

Watson Assistant for IBM Cloud Private

© IBM Corporation 2019



Edition notices

This PDF was created as a supplement to the Watson Assistant for IBM Cloud Private documentation. It may not be a complete set of information or the latest version. For the latest Watson Assistant for IBM Cloud Private documentation, see the IBM Cloud documentation at <https://cloud.ibm.com/docs/services/assistant-icp?topic=assistant-private-getting-started>.

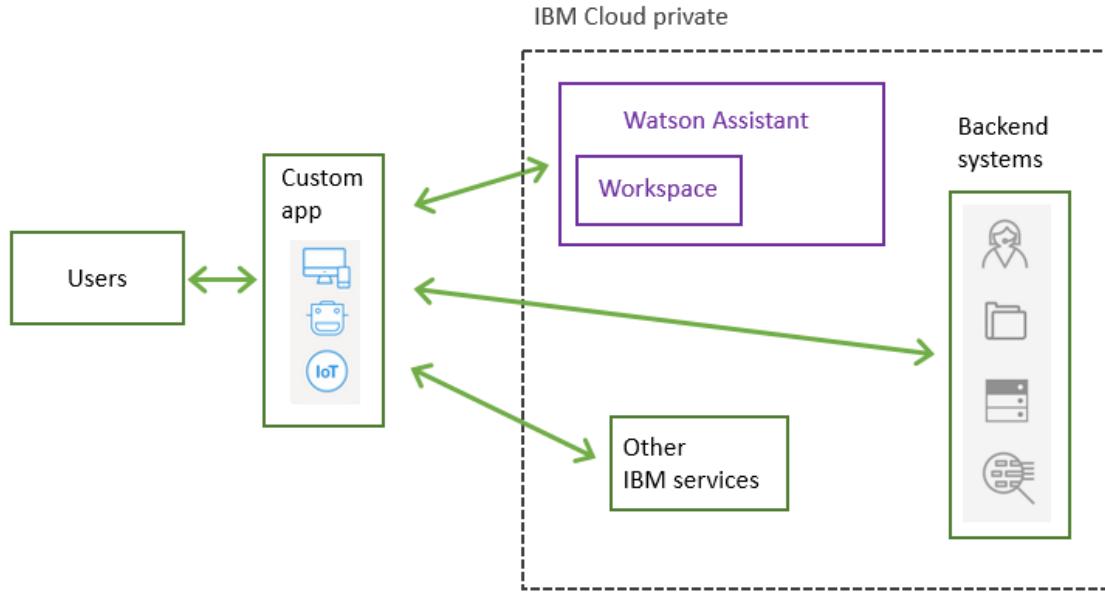
Watson Assistant for IBM Cloud Private: Learn

About

With IBM Watson™ Assistant for IBM® Cloud Private, you can build a solution that understands natural-language input and uses machine learning to respond to customers in a way that simulates a conversation between humans.

How it works

This diagram shows the overall architecture of a complete solution:



- Users interact with your application through the **user interface** that you implement. For example, a simple chat window or a mobile app, or even a robot with a voice interface. The client application can be hosted either inside or outside the private cloud infrastructure.
- The **application** sends the user input to the Watson Assistant service.
 - The application connects to a **workspace**, which is a container for your dialog flow and training data.
 - The service interprets the user input, directs the flow of the conversation and gathers information that it needs.
- The application can interact with the following resources:
 - **Other IBM services**: Connect with additional Watson services to analyze user input, such as Tone Analyzer or Speech to Text.
 - **Back-end systems**: Based on the user's intent and additional information, extract information or perform transactions by interacting with your back-end systems. For example, answer question, open tickets, update account information, or place orders. There is no limit to what you can do.

The tool does not currently include integrations for deploying the finished assistant nor metrics for analyzing conversations that your assistant is having with your users. Such deployment and metrics features are available from the public cloud version of the service only.

Implementation

Here's how you will implement your conversation:

- **Configure a workspace.** With the easy-to-use graphical environment, set up the training data and dialog for your conversation.

The training data consists of the following artifacts:

- **Intents:** Goals that you anticipate your users will have when they interact with the service. Define one intent for each goal that can be identified in a user's input. For example, you might define an intent named *store_hours* that answers questions about store hours. For each intent, you add sample utterances that reflect the input customers might use to ask for the information they need, such as, What time do you open?

Or use prebuilt **content catalogs** provided by IBM to get started with data that addresses common customer goals.

- **Entities:** An entity represents a term or object that provides context for an intent. For example, an entity might be a city name that helps your dialog to distinguish which store the user wants to know store hours for.

As you add training data, a natural language classifier is automatically added to the workspace, and is trained to understand the types of requests that you have indicated the service should listen for and respond to.

Use the dialog tool to build a dialog flow that incorporates your intents and entities. The dialog flow is represented graphically in the tool as a tree. You can add a branch to process each of the intents that you want the service to handle. You can then add branch nodes that handle the many possible permutations of a request based on other factors, such as the entities found in the user input or information that is passed to the service from your application or another external service.

- **Deploy your workspace.** Deploy the configured workspace to users by connecting it to a front-end user interface that you build.

Read more about these implementation steps by following these links:

- [Intent creation overview](#)
- [Dialog overview](#)
- [Entity creation overview](#)
- [Building a client application](#)

Browser support

The Watson Assistant service tooling requires the same level of browser software as is required by IBM Cloud Private. See [Supported browsers](#) for details.

Language support

Language support by feature is detailed in the [Supported languages](#) topic.

Next steps

- [Get started](#) with the service
- Try out some [demos](#).
- View the list of [developer resources](#).

Still have questions? Contact [IBM Sales](#).

Terms and notices

See [IBM Cloud Terms and Notices](#) [] for information about the terms of service.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at [Copyright and trademark information](#) [].

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.



Installing

Installation overview

Learn how to install Watson Assistant for IBM Cloud Private.

The IBM Cloud Private environment is a Kubernetes-based container platform that can help you quickly modernize and automate workloads that are associated with the applications and services you use. You can develop and deploy on your own infrastructure and in your data center which helps to mitigate risk and minimize vulnerabilities.

The installation process differs depending on the version you are installing. The following table shows the available versions.

Available versions		
Watson Assistant chart IBM Cloud Private cluster Installation checklist		
1.1	3.1.0	checklist
1.0.1	2.1.0.3	checklist

Install 1.1 on private cloud 3.1.0

Watson Assistant for IBM Cloud Private version 1.1.0 runs on IBM Cloud Private version 3.1.0.

Version 1.1.0 is compatible with IBM® Cloud Private for Data version 1.2, meaning that both Watson Assistant and IBM Cloud Private for Data can run on the same instance of IBM Cloud Private version 3.1.0. See [Overview of IBM Cloud Private for Data](#) for more information about that offering, including installation instructions.

If you already have IBM Cloud Private for Data version 1.2 running on IBM Cloud Private version 3.1.0, you can install Watson Assistant 1.1.0 on that same IBM Cloud Private version 3.1.0 instance.

Software requirements

- IBM Cloud Private 3.1.0
- Kubernetes 1.11.1
- Helm 2.9.1
- Tiller (Helm server) 2.9.1+icp

System requirements

See [Hardware requirements and recommendations](#) for information about what is required for IBM Cloud Private itself.

All nodes, with the exception of the worker nodes, host the IBM Cloud Private cluster infrastructure. The worker nodes host the Watson Assistant resources. The worker nodes must provide the following number of Virtual Private CPUs (VPCs) to support Watson Assistant for IBM Cloud Private at a minimum.

- **Development:** 21 CPU with 75 GB Memory across a minimum of 1 worker node
- **Production:** 27 CPU with 112 GB Memory across a minimum of 4 worker nodes

These numbers reflect the bare minimum requirements. In a cluster environment, where CPU and memory are assigned to containers dynamically, CPU and memory resources can become stranded on nodes, leaving insufficient resources to schedule subsequent workloads. In particular, the process of training a machine learning model requires at least one node to have 4 CPUs that can be dedicated to training. This capacity is only needed when training occurs, which happens after changes are made to the training data for an assistant.

Tested deployment configurations

Table 1. Hardware verified to support a development deployment

Node type	Non-production hardware requirements				Description
	Number of nodes	CPU per node	Memory per node (GB)	Disk per node (GB)	
boot, master, management, proxy	1	16	128	600	Supports IBM Cloud Private infrastructure
worker	1	24	256	500	Supports the Watson Assistant service

Table 2. Hardware verified to support a production deployment

Production hardware requirements				
Node type	Number of nodes	CPU per node	Memory per node (GB)	Disk per node (GB)
boot, master, proxy	1	16	32	250
management	1	16	32	250
worker	5	8	32	500

The systems that host Watson Assistant must meet these requirements:

- Watson Assistant for IBM Cloud Private can run on Intel architecture nodes only.
- CPUs must have 2.4 GHz or higher clock speed
- CPUs must support Linux SSE 4.2
- CPUs must support the AVX instruction set extension See the [Advanced Vector Extensions](#) Wikipedia page for a list of CPUs that include this support (most CPUs since 2012). The service cannot function properly without AVX support.

Storage requirements

The following table lists the storage resources that have been verified to support a deployment.

Table 3. Storage requirements

Resource requirements			
Component	Number of replicas	Space per pod	Storage type
Postgres	3	10 GB	local-storage
etcd	3	10 GB	local-storage
Minio	4	5 GB	local-storage
MongoDB	3	80 GB	local-storage

Microservices

Microservices are individual components that together comprise the service. The Watson Assistant service consists of the following microservices:

- **CLU (Conversational Language Understanding)**: Interface for store to communicate with the back-end to initiate ML training.
- **Dialog**: Dialog runtime, or user-chat capability.
- **ed-mm**: Manages contextual entity capabilities.
- **Master**: Controls the lifecycle of underlying intent and entity models.
- **SIREG** - Manages tokenization and system entity capabilities.
- **SLAD**: Manages service training capabilities.
- **Store**: API endpoints.
- **TAS**: Manages services model inferencing.
- **Tooling**: Provides the developer user interface.

In addition to these microservices, the Helm chart installs the following resources:

- **PostgreSQL**: Stores training data.
- **MongoDB**: Stores word vectors.
- **Redis**: Used by the Watson Assistant tool to store web session-related data.
- **etcd**: Manages service registration and discovery.
- **Minio**: Stores CLU models.

Language considerations

The components that are necessary to process natural languages require significant amounts of data. The base set of supported languages require 40 GB of memory per node. The following languages require that you add additional resources to support them:

Table 5. Language resource requirements

The following resources are what you need to support a production deployment, where two pods are dedicated to each additional language. In a development deployment, only one pod is used per language, which means you need half the number of resources in a development deployment.

Language	Language resource requirements	
	Additional memory requirements per pod	Additional VPCs required*
Chinese (Simplified or Traditional or both)	8 GB	1
German	1 GB	1
Japanese	1 GB	1
Korean	2 GB	1

*Again, for a development environment, you only need a half of a VPC for each of these languages.

For the full list of supported languages, see [Supported languages](#).

Overview of the steps

Follow these steps to install Watson Assistant for IBM Cloud Private

1. [Purchase and download installation artifacts](#)
2. [Prepare the cloud environment](#)
3. [Upload the Watson Assistant chart to IBM Cloud Private](#)
4. [Create persistent volumes](#)
5. [Install the service from the catalog](#)
6. [Verify that the installation was successful](#)

7. [Launch the tool](#)

Step 1: Purchase and download installation artifacts

1. Purchase Watson Assistant for IBM Cloud Private from [Passport Advantage](#).

2. Download the appropriate package for your environment.

Watson Assistant for IBM Cloud Private includes IBM Cloud Private Foundation version 3.1.0.

If you have IBM Cloud Private version 3.1.0 set up in your environment already, you can download the archive for Watson Assistant alone.

If you do not have IBM Cloud Private or have a later major version of IBM Cloud Private set up in your organization (4.x, for example), you must install version 3.1.0. To get the IBM Cloud Private 3.1.0 archive included, choose the e-assembly that includes IBM Cloud Private.

If you have IBM Cloud Private 2.1.0.3 set up and you do not want to upgrade to using version 3.1.0, then install version 1.0.1 of Watson Assistant instead. Download the PPA file named `IWAICP_V1.0.1.tar.gz`.

The Passport Advantage archive (PPA) file for Watson Assistant contains a Helm chart and images. Helm is the Kubernetes package management system that is used for application management inside an IBM Cloud Private cluster.

Step 2: Prepare the cloud environment

You must have cluster administrator or team administrator access to the systems in your cluster.

1. If you do not have IBM Cloud Private version 3.1.0 set up, install it. See [Installing a standard IBM Cloud Private environment](#).

2. If you do not have Docker installed on your local system, install it now.

You can download the Docker Community Edition for free from the [Docker site](#). Pick the appropriate installation package for your operating system. No Docker Proxy is configured, so you can skip the instructions related to a Docker Proxy.

3. If you have not done so, install the IBM Cloud Private command line interface and log in to your cluster. See [Installing the IBM Cloud Private CLI](#).

After completing the step, you should be able to run the following command:

```
cloudctl login -a https://<cluster_host_name>:8443 --skip-ssl-validation
```

To log in to your cluster, choose the account, and then choose the ibmcom namespace.

4. Configure authentication from your computer to the Docker private image registry host and log in to the private registry. See [Configuring authentication for the Docker CLI](#).

5. If you are not a root user, ensure that your account is part of the docker group. See [Post-installation steps](#) in the Docker documentation.

6. Ensure that you have a stable network connection between your computer and the cluster.

7. Install the Kubernetes command line tool, kubectl, and configure access to your cluster. See [Accessing your cluster from the kubectl CLI](#).

After completing this step, you should be able to run the following command:

```
kubectl cluster-info
```

8. Set up the Helm command line interface.

See [Setting up the Helm CLI](#) for details.

1. Initialize the Helm command line interface.

```
helm init --client-only
```

Important: Do not use the `--upgrade` flag with the `init` command. Adding the `--upgrade` flag replaces the server version of Helm Tiller that is installed with IBM Cloud Private.

2. You can confirm the version numbers by running the following command:

```
helm version --tls
```

The response indicates version numbers similar to these:

```
Client: &version.Version{SemVer:"v2.9.1", ... }  
Server: &version.Version{SemVer:"v2.9.1+icp", ... }
```

9. If you cannot successfully run the commands specified in the previous steps, then stop here.

Review the installation and setup instructions for these command line tools to get them working before you proceed with the installation. You will not be able to successfully install and manage software on your IBM Cloud Private deployment without these tools.

10. Take action now to prevent your users from seeing security warnings when they try to access Watson Assistant later.

After you install Watson Assistant, the Watson Assistant tool will be available from the proxy cluster nodes at the following path `/{{ release.name }}/assistant/ui`. Unlike in the previous release, the tool UI does not have its own subdomain, so a certificate for the tool subdomain cannot be specified in the Helm configuration. If a cluster administrator does not set a certificate for the proxy nodes, then the default self-signed certificate is used. When your users go to the tool UI URL in a web browser to access a node with a self-signed certificate, they will see security warnings. To prevent users from seeing such warnings later, you must perform one of the following procedures now:

- If the cluster node with the proxy role is different from the cluster node with the management role, meaning the proxy and management nodes are different and have different hostnames, then complete these steps:

1. If you don't have a certificate for the proxy hostname that is signed by a trusted Certificate Authority, then get one.
2. Store the certificate and private key for the proxy under the `kube-system` namespace as the `proxy-certs` secret.
3. Modify the proxy (`nginx-ingress-controller`) so it refers to the `proxy-certs` secret. Use the follow command to apply the patch:

```
kubectl patch --namespace kube-system daemonset nginx-ingress-controller -  
-type=json --patch='[{"op": "add", "path":  
"/spec/template/spec/containers/0/args/-1", "value": "--default-ssl-  
certificate=kube-system/proxy-certs"}]'
```

- If the proxy and management nodes have the same hostname:

1. Follow the instructions in the *Replace the authentication certificate for the IBM Cloud Private management console* section of the [Create a new certificate authority \(CA\)](#) topic to

get and set a private key and certificate signed by a trusted Certificate Authority.

2. Start using the `router-certs` for the proxy node. Run the following command to apply the patch:

```
kubectl patch --namespace kube-system daemonset nginx-ingress-controller -  
-type=json --patch='[{"op": "add", "path":  
"/spec/template/spec/containers/0/args/-1", "value": "--default-ssl-  
certificate=kube-system/router-certs"}]'
```

Step 3: Upload the Watson Assistant chart to IBM Cloud Private

Add the Watson Assistant Helm chart to IBM Cloud Private.

1. Make sure that you have enough Docker disk space to load the images in the compressed files to your computer.

You need 60 GB of space on your local system to support the extraction and loading of the archive file. To add more disk space, take one of the following actions:

- Prune or remove old Docker containers, images, and volumes.
- If you are on a client computer with a dockerd, increase the amount of storage that the Docker daemon uses.
To increase the amount of storage that the Docker daemon uses, see the entry for `dm.basesize` in the [dockerd Docker documentation](#).
- If you are on a Mac with Docker Desktop, make sure the Disk Image size is set to 60 GB or greater in the Disk Image Location properties.

If the Disk Image file has grown too close to the total allowed Disk Image size, docker image save your important images, and then remove this file and restart Docker to free up space.

2. If you have not, log in to your cluster from the IBM Cloud Private CLI and log in to the Docker private image registry.

```
cloudctl login -a https://'{icp_url}':8443 [--skip-ssl-validation]  
docker login '{icp_url}':8500 -u '{cluster_username}'
```

The `{icp-url}` is the certificate authority (CA) domain. If you did not specify a CA domain, the default value is `mycluster.icp`. See [Specifying your own certificate authority \(CA\) for IBM Cloud Private services](#).

The `skip-ssl-validate` parameter is needed if your cluster is not set up with a certificate that you trust, such as a self-signed certificate.

The credentials that you must provide to log in to Docker are the same administrative credentials that you use to log into the IBM Cloud Private admin console. You will be prompted to provide the password that is associated with the user name that you specify in `u` parameter.

Step 4: Create persistent volumes

A PersistentVolume (PV) is a unit of storage in the cluster. In the same way that a node is a cluster resource, a persistent volume is also a resource in the cluster.

For an overview, see [Persistent Volumes in the Kubernetes documentation](#).

When you install the service, persistent volume claims are created for the components automatically. However, because the preferred storage class for the service is `local-storage`, you must explicitly create persistent volumes before you install the service. Create one persistent volumes for each replica specified in the [system requirements](#) table earlier. See [Creating a PersistentVolume](#) for the steps to take to create one.

Note: You must be a cluster administrator to create local storage volumes.

To create the volumes, you can define each volume configuration in a YAML file, and then use the Kubectl command line to push the configuration changes to the cluster. Use the [apply](#) command in a command with the following syntax:

```
kubectl apply -f {pv-yaml-file-name}
```

1. Create 13 YAML files, one for each volume.

Specify the following settings in the YAML files.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  finalizers:
    - kubernetes.io/pv-protection
  name: {name}
spec:
  capacity:
    storage: {size}
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: {path}
    type: ''
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: local-storage
```

Replace the variables in this snippet with the appropriate value for the volume.

- {name}: If you use a naming convention that includes the storage size information, it will be easier to recognize the volumes later. For example, you could use names like these:
 - For volumes 1 through 6 that have a size of 10Gi, use pv-10g-n where n starts at 1 and goes up to 6.
 - For volumes 7-10 that have a size of 5Gi, use pv-5g-n where n starts at 1 and goes up to 4.
 - For volumes 11-13 that have a size of 80Gi, use pv-80g-n where n starts at 1 and goes up to 3.
- {size}: Reflect the size specified in the [resource requirements table](#).
- {path}: Path on the worker node where you create the persistent volume. For example, /mnt/local-storage/storage/{dir-name}. For {dir-name}, use the same value that you use for {name} so you can map the volume name to its physical location.

2. Run the apply command on each YAML file that you create.

For example: `kubectl apply -f pv_001.yaml`. Rerun this command for each file up to `kubectl apply -f pv_013.yaml`.

You can use the following command to verify that the persistent volumes were created successfully:

```
kubectl get pv | grep g-
```

The resulting list should look something like this:

```
pv-10g-1 10G RWO Recycle Available local-storage 11m
pv-10g-2 10G RWO Recycle Available local-storage 7m
pv-10g-3 10G RWO Recycle Available local-storage 7m
pv-10g-4 10G RWO Recycle Available local-storage 7m
```

```
pv-10g-5 10G RWO Recycle Available local-storage 7m
pv-10g-6 10G RWO Recycle Available local-storage 7m
pv-5g-1 5G RWO Recycle Available local-storage 7m
pv-5g-2 5G RWO Recycle Available local-storage 7m
pv-5g-3 5G RWO Recycle Available local-storage 7m
pv-5g-4 5G RWO Recycle Available local-storage 7m
pv-80g-1 80G RWO Recycle Available local-storage 7m
pv-80g-2 80G RWO Recycle Available local-storage 7m
pv-80g-3 80G RWO Recycle Available local-storage 7m
```

Step 5: Install the service from the catalog

- From the Kubernetes command line tool, create the namespace in which to deploy the service. Use the following command to create the namespace:

```
kubectl create namespace {namespace-name}
```

For example:

```
kubectl create namespace conversation
```

If you do not have the Kubernetes command line tool set up, complete the steps in [Prepare the cloud environment]install-110-install-icp).

- Check whether you can log in to Docker by using the following command:

```
docker login <cluster_CA_domain>:8500
```

If you cannot log in, then you need to get a certificate from your IBM Cloud Private cluster and install it to Docker or add the {cluster_CA_domain} as a Docker Daemon insecure registry. You must do one or the other for Docker to be able to pull from your IBM Cloud Private cluster.

Follow the steps documented in [Configuring authentication for the Docker CLI](#).

- To load the file from Passport Advantage into IBM Cloud Private, enter the following command in the IBM Cloud Private command line interface.

The load process takes a while. Be sure you have a wired connection to the network; do not try to load the chart over a wireless connection.

```
cloudctl catalog load-archive --archive '{result_ppa}' --registry '{icp_url}:8500'
```

- {result_ppa} is the name of the file that you downloaded from Passport Advantage. For example, ibm-watson-assistant-prod-1.1.0.tar.gz.
- {icp_url}:8500 is the ICP cluster docker registry. {icp_url} is the IBM Cloud Private cluster domain.

It is loaded to the default local-charts repo.

- View the charts in the IBM Cloud Private Catalog. From the IBM Cloud Private management console navigation menu, click **Manage > Helm Repositories**.

- Click **Sync Repositories**.

You must have the *cluster administrator* user type or access level to sync repositories.

- From the navigation menu, select **Catalog**.

- Scroll to find the **ibm-watson-assistant-prod** package, and then click the **Configuration** tab.

- Specify values for the installation details fields.

3. Specify values for the installation details fields.

- Helm release name.

The release name must start with an alphabetic character, end with an alphanumeric character, and consist of lower case alphanumeric characters or a hyphen (-). For example *my-110-wa*.

- Target namespace

6. Click License agreement.

Click **Next** multiple times to read the full agreement, and then click **OK** to close the license terms. If you have read and agree to the license agreement, select the checkbox.

7. When you install the service, many configuration settings are applied to it for you unless you override them with your own values.

At a minimum, provide your own values for the following configurable setting:

- Deployment Type
- Languages
- Hostname of the ICP cluster Master node

If you did not define a domain name for the master node of your private cloud instance, you are using the default hostname `mycluster.icp`, for example, then you must also specify values for these configuration settings:

- IP address of the master node
- Host name of the proxy node

8. You might want to change additional configuration settings at this time. You cannot change these settings after you complete the installation.

See [Configuration details](#) for help understanding the configuration choices.

9. Click Install.

You can check the Helm releases page to find out the status of the installation. See [Verify that the installation was successful](#).

Configuration details

Currently, the service does not support the ability to provide your own instances of resources, such as Postgres or MongoDB. There are configuration settings that suggest you can do so. However, do not change these settings from their default value of `true`.

Table 6. Configuration settings

Setting	Configuration settings	Description
Helm release name	A unique ID for this deployment. The release name must start with an alphabetic character, end with an alphanumeric character, and consist of lower case alphanumeric characters or a hyphen (-). For example <i>my-110-wa</i> . When you install the service from the command line, you set this value by using the <code>--name</code> parameter.	
Target namespace	Namespace in the cluster where Watson Assistant will be installed. This is the namespace you created earlier, and to which you uploaded the product archive file.	

Setting	Description
Deployment Type	Options are Development and Production . Development is a Private cloud environment that you can use for testing purposes. It contains a single pod for each microservice. Production is a Private cloud environment that you can use to host applications and services used in production. Contains two replicas of each microservice pod. Development is the default.
Hostname of the ICP cluster Master node	Required. Specify the cluster_CA_domain hostname of the master node of your private cloud instance. This is the domain where you log in to the cluster. For example: my.company.name.icp.net. Specify the hostname only, without a protocol prefix (<code>https://</code>) and without a port number (<code>:8443</code>). This unique URL is typically referred to as the <code>{icp-url}</code> in this documentation. Corresponds to the <code>global.icp.masterHostname</code> value in the <code>values.yaml</code> file.
IP (v4) address of the master node	Required only if the hostname of the master node is not a DNS-resolvable name, such as <code>mycluster.icp</code> . Specify the IP address of the master node. Corresponds to the <code>global.icp.masterIP</code> value in the <code>values.yaml</code> file.
Hostname of the ICP cluster proxy node	Specify the hostname of the proxy node from which the ingress and services are accessed. To discover this value, run the command: <code>kubectl get nodes --show-labels</code> , and then find the node that shows <code>proxy=true</code> , and get the hostname value. It might be an IP address instead of a typical hostname. Corresponds to the <code>global.icp.proxyHostname</code> value in the <code>values.yaml</code> file.
Ingress path	Specifies how to access the Watson Assistant. You can access the service from <code>https://{{ global.icp.proxyHostname }}{{ global.icp.ingress.path }}</code> . If left empty, the default value <code>/{{ .Release.Name }}/assistant</code> is used for the ingress path. - The API is available from <code>https://{{ global.icp.proxyHostname }}{{ global.icp.ingress.path }}/api</code> . - The tool is available from <code>https://{{ global.icp.proxyHostname }}{{ global.icp.ingress.path }}/ui</code> .
Languages	Specify the languages you want to support in addition to. English is required; do not deselect it. See Language considerations for more information.
Create COS	Boolean. Indicates whether you want a cloud object storage instance to be automatically created for you during the installation or you want to provide your own instance. If true, a Minio cloud object store is created. The default value is true. Do not set to false. The service does not currently support providing your own data store.
COS Bucket Prefix	Prefix of the bucket names to be used. <code>icp</code> is the default prefix.
COS Access Protocol	Only used if Create COS is deselected. Specifies the protocol used to connect to COS. Options are <code>http</code> and <code>https</code> . Typical protocol is <code>http</code> .
COS Hostname	Only used if Create COS is deselected. Hostname to connect to the store. Typical hostname when COS is running in the cluster is <code>{{ COS ServiceName }}.{{ namespace }}.svc.cluster.local</code> .
COS Port	Only used if Create COS is deselected. Port where COS is listening. Typical port is 443 and 9000.
COS Secret Name	Name of the secret you created to hold the keys (<code>accesskey</code> and <code>secretkey</code>) for Minio. If not specified, the secret is created for you with randomly generated keys.
Create Redis	Boolean. Specify <code>true</code> to have a Redis store created for you. Specify <code>false</code> to provide your own instance. The default value is true. Do not set to false. The service does not currently support providing your own data store.

Setting	Description
Redis Secret	If you want to specify a custom password for Redis, create a secret named <code>password</code> and specify the <code>password</code> value for it. If not specified, a secrets is created for you with a randomly generated password.
Redis Hostname	Used only if Create Redis is deselected. Specifies the hostname of the running Redis service.
Redis Port	Used only if Create Redis is deselected. Specifies the port from which the running Redis service can be accessed. The port is typically 6379.
Create Postgres	Boolean. Specify <code>true</code> to have a Postgres database created for you. Specify <code>false</code> to provide your own instance. The default value is <code>true</code> . Do not set to false. The service does not currently support providing your own data store.
Postgres Hostname	Used only if Create Postgres is deselected. Specifies the hostname of the running Postgres database server.
Postgres Port	Used only if Create Postgres is deselected. Specifies the port from which the running Postgres database server can be accessed. The port is typically 5432.
Postgres Admin Name	User ID for a Postgres super user with rights to create databases and users in the Postgres database. If you use your own instance, then you must provide the correct admin name and, in a secret, its password.
Postgres Admin Secret Name	Secret that contains the password associated with the user ID for a Postgres super user with rights to create databases and users in the Postgres database. If you use your own instance, then specify the name of the secret that contains your password. If not specified, a secret is created for you with a randomly generated password.
Postgres Admin Database	Name of the database to connect to. The name is typically <code>postgres</code> .
SSL Mode	SSL mode to use for the Postgres connection. Options include <code>verify-ca</code> and <code>verify-full</code> . Do not change from the default value of <code>verify-full</code> .
Postgres SSL Secret Name	Name of the secret that holds the certificate for the SSL connection to the Postgres server (<code>key tls.crt</code>). When you use the Postgres database that is installed for you, but want to use a custom certificate, then you must specify the private key (<code>tls.key</code>) also.
Create Database	Boolean. Specify <code>true</code> to have the database and user created for you. If you specify <code>false</code> , then you must create the database and database user yourself. The default value is <code>true</code> . Do not set to false. The service does not currently support providing your own database.
Create Schema	Boolean. Specify <code>true</code> to have the required tables, functions, and so on applied to the database that is created for you. The default value is <code>true</code> . Do not set to false. The service does not currently support providing your own schema.
Store Database Name	Database name that the store microservice uses. If left empty, the default value <code>conversation-icp-{{ .Release.Name }}</code> is used.
Postgres User Name	User name that the store miroservice uses to connect to the Postgres database. If left empty, the default value <code>store-icp-{{ .Release.Name }}</code> is used.

Setting	Description
Postgres Secret Name	Name of the secret that holds the password key of the postgres user for store microservice. If not specified, a secret is created for you with a randomly generated password value.
Create Etcd Cluster	Boolean. Specify true to have the etcd cluster for the Watson Assistant service created for you. Specify false to provide your own etcd instance and the associated credentials. The default value is true . Do not set to false. The service does not currently support providing your own cluster.
Etcd User	User ID used to access etcd. The default value is root.
Etcd Secret Name	Name of the secret that holds the password key for the Redis (super)user. If not specified, the secrets is created for you with a randomly generated password.
Connections details for etcd	Used only if Create Etcd Cluster is deselected. Etcd connection details. The format corresponds to the --endpoints parameter of the etcdctl tool. The syntax used is schema://hostname:port where schema is either http or https and port is typically 2379. If multiple endpoints are provided, they are separated by commas (,)`.
Create MongoDB	Boolean. Specify true to have a MongoDB collection created for you. Specify false to provide your own instance. The default value is true . Do not set to false. The service does not currently support providing your own database.
MongoDB Hostname	Used only if Create MongoDB is deselected. Specifies the hostname of the running MongoDB collection.
MongoDB Port	Used only if Create MongoDB is deselected. Specifies the port from which the running MongoDB collection can be accessed.
Enable Authentication	Boolean. If true, authentication is enabled. The default value is true.
MongoDB Admin User	User ID for a MongoDB super user. If you use your own instance, then you must change this name and its associated password.
MongoDB Admin Password	Password associated with the user ID for a MongoDB super user. If you use your own instance, then you must change this password and the associated name.
Enable TLS	Boolean. Indicates whether to enable MongoDB TLS support. The default value is true.
TLS Secret Name	Name of the secret that holds the certificate (key <code>tls.crt</code>) and private key.
COS Configuration PVC Size	Specifies the size of the persistent volume claim to be used by the cloud object store. Default size is 5 Gi.
COS Configuration Storage Class	Specifies the persistent volume storage class. Default class is <code>local-storage</code> .
etcd Configuration PVC Size	Specifies the size of the persistent volume claim to be used by the etcd data store. Default size is 10 Gi.

Setting	Description
etcd Configuration Storage Class	Specifies the persistent volume storage class. Default class is local-storage.
PostgreSQL Datoare Storage Class	Specifies the storage class of the persistent volume for the PostgreSQL database. Default class is local-storage.

Step 6: Verify that the installation was successful

To check the status of the installation process:

1. Log in to the IBM Cloud Private management console.
2. From the main menu, expand **Workloads**, and then choose **Helm releases**.
3. Find the release name you used for the deployment in the list.

- o If the deployment failed, use the following command to get information about each pod:

```
kubectl get pods -n {namespace-name} | grep {release name}
```

And then use this command to see the logs for a pod that failed:

```
kubectl logs {podname} -n {namespace} -f --timestamps
```

A common issue that can block successful deployments is that the persistent volume claims (PVCs) can bind themselves to the wrong persistent volumes, volumes that you did not create.

Next, complete the steps in [Uninstalling the service](#), so you can start the installation over with a clean set of nodes.

- o If the deployment process was successful, test Watson Assistant by running a test Helm chart.
 1. From the Helm command line interface, run the following command:


```
helm test --tls {release name} --timeout 900
```
 2. If one of the tests fails, review the logs to learn more. To see the log, use a command with the syntax `kubectl logs {podname} -n {namespace-name} -f --timestamps`. For example:


```
kubectl logs my-release-test -n conversation -f --timestamps
```
 3. To run the test script again, first delete the test pods by using a command with the syntax `kubectl delete pod {podname} --namespace {namespace-name}`. For example:


```
kubectl delete pod my-release-test --namespace conversation
```
- You must delete all of the `{podname}` pods, not just the one that failed, or the other tests will fail also.

Uninstalling the service

If you need to start the deployment over, be sure to remove all trace of the current installation before you try to install again.

1. To uninstall and delete your deployment, run the following command from the Helm command line interface:

```
helm delete --tls --no-hooks --purge {release name}
```

This command removes the release resources.

To check whether any resources are left behind on the cluster, run this command:

```
kubectl get  
job,deploy,rs,pod,statefulset,configmap,secret,ingress,service,serviceaccount,role,rolebinding,pvc -l release={release name}
```

You can then delete individual resources that you want to remove by using a command like this:

```
kubectl delete {resource name}
```

Also check for any remaining release resources associated with the namespace you used, and delete them. You can use this command to find them:

```
kubectl get all -n {namespace-name} | grep {release name}
```

And then delete them by resource name. For example:

```
kubectl delete configmap/stolon-cluster-{release name}
```

2. The PersistentVolumeClaims will not be deleted and will remain bound to persistent volumes. You must remove them manually. See [Deleting a PersistentVolumeClaim](#) for details.

3. Remove all content from any persistent volumes that you used for the previous deployment before you restart the installation. See [Deleting a PersistentVolume](#) for more information.

Installing from the command line

If you have trouble when you install from the catalog, you can install by using the command line interface instead.

Be sure to check that the installation from the catalog did not complete. Even if you see an error message, the installation might complete successfully.

To install from the command line, complete these steps:

1. From the Kubernetes command line tool, create the namespace in which to deploy the service. Use the following command to create the namespace:

```
kubectl create namespace {namespace-name}
```

For example:

```
kubectl create namespace conversation
```

If you do not have the Kubernetes command line tool set up, see [Accessing your cluster from the kubectl CLI](#) for instructions.

2. Get a certificate from your IBM Cloud Private cluster and install it to Docker or add the {cluster_CA_domain} as a Docker Daemon insecure registry. You must do one or the other for Docker to be able to pull from your IBM Cloud Private cluster.

See [Specifying your own certificate authority \(CA\) for IBM Cloud Private services](#).

3. To load the file from Passport Advantage into IBM Cloud Private, enter the following command in the IBM Cloud Private command line interface.

```
cloudctl catalog load-archive --registry {icp_url}:8500 --archive ibm-watson-assistant.1.1.0.tar.gz --repo local-charts
```

4. If you have a pre-existing version of the service on your cluster, remove its TGZ file. For example:

```
rm ibm-watson-assistant-prod-1.0.1.tgz
```

5. Run this command to download the chart from the IBM Cloud Private repository:

```
wget --no-check-certificate https://{{cluster_CA_domain}}:8443/helm-repo/requiredAssets/ibm-watson-assistant-prod-1.1.0.tgz
```

6. Extract the TAR file from the TGZ file, and then extract files from the TAR file by using the following command:

```
tar -xvzf /path/to/ibm-watson-assistant-prod-1.1.0.tgz
```

7. Extract the TAR file from the TGZ file, and then extract files from the TAR file by using the following command:

```
tar -xvzf /path/to/ibm-watson-assistant-prod-1.1.0.tgz
```

8. Edit values in the values.yaml file.

To do so, first make a copy of the values.yaml. The values.yaml file is stored with the chart. Rename the file. For example, cp ibm-watson-assistant-prod/values.yaml my-override.yaml.

In your copy of the file, comment out or remove all but the configuration settings that you want to replace with your own values.

Validate the Docker image repository values in the file with values that reflect your environment. Each image repository value must have the syntax {{icp_url}}:8500/{{namespace-name}}/{{unique-path-value}} where {{unique-path-value}} reflects the path from the existing YAML file, such as icp-wa-tooling-patched.

At a minimum, you must provide your own values for the following configurable settings also:

- global.deploymentType: Specify whether you want to set up a development or production instance.
- global.icp.masterHostname: Specify the hostname of the master node of your private cloud instance. Do not include the protocol prefix (<https://>) or port number (:8443). For example: my.company.name.icp.net.
- global.icp.masterIP: If you did not define a domain name for the master node of your private cloud instance, you are using the default hostname mycluster.icp, for example, then you must also specify this IP address.
- global.icp.proxyHostname: Specify the hostname (or IP address) of the proxy node of your private cloud instance.

Attention: Currently, the service does not support the ability to provide your own instances of resources, such as Postgres or MongoDB. The values YAML file has {{resource-name}}.create settings that suggest you can do so. However, do not change these settings from their default value of true.

9. After you define any custom configuration settings, you can install the chart from the Helm command line interface. Enter the following command from the directory where the package was loaded in your local system:

```
helm install --tls --values {{override-file-name}} --namespace {{namespace-name}} --name {{my-release}} ibm-watson-assistant-prod-1.1.0.tgz
```

- Replace {{my-release}} with a name for your release. The release name must start with an alphabetic character, end with an alphanumeric character, and consist of lower case alphanumeric characters or a hyphen (-). For example my-110-wa.
- Replace {{override-file-name}} with the path to the file that contains the values that you want to override from the values.yaml file provided with the chart package. For example: my-override.yaml
- Replace {{namespace-name}} with the name of the Kubernetes namespace that hosts the Docker pods.

- The `ibm-watson-assistant-prod-1.1.0.tgz` parameter represents the name of the downloaded file that contains the Helm chart.

After the installation finishes, [verify](#) that it was successful.

Step 7: Launch the tool

1. Open a new tab in a web browser, and then enter a URL with the syntax `https://{{global.icp.proxyHostname}}{{global.icp.ingress.path}}/ui`. For example: `https://myproxy.icp/my-release/assistant/ui`.
2. Log in using the same credentials you used to log into the IBM Cloud Private dashboard.

Troubleshooting installation issues

Investigating issues

The first step to take if you hit an installation issue, such as a cluster node is not starting as expected, is to get logs from the cluster which can provide more detail.

To get log files, complete the following steps:

1. Log into the cluster with administrator credentials.
2. Run the following command to get a list of the jobs that are currently running in the cluster and whether the job was successful:

```
kubectl get jobs
```

3. For any jobs that show a success status of 0, get the log file for the job by entering the following command:

```
kubectl log {job-name} -f
```

To check the configuration

If you want to check what configuration settings were applied to a deployment when it was set up, you can run the following command:

```
helm get <release-name> --tls
```

The user-provided configuration values are listed at the start of the information that is returned.

Next steps

Use the Watson Assistant tool to build training data and a dialog that can be used by your assistant.

- To learn more about the service first, read the [overview](#).
- To see how it works for yourself, follow the steps in the [getting started tutorial](#).

Install 1.0.1 on private cloud 2.1.0.3

Watson Assistant version 1.0.1 runs on IBM® Cloud Private version 2.1.0.3.

Software requirements

- IBM Cloud Private 2.1.0.3

- Kubernetes 1.10.0
- Helm 2.7.2
- Tiller (Helm server) 2.7.3+icp

System requirements

Table 1. Minimum hardware requirements for a development environment

Minimum non-production hardware requirements				
Node type	Number of nodes	CPU per node	Memory per node	Disk per node
boot	1	2	8	250
master	1	4	8	250
management	1	4	8	250
proxy	1	2	4	140
worker	4	8	32	500

All nodes, with the exception of the worker nodes, host the cluster infrastructure. The worker nodes host the Watson Assistant resources.

The systems must meet these requirements:

- Watson Assistant for IBM Cloud Private can run on Intel architecture nodes only.
- CPUs must have 2.4 GHz or higher clock speed
- CPUs must support Linux SSE 4.2
- CPUs must support the AVX instruction set extension. The ed-mm microservice cannot function properly without AVX support.

See [Hardware requirements and recommendations](#) [] for information about what is required for IBM Cloud Private itself.

Resource requirements

The following table lists the system resources that have been verified to support a deployment.

Table 2. Resource requirements

Resource requirements			
Component	Number of replicas	Space per pod	Storage type
Postgres	3	10 GB	local-storage
etcd	3	10 GB	local-storage
Minio	4	5 GB	local-storage
MongoDB	3	80 GB	local-storage

You must provide at least 60 Virtual Private CPUs (VPCs) to support Watson Assistant for IBM Cloud Private.

Microservices

Microservices are individual components that together comprise the service. The Watson Assistant service consists of the following microservices:

- **Dialog**: Dialog runtime, or user-chat capability.
- **Store**: API endpoints.
- **CLU (Conversational Language Understanding)**: Interface for store to communicate with the back-end to initiate ML training.
- **Master**: Controls the lifecycle of underlying intent and entity models.
- **TAS**: Manages services model inferencing.
- **SLAD**: Manages service training capabilities.
- **SIREG** - Manages tokenization and system entity capabilities.
- **ed-mm**: Manages contextual entity capabilities.
- **Tooling**: Provides the developer user interface.

In addition to these microservices, the Helm chart installs the following resources:

- **PostgreSQL**: Stores training data.
- **MongoDB**: Stores word vectors.
- **Redis**: Caches data.
- **etcd**: Manages service registration and discovery.
- **Minio**: Stores CLU models.

Language considerations

The components that are necessary to process different natural languages require significant amounts of data. Resources to support English are always provided. Each other language that you enable during the installation process (besides Czech) increases the amount of resources that you need to support it.

Table 3. Language resource requirements

Language resource requirements	
Language	Memory requirements per pod
Base service + English, Czech	40 GB
Each other language you add	10 GB (12 GB for Portuguese and Chinese)

Overview of the steps

1. [Download service installation artifacts](#)
2. [Prepare the cloud environment](#)
3. [Add the service chart to the cloud repository](#)
4. [Install the service](#)
5. [Verify that the installation was successful](#)
6. [Launch the tool](#)

Step 1: Purchase and download installation artifacts

1. Purchase Watson Assistant for IBM Cloud Private from [Passport Advantage](#).
2. Download the appropriate package for your environment.

Watson Assistant for IBM Cloud Private includes IBM Cloud Private Foundation version 2.1.0.3.

If you have IBM Cloud Private version 2.1.0.3 set up in your environment already, you can download the archive for

Watson Assistant only.

If you do not have IBM Cloud Private set up in your organization, then you must install version 2.1.0.3. To get the IBM Cloud Private 2.1.0.3 archive included, choose the e-assembly that includes IBM Cloud Private.

The Passport Advantage archive (PPA) file for Watson Assistant contains a Helm chart and images. Helm is the Kubernetes native package management system that is used for application management inside an IBM Cloud Private cluster.

Attention: If you downloaded the PPA file before 23 November 2018, then you have an earlier version of the service. The installation process was simplified with the PPA file version 1.0.1 made available on 23 November 2018. Download the latest version of the PPA file, named ibm-watson-assistant-prod-1.0.1.tgz.

Step 2: Prepare the cloud environment

You must have cluster administrator or team administrator access to the systems in your cluster.

1. If you do not have IBM Cloud Private version 2.1.0.3 set up, install it. See [Installing a standard IBM Cloud Private environment](#).
2. Synchronize the clocks of the client computer and the nodes in the IBM Cloud Private cluster. To synchronize your clocks, you can use network time protocol (NTP). For more information about setting up NTP, see the user documentation for your operating system.
3. If you have not done so, install the IBM Cloud Private command line interface and log in to your cluster. See [Installing the IBM Cloud Private CLI](#).
4. Configure authentication from your computer to the Docker private image registry host and log in to the private registry. See [Configuring authentication for the Docker CLI](#).
5. If you are not a root user, ensure that your account is part of the docker group. See [Post-installation steps](#) in the Docker documentation.
6. Ensure that you have a stable network connection between your computer and the cluster.
7. Install the Kubernetes command line tool, kubectl, and configure access to your cluster. See [Accessing your IBM Cloud Private cluster by using the kubectl CLI](#).
8. Obtain access to the boot node and the cluster administrator account, or request that someone with that access level create your certificate. If you cannot access the cluster administrator account, you need a IBM Cloud Private account that is assigned to the operator or administrator role for a team and can access the kube-system namespace.
9. Set up the Helm command line interface.

See [Setting up the Helm CLI](#) for details.

Attention: The instructions ask you to choose between two ways of installing Helm. Here are some things to know about each installation method:

- If you download Helm directly from GitHub, get version 2.7.2. Add the Helm executable binary file to your PATH.
- If you use the Helm package that is included with IBM Cloud Private, you are instructed to run a Docker command to install it. The Docker command downloads Helm from a publicly available Docker image (from Dockerhub), extracts a single Helm file from it, and then copies this file to a directory on your PATH.
- Initialize the Helm command line interface.

