary to make a restaurant reservation.

## **Learning objectives**

By the time you finish the tutorial, you will understand how to:

* Test a node with slots
* Add slot response conditions that address common user interactions
* Anticipate and address unrelated user input
* Handle unexpected user responses

### Duration

This tutorial will take approximately 2 to 3 hours to complete.

### Prerequisite

Before you begin, complete the [Adding a node with slots to a dialog](https://console.bluemix.net/docs/services/conversation/tutorial-slots.html). You must complete the first slots tutorial before you begin this one because you will build on the node with slots that you create in the first tutorial.

## **Step 1: Improve the format of the responses**

When the date and time system entity values are saved, they are converted into a standardized format. This standardized format is useful for performing calculations on the values, but you might not want to expose this reformatting to users. In this step, you will reformat the date (2017-12-29) and time (17:00:00) values that are referenced by the dialog.

1. To reformat the $date context variable value, click the **Edit response** Edit response icon for the @sys-date slot.
2. From the **More** More icon menu at the top of the page, select **Open JSON editor**, and then edit the JSON that defines the context variable. Add a method that reformats the date so that it converts the 2017-12-29 value into a full day of the week, followed by the full month and day. Edit the JSON as follows:
3. {
4. "context": {
5. "date": "<? @sys-date.reformatDateTime('EEEE, MMMM d') ?>"
6. }
7. }

The EEEE indicates that you want to spell out the day of the week. If you use 3 Es (EEE), the day of the week will be shortened to Fri instead of Friday, for example. The MMMM indicates that you want to spell out the month. Again, if you use only 3 Ms (MMM), the month is shortened to Dec instead of December.

1. Click **Save**.
2. To change the format in which the time value is stored in the $time context variable to use the hour, minutes and indicate AM or PM, click the **Edit response** Edit response icon for the @sys-time slot.
3. From the **More** More icon menu at the top of the page, select **Open JSON editor**, and then edit the JSON that defines the context variable so that it reads as follows:
4. {
5. "context": {
6. "time": "<? @sys-time.reformatDateTime('h:mm a') ?>"
7. }
8. }
9. Click **Save**.
10. Test the node again. Open the "Try it out" pane, and click **Clear** to delete the slot context variable values that you specified when you tested the node with slots earlier. To see the impact of the changes you made, use the following script:

|  |  |
| --- | --- |
| **Speaker** | **Utterance** |
| You | i want to make a reservation |
| Watson | What day would you like to come in? |
| You | Friday |
| Watson | What time do you want the reservation to be made for? |
| You | 5pm |
| Watson | How many people will be dining? |
| You | 6 |
| Script details | |

1. This time Watson responds with, OK. I am making you a reservation for 6 on Friday, December 29 at 5:00 PM.

You have successfully improved the format that the dialog uses when it references context variable values in its responses. The dialog now uses Friday, December 29 instead of the more technical, 2017-12-29. And it uses5:00 PM instead of 17:00:00. To learn about other SpEL methods you can use with date and time values, see [Methods to process values](https://console.bluemix.net/docs/services/conversation/dialog-methods.html#date-time).

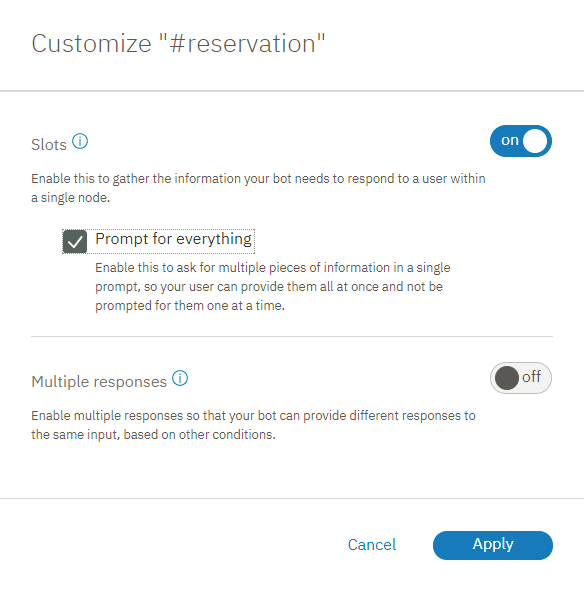
## **Step 2: Ask for everything at once**

Now that you have tested the dialog more than once, you might have noticed that it can be annoying to have to answer one slot prompt at a time. To prevent users from having to provide one piece of information at a time, you can ask for every piece of information that you need up front. Doing so gives the user a chance to provide all or some of the information in a single input.

The node with slots is designed to find and save any and all slot values that the user provides while the current node is being processed. You can help users to take advantage of the design by letting them know what values to specify.

In this step, you will learn how to prompt for everything at once.

1. From the main node with slots, click **Customize**.
2. Select the **Prompt for everything** checkbox to enable the intial prompt, and then click **Apply**.