```
helm init --client-only
```

Important: Do not use the --upgrade flag with the init command. Adding the --upgrade flag replaces the server version of Helm Tiller that is installed with IBM Cloud Private.

- You can confirm the version numbers by running the following command:

```
helm version --tls
```

The response indicates version numbers similar to these:

```
Client: &version.Version{SemVer:"v2.7.2" ... } Server: &version.Version{SemVer:"v2.7.3+icp" ... }
```

10. Validate that AVX is supported on the systems in your cluster. (AVX support is a hardware requirement.) To do so, run the following command:

```
kubectl exec {name-of-pod-hosting-service} -n {namespace-name} -- cat /proc/cpuinfo | grep avx
```

You should see one hit for each CPU on the worker node.

Step 3: Add the Helm chart to the cloud repository

Add the Watson Assistant Helm chart to the IBM Cloud Private internal repository.

1. Ensure that you have enough disk space to load the images in the compressed files to your computer.

You need 30 GB of space on your local system to support the extraction and loading of the archive file.

1. Check the Docker disk usage. Run the following command:

```
docker system df
```

For more command options, see [docker system df in the Docker documentation](#).

2. If you need more disk space, take one of the following actions:

- Remove old Docker images.
- Increase the amount of storage that the Docker daemon uses. To increase the amount of storage that the Docker daemon uses, see the entry for `dm.basesize` in the [dockerd Docker documentation](#).

2. If you have not, log in to your cluster from the IBM Cloud Private CLI and log in to the Docker private image registry.

```
bx pr login -a https://{{icp-url}}:8443 --skip-ssl-validation  
docker login {{icp-url}}:8500
```

Where {{icp-url}} is the certificate authority (CA) domain. If you did not specify a CA domain, the default value is

mycluster.icp. See [Specifying your own certificate authority \(CA\) for IBM Cloud Private services](#).

.

Step 4: Install the service

- [4.1 Configure DNS name resolution](#)
- [4.2 Create persistent volumes](#)
- [4.3 Gather information about your environment](#)
- [4.4 Install the service from the catalog](#)

4.1 Configure DNS name resolution

Work with your DNS provider to create a subdomain on your cluster named `assistant` that can be used by the Watson Assistant tool user interface.

For example, if you are using SoftLayer, log in to the SoftLayer portal, and go to Network > DNS > Forward Zones. In the DNS Forwarding Zone for your IBM Cloud Private cluster, add a new record with the host name 'assistant' that points to the IP address of your IBM Cloud Private cluster.

The installation process and all worker nodes must be able to resolve the following components by name (not IP address):

- IBM Cloud Private cluster (**Hostname of the ICP cluster Master node** or `global.icp.masterHostname`)
- Watson Assistant tool user interface (**Subdomain** or `ui.subdomain`)

You must be able to ping both URLs and get replies.

4.2 Create persistent volumes

A PersistentVolume (PV) is a unit of storage in the cluster. In the same way that a node is a cluster resource, a persistent volume is also a resource in the cluster.

For an overview, see [Persistent Volumes in the Kubernetes documentation](#).

When you install the service, persistent volume claims are created for the components automatically. However, because the preferred storage class for the service is **local-storage**, you must explicitly create persistent volumes before you install the service. Create one persistent volumes for each replica specified in the [system requirements](#) table earlier. See [Creating a PersistentVolume](#) for the steps to take to create one.

Note: You must be a cluster administrator to create local storage volumes.

To create the volumes, you can define each volume configuration in a YAML file, and then use the Kubectl command line to push the configuration changes to the cluster. Use the [apply](#) command in a command with the following syntax:

```
kubectl apply -f {pv-yaml-file-name}
```

1. Create 13 YAML files, one for each volume.

Specify the following settings in the YAML files.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  finalizers:
    - kubernetes.io/pv-protection
  name: {name}
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: {size}
  hostPath:
    path: /mnt/local-storage/storage/{dir-name}
    type: ''
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: local-storage
```

Replace the variables in this snippet with the appropriate value for the volume.

- {name}: If you use a naming convention that includes the storage size information, it will be easier to recognize the volumes later. For example, you could use names like these:
 - For volumes 1 through 6 that have a size of 10Gi, use `pv-10g-n` where n starts at 1 and goes up to 6.

- For volumes 7-10 that have a size of 5Gi, use pv-5g-n where n starts at 1 and goes up to 4.
- For volumes 11-13 that have a size of 80Gi, use pv-80g-n where n starts at 1 and goes up to 3.
- {dir-name}: Use the same value that you use for {name} so you can map the volume name to its physical location.
- {size}: Reflect the size specified in the [resource requirements table](#).

2. Run the apply command on each YAML file that you create.

For example: kubectl apply -f pv_001.yaml. Rerun this command for each file up to kubectl apply -f pv_013.yaml.

The result is 13 volumes with names like these:

- pv-10g-1
- pv-10g-2
- pv-10g-3
- pv-10g-4
- pv-10g-5
- pv-10g-6
- pv-5g-1
- pv-5g-2
- pv-5g-3
- pv-5g-4
- pv-80g-1
- pv-80g-2
- pv-80g-3

4.3 Gather information about your environment

When you install the service, many configuration settings are applied to it for you unless you override them with your own values. You might want to change things like user names and passwords for databases or stores that are created for you, for example. **You cannot change these settings after you complete the installation.**

Other actions you might want to take before starting the installation include:

- **Generate a MongoDB TLS certificate:** If you create your own certificate authority, you can replace the default values for the CA with details for your own certificate and key.

1. Generate your own TLS certificate authority for MongoDB.

```
$ openssl genrsa -out ca.key 2048
$ openssl req -x509 -new -nodes -key ca.key -days 10000 -out ca.crt -subj
  "/CN=mydomain.com"
```

2. Encode it in base64 format.

```
$ cat ca.key | base64 -w0
```

where -w0 prevents base64 from wrapping the line when it reaches more than 80 characters in length. This option might not be supported on all base64 versions. If your version does not, then use another method to ensure that the output is a single line.

3. Update the base64 encoded certificate and base64 encoded key configuration settings (or override the configuraton values global.mongodb.tls.cacert and global.mongodb.tls.cakey) with the new certificate and key values.

- **Create a TLS secret for the tool:** To prevent users from seeing browser security warnings when using the tooling, you can install a certificate.

Before you install, obtain a certificate for the tooling subdomain `assistant.{icp-url}`. The certificate should be signed by a trusted certificate authority. Because assistant tooling uses a subdomain the certificate must either be specifically for that subdomain, or a wildcard certificate for your IBM Cloud Private domain.

After you get the certificate and private key, create a secret with keys named `tls.crt` and `tls.key` that contain the certificate and private key. See [TLS in the Kubernetes documentation](#) for information about how to create the secret.

After you create the secret, update the TLS Secret configuration setting (or override the configuration value `ui.ingress.tlsSecret`) with the name of that secret. Ingress will use that certificate when users access the tooling.

4.4 Install the service from the catalog

1. From the Kubernetes command line tool, create the namespace in which to deploy the service. If you enable a language other than English and Czech, then you must specify `conversation` as the namespace. Otherwise, you can use any namespace you choose. Use the following command to create the namespace:

```
kubectl create namespace {name}
```

For example:

```
kubectl create namespace conversation
```

If you do not have the Kubernetes command line tool set up, see [Accessing your IBM Cloud Private cluster by using the kubectl CLI](#) for instructions.

2. Get a certificate from your IBM Cloud Private cluster and install it to Docker or add the `{cluster_CA_domain}` as a Docker Daemon insecure registry. You must do one or the other for Docker to be able to pull from your IBM Cloud Private cluster.

See [Specifying your own certificate authority \(CA\) for IBM Cloud Private services](#)

3. To load the file from Passport Advantage into IBM Cloud Private, enter the following command in the IBM Cloud Private command line interface.

```
bx pr load-ppa-archive --archive {compressed_file_name} --clusternamespace {cluster_CA_domain} --namespace {name}
```

- `{compressed_file_name}` is the name of the file that you downloaded from Passport Advantage.
- `{cluster_CA_domain}` is the IBM Cloud Private cluster domain, often referred to in this documentation as `{icp-url}`.
- `namespace` is the Docker namespace that hosts the Docker image that you created in Step 1.

4. View the charts in the IBM Cloud Private Catalog. From the IBM Cloud Private management console navigation menu, click **Manage > Helm Repositories**.

5. Click **Sync Repositories**.

You must have the *cluster administrator* user type or access level to sync repositories.

6. From the navigation menu, select **Catalog**.

7. Scroll to find the **ibm-watson-assistant-prod** package, and then click **Configure**.

8. Specify values for the configurable fields.

When you install the service, many configuration settings are applied to it for you unless you override them with your own values. **You cannot change these settings after you complete the installation.**

See [Configuration details](#) for help understanding the configuration choices. At a minimum, you must provide your own values for the following configurable settings:

- Helm release name
- Deployment Type
- Hostname of the IBM Cloud Private cluster Master node
- IP (v4) address of the master node
- Hostname of the IBM Cloud Private cluster proxy node
- Languages: If you do not need to support Czech, deselect it.
- Repository fields: Repository values have the syntax: {icp_url}:8500/{namespace-name}/{unique-path-value}. Edit the {icp_url} and {namespace-name} values to reflect your environment. Leave the {unique-path-value} (such as icp-wa-tooling-pathed) as is.

9. Click license agreements.

Click **Next** multiple times to read the full agreement, and then click **Accept** to accept the license terms.

10. Click **Install**.

Attention: You might see an error message (that begins with `Error making request: Error: ESOCKETTIMEDOUT POST`) during the installation process. However, you can ignore the message; the installation continues in the background. Give it time to complete. Check the Helm releases page for the status. See [Verify that the installation was successful](#).

Configuration details

Currently, the service does not support the ability to provide your own instances of resources, such as Postgres or MongoDB. There are configuration settings that suggest you can do so. However, do not change these settings from their default value of `true`.

Table 5. Configuration settings

Setting	Configuration settings	Description
Helm release name	A unique ID for this deployment. When you install the service from the command line, you set this value by using the <code>--name</code> parameter.	
Target namespace	Namespace used within the cluster to identify this service. Specify the namespace that you created earlier. The namespace must be <code>conversation</code> if you enable a language other than English and Czech.	
Deployment Type	Options are Development and Production . Development is a Private cloud environment that you can use for testing purposes. It contains a single pod for each microservice. Production is a Private cloud environment that you can use to host applications and services used in production. Contains two replicas of each microservice pod. Development is the default.	
Hostname of the ICP cluster Master node	Required. Specify the <code>cluster_CA_domain</code> hostname of the master node of your private cloud instance. For example: <code>my.company.name.icp.net</code> . Specify the hostname only, without a protocol prefix (<code>https://</code>) and without a port number (<code>:8443</code>). This unique URL is typically referred to as <code>{icp-url}</code> in this documentation. Corresponds to the <code>global.icp.masterHostname</code> value in the <code>values.yaml</code> file.	

Setting	Description
IP (v4) address of the master node	Required only if the hostname of the master node is <code>mycluster.icp</code> . Specify the IP address of the master node of your private cloud instance. Corresponds to the <code>global.icp.masterIP</code> value in the <code>values.yaml</code> file.
Hostname of the ICP cluster proxy node	Specify the hostname of the proxy node of your private cloud instance. To discover this value, run the command: <code>kubectl get nodes --show-labels</code> , and then find the node that shows <code>proxy=true</code> , and get the hostname value. It might be an IP address instead of a typical hostname. Corresponds to the <code>global.icp.proxyHostname</code> value in the <code>values.yaml</code> file.
Languages	Specify the languages you want to support in addition to. English is required; do not deselect it. For more information about language options, see Supported languages .
Create COS	Boolean. Indicates whether you want to provide your own cloud object store or have one created for you. If <code>true</code> , a Minio cloud object store is created. The default value is <code>true</code> . Do not deselect this checkbox. The service does not currently support providing your own store.
COS Access Key	Credential to access the store.
COS Secret Key	Access key to the store used by CLU components.
Bucket Prefix	Prefix of the bucket names to be used.
COS Access Protocol	Only used if Create COS is deselected. Specifies the protocol used to connect to COS. Options are <code>http</code> and <code>https</code> . Typical protocol is <code>http</code> .
COS Hostname	Only used if Create COS is deselected. Hostname to connect to the store. Typical hostname when COS is running in the cluster is <code>cos.namespace.svc.cluster.local</code> .
COS Port	Only used if Create COS is deselected. Port where COS is listening. Typical port is <code>443</code> .
Create Redis	Boolean. Specify <code>true</code> to have a Redis store created for you. Specify <code>false</code> to provide your own instance. The default value is <code>true</code> . Do not deselect this checkbox. The service does not currently support providing your own store.
Redis Hostname	Used only if Create Redis is deselected. Specifies the hostname of the running Redis service.
Redis Password	Password for accessing Redis. (User is root.)
Redis Port	Used only if Create Redis is deselected. Specifies the port from which the running Redis service can be accessed.
Create Postgres	Boolean. Specify <code>true</code> to have a Postgres store created for you. Specify <code>false</code> to provide your own instance. The default value is <code>true</code> . Do not deselect this checkbox. The service does not currently support providing your own store.
Postgres Hostname	Used only if Create Postgres is deselected. Specifies the hostname of the running Postgres service.

Setting	Description
Postgres Port	Used only if Create Postgres is deselected. Specifies the port from which the running Postgres service can be accessed.
Postgres Admin Name	User ID for a Postgres super user with rights to create databases and users in the Postgres database. If you use your own instance, then you must change this name and its associated password.
Postgres Admin Password	Password associated with the user ID for a Postgres super user with rights to create databases and users in the Postgres database. If you use your own instance, then you must change this password and the associated name.
Postgres Admin Database	Name of the database to connect to.
SSL Mode	SSL mode to use for the Postgres connection. Options such as verify-ca or verify-full are not currently supported by the store microservice. Currently, only SSL is supported. Do not change from the default value. The default value is allow .
Postgres Server Certificate	Used only if Create Postgres is deselected. Server certificate details.
Create Database	Boolean. Specify true to have the database and user created for you. If you specify false , then you must create the database and database user yourself. The default value is true . Do not deselect this checkbox. The service does not currently support providing your own database.
Create Schema	Boolean. Specify true to have the required tables, functions, and so on applied to the database that is created for you. The default value is true . Do not deselect this checkbox. The service does not currently support providing your own schema.
Store Database Name	Database name that the store microservice uses. If left empty, the default value " <i>conversation icp{{ .Release.Name }}</i> " is used.
Postgres User Name	User name that the store miroservice uses to connect to the Postgres database. If left empty, the default value " <i>storeicp{{ .Release.Name }}</i> " is used.
Postgres User Password	Password associated with the user ID specified in Postgres User Name.
Create Etcd Cluster	Boolean. Specify true to have the etcd cluster for the Watson Assistant service created for you. Specify false to provide your own etcd instance and the associated credentials. The default value is true . Do not deselect this checkbox. The service does not currently support providing your own cluster.
Enable Etcd Authentication	Boolean. If set to true the authentication is enabled in etcd. Watson Assistant requires authentication. Set to false only if you provide your own etcd cluster where authentication is already enabled. The default value is true .
Etcd User	User ID used to access etcd. The default value is root .
Etcd Password	Password associated with the Etcd User.

Setting	Description
Connections details for etcd	Used only if Create Etcd Cluster is deselected. Etcd connection details. If you do not specify a connection, these default connection details are used: schema:http, host:etcd-ibm-wcd-etcd.default.svc.cluster.local, port:2379. No connection details are specified by default.
Create MongoDB	Boolean. Specify true to have a MongoDB database created for you. Specify false to provide your own instance. The default value is true. Do not deselect this checkbox. The service does not currently support providing your own database.
MongoDB Hostname	Used only if Create MongoDB is deselected. Specifies the hostname of the running MongoDB database service.
MongoDB Port	Used only if Create MongoDB is deselected. Specifies the port from which the running MongoDB database service can be accessed.
Enable Authentication	Boolean. If true, authentication is enabled. The default value is true.
MongoDB Admin User	User ID for a MongoDB super user with rights to create databases and users in the MongoDB database. If you use your own instance, then you must change this name and its associated password.
MongoDB Admin Password	Password associated with the user ID for a MongoDB super user with rights to create databases and users in the MongoDB database. If you use your own instance, then you must change this password and the associated name.
Enable TLS	Boolean. Indicates whether to enable MongoDB TLS support. The default value is true.
base64 encoded certificate	Replace the certificate with your own. See the Configuration page for certificate details.
base64 encoded key	Replace the key with your own. See the Configuration page for key details.
Subdomain	URL from which to access the Watson Assistant tool. The url syntax is typically https://{{ subdomain }}.{{ icp-url }}/{{ applicationContext }}. The default value is assistant.
TLS Secret	Name of the secret that has the certificate and private key for the subdomain {{ subdomain }}.{{ icp-url }}. If empty, a secret with a self-signed certificate is created. By default, this setting is empty.

Attention: Do not select the **Install recommends** checkbox. It is not fully supported at the moment.

Step 5: Verify that the installation was successful

To check the status of the installation process:

1. Log in to the IBM Cloud Private management console.
2. From the main menu, expand **Workloads**, and then choose **Helm releases**.
3. Find the release name you used for the deployment in the list.

To run a test Helm chart:

1. From the Helm command line interface, run the following command:

```
helm test --tls {release name} --timeout 900
```

2. If one of the tests fails, review the logs to learn more. To see the log, use a command with the syntax `kubectl logs {testname} -n {name} -f --timestamps`. If you enable a language other than English and Czech, then you must specify `conversation` as the namespace name. For example:

```
kubectl logs my-release-bdd-test -n conversation -f --timestamps
```

3. To run the test script again, first delete the test pods by using a command with the syntax `kubectl delete pod {podname} --namespace {name}`. If you enable a language other than English and Czech, then you must specify `conversation` as the namespace name. For example:

```
kubectl delete pod my-release-bdd-test --namespace conversation
```

Uninstalling the service

If you need to start the deployment over, be sure to remove all trace of the current installation before you try to install again.

1. To uninstall and delete the `my-release` deployment, run the following command from the Helm command line interface:

```
helm delete --tls my-release
```

To irrevocably uninstall and delete the `my-release` deployment, run the following command:

```
helm delete --tls --no-hooks --purge my-release
```

If you omit the `--purge` option, Helm deletes all resources for the deployment but retains the record with the release name. This allows you to roll back the deletion. If you include the `--purge` option, Helm removes all records for the deployment so that the name can be used for another installation.

2. Remove all content from any persistent volumes that you used for the previous deployment before you restart the installation. See [Deleting a PersistentVolume](#) for more information.

3. The `PersistentVolumeClaims` will not be deleted and will remain bound to persistent volumes. You must remove them manually. See [Deleting a PersistentVolumeClaim](#) for details.

4. Delete the Helm chart.

```
bx pr delete-helm-chart --name ibm-watson-assistant-prod
```

5. Delete the namespace you used.

```
```bash kubectl delete namespace conversation
```

## Installing from the command line

If you have trouble when you install from the catalog, you can install by using the command line interface instead.

**Be sure to check that the installation from the catalog did not complete.** Even if you see an error message, the installation might complete successfully.

To install from the command line, complete these steps:

1. From the Kubernetes command line tool, create the namespace in which to deploy the service. If you enable a language other than English and Czech, then you must specify `conversation` as the namespace. Otherwise, you can use any namespace you choose. Use the following command to create the namespace:

```
kubectl create namespace {name}
```

For example:

```
kubectl create namespace conversation
```

If you do not have the Kubernetes command line tool set up, see [Accessing your IBM Cloud Private cluster by using the kubectl CLI](#) for instructions.

2. Get a certificate from your IBM Cloud Private cluster and install it to Docker or add the {cluster\_CA\_domain} as a Docker Daemon insecure registry. You must do one or the other for Docker to be able to pull from your IBM Cloud Private cluster.

See [Specifying your own certificate authority \(CA\) for IBM Cloud Private services](#)

3. To load the file from Passport Advantage into IBM Cloud Private, enter the following command in the IBM Cloud Private command line interface.

```
bx pr load-ppa-archive --archive {compressed_file_name} --clusternamespace {cluster_CA_domain} --namespace {name}
```

- {compressed\_file\_name} is the name of the file that you downloaded from Passport Advantage.
- {cluster\_CA\_domain} is the IBM Cloud Private cluster domain, often referred to in this documentation as {icp-url}.
- {name} is the Docker namespace that hosts the Docker image. This is the namespace you created in Step 1.

4. Download the chart from [https://{{cluster\\_CA\\_domain}}:8443/helm-repo/requiredAssets/ibm-watson-assistant-prod-1.0.1.tgz](https://{{cluster_CA_domain}}:8443/helm-repo/requiredAssets/ibm-watson-assistant-prod-1.0.1.tgz).

5. Extract the TAR file from the TGZ file, and then extract files from the TAR file.

6. Edit values in the values.yaml file.

To do so, first make a copy of the values.yaml. The values.yaml file is stored with the chart. Rename the file. For example, my-override.yaml.

In your copy of the file, remove all but the configuration settings that you want to replace with your own values.

Edit the Docker image repository values in the file with values that reflect your environment. Each image repository value must have the syntax {icp\_url}:8500/{namespace-name}/{unique-path-value} where {unique-path-value} reflects the path from the existing YAML file, such as icp-wa-tooling-pathed.

At a minimum, you must provide your own values for the following configurable settings also:

- global.deploymentType: Specify whether you want to set up a development or production instance.
- global.icp.masterHostname: Specify the hostname of the master node of your private cloud instance. Do not include the protocol prefix (https://) or port number (:8443). For example: my.company.name.icp.net.
- global.icp.masterIP: If you did not define a domain name for the master node of your private cloud instance, you are using the default hostname mycluster.icp, for example, then you must also specify this IP address.
- global.icp.proxyHostname: Specify the hostname (or IP address) of the proxy node of your private cloud instance.

**Attention:** Currently, the service does not support the ability to provide your own instances of resources, such as Postgres or MongoDB. The values YAML file has {resource-name}.create settings that suggest you can do so. However, do not change these settings from their default value of true.

7. After you define any custom configuration settings, you can install the chart from the Helm command line interface.

Enter the following command from the directory where the package was loaded in your local system:

```
helm install --tls --values {override-file-name} --namespace {name} --name {my-release}
ibm-watson-assistant-prod
```

- Replace {my-release} with a name for your release.
- Replace {override-file-name} with the path to the file that contains the values that you want to override from the values.yaml file provided with the chart package. For example: ibm-watson-assistant-prod/my-override.yaml
- Replace {name} with the namespace you created for the service. If you enable a language other than English and Czech, then the namespace must be set to conversation.
- The ibm-watson-assistant-prod parameter represents the name of the Helm chart that you downloaded and extracted. Alternatively, you can specify the name of the downloaded chart by using ibm-watson-assistant-prod-1.0.1.tgz as the value instead.

After the installation finishes, [verify](#) that it was successful.

## Step 6: Launch the tool

1. Log in to the IBM Cloud Private management console.
2. From the main menu, expand **Workloads**, and then choose **Deployments**.
3. Find the deployment named {release-name}-ui.

If you don't know the {release-name}, filter the list of deployments to include only those associated with the namespace you used for the service, and then search on -ui to find it.

4. Click **Launch**.

A new web browser tab opens and shows the Watson Assistant tool login page. The tool URL has the syntax {{ ui.subdomain }}.{{ global.masterHostname }}. For example:  
<https://assistant.assistant.mycluster.icp>.

5. Log in using the same credentials you used to log into the IBM Cloud Private dashboard.

## Next steps

Use the Watson Assistant tool to build training data and a dialog that can be used by your assistant.

- To learn more about the service first, read the [overview](#).
- To see how it works for yourself, follow the steps in the [getting started tutorial](#).

# Getting started tutorial

In this short tutorial, we introduce the Watson Assistant tool and go through the process of creating your first assistant.

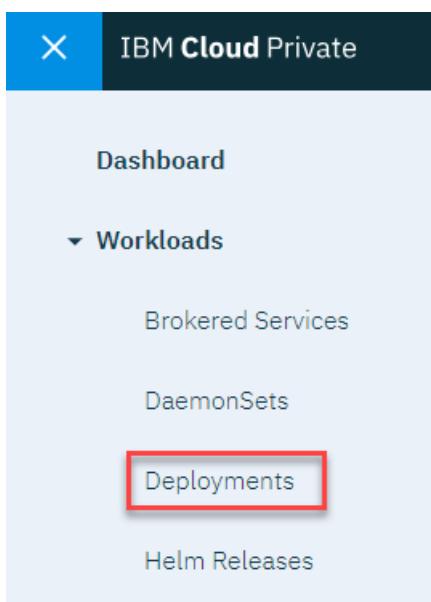
## Before you begin

You'll need a service instance to start. The administrator must install the service for you. Details are provided in the [installation checklist](#).

If a Watson Assistant service instance is already set up in your IBM® Cloud Private Cloud environment, then you're all set with these prerequisites. Go to [Step 1](#).

## Step 1: Open the tool

1. Log in to the IBM Cloud Private management console.
2. From the main menu, expand **Workloads**, and then choose **Deployments**.



3. Filter the list of deployments by the namespace you specified for your instance.

The screenshot shows a dark-themed dashboard with a navigation bar at the top containing links for 'Create resource', 'Catalog', 'Docs', and 'Support'. Below the navigation bar is a sidebar titled 'All namespaces' with a dropdown arrow. A list of namespaces is displayed, including 'All namespaces', 'conversation' (which is highlighted with a red box), 'default', 'istio-system', and 'kube-public'. At the bottom of the sidebar, there are buttons for '1 of 1 p' and 'LAUNCHED 1 hour ago', followed by a 'Launch' button.

4. Find the deployment named `{release-name}-ui`.

If you don't know the `{release-name}`, search on `-ui` to find it.

5. Click **Launch**.

A new web browser tab opens and shows the Watson Assistant tool login page.

6. Log in using the same credentials you used to log into the IBM Cloud Private dashboard.

## Step 2: Create a workspace

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

A workspace is a container for the artifacts that define the conversation flow.

1. From the home page of the Watson Assistant tool, click the **Workspaces** tab.

2. Click **Create**.

The screenshot shows the Watson Assistant tool's interface. At the top, there is a header with the IBM Watson Assistant logo and a user profile icon. Below the header, there are tabs for 'Home' and 'Workspaces', with 'Workspaces' being the active tab. On the left, there is a list of existing workspaces, including 'Customer Service - Sample'. To the right of this list, there is a 'Create' button and a small 'Up' arrow icon. A modal window is open in the center, prompting the user to 'Create a new workspace'. The modal contains text explaining that workspaces enable maintaining separate intents, user examples, entities, and dialogs for each use or application. It also states that 0 of 5 available workspaces have been used. At the bottom of the modal is another 'Create' button with a plus sign icon. The bottom of the screen features an 'IBM' logo and a 'Cookie Preferences' link.

3. 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**. You'll land on the **Intents** tab of your new workspace.

The screenshot shows the IBM Watson Assistant interface. The top navigation bar includes 'IBM Watson Assistant', 'Home', 'Workspaces', and user profile icons. Below the navigation is a 'Workspaces' section with a 'Create' button. A modal window titled 'Create a new workspace' is displayed, containing text about workspaces and a 'Create' button. At the bottom right of the main workspace area, there is a 'Try out the latest features' button.

## Step 3: 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 **General** content catalog so your dialog can greet users, and end conversations with them.

1. In the Watson Assistant tool, click the **Content Catalog** tab.
2. Find **General** in the list, and then click **Add to workspace**.

The screenshot shows the 'Content Catalog' tab selected in the Watson Assistant tool. The page displays a list of categories with their descriptions and intent counts. The 'General' category is highlighted with a red box, and its 'Add to workspace' button is also highlighted with a red box.

Category	Description	Intents	Action
Banking	Basic transactions for a banking use case.	13	+ Add to workspace
Bot Control	Functions that allow navigation within a conversation.	9	+ Add to workspace
Customer Care	Understand and assist customers with information about themselves and your business.	18	+ Add to workspace
eCommerce	Payment, billing, and basic management tasks for orders.	14	+ Add to workspace
General	General conversation topics most users ask.	10	+ Add to workspace
Insurance	Issues related to insurance policies and claims.	12	+ 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 `#General_`. You will add the `#General_Greetings` and `#General_Ending` intents to your dialog in the next step.

IBM Watson Assistant

Workspaces / Watson Assistant tutorial / Build

Add intent  Try it

<input type="checkbox"/>	Intent (10) ▾	Description	Modified ▾	Examples
<input type="checkbox"/>	#General_About_You	Request generic personal attributes.	a few seconds ago	20
<input type="checkbox"/>	#General_Capabilities	Request capabilities of the bot.	a few seconds ago	31
<input type="checkbox"/>	#General_Connect_to_Agent	Request a human agent.	a few seconds ago	38
<input type="checkbox"/>	#General_Ending	End the conversation.	a few seconds ago	37
<input type="checkbox"/>	#General_Greetings	Greet the bot.	a few seconds ago	27
<input type="checkbox"/>	#General_Human_or_Bot	Ask if speaking to a human or a bot.	a few seconds ago	12
<input type="checkbox"/>	#General_Jokes	Request a joke.	a few seconds ago	17
<input type="checkbox"/>	#General_Negative_Feedback	Express unfavorable feedback.	a few seconds ago	20
<input type="checkbox"/>	#General_Positive_Feedback	Express positive sentiment or gratitude.	a few seconds ago	19
<input type="checkbox"/>	#General_Security_Assurance	Express concerns about the security of the bot.	a few seconds ago	26

You have successfully started to build your training data by adding prebuilt content from IBM to your workspace.

## Step 4: Build a dialog

A [dialog](#) 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 the #General\_Greetings and #General\_Ending intents, each with a single node.

1. In the Watson Assistant tool, click the **Dialog** tab.

IBM Watson Assistant

Workspaces / Watson Assistant tutorial / Build

Try it

<input type="checkbox"/>	Entities	<b>Dialog</b>	Content Catalog
--------------------------	----------	---------------	-----------------

No dialog yet

A dialog uses intents, entities, and context from your application to define a response to each user's input. Creating a dialog defines how your virtual assistant will respond to what your users are saying.

Create 

2. Click **Create**.

You'll see two nodes:

- o **Welcome**: Contains a greeting that is displayed to your users when they first engage with the bot.
- o **Anything else**: Contains phrases that are used to reply to users when their input is not recognized.

## IBM Watson Assistant

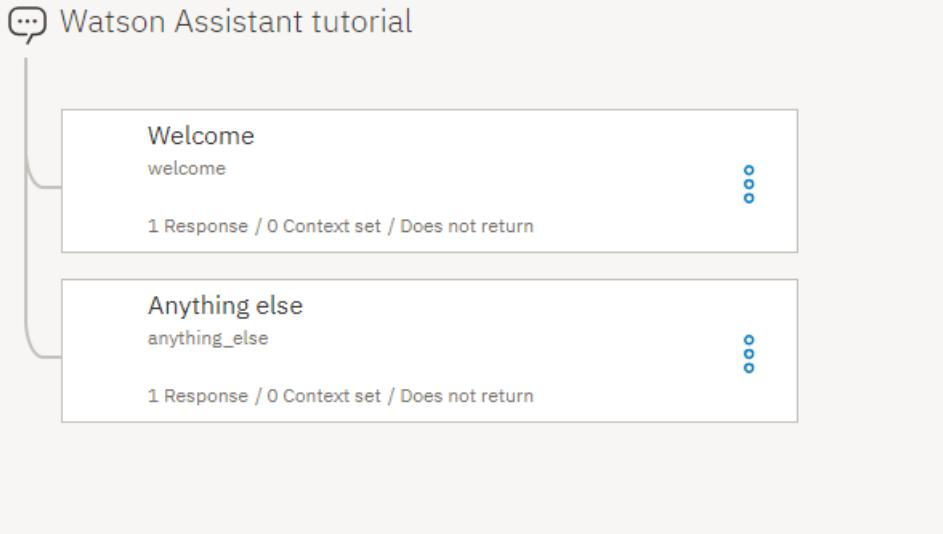
[Workspaces](#) / Watson Assistant tutorial / Build

Intents Entities **Dialog** Content Catalog

Add node

Add child node

Add folder



3. Click the **Welcome** node to open it in the edit view.

4. Replace the default response with the text, **Welcome to the Watson Assistant tutorial!**.

IBM Watson Assistant

[Workspaces](#) / Watson Assistant tutorial / Build

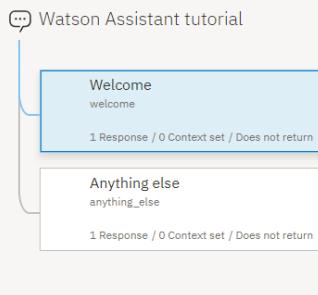


Try it

Intents Entities **Dialog** Content Catalog

Add node

Add child node



Welcome

Customize



If bot recognizes:

welcome - +

Then respond with:

1. Welcome to the Watson Assistant tutorial!|



Add a variation to this response

5. Click X to close the edit view.

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

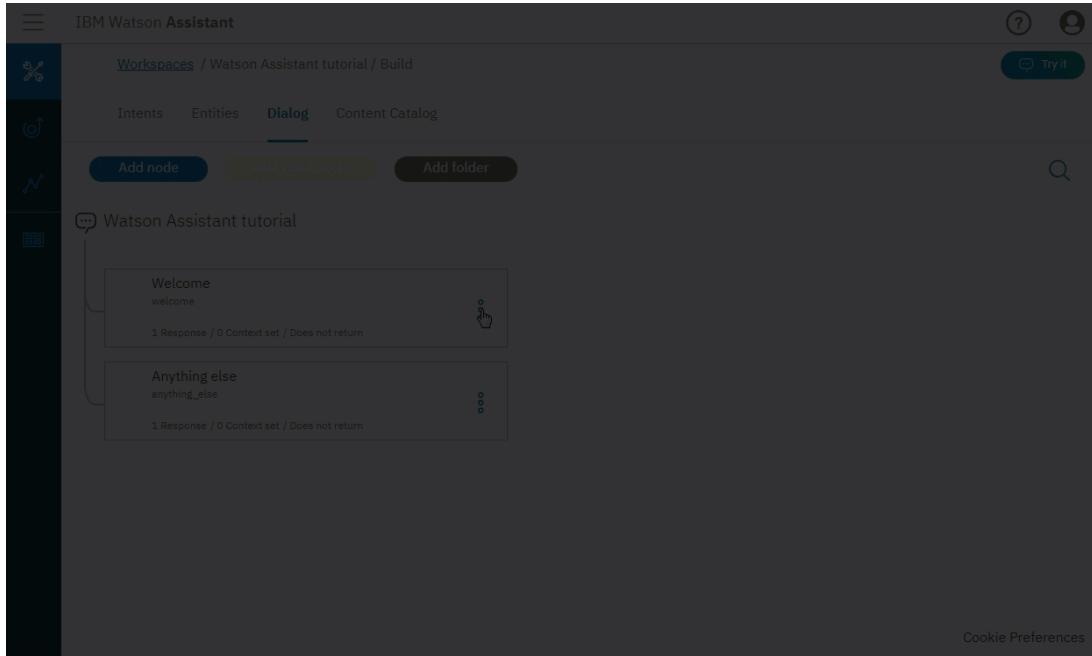
6. Click the More icon ••• on the **Welcome** node, and then select **Add node** below.

7. Type **#General\_Greetings** in the **Enter a condition** field of this node. Then select the **#General\_Greetings**

option.

#### 8. Add the response, Good day to you!

9. Click  to close the edit view.



The screenshot shows the IBM Watson Assistant interface. The top navigation bar includes 'IBM Watson Assistant', 'Workspaces / Watson Assistant tutorial / Build', and a 'Try it' button. Below the navigation are tabs for 'Intents', 'Entities', 'Dialog' (which is selected), and 'Content Catalog'. A toolbar at the top has buttons for 'Add node', 'Edit node', and 'Add folder'. A search bar is on the right. The main area displays a dialog tree under the workspace 'Watson Assistant tutorial'. It contains two nodes: 'Welcome' (with condition 'welcome') and 'Anything else' (with condition 'anything\_else'). Each node has a 'More' icon (three dots) in its bottom right corner. A tooltip for the 'More' icon indicates '1 Response / 0 Context set / Does not return'.

10. Click the More icon  on this node, and then select **Add node below** to create a peer node. In the peer node, specify `#General_Ending` as the condition, and `OK. See you later.` as the response.

11. Click  to close the edit view.

Intents Entities **Dialog** Content Catalog**Add node****Add child node****Add folder**

Watson Assistant tutorial

- Welcome  
welcome  
1 Response / 0 Context set / Does not return
- #General\_Greetings  
1 Response / 0 Context set / Does not return
- #General\_Ending  
1 Response / 0 Context set / Does not return
- Anything else  
anything\_else  
1 Response / 0 Context set / Does not return

## Step 5: Test the dialog

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  icon to open the "Try it out" pane. The welcome message that you added is displayed.

[Workspaces](#) / Watson Assistant tutorial / BuildIntents Entities **Dialog** Content Catalog[Add node](#)[Add child node](#)[Add folder](#)

Watson Assistant tutorial

Welcome  
welcome

1 Response / 0 Context set / Does not return

Try it out [Clear](#) [Manage Context](#) [X](#)Welcome to the Watson Assistant  
tutorial!

2. At the bottom of the pane, type Hello and press Enter. The output indicates that the #General\_Greetings intent was recognized, and the appropriate response (Good day to you.) appears.

3. Try the following input:

- bye
- howdy
- see ya
- good morning
- sayonara

Try it out [Clear](#) [Manage Context](#) [X](#)

Welcome to the Watson Assistant  
tutorial!

hell

Watson can recognize your intents even when your input doesn't exactly match the examples you included. The dialog uses

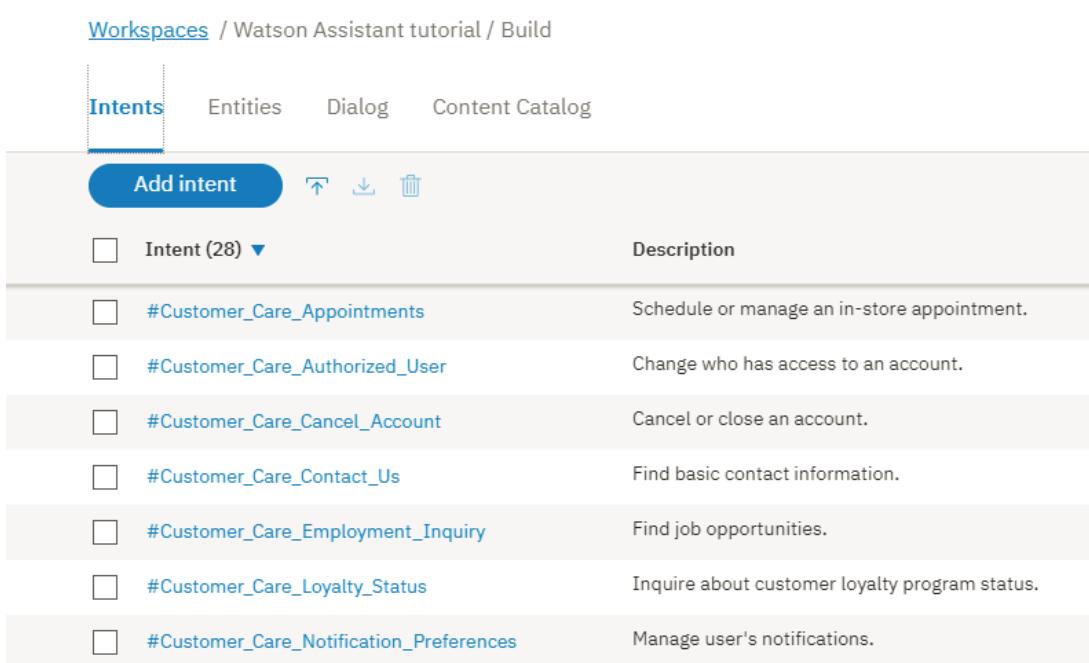
intents to identify the purpose of the user's input regardless of the precise wording used, and then responds in the way you specify.

## Step 6: Add a business function to the dialog

Add the *Customer Care* content catalog to your training data, so your dialog can address user requests for contact information.

1. In the Watson Assistant tool, click the **Content Catalog** tab.
2. Find **Customer Care** 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 `#Customer_Care_`. You will add the `#Customer_Care_Contact_Us` intent to your dialog in a later step.

### IBM Watson Assistant



The screenshot shows the IBM Watson Assistant interface with the 'Intents' tab selected. At the top, there are tabs for 'Workspaces' (selected), 'Watson Assistant tutorial / Build', 'Entities', 'Dialog', and 'Content Catalog'. Below the tabs, there is a button 'Add intent' and icons for sorting. A table lists 28 intents, each with a checkbox and a description:

Intent (28) ▾	Description
<input type="checkbox"/> #Customer_Care_Appointments	Schedule or manage an in-store appointment.
<input type="checkbox"/> #Customer_Care_Authorized_User	Change who has access to an account.
<input type="checkbox"/> #Customer_Care_Cancel_Account	Cancel or close an account.
<input type="checkbox"/> #Customer_Care_Contact_Us	Find basic contact information.
<input type="checkbox"/> #Customer_Care_Employment_Inquiry	Find job opportunities.
<input type="checkbox"/> #Customer_Care_Loyalty_Status	Inquire about customer loyalty program status.
<input type="checkbox"/> #Customer_Care_Notification_Preferences	Manage user's notifications.

4. Open the **Dialog** tab. Click the More icon  on the `#General_Greetings` node, and then select **Add node below** to create a peer node. In the peer node, specify `#Customer_Care_Contact_Us` as the condition.

5. Add the following text as the response:

Contact us by phone 24 hours a day, 7 days a week at 958-234-3456. To give us feedback, submit a feedback form through our <a href="<https://www.example.com/feedback.html>" target="\_blank">website</a>.

Intents Entities Dialog Content Catalog

Add node Add child node

Watson Assistant tutorial

- Welcome welcome  
1 Response / 0 Context set / Does not return
- #General\_Greetings  
1 Response / 0 Context set / Does not return
- #Customer\_Care\_Contact\_Us  
1 Response / 0 Context set / Does not return
- #General\_Ending  
1 Response / 0 Context set / Does not return

Name this node...

If bot recognizes:  
#Customer\_Care\_Contact\_Us

Then respond with:

Text

Contact us by phone 24 hours a day, 7 days a week at 958-234-3456. To give us feed

Enter response variation

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

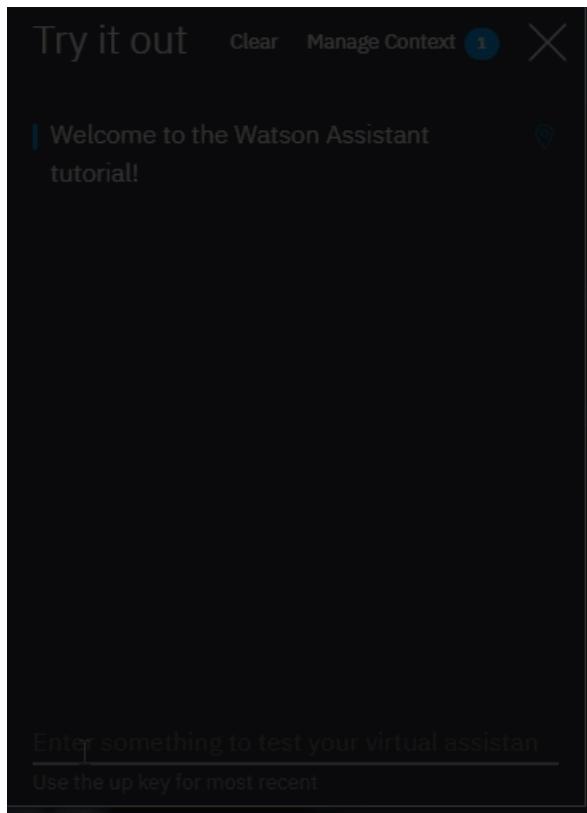
6. Click to close the edit view.

Watson Assistant tutorial

- Welcome welcome  
1 Response / 0 Context set / Does not return
- #General\_Greetings  
1 Response / 0 Context set / Does not return
- #Customer\_Care\_Contact\_Us  
1 Response / 0 Context set / Does not return
- #General\_Ending  
1 Response / 0 Context set / Does not return
- Anything else anything\_else  
1 Response / 0 Context set / Does not return

7. Test the node you just added. Click the icon to open the "Try it out" pane, and then enter How can I contact you?

The `#Customer_Care_Contact_Us` intent is recognized, and the response that you specified for it to return is displayed.



You have successfully added a node to the dialog that addresses the type of business-related question that real users might ask.

## Step 7: 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 more complex dialog.

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

## Next steps

This tutorial is built around a simple example. For a real application, you'll need to define some more interesting intents, some entities, and a more complex dialog.

- Try the advanced [tutorial](#) to add entities and clarify a user's purpose.
- Check out the [sample apps](#).

## Video: Tool overview for Watson Assistant

With the Watson Assistant tool, you can create and manage your Watson Assistant workspace. Watch this video to learn about different areas of the tool.

**Note:** The overview discusses how you can use metrics that are collected to improve the training data. Metrics are included only in the public cloud version of the service right now.

# Additional resources

Links to resources, videos, tutorials, and articles that can help you to get started faster and go further.

## Developer resources

- [IBM Code Bot Asset Exchange](#)
- [Retail chat app](#)
- [Investment management chat app](#)
- [Cognitive Banking chatbot](#)
- [Pizza ordering chatbot using slots](#)
- [Building an Amazon Alexa Skill with the service](#)
- [Adding a chatbot to WordPress with the service](#)
- [Tutorial: Build a database-driven Slackbot](#)
- [Tutorial: Build a voice-enabled Android chatbot](#)

## Building with Watson videos

From an IBM technical deep-dive series. These recordings provide step-by-step instructions and answers.

**Note:** Excuse the references to the legacy service name (IBM Watson Conversation) that these videos use to refer to the service. The principles described in them remain true for Watson Assistant.

- [Advanced integrations](#)
- [Training the service to detect user intent](#)
- [New tools for dialog scripting](#)

## Other videos

Additional videos created by IBMers. Again, some of these videos refer to the service by its former name, the Watson Conversation service. However, they continue to be relevant for Watson Assistant.

- [Digital Technical Enablement](#): A collection of resources, including a product tour and hands-on lab.
- [Show me how to build conversational skills](#): A 3-minute overview of the service.
- [Walkthrough of the tool](#): A 14-minute technical walkthrough.

## IBM Watson Academy

- [Watson Academy](#)

## Simon Burns on Medium

A set of posts by Simon Burns, User Experience Developer at IBM Watson™

- [Getting Chatty with IBM Watson](#)
- [Chatbot: Deconstructed](#)
- [The Bots Are Coming](#)
- [Conversation Patterns with IBM Watson](#)

- [Seriously, what do I need a bot for?](#)
- [Purple Brain](#)
- [Conversational Design](#)
- [Bot Feedback with IBM Watson](#)

# **Watson Assistant for IBM Cloud Private: How To**

# 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:** 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:** 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:** 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

### Workspaces per service instance

50

The sample workspace does not count toward your workspace limit unless you edit or duplicate the sample.

## 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.

1. [Launch the tool](#).
2. Click the **Workspaces** tab.
3. From the tool, 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  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](#).)
- 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.

**Note:** If you get an error message when importing a JSON file for a workspace that you created in the public cloud environment, then it might mean that your workspace uses features that are not supported in the private cloud environment. For example, it might contain @sys-person or @sys-location references, which are system entities that are available in public cloud only.

#### 4. 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.

You can export a workspace by using the API also. Include the `export=true` parameter with the request. See the [API reference](#) for more details.

# Using content catalogs

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

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.

The screenshot shows the Watson Assistant interface with the 'Content Catalog' tab selected. A message at the top says: 'Get started faster by adding existing intents from the content catalog. These intents are trained on common questions that users may ask.' Below is a table listing categories and their details:

Category	Description	Intents	Action
Banking	Basic transactions for a banking use case.	13	Add to workspace
Bot Control	Functions that allow navigation within a conversation.	9	Add to workspace
Customer Care	Understand and assist customers with information about themselves and your business.	18	Add to workspace
eCommerce	Payment, billing, and basic management tasks for orders.	14	Add to workspace
General	General conversation topics most users ask.	10	Add to workspace
Insurance	Issues related to insurance policies and claims.	12	Add to workspace
Telco	Questions and issues related to a user's telephony service, device, and plan.	21	Add to workspace
Utilities	Help a user with utility emergencies and their utility service.	10	Add to workspace

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

The screenshot shows the 'Banking' category details. It includes a description: 'Basic transactions for a banking use case.' Below is a table with intents and examples:

Description (13)	Intent	Examples
View the activity on an account.	#Banking_View_Activity	Can I get statement for account activities in past 10 days? Do you know about account activities of my checking account? From where did I get the latest credit in my account 17 more examples...
Cancel a card.	#Banking_Cancel_Card	What is the process for closing credit card? What is the procedure for credit card deactivation? What are the ways for closing credit card? 17 more examples...
Activate a card.	#Banking_Activate_Card	When will my credit card begin working? What should I do to setup my credit card for online payments? What is the process to activate credit card after cancellation? 17 more examples...

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

3. Select to return to the **Content Catalog** tab.

4. 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.

Workspaces / Car Dashboard / Build

Banking added: 13 intents have been added to your workspace

Try it

Intents Entities Dialog Content Catalog

Get started faster by adding existing intents from the content catalog. These intents are trained on common questions that users may ask.

Category	Description	Intents	Action
Banking	Basic transactions for a banking use case.	13	<button>Add to workspace</button>
Bot Control	Functions that allow navigation within a conversation.	9	<button>Add to workspace</button>
Customer Care	Understand and assist customers with information about themselves and your business.	18	<button>Add to workspace</button>
eCommerce	Payment, billing, and basic management tasks for orders.	14	<button>Add to workspace</button>

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

Workspaces / Car Dashboard / Build

Try it

Intents Entities Dialog Content Catalog

Add intent

	Description	Modified	Examples
<input type="checkbox"/> Intent (102) ▾			
<input type="checkbox"/> #Banking_Cancel_Card	Cancel a card.	2 minutes ago	20
<input type="checkbox"/> #Banking_Fee_Inquiry	Inquire about fees associated with a card.	2 minutes ago	20
<input type="checkbox"/> #Banking_Replace_Card	Replace a card.	2 minutes ago	20
<input type="checkbox"/> #Banking_Report_Missing_Card	Report a lost or stolen card.	2 minutes ago	20
<input type="checkbox"/> #Banking_Request_Card_Member_Agreement	Request a card member agreement.	2 minutes ago	20
<input type="checkbox"/> #Banking_Request_Checkbook	Request a checkbook.	2 minutes ago	20

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**.

## Intent name

#Banking\_Cancel\_Card

## Description

Cancel a card.

## Add user examples

Add user examples to this intent

[Add example](#)[Move... ↪](#) [Delete](#) 1 item selected [Cancel](#) User examples (20) ▾ Annual charges are high hence want to cancel my credit card Can I cancel a credit card I just applied for? Can I cancel my credit card if I have some pending dues? Cancel a card Details on how to apply for cancellation of credit card

# Defining intents

**Intents** 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.

Note: The video was created in a test environment in which the workspace is referred to as a skill.

## Intent creation overview

- Plan the intents for your application.

Consider what your customers might want to do, and what you want your application to be able to handle on their behalf. For example, you might want your application to help your customers make a purchase. If so, you can add a #buy\_something intent. (The # prepended to the intent name helps to clearly identify it as an intent.)

- Teach Watson about your intents.

Once you decide which business requests you want your application to handle for your customers, you must teach Watson about them. For each business goal (such as #buy\_something), you must provide at least 10 examples of utterances that your customers typically use to indicate their interest in that goal. For example, I want to make a purchase. Ideally, go and find real-world user examples from existing business processes. The user examples should be tailored to your specific business. For example, if you are an insurance company, your user examples might look more like this, I want to buy a new XYZ insurance plan. These examples are used by the service to build a machine learning model that can recognize the same and similar types of utterances and map them to the appropriate intent.

Start with a few intents, and test them as you iteratively expand the scope of the application.

## Intent limits

## Intent limits

### Intents per workspace Examples per workspace

2,000                  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 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.

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

The screenshot shows the 'Create new intent' interface. At the top left is a back arrow and the text 'Create new intent'. At the top right are two icons: a downward arrow and a trash bin. Below this is a form with two fields: 'Intent name' containing '#pay\_bill' and 'Description' containing 'Bill payment intent'. A blue 'Create intent' button is at the bottom left. Below the button, the text 'No examples' is displayed, followed by a note: 'Train your bot with this intent by adding unique examples of what your users would say.'

5. 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 entity values and synonyms in intent examples

If you have defined, or plan to define, entities that are related to this intent, mention the entity values or synonyms in some of the examples. Doing so helps to establish a relationship between the intent and entities.

Intent name  
#pay\_bill

Description  
Bill payment intent

Add user examples  
Pay my account balance

Add example Remove examples

User examples ▾

I need to pay my bill

You can also add entity annotations directly from user examples. See [Defining contextual 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 name in 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.

Intent name  
#order\_phone

Description  
Order a phone intent

Add user examples  
Can I get a @PhonemodelName?

Add example

User examples ▾

Add a @PhonemodelName to my order

Help me order a @PhonemodelName

Is the @PhonemodelName in stock?

**Note:** Currently, you can only directly reference synonym entities that you define (pattern values are ignored). You cannot use [system entities](#).

**Important:** 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 @ServiceCarrier 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 @Entity references.

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.

6. Click **Add example** to save the example.

7. 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.

8. When you have finished adding examples, select to finish creating the intent.

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**.

Move ↪   Delete

1 item selected   Cancel

User examples ▾

I need to pay my bill

Pay my account balance

There is a problem with my bill

## 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:

<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).

2. 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 to open the page.

3. Select the *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.

Intents Entities Dialog

Add intent   

<input type="checkbox"/> Intent ▾	Description	Modified	Examples
<input type="checkbox"/> #about_VA		5 months ago	35
<input type="checkbox"/> #capabilities		5 months ago	120
<input type="checkbox"/> #compound_questions		5 months ago	36
<input type="checkbox"/> #decision_replies		5 months ago	10
<input type="checkbox"/> #goodbyes		5 months ago	50
<input type="checkbox"/> #greetings		5 months ago	48

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

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*.

Intents Entities Dialog

  1 item selected Cancel

<input checked="" type="checkbox"/> Intent ▾	Description	Modified ▾	Examples
<input checked="" type="checkbox"/> #about_VA		5 months ago	33
<input type="checkbox"/> #capabilities		5 months ago	120
<input type="checkbox"/> #compound_questions		5 months ago	36
<input type="checkbox"/> #decision_replies		5 months ago	10
<input type="checkbox"/> #goodbyes		5 months ago	50
<input type="checkbox"/> #greetings		5 months ago	48
<input type="checkbox"/> #improving_system		5 months ago	8
<input type="checkbox"/> #information_request		5 months ago	14

## 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*.

Intents			
<input type="button" value="Export"/>	<input style="border: 2px solid red; color: red; background-color: white; padding: 2px 5px; border-radius: 5px;" type="button" value="Delete"/>	1 item selected	<input type="button" value="Cancel"/>
	Description	Modified ▾	Examples
<input type="checkbox"/>	#about_VA	5 months ago	33
<input checked="" type="checkbox"/>	#capabilities	5 months ago	120
<input type="checkbox"/>	#compound_questions	5 months ago	36
<input type="checkbox"/>	#decision_replies	5 months ago	10
<input type="checkbox"/>	#goodbyes	5 months ago	50
<input type="checkbox"/>	#greetings	5 months ago	48
<input type="checkbox"/>	#improving_system	5 months ago	8
<input type="checkbox"/>	#information_request	5 months ago	14

## 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 **Try it** icon.

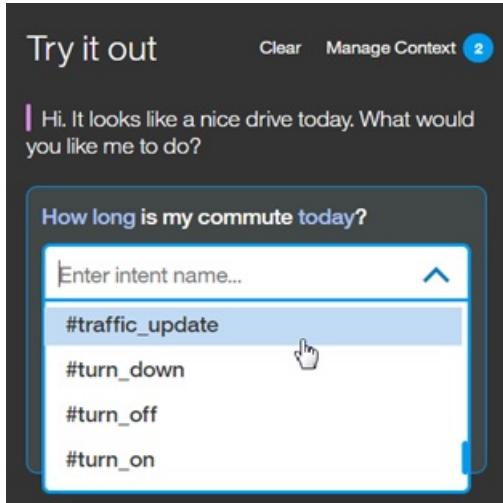
2. In the *Try it out* pane, enter a question or other text string and press Enter to see which intent is recognized. If the wrong intent is recognized, you can improve your model by adding this text as an example to the correct intent.

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:

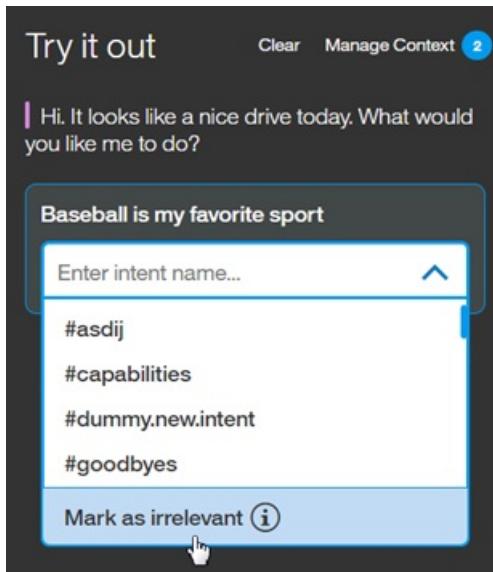
The response indicates which intent was recognized from your input.

3. 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.



4. 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](#).

### 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](#) for the availability of this feature.

After you upgrade your workspace, you can [test input](#) 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.

# Building a dialog

## Dialog overview

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

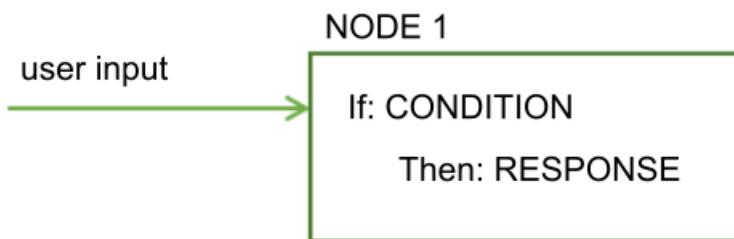
The dialog matches intents (what users say) to responses (what the bot says back). The response might be the answer to a question such as `Where can I get some gas?` or the execution of a command, such as turning on the radio. The intent and entity might be enough information to identify the correct 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.

Note: The video is 15 minutes in duration; the first 5 minutes cover how to add a node. It was created in a test environment in which the workspace is referred to as a skill.

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

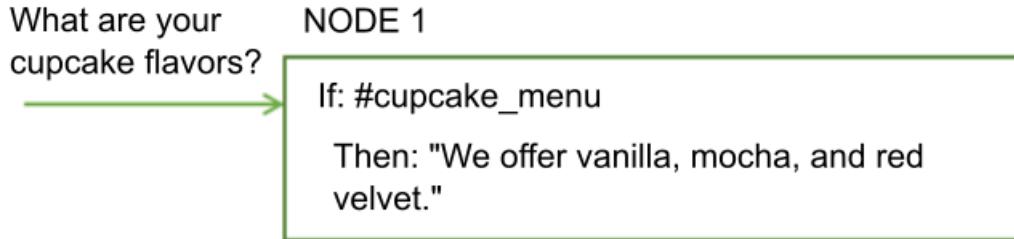
Each dialog node contains, at a minimum, a condition and a response.



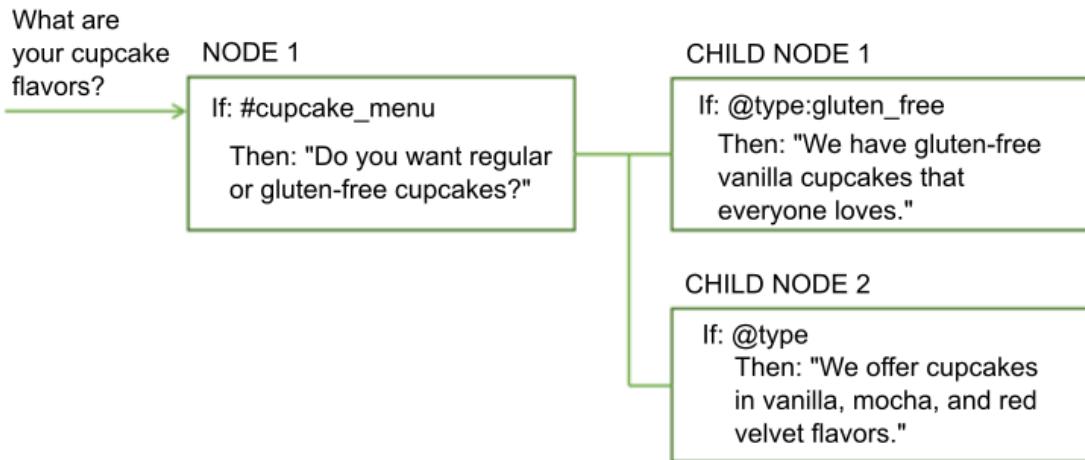
- Condition: Specifies the information that must be present in the user input for this node in the dialog to be triggered. The information is typically a specific intent. It might also be an entity type, an entity value, or a context variable value. See [Conditions](#) for more information.
- Response: The utterance that the service uses to respond to the user. The response can also be configured to show an image or a list of options, or to trigger programmatic actions. See [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.

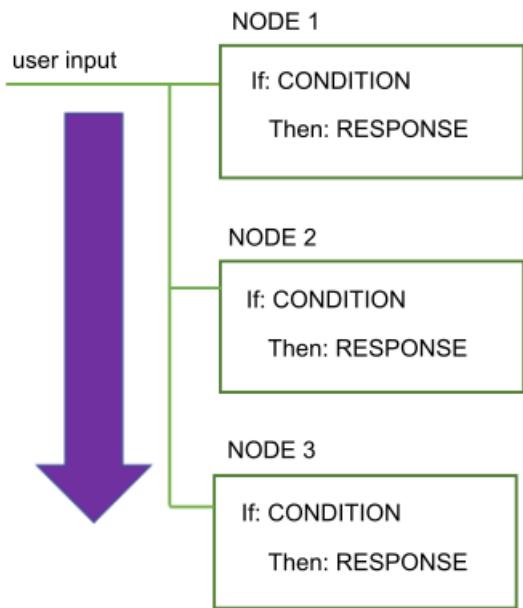


A single node with one condition and response can handle simple user requests. But, more often than not, users have more sophisticated questions or want help with more complex tasks. You can add child nodes that ask the user to provide any additional information that the service needs.



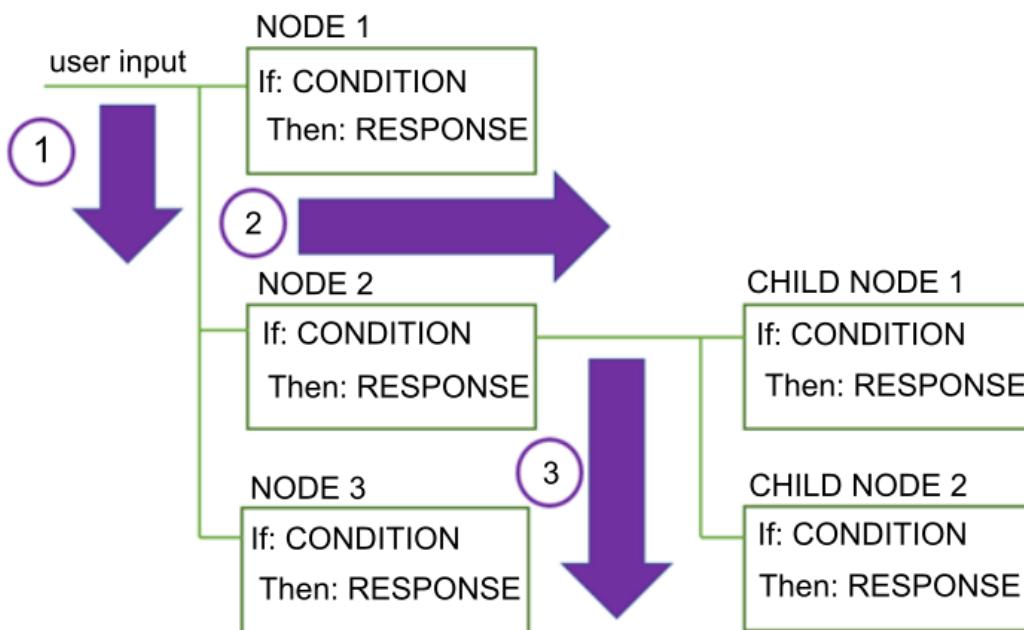
## Dialog flow

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 service 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 in the following ways:

- 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](#) for more information.
- By configuring conditional responses to jump to other nodes. See [Conditional responses](#) for more information.
- By configuring digression settings for dialog nodes. Digressions can also impact how users move through the nodes at run time. If you enable digressions away from most nodes and configure returns, users can jump from one node to another and back again more easily. See [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](#)
- [Special conditions](#)
- [Condition syntax details](#)

For tips on performing more advanced actions in conditions, see [Condition usage tips](#).

### 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 node conditions, this artifact type is typically used with an AND or OR operator and another condition value. That's because something in the user input must trigger the node; the context variable value being matched alone is not enough to trigger it. If the user input object sets the context variable value somehow, for example, then the node is triggered.

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

For response conditions, this artifact type can be used alone. You can change the response based on a specific context variable value. For example, `$city:Boston` checks whether the `$city` context variable contains the value, Boston. If so, the response is returned.

For more information about context variables, see [Context variables](#).

- **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.

Consider creating 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](#).

- **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 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.

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 and recognizing entity pattern groups in input](#) for more information.

- **Intent:** The simplest condition is a single intent. The node is used if, after the service's natural language processing evaluates the user's input, it determines that the purpose of the user's input maps to the pre-defined intent. Use the syntax, #intent\_name. For example, #weather checks if the user input is asking for a weather forecast. If so, the node with the #weather intent condition is processed.

For more information about intents, see [Defining intents](#).

- **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

### 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 <b>Anything else</b> node is triggered by this condition.
conversation_start	Like <b>welcome</b> , this condition is evaluated as true during the first dialog turn. Unlike <b>welcome</b> , it is true whether or not the initial request from the application contains user input. A node with the <b>conversation_start</b> 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 <b>Jump to</b> 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 <b>Welcome</b> 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.

## 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](#).
- Spring Expression (SpEL) language, which is an expression language that supports querying and manipulating an object graph at run time. See [Spring Expression Language \(SpEL\) language](#) 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](#) for more details.

## Responses

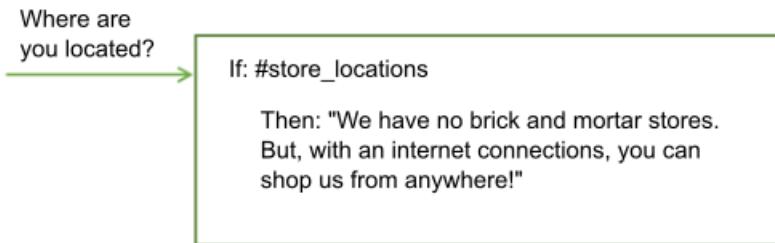
The dialog response defines how to reply to the user.

You can reply in the following ways:

- [Simple text response](#)
- [Rich responses](#)
- [Conditional responses](#)

### 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 `$variable_name` to specify it. See [Context variables](#) for more information. For example, if you know that the `$user` context variable is set to the current user's name before a node is processed, then you can refer to it in the text response of the node like this:

Hello `$user`

If the current user's name is Norman, then the response that is displayed to Norman is Hello Norman.

If you include one of these special characters in a text response, escape it by adding a backslash ( \ ) in front of it. Escaping the character prevents the service from misinterpreting it as being one of the following artifact types:

Special characters to escape in responses		
Special character	Artifact	Example
\$	Context variable	The transaction fee is \\$2.
@	Entity	Send us your feedback at feedback@example.com.
#	Intent	We are the #1 seller of lobster rolls in Maine.

You can include a hypertext link in a response by using HTML syntax. For example: Contact us at <a href="https://www.ibm.com">ibm.com</a>. Do not try to escape the quotations mark (with a backslash \\ ", for example).

The HTML is rendered properly in the "Try it out" pane. However, be sure to test that the client application you use to deploy the assistant can render HTML syntax properly.

## Learn more about simple responses

- [Adding multiple lines](#)
- [Adding variety](#)

### Adding multiple lines

If you want a single text response to include multiple lines separated by carriage returns, then follow these steps:

1. Add each line that you want to show to the user as a separate sentence into its own response variation field. For example:

Multiple line response

**Response variations**

Hi.

How are you today?

2. For the response variation setting, choose **multiline**.

**Note:** If you are using a workspace that was created before support for rich response types was added to the service, then you might not see the *multiline* option. Add a second text response type to the current node response. This action changes how the response is represented in the underlying JSON. As a result, the multiline option becomes available. Choose the multiline variation type. Now, you can delete the second text response type that you added to the response.

When the response is shown to the user, both response variations are displayed, one on each line, like this:

Hi.

How are you today?

### 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:

Where are  
you located?

```
If: #store_locations
Then:
1. "We have no brick and mortar stores. But,
with an internet connection, you can shop
us from anywhere!"
2. "You can find us online."
3. "Search for us on your favorite social media
platform."
```

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.

To change the sequence in which individual text responses are returned, complete the following steps:

1. Add each variation of the response into its own response variation field. For example:

Varying responses

**Response variations**

Hello.

Hi.

Howdy!

2. For the response variation setting, choose one of the following settings:

- o **sequential:** The system returns the first response variation the first time the dialog node is triggered, the second response variation the second time the node is triggered, and so on, in the same order as you define the variations in the node.

Results in responses being returned in the following order when the node is processed:

- First time:

Hello.

- Second time:

Hi.

- Third time:

Howdy !

- o **random:** The system randomly selects a text string from the variations list the first time the dialog node is triggered, and randomly selects another variation the next time, but without repeating the same text string consecutively.

Example of the order that responses might be returned in when the node is processed:

- First time:

Howdy !

- Second time:

Hi.

- Third time:

Hello.

## Rich responses

You can return responses with multimedia or interactive elements such as images or clickable buttons to simplify the interaction model of your application and enhance the user experience.

In addition to the default response type of **Text**, for which you specify the text to return to the user as a response, the following response types are supported:

- **Image:** Embeds an image into the response. The source image file must be hosted somewhere and have a URL that you can use to reference it. It cannot be a file that is stored in a directory that is not publicly accessible.
- **Option:** Adds a list of one or more options. When a user clicks one of the options, an associated user input value is sent to the service. How options are rendered can differ depending on where you deploy the dialog. For example, in one integration channel the options might be displayed as clickable buttons, but in another they might be displayed as a dropdown list.
- **Pause:** Forces the application to wait for a specified number of milliseconds before continuing with processing. You can choose to show an indicator that the dialog is working on typing a response. Use this response type if you need to perform an action that might take some time. For example, a parent node makes a Cloud Function call and displays the result in a child node. You could use this response type as the response for the parent node to give the programmatic call time to complete, and then jump to the child node to show the result. This response type does not render in the "Try it out" pane. You must access a node that uses this response type from a test deployment to see how your users will experience it.

## Adding rich responses

To add a rich response, complete the following steps:

1. Click the drop-down menu in the response field to choose a response type, and then provide any required information:

- **Image.** Add the full URL to the hosted image file into the **Image source** field. The image must be in .jpg, .gif, or .png format. The image file must be stored in a location that is publicly addressable by URL.

For example: <https://www.example.com/assets/common/logo.png>.

If you want to display an image title and description above the embedded image in the response, then add them in the fields provided.

- **Option.** Complete the following steps:

1. Click **Add option**.

2. In the **List label** field, enter the option to display in the list.

3. In the corresponding **Value** field, enter the user input to pass to the service when this option is selected.

Specify a value that you know will trigger the correct intent when it is submitted. For example, it might be a user example from the training data for the intent.

4. Repeat the previous steps to add more options to the list.

5. Optionally, add a list introduction in the **Title** field and additional information in the **Description** field to be displayed above the option list in the response.

For example, you can construct a response like this:

List title	List description	Response options	Option label	User input submitted when clicked
Insurance types	Which of these items do you want to insure?			
		Boat	I want to buy boat insurance	
		Car	I want to buy car insurance	
		Home	I want to buy home insurance	

- **Pause.** Add the length of time for the pause to last as a number of milliseconds (ms) to the **Duration** field.

The value cannot exceed 10,000 ms. Users are typically willing to wait about 8 seconds (8,000 ms) for someone to enter a response. To prevent a typing indicator from being displayed during the pause, choose **Off**.

Add another response type, such as a text response type, after the pause to clearly denote that the pause is over.

- **Text.** Add the text to return to the user in the text field. Optionally, choose a variation setting for the text response. See [Simple text response](#) for more details.

## 2. Click **Add response type** to add another response type to the current response.

You might want to add multiple response types to a single response to provide a richer answer to a user query. For example, if a user asks for store locations, you could show a map and display a button for each store location that the user can click to get address details. To build that type of response, you can use a combination of image, options, and text response types. Another example is using a text response type before a pause response type so you can warn users before pausing the dialog.

**Note:** You cannot add more than 5 response types to a single response. Meaning, if you define three conditional responses for a dialog node, each conditional response can have no more than 5 response types added to it.

## 3. If you added more than one response type, you can click the **Move** up or down arrows to arrange the response types in the order you want the service to process them.

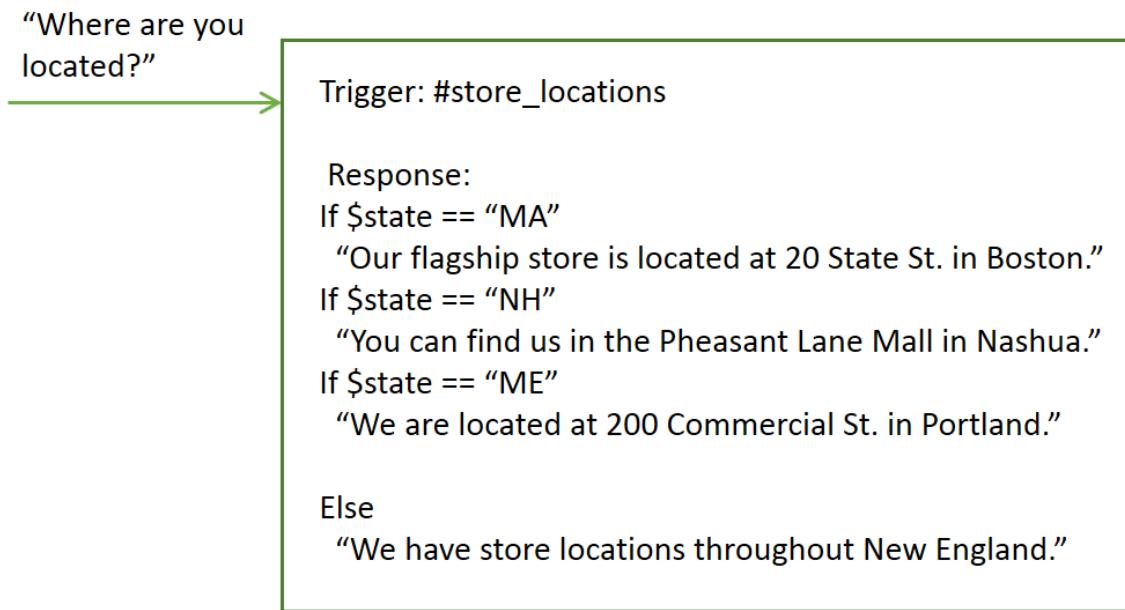
For information about the JSON format of these rich response types, see [Defining responses using the JSON editor](#).

### 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](#) 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.

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

You must open the response for editing to complete the following tasks:

- **Update context.** To change the value of a context variable when the response is triggered, specify the context value in the context editor. You update context for each individual conditional response; there is no common context editor or JSON editor for all conditional responses.
- **Add rich responses.** To add more than one text response or to add response types other than text responses to a single conditional response, you must open the edit response view.
- **Configure a jump.** To instruct the service to jump to a different node after this conditional response is processed, select **Jump to** from the *And finally* section of the response edit view. Identify the node that you want the service to process next. See [Configuring the Jump to action](#) for more information.

A **Jump to** action that is configured for the node is not processed until all of the conditional responses are processed. Therefore, if a conditional response is configured to jump to another node, and the conditional response is triggered, then the jump configured for the node is never processed, and so does not occur.

## 3. 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.

## 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.

**Note:** Avoid choosing this option when configuring a jump-to from a conditional response that goes to a node situated above the current node in the dialog tree. Otherwise, you can create an infinite loop. If the service jumps to the earlier node and checks its condition, it is likely to return false because the same user input is being evaluated that triggered the current node last time through the dialog. The service will go to the next sibling or back to root to check the conditions on those nodes, and will likely end up triggering this node again, which means the process will repeat itself.

- **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.

You can also use the API to add nodes or otherwise edit a dialog. See [Modifying a dialog using the API](#) for more information.

## 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](#) 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 return to the user.

```
{
 "output": {
 "generic": [
 {
 "values": [
 {
 "text": "This is my response text."
 }
],
 "response_type": "text",
 "selection_policy": "sequential"
 }
]
 }
}
```

In the resulting API /message response, the text response is formatted as follows:

```
{
 "text": "This is my response text.",
 "response_type": "text"
}
```

There are response types other than a text response that you can define. See [Responses](#) for more details.

**Note:** The following output object format is supported for backwards compatibility. Any workspaces that specify a text response by using this format will continue to function properly. With the introduction of rich response types, the `output.text` structure was augmented with the `output.generic` structure to facilitate supporting other types of responses in addition to text. Use the new format when you create new nodes to give yourself more flexibility, because you can subsequently change the response type, if needed.

```
{
 "output": {
 "text": {
 "values": [
 "This is my response text."
]
 }
 }
}
```

If you specify an API version that pre-dates the introduction of the rich response types (version 2018-07-10), then a workspace that contains non-textual or multiple response types will produce the first text response only. Only one text response can fit into the message `output.text` object. With version 2018-07-10, existing workspaces with text responses in the older format produce both the `output.text` and `output.generic` objects to represent the text response.

You can learn more about the /message API call from the [API reference](#).

## 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.

## Context variables

A context variable is a variable that you define in a node. You can specify a default value for it. Other nodes, application logic, or user input 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. You can also reference a context variable from dialog node response conditions to show different responses depending on a value provided by an external service or by the user.

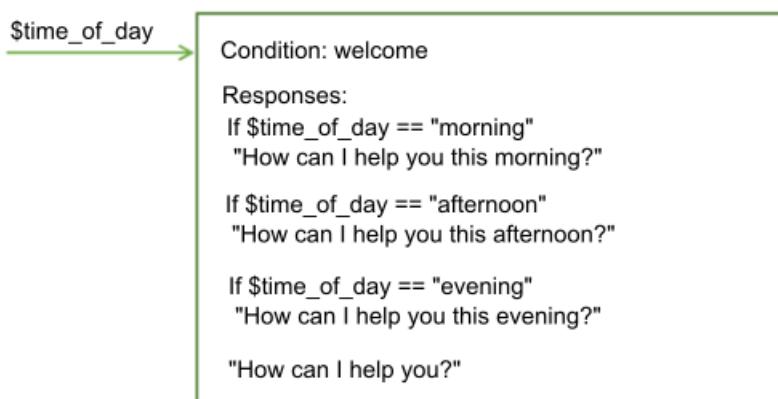
Learn more:

- [Passing context from the application](#)
- [Passing context from node to node](#)
- [Defining a context variable](#)
- [Common context variable tasks](#)
- [Deleting a context variable](#)
- [Updating a context variable](#)
- [How context variables are processed](#)
- [Order of operation](#)
- [Adding context variables to a node with slots](#)

### 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.

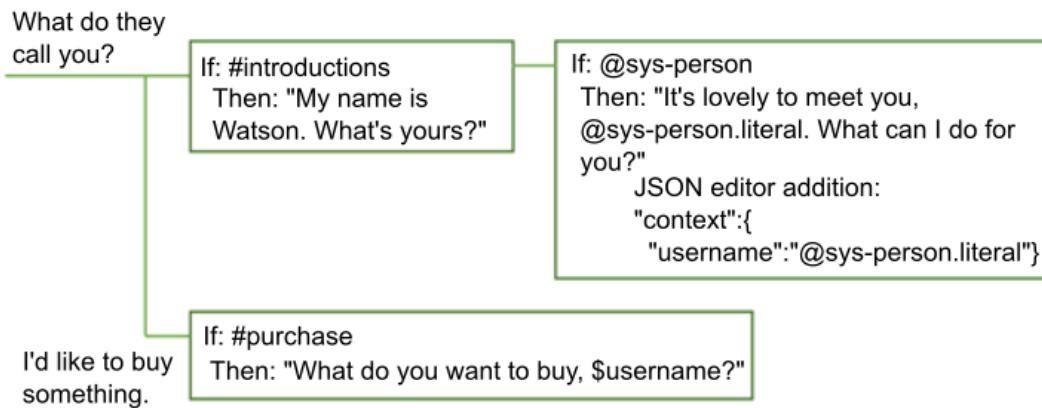


In this example, the dialog knows that the application sets the variable to one of these values: *morning*, *afternoon*, or *evening*. It can check for each value, and depending on which value is present, return the appropriate greeting. If the variable is not passed or has a value that does not match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

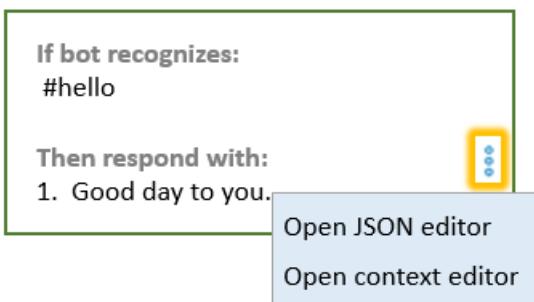


In this example, the system entity @sys-person is used to extract the user's name from the input if the user provides one. In the JSON editor, the username context variable is defined and set to the @sys-person value. In a subsequent node, the \$username context variable is included in the response to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

### Defining a context variable

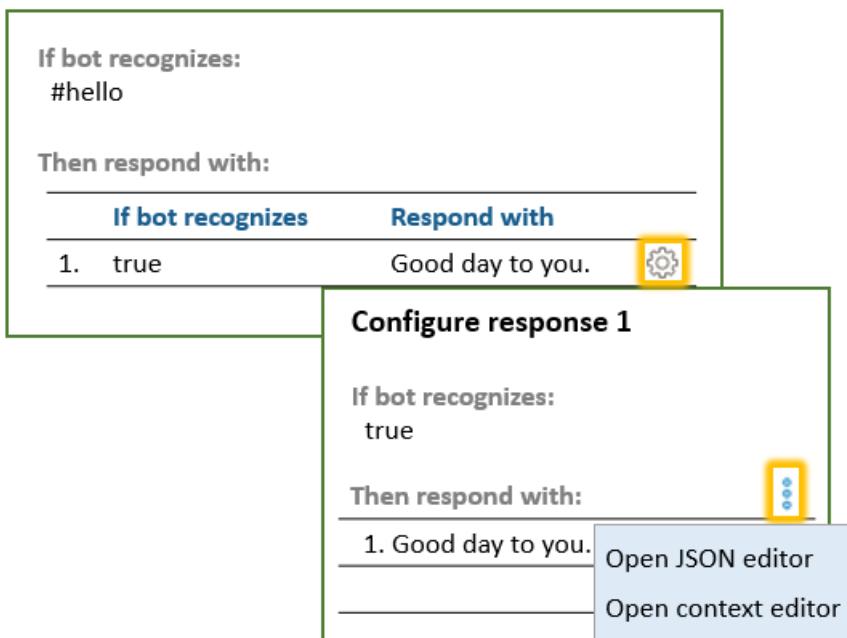
Define a context variable by adding the variable name to the **Variable** field and adding a default value for it to the **Value** field in the node's edit view.

1. Click to open the dialog node to which you want to add a context variable.
2. Click the **Options** icon that is associated with the node response, and then click **Open context editor**.



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

response with which you want to associate the context variable.



### 3. Add the variable name and value pair to the **Variable** and **Value** fields.

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

You can include other characters, such as periods and hyphens, in the name. However, if you do, then you must specify the shorthand syntax `$(variable-name)` every time you subsequently reference the variable. See [Expressions for accessing objects](#) for more details.

- The value can be any supported JSON type, such as a simple string variable, a number, a JSON array, or a JSON object.

The following table shows some examples of how to define name and value pairs for different types of values:

Variable	Value	Value Type
dessert	"cake"	String
age	18	Number
toppings_array	["onions","olives"]	JSON Array
full_name	{"first":"John","last":"Doe"}	JSON Object

To subsequently refer to these context variables, use the syntax `$name` where `name` is the name of the context variable that you defined.

For example, you might specify the following expression as the dialog response:

The customer, \$age-year-old <? \$full\_name.first ?>, wants a pizza with <? \$toppings\_array.join(' and ') ?>, and then \$dessert.

The resulting output is displayed as follows:

The customer, 18-year-old John, wants a pizza with onions and olives, and then cake.

You can use the JSON editor to define context variables also. You might prefer to use the JSON editor if you want to add a complex expression as the variable value. See [Context variables in the JSON editor](#) for more details.

### Common context variable tasks

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

Variable	Value
repeat	<code>&lt;?input.text?&gt;</code>

For example, the user input is, I want to order a device. If the node response is, You said: \$repeat, then the response would be displayed as, You said: I want to order a device.

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

Variable	Value
place	<code>@place</code>

For example, the user input is, I want to go to Paris. If your @place entity recognizes Paris, then the service saves Paris in the \$place context variable.

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	<code>&lt;?input.text.extract('[\d]+',0)?&gt;</code>

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	<code>&lt;? @email.literal ?&gt;</code>

For example, the user input is Contact me at joe@example.com. Your entity named @email recognizes the name@domain.com email format. By configuring the context variable to store @email.literal, you indicate that you want to store the part of the input that matched the pattern. If you omit the `.literal` property from the value expression, then the entity value name that you specified for the pattern is returned instead of the segment of user input that matched the pattern.

Many of these value examples use methods to capture different parts of the user input. For more information about the methods available for you to use, see [Expression language methods](#).

### Deleting a context variable

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

Variable	Value
	<code>null</code>

Variable	Value
----------	-------

order_form	null
------------	------

Alternatively you can delete the context variable in your application logic. For information about how to remove the variable entirely, see [Deleting a context variable in JSON](#).

### Updating a context variable value

To update a context variable's value, define a context variable with the same name as the previous context variable, but this time, specify a different value for it.

When more than one node sets the value of the same context variable, the value for the context variable can change over the course of a conversation with a user. Which value is applied at any given time depends on which node is being triggered by the user in the course of the conversation. The value specified for the context variable in the last node that is processed overwrites any values that were set for the variable by nodes that were processed previously.

For information about how to update the value of a context variable when the value is a JSON object or JSON array data type, see [Updating a context variable value in JSON](#)

### How context variables are processed

Where you define the context variable matters. The context variable is not created and set to the value that you specify for it until the service processes the part of the dialog node where you defined the context variable. In most cases, you define the context variable as part of the node response. When you do so, the context variable is created and given the specified value when the service returns the node response.

For a node with conditional responses, the context variable is created and set when the condition for a specific response is met and that response is processed. For example, if you define a context variable for conditional response #1 and the service processes conditional response #2 only, then the context variable that you defined for conditional response #1 is not created and set.

For information about where to add context variables that you want the service to create and set as a user interacts with a node with slots, see [Adding context variables to a node with slots](#).

### Order of operation

When you define multiple variables to be processed together, the order in which you define them does not determine the order in which they are evaluated by the service. The service evaluates the variables in random order. Do not set a value in the first context variable in the list and expect to be able to use it in the second variable in the list, because there is no guarantee that the first context variable will be executed before the second one. For example, do not use two context variables to implement logic that checks whether the user input contains the word Yes in it.

Variable	Value
----------	-------

user_input	<? input.text ?>
------------	------------------

contains_yes	<? \$user_input.contains('Yes') ?>
--------------	------------------------------------

Instead, use a slightly more complex expression to avoid having to rely on the value of the first variable in your list (user\_input) being evaluated before the second variable (contains\_yes) is evaluated.

Variable	Value
----------	-------

contains_yes	<? input.text.contains('Yes') ?>
--------------	----------------------------------

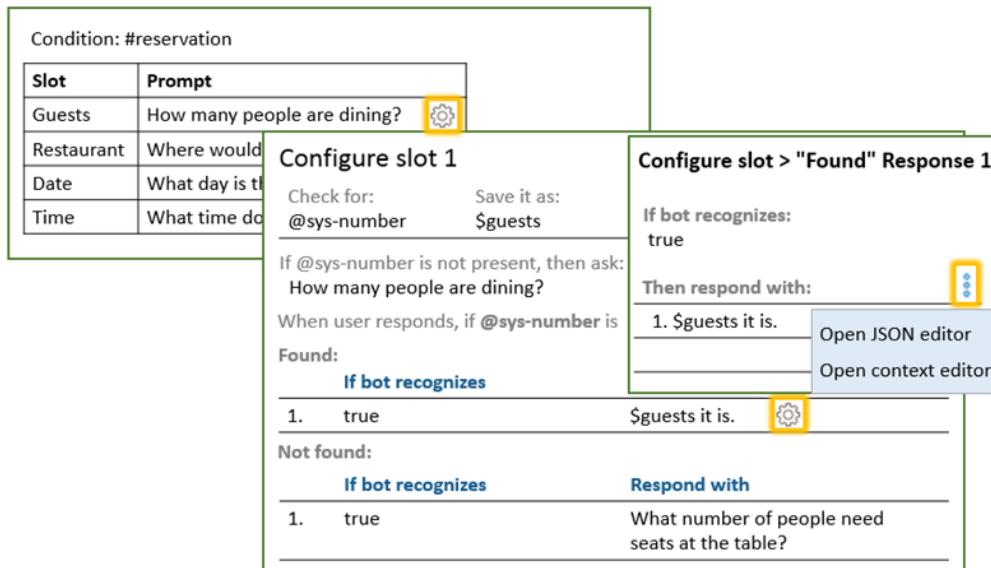
## Adding context variables to a node with slots

For more information about slots, see [Gathering information with slots](#).

### 1. Open the node with slots in the edit view.

- To add a context variable that is processed after a response condition for a slot is met, perform the following steps:

- Click the **Edit slot**  icon.
- Click the **Options**  icon, and then select **Enable conditional responses**.
- Click the **Edit response**  icon next to the response with which you want to associate the context variable.
- Click the **Options**  icon in the response section, and then click **Open context editor**.
- Add the variable name and value pair to the **Variable** and **Value** fields.



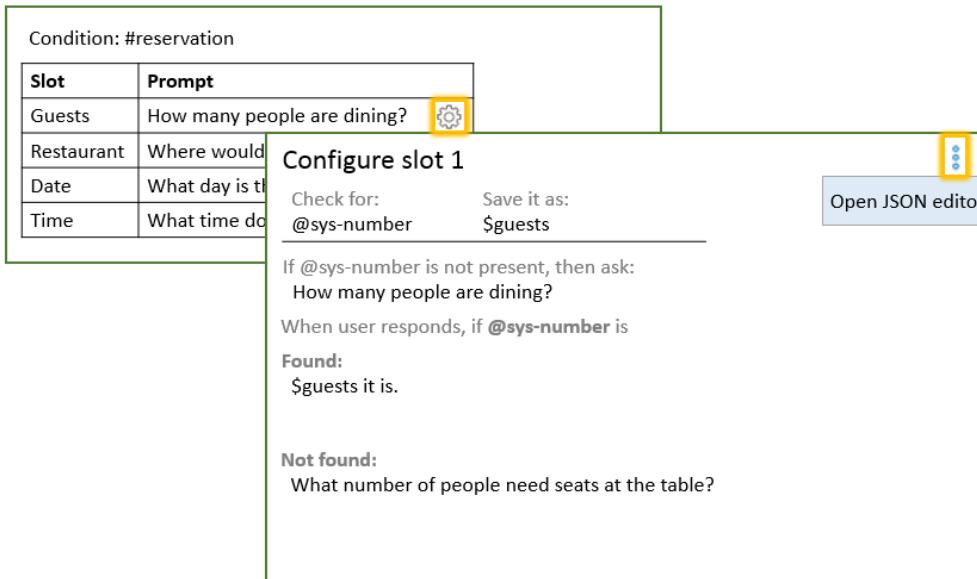
The screenshot shows the configuration of a slot named "Guests" with a prompt "How many people are dining?". A yellow box highlights the gear icon for "Edit slot". The "Configure slot 1" panel shows the variable "@sys-number" is checked and saved as "\$guests". The "Configure slot > 'Found' Response 1" panel shows a response for when the bot recognizes "true". It includes a yellow box around the "Open context editor" button. The JSON editor shows the variable \$guests is true. The "Not found" section shows a response for when the bot recognizes "true" with the message "What number of people need seats at the table?".

- To add a context variable that is set or updated after a slot condition is met, complete the following steps:

- Click the **Edit slot**  icon.
- From the **Options**  menu in the *Configure slot* view header, click **Open JSON editor**.
- Add the variable name and value pair in JSON format.