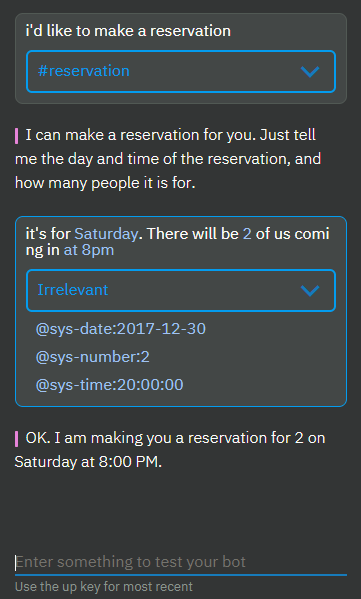


1. Back in the node edit view, scroll down to the newly-added **If no slots are pre-filled, ask this first** field. Add the following initial prompt for the node, I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for.
2. Click Close to close the node edit view.
3. Test this change from the "Try it out" pane. Open the pane, and then click **Clear** to nullify the slot context variable values from the previous test.
4. Enter i'd like to make a reservation.

The dialog now responds with, I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for.

1. Enter, it's for Saturday. There will be 2 of us coming in at 8pm

The dialog responds with, OK. I am making you a reservation for 2 on Saturday at 8:00 PM.



**Note**: If the user provides any one of the slot values in their initial input, then the prompt that asks for everything is not displayed. For example, the initial input from the user might be, I want to make a reservation for this Friday night. In this case, the initial prompt is skipped because you do not want to ask for information that the user already provided - the date (Friday), in this example. The dialog shows the prompt for the next empty slot instead.

## **Step 3: Validate user input**

So far, we have assumed that the user will provide the appropriate value types for the slots. That is not always the case in reality. You can account for times when users might provide an invalid value by adding conditional responses to slots. In this step, you will use conditional slot responses to perform the following tasks:

* Ensure that the date requested is not in the past.
* Check whether a requested reservation time falls within the seating time window.
* Confirm the user's input.
* Indicate that you are replacing one value with another.

To validate user input, complete the following steps:

1. From the edit view of the node with slots, click the **Edit slot** Edit slot icon for the @sys-date slot.
2. From the **Options** More icon menu in the Configure slot 1 header, select **Enable conditional responses**.
3. In the **Found** section, add a conditional response by clicking the **Edit response** Edit response icon.
4. Add the following condition and response to check whether the date that the user specifies falls before today:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| @sys-date.before(now()) | You cannot make a reservation for a day in the past. | Clear slot and prompt again |
| Slot 1 conditional response 1 details | | |

1. Add a second conditional response that is displayed if the user provides a valid date. This type of simple confirmation lets the user know that her response was understood.

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| true | $date it is | Move on |
| Slot 1 conditional response 2 details | | |

1. From the edit view of the node with slots, click the **Edit slot** Edit slot icon for the @sys-time slot.
2. From the **Options** More icon menu in the Configure slot 2 header, select **Enable conditional responses**.
3. In the **Found** section, add a conditional response by clicking the **Edit response** Edit response icon.
4. Add the following conditions and responses to check whether the time that the user specifies falls within the allowed time window:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| @sys-time.after('21:00:00') | Our last seating is at 9 PM. | Clear slot and prompt again |
| @sys-time.before('09:00:00') | Our first seating is at 9 AM. | Clear slot and prompt again |
| Slot 2 conditional response details | | |

1. Add a third conditional response that is displayed if the user provides a valid time that falls within the window. This type of simple confirmation lets the user know that her response was understood.

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| true | Ok, the reservation is for $time. | Move on |
| Slot 2 conditional response 3 details | | |

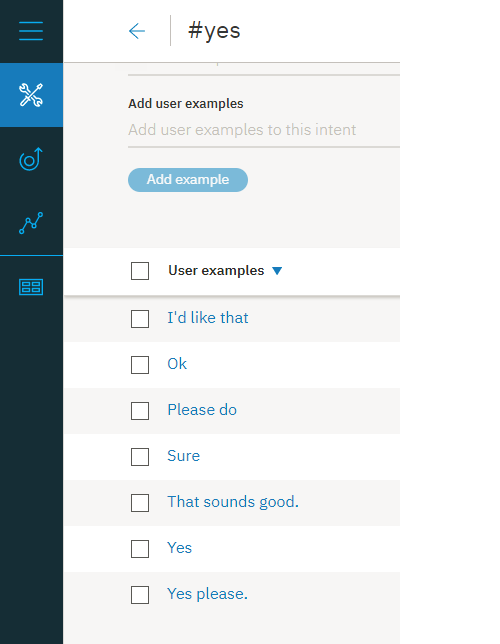
1. Edit the @sys-number slot to anticipate and address the case when the user changes the number of guests. If, at any point while the node with slots is being processed, the user changes a slot value, the corresponding slot context variable value is updated. However, it can be useful to let the user know that the value is being replaced, both to give clear feedback to the user and to give the user a chance to rectify it if the change was not what she intended. From the edit view of the node with slots, click the **Edit slot** Edit slot icon for the @sys-number slot.
2. From the **Options** More icon menu in the Configure slot 3 header, select **Enable conditional responses**.
3. In the **Found** section, add a conditional response by clicking the Edit response icon, and then add the following condition and response:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| (event.previous\_value != null) && (event.previous\_value != event.current\_value) | Ok, updating the number of guests from <? event.previous\_value ?> to <? event.current\_value ?>. | Move on |
| true | Ok. The reservation is for $guests guests. | Move on |
| Slot 3 conditional response details | | |