```
{
 "time_of_day": "morning"
}
```

**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. You must use the JSON editor instead. For more information about using the JSON editor, see [Context variables in the JSON editor](#).



## Context variables in the JSON editor

You can also define a context variable in the JSON editor. You might want to use the JSON editor if you are defining a complex context variable and want to be able to see the full SpEL expression as you add or change it.

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.

You can include other characters, such as periods and hyphens, in the name. However, if you do, then you must specify the shorthand syntax `$(variable-name)` every time you subsequently reference the variable. See [Expressions for accessing objects](#) for more details.

- The value can be any supported JSON type, such as a simple string variable, a number, a JSON array, or a JSON object.

The following JSON sample defines values for the `$dessert` string, `$toppings_array` array, `$age` number, and `$full_name` object context variables:

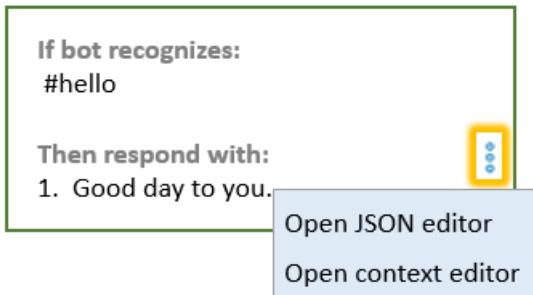
```
{
 "context": {
 "dessert": "cake",
 "toppings_array": [
 "onions",
 "olives"
],
 "age": 18,
 "full_name": {
 "first": "Jane",
 "last": "Doe"
 }
 },
 "output": {}
}
```

To define a context variable in JSON format, complete the following steps:

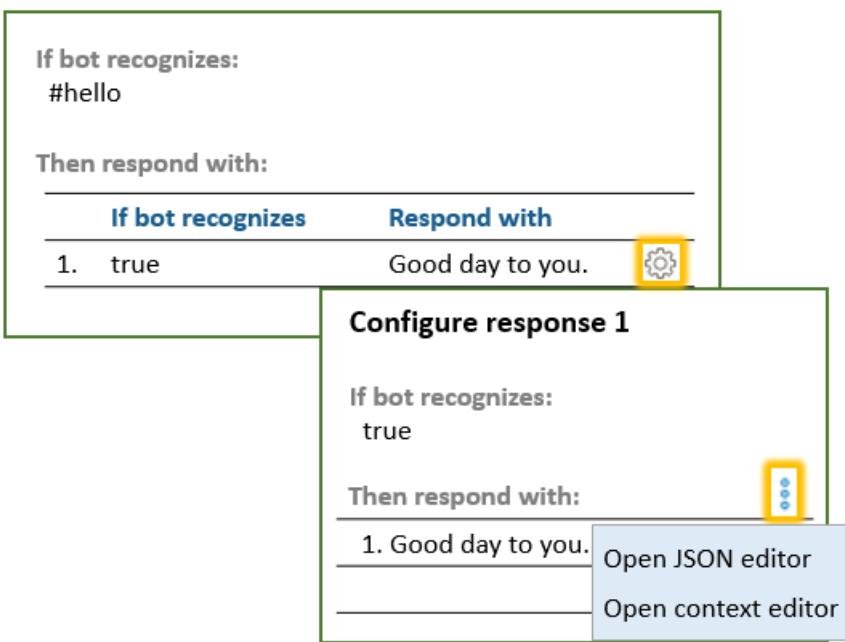
1. Click to open the dialog node to which you want to add the context variable.

**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 the JSON editor; the following steps describe how to access the menu.

2. Click the **Options**  icon that is associated with the response, and then click **Open JSON editor**.



If the **Multiple responses** setting is **On** for the node, then you must first click the **Edit response**  icon for the response with which you want to associate the context variable.



3. Add a "context":{} block if one is not present.

```
{
 "context":{},
 "output":{}
}
```

4. 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`.

Learn more:

- [Deleting a context variable in JSON](#)
- [Updating a context variable value in JSON](#)
- [Setting one context variable equal to another](#)

### Deleting a context variable in JSON

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.

### Updating a context variable value in JSON

In general, if a node sets the value of a context variable that is already set, then the previous value is overwritten by the new value.

### 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](#) 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](#) for more information about methods you can perform on arrays.

Setting one context variable equal to another

When you set one context variable equal to another context variable, you define a pointer from one to the other. If the value of one of the variables subsequently changes, then the value of the other variable is changed also.

For example, if you specify a context variable as follows, then when the value of either \$var1 or \$var2 subsequently changes, the value of the other changes too.

Variable Value

```
var2      var1
```

Do not set one variable equal to another to capture a point in time value. When dealing with arrays, for example, if you want to capture an array value stored in a context variable at a certain point in the dialog to save it for later use, you can create a new variable based on the current value of the variable instead.

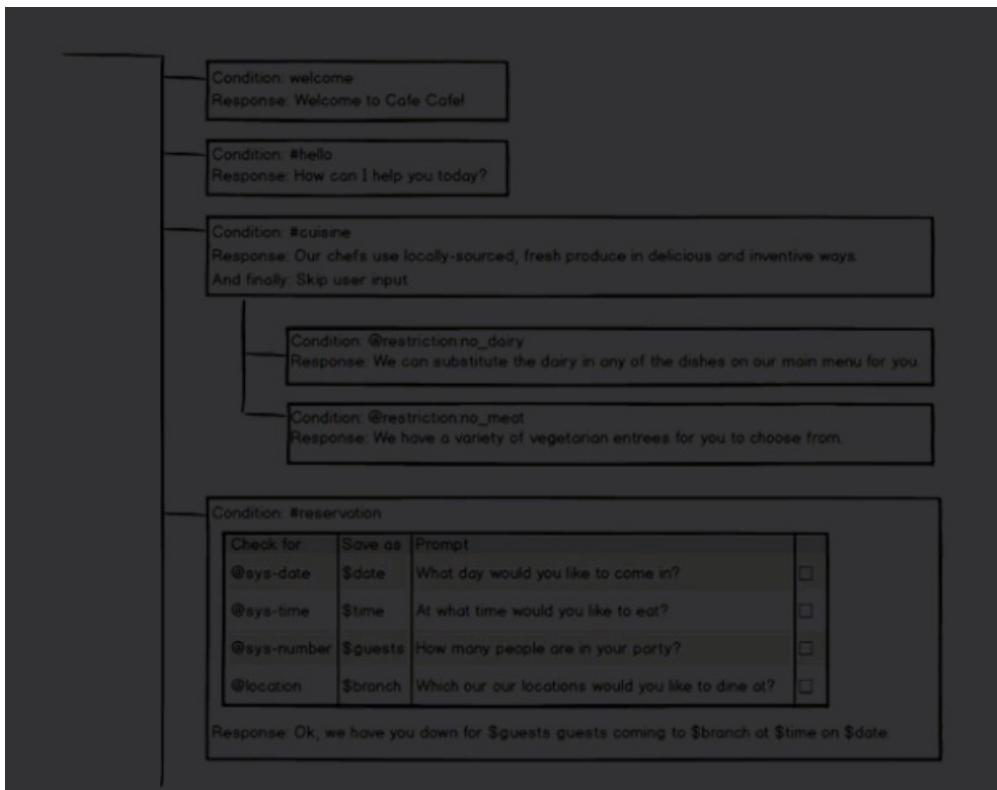
For example, to create a copy of the values of an array at a certain point of the dialog flow, add a new array that is populated with the values for the existing array. To do so, you can use the following syntax:

```
{
  "context": {
    "var2": "<? output.var2?:new JsonArray().append($var1) ?>"
  }
}
```

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](#)
- [Customizing digressions](#)
- [Digression usage tips](#)
- [Disabling digressions into a root node](#)
- [Digression tutorial](#)
- [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, digressions 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 response* toggle to **No** to prevent the dialog from returning to the current node and continuing to process its branch.

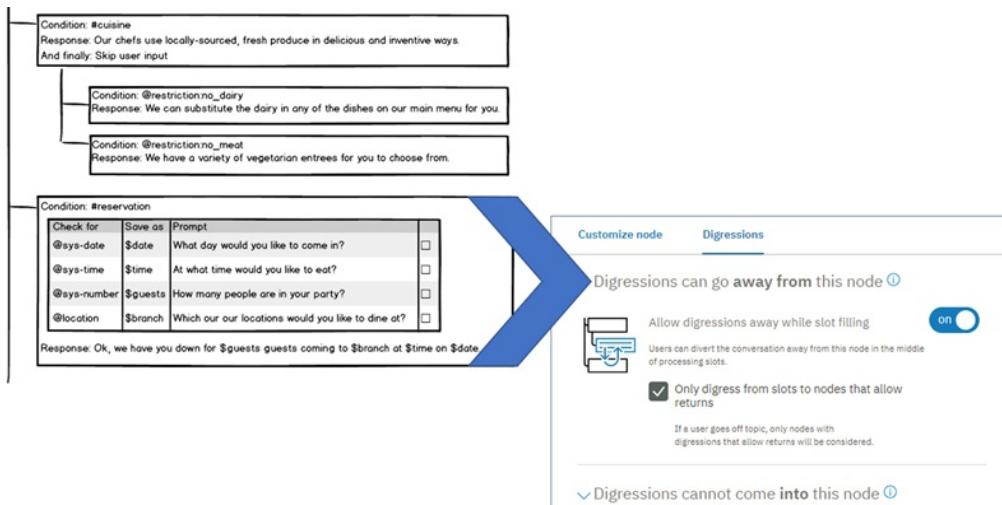
For example, if the user asks, **Do you sell cupcakes?** and the response, **We offer cupcakes in a variety of flavors and sizes** is displayed before the user changes subjects, you might not want the dialog to return to where it left off. Especially, if the child nodes only address possible follow-up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

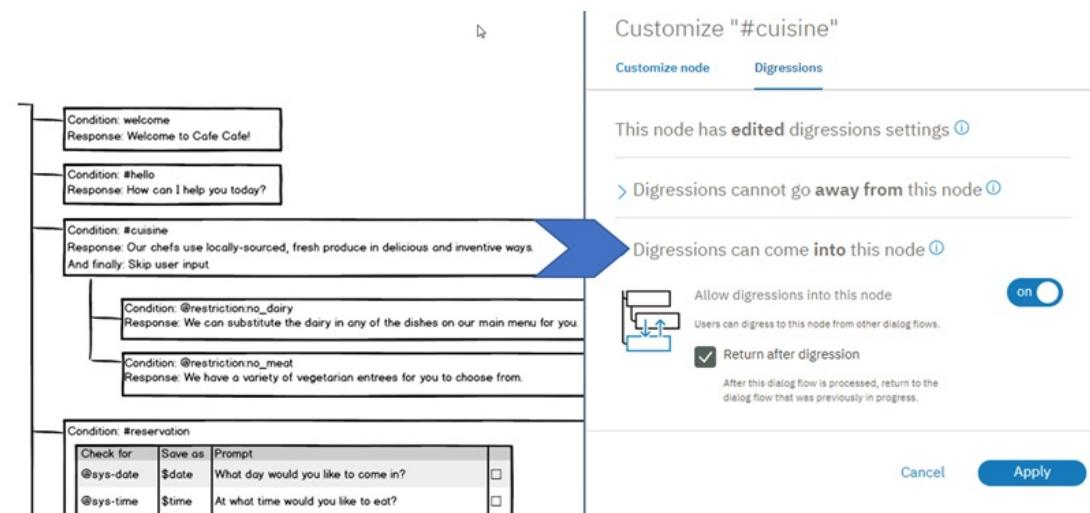
When selected, as the dialog looks for a node to answer the user's unrelated question, it ignores any root nodes that are not configured to return after the digression. Select this checkbox if you want to prevent users from being able to permanently leave the node before they have finished filling the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

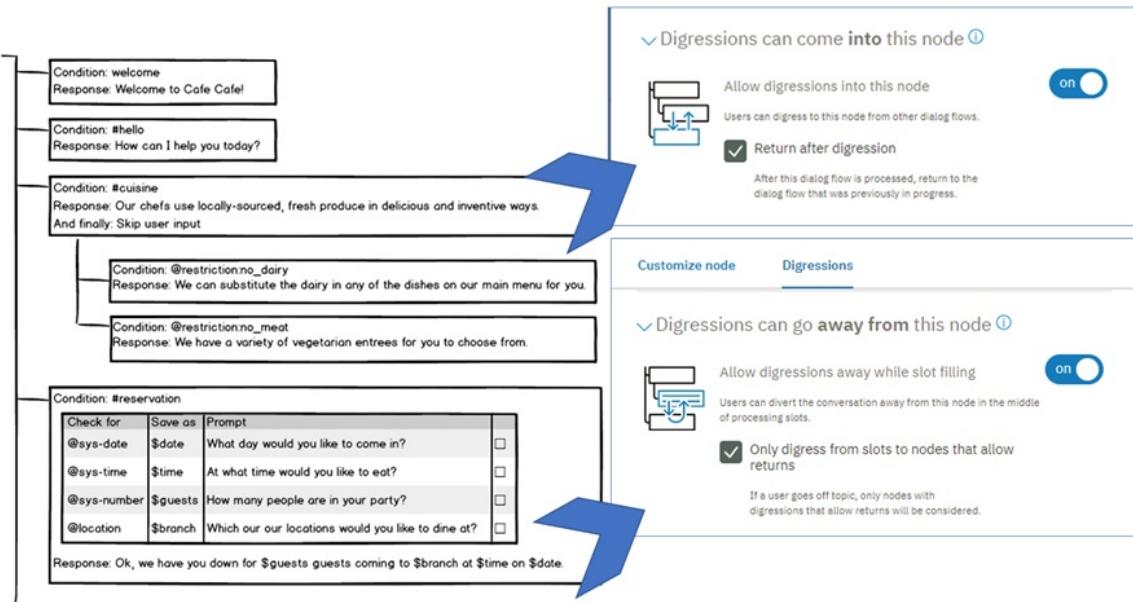


3. Click **Apply**.

4. 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.



Digression usage tips

This section describes solutions to situations that you might encounter when using digressions.

- Custom return message:** For any nodes where you enable returns from digressions away, consider adding wording that lets users know they are returning to where they left off in a previous dialog flow. In your text response, use a special syntax that lets you add two versions of the response.

If you do not take action, the same text response is displayed a second time to let users know they have returned to the node they digressed away from. You can make it clearer to users that they have returned to the original conversation thread by specifying a unique message to be displayed when they return.

For example, if the original text response for the node is, `What's the order number?`, then you might want to display a message like, `Now let's get back to where we left off. What is the order number?` when users return to the node.

To do so, use the following syntax to specify the node text response:

```
<? (returning_from_digression)? "post-digression message" : "first-time message" ?>
```

For example:

```
<? (returning_from_digression)? "Now, let's get back to where we left off.  
What is the order number?" : "What's the order number?" ?>
```

Note: You cannot include SpEL expressions or shorthand syntax in the text responses that you add. In fact, you cannot

use shorthand syntax at all. Instead, you must build the message by concatenating the text strings and full SpEL expression syntax together to form the full response. For example, use the following syntax to include a context variable in a text response that you would normally specify as, What can I do for you, \$username?:

```
<? (returning_from_digression)? "Where were we, " +
  context["username"] + "? Oh right, I was asking what can I do
  for you today." : "What can I do for you today, " +
  context["username"] + "?" ?>
```

For full SpEL expression syntax details, see [Expression for accessing objects](#).

- **Preventing returns:** In some cases, you might want to prevent a return to the interrupted conversation flow based on a choice the user makes in the current dialog flow. You can use special syntax to prevent a return from a specific node.

For example, you might have a node that conditions on `#General_Connect_To_Agent` or a similar intent. When triggered, if you want to get the user's confirmation before you transfer them to an external service, you might add a response such as, Do you want me to transfer you to an agent now? You could then add two child nodes that condition on `#yes` and `#no` respectively.

The best way to manage digressions for this type of branch is to set the root node to allow digression returns. However, on the `#yes` node, include the SpEL expression `<? clearDialogStack() ?>` in the response. For example:

OK. I will transfer you now. `<? clearDialogStack() ?>`

This SpEL expression prevents the digression return from happening from this node. When a confirmation is requested, if the user says yes, the proper response is displayed, and the dialog flow that was interrupted is not resumed. If the user says no, then the user is returned to the flow that was interrupted.

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](#) for more information.

Digression tutorial

Follow the [tutorial](#) 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_else` condition 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.

Disambiguation

This feature is available only to Premium users.

When you enable disambiguation, you instruct the service to ask users for help when it finds that more than one dialog node can respond to their input. Instead of guessing which node to process, your assistant shares a list of the top node options

with the user, and asks the user to pick the right one.



If enabled, disambiguation is not triggered unless the following conditions are met:

- The confidence score of one or more of the runner-up intents detected in the user input is greater than 55% of the confidence score of the top intent.
- The confidence score of the top intent is above 0.2.

Even when these conditions are met, disambiguation does not occur unless two or more independent nodes in your dialog meet the following criteria:

- The node condition includes one of the intents that triggered disambiguation. Or the node condition otherwise evaluates to true. For example, if the node checks for an entity type and the entity is mentioned in the user input, it is eligible.
- There is text in the node's *node purpose* field.

Learn more

- [Disambiguation example](#)
- [Enabling disambiguation](#)
- [Choosing nodes](#)
- [Handling none of the above](#)
- [Testing disambiguation](#)

Disambiguation example

For example, you have a dialog that has two nodes with intent conditions that address cancellation requests. The conditions are:

- eCommerce_Cancel_Product_Order
- Customer_Care_Cancel_Account

If the user input is `i must cancel it today`, then the following intents might be detected in the input:

```
[{"intent": "Customer_Care_Cancel_Account", "confidence": 0.6618281841278076},  
 {"intent": "eCommerce_Cancel_Product_Order", "confidence": 0.4330700159072876},  
 {"intent": "Customer_Care_Appointments", "confidence": 0.2902342438697815},  
 {"intent": "Customer_Care_Store_Hours", "confidence": 0.2550420880317688}, ...]
```

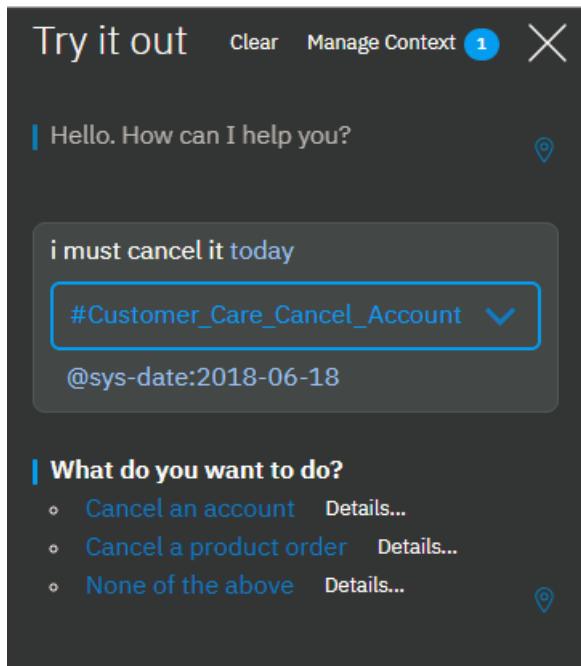
The service is 0.6618281841278076 (66%) confident that the user goal matches the `#Customer_Care_Cancel_Account` intent. If any other intent has a confidence score that is greater than 55% of 66%, then it fits the criteria for being a disambiguation candidate.

$$0.66 \times 0.55 = 0.36$$

Intents with a score that is greater than 0.36 are eligible.

In our example, the `#eCommerce_Cancel_Product_Order` intent is over the threshold, with a confidence score of `0.4330700159072876`.

When the user input is `i must cancel it today`, both dialog nodes will be considered viable candidates to respond. To determine which dialog node to process, the assistant asks the user to pick one. And to help the user choose between them, the assistant provides a short summary of what each node does. The summary text it displays is extracted directly from the `node purpose` information that was specified for each node.



Notice that the service recognizes the term `today` in the user input as a date, a mention of the `@sys-date` entity. If your dialog tree contains a node that condition on the `@sys-date` entity, then it is also included in the list of disambiguation choices. This image shows it included in the list as the *Capture date information* option.

The following video provides an overview of disambiguation.

Enabling disambiguation

To enable disambiguation, complete the following steps:

1. From the Dialogs page, click **Settings**.
2. Click **Disambiguation**.
3. In the *Enable disambiguation* section, switch the toggle to **On**.
4. In the prompt message field, add text to show before the list of dialog node options. For example, *What do you want to do?*
5. **Optional:** In the none of the above message field, add text to display as an additional option that users can pick if none of the other dialog nodes reflect what the user wants to do. For example, *None of the above*.

Keep the message short, so it displays inline with the other options. The message must be less than 512 characters. For information about what the service does if a user chooses this option, see [Handling none of the above](#).

6. Click **Close**

7. Decide which dialog nodes you want the assistant to ask for help with.

- You can pick nodes at any level of the tree hierarchy.
- You can pick nodes that condition on intents, entities, special conditions, context variables, or any combination of these values.

See [Choosing nodes](#) for tips.

For each node that you want to opt in to disambiguation, complete the following steps:

1. Click to open the node in edit view.
2. In the *node purpose* field, describe the user task that this dialog node is designed to handle. For example, *Cancel an account*.

And finally

Wait for user input ▾

If virtual-assistant needs to represent node to users, then use:

Enter node purpose 

 Status: this node will not be used for features like disambiguation unless a node purpose is provided. 

Choosing nodes

Choose nodes that serve as the root of a distinct branch of the dialog to be disambiguation choices. These can include nodes that are children of other nodes. The key is for the node to condition on some distinct value or values that distinguish it from everything else.

Keep in mind:

- For nodes that condition on intents, if the service is confident that the node's intent condition matches the user's intent, then the node is included as a disambiguation option.
- For nodes with boolean conditions (conditions that evaluate to either true or false), the node is included as a disambiguation option if the condition evaluates to true. For example, when the node conditions on an entity type, if the entity is mentioned in the input that triggers disambiguation, then the node is included.
- The order of nodes in the tree hierarchy impacts disambiguation.
 - It impacts whether disambiguation is triggered at all

Look at the [scenario](#) that is used earlier to introduce disambiguation, for example. If the node that conditions on @sys-date was placed higher in the dialog tree than the nodes that condition on the #Customer_Care_Cancel_Account and #eCommerce_Cancel_Product_Order intents, disambiguation would never be triggered when a user enters, *i must cancel it today*. That's because the service would consider the date mention (*today*) to be more important than the intent references due to the placement of the corresponding nodes in the tree.

- It impacts which nodes are included in the disambiguation options list

Sometimes a node is not listed as a disambiguation option as expected. This can happen if a condition value is also referenced by a node that is not eligible for inclusion in the disambiguation list for some reason. For example, an entity mention might trigger a node that is situated earlier in the dialog tree but is not enabled for disambiguation. If the same entity is the only condition for a node that is enabled for disambiguation, but is situated lower in the tree, then it is not added as a disambiguation option because the service never reaches it. It matched against the earlier node and was omitted, so the service does not process the later node.

For each node that you opt in to disambiguation, test scenarios in which you expect the node to be included in the disambiguation options list. Testing gives you a chance to make adjustments to the node order or other factors that might impact how well disambiguation works at run time.

Handling none of the above

When a user clicks the *None of the above* option, the service strips the intents that were recognized in the user input from

the message and submits it again. This action typically triggers the anything else node in your dialog tree.

To customize the response that is returned in this situation, you can add a root node with a condition that checks for a user input with no recognized intents (the intents are stripped, remember) and contains a suggestion_id property. A suggestion_id property is added by the service when disambiguation is triggered.

Add a root node with the following condition:

```
intents.size() == 0 && input.suggestion_id
```

This condition is met only by input that has triggered a set of disambiguation options of which the user has indicated none match her goal.

Add a response that lets users know that you understand that none of the options that were suggested met their needs, and take appropriate action.

Again, the placement of nodes in the tree matters. If a node that conditions on an entity type that is mentioned in the user input is higher in the tree than this node, its response is displayed instead.

Testing disambiguation

To test disambiguation, complete the following steps:

1. From the "Try it out" pane, enter a test utterance that you think is a good candidate for disambiguation, meaning two or more of your dialog nodes are configured to address utterances like it.
2. If the response does not contain a list of dialog node options for you to choose from as expected, first check that you added summary information to the node purpose field for each of the nodes.
3. If disambiguation is still not triggered, it might be that the confidence scores for the nodes are not as close in value as you thought.

You can get information about the intents, entities, and other properties that are returned for certain user inputs.

- To see the confidence scores of the intents that were detected in user input, temporarily add <? intents ?> to the end of the node response for a node that you know will be triggered.

This SpEL expression shows the intents that were detected in the user input as an array. The array includes the intent name and the level of confidence that the service has that the intent reflects the user's intended goal.

- To see which entities, if any, were detected in the user input, you can temporarily replace the current response with a single text response that contains the SpEL expression, <? entities ?>.

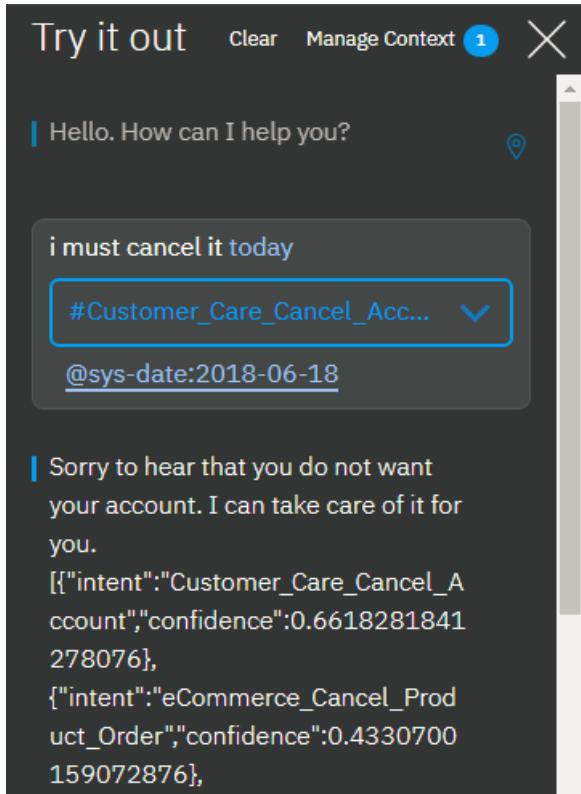
This SpEL expression shows the entities that were detected in the user input as an array. The array includes the entity name, location of the entity mention within the user input string, the entity mention string, and the level of confidence that the service has that the term is a mention of the entity type specified.

- To see details for all of the artifacts at once, including other properties, such as the value of a given context variable at the time of the call, you can inspect the entire API response. See [Viewing API call details](#).

4. Temporarily remove the description you added to the *node purpose* field for at least one of the nodes that you anticipate will be listed as a disambiguation option.

5. Enter the test utterance into the "Try it out" pane again.

If you added the <? intents ?> expression to the response, then the text returned includes a list of the intents that the service recognized in the test utterance, and includes the confidence score for each one.



After you finish testing, remove any SpEL expressions that you appended to node responses, or add back any original responses that you replaced with expressions, and repopulate any *node purpose* fields from which you removed text.

Creating a dialog

Use the Watson Assistant tool to create your dialog.

Dialog node limits

Dialog node limits	
Dialog nodes per workspace	Tree depth
100,000	2,000 supported; 20 or fewer recommended

The welcome and anything_else dialog nodes that are prepopulated in the tree count toward the total.

Building a dialog

To build 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:

- o **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.
- o **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.

2. To add more nodes to the dialog tree, click the **More**  icon on the **Welcome** node, and then select **Add node below**.

3. 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	Condition builder syntax 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

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 and that 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 2,048 characters in length.

For more information about how to test for values in conditions, see [Conditions](#).

4. **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](#) for more details.

5. Enter a response.

- Add the text or multimedia elements 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, rich responses, or how to add variety to responses, see [Responses](#).

6. 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](#) for more details.

7. **Optional:** If you want this node to be considered when users are shown a set of node choices at run time, and asked to pick the one that best matches their goal, then add a short description of the user goal handled by this node to the **node purpose** field. For example, *Place an order*.

 The **node purpose** field is only displayed to Premium plan users. See [Disambiguation](#) for more

details.

8. 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.

9. To add more nodes, select a node in the tree, and then click the More 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](#).

10. Test the dialog as you build it. See [Testing your dialog](#) for more information.

Testing your dialog

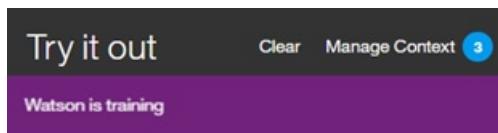
As you make changes to your dialog, you can test it at any time to see how it responds to input.



1. From the Dialog tab, click the Try it icon.

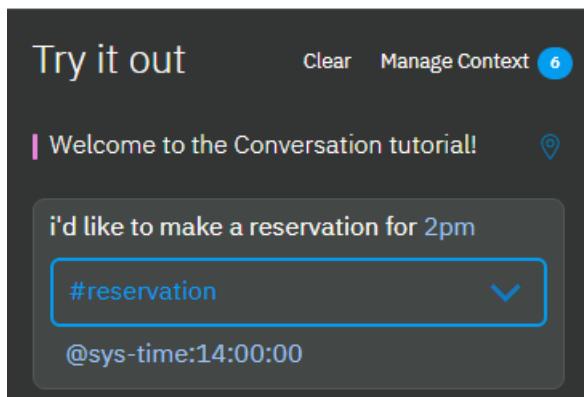
2. In the chat pane, type some text and then press Enter.

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:



3. 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:



4. If you want to know which node in the dialog tree triggered a response, click the **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.

5. 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](#) for more information.

6. Continue to interact with the dialog to see how the conversation flows through it.

- To find and resubmit a test utterance, you can press the Up key to cycle through your recent inputs.
- 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.

See [Dialog building tips](#) for tips that might help you as you get started.

Finding a dialog node by its node ID

You can search for a dialog node by its node ID. 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.

Discover a node based on its node ID 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`

4. 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.

5. 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 - `copyn` 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 `copyn`.

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**  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**  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 `false` condition to prevent them from being processed.

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

- Condition: If no condition is specified, then the service processes the nodes within the folder directly. If a condition is 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 a root level node into a folder that is a child of another node, then the root node becomes a child 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 has no condition or its 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, then the folder is transparent to the service, and each node in the folder is treated like any other individual node in the tree.

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**  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**  icon on the child node, and then select **Add folder**.

The folder is opened in edit view.

2. **Optional:** Name the folder.

3. **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.

4. 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**  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**  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**  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 to be.

Dialog building tips

Learn how to approach building a dialog and get some tips on completing more complex steps.

Review these tips from experienced dialog designers.

Planning the overall dialog

- Plan out the design of the dialog that you want to build before you add a single dialog node in the tool. Sketch it out on paper, if necessary.
- Whenever possible, base your design decisions on data from real-world evidence and behaviors. Do not add nodes to handle a situation that someone *thinks* might occur.
- Avoid copying business processes as-is. They are rarely conversational.
- If people already use a process, examine how they approach it. People typically optimize the process from a conversational perspective.
- Decide on the tone, personality, and positioning of your assistant. Consistently reflect these choices in the dialog you create.
- Never misrepresent the assistant as being a human. If users believe the assistant is a person, then find out it's not, they are likely to distrust it.
- Not everything has to be a conversation. Sometimes a web form works better.

Adding nodes

- Add a node name that describes the purpose of the node.

You know what the node does right now, but months from now you might not. Your future self and any team members will thank you for adding a descriptive node name. And the node name is displayed in the log, which can help you debug a conversation later.

- To gather the information that is required to perform a task, try using a node with slots instead of a bunch of separate nodes to elicit information from users. See [Gathering information with slots](#).
- For a complex process flow, tell users about any information they will need to provide at the start of the process.
- Understand how the service travels through the dialog tree and the impact that folders, branches, jump-tos, and digressions have on the route. See [Dialog flow](#).
- Do not add jump-tos everywhere. They increase the complexity of the dialog flow, and make it harder to debug the dialog later.
- To jump to a node in the same branch as the current node, use *Skip user input* instead of a *Jump-to*.

This choice prevents you from having to edit the current node's settings when you remove or reorder the child nodes

being jumped to. See [Defining what to do next](#).

- Before you enable digressions away from a node, test the most common user scenarios. And be sure that likely digressed-to nodes are configured to return. See [Digressions](#).

Adding responses

- Keep answers short and useful.
- Reflect the user's intent in the response.

Doing so assures users that the bot is understanding them, or if it is not, gives users a chance to correct a misunderstanding right away.

- Only include links to external sites in responses if the answer depends on data that changes frequently.
- Avoid overusing buttons. Encouraging users to pick predefined options from a set of buttons is less like a real conversation, and decreases your ability to learn what users really want to do. When you let real users ask for things in their own words, you can use the input to train the system and derive better intents.
- Avoid using a bunch of nodes when one node will do. For example, add multiple conditional responses to a single node to return different responses depending on details provided by the user. See [Conditional responses](#).
- Word your responses carefully. You can change how someone reacts to your system based simply on how you phrase a response. Changing one line of text can prevent you from having to write multiple lines of code to implement a complex programmatic solution.
- Back up your workspace frequently. See [Exporting and copying workspace](#).

Tips for capturing information from user input

It can be difficult to know the syntax to use in your dialog node to accurately capture the information you want to find in the user input. Here are some approaches you can use to address common goals.

- **Returning the user's input:** You can capture the exact text uttered by the user and return it in your response. Use the following SpEL expression in a response to repeat the text that the user specified back in the response:

You said: <? input.text ?>.

- **Determining the number of words in user input:** You can perform any of the supported String methods on the input.text object. For example, you can find out how many words there are in a user utterance by using the following SpEL expression:

```
input.text.split(' ').size()
```

See [Expression language methods for String](#) to learn about more methods you can use.

- **Dealing with multiple intents:** A user enters input that expresses a wish to complete two separate tasks. I want to open a savings account and apply for a credit card. How does the dialog recognize and address both of them? See the [Compound questions](#) entry from Simon O'Doherty's blog for strategies you can try. (Simon is a developer on the Watson Assistant team.)
- **Dealing with ambiguous intents:** A user enters input that expresses a wish that is ambiguous enough that the service finds two or more nodes with intents that could potentially address it. How does the dialog know which dialog branch to follow? If you enable disambiguation, it can show users their options and ask the user to pick the right one. See [Disambiguation](#) for more details.
- **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.

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 comparing numbers, first make sure the entity or variable you are checking has a value. If the entity or variable does not have a number value, it is treated as having a null value (0) in a numeric comparison.

For example, you want to check whether a dollar value that a user specified in user input is less than 100. If you use 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 this type of inaccurate result, use a condition such as @price AND @price < 100. If @price has 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.

Storing and recognizing entity pattern groups in input

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.

Variable	Value
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:

Variable	Value
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 details			
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) (555))	958	String that matches the regex for the first defined group, which is ((958) (555)).
groups[2]	(958)	958	Match against the group that is included as the first operand in the OR expression ((958) (555))
groups[3]	(555)	null	No match against the group that is included as the second operand in the OR expression ((958) (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.

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:

Variable	Value
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 elements
Array element value
Array element number

"958-123-2345"	0
----------------	---

Array element value Array element number

"958"	1
"958"	2
null	3
"123"	4
"2345"	5

From the result, you can 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:

Variable	Value
----------	-------

```
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.

Variable	Value
----------	-------

```
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.

Viewing API call details

As you test your dialog with the "Try it out" pane, you might want to know what the underlying API calls look like that are being returned from the service. You can use the developer tools provided by your web browser to inspect them.

From Chrome, for example, open the Developer tools. Click the Network tool. The Name section lists multiple API calls. Click the message call associated with your test utterance, and then click the Response column to see the API response body. It lists the intents and entities that were recognized in the user input with their confidence scores, and the values of context variables at the time of the call. To view the response body in structured format, click the Preview column.

The screenshot shows the IBM Watson Assistant interface. On the left, there's a sidebar with 'Try it out' and 'Manage Context'. The main area displays a conversation history:

- User: 'hello top 20'
- Watson: '#General_Greetings' (highlighted in blue)
- User: '@sys-number:20'
- Watson: 'Good day to you!'

Below the conversation is a text input field: 'Enter something to test your virtual assistant'.

On the right, a developer tools window is open with the 'Network' tab selected. The 'Response' tab is also highlighted. The response body shows a JSON object:

```
{"intents": [{"intent": "General_Greetings", "confidence": 0.6077133655548096}], "entities": [{"entity": "number", "value": "20", "start": 10, "end": 13, "type": "sys-number"}]}
```

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.

When do you open?

Condition: #operating_hours

Slot	Prompt
Location	Are you visiting our store downtown or the one in the mall?

Response:

```
$location == "downtown"  
"We are open from 8AM to 8PM seven days a week."  
$location == "mall"  
"Our retail store in the mall follows the mall operating hours of 9AM to 9PM."
```

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.

I'd like to make a reservation.

Condition: #reservation

Slot	Prompt
Guests	How many guests are dining?
Restaurant	Where would you like to eat?
Date	What day is the reservation for?
Time	At what time would you like to dine?

How many guests are dining?

The user might provide values for multiple 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.

I'd like to make
a reservation.

There will be **6** of
us dining at **7PM**.

Where would
you like to eat?

Condition: #reservation

Slot	Prompt	
Guests	How many people are dining?	✓
Restaurant	Where would you like to eat?	
Date	What day is the reservation for?	
Time	At what time do you want to dine?	✓

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.

What's the
weather forecast?

In Boston today
it will be sunny.

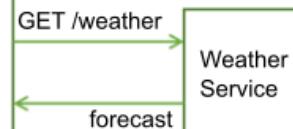
What about in
NYC tomorrow?

In NYC tomorrow
it will be rainy.

Condition: #weather

Slot	Prompt	Save as
Location	For which city do you want a forecast?	(default: Boston, MA)
Date	What day?	(default: today)

Slot	Prompt	Save as
Location	For which city do you want a forecast?	@sys-location = New York, NY
Date	What day?	@sys-date = tomorrow



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](#) 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](#).

4. **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 therefore 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**  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](#).

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.

Example slots for pizza order

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.

5. 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.

Optional slot
Information **Check for Save as**
Wheat restriction @dietary \$dairy

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 \$dairy 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.

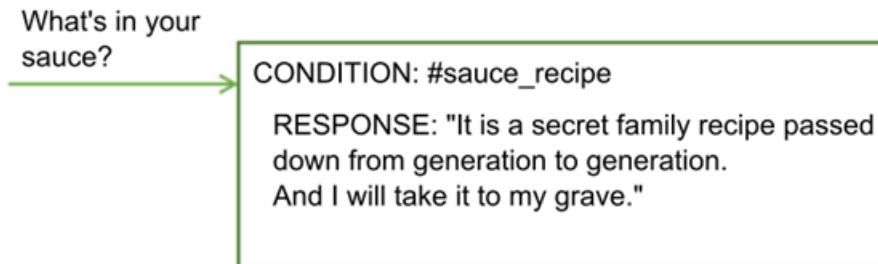
- **Conditional:** If you want a slot to be enabled only under certain conditions, then you can add a condition to it. For example, if slot 1 asks for a meeting start time, slot 2 captures the meeting duration, and slot 3 captures the end time, then you might want to enable slot 3 (and ask for the meeting 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 ☰ 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.

6. 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](#) for more ways to use the slot handler.

7. 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](#).

8. 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](#).

Slots usage tips

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

Slot properties

Property	Description
<code>name</code>	
<code>all_slots_filled</code>	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 for a usage example.
<code>event.current_value</code>	Current value of the context variable for this slot. See Replacing a slot context variable value for a usage example for this property and the <code>event.previous_value</code> property.
<code>event.previous_value</code>	Previous value of the context variable for this slot.
<code>has_skipped_slots</code>	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 for more information about next step options for slots and Handling requests to exit a process for information about next step options for slot handlers.

Property	Description
name	
slot_in_fo	Forces the slot condition to be applied to the current slot only. See Getting confirmation for more details.
cus	

Consider using these approaches for handling common tasks.

- [Asking for everything at once](#)
- [Capturing multiple values](#)
- [Reformatting values](#)
- [Dealing with zeros](#)
- [Getting confirmation](#)
- [Replacing a slot context variable value](#)
- [Avoiding number confusion](#)
- [Adding conditions to Found and Not found responses](#)
- [Moving on after multiple failed attempts](#)
- [Preventing a Found response from displaying when it is not needed](#)
- [Handling requests to exit a process](#)

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 initial 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.

Multiple value slot				
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 ...

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](#) for other reformatting ideas.

Dealing with zeros

Using `@sys-number` in a slot condition is helpful for capturing any numbers that users specify in their input. However, it does not behave as expected when users specify the number zero (0). Instead of treating zero as a valid number, the condition is evaluated to false, and the service prompts the user for a number again. To prevent this behavior, check for `@sys-number` or `@sys-number:0` in the slot condition.

To ensure that a slot condition that checks for number mentions deals with zeros properly, complete the following steps:

1. Add `@sys-number || @sys-number:0` to the slot condition field, and then provide the context variable name and text prompt.
2. Click the **Edit response**  icon.
3. Click the **More**  menu, and then select **Open JSON editor**.
4. Update the context variable which now has the syntax, `"number": "@sys-number || @sys-number:0"`, to specify `@sys-number` only.

```
{
  "context": {
    "number": "@sys-number"
  }
}
```

If you do not want to accept a zero as the number value, then you can add a conditional response for the slot to check for zero, and tell the user that they must provide a number greater than zero. But, it is important for the slot condition to be able to recognize a zero when it is provided as input.

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	Confirmation slot	Prompt	Follow-up if found	Follow-up if not found

```
#yes || $confirmation I'm going to order you a $size pizza for delivery at
#no           $time. Should I go ahead?           Your pizza is on     see Complex
                                                its way!             response
```

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_focus` property 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**  icon for the slot to which you want to add conditional Found and Not found responses.
2. From the **More**  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 for* field 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.*

4. If you want to customize what happens next if the condition is met, then click the **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.

5. 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 **true** special condition.

6. 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 no 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.
- If digressions away are allowed, root level node conditions are checked for a match.
- Current slot level Not found conditions.

Be careful about adding conditions that always evaluate to true (such as the special conditions, true or 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](#).

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.

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](#) 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`.

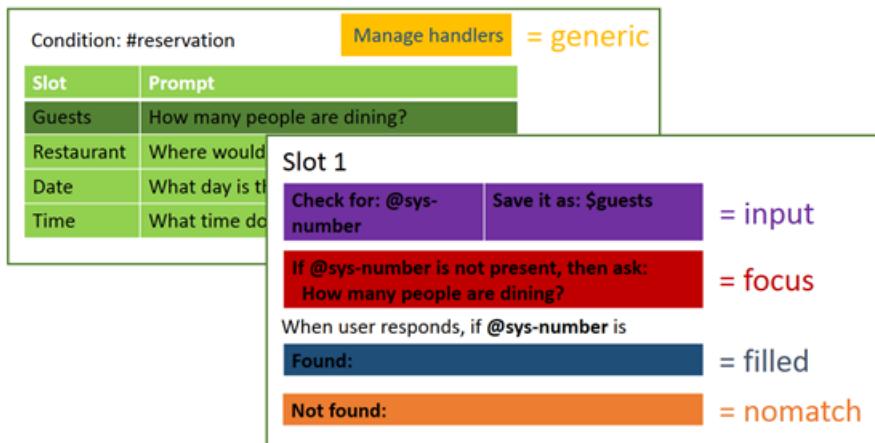
- `slot`: A child node of a node of type `frame`.

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.



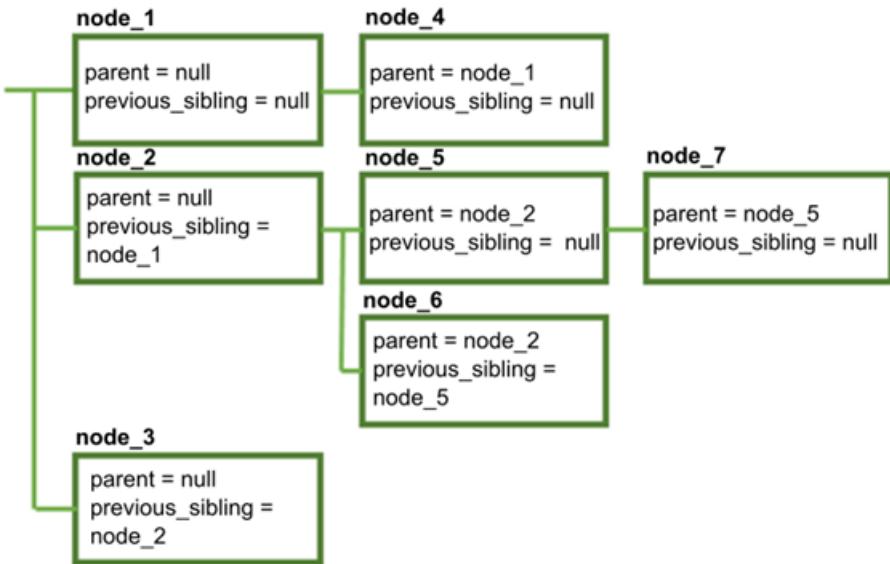
- A node of type `event_handler` with an `event_name` of `generic` can have a parent of type `slot` or `frame`.
- A node of type `event_handler` with an `event_name` of `focus`, `input`, `filled`, or `nomatch` must have a parent of type `slot`.
- If more than one `event_handler` with the same `event_name` 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`:
 1. `focus`
 2. `input`
 3. `filled`
 4. `generic*`
 5. `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

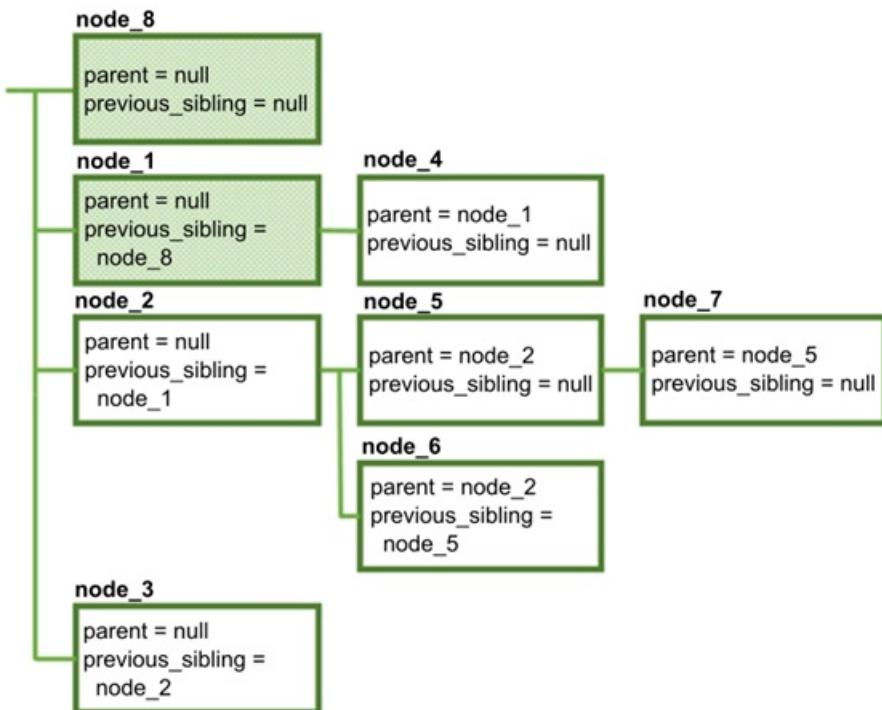
Consider the following simple dialog tree:



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

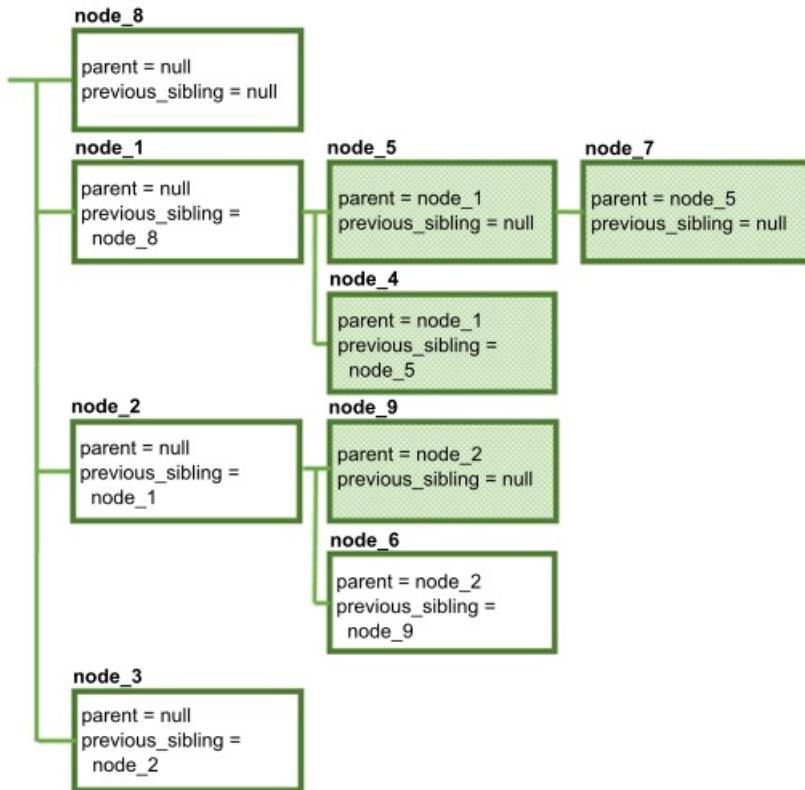
Let's move **node_5** to a different parent by using the POST /dialog_nodes/node_5 method with the following body:

```
{  
  "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:

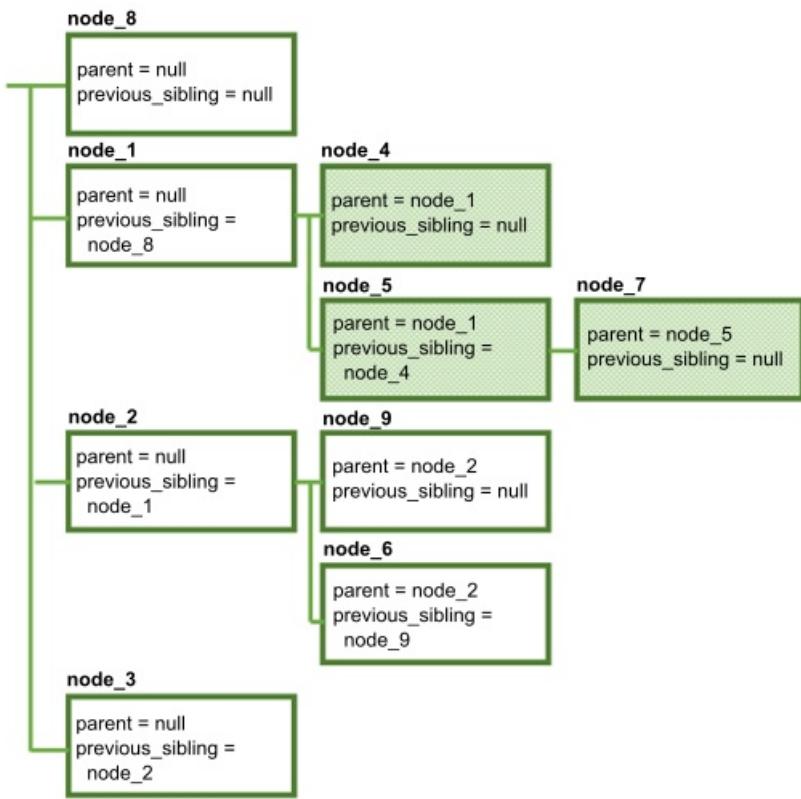
```
{  
  "previous_sibling": "node_4"  
}
```

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:

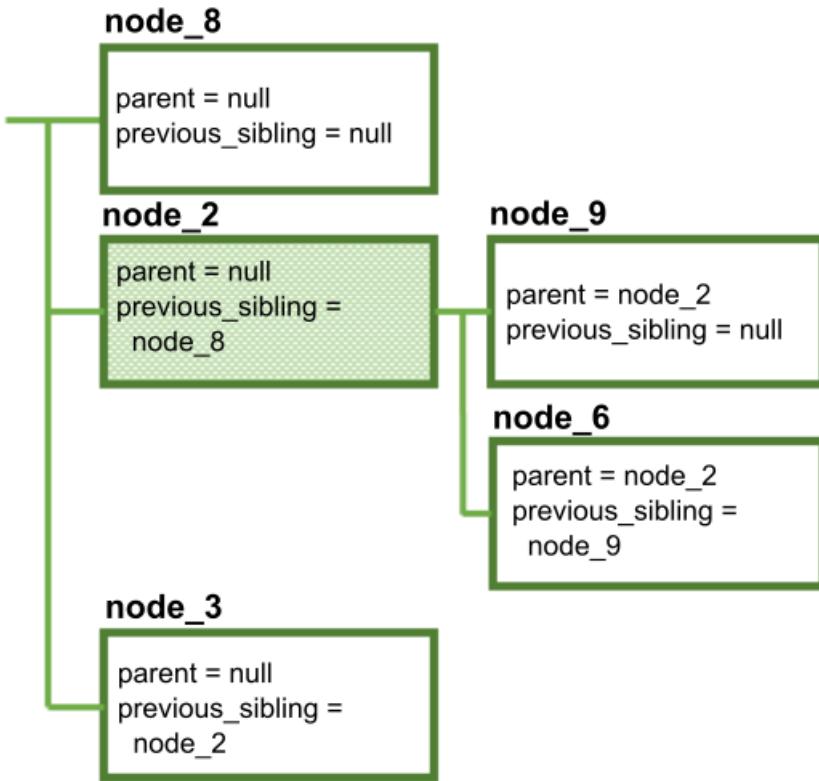


Note that once again, **node_7** stays with its parent. In addition, **node_4** is modified so that its `previous_sibling` is `null`, because it is now the first sibling.

Deleting a node

Now let's delete **node_1**, using the `DELETE /dialog_nodes/node_1` method.

The result is this:



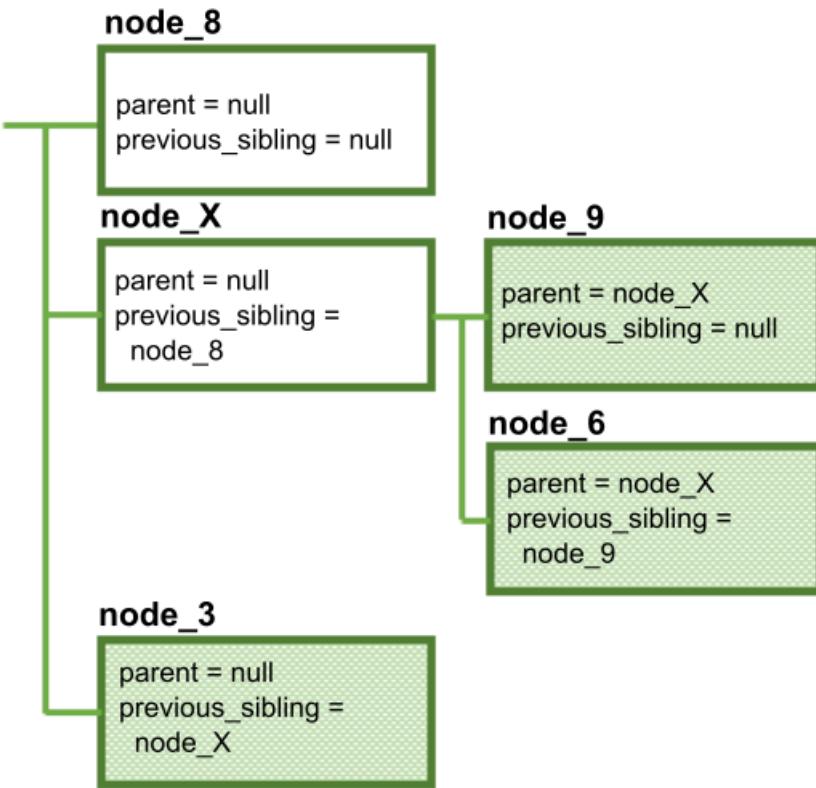
Note that **node_1**, **node_4**, **node_5**, and **node_7** were all deleted. When you delete a node, all descendants of that node are deleted as well. Therefore, if you delete a root node, you are actually deleting an entire branch of the dialog tree. Any other references to the deleted node (such 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.

Defining responses using the JSON editor

In some situations, you might need to define responses using the JSON editor. (For more information about dialog responses, see [Responses](#)). Editing the response JSON gives you direct access to the data that will be returned to the communication channel or custom application.

Generic JSON format

The generic JSON format for responses is used to specify responses that are intended for any channel. This format can accommodate various response types that are supported by Slack and Facebook integrations, and can also be implemented by a custom client application. (This is the format that is used by default for dialog responses defined using the Watson Assistant tool.)

For information about how to open the JSON editor for a dialog node response from the tool, see [Context variables in the JSON editor](#).

To specify an interactive response in the generic JSON format, insert the appropriate JSON objects into the `output.generic` field of the dialog node response. The following example shows how you might send a response containing multiple response types (text, an image, and clickable options):

{

```

"output": [
  "generic": [
    {
      "response_type": "text",
      "values": [
        {
          "text": "Here are your nearest stores."
        }
      ]
    },
    {
      "response_type": "image",
      "source": "http://example.com/image.jpg",
      "title": "Example image",
      "description": "Some description for the image."
    },
    {
      "response_type": "option",
      "title": "Click on one of the following",
      "options": [
        {
          "label": "Location 1",
          "value": {
            "input": {
              "text": "Location 1"
            }
          }
        },
        {
          "label": "Location 2",
          "value": {
            "input": {
              "text": "Location 2"
            }
          }
        },
        {
          "label": "Location 3",
          "value": {
            "input": {
              "text": "Location 3"
            }
          }
        }
      ]
    }
  ]
}

```

For more information about how to specify each supported response type using JSON objects, see [Response types](#).

If you are using the Watson Assistant connector, the response is converted at run time into the format expected by the channel (Slack or Facebook Messenger). If the response contains multiple media types or attachments, the generic response is converted into a series of separate message payloads as needed. The connector then sends each message payload to the channel in a separate message.

Note: When a response is split into multiple messages, the Watson Assistant connector sends these messages to the channel in sequence. It is the responsibility of the channel to deliver these messages to the end user; this can be affected by network or server issues.

Native JSON format

In addition to the generic JSON format, the dialog node JSON also supports channel-specific responses written using the native Slack and Facebook Messenger formats. These formats are also supported by the Watson Assistant connector. You might want to use the native JSON formats if you know your workspace will only be integrated with one channel type, and you need to specify a response type that is not currently supported by the generic JSON format.

You can specify native JSON for Slack or Facebook using the appropriate field in the dialog node response:

- `output.slack`: insert any JSON you want to be included in the `attachment` field of the Slack response. For more information about the Slack JSON format, see the Slack [documentation](#).
- `output.facebook`: insert any JSON you want included in the `message.attachment.payload` field of the Facebook response. For more information about the Facebook JSON format, see the Facebook [documentation](#).

Response types

The following response types are supported by the generic JSON format.

Image

Displays an image specified by a URL.

Fields

Name	Type	Description	Required?
<code>response_type</code>	enum <code>image</code>		Y
<code>source</code>	string	The URL of the image. The specified image must be in .jpg, .gif, or .png format.	Y
<code>title</code>	string	The title to show before the image.	N
<code>description</code>	string	The text of the description that accompanies the image.	N

Example

This example displays an image with a title and descriptive text.

```
{  
  "output": {  
    "generic": [  
      {  
        "response_type": "image",  
        "source": "http://example.com/image.jpg",  
        "title": "Example image",  
        "description": "A sample image for demonstration."  
      }  
    ]  
  }  
}
```

```

        "description": "An example image returned as part of a multimedia response."
    }
]
}

```

Option

Displays a set of buttons or a drop-down list users can use to choose an option. The specified value is then sent to the workspace as user input.

Fields

Name	Type	Description	Required?
response_type	enum	option	Y
title	string	The title to show before the options.	Y
description	string	The text of the description that accompanies the options.	N
preference	enum	The preferred type of control to display, if supported by the channel (dropdown or button). The Watson Assistant connector currently supports only button.	N
options	list	A list of key/value pairs specifying the options from which the user can choose.	Y
options[].label	string	The user-facing label for the option.	Y
options[].value	object	An object defining the response that will be sent to the Watson Assistant service if the user selects the option.	Y
options[].value.input	object	An input object that includes the input text corresponding to the option.	N
options[].value.input.text	string	The text that will be sent to the service for the option.	N

Example

This example displays two options:

- Option 1 (labeled Buy something) sends a simple string message (Place order).
- Option 2 (labeled Exit) sends a simple string message (Exit).

Whichever option is chosen by the user, its associated message is sent as input text to be evaluated by the service.

```
{
  "output": {
    "generic": [
      {
        "response_type": "option",
        "title": "Choose from the following options:",
        "options": [
          {
            "label": "Buy something",
            "value": {
              "input": {
                "text": "Place order"
              }
            }
          },
          {
            "label": "Exit",
            "value": {
              "input": {
                "text": "Exit"
              }
            }
          }
        ]
      }
    ]
  }
}
```

```

"preference": "button",
"options": [
  {
    "label": "Buy something",
    "value": {
      "input": {
        "text": "Place order"
      }
    }
  },
  {
    "label": "Exit",
    "value": {
      "input": {
        "text": "Exit"
      }
    }
  }
]
}
]
}
}

```

Pause

Pauses before sending the next message to the channel, and optionally sends a "user is typing" event (for channels that support it).

Fields

Name	Type	Description	Required?
response_type	enum	pause	Y
time	int	How long to pause, in milliseconds.	Y
typing	boolean	Whether to send the "user is typing" event during the pause. Ignored if the channel does not support this event.	N

Example

This examples sends the "user is typing" event while pausing for 5 seconds.

```
{
  "output": {
    "generic": [
      {
        "response_type": "pause",
        "time": 5000,
        "typing": true
      }
    ]
  }
}
```

```
    ]  
}  
}
```

Text

Displays text. To add variety, you can specify multiple alternative text responses. If you specify multiple responses, you can choose to rotate sequentially through the list, choose a response randomly, or output all specified responses.

Fields

Name	Type	Description	Required?
response_type	enum text		Y
values	list	A list of one or more objects defining text response.	Y
values.[n].text	string	The text of a response. This can include newline characters (\n) and Markdown tagging, if supported by the channel. (Any formatting not supported by the channel is ignored.)	N
selection_policy	string	How a response is selected from the list, if more than one response is specified. The possible values are sequential, random, and multiline.	N
delimiter	string	The delimiter to output as a separator between responses. Used only when selection_policy=multiline. The default delimiter is newline (\n).	N

Example

This examples displays a greeting message to the user.

```
{  
  "output": {  
    "generic": [  
      {  
        "response_type": "text",  
        "values": [  
          { "text": "Hello." },  
          { "text": "Hi there." }  
        ],  
        "selection_policy": "random"  
      }  
    ]  
  }  
}
```

Defining entities

Entities represent information in the user input that is relevant to the user's purpose.

If intents represent verbs (the action a user wants to do), entities represent nouns (the object of, or the context for, that action). For example, when the *intent* is to get a weather forecast, the relevant location and date *entities* are required before the application can return an accurate forecast.

Recognizing entities in the user's input helps you to craft more useful, targeted responses. For example, you might have a #buy_something intent. When a user makes a request that triggers the #buy_something intent, the assistant's response should reflect an understanding of what the *something* is that the customer wants to buy. You can add a @product entity, and then use it to extract information from the user input about the product that the customer is interested in. (The @ prepended to the entity name helps to clearly identify it as an entity.)

Finally, you can add multiple responses to your dialog tree with wording that differs based on the @product value that is detected in the user's request.

Note: The video was created in a test environment in which the workspace is referred to as a skill.

Entity creation overview

You can create the following types of entities:

- **Synonym entity:** You define a category of terms as an entity (color), and then one or more values in that category (blue). For each value you specify a bunch of synonyms (aqua, navy).

At run time, the service recognizes terms in the user input that exactly match (or, if fuzzy matching is enabled, closely match) the values or synonyms that you defined for the entity as mentions of that entity.

- **Pattern entity:** You define a category of terms as an entity (contact_info), and then one or more values in that category (email). For each value, you specify a regular expression that defines the textual pattern of mentions of that value type. For an email entity value, you might want to define a pattern like, text@text.com, for example.

At run time, the service looks for patterns matching your regular expression in the user input, and identifies any matches as mentions of that entity.

- **Contextual entity:** First, you define a category of terms as an entity (product), and then optionally one or more values (handbag). Next, you go to the *Intents* page and mine your existing intent user examples to find any mentions of the entity type, and label them as such. For example, you might go to the #buy_something intent, and find a user example that says, I want to buy a Coach bag. You can label Coach bag as a mention of the @product:handbag entity.

At run time, the service evaluates the context in which the term is used in the sentence. If the structure of a user request that mentions the term matches the structure of a user example sentence in which a mention is labeled, then the service identifies the term as a mention of that entity. For example, the user input might include the utterance, I want to buy a Gucci bag. Due to the similarity of the structure of this sentence to the user example that you annotated (I want to buy a Coach bag), the service recognizes Gucci bag as a @product:handbag entity mention.

- **System entity:** Synonym entities that are prebuilt for you by IBM. They cover commonly used categories, such as numbers, dates, and times. You simply enable a system entity to start using it.

Entity limits

Entity limits			
Entities per workspace	Entity values per workspace	Entity synonyms per workspace	Contextual entities and annotations
1,000	100,000	100,000	30 contextual entities with 3000 annotations

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 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](#) for more detail.

3. 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

Do not include spaces in the name. The name cannot be longer than 64 characters. Do not begin the name with the string sys- because it is reserved for system entities.

The tool automatically includes the @ character in the entity name, so you do not have to add one.

4. Select **Create entity**.

The screenshot shows a user interface for creating a new entity. At the top, there's a back arrow and a 'Create new entity' button. Below that, the 'Entity name' field contains '@menu'. To the right of the field is a 'Fuzzy Matching' section with a 'BETA' badge and a toggle switch that is currently off. A note below the entity name says, 'Once you've named your entity, begin by adding values, synonyms, and patterns to entities to help your bot learn and understand important details that your users mention.' A message at the bottom left says, 'No values yet.'

5. 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.

6. 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](#) topic.

Fuzzy matching

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*.

7. 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.

The screenshot shows the entity configuration interface. The 'Value name' field contains 'burger'. The 'Type' dropdown is set to 'Synonyms', with 'Enter synonym' and a '+' button next to it. Below the dropdown is a table with two rows: 'Synonyms' and 'Patterns'. A note at the bottom says, 'Once you've named your entity, begin by adding values, synonyms, and patterns to entities to help your bot learn and understand important details that your users mention.' A message at the bottom left says, 'No values yet.'

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.

The screenshot shows the 'Entity name' field set to '@animal'. The 'Fuzzy Matching' toggle is off. Below the main input, there are sections for 'Value name' (with a placeholder 'Enter value') and 'Synonyms' (with a placeholder 'Add synonym...'). Buttons for 'Add value' and 'Show recommendations' are present. A table below lists four entity values: 'amphibian', 'mammal', and 'reptile', each with its corresponding synonyms and types.

Entity Values (4)	Type
amphibian	Synonyms
mammal	Synonyms
reptile	Synonyms

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 512 characters.

The screenshot shows the 'Entity name' field set to '@ContactInfo'. The 'Fuzzy Matching' toggle is off. Below the main input, there are sections for 'Value name' (set to 'localPhone') and 'Patterns' (with a placeholder '(\\d{3})-(\\d{4})'). A button for 'Enter synonym' is also present. A table below lists four entity values: 'email', 'fullUSphone', 'internationalPhone', and 'website', each with its corresponding patterns.

Value	Type
email	Patterns
fullUSphone	Patterns
internationalPhone	Patterns
website	Patterns

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: $\b[A-Za-z0-9._%+-]+\@[A-Za-z]{2,}\b$, e.g. name@ibm.com

- website: (<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. For additional information, see [Defining a context variable](#).

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:

- Variable: email
- Value: <? @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:

- Variable: area_code
- Value: <? @fullUSphone.groups[1] ?>

For additional information about using capture groups in your dialog, see [Storing and recognizing entity pattern groups in input](#).

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 zkouška), 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.

8. Click **Add value** and repeat the process to add more entity values.

9. When you are finished adding entity values, select  to finish creating the entity.

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

Defining contextual entities BETA

When you define specific values for an entity, the service finds entity mentions only when a term in the user input exactly matches (or closely matches if fuzzy matching is enabled) a value or synonym defined. When you define a contextual entity, a model is trained on both the *annotated term* and the *context* in which the term is used in the sentences you annotate. This new contextual entity model enables the service to calculate a confidence score that identifies how likely a word or phrase is to be an instance of an entity, based on how it is used in the user input.

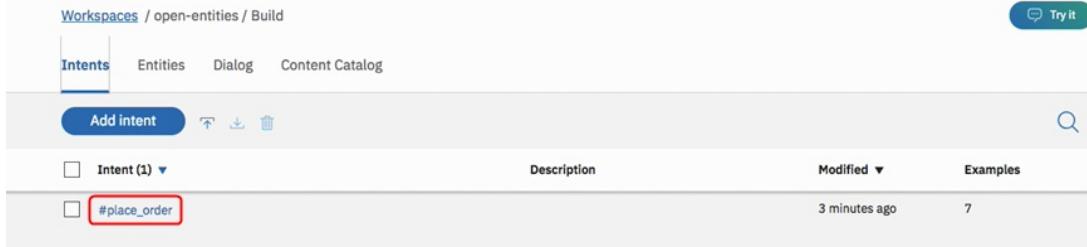
The following video demonstrates how to annotate entity mentions.

To walk through a tutorial that shows you how to define contextual entities before you add your own, go to [Tutorial: Defining contextual entities](#).

Creating contextual entities from the **Intents** tab

In order to train a contextual entity model, you can take advantage of your intent examples, which provide readily-available sentences to annotate. Using intent user examples to define contextual entities does not affect the classification of an intent.

1. In the Watson Assistant tool, open your skill and then click the **Intents** tab. If **Intents** is not visible, use the  menu to open the page.
2. Select an intent. For this example, the intent `#place_order` defines the order function for an online retailer.



The screenshot shows the Watson Assistant interface with the 'Intents' tab selected. The top navigation bar includes 'Workspaces / open-entities / Build' and a 'Try it' button. Below the navigation are tabs for 'Intents' (selected), 'Entities', 'Dialog', and 'Content Catalog'. A search bar is at the top right. The main area has a header with 'Add intent' and file operations ('New', 'Import', 'Delete'). A table lists intents, with the first row for '#place_order' highlighted by a red border. The table columns are 'Intent (1)', 'Description', 'Modified', and 'Examples'. The '#place_order' row shows '3 minutes ago' and '7' examples.

Intent (1)	Description	Modified	Examples
<input type="checkbox"/> #place_order		3 minutes ago	7

3. Review the intent examples for potential entity values. Highlight a potential entity value from the intent examples, in this case **computer**.

[|](#) #place_order

Last modified 3 minutes ago [Download](#) [Delete](#) [Try it](#)

Intent name
#place_order

Description
Add a description to this intent

Add user examples
Add user examples to this intent

[Add example](#)

User examples (7) ▾

- Can I get a computer 
- Could I get a new blue toner? 
- I'd like to get a box of plain white paper 
- I need black pens 
- I need to get a silver iphone 
- I need to order some red pens 
- I would like a maroon shirt 

Note: To directly edit an intent example, select the Edit icon  instead of highlighting a value for annotation.

4. A Search box opens, allowing you to search for an appropriate entity for the highlighted entity value.

[|](#) #place_order

Last modified 3 minutes ago [Download](#) [Delete](#) [Try it](#)

Intent name
#place_order

Description
Add a description to this intent

Add user examples
Add user examples to this intent

[Add example](#)

User examples (7) ▾

- Can I get a computer 
- Could I get  
No results found.
- I'd like to 
- I need black pens 
- I need to get a silver iphone 
- I need to order some red pens 
- I would like a maroon shirt 

5. In this example, searching prod brings up matches for both the @product entity, and for entity values shirt and pens. This is an important distinction - @product is an entity that can contain multiple entity values, in this case @product:pencil, @product:shirt and @product:pens

Intent name
#place_order

Description
Add a description to this intent

Add user examples
Add user examples to this intent

Add example

User examples (7)

- Can I get a computer
- Could I get prod
 - @product: pencil, shirt, pens
 - @(create new entity)
- I'd like to g
- I need black pens
- I need to get a silver iphone
- I need to order some red pens
- I would like a maroon shirt

You can also create a new entity by choosing @(create new entity).

6. Select @product to add computer as a value for that entity.

NOTE: You should create *at least 10* annotations for each contextual entity; more annotations are recommended for production use.

7. Now, click the annotation you just created. A box will appear with the words Go to: @product at the bottom. Clicking that link will take you directly to the entity.

Workspaces / open-entities / Build

Try it

Intents Entities Dialog Content Catalog

My entities System entities

Add entity

Entity (1)	Values	Modified
<input type="checkbox"/> @product	shirt, computer, pens	19 hours ago

To see all of the mentions you annotated for a particular entity, from the entity's configuration page, click the Annotations tab.

Working with counterexamples

If you have an intent example with an annotation, and a word in that example is part of your entity values, but the value is **not** annotated, it is considered a counterexample:

1. The #Customer_Care_Appointments intent includes two intent examples with the word visit.

#Customer_Care_Appointments

Last modified 2 days ago

Add user examples
Add user examples to this intent

[Add example](#)

User examples (20) ▾

- I'd like to make an appointment
- I prefer a face to face visit ←
- I want to talk in person with someone about my case
- I would like to discuss my situation face to face
- I would like to make an appointment to visit the nearest store to my location. ←
- Make an appointment
- Set up an appt

2. In the first example, you want to annotate the word **visit** as an entity value of the @meeting entity. This makes **visit** equivalent to other @meeting entity values such as **appointment**, as in "I'd like to make an appointment" or "I'd like to schedule a visit".

#Customer_Care_Appointments

Last modified 8 minutes ago

Add user examples
Add user examples to this intent

[Add example](#)

User examples (20) ▾

0 examples in conflict Show only conflicts

I'd like to make an appointment

I prefer a face to face visit

I want to talk in perso
meeting X

I would like to discuss
Go to @meeting X

I would like to make an appointment to visit the nearest store to my location.

Make an appointment

Set up an appt

Store appointment

3. For the second example, the word **visit** is being used in a different context than a meeting. In this case, you can select the word **appointment** from the intent example, and annotate it as an entity value of the @meeting entity. Because the word **visit** in the same example is not annotated, it will serve as a counterexample.

Last modified a few seconds ago Try it

Add user examples
Add user examples to this intent

User examples (20) ▼ 0 examples in conflict Show only conflicts 1

- i'd like to make an appointment ↗
- I prefer a face to face visit ↗
- I want to talk in person with someone about my case ↗
- I would like to discuss my situation face to face ↗
- I would like to make an appointment to visit the nearest store to my location. ↗

Make an appointment @meet Remove

Set up an appt @meeting: appointment, face to face, appt, v... @(create new entity)

Store appointment Want to change my visit ↗

Viewing annotations from the Entities tab

To see the intent examples you have used in annotating your contextual entities:

- From the **Entities** tab, open an entity, for example, `@cuisine`.

IBM Watson Assistant Try it

Workspaces / Car Dashboard - Sample / Build

Intents Entities Dialog Content Catalog

My entities System entities Add entity ▼ ↑ ↓ Delete Search

Entity (10) ▼	Values	Modified ▼
<input type="checkbox"/> @amenity	gas, place, restaurant, restroom	7 months ago
<input type="checkbox"/> @appliance	AC, fan, heater, lights, music, volume, wipers	7 months ago
<input type="checkbox"/> @cuisine	african, american, austrian, chinese, creole, french, indian, souther ...	7 months ago
<input type="checkbox"/> @genre	classical, jazz, pop, rock	7 months ago
<input type="checkbox"/> @phone	call, text	7 months ago
<input type="checkbox"/> @response_types	negative, positive, uncertain	7 months ago

- Select the *Annotation view*.

The screenshot shows the Watson Assistant Entity view for the entity '@cuisine'. At the top, there's a header with the entity name and some status indicators like 'Last modified 7 months ago' and 'Try it'. Below the header, there's a section for 'Value name' with an 'Enter value' input field, a 'Synonyms' dropdown, and a 'Add synonym...' button. To the right of this is a 'Fuzzy Matching' toggle switch set to 'Off'. The main content area is divided into two tabs: 'Dictionary' (selected) and 'Annotations'. The 'Dictionary' tab lists 21 entity values: african, american, austrian, chinese, creole, french, indian, and southern, each with a 'Type' column showing 'Synonyms'. The 'Annotations' tab is currently empty.

If you have already [created annotated entities from the Intents tab](#), you will see a list of user examples with their associated intents.

NOTE: Contextual entities understand values that you have not explicitly defined. The system makes predictions about additional entity values based on how your user examples are annotated, and uses those values to train other entities. Any similar user examples are added to the *Annotation* view, so you can see how this option impacts training.

If you do not want your contextual entities to use this expanded understanding of entity values, select all the user examples in the *Annotation* view for that entity and choose **Delete**.

The screenshot shows a modal dialog titled 'Delete' with a trash icon. It displays a list of 14 selected items under the heading 'User examples (14)'. Each item has a checkbox and its corresponding intent listed next to it. The items are: 'do you know a restaurant in new york city' (intent: #locate_amenity), 'find chinese restraint' (intent: #locate_amenity), and 'french restaurants are my fav' (intent: #locate_amenity). At the bottom of the dialog are buttons for 'Delete' and 'Cancel'.

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.

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:

```
<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.

2. Click  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.

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**.

The screenshot shows the 'Entities' tab in the Watson Assistant interface. At the top, there are tabs for 'Intents', 'Entities' (which is selected and highlighted in blue), and 'Dialog'. Below the tabs, a header bar includes 'My entities' and 'System entities' buttons, and a toolbar with 'Export' (with a dropdown arrow), 'Delete' (with a trash can icon), '1 item selected', and 'Cancel'. A message '1 item selected' is displayed above the entity list. The entity list table has columns for 'Entity', 'Values', and 'Modified'. There are five rows of data:

Entity	Values	Modified
<input type="checkbox"/> @ContactInfo	website, email, internationalPhone, fullUSphone	11 minutes ago
<input type="checkbox"/> @days_of_the_week	Friday, Monday, Saturday, Sunday, Tuesday, Wednesday, Thursday	10 days ago
<input type="checkbox"/> @diets	gluten free, halal, kosher, vegan, vegetarian	10 days ago
<input checked="" type="checkbox"/> @holidays	christmas, christmas eve, halloween, independence day, labor day, memor	10 days ago

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**.

The screenshot shows the 'Entities' tab in the Watson Assistant interface. The 'Delete' button in the toolbar is highlighted with a red box. The entity list table has columns for 'Entity', 'Values', and 'Modified'. There are five rows of data, with the first row (@ContactInfo) having its checkbox checked:

Entity	Values	Modified
<input checked="" type="checkbox"/> @ContactInfo	website, email, internationalPhone, fullUSphone	11 minutes ago
<input type="checkbox"/> @days_of_the_week	Friday, Monday, Saturday, Sunday, Tuesday, Wednesday, Thursday	10 days ago
<input type="checkbox"/> @diets	gluten free, halal, kosher, vegan, vegetarian	10 days ago
<input type="checkbox"/> @holidays	christmas, christmas eve, halloween, independence day, labor day, memor	10 days ago

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**.



2. Browse through the list of system entities to choose the ones that are useful for your application.

- o To see more information about a system entity, including examples of matching input, click the entity in the list.
- o For details about the available system entities, see [System entities](#).

3. Click the toggle switch next to a system entity to enable or disable it.

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

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 3, or Java by clicking the language selector in the upper right.

Setting up the Watson Assistant service

The example application we will create in this section connects 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](#).
2. [Import the workspace](#) 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.

For information about how to find your service credentials, see [Service information](#).

To find the workspace ID from your workspace, go to the **Workspaces** page of the Watson Assistant tool and then select **View details** from the menu of the workspace tile.

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({
  url: 'https://icp_cluster_host:{port}/assistant/api', // replace with URL
  username: '{username}', // replace with service username
  password: '{password}', // replace with service password
  version: '2018-09-20'
});

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. Assumes a single text response.
if (response.output.generic.length != 0) {
  console.log(response.output.generic[0].text);
}
}

# Example 1: sets up service wrapper, sends initial message, and
# receives response.

import watson_developer_cloud

# Set up Assistant service.
service = watson_developer_cloud.AssistantV1(
  url = 'https://{{icp_cluster_host}}:{{port}}/assistant/api', # replace with URL
  username = '{{username}}', # replace with service username
  password = '{{password}}', # replace with service password
  version = '2018-09-20'
)
workspace_id = '{{workspace_id}}' # replace with workspace ID

# Start conversation with empty message.
response = service.message(
  workspace_id = workspace_id,
  input = {
    'text': ''
  }
)

# Print the output from dialog, if any. Assumes a single text response.
if response['output']['generic']:
  print(response['output']['generic'][0]['text'])

/*
 * Example 1: sets up service wrapper, sends initial message, and
 * receives response.
 */

import com.ibm.watson.developer_cloud.assistant.v1.Assistant;
import com.ibm.watson.developer_cloud.assistant.v1.model.DialogRuntimeResponseGeneric;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageOptions;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageRequest;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageResponse;
import java.util.List;
import java.util.logging.LogManager;

public class AssistantSimpleExample {
  public static void main(String[] args) {

    // Suppress log messages in stdout.
    LogManager.getLogManager().reset();

```

```

// Set up Assistant service.
Assistant service = new Assistant("2018-09-20");
service.setEndPoint("https://{{icp_cluster_host}}:{{port}}/assistant/api"); // replace with
URL
service.setUsernameAndPassword("{username}", // replace with service username
                           "{password}"); // replace with service password
String workspaceId = "{workspace_id}"; // replace with workspace ID

// Start assistant with empty message.
MessageOptions options = new MessageOptions.Builder(workspaceId).build();
MessageResponse response = service.message(options).execute();

// Print the output from dialog, if any. Assumes a single text response.
List<DialogRuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
if(responseGeneric.size() > 0) {
    System.out.println(responseGeneric.get(0).getText());
}
}
}

```

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.

In this Python example, the wrapper is an instance of `watson_developer_cloud.AssistantV1`, stored in the variable `service`. The Watson SDKs for other languages provide equivalent mechanisms for instantiating a service wrapper.

In this Java example, the wrapper is an instance of `Assistant`, 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.

Use the `python3 <filename.py>` command to run the example application.

Paste the example code into a file named `AssistantSimpleExample.java`. You can then compile and run it.

Note: Make sure you have installed the Watson SDK for Node.js using `npm install watson-developer-cloud`.

Note: Make sure you have installed the Watson SDK for Python using `pip install --upgrade watson-developer-cloud` or `easy_install --upgrade watson-developer-cloud`.