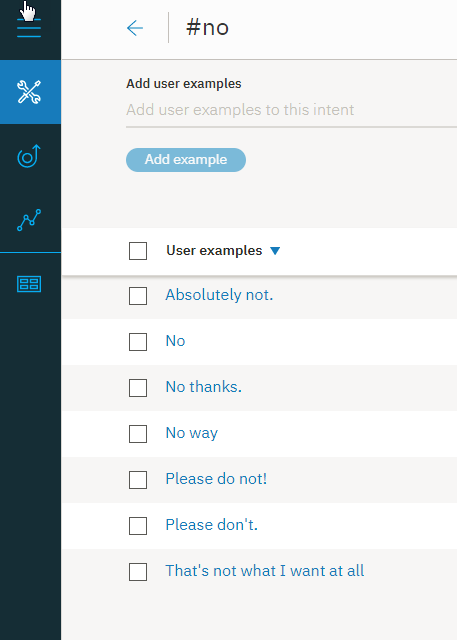
## **Step 4: Add a confirmation slot**

You might want to design your dialog to call an external reservation system and actually book a reservation for the user in the system. Before your application takes this action, you probably want to confirm with the user that the dialog has understood the details of the reservation correctly. You can do so by adding a confirmation slot to the node.

1. The confirmation slot will expect a Yes or No answer from the user. You must teach the dialog to be able to recognize a Yes or No intent in the user input first.
2. Click the **Intents** tab to return to the Intents page. Add the following intents and example utterances.
3. #yes
4. Yes
5. Sure
6. I'd like that
7. Please do
8. Yes please.
9. Ok
10. That sounds good.



1. #no
2. No
3. No thanks.
4. Please don't.
5. Please do not!
6. That's not what I want at all
7. Absolutely not.
8. No way



1. Return to the **Dialog** tab, and then click to edit the node with slots. Click **Add slot** to add a fourth slot, and then specify the following values for it:

|  |  |  |
| --- | --- | --- |
| **Check for** | **Save it as** | **If not present, ask** |
| (#yes || #no) && slot\_in\_focus | $confirmation | I'm going to reserve you a table for $guests on $date at $time. Should I go ahead? |
| Confirmation slot details | | |

1. This condition checks for either answer. You will specify what happens next depending on whether the user answer Yes or No by using conditional slot responses. The slot\_in\_focus property forces the scope of this condition to apply to the current slot only. This setting prevents random statements that could match against a #yes or #no intent that the user might make from triggering this slot.
2. For example, the user might be answering the number of guests slot, and say something like, Yes, there will be 5 of us. You do not want the Yes included in this response to accidentally fill the confirmation slot. By adding the slot\_in\_focus property to the condition, a yes or no indicated by the user is applied to this slot only when the user is answering the prompt for this slot specifically.
3. Click the **Edit slot** Edit slot icon. From the **Options** More icon menu in the Configure slot 4 header, select **Enable conditional responses**.
4. In the **Found** prompt, add a condition that checks for a No response (#no). Use the response, Alright. Let's start over. I'll try to keep up this time. Otherwise, you can assume the user confirmed the reservation details and proceed with making the reservation.

When the #no intent is found, you also must reset the context variables that you saved earlier to null, so you can ask for the information again. You can reset the context variable values by using the JSON editor. Click the **Edit response** Edit response icon for the conditional response you just added. From the **Options** Advanced response menu, click **Open JSON editor**. Add a context block that sets the slot context variables to null, as shown below.

{

"conditions": "#no",

"output":{

"text": {

"values": [

"Alright. Let's start over. I'll try to keep up this time."

]

}

},

"context":{

"date": null,

"time": null,

"guests": null

}

}

1. Click **Back**, and then click **Save**.
2. Click the **Edit slot** Edit slot icon for the confirmation slot again. In the **Not found** prompt, clarify that you are expecting the user to provide a Yes or No answer. Add a response with the following values.

|  |  |
| --- | --- |
| **Condition** | **Response** |
| true | Respond with Yes to indicate that you want the reservation to be made as-is, or No to indicate that you do not. |
| Not found response details | |

1. Click **Save**.
2. Now that you have confirmation responses for slot values, and you ask for everything at once, you might notice that the individual slot responses are displayed before the confirmation slot response is displayed, which can appear repetitive to users. Edit the slot found responses to prevent them from being displayed under certain conditions.
3. Replace the true condition that is specified in the JSON snippet for the last conditional response in the @sys-date slot with !($time && $guests). For example:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| !($time && $guests) | $date it is | Move on |
| Slot 1 conditional response 2 details | | |

1. Replace the true condition that is specified in the JSON snippet for the last conditional response in the @sys-time slot with !($date && $guests). For example:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| !($date && $guests) | Ok, the reservation is for $time. | Move on |
| Slot 2 conditional response 3 details | | |

1. Replace the true condition that is specified in the JSON snippet for the last conditional response in the @sys-number slot with !($date && $time). For example:

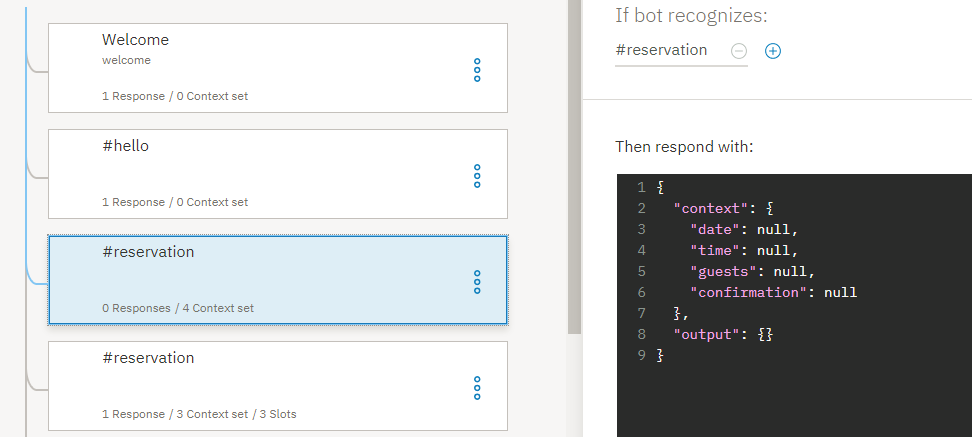
|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| !($date && $time) | Ok. The reservation is for $guests guests. | Move on |
| Slot 3 conditional response 2 details | | |

If you add more slots later, you must edit these conditions to account for the associated context variables for the additional slots. If you do not include a confirmation slot, you can specify !all\_slots\_filled only, and it would remain valid no matter how many slots you add later.

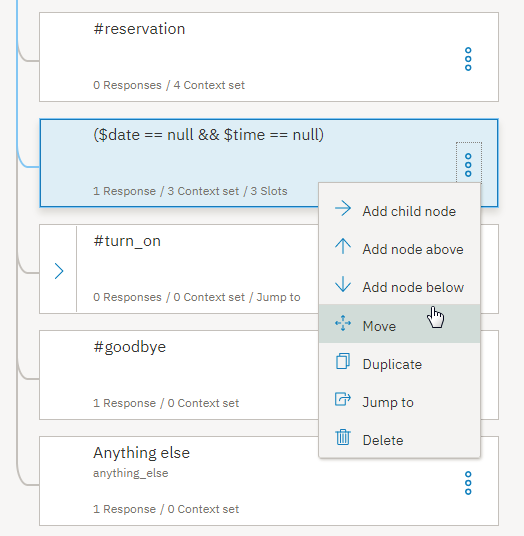
## **Step 5: Reset the slot context variable values**

You might have noticed that before each test, you must clear the context variable values that were created during the previous test. You must do so because the node with slots only prompts users for information that it considers to be missing. If the slot context variables are all filled with valid values, no prompts are displayed. The same is true for the dialog at run time. You must build into the dialog a mechanism by which you reset the slot context variables to null so that the slots can be filled anew by the next user. To do so, you are going to add a parent node to the node with slots that sets the context variables to null.

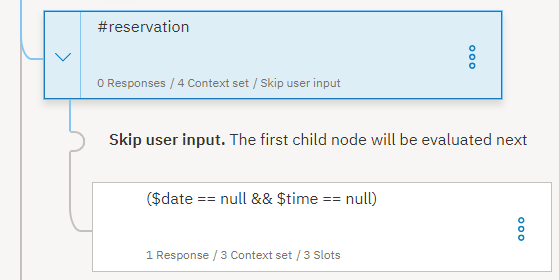
1. From the tree view of the dialog, click the **More** More icon icon on the node with slots, and then select **Add node above**.
2. Specify #reservation as the condition for the new node. (This is the same condition that is used by the node with slots, but you will change the condition for the node with slots later in this procedure.)
3. Click the **Options** More icon icon next to the node response, and then click **Open JSON editor**. Add an entry for each slot context variable that you defined in the node with slots, and set it equal to null.
4. {
5. "context": {
6. "date": null,
7. "time": null,
8. "guests": null,
9. "confirmation": null
10. },
11. "output": {}
12. }



1. Click to edit the other #reservation node, the one you created previously and to which you added the slots.
2. Change the node condition from #reservation to ($date == null && $time == null), and then close the node edit view by clicking Close.
3. Click the **More** More icon icon on the node with slots, and then select **Move**.



1. Select the #reservation node as the move-to location target, and then choose **As child node** from the menu.
2. Click to edit the #reservation node. In the And finally section, change the action from Wait for user input to **Skip user input**.

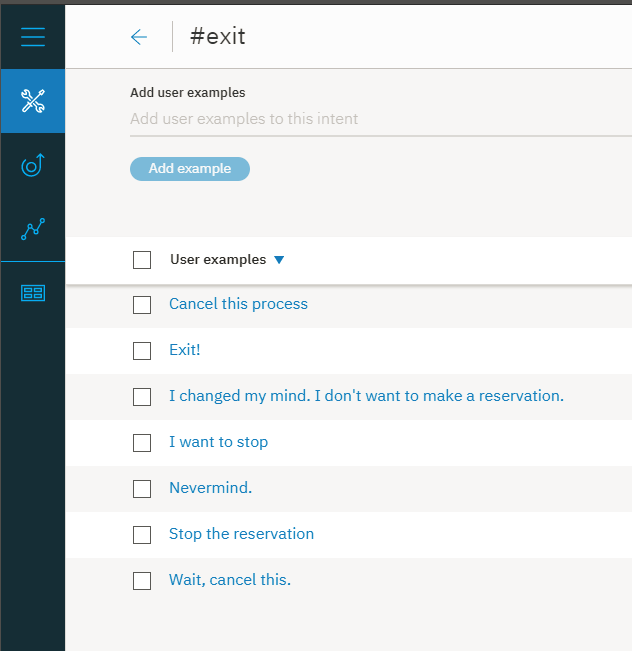


When a user input matches the #reservation intent, this node is triggered. The slot context variables are all set to null, and then the dialog jumps directly to the node with slots to process it.