Note: Make sure you have installed the [Watson SDK for Java](#).

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`.) We can use the Python 3 `input` function to do this. We can use the Java `Console.readLine()` function to do this.

```
// 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({
  url: 'https://{{icp_cluster_host}}:{{port}}/assistant/api', // replace with URL
  username: '{{username}}', // replace with service username
  password: '{{password}}', // replace with service password
  version: '2018-09-20'
});

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. Assumes a single text response.
  if (response.output.generic.length != 0) {
    console.log(response.output.generic[0].text);
  }

  // Prompt for the next round of input.
  var newMessageFromUser = prompt('>> ');
  service.message({
    workspace_id: workspace_id,
    input: { text: newMessageFromUser }
  }, processResponse)
}

# Example 2: adds user input and detects intents.

import watson_developer_cloud
```

```

# Set up Assistant service.
service = watson_developer_cloud.AssistantV1(
    url = 'https://{{icp_cluster_host}}:{{port}}/assistant/api', # replace with URL
    username = '{{username}}', # replace with service username
    password = '{{password}}', # replace with service password
    version = '2018-09-20'
)
workspace_id = '{{workspace_id}}' # replace with workspace ID

# Initialize with empty value to start the conversation.
user_input = ''

# Main input/output loop
while True:

    # Send message to Assistant service.
    response = service.message(
        workspace_id = workspace_id,
        input = {
            'text': user_input
        }
    )

    # If an intent was detected, print it to the console. Assumes a single text response.
    if response['intents']:
        print('Detected intent: #' + response['intents'][0]['intent'])

    # Print the output from dialog, if any.
    if response['output']['generic']:
        print(response['output']['generic'][0]['text'])

    # Prompt for next round of input.
    user_input = input('>> ')

/*
 * Example 2: adds user input and detects intents.
 */

import com.ibm.watson.developer_cloud.assistant.v1.Assistant;
import com.ibm.watson.developer_cloud.assistant.v1.model.DialogRuntimeResponseGeneric;
import com.ibm.watson.developer_cloud.assistant.v1.model.InputData;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageOptions;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageRequest;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageResponse;
import com.ibm.watson.developer_cloud.assistant.v1.model.RuntimeIntent;
import java.util.List;
import java.util.logging.LogManager;

public class AssistantSimpleExample {
    public static void main(String[] args) {

        // Suppress log messages in stdout.
        LogManager.getLogManager().reset();

        // Set up Assistant service.
        Assistant service = new Assistant("2018-09-20");

```

```

        service.setEndPoint("https://{{icp_cluster_host}}:{{port}}/assistant/api"); // replace with
URL
        service.setUsernameAndPassword("{username}", // replace with service username
                                         "{password}"); // replace with service password
        String workspaceId = "{workspace_id}"; // replace with workspace ID

        // Initialize with empty value to start the conversation.
        MessageOptions options = new MessageOptions.Builder(workspaceId).build();

        // Main input/output loop
        do {
            // Send message to Assistant service.
            MessageResponse response = service.message(options).execute();

            // If an intent was detected, print it to the console.
            List<RuntimeIntent> responseIntents = response.getIntents();
            if(responseIntents.size() > 0) {
                System.out.println("Detected intent: #" + responseIntents.get(0).getIntent());
            }

            // Print the output from dialog, if any. Assumes a single text response.
            List<DialogRuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
            if(responseGeneric.size() > 0) {
                System.out.println(responseGeneric.get(0).getText());
            }

            // Prompt for next round of input.
            System.out.print(">> ");
            String inputText = System.console().readLine();
            InputData input = new InputData.Builder(inputText).build();
            options = new MessageOptions.Builder(workspaceId).input(input).build();
        } while(true);
    }
}

```

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 prompts for the next round of user input.

It then displays any intent detected by the dialog along with the output text, and then it prompts for the next round of user input. (We're using a while True loop for now, since we haven't yet implemented a way of ending the conversation.)

It then displays any intent detected by the dialog along with the output text, and then it prompts for the next round of user input. (We're using a do-while loop with `while(true)` for now, since we haven't yet implemented a way of ending the conversation.)

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 workspace 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 dialog 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({
  url: 'https://{{icp_cluster_host}}:{port}/assistant/api',
  username: '{{username}}', // replace with service username
  password: '{{password}}', // replace with service password
  version: '2018-09-20'
});

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. Assumes a single text response.
if (response.output.generic.length != 0) {
  console.log(response.output.generic[0].text);
}

// 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)
}

# Example 3: maintains state.

import watson_developer_cloud

# Set up Assistant service.
service = watson_developer_cloud.AssistantV1(
  url = 'https://{{icp_cluster_host}}:{{port}}/assistant/api', # replace with URL
  username = '{{username}}', # replace with service username
  password = '{{password}}', # replace with service password
  version = '2018-09-20'
)
workspace_id = '{{workspace_id}}' # replace with workspace ID

# Initialize with empty value to start the conversation.
user_input = ''
context = {}

# Main input/output loop
while True:

  # Send message to Assistant service.
  response = service.message(
    workspace_id = workspace_id,
    input = {
      'text': user_input
    },
    context = context
  )

  # If an intent was detected, print it to the console.
  if response['intents']:
    print('Detected intent: #' + response['intents'][0]['intent'])

  # Print the output from dialog, if any. Assumes a single text response.
  if response['output']['generic']:
    print(response['output']['generic'][0]['text'])

```

```

# Update the stored context with the latest received from the dialog.
context = response['context']

# Prompt for next round of input.
user_input = input('>> ')

/*
 * Example 3: maintains state.
 */

import com.ibm.watson.developer_cloud.assistant.v1.Assistant;
import com.ibm.watson.developer_cloud.assistant.v1.model.Context;
import com.ibm.watson.developer_cloud.assistant.v1.model.DialogRuntimeResponseGeneric;
import com.ibm.watson.developer_cloud.assistant.v1.model.InputData;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageOptions;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageRequest;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageResponse;
import com.ibm.watson.developer_cloud.assistant.v1.model.RuntimeIntent;
import java.util.List;
import java.util.logging.LogManager;

public class AssistantSimpleExample {
    public static void main(String[] args) {

        // Suppress log messages in stdout.
        LogManager.getLogManager().reset();

        // Set up Assistant service.
        Assistant service = new Assistant("2018-09-20");
        service.setEndPoint("https://{{icp_cluster_host}}:{{port}}/assistant/api"); // replace with
URL
        service.setUsernameAndPassword("{username}", // replace with service username
                                      "{password}"); // replace with service password
        String workspaceId = "{workspace_id}"; // replace with workspace ID

        // Initialize with empty value to start the conversation.
        MessageOptions options = new MessageOptions.Builder(workspaceId).build();
        Context context = new Context();

        // Main input/output loop
        do {
            // Send message to Assistant service.
            MessageResponse response = service.message(options).execute();

            // If an intent was detected, print it to the console.
            List<RuntimeIntent> responseIntents = response.getIntents();
            if(responseIntents.size() > 0) {
                System.out.println("Detected intent: #" + responseIntents.get(0).getIntent());
            }

            // Print the output from dialog, if any. Assumes a single text response.
            List<DialogRuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
            if(responseGeneric.size() > 0) {
                System.out.println(responseGeneric.get(0).getText());
            }
        }
    }
}

```

```

// Update the stored context with the latest received from the dialog.
context = response.getContext();

// Prompt for next round of input.
System.out.print(">> ");
String inputText = System.console().readLine();
InputData input = new InputData.Builder(inputText).build();
options = new MessageOptions.Builder(workspaceId).input(input).context(context).build();
} while(true);
}
}

```

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)

```

The only change from the previous example is that we are now storing the context received from the dialog in a variable called `context`, and we're sending it back with the next round of user input:

```

response = service.message(
  workspace_id = workspace_id,
  input = {
    'text': user_input
  },
  context = context
)

```

The only change from the previous example is that we are now storing the context received from the dialog in a variable called `context`, and we're including it as part of the options sent back with the next round of user input:

```
options = new MessageOptions.Builder(workspaceId).input(input).context(context).build();
```

This ensures that the context is maintained from one turn to the next, so the Watson Assistant dialog 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 `actions` property in the response JSON to signal when the application needs to carry out an action, based on the detected intents.

The `actions` property is an array of objects representing any actions the dialog needs to invoke in response to user input. Each object in this array includes properties that describe a requested action. An action can be either a client action or a server action, indicated by the `type` property; for our purposes, we're only interested in client actions, which are actions the dialog is asking the client app to perform.

Our application code needs to check the `actions` property for requested client actions. (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({
  url: 'https://{{icp_cluster_host}}:{{port}}/assistant/api',
  username: '{{username}}', // replace with service username
  password: '{{password}}', // replace with service password
  version: '2018-09-20'
});

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 client actions requested by the dialog. Assumes at most a single
  // action.
  if (response.actions) {
    if (response.actions[0].type === 'client') {
      if (response.actions[0].name === '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.actions[0].name === 'end_conversation') {
        // User said goodbye, so we're done.
        console.log(response.output.generic[0].text);
        endConversation = true;
      }
    }
  } else {
    // Display the output from dialog, if any. Assumes a single text response.
    if (response.output.generic.length != 0) {
      console.log(response.output.generic[0].text);
    }
  }
}
```

```

    }
}

// 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)
}
}

# Example 4: implements app actions.

import watson_developer_cloud
import time

# Set up Assistant service.
service = watson_developer_cloud.AssistantV1(
  url = 'https://{{icp_cluster_host}}:{{port}}/assistant/api', # replace with URL
  username = '{{username}}', # replace with service username
  password = '{{password}}', # replace with service password
  version = '2018-09-20'
)
workspace_id = '{{workspace_id}}' # replace with workspace ID

# Initialize with empty value to start the conversation.
user_input = ''
context = {}
current_action = ''

# Main input/output loop
while current_action != 'end_conversation':

  # Send message to Assistant service.
  response = service.message(
    workspace_id = workspace_id,
    input = {
      'text': user_input
    },
    context = context
  )

  # Print the output from dialog, if any. Assumes a single text response.
  if response['output'][0]['generic']:
    print(response['output'][0]['generic'][0]['text'])

  # Update the stored context with the latest received from the dialog.
  context = response['context']
  # Check for client actions requested by the dialog.
  if 'actions' in response:
    if response['actions'][0]['type'] == 'client':
      current_action = response['actions'][0]['name']

```

```

# User asked what time it is, so we output the local system time.
if current_action == 'display_time':
    print('The current time is ' + time.strftime('%I:%M:%S %p') + '.')
# If we're not done, prompt for next round of input.
if current_action != 'end_conversation':
    user_input = input('>> ')

/*
 * Example 4: implements app actions.
 */

import com.ibm.watson.developer_cloud.assistant.v1.Assistant;
import com.ibm.watson.developer_cloud.assistant.v1.model.Context;
import com.ibm.watson.developer_cloud.assistant.v1.model.DialogRuntimeResponseGeneric;
import com.ibm.watson.developer_cloud.assistant.v1.model.InputData;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageOptions;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageRequest;
import com.ibm.watson.developer_cloud.assistant.v1.model.MessageResponse;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.List;
import java.util.Map;
import java.util.logging.LogManager;

public class AssistantSimpleExample {
    public static void main(String[] args) {

        // Suppress log messages in stdout.
        LogManager.getLogManager().reset();

        // Set up Assistant service.
        Assistant service = new Assistant("2018-09-20");
        service.setEndPoint("https://{{icp_cluster_host}}:{{port}}/assistant/api"); // replace with
URL
        service.setUsernameAndPassword("{username}", // replace with service username
                                      "{password}"); // replace with service password
        String workspaceId = "{workspace_id}"; // replace with workspace ID

        // Initialize with empty value to start the conversation.
        MessageOptions options = new MessageOptions.Builder(workspaceId).build();
        Context context = new Context();
        String currentAction = "";

        // Main input/output loop
        do {
            // Send message to Assistant service.
            MessageResponse response = service.message(options).execute();

            // Print the output from dialog, if any. Assumes a single text response.
            List<DialogRuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
            if(responseGeneric.size() > 0) {
                System.out.println(responseGeneric.get(0).getText());
            }

            // Update the stored context with the latest received from the dialog.
            context = response.getContext();
        }
    }
}

```

```

// Check for client actions requested by the dialog.
if(response.get("actions") != null) {
    List<Map<String, String>> actions = (List<Map<String, String>>)
response.get("actions");
    currentAction = actions.get(0).get("name");
} else {
    currentAction = "";
}

// User asked wht time it is, so we output the local system time.
if(currentAction.equals("display_time")) {
    DateTimeFormatter fmt = DateTimeFormatter.ofPattern("h:mm:ss a");
    LocalTime time = LocalTime.now();
    System.out.println("The current time is " + time.format(fmt) + ".");
}

// If we're not done, prompt for next round of input.
if(!currentAction.equals("end_conversation")) {
    System.out.print(">> ");
    String inputText = System.console().readLine();
    InputData input = new InputData.Builder(inputText).build();
    options = new
MessageOptions.Builder(workspaceId).input(input).context(context).build();
}

} while(!currentAction.equals("end_conversation"));
}
}

```

The `processResponse()` function now checks the `actions` property of the response JSON to see if it contains either of the possible client actions (`display_time` or `end_conversation`). If so, the application carries out the appropriate action.

The app now checks the `actions` property of the response JSON to see if it contains either of the possible client actions. If the value is `display_time`, the application carries out the appropriate action. If the value is `end_conversation`, the app knows not to prompt for more user input, and the `while` loop ends.

The app now checks the `actions` property of the response JSON to see if it contains either of the possible client actions. If the value is `display_time`, the application carries out the appropriate action. If the value is `end_conversation`, the app knows not to prompt for more user input, and the `do-while` loop ends.

```

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](#).

Upgrading

Learn how to upgrade from one version of IBM Watson™ Assistant for IBM Cloud Private to another.

Bulk feature updates are announced as they become available. You can choose whether to upgrade your instance to the latest version or not. If you want continuous updates to be applied to your service instance automatically, consider creating a service instance in the public IBM Cloud.

To upgrade your instance, complete these steps:

1. From the earlier version of the service, [export any workspaces](#) you want to keep.
2. Uninstall the previous version.
3. Install the new version of the service.
4. [Import the workspaces](#) that you exported earlier.

Tutorials

Tutorial: Building a complex dialog

In this tutorial, you will use the Watson Assistant service to create a dialog for an assistant that helps users with inquiries about a fictitious restaurant called *Truck Stop Gourmand*.

Learning objectives

By the time you finish the tutorial, you will understand how to:

- Plan a dialog
- Define custom intents
- Add dialog nodes that can handle your intents
- Add entities to make your responses more specific
- Add a pattern entity, and use it in the dialog to find patterns in user input
- Set and reference context variables

Duration

This tutorial will take approximately 2 to 3 hours to complete.

Prerequisite

Before you begin, complete the [Getting Started tutorial](#).

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.

If you do not have the workspace, you can add it to your instance by importing the [watson_assistantTutorial.json](#) file.

Step 1: Plan the dialog

You are building an assistant for a restaurant named *Truck Stop Gourmand* that has one location and a thriving cake-baking business. You want the simple assistant to answer user questions about the restaurant, its menu, and to cancel customer cake orders. Therefore, you need to create intents that handle inquiries related to the following subjects:

- Restaurant information
- Menu details
- Order cancelations

You'll start by creating intents that represent these subjects, and then build a dialog that responds to user questions about them.

Step 2: Answer questions about the restaurant

Add an intent that recognizes when customers ask for details about the restaurant itself. An intent is the purpose or goal expressed in user input. The #General_About_You intent that is provided with the *General* content catalog serves a similar function, but its user examples are designed to focus on queries about the assistant as opposed to the business that is using the assistant to help its customers. So, you will add your own intent.

Add the #about_restaurant intent

1. Click the **Intents** tab.

2. Click **Add intent**.

The screenshot shows the IBM Watson Assistant interface. At the top, there's a header bar with the text "IBM Watson Assistant". Below it is a navigation bar with four tabs: "Workspaces" (highlighted in blue), "Watson Assistant tutorial / Build", "Intents" (highlighted in blue), "Entities", "Dialog", and "Content Catalog". On the left side, there's a sidebar with three icons: a wrench and screwdriver, a circular arrow, and a line graph. In the center, there's a button labeled "Add intent" with up and down arrows and a trash bin icon. The main area is currently empty, showing a light gray background.

3. Enter about_restaurant in the *Intent name* field, and then click **Create intent**.

The screenshot shows the "Create new intent" dialog box. At the top, there's a back arrow, the text "Create new intent", and a "Try it" button with a speech bubble icon. Below that is a "Intent name" input field containing "#about_restaurant". To the right of the input field is a magnifying glass search icon. Underneath the input field is a "Description" section with the placeholder "Add a description to this intent". At the bottom of the dialog is a "Create intent" button. The main area below the dialog shows the message "No examples yet." and the instruction "Train your virtual assistant with this intent by adding unique examples of what your users would say." The sidebar on the left is identical to the one in the previous screenshot.

4. Add the following user examples:

Tell me about the restaurant

i want to know about you

who are the restaurant owners and what is their philosophy?

What's your story?

Where do you source your produce from?

Who is your head chef and what is the chef's background?

How many locations do you have?

do you cater or host functions on site?

Do you deliver?

Are you open for breakfast?

5. Click the **Close** icon to finish adding the #about_restaurant intent.

You added an intent and provided examples of utterances that real users might enter to trigger this intent.

Add a dialog node that is triggered by the #about_restaurant intent

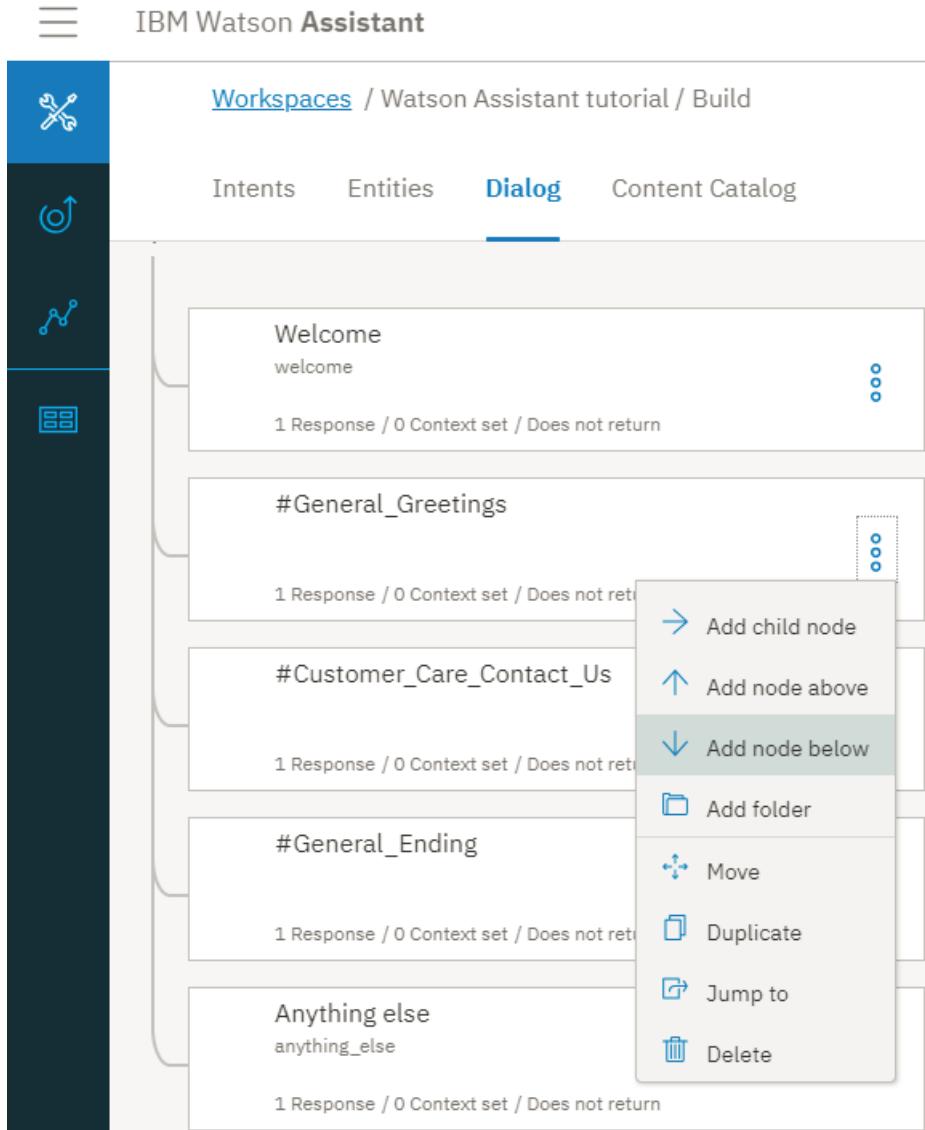
Add a dialog node that recognizes when the user input maps to the intent that you created in the previous step, meaning its condition checks whether the service recognized the #about_restaurant intent from the user input.

1. Click the **Dialogs** tab.

2. Find the #General_Greetings node in the dialog tree.

You will add a node that checks for questions about the restaurant below this initial greeting node to reflect the flow you might expect to encounter in a normal conversation. For example, Hello. then Tell me about yourself.

3. Click the More  icon on the #General_Greetings node, and then select Add node below.



The screenshot shows the IBM Watson Assistant interface. The top navigation bar includes 'Workspaces / Watson Assistant tutorial / Build', 'Intents', 'Entities', 'Dialog' (which is selected), and 'Content Catalog'. On the left, there's a sidebar with icons for 'Workspaces', 'Intents', 'Entities', 'Dialog', and 'Content Catalog'. The main area displays a list of dialog nodes:

- Welcome
- welcome
- 1 Response / 0 Context set / Does not return
- #General_Greetings
- 1 Response / 0 Context set / Does not return
- #Customer_Care_Contact_Us
- 1 Response / 0 Context set / Does not return
- #General_Ending
- 1 Response / 0 Context set / Does not return
- Anything else
- anything_else
- 1 Response / 0 Context set / Does not return

A context menu is open over the #General_Greetings node, listing the following options:

- Add child node
- Add node above
- Add node below** (this option is highlighted)
- Add folder
- Move
- Duplicate
- Jump to
- Delete

4. Start to type #about_restaurant into the Enter a condition field of this node. Then select the #about_restaurant option.

5. Add the following text as the response:

Truck Stop Gourmand is the brain child of Gloria and Fred Smith. What started out as a food truck in 2004 has expanded into a thriving restaurant. We now have one brick and mortar restaurant in downtown Portland. The bigger kitchen brought with it new chefs, but each one is faithful to the philosophy that made the Smith food truck so popular to begin with: deliver fresh, local produce in inventive and delicious ways. Join us for lunch or dinner seven days a week. Or order a cake from our bakery.

6. Let's add an image to the response also.

Click **Add response type**. Select **Image** from the drop-down list. In the **Image source** field, add <https://www.ibmlearningcenter.com/wp-content/uploads/2018/02/IBM-Learning-Center-Food4.jpg>.

7. Move the image response type up, so it is displayed in the response before the text is displayed. Click the **Move up** arrow to reorder the two response types.

The screenshot shows the IBM Watson Assistant interface in the 'Dialog' workspace. A node named '#about_restaurant' is selected. The response section contains two items:

- Image:** Set to 'Image'. Title (optional) is 'Add title text'. Description (optional) is 'Add image description'. Image source is set to <https://www.ibmlearningcenter.com/wp-content/uploads/2018/02/IBM-Learning-Center-Food4.j>.
- Text:** Set to 'Text'. Utterance is 'Truck Stop Gourmand is the brain child of Gloria and Fred Smith. What started out as'. Variation is 'Enter response variation'. Note: Response variations are set to sequential. Set to random | multiline.

8. Click to close the edit view.

Test the #about_restaurant dialog node

Test the intent by checking whether user utterances that are similar to, but not exactly the same as, the examples you added to the training data have successfully trained the service to recognize input with an #about_restaurant intent.

1. Click the icon to open the "Try it out" pane.

2. Enter, I want to learn more about your restaurant.

The service indicates that the #about_restaurant intent is recognized, and returns a response with the image and text that you specified for the dialog node.

Try it out Clear Manage Context 1 X

i want to learn more about your restaurant

#about_restaurant



Truck Stop Gourmand is the brain child of Gloria and Fred Smith. What started out as a food truck in 2004 has expanded into a thriving restaurant. We now have one brick and mortar restaurant in downtown Portland. The bigger kitchen brought with it new chefs, but each one is faithful to the philosophy that made the Smith food truck so popular to begin with: deliver fresh, local produce in inventive and delicious ways. Join us for lunch or dinner seven days a week. Or order a cake from our bakery.

Enter something to test your virtual assistant
Use the up key for most recent

Congratulations! You have added a custom intent, and a dialog node that knows how to handle it.

The #about_restaurant intent is designed to recognize a variety of general questions about the restaurant. You added a single node to capture such questions. The response is long, but it is a single statement that can potentially answer questions about all of the following topics:

- The restaurant owners
- The restaurant history
- The philosophy
- The number of sites
- The days of operation
- The meals served
- The fact that the restaurant bakes cakes to order

For general, low-hanging fruit types of questions, a single, general answer is suitable.

Step 3: Answer questions about the menu

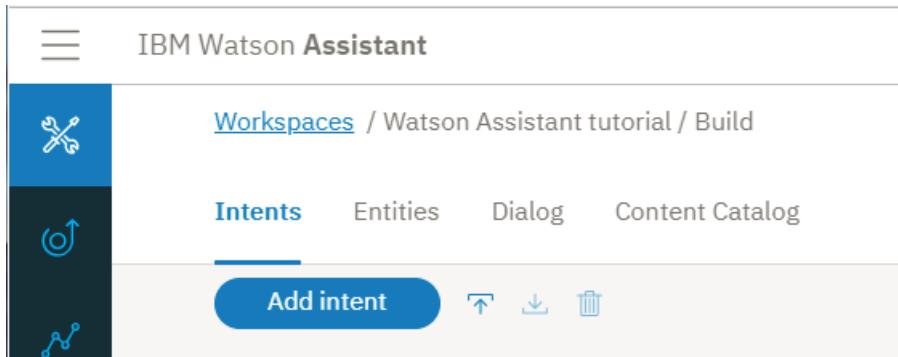
A key question from potential restaurant customers is about the menu. The Truck Stop Gourmand restaurant changes the

menu daily. In addition to its standard menu, it has vegetarian and cake shop menus. When a user asks about the menu, the dialog needs to find out which menu to share, and then provide a hyperlink to the menu that is kept up to date daily on the restaurant's website. You never want to hard-code information into a dialog node if that information changes regularly.

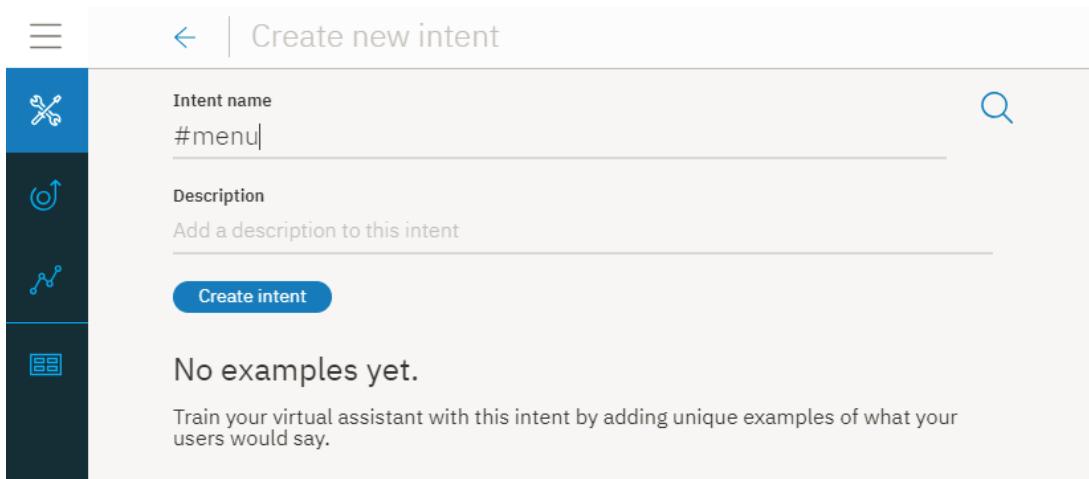
Add a #menu intent

1. Click the **Intents** tab.

2. Click **Add intent**.



3. Enter menu in the *Intent name* field, and then click **Create intent**.



4. Add the following user examples:

I want to see a menu
What do you have for food?
Are there any specials today?
where can i find out about your cuisine?
What dishes do you have?
What are the choices for appetizers?
do you serve desserts?
What is the price range of your meals?
How much does a typical dish cost?
tell me the entree choices
Do you offer a prix fixe option?

5. Click the **Close** icon to finish adding the #menu intent.

Add a dialog node that is triggered by the #menu intent

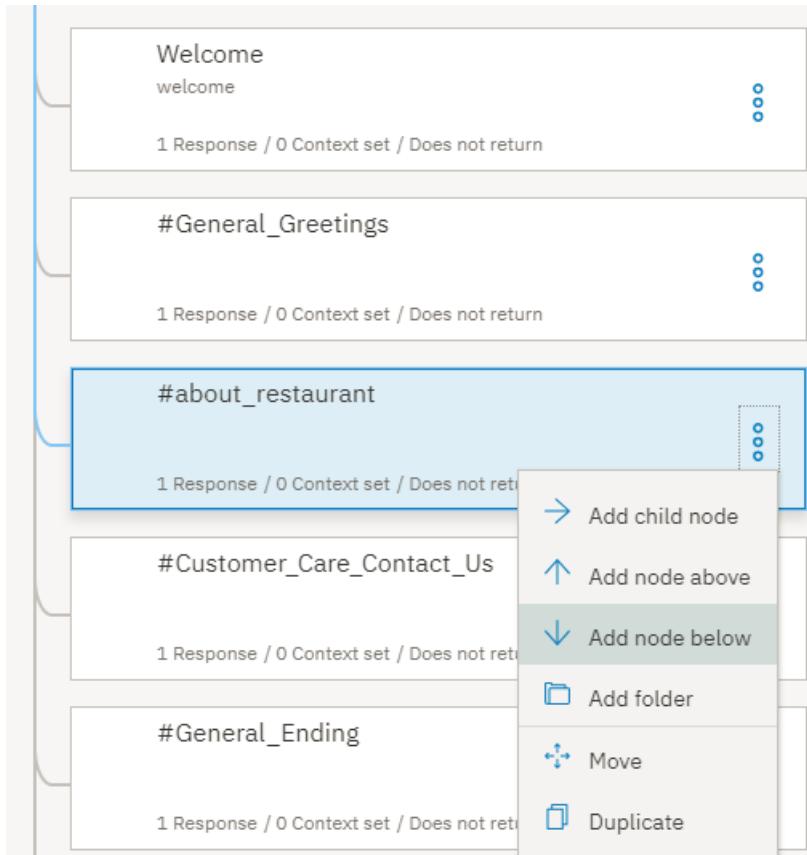
Add a dialog node that recognizes when the user input maps to the intent that you created in the previous step, meaning its condition checks whether the service recognized the #menu intent from the user input.

1. Click the **Dialogs** tab.

2. Find the #about_restaurant node in the dialog tree.

You will add a node that checks for questions about the menu below this node.

3. Click the **More**  icon on the #about_restaurant node, and then select **Add node below**.



4. Start to type #menu into the **Enter a condition** field of this node. Then select the #menu option.

If bot recognizes:

#me  

Intents:

#menu I want to see a menu

#me Create new intent

5. Add the following text as the response:

In keeping with our commitment to giving you only fresh local ingredients, our menu changes daily to accommodate the produce we pick up in the morning. You can find today's menu on our website.

6. Add an *option* response type that provides a list of options for the user to choose from. In this case, the list of options includes the different versions of the menu that are available.

Click **Add response type**. Select **Option** from the drop-down list.

Then respond with:

The screenshot shows the Watson Assistant configuration interface. At the top, there is a dropdown menu set to "Text". To the right of the dropdown are "Move" controls (up, down, delete) and a three-dot menu icon. Below the dropdown is a text input field containing the message: "In keeping with our commitment to giving you only fresh local ingredients, our menu". A small "minus" icon is to the right of the text. Underneath the text input is a placeholder "Enter response variation". At the bottom of the main panel, a note says "Response variations are set to sequential. Set to random | multiline" with an info icon. On the left side, there is a sidebar titled "+ Add response type" which lists four options: "Text" (selected), "Option" (highlighted in blue), "Pause", and "Image".

7. In the **Title** field, add *Which menu do you want to see?*

The screenshot shows the continuation of the Watson Assistant configuration. The top section is identical to the previous one, with a "Text" response type selected. The main panel contains the same message: "In keeping with our commitment to giving you only fresh local ingredients, our menu". Below the message is a "Move" control and a "minus" icon. The "Enter response variation" section is present. The "Response variations are set to sequential. Set to random | multiline" note is also here. On the left, the "+ Add response type" sidebar is visible, with "Option" selected. The main panel now shows the configuration for the "Option" response type. It has fields for "Title" (containing "Which menu do you want to see?") and "Description (optional)" (containing "Add description"). There are also "List label" and "Value" sections, both currently empty ("No options"). At the bottom of this panel is a prominent blue button with a plus sign and the text "+ Add option", which is highlighted with a red rectangular box. At the very bottom of the interface, there is a link "+ Add response type".

8. Click **Add option**.

9. In the **Label** field, add **Standard**. The text you add as the label is displayed in the response to the user as a selectable option.

10. In the **Value** field, add **standard menu**. The text you specify as the value is what gets sent to the service as new user input when a user chooses this option from the list, and clicks it.

11. Repeat the previous two steps to add label and value information for the remaining menu types:

Option response type details

Label **Value**

Vegetarian **vegetarian menu**

Cake shop **cake shop menu**

List label	Value
1 Standard	standard menu
2 Vegetarian	vegetarian menu
3 Cake shop	cake shop menu

Move: ▲ ▼ ✖

+ Add option

12. Click to close the edit view.

Add a @menu entity

To recognize the different types of menus that customers indicate they want to see, you will add a @menu entity. Entities represent a class of object or a data type that is relevant to a user's purpose. By checking for the presence of specific entities in the user input, you can add more responses, each one tailored to address a distinct user request. In this case, you will add a @menu entity that can distinguish between different menu types.

1. Click the **Entities** tab.

The screenshot shows the 'Entities' tab selected in the IBM Watson Assistant interface. A large message 'No entities yet.' is displayed with a circular icon containing a network graph symbol above it. Below the message, a descriptive text explains what entities are and how they help a virtual assistant learn. At the bottom right, there are 'Import' and 'Add entity' buttons.

2. Click **Add entity**.

3. Enter menu into the entity name field.

The screenshot shows the 'Create new entity' dialog. The 'Entity name' field contains the text '@menu'. A 'Create entity' button is visible at the bottom.

4. Click **Create entity**.

5. Add standard to the *Value name* field, and then add **standard** menu to the **Synonyms** field, and press Enter.

6. Add the following additional synonyms:

- bill of fare
- cuisine
- carte du jour

Last modified a minute ago Try it

Fuzzy Matching BETA Off ? Q

Add value Show recommendations

7. Click **Add value** to add the @menu:standard value.
8. Add vegetarian to the *Value name* field, and then add vegetarian menu to the **Synonyms** field, and press Enter.
9. Click **Show recommendations**, and then click the checkboxes for *meatless diet*, *meatless*, and *vegan diet*.
10. Click **Add selected**.
11. Click the empty *Add synonym* field, and then add these additional synonyms:

- vegan
- plants-only

Last modified a few seconds ago Try it

Fuzzy Matching BETA Off ? Q

Add value Show recommendations

12. Click **Add value** to add the @menu:vegetarian value.
13. Add cake to the *Value name* field, and then add cake menu to the **Synonyms** field, and press Enter.
14. Add the following additional synonyms:

- cake shop menu
- dessert menu
- bakery offerings

Last modified a few seconds ago Try it

Fuzzy Matching BETA Off ? Q

Add value Show recommendations

15. Click **Add value** to add the @menu:cake value.
16. Click the **Close** icon to finish adding the @menu entity.

Add child nodes that are triggered by the @menu entity types

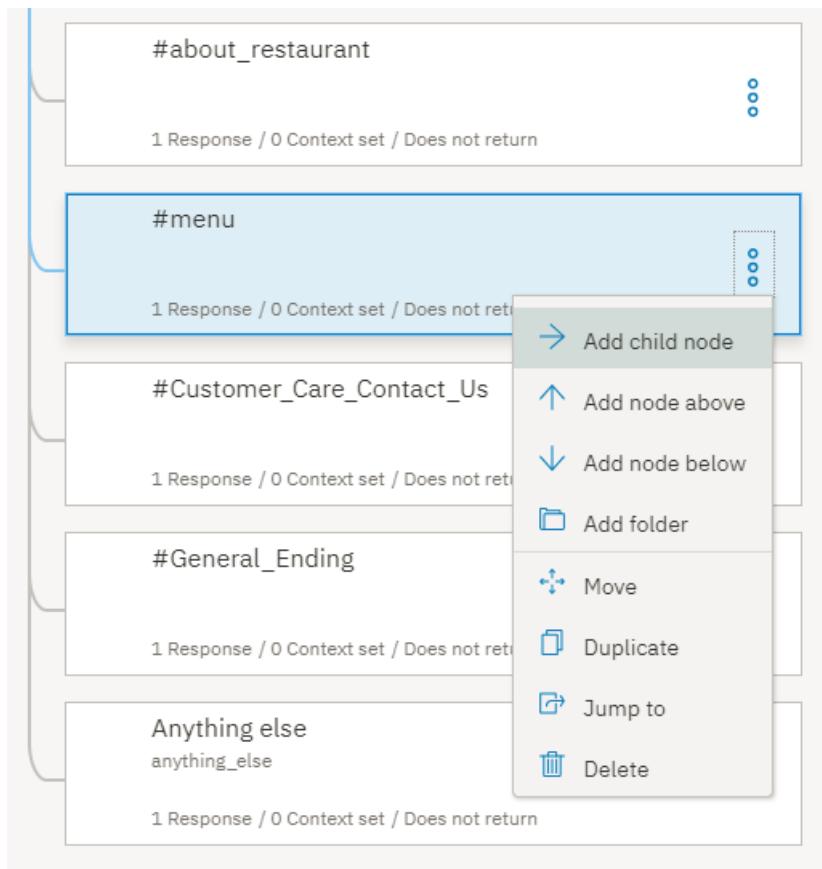
In this step, you will add child nodes to the dialog node that checks for the #menu intent. Each child node will show a different response depending on the @menu entity type the user chooses from the options list.

1. Click the **Dialogs** tab.

2. Find the #menu node in the dialog tree.

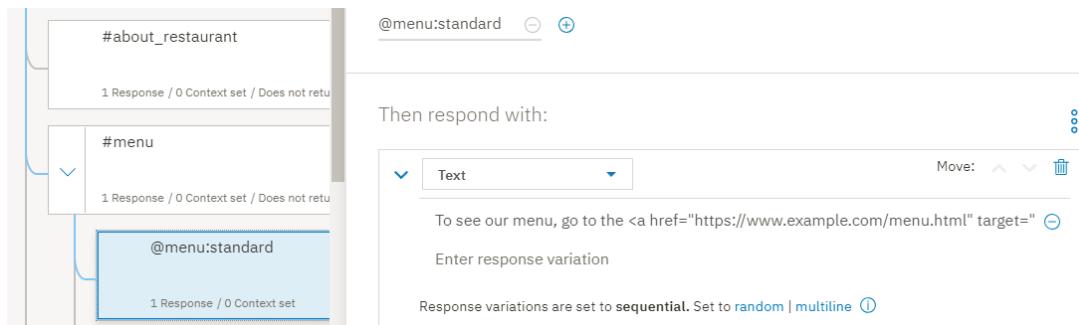
You will add a child node to handle each menu type option that you added to the #menu node.

3. Click the **More**  icon on the #menu node, and then select **Add child node**.



4. Start to type @menu:standard into the **Enter a condition** field of this node. Then select the @menu:standard option.

5. Add the following message in the response text field, To see our menu, go to the menu page on our website.



6. Click  to close the edit view.

7. Click the **More** icon on the @menu:standard node, and then select **Add node below**.
8. Start to type @menu:vegetarian into the **Enter a condition** field of this node. Then select the @menu:vegetarian option.
9. Add the following message in the response text field, To see our vegetarian menu, go to the vegetarian menu page on our website.

The screenshot shows the Watson Assistant dialog builder interface. On the left, a tree view of nodes under '#menu':

- #menu (1 Response / 0 Context set / Does not return)
- @menu:standard (1 Response / 0 Context set)
- @menu:vegetarian (1 Response / 0 Context set)

On the right, the configuration for the @menu:vegetarian node:

If bot recognizes: @menu:vegetarian

Then respond with:

Text: To see our vegetarian menu, go to the <a href="https://www.example.com/vegetaria...

10. Click **X** to close the edit view.
11. Click the **More** icon on the @menu:vegetarian node, and then select **Add node below**.
12. Start to type @menu:cake into the **Enter a condition** field of this node. Then select the @menu:cake option.
13. Add the following message in the response text field, To see our cake shop menu, go to the cake shop menu page on our website.

The screenshot shows the Watson Assistant dialog builder interface. On the left, a tree view of nodes under '#menu':

- #menu (1 Response / 0 Context set / Does not return)
- @menu:standard (1 Response / 0 Context set)
- @menu:vegetarian (1 Response / 0 Context set)
- @menu:cake (1 Response / 0 Context set)

On the right, the configuration for the @menu:cake node:

If bot recognizes: @menu:cake

Then respond with:

Text: To see our cake shop menu, go to the <a href="https://www.example.com/menu.htm...

Enter response variation:

14. Click **X** to close the edit view.
15. The standard menu is likely to be requested most often, so move it to the bottom of the child nodes list. Placing it last can help prevent it from being triggered accidentally when someone asks for a specialty menu instead the standard menu.
16. Click the **More** icon on the @menu:standard node, and then select **Move**.

Select where you want to move the node to.

1 Response / 0 Context set / Does not return

@menu:standard

1 Response / 0 Context set

@menu:vegetarian

1 Response / 0 Context set

@menu:cake

1 Response / 0 Context set

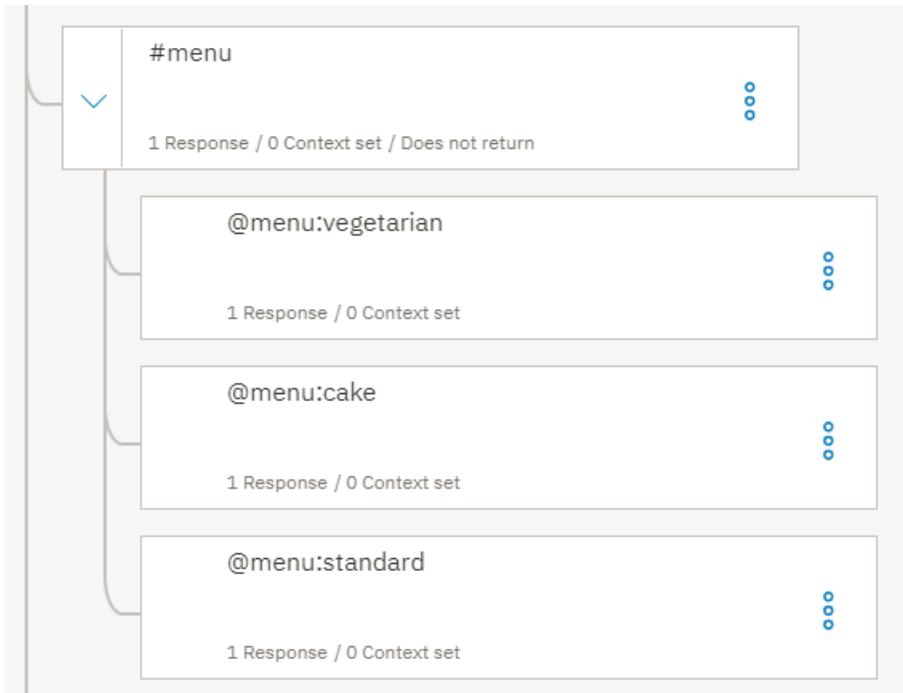
#Customer_Care_Contact_Us

1 Response / 0 Context set / Does not return

Move to...

- As child node
- Above node
- Below node

17. Select the @menu:cake node, and then choose **Below node**.



You have added nodes that recognize user requests for menu details. Your response informs the user that there are three types of menus available, and asks them to choose one. When the user chooses a menu type, a response is displayed that provides a hypertext link to a web page with the requested menu details.

Test the menu options dialog nodes

Test the dialog nodes that you added to recognize menu questions.

1. Click the  icon to open the "Try it out" pane.
2. Enter, What type of food do you serve?

The service indicates that the #menu intent is recognized, and displays the list of menu options for the user to choose from.

Try it out Clear Manage Context 1 X

Welcome to the Watson Assistant tutorial!

what type of food do you serve?

#menu ▾

In keeping with our commitment to giving you only fresh local ingredients, our menu changes daily to accommodate the produce we pick up in the morning. You can find today's menu on our website.

Which menu do you want to see?

- Standard
- Vegetarian
- Cake shop

3. Click the *Cake shop* option.

The service recognizes the `#menu` intent and `@menu:cake` entity reference, and displays the response, To see our cake shop menu, go to the cake shop page on our website.

Which menu do you want to see?

- Standard
- Vegetarian
- Cake shop

cake shop menu

#menu ▾

@menu:cake

To see our cake shop menu, go to the [cake shop menu](#) page on our website.

4. Click the *cake shop* hyperlink in the response.

A new web browser page opens and displays the example.com website.

5. Close the web browser page.

Well done. You have successfully added an intent and entity that can recognize user requests for menu details, and can direct users to the appropriate menu.

The #menu intent represents a common, key need of potential restaurant customers. Due to its importance and popularity, you added a more complex section to the dialog to address it well.

Step 4: Manage cake orders

Customers place orders in person, over the phone, or by using the order form on the website. After the order is placed, users can cancel the order through the virtual assistant. First, define an entity that can recognize order numbers. Then, add an intent that recognizes when users want to cancel a cake order.

Adding an order number pattern entity

You want the assistant to recognize order numbers, so you will create a pattern entity to recognize the unique format that the restaurant uses to identify its orders. The syntax of order numbers used by the restaurant's bakery is 2 upper-case letters followed by 5 numbers. For example, YR34663. Add an entity that can recognize this character pattern.

1. Click the **Entities** tab.
2. Click **Add entity**.
3. Enter `order_number` into the entity name field.
4. Click **Create entity**.

The screenshot shows the Entity creation interface in Watson Assistant. The left sidebar has icons for Entities, Flows, and Models. The main area has a header with a back arrow, the entity name '@order-number', and a 'Try it' button. Below the header are fields for 'Entity name' (@order-number), 'Value name' (Enter value), 'Synonyms' (dropdown), and 'Fuzzy Matching' (BETA toggle off). A message at the bottom says 'No values yet.' and 'Once you've named your entity, begin by adding values, synonyms, and patterns to entities to help your virtual assistant learn and understand important details that your users mention.'

5. Add `order_syntax` to the *Value name* field, and then click the down arrow next to **Synonyms** to change the type to **Patterns**.

The screenshot shows the entity configuration screen for '@order_number'. On the left is a sidebar with icons for edit, undo, redo, and a grid. The main area has 'Entity name' set to '@order_number' and 'Value name' set to 'order_syntax'. A dropdown menu is open over the 'order_syntax' field, showing options: 'Synonyms' (selected), 'Synonyms', and 'Patterns'. Below the value name are buttons for 'Add value' and 'Show re...'.

6. Add the following regular expression to the Pattern field: [A-Z]{2}\d{5}

The screenshot shows the entity configuration screen for '@order-number'. The 'Value name' is 'order_syntax'. Under the 'Patterns' dropdown, the regular expression '[A-Z]{2}\d{5}' is listed. Below the patterns section, it says 'No values yet.' and provides instructions for naming entities.

7. Click **Add value**.

The screenshot shows the entity configuration screen for '@order-number'. The 'Value name' is 'order_syntax'. Under the 'Synonyms' dropdown, there is a link 'Add synonym...'. Below the synonyms section, it shows 'Entity values (1)' with 'order_syntax' listed under 'Type'.

8. Click the **Close** icon to finish adding the @order_number entity.

The screenshot shows the Watson Assistant interface with the 'Entities' tab selected. On the left is a sidebar with icons for Workspaces, Intents, Entities, Dialog, and Content Catalog. The main area has tabs for 'My entities' and 'System entities'. Below is a search bar and a button to 'Add entity'. A table lists two entities:

	Entity (2) ▾	Values	Modified ▾
<input type="checkbox"/>	@menu	vegetarian, cake, standard	an hour ago
<input type="checkbox"/>	@order_number	order_syntax	a minute ago

Add a cancel order intent

1. Click the **Intents** tab.
2. Click **Add intent**.
3. Enter `cancel_order` in the *Intent name* field, and then click **Create intent**.
4. Add the following user examples:

I want to cancel my cake order
I need to cancel an order I just placed
Can I cancel my cake order?
I'd like to cancel my order
There's been a change. I need to cancel my bakery order.
please cancel the birthday cake order I placed last week
The party theme changed; we don't need a cake anymore
that order i placed, i need to cancel it.

User examples (8) ▾

- Can I cancel my cake order? edit
- I'd like to cancel my order edit
- I need to cancel an order I just placed edit
- I want to cancel my cake order edit
- please cancel the birthday cake order I placed last week edit
- that order i placed, i need to cancel it. edit
- The party theme changed; we don't need a cake anymore edit
- There's been a change. I need to cancel my bakery order. edit

5. Click the **Close** ← icon to finish adding the `#cancel_order` intent.

Add a yes intent

Before you perform an action on the user's behalf, you must get confirmation that you are taking the proper action. Add a `#yes` intent to the dialog that can recognize when a user agrees with what the service is proposing.

1. Click the **Intents** tab.
2. Click **Add intent**.
3. Enter `yes` in the *Intent name* field, and then click **Create intent**.
4. Add the following user examples:

Yes
Correct
Please do.
You've got it right.
Please do that.
that is correct.
That's right
yeah
Yup
Yes, I'd like to go ahead with that.

The screenshot shows the Watson Assistant interface with the '#yes' intent selected. On the left is a sidebar with icons for Home, Create, Examples, and Settings. The main area has a back arrow, the intent name '#yes', and a note 'Last modified a few minutes ago'. A text input field says 'Add user examples to this intent' with a placeholder 'Add example'. Below it is a list of 10 user examples, each with a checkbox and a pencil icon for editing:

- User examples (10) ▾
- Correct ✓
- Please do. ✓
- Please do that. ✓
- That is correct. ✓
- That's right ✓
- yeah ✓
- Yes ✓
- Yes, I'd like to go ahead with that. ✓
- You've got it right. ✓
- Yup ✓

5. Click the **Close** icon to finish adding the #yes intent.

Add dialog nodes that can manage requests to cancel an order

Now, add a dialog node that can handle requests to cancel a cake order.

1. Click the **Dialog** tab.
2. Find the `#menu` node. Click the **More** icon on the `#menu` node, and then select **Add node below**.
3. Start to type `#cancel_order` into the **Enter a condition** field of this node. Then select the `#cancel_order` option.
4. Add the following message in the response text field:

If the pickup time is more than 48 hours from now, you can cancel your order.

The screenshot shows the Watson Assistant Build interface. On the left is a sidebar with icons for Workspaces, Intents, Entities, Dialog (which is selected), and Content Catalog. The main area has a header with 'Workspaces / Watson Assistant tutorial / Build' and a 'Try it' button. Below the header, tabs for 'Intents', 'Entities', 'Dialog' (highlighted in blue), and 'Content Catalog' are visible. A 'Name this node...' input field contains 'If bot recognizes:'. Underneath, a condition '#cancel_order' is listed with a minus sign and a plus sign. A 'Then respond with:' section follows, containing a 'Text' dropdown set to 'Text' with a minus sign and a plus sign. Below the dropdown is the response 'If the pickup time is more than 48 hours from now, you can cancel your order.' To the right of the response is a trash icon and a three-dot menu icon.

Before you can actually cancel the order, you need to know the order number. The user might specify the order number in the original request. So, to avoid asking for the order number again, check for a number with the order number pattern in the original input. To do so, define a context variable that would save the order number if it is specified.

5. Open the context editor. Click the More icon, and select **Open context editor**.

The screenshot shows the same Watson Assistant interface as above. The 'Then respond with:' section is expanded, revealing a 'Text' dropdown set to 'Text' with a minus sign and a plus sign. Below the dropdown is the response 'If the pickup time is more than 48 hours from now, you can cancel your order.' To the right of the response is a trash icon and a three-dot menu icon. A context menu is open over the three-dot menu icon, listing 'Open JSON editor' and 'Open context editor'. The 'Open context editor' option is highlighted with a grey background.

6. Enter the following context variable name and value pair:

Order number context variable details

Variable **Value**

\$ordernumber<? @order_number.literal ?>

The context variable value (<? @order_number.literal ?>) is a SpEL expression that captures the number that the user specifies that matches the pattern defined by the @order_number pattern entity. It saves it to the \$ordernumber variable.

If bot recognizes:

#cancel_order - +

Then set context:

Variable	Value	More
\$ ordernumber	<? @order_number.literal ?>	-
+ Add variable		

And respond with:

▼Text▼Move: ^ v --

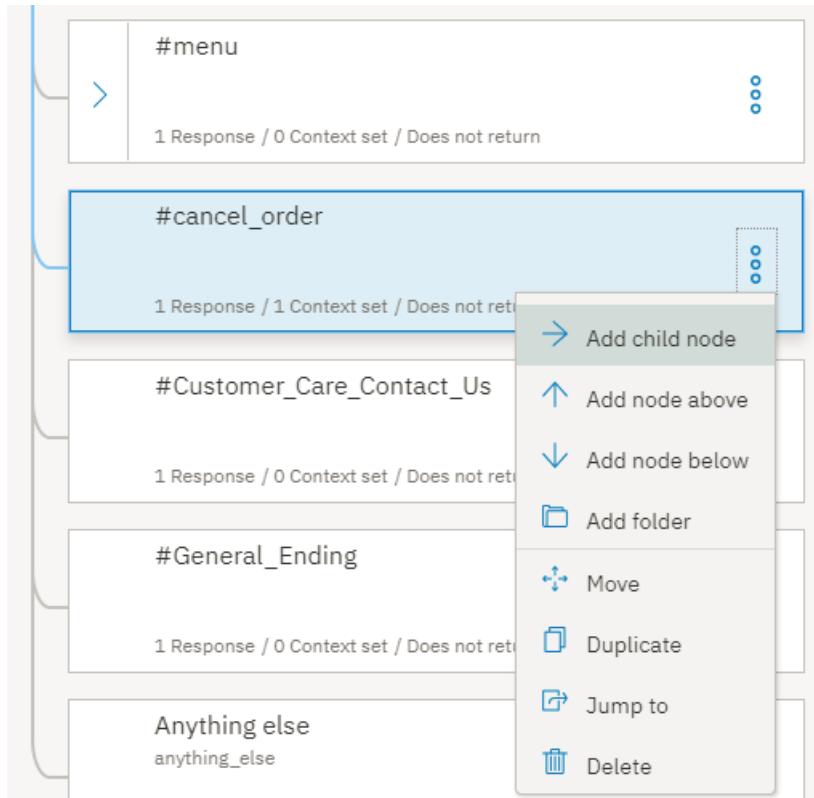
If the pickup time is more than 48 hours from now, you can cancel your order.

-

7. Click X to close the edit view.

Now, add child nodes that either ask for the order number or get confirmation from the user that she wants to cancel an order with the detected order number.

8. Click the **More** : icon on the `#cancel_order` node, and then select **Add child node**.



9. Add a label to the node to distinguish it from other child nodes you will be adding. In the name field, add `Ask for`

`order number`. Type `true` into the **Enter a condition** field of this node.

10. Add the following message in the response text field:

What is the order number?

The screenshot shows a flowchart on the left and configuration details on the right. The flowchart starts with a node labeled '#menu' (1 Response / 0 Context set / Does not return). This leads to a node labeled 'Ask for order number' (1 Response / 0 Context set / Does not return). From this node, two paths emerge: one labeled '#cancel_order' (1 Response / 1 Context set / Does not return) and another labeled 'Ask for order number' (1 Response / 0 Context set / Does not return). This second path leads to a node labeled '#Customer_Care_Contact_Us' (1 Response / 0 Context set / Does not return), which finally leads to a node labeled '#General_Ending'. On the right, under 'Ask for order number', there is a condition 'If bot recognizes: true'. Below this, the 'Then respond with:' section contains a 'Text' input field containing the message 'What is the order number?'. A note below the input field states: 'Response variations are set to sequential. Set to random | multiline'.

11. Click to close the edit view.

Now, add another child node that informs the user that you are canceling the order.

12. Click the **More** icon on the Ask for order number node, and then select **Add child node**.

13. Type `@order_number` into the **Enter a condition** field of this node.

14. Open the context editor. Click the **More** icon, and select **Open context editor**.

15. Enter the following context variable name and value pair:

Order number context variable details

Variable	Value
----------	-------

\$ordernumber<? @order_number.literal ?>
--

The context variable value (`<? @order_number.literal ?>`) is a SpEL expression that captures the number that the user specifies that matches the pattern defined by the `@order_number` pattern entity. It saves it to the `$ordernumber` variable.

16. Add the following message in the response text field:

Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a cake for you sometime soon.

The screenshot shows the Watson Assistant interface. On the left is a node editor with nodes for '#menu', '#cancel_order' (expanded), 'Ask for order number true', '@order_number', and '#Customer_Care_Contact_Us'. The '#cancel_order' node has a condition 'true' and a response '1 Response / 1 Context set / Return context'. The '@order_number' node has a response '1 Response / 1 Context set'. On the right, there are configuration panels:

- Name this node...**: A text input field with placeholder 'Name this node...' and a 'Customize' button.
- @order_number**: A context variable entry with a delete icon and a plus icon.
- Then set context:** A table for setting variables. It has one row with 'Variable' '\$ordernumber' and 'Value' '<? @order_number.literal ?>'. There are buttons for '+ Add variable' and a trash bin.
- And respond with:** A section for responses. It shows a dropdown menu set to 'Text', a 'Move' button, and a text input field containing 'Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a new one soon.' with a delete icon.

17. Click to close the edit view.
18. Add another node to capture the case where a user provides a number, but it is not a valid order number. Click the **More** icon on the @order_number node, and then select **Add node below**.
19. Type **true** into the **Enter a condition** field of this node.
20. Add the following message in the response text field:

I need the order number to cancel the order for you. If you don't know the order number, please call us at 958-234-3456 to cancel over the phone.

The screenshot shows the Watson Assistant interface. On the left is a node editor with nodes for '#cancel_order' (expanded), 'Ask for order number true', '@order_number', and 'true' (selected). The 'true' node has a response '1 Response / 0 Context set'. On the right, there are configuration panels:

- If bot recognizes:** A table with one row for 'true' with a delete icon and a plus icon.
- Then respond with:** A section for responses. It shows a dropdown menu set to 'Text', a 'Move' button, and a text input field containing 'I need the order number to cancel the order for you. If you don't know the order num' with a delete icon. Below it is a note 'Enter response variation' and a note 'Response variations are set to sequential. Set to random | multiline '. There is also a '+ Add response type' button.

21. Click to close the edit view.
22. Add a node below the initial order cancellation request node that responds in the case where the user provides the order number in the initial request, so you don't have to ask for it again. Click the **More** icon on the #cancel_order node, and then select **Add child node**.
23. Add a label to the node to distinguish it from other child nodes. In the name field, add **Number provided**. Type **@order_number** into the **Enter a condition** field of this node.
24. Add the following message in the response text field:

Just to confirm, you want to cancel order \$ordernumber?

The screenshot shows the Watson Assistant interface with a node tree on the left and a configuration panel on the right.

Node Tree:

- #cancel_order
 - 1 Response / 1 Context set / Does not return
 - Ask for order number
 - true
 - 1 Response / 0 Context set / Return all
 - @order_number
 - 1 Response / 1 Context set
 - true
 - 1 Response / 0 Context set
 - Number provided
 - @order_number
 - 1 Response / 0 Context set

Configuration Panel:

- Number provided**: A text input field containing "Number provided".
- If bot recognizes:** A condition field with "@order_number" selected.
- Then respond with:**
 - A **Text** node with the message: "Just to confirm, you want to cancel order \$ordernumber?"
 - An **Enter response variation** section.
 - A note stating: "Response variations are set to sequential. Set to random | multiline ⓘ".
 - A link to "[Add response type](#)".

25. Click to close the edit view.

You must add child nodes that check for the user's response to your confirmation question.

26. Click the **More** icon on the Number provided node, and then select **Add child node**.

27. Type #yes into the **Enter a condition** field of this node.

28. Add the following message in the response text field:

Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a cake for you sometime soon.

The screenshot shows the Watson Assistant interface with a node tree on the left and a configuration panel on the right.

Node Tree:

- #cancel_order
 - 1 Response / 1 Context set / Does not return
 - Ask for order number
 - true
 - 1 Response / 0 Context set / Return all
 - Number provided
 - @order_number
 - 1 Response / 0 Context set / Return all
 - #yes
 - 1 Response / 0 Context set

Configuration Panel:

- #yes**: A condition field with "#yes" selected.
- Then respond with:**
 - A **Text** node with the message: "Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a cake for you sometime soon."
 - An **Enter response variation** section.
 - A note stating: "Response variations are set to sequential. Set to random | multiline ⓘ".
 - A link to "[Add response type](#)".

29. Click to close the edit view.

30. Click the **More** icon on the #yes node, and then select **Add node below**.

31. Type true into the **Enter a condition** field of this node.

Do not add a response. Instead, you will redirect users to the branch that asks for the order number details that you created earlier.

32. In the **And finally** section, choose **Jump-to**.

The screenshot shows the Watson Assistant node editor interface. On the left, a vertical stack of nodes is displayed:

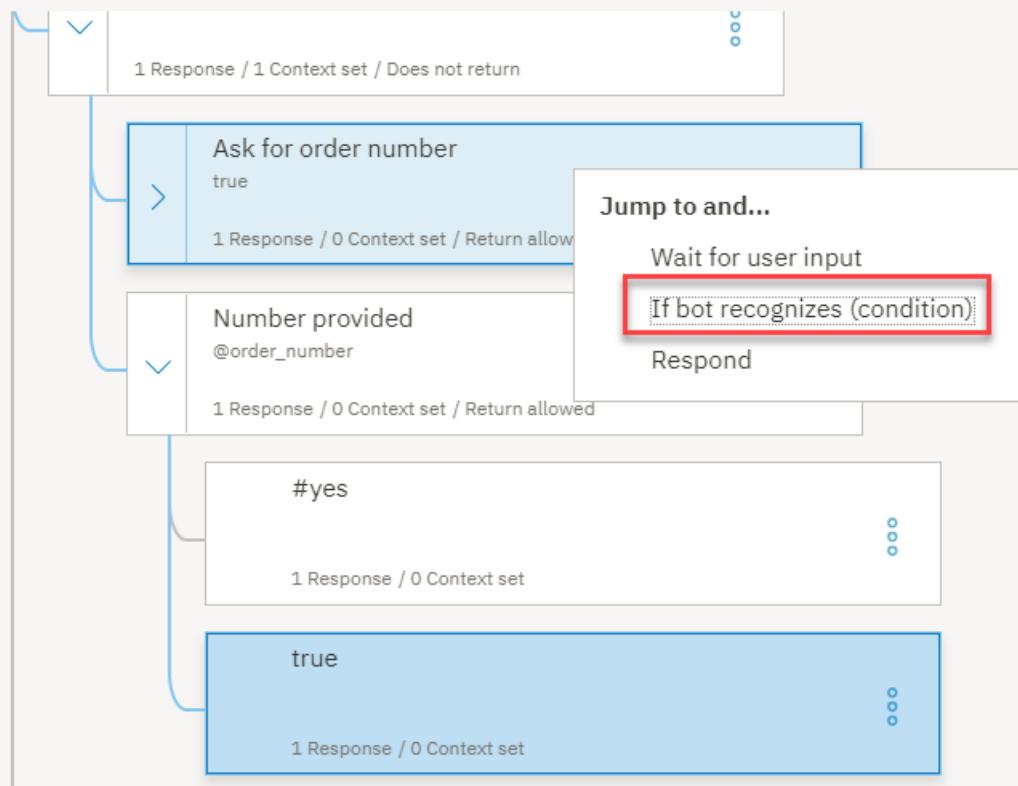
- #cancel_order (selected node, expanded):
 - Condition: 1 Response / 1 Context set / Does not return
 - Action: Ask for order number (true)
 - Condition: 1 Response / 0 Context set / Return all
 - Action: Number provided (@order_number)
 - Condition: 1 Response / 0 Context set / Return all
 - Action: #yes
 - Condition: 1 Response / 0 Context set
 - Action: true
 - Condition: 1 Response / 0 Context set- #Customer_Care_Contact_Us (unselected node)

On the right, there are several configuration panels:

- Name this node... (text input field)
- Then respond with:
 - Type: Text (dropdown menu)
 - Enter response text (text input field)
 - Response variations are set to sequential (text)
 - + Add response type (button)
- And finally:
 - Wait for user input (dropdown menu, currently selected)
 - Wait for user input (disabled state)
 - Skip user input (disabled state)
 - Jump to... (disabled state)

33. Select the *Ask for order number* node's condition.

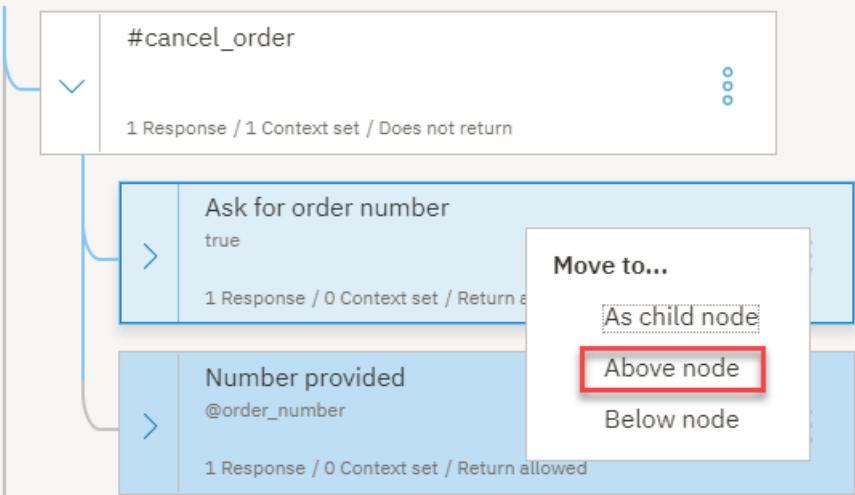
Select a destination node (origin: true)



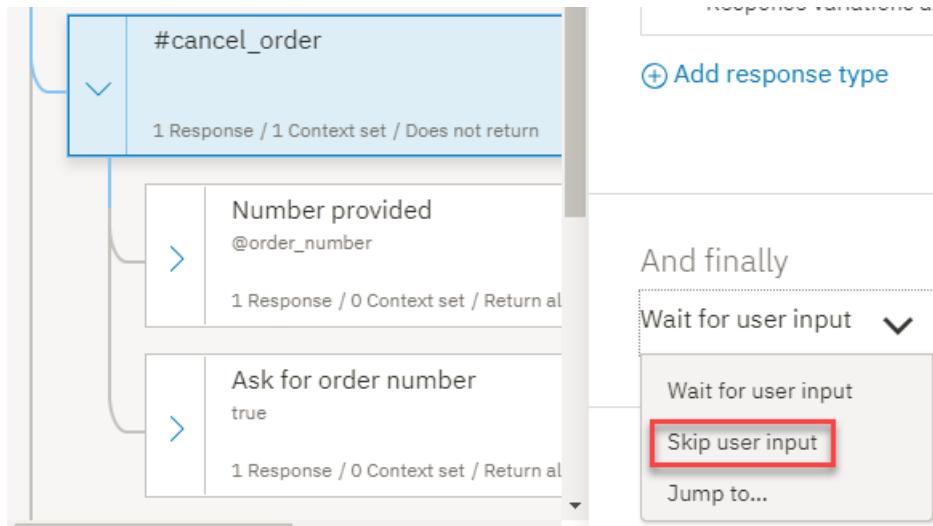
34. Click to close the edit view.

35. Move the *Number provided* node above the *Ask for order number* node. Click the **More** icon on the *Number provided* node, and then select **Move**. Select the *Ask for order number* node, and then click **Above node**.

Select where you want to move the node to.



36. Force the conversation to evaluate the child nodes under the `#cancel_order` node at run time. Click to open the `#cancel_order` node in the edit view, and then, in the `And finally` section, select `Skip user input`.



Test order cancellations

Test whether the service can recognize character patterns that match the pattern used for product order numbers in user input.



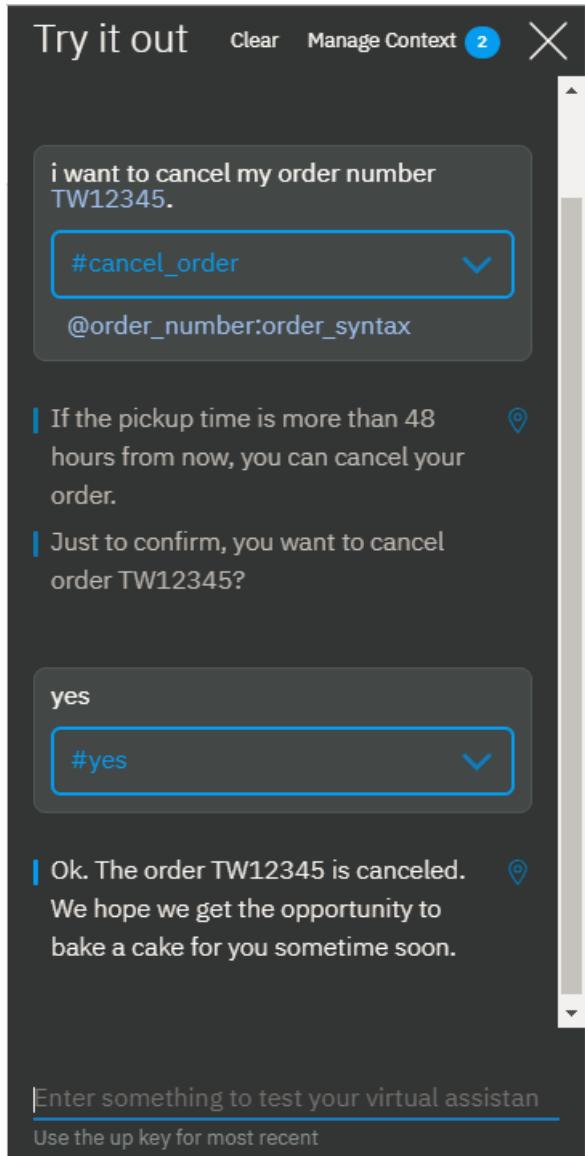
1. Click the icon to open the "Try it out" pane.

2. Enter, i want to cancel my order number TW12345.

The service recognizes both the `#cancel_order` intent and the `@order_number` entity. It responds with, If the pickup time is more than 48 hours from now, you can cancel your order. Just to confirm, you want to cancel order TW12345?

3. Enter, Yes.

The service recognizes the `#yes` intent and responds with, Ok. The order TW12345 is canceled. We hope we get the opportunity to bake a cake for you sometime soon.



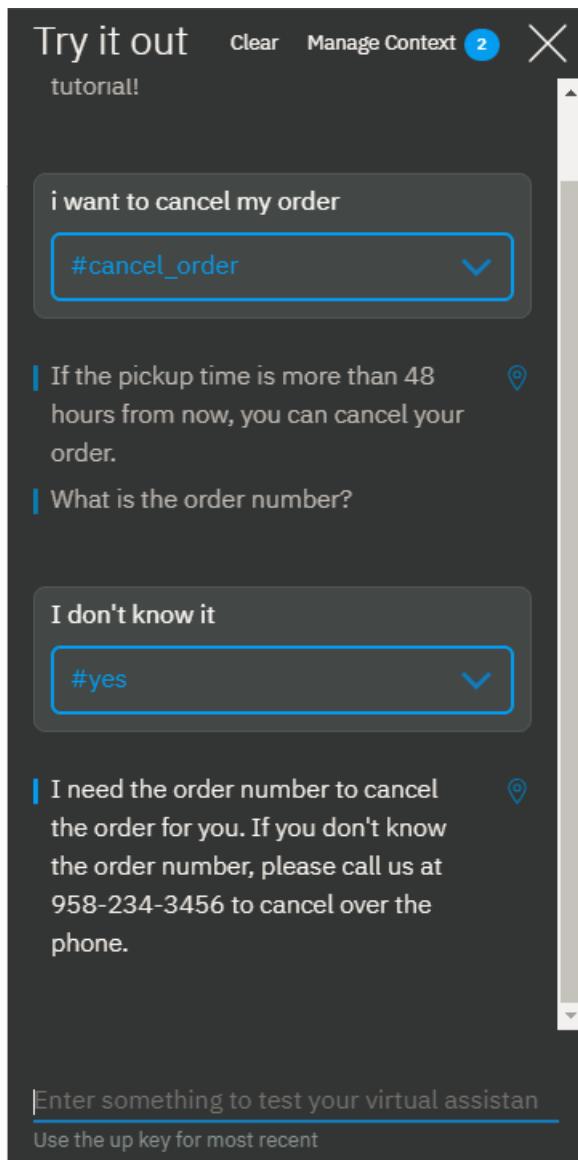
Now, try it when you don't know the order number.

4. Click **Clear** in the "Try it out" pane to start over. Enter, I want to cancel my order.

The service recognizes the `#cancel_order` intent, and responds with, If the pickup time is more than 48 hours from now, you can cancel your order. What is the order number?

5. Enter, I don't know.

The service responds with, I need the order number to cancel the order for you. If you don't know the order number, please call us at 958-234-3456 to cancel over the phone.



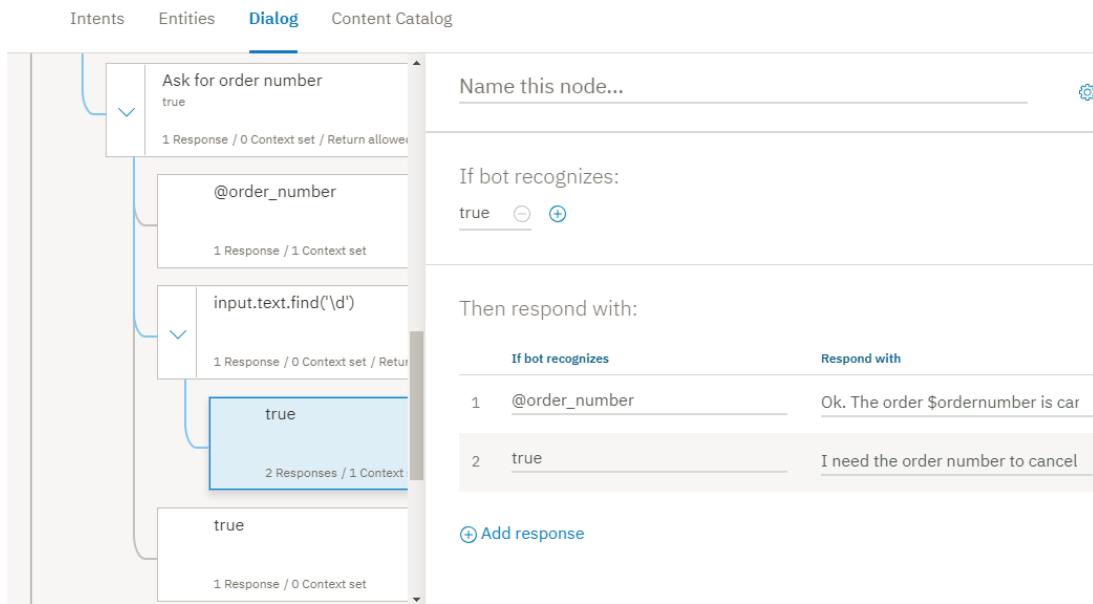
If you do more testing, you might find that the dialog isn't very helpful in scenarios where the user does not remember the order number format. The user might include only the numbers or the letters too, but forget that they are meant to be uppercase. So, it would be a nice touch to give them a hint in such cases, right? If you want to be kind, add another node to the dialog tree that checks for numbers in the user input.

1. Find the @order-number node that is a child of the *Ask order number* node.
2. Click the **More**  icon on the @order-number node, and then select **Add node below**.
3. In the condition field, add `input.text.find('\d')`, which is a SpEL expression that says if you find one or more numbers in the user input, trigger this response.
4. In the text response field, add the following response:

The correct format for our order numbers is AAnnnnn. The A's represents 2 upper-case letters, and the n's represents 5 numbers. Do you have an order number in that format?

5. Click  to close the edit view.
6. Click the **More**  icon on the `input.text.find('\d')` node, and then select **Add child node**.

7. Type **true** into the **Enter a condition** field of this node.
8. Enable conditional responses by clicking **Customize**, and then switching the *Multiple responses* toggle to **on**.
9. Click **Apply**.
10. In the newly-added *If bot recognizes* field, type **@order_number**, and in the *Respond with* field, type:
Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a cake for you sometime soon.
11. Click **Add response**.
12. In the *If bot recognizes* field, type **true**, and in the *Respond with* field, type:
I need the order number to cancel the order for you. If you don't know the order number, please call us at 958-234-3456 to cancel over the phone.



13. Click **X** to close the edit view.

Now, when you test, you can provide a set of numbers or a mix of numbers and text as input, and the dialog reminds you of the correct order number format. You have successfully tested your dialog, found a weakness in it, and corrected it.

Another way you can address this type of scenario is to add a node with slots. See the [Adding a node with slots to a dialog](#) tutorial to learn more about using slots.

Step 5: Add the personal touch

If the user shows interest in the bot itself, you want the virtual assistant to recognize that curiosity and engage with the user in a more personal way. You might remember the `#General_About_You` intent, which is provided with the *General* content catalog, that we considered using earlier, before you added your own custom `#about_restaurant` intent. It is built to recognize just such questions from the user. Add a node that conditions on this intent. In your response, you can ask for the user's name and save it to a `$username` variable that you can use elsewhere in the dialog, if available.

First, you need to make sure the service will recognize a name if the user provides one. So, you can enable the `@sys-person` entity, which is designed to recognize common first and last names (in English).

Add a person system entity

The service provides a number of *system entities*, which are common entities that you can use for any application.

1. Click the **Entities** tab, and then click **System entities**.
2. Find the `@sys-person` entity toggle, and then switch it **On**.

The screenshot shows the IBM Watson Assistant interface with the 'Entities' tab selected. In the 'System entities' section, the '@sys-person' entity is listed with its toggle switch turned 'On'. Other entities like '@sys-percentage' and '@sys-time' are also listed with their toggles off.

Add a node that handles questions about the bot

Now, add a dialog node that can recognize the user's interest in the bot, and respond.

1. Click the **Dialogs** tab.
2. Find the **Welcome** node in the dialog tree.
3. Click the **More** icon on the **Welcome** node, and then select **Add node below**.
4. Start to type `#General_About_You` into the **Enter a condition** field of this node. Then select the `#General_About_You` option.
5. Add the following message in the response text field:

I am a virtual assistant that is designed to answer your questions about the Truck Stop Gourmand restaurant. What's your name?

The screenshot shows the IBM Watson Assistant interface with the 'Dialog' tab selected. A new node named '#General_About_You' has been added below the 'Welcome' node. The configuration for the '#General_About_You' node includes the condition '#General_About_You' and the response 'I am a chat bot that is designed to answer your questions about the Truck Stop Gourmand restaurant.'

6. Click to close the edit view.
7. Click the **More** icon on the `#General_About_You` node, and then select **Add child node**.
8. Start to type `@sys-person` into the **Enter a condition** field of this node. Then select the `@sys-person` option.
9. Add the following message in the response text field:

Hello, <? `@sys-person.literal` ?>! It's lovely to meet you. How can I help you today.

10. To capture the name that the user provides, add a context variable to the node. Click the **More**  icon, and select **Open context editor**.

11. Enter the following context variable name and value pair:

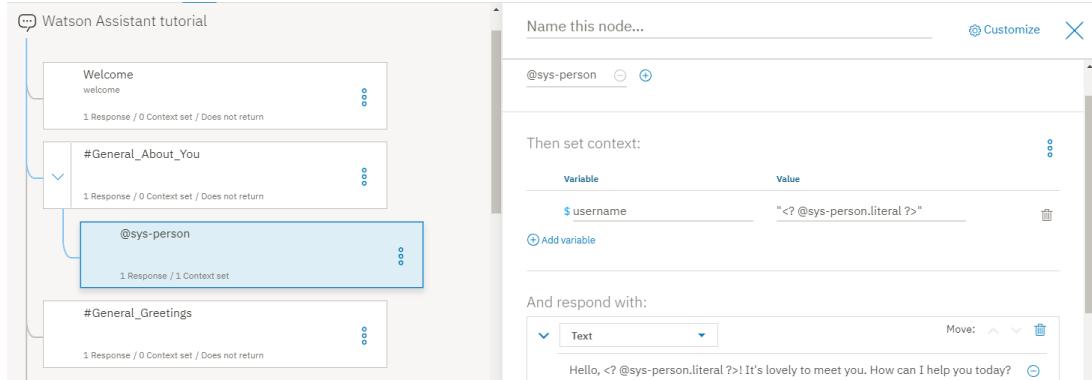
User name context variable details

Variable

Value

\$username <? @sys-person.literal ?>

The context variable value (<? @sys-person.literal ?>) is a SpEL expression that captures the user name as it is specified by the user, and then saves it to the \$username context variable.



The screenshot shows the Watson Assistant interface. On the left, a tree view of nodes: 'Welcome' (1 Response / 0 Context set / Does not return), '#General_About_You' (1 Response / 0 Context set / Does not return), '@sys-person' (1 Response / 1 Context set, highlighted in blue), '#General_Greetings' (1 Response / 0 Context set / Does not return). On the right, the 'Name this node...' field has '@sys-person' entered. Below it, 'Then set context:' shows a table with a single row: Variable '\$username' and Value '<? @sys-person.literal ?>'. Under 'And respond with:', there is a 'Text' dropdown containing the message 'Hello, <? @sys-person.literal ?>! It's lovely to meet you. How can I help you today?'.

12. Click  to close the edit view.

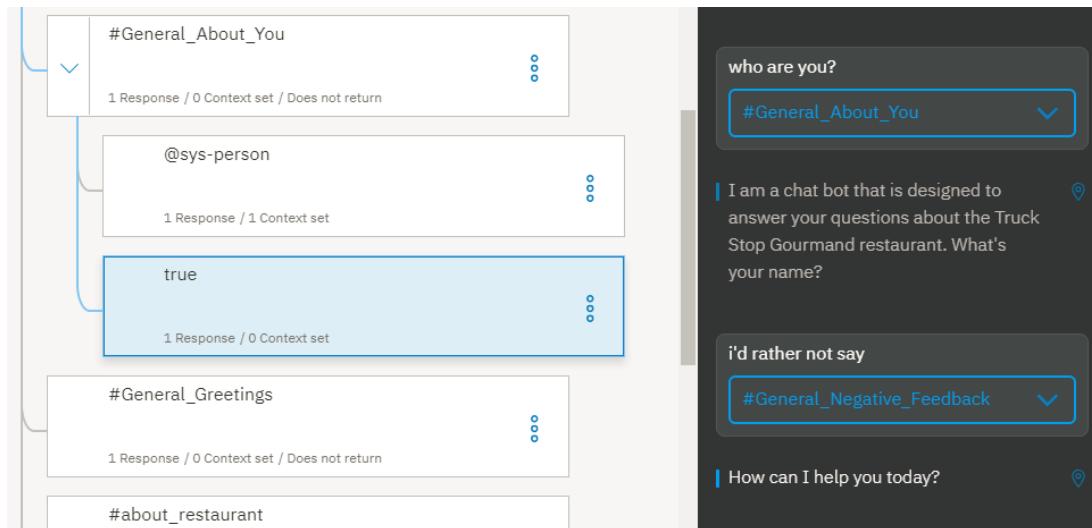
13. Click the **More**  icon on the @sys-person node, and then select **Add node below**.

You will add a node to capture user responses that do not include a name. If the user chooses not to share it, you want the bot to keep the conversation going anyhow.

14. Type **true** into the **Enter a condition** field of this node.

15. Add the following message in the response text field:

How can I help you today?



The screenshot shows the Watson Assistant interface. On the left, the node tree includes the newly added 'true' node (1 Response / 0 Context set) positioned between '#General_About_You' and '#General_Greetings'. On the right, the bot's response screen shows a message from the bot: 'I am a chat bot that is designed to answer your questions about the Truck Stop Gourmand restaurant. What's your name?'. Below it, another message says 'i'd rather not say'.

16. Click  to close the edit view.

If, at run time, the user triggers this node and provides a name, then you will know the user's name. If you know it, you should use it! Add conditional responses to the greeting dialog node you added previously to include a conditional response that uses the user name, if it is known.

Add the user name to the greeting

If you know the user's name, you should include it in your greeting message. To do so, add conditional responses, and include a variation of the greeting that includes the user's name.

1. Find the `#General_Greetings` node in the dialog tree, and click to open it in the edit view.
2. Click **Customize**, and then switch the *Multiple responses* toggle to **on**.

Customize "#General_Greetings"

Customize node **Digressions**

Slots ⓘ

Enable this to gather the information your virtual assistant needs to respond to a user within a single node.

Prompt for everything

Enable this to ask for multiple pieces of information in a single prompt, so your user can provide them all at once and not be prompted for them one at a time.

Multiple responses ⓘ

Enable multiple responses so that your virtual assistant can provide different responses to the same input, based on other conditions.

Cancel **Apply**

3. Click **Apply**.

4. Click **Add response**.

5. In the *If bot recognizes* field, type **\$username**, and in the *Respond with* field, type:

Good day to you, \$username!

6. Click the up arrow for response number 2 to move it so it is listed before response number 1 (**Good day to you!**).

If bot recognizes	Respond with
1 \$username	Good day to you, \$username!
2 Enter an intent, entity or context va	Good day to you!

7. Click **X** to close the edit view.

Test personalization

Test whether the service can recognize and save a user's name, and then refer to the user by it later.

1. Click the **Try it** icon to open the "Try it out" pane.

2. Click **Clear** to restart the conversation session.

3. Enter, Who are you?

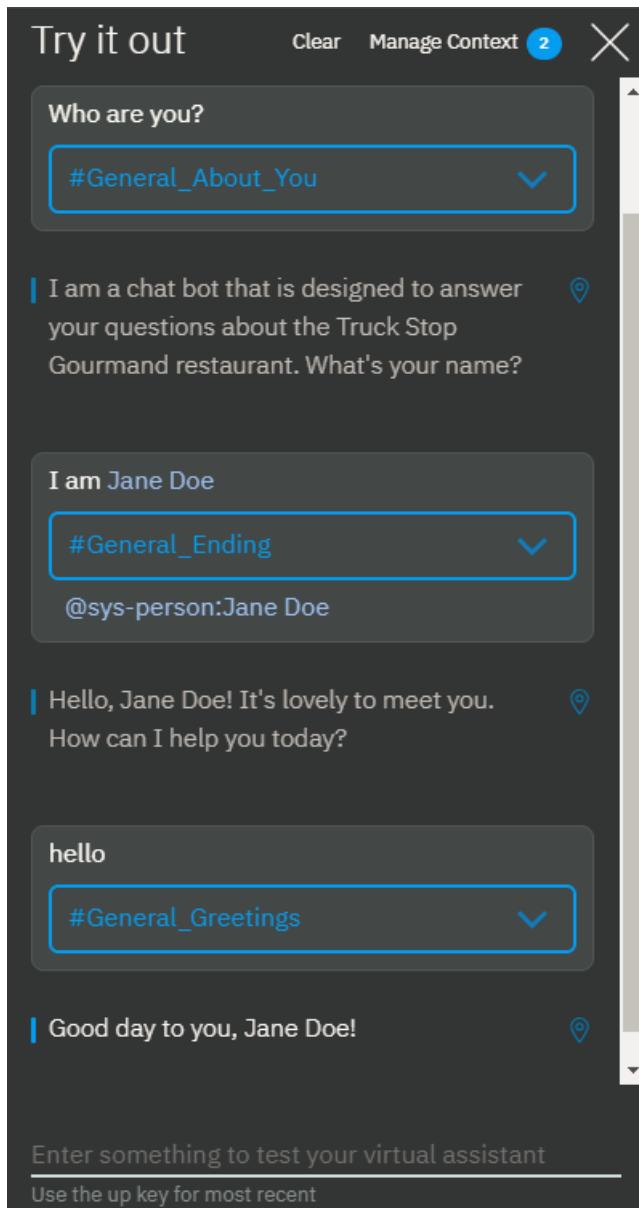
The service recognizes the `#General_About_You` intent. Its response ends with the question, What's your name?