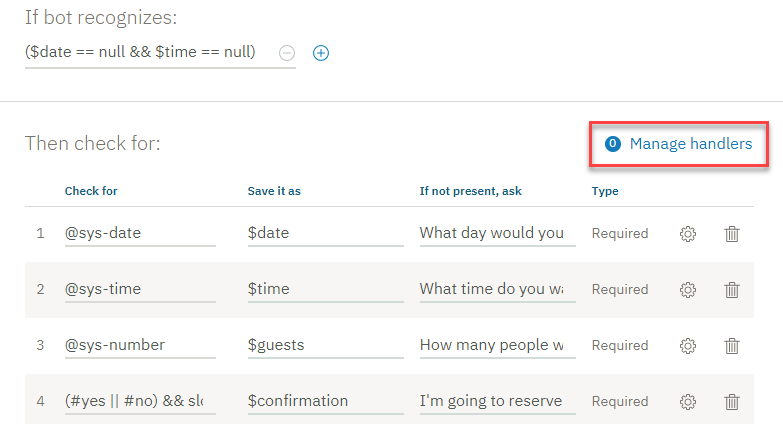
## **Step 6: Give users a way to exit the process**

Adding a node with slots is powerful because it keeps users on track with providing the information you need to give them a meaningful response or perform an action on their behalf. However, there might be times when a user is in the middle of providing reservation details, but decides to not go through with placing the reservation. You must give users a way to exit the process gracefully. You can do so by adding a slot handler that can detect a user's desire to exit the process, and exit the node without saving any values that were collected.

1. You must teach the dialog to be able to recognize an #exit intent in the user input first.
2. Click the **Intents** tab to return to the Intents page. Add the #exit intent with the following example utterances.
3. I want to stop
4. Exit!
5. Cancel this process
6. I changed my mind. I don't want to make a reservation.
7. Stop the reservation
8. Wait, cancel this.
9. Nevermind.



1. Return to the dialog by clicking the **Dialog** tab. Click to open the node with slots, and then click **Manage handlers**.

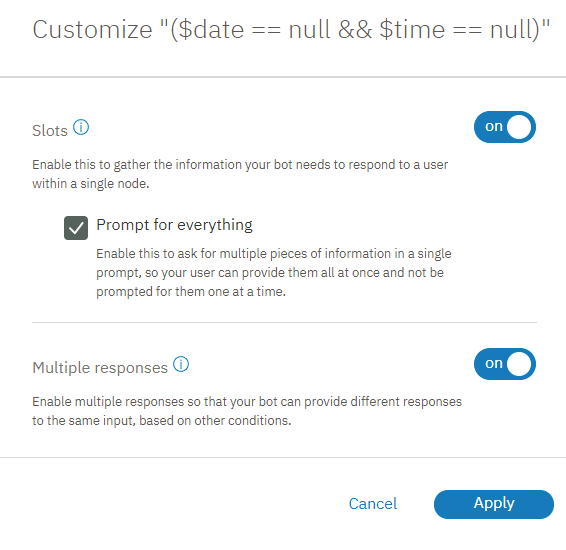


1. Add the following values to the fields.

|  |  |  |
| --- | --- | --- |
| **Condition** | **Response** | **Action** |
| #exit | Ok, we'll stop there. No reservation will be made. | Skip to response |
| Node-level handler details | | |

1. The **Skip to response** action jumps directly to the node-level response without displaying the prompts associated with any of the remaining unfilled slots.
2. Click **Back**, and then click **Save**.
3. Now, you need to edit the node-level response to make it recognize when a user wants to exit the process rather than make a reservation. Add a conditional response for the node.

From the edit view of the node with slots, click **Customize**, click the **Multiple responses** toggle to turn it **on**, and then click **Apply**.



1. Scroll down to the response section for the node with slots, and then click **Add response**.
2. Add the following values to the fields.

|  |  |
| --- | --- |
| **Condition** | **Response** |
| has\_skipped\_slots | I look forward to helping you with your next reservation. Have a good day. |
| Node-level conditional response details | |

1. The has\_skipped\_slots condition checks the properties of the slots node to see if any of the slots were skipped. The #exit handler skips all remaining slots to go directly to the node response. So, when the has\_skipped\_slots property is present, you know the #exit intent was triggered, and the dialog can display an alternate response.
2. **Note**: If you configure more than one slot to skip other slots, or configure another node-level event handler to skip slots, then you must use a different approach to check whether the #exit intent was triggered. See [Handling requests to exit a process](https://console.bluemix.net/docs/services/conversation/dialog-slots.html#slots-node-level-handler) for an alternate way to do so.
3. You want the service to check for the has\_skipped\_slots property before it displays the standard node-level response. Move the has\_skipped\_slots conditional response up so it gets processed before the original conditional response or it will never be triggered. To do so, click the response you just added, use the **up arrow** to move it up, and then click **Save**.
4. Test this change by using the following script in the "Try it out" pane.

|  |  |
| --- | --- |
| **Speaker** | **Utterance** |
| You | i want to make a reservation |
| Watson | I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for. |
| You | it's for 5 people |
| Watson | Ok. The reservation is for 5 guests. What day would you like to come in? |
| You | Nevermind |
| Watson | Ok, we'll stop there. No reservation will be made. I look forward to helping you with your next reservation. Have a good day. |
| Script details | |

## **Step 7: Apply a valid value if the user fails to provide one after several attempts**

In some cases, a user might not understand what you are asking for. They might respond again and again with the wrong types of values. To plan for this possibility, you can add a counter to the slot, and after 3 failed attempts by the user to provide a valid value, you can apply a value to the slot on the user's behalf and move on.

For the $time information, you will define a follow-up statement that is displayed when the user does not provide a valid time.

1. Create a context variable that can keep track of how many times the user provides a value that does not match the value type that the slot expects. You want the context variable to be initialized and set to 0 before the node with slots is processed, so you will add it to the parent #reservation node.
2. Click to edit the #reservation node. Open the JSON editor associated with the node response, by clicking the **Options** More icon icon in the response section, and choosing **Open JSON editor**. Add a context variable called counter to the bottom of the existing "context" block, below the confirmation variable. Set the counter variable equal to 0.
3. {
4. "context": {
5. "date": null,
6. "time": null,
7. "guests": null,
8. "confirmation": null,
9. "counter": 0
10. },
11. "output": {}
12. }
13. From the tree view, expand the #reservation node, and then click to edit the node with slots.
14. Click the **Edit slot** Edit slot icon for the @sys-time slot.
15. From the **Options** More icon menu in the Configure slot 2 header, select **Enable conditional responses**.
16. In the **Not found** section, add a conditional response.

|  |  |
| --- | --- |
| **Condition** | **Response** |
| true | Please specify the time that you want to eat. The restaurant seats people between 9AM and 9PM. |
| Not found response details | |

1. Add a 1 to the counter variable each time this response is triggered. Remember, this response is only triggered when the user does not provide a valid time value. Click the **Edit response** Edit response icon.
2. Click the **Options** More icon icon, and select **Open JSON editor**. Add the following context variable definition.
3. {
4. "conditions": "true",
5. "output": {
6. "text": {
7. "values": [
8. "Please specify the time that you want to eat.
9. The restaurant seats people between 9AM and 9PM."
10. ]
11. }
12. },
13. "context": {
14. "counter": "<? context['counter'] + 1 ?>"
15. }
16. }

This expression adds a 1 to the current counter tally.

1. Click **Back**, and then click **Save**.
2. Reopen the @sys-time slot by clicking the **Edit slot** Edit slot icon.

You will add a second conditional response to the **Not found** section that checks whether the counter is greater than 1, which indicates that the user has provided an invalid response 3 times previously. In this case, the dialog assigns the time value on the user's behalf to the popular dinner reservation time of 8 PM. Don't worry; the user will have a chance to change the time value when the confirmation slot is triggered. Click **Add a response**.

1. Add the following condition and response.

|  |  |
| --- | --- |
| **Condition** | **Response** |
| $counter > 1 | You seem to be having trouble choosing a time. I will make the reservation at 8PM for you. |
| Not found response details | |

1. You must set the $time variable to 8PM, so click the **Edit response** Edit response icon. Select **Open JSON editor**, add the following context variable definition, and then click **Back**.
2. {
3. "conditions": "$counter > 1",
4. "output": {
5. "text": {
6. "values": [
7. "You seem to be having trouble choosing a time.
8. I will make the reservation at 8 PM for you."
9. ]
10. }
11. },
12. "context": {
13. "time": "<? '20:00:00'.reformatDateTime('h:mm a') ?>"
14. }
15. }
16. The conditional response that you just added has a more precise condition than the true condition that is used by the first conditional response. You must move this response so it comes before the original conditional response or it will never be triggered. Click the response you just added, and use the up arrow to move it up, and then click **Save**.
17. Test your changes by using the following script.

| **Speaker** | **Utterance** |
| --- | --- |
| You | i want to make a reservation |
| Watson | I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for. |
| You | tomorrow |
| Watson | Friday, December 29 it is. What time do you want the reservation to be made for? |
| You | orange |
| Watson | Please specify the time that you want to eat. The restaurant seats people between 9AM and 9PM. |
| You | pink |
| Watson | Please specify the time that you want to eat. The restaurant seats people between 9AM and 9PM. |
| You | purple |
| Watson | You seem to be having trouble choosing a time. I will make the reservation at 8PM for you. How many people will be dining? |

## **Step 8: Connect to an external service**

Now that your dialog can collect and confirm a user's reservation details, you can call an external service to actually reserve a table in the restaurant's system or through a multi-restaurant online reservations service. See [Making programmatic calls from a dialog node](https://console.bluemix.net/docs/services/conversation/dialog-actions.html) for more details.

In the logic that calls the reservation service, be sure to check for has\_skipped\_slots and do not continue with the reservation if it is present.

### Summary

In this tutorial you tested a node with slots and made changes that optimize how it interacts with real users. For more information about this subject, see [Gathering information with slots](https://console.bluemix.net/docs/services/conversation/dialog-slots.html).

## **Next steps**

Deploy your workspace by connecting it to a user interface. There are several ways you can do this. See [Deployment overview](https://console.bluemix.net/docs/services/conversation/deploy.html) for more details.

Tutorial: Understanding digressions

In this tutorial, you will see firsthand how digressions work.

## **Learning objectives**

By the time you finish the tutorial, you will understand how:

* digressions are designed to work
* digression settings impact the flow of the dialog
* to test digression settings for a dialog

### Duration

This tutorial will take approximately 20 minutes to complete.

### Prerequisite

If you do not have a Watson Assistant instance, complete the **Before you begin** step from the [Getting Started tutorial](https://console.bluemix.net/docs/services/conversation/getting-started.html#prerequisites) to create one.

## **Step 1: Import the Digressions showcase workspace**

First you will need to import the Digression showcase workspace into your Watson Assistant instance.

1. Download the [digression-showcase.json](https://github.com/watson-developer-cloud/community/raw/master/watson-assistant/digression-showcase.json) file.
2. In your Watson Assistant instance, click the Import workspace icon.
3. Click **Choose a file**, and then select the **digression-showcase.json** file that you downloaded earlier.
4. Click **Import** to finish importing the workspace.

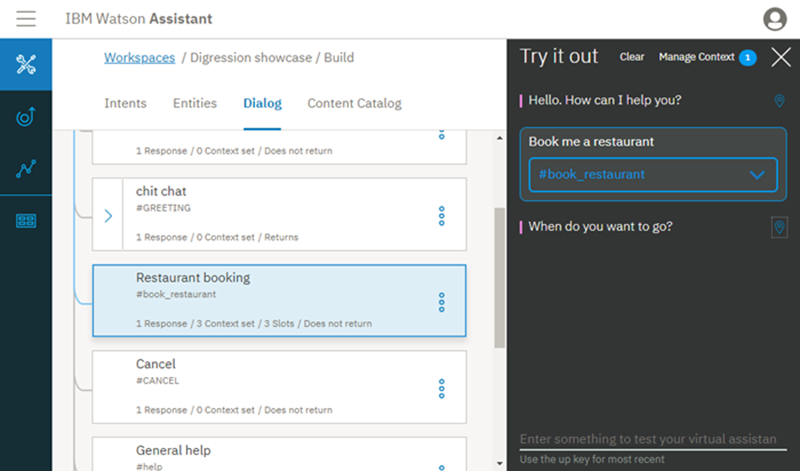
## **Step 2: Temporarily digressing away from dialog**

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

1. Click **Dialog** to switch from the page with intents to a view of the dialog tree.
2. Click the Try it icon to open the "Try it out" pane.
3. Type Book me a restaurant into the text field.

The service responds with a prompt for the day to reserve, When do you want to go?

1. Click the **Location** Location icon next to the response to highlight the node that triggered the response, the **Restaurant booking** node, in the dialog tree.

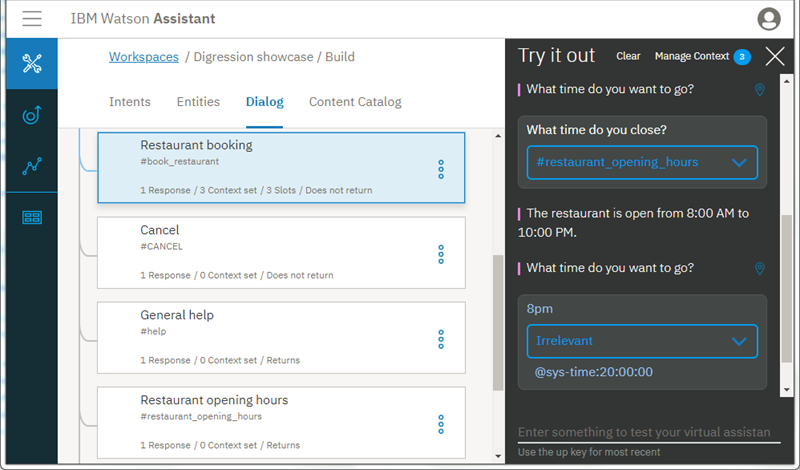


1. Type Tomorrow.

The service responds with a prompt for the time to reserve, What time do you want to go?

1. You do not know when the restaurant closes, so you ask, What time do you close?

The bot digresses away from the restaurant booking node to process the **Restaurant opening hours** node. It responds with, The restaurant is open from 8:00 AM to 10:00 PM. The service then returns to the restaurant booking node, and prompts you again for the reservation time.



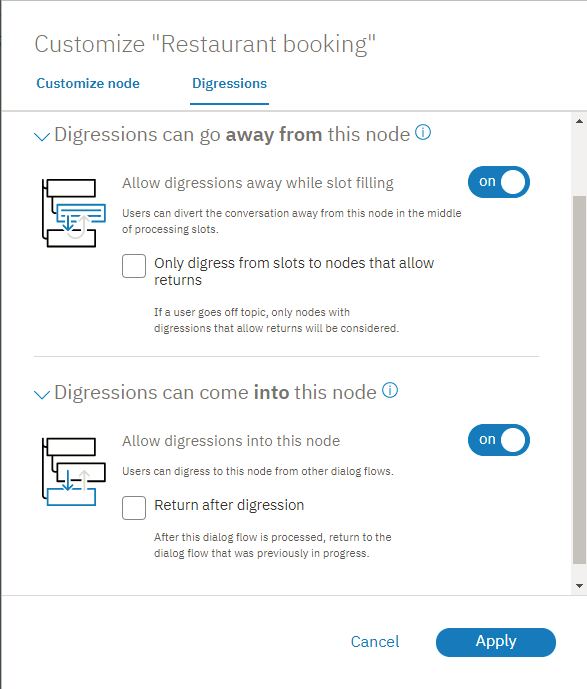
1. **Optional**: To complete the dialog flow, type 8pm for the reservation time and 2 for the number of guests.

Congratulations! You have successfully digressed away from and returned to a dialog flow.

## **Step 3: Disabling slot digressions**

In this step, you will edit the digression settting for the restaurant booking node to prevent users from digressing away from it, and see how the setting change impacts the dialog flow.

1. Let's look at the current digression settings for the **Restaurant booking** node. Click the node to open it in edit view.
2. Click **Customize**, and then click the **Digressions** tab.



1. Change the **Allow digressions away** toggle from on to **off**, and then click **Apply**.
2. Click Close to close the node edit view.
3. Click **Clear** in the "Try it out" pane to reset the dialog.
4. Type Book me a restaurant.

The service responds with a prompt for the day to reserve, When do you want to go?

1. Type Tomorrow.

The service responds with a prompt for the time to reserve, What time do you want to go?

1. Ask, What time do you close?

The service recognizes that the question triggers the #restaurant\_opening\_hours intent, but ignores it and displays the prompt associated with the @sys-time slot again instead.

You successfully prevented the user from digressing away from the restaurant booking process.

## **Step 4: Digressing to a node that does not return**

You can configure a dialog node to not go back to the node that the service digressed away from for the current node to be processed. To demonstrate this, you will change the digression setting for the restaurant hours node. In Step 2, you saw that after digressing away from the restaurant booking node to go to the restaurant opening hours node, the service went back to the restaurant booking node to continue with the reservation process. In this exercise, after you change the setting, you will digress away from the **Job opportunities** dialog to ask about restaurant opening hours and see that the service does not return to where it left off.

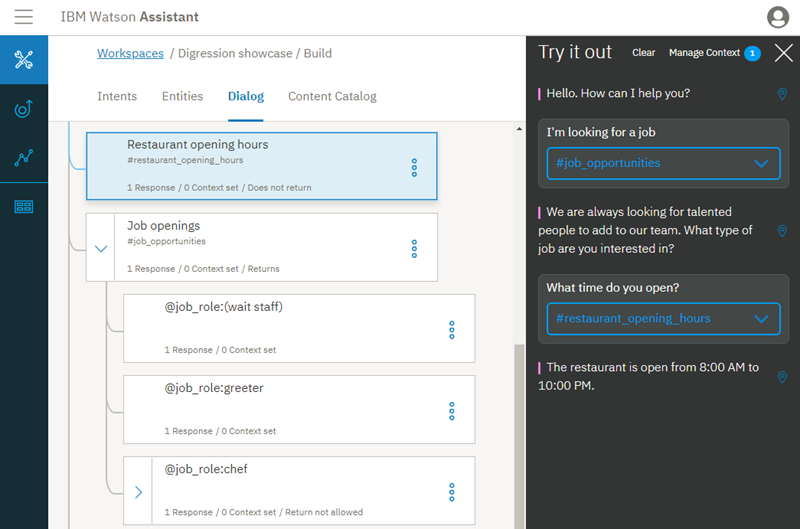
1. Click to open the **Restaurant opening hours** node.
2. Click **Customize**, and then click the **Digressions** tab.
3. Expand the **Digressions can come into this node** section, and deselect the **Return after digression** checkbox. Click **Apply**, and then click Close to close the node edit view.
4. Click **Clear** in the "Try it out" pane to reset the dialog.
5. To engage the **Job opportunities** dialog node, type I'm looking for a job.

The service responds by saying, We are always looking for talented people to add to our team. What type of job are you interested in?

1. Instead of answering this question, ask the bot an unrelated question. Type What time do you open?

The service digresses away from the Job opportunities node to the Restaurant opening hours node to answer your question. The service responds with The restaurant is open from 8:00 AM to 10:00 PM.

Unlike in the previous test, this time the dialog does not pick up where it left off in the **Job opportunities** node. The service does not return to the dialog that was in progress because you changed the setting on the **Restaurant opening hours** node to not return.



Congratulations! You have successfully digressed away from a dialog without returning.

## **Summary**

In this tutorial you experienced how digressions work, and saw how individual dialog node settings can impact the digressions behavior.

## **Next steps**

For help as you configure digressions for your own dialog, see [Digressions](https://console.bluemix.net/docs/services/conversation/dialog-runtime.html#digressions).