4. Enter, I am Jane Doe.

The service recognizes Jane Doe as an `@sys-person` entity mention. It comments on your name, and then asks how it can help you.

5. Enter, Hello.

The service recognizes the `#General_Greetings` intent and says, Good day to you, Jane Doe! It uses the conditional response that includes the user's name because the `$username` context variable contains a value at the time that the greeting node is triggered.



You can add a conditional response that conditions on and includes the user's name for any other responses where personalization would add value to the conversation.

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.

Build your own front-end application

You can use the Watson SDKs to [build your own](#) front-end application that connects to your workspace using the Watson Assistant REST API.

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](#). 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**:

`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.

3. In the **Add user examples** field, type the following utterance, and then click **Add example**:

`i'd like to make a reservation`

4. Add these additional examples to help Watson recognize the #reservation intent.

I want to reserve a table for dinner
Can 3 of us get a table for lunch?
do you have openings for next Wednesday at 7?
Is there availability for 4 on Tuesday night?
i'd like to come in for brunch tomorrow
can i reserve a table?

5. Click the **Close** 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](#) 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](#) 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  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**.

Customize "#reservation"

The screenshot shows the configuration interface for the "#reservation" node. It includes sections for "Slots" (on), "Prompt for everything" (checkbox), "Multiple responses" (off), and buttons for "Cancel" and "Apply".

Slots *i* **on**

Enable this to gather the information your bot needs to respond to a user within a single node.

Prompt for everything
Enable this to ask for multiple pieces of information in a single prompt, so your user can provide them all at once and not be prompted for them one at a time.

Multiple responses *i* **off**

Enable multiple responses so that your bot can provide different responses to the same input, based on other conditions.

Cancel **Apply**

5. Define the following slots:

Slot details

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?

6. As the response, specify OK. I am making you a reservation for \$guests on \$date at \$time.

Check for	Save it as	If not present, ask	Type		
1 @sys-date	\$date	What day would you	Required		
2 @sys-time	\$time	What time do you w:	Required		
3 @sys-number	\$guests	How many people w	Required		

[⊕ Add slot](#)

Then respond with:



1. OK. I am making you a reservation for \$guests on \$date at \$time.



7. Click to close the node edit view.

Step 4: Test the dialog



1. Select the 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?`.

3. Type `Friday`.

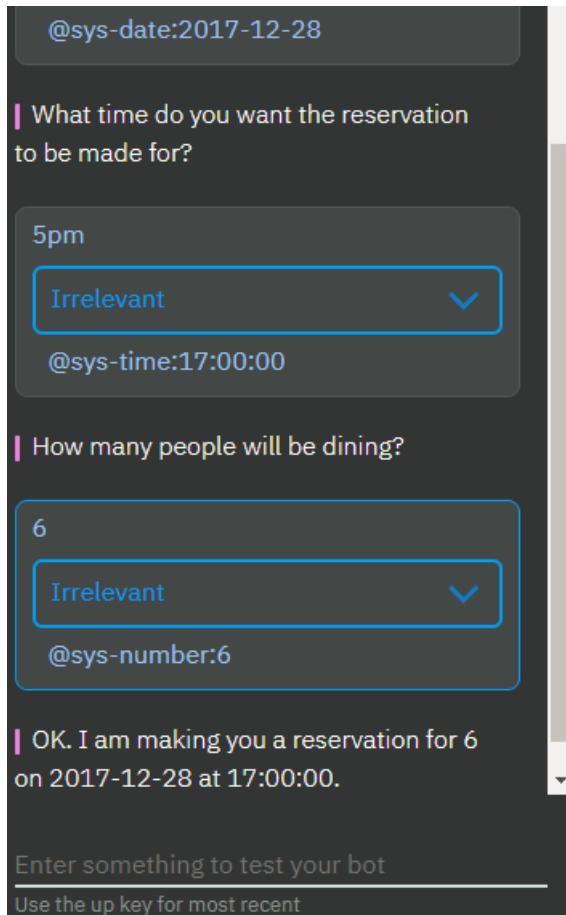
The bot recognizes the value, and uses it to fill the `$date` context variable for the first slot. It then shows the prompt for the next slot, `What time do you want the reservation to be made for?`

4. 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?`

5. 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](#). 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](#). 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.

- To reformat the \$date context variable value, click the **Edit response**  icon for the @sys-date slot.
- From the **More**  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:

```
{
  "context": {
    "date": "<? @sys-date.reformatDateTime('EEEE, MMMM d') ?>"
  }
}
```

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.

- Click **Save**.
- 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**  icon for the @sys-time slot.
- From the **More**  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:

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

- Click **Save**.
- 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:

	Script details
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

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 uses 5:00 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](#).

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 initial prompt, and then click **Apply**.

Customize "#reservation"

Slots 

 on

Enable this to gather the information your bot needs to respond to a user within a single node.

Prompt for everything

Enable this to ask for multiple pieces of information in a single prompt, so your user can provide them all at once and not be prompted for them one at a time.

Multiple responses 

 off

Enable multiple responses so that your bot can provide different responses to the same input, based on other conditions.

[Cancel](#)

[Apply](#)

3. 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.

4. Click  to close the node edit view.

5. 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.

6. 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.

7. 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: Treat zeros properly

When you use the `sys-number` system entity in a slot condition, it does not deal with zeros properly. Instead of setting the context variable that you define for the slot to 0, the service sets the context variable to false. As a result, the slot does not think it is full and prompts the user for a number again and again until the user specifies a number other than zero.

1. Test the node so you can better understand the problem. 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. Use the following script:

	Script details
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	We want to dine May 23 at 8pm. There will be 0 guests.
Watson	How many people will be dining?

You 0

Watson How many people will be dining?

You will be stuck in this loop until you specify a number other than 0.

2. To ensure that the slot treats zeros properly, change the slot condition from `@sys-number` to `@sys-number || @sys-number:0`.
3. Click the **Edit response**  icon for the slot.
4. When the context variable is created, it automatically uses the same expression that is specified for the slot condition. However, the context variable must save a number only. Edit the value that was saved as the context variable to remove the OR operator from it. From the **More**  menu at the top of the page, select **Open JSON editor**, and then edit the JSON that defines the context variable. Change the variable from `"guests": "@sys-number || @sys-number:0"` to use the following syntax:

```
{  
  "context": {  
    "guests": "@sys-number"  
  }  
}
```

5. Click **Save**.

6. 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:

Script details	
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	We want to dine May 23 at 8pm. There will be 0 guests.

This time Watson responds with, OK. I am making you a reservation for 0 on Wednesday, May 23 at 8:00 PM.

You have successfully formatted the number slot so that it can recognize zeros properly. Of course, you might not want the node to accept zero as a valid number of guests. You will learn how to validate values that are specified by users in the next step.

Step 4: 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.
- Ensure that the number of guests provided is larger than zero.

- 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**  icon for the @sys-date slot.
2. From the **Options**  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**  icon.
4. Add the following condition and response to check whether the date that the user specifies falls before today:

Slot 1 conditional response 1 details

Condition	Response	Action
-----------	----------	--------

@sys-date.before(now()) You cannot make a reservation for a day in the past. Clear slot and prompt again		
--	--	--

5. 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.

Slot 1 conditional response 2
details

Condition	Response	Action
-----------	----------	--------

true \$date it is Move on		
---------------------------	--	--

6. From the edit view of the node with slots, click the **Edit slot**  icon for the @sys-time slot.

7. From the **Options**  menu in the *Configure slot 2* header, select **Enable conditional responses**.

8. In the **Found** section, add a conditional response by clicking the **Edit response**  icon.

9. Add the following conditions and responses to check whether the time that the user specifies falls within the allowed time window:

Slot 2 conditional response details

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		
--	--	--

10. 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.

Slot 2 conditional response 3 details

Condition	Response	Action
-----------	----------	--------

true Ok, the reservation is for \$time. Move on		
---	--	--

11. Edit the @sys-number slot to validate the value provided by the user in the following ways:

- Check that the number of guests specified is larger than zero.

- 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.

12. From the edit view of the node with slots, click the **Edit slot**  icon for the @sys-number slot.
13. From the **Options**  menu in the *Configure slot 3* header, select **Enable conditional responses**.
14. In the **Found** section, add a conditional response by clicking the  icon, and then add the following condition and response:

Slot 3 conditional response details		
Condition	Response	Action
entities['sys-number'].value == 0	Please specify a number that is larger than 0.	Clear slot and prompt again
(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

Step 5: 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

```
Yes
Sure
I'd like that
Please do
Yes please.
Ok
That sounds good.
```

The screenshot shows the Watson Assistant interface. On the left is a dark sidebar with icons for navigation, settings, and other functions. The main area has a light gray background. At the top, there's a back arrow and the text '#yes'. Below this is a section titled 'Add user examples' with a sub-section 'Add user examples to this intent'. A blue button labeled 'Add example' is present. A list of user examples follows, each with a checkbox:

- User examples ▾
- I'd like that
- Ok
- Please do
- Sure
- That sounds good.
- Yes
- Yes please.

4. #no

No

No thanks.
Please don't.
Please do not!
That's not what I want at all
Absolutely not.
No way

5. 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:

Confirmation slot details		
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?	

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.

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.

6. Click the **Edit slot** icon. From the **Options** menu in the *Configure slot 4* header, select **Enable conditional responses**.

7. 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**  icon for the conditional response you just added. From the **Options**  menu, click **Open JSON editor**. Add a context block that sets the slot context variables to null, as shown below.

```
{  
  "output": {  
    "text": {  
      "values": [  
        "Alright. Let's start over. I'll try to keep up this time."  
      ]  
    }  
  },  
  "context": {  
    "date": null,  
    "time": null,  
    "guests": null  
  }  
}
```

8. Click **Back**, and then click **Save**.

9. Click the **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.

Not found response details

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.

10. Click **Save**.

11. 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.

12. 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:

Slot 1 conditional response 2 details

Condition	Response	Action
!(\$time && \$guests)	\$date	it is Move on

13. 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:

Slot 2 conditional response 3 details

Condition	Response	Action
!(\$date && \$guests)	Ok, the reservation is for \$time.	Move on

14. 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:

Slot 3 conditional response 2 details

Condition	Response	Action
!(\$date && \$time)	Ok. The reservation is for \$guests guests. Move on	

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 6: 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** 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** 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.

```
{  
  "context": {  
    "date": null,  
    "time": null,  
    "guests": null,  
    "confirmation": null  
  },  
  "output": {}  
}
```

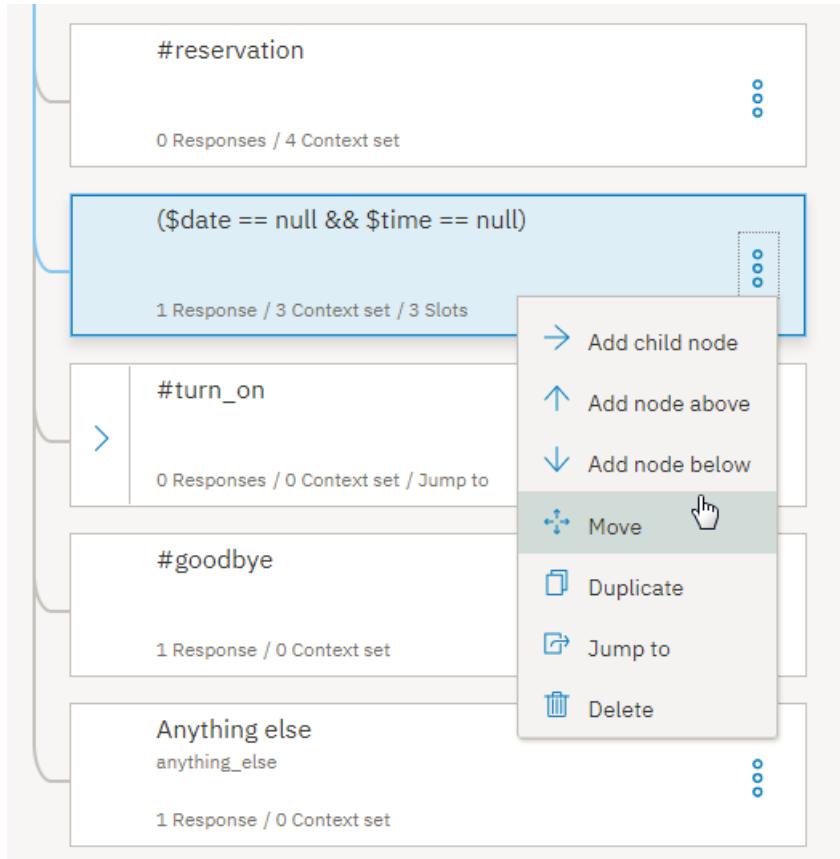
If bot recognizes:
`#reservation`

Then respond with:

```
1 {  
2   "context": {  
3     "date": null,  
4     "time": null,  
5     "guests": null,  
6     "confirmation": null  
7   },  
8   "output": {}  
9 }
```

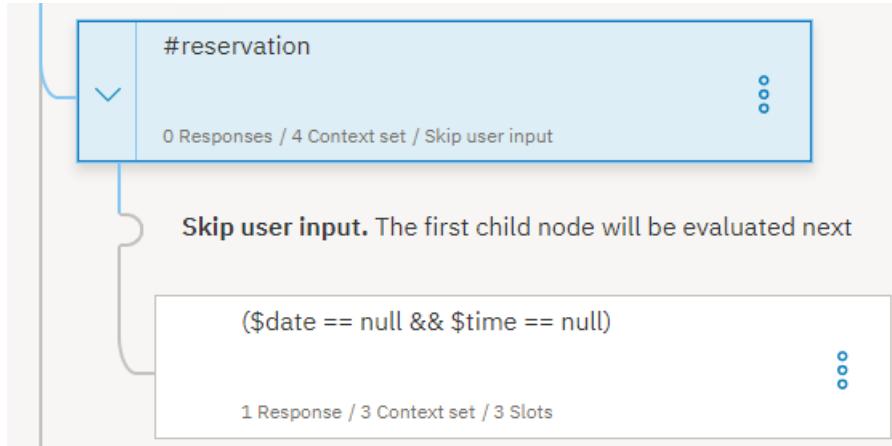
4. Click to edit the other `#reservation` node, the one you created previously and to which you added the slots.
5. Change the node condition from `#reservation` to `($date == null && $time == null)`, and then close the node edit view by clicking .

6. Click the More  icon on the node with slots, and then select **Move**.



7. Select the #reservation node as the move-to location target, and then choose **As child node** from the menu.

8. 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 7: 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.

I want to stop
Exit!
Cancel this process
I changed my mind. I don't want to make a reservation.
Stop the reservation
Wait, cancel this.
Nevermind.

The screenshot shows the Watson Assistant interface with the 'Intents' tab selected. On the left is a sidebar with icons for Home, Intents, Dialog, and Metrics. The main area shows the '#exit' intent. At the top, there are buttons for 'Add user examples' and 'Add user examples to this intent'. A large blue button labeled 'Add example' is prominent. Below this, a section titled 'User examples ▾' lists nine examples, each with a checkbox:

- Cancel this process
- Exit!
- I changed my mind. I don't want to make a reservation.
- I want to stop
- Nevermind.
- Stop the reservation
- Wait, cancel this.

3. Return to the dialog by clicking the **Dialog** tab. Click to open the node with slots, and then click **Manage handlers**.

If bot recognizes:

`($date == null && $time == null)`  

Then check for:



Check for	Save it as	If not present, ask	Type		
1 @sys-date	\$date	What day would you	Required		
2 @sys-time	\$time	What time do you w	Required		
3 @sys-number	\$guests	How many people w	Required		
4 (#yes #no) && slot	\$confirmation	I'm going to reserve	Required		

4. Add the following values to the fields.

Node-level handler details

Condition	Response	Action
#exit	Ok, we'll stop there. No reservation will be made. Skip to response	

The **Skip to response** action jumps directly to the node-level response without displaying the prompts associated with any of the remaining unfilled slots.

5. Click **Back**, and then click **Save**.

6. 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**.

Customize "(\$date == null && \$time == null)"

Slots 

 on

Enable this to gather the information your bot needs to respond to a user within a single node.

Prompt for everything

Enable this to ask for multiple pieces of information in a single prompt, so your user can provide them all at once and not be prompted for them one at a time.

Multiple responses 

 on

Enable multiple responses so that your bot can provide different responses to the same input, based on other conditions.

[Cancel](#)

[Apply](#)

7. Scroll down to the response section for the node with slots, and then click **Add response**.

8. Add the following values to the fields.

Node-level conditional response details

Condition

Response

has_skipped_slots I look forward to helping you with your next reservation. Have a good day.

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.

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](#) for an alternate way to do so.

9. 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**.

10. Test this change by using the following script in the "Try it out" pane.

Script details

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.

Step 8: 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  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.

```
{  
  "context": {  
    "date": null,  
    "time": null,  
    "guests": null,  
    "confirmation": null,  
    "counter": 0  
  },  
  "output": {}  
}
```

3. From the tree view, expand the #reservation node, and then click to edit the node with slots.

4. Click the Edit slot  icon for the @sys-time slot.

5. From the Options  menu in the Configure slot 2 header, select Enable conditional responses.

6. In the Not found section, add a conditional response.

Not found response details

Condition	Response
-----------	----------

true Please specify the time that you want to eat. The restaurant seats people between 9AM and 9PM.

7. 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  icon.

8. Click the **Options**  icon, and select **Open JSON editor**. Add the following context variable definition.

```
{  
  "output": {  
    "text": {  
      "values": [  
        "Please specify the time that you want to eat.  
        The restaurant seats people between 9AM and 9PM."  
      ]  
    }  
  },  
  "context": {  
    "counter": "<? context['counter'] + 1 ?>"  
  }  
}
```

This expression adds a 1 to the current counter tally.

9. Click **Back**, and then click **Save**.

10. Reopen the @sys-time slot by clicking the **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**.

11. Add the following condition and response.

Not found response details

Condition	Response
-----------	----------

\$counter > 1 You seem to be having trouble choosing a time. I will make the reservation at 8PM for you.

You must set the \$time variable to 8PM, so click the **Edit response**  icon. Select **Open JSON editor**, add the following context variable definition, and then click **Back**.

```
{  
  "output": {  
    "text": {  
      "values": [  
        "You seem to be having trouble choosing a time.  
        I will make the reservation at 8 PM for you."  
      ]  
    }  
  },  
  "context": {  
    "time": "<? '20:00:00'.reformatDateTime('h:mm a') ?>"  
  }  
}
```

12. 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**.

13. 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 9: 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.

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](#).

Next steps

Deploy your workspace by connecting it to a user interface.

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](#) 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](#) file.
2. In your Watson Assistant instance, click the  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

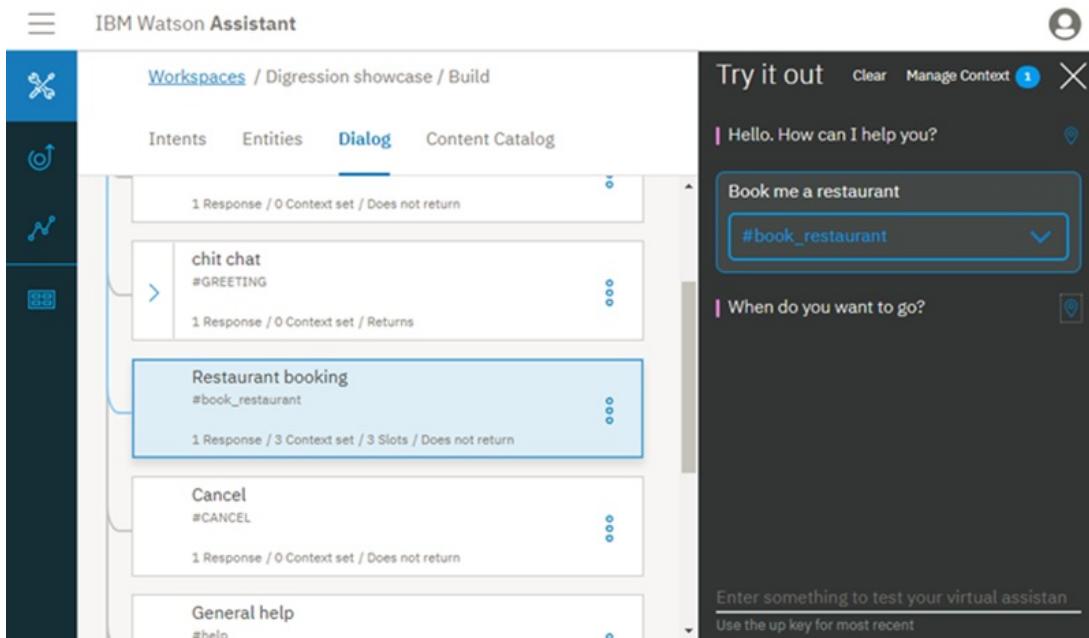
Digressions allow for the user to break away from a dialog branch in order to temporarily change the topic before returning to the original dialog flow. In this step, you will start to book a restaurant reservation, then digress away to ask for the restaurant's hours. After providing the opening hours 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  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?`

4. Click the **Location**  icon next to the response to highlight the node that triggered the response, the **Restaurant booking** node, in the dialog tree.



The screenshot shows the IBM Watson Assistant interface. On the left, the 'Dialog' tab is selected in the navigation bar, displaying a tree structure of dialog nodes. The 'Restaurant booking' node is highlighted with a blue border. In the center, the 'Try it out' pane shows a conversation history: the user types 'Book me a restaurant' and the system responds with '#book_restaurant'. Below this, the user types 'When do you want to go?' and the system responds with a location pin icon. At the bottom of the pane, there is a text input field with placeholder text: 'Enter something to test your virtual assistant' and 'Use the up key for most recent'.

5. Type Tomorrow.

The service responds with a prompt for the time to reserve, **What time do you want to go?**

6. 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.

The screenshot shows the IBM Watson Assistant interface. On the left, under the 'Dialog' tab, there is a list of nodes:

- Restaurant booking** (#book_restaurant): 1 Response / 3 Context set / 3 Slots / Does not return
- Cancel** (#CANCEL): 1 Response / 0 Context set / Does not return
- General help** (#help): 1 Response / 0 Context set / Returns
- Restaurant opening hours** (#restaurant_opening_hours): 1 Response / 0 Context set / Returns

On the right, the 'Try it out' panel shows a conversation:

- What time do you want to go?
- What time do you close?
- The restaurant is open from 8:00 AM to 10:00 PM.
- What time do you want to go?
- 8pm

The 'Irrelevant' slot value is shown as '@sys-time:20:00:00'. There is also a note at the bottom: 'Enter something to test your virtual assistant' and 'Use the up key for most recent'.

7. 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 setting 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.

Customize "Restaurant booking"

Customize node

Digressions

▼ Digressions can go away from this node ⓘ



Allow digressions away while slot filling

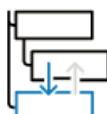
on

Users can divert the conversation away from this node in the middle of processing slots.

Only digress from slots to nodes that allow returns

If a user goes off topic, only nodes with digressions that allow returns will be considered.

▼ Digressions can come into this node ⓘ



Allow digressions into this node

on

Users can digress to this node from other dialog flows.

Return after digression

After this dialog flow is processed, return to the dialog flow that was previously in progress.

Cancel

Apply

3. Change the **Allow digressions away** toggle from **on** to **off**, and then click **Apply**.

4. Click  to close the node edit view.

5. Click **Clear** in the "Try it out" pane to reset the dialog.

6. Type **Book me a restaurant**.

The service responds with a prompt for the day to reserve, **When do you want to go?**

7. Type **Tomorrow**.

The service responds with a prompt for the time to reserve, **What time do you want to go?**

8. 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.

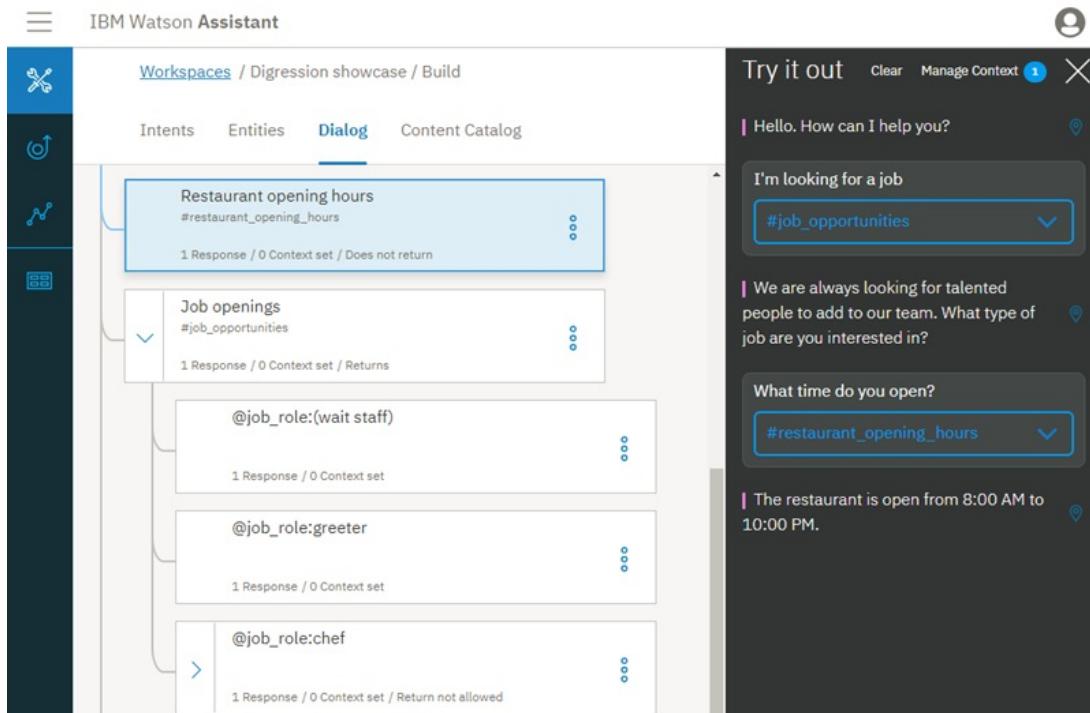
1. Click to open the **Restaurant opening hours** node.
2. Click **Customize**, and then click the **Digressions** tab.
3. Expand the **Digressions can come into this node** section, and deselect the **Return after digression** checkbox. Click **Apply**, and then click  to close the node edit view.
4. 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?

6. 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](#).

Tutorial: Building a cognitive car dashboard 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](#).

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.

If you do not have the workspace, you can add it to your instance by importing the [watson_assistant_tutorial.json](#) file.

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**:

`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.

3. In the **Add user example** field, type the following utterance, and then click **Add example**:

`I need lights`

4. Add these 5 more examples to help Watson recognize the `#turn_on` intent.

`Play some tunes`

`Turn on the radio`

```
turn on
Air on please
Crank up the AC
Turn on the headlights
```

5. Click the **Close**  icon to finish adding the #turn_on intent.

You now have four intents: the #turn_on intent that you just added, and the #General_Greetings, #General_Endings, and #Customer_Care_Contact_Us 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:

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.

4. Add the following value to the **Value name** field:

radio

The value represents a specific appliance that users might want to turn on.

5. 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.

music
tunes

6. Click **Add value** to finish defining the radio value for the @appliance entity.

7. Add other types of appliances.

Other types of @appliances

Value	Synonyms
-------	----------

headlights	lights
------------	--------

air conditioning	air and AC
------------------	------------

8. 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.

9. Click the **Close**  icon to finish adding the @appliance entity.

10. Repeat Steps 2-8 to create the @genre entity with fuzzy matching on, and these values and synonyms:

Other types of @genre	
Value	Synonyms
classical	symphonic
rhythm and blues r&b	
rock	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  on the #General_Greetings 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  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

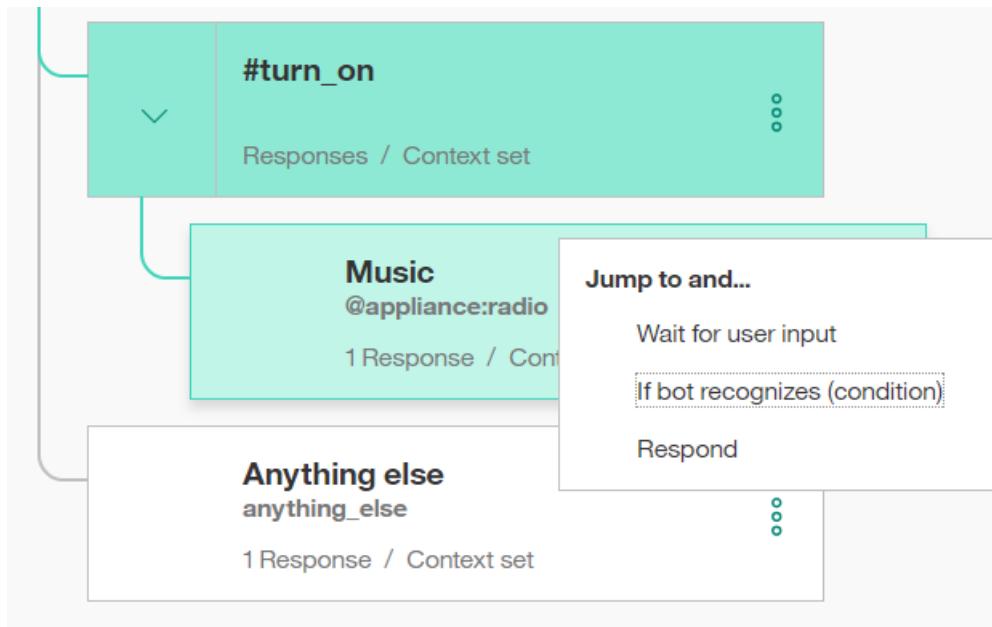
1. Click the More icon  on the #turn_on node, and select **Add child node**.
2. In the condition field, enter @appliance:radio. This condition is true if the value of the @appliance entity is

- 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  to close the node edit view.

Add a jump from the #turn_on node to the Music node

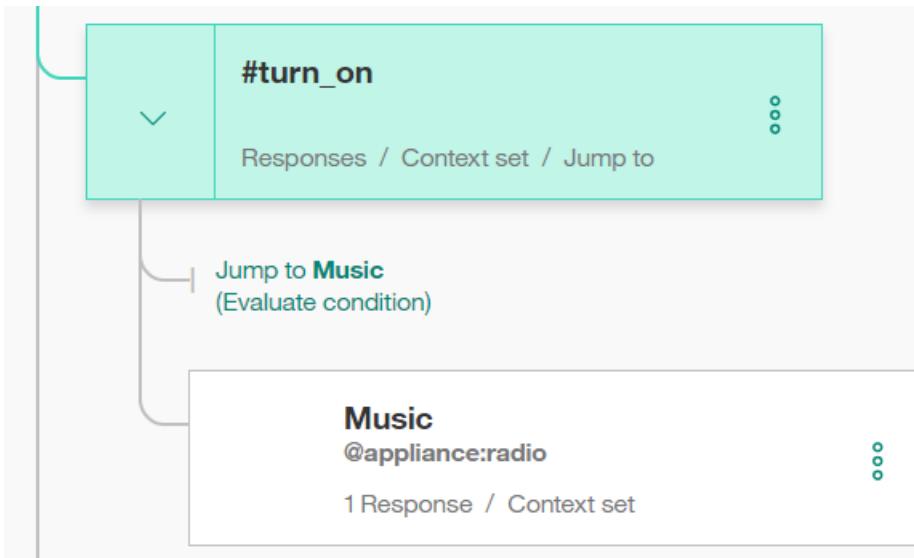
Jump directly from the #turn_on node to the Music node without asking for any more user input. To do this, you can use a **Jump to** action.

1. Click the More icon  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  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`.

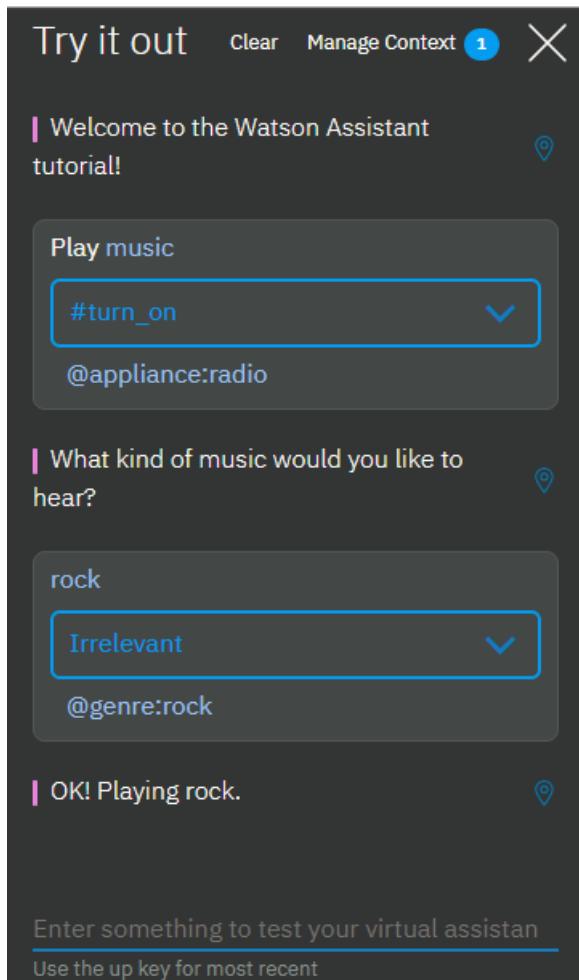
1. Click the More icon  on the `@genre` node, and select **Add node below** to create a peer node.
2. Enter `true` in the condition field. The true condition is a special condition. It specifies that if the dialog flow reaches this node, it should always evaluate as true. (If the user specifies a valid `@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  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.



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

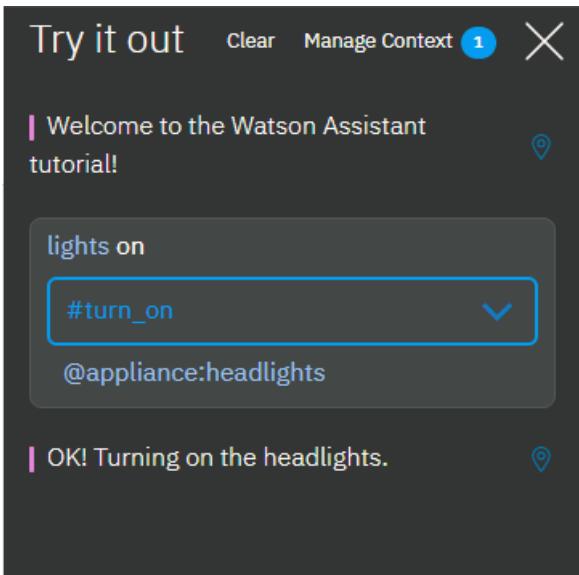
Add a node that is triggered when any other valid value for @appliance is provided by the user. For the other values of @appliance, the bot doesn't need to ask for any more input. It just returns a positive response.

1. Click the More icon  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  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.



3. Type turn on the ac.

The bot recognizes the #turn_on intent and the @appliance:(air conditioning) entity, and it responds with OK, turning on the air conditioning.

4. Try variations on all of the supported commands based on the example utterances and entity synonyms you defined.

Address Scenario 3

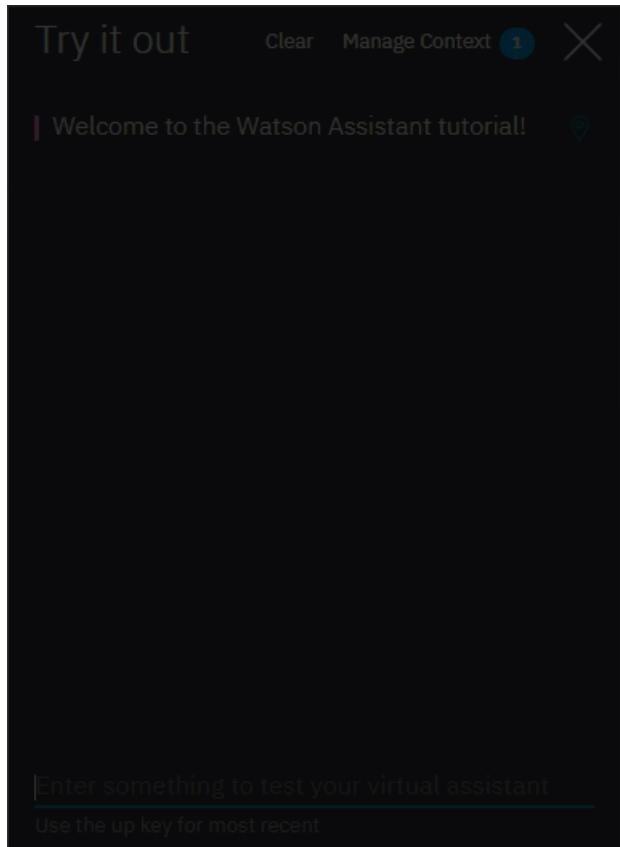
Now add a peer node that is triggered if the user does not specify a valid appliance type.

1. Click the More icon  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 import the complete [Car Dashboard Tutorial](#) workspace as a JSON file to see this same use case fleshed out even more with a longer dialog and additional functionality.

Next steps

Interact with a demo application that was built by deploying a similar car dashboard workspace as a web application. See the [Car Dashboard Demo](#).

Expression language methods

You can process values extracted from user utterances that you want to reference in a context variable, condition, or elsewhere in the response.

Evaluation syntax

To expand variable values inside other variables, or apply methods to output text or context variables, use the <? expression ?> expression syntax. For example:

- **Incrementing a numeric property**
 - "output": {"number": "<? output.number + 1 ?>"}
- **Invoking a method on an object**
 - "context": {"toppings": "<? context.toppings.append('onions') ?>"}

The following sections describe methods you can use to process values. They are organized by data type:

- [Arrays](#)
- [Date and Time](#)
- [Numbers](#)
- [Objects](#)
- [Strings](#)

Arrays

You cannot use these methods to check for a value in an array in a node condition or response condition within the same node in which you set the array values.

JSONArray.append(object)

This method appends a new value to the JSONArray and returns the modified JSONArray.

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"]  
  }  
}
```

JSONArray.clear()

This method clears all values from the array and returns null.

Use the following expression in the output to define a field that clears an array that you saved to a context variable (\$toppings_array) of its values.

```
{  
  "output": {  
    "array_eraser": "<? $toppings_array.clear() ?>"  
  }  
}
```

If you subsequently reference the \$toppings_array context variable, it returns '[]' only.

JSONArray.contains(Object value)

This method returns true if the input JSONArray contains the input value.

For this Dialog runtime context which is set by a previous node or external application:

```
{  
  "context": {  
    "toppings_array": ["onion", "olives", "ham"]  
  }  
}
```

Dialog node or response condition:

```
$toppings_array.contains('ham')
```

Result: True because the array contains the element ham.

JSONArray.containsIntent(String intent_name, Double min_score, [Integer top_n])

This method returns true if the intents JSONArray specifically contains the specified intent, and that intent has a confidence score that is equal to or higher than the specified minimum score. Optionally, you can specify a number to indicate that the intent must be included within that number of top elements in the array.

Returns false if the specified intent is not in the array, does not have a confidence score that is equal to or higher than the minimum confidence score, or the intent is lower in the array than the specified index location.

The service automatically generates an intents array that lists the intents that the service detects in the input whenever user input is submitted. The array lists all intents that are detected by the service in order of highest confidence first.

You can use this method in a node condition to not only check for the presence of an intent, but to set a confidence score threshold that must be met before the node can be processed and its response returned.

For example, use the following expression in a node condition when you want to trigger the dialog node only when the following conditions are met:

- The #General_Ending intent is present.
- The confidence score of the #General_Ending intent is over 80%.
- The #General_Ending intent is one of the top 2 intents in the intents array.

```
intents.containsIntent("General_Ending", 0.8, 2)
```

JSONArray.filter(temp, "temp.property operator comparison_value")

Filters an array by comparing each array element value to a value you specify. This method is similar to a [collection projection](#). A collection projection returns a filtered array based on a name in an array element name-value pair. The filter method returns a filtered array based on a value in an array element name-value pair.

The filter expression consists of the following values:

- **temp**: Name of a variable that is used temporarily as each array element is evaluated. For example, `city`.
- **property**: Element property that you want to compare to the `comparison_value`. Specify the property as a property of the temporary variable that you name in the first parameter. Use the syntax: `temp.property`. For example, if `latitude` is a valid element name for a name-value pair in the array, specify the property as `city.latitude`.
- **operator**: Operator to use to compare the property value to the `comparison_value`.

Supported operators are:

Supported filter operators

Operator	Description
-----------------	--------------------

<code>==</code>	Is equal to
<code>></code>	Is greater than
<code><</code>	Is less than
<code>>=</code>	Is greater than or equal to
<code><=</code>	Is less than or equal to
<code>!=</code>	Is not equal to

- **comparison_value**: Value that you want to compare each array element property value against. To specify a value that can change depending on the user input, use a context variable or entity as the value. If you specify a value that can vary, add logic to guarantee that the `comparison_value` value is valid at evaluation time or an error will occur.

Filter example 1

For example, you can use the filter method to evaluate an array that contains a set of city names and their population numbers to return a smaller array that contains only cities with a population over 5 million.

The following `$cities` context variable contains an array of objects. Each object contains a `name` and `population` property.

```
[  
  {  
    "name": "Tokyo",  
    "population": 9273000  
  },  
  {  
    "name": "Rome",  
    "population": 2868104  
  },  
  {  
    "name": "Beijing",  
    "population": 20693000
```

```

},
{
  "name": "Paris",
  "population": 2241346
}
]

```

In the following example, the arbitrary temporary variable name is `city`. The SpEL expression filters the `$cities` array to include only cities with a population of over 5 million:

```
$cities.filter("city", "city.population > 5000000")
```

The expression returns the following filtered array:

```

[
  {
    "name": "Tokyo",
    "population": 9273000
  },
  {
    "name": "Beijing",
    "population": 20693000
  }
]
```

You can use a collection projection to create a new array that includes only the city names from the array returned by the filter method. You can then use the `join` method to display the two name element values from the array as a String, and separate the values with a comma and a space.

```
The cities with more than 5 million people include <? T(String).join(", ", ($cities.filter("city", "city.population > 5000000")).![name]) ?>.
```

The resulting response is: The cities with more than 5 million people include Tokyo, Beijing.

Filter example 2

The power of the filter method is that you do not need to hard code the `comparison_value` value. In this example, the hard coded value of 5000000 is replaced with a context variable instead.

In this example, the `$population_min` context variable contains the number 5000000. The arbitrary temporary variable name is `city`. The SpEL expression filters the `$cities` array to include only cities with a population of over 5 million:

```
$cities.filter("city", "city.population > $population_min")
```

The expression returns the following filtered array:

```

[
  {
    "name": "Tokyo",
    "population": 9273000
  },
  {
    "name": "Beijing",
    "population": 20693000
  }
]
```

When comparing number values, be sure to set the context variable involved in the comparison to a valid value before the

filter method is triggered. Note that null can be a valid value if the array element you are comparing it against might contain it. For example, if the population name and value pair for Tokyo is "population":null, and the comparison expression is "city.population == \$population_min", then null would be a valid value for the \$population_min context variable.

You can use a dialog node response expression like this:

```
The cities with more than $population_min people include <? T(String).join(", ",  
($cities.filter("city", "city.population > $population_min")).![name]) ?>.
```

The resulting response is: The cities with more than 5000000 people include Tokyo, Beijing.

Filter example 3

In this example, an entity name is used as the comparison_value. The user input is, What is the population of Tokyo? The arbitrary temporary variable name is y. You created an entity named @city that recognizes city names, including Tokyo.

```
$cities.filter("y", "y.name == @city")
```

The expression returns the following array:

```
[  
  {  
    "name": "Tokyo",  
    "population": 9273000  
  }  
]
```

You can use a collection project to get an array with only the population element from the original array, and then use the get method to return the value of the population element.

```
The population of @city is: <? ($cities.filter("y", "y.name == @city").![population]).get(0) ?  
>.
```

The expression returns: The population of Tokyo is 9273000.

JSONArray.get(Integer)

This method returns the input index from the JSONArray.

For this Dialog runtime context which is set by a previous node or external application:

```
{  
  "context": {  
    "name": "John",  
    "nested": {  
      "array": [ "one", "two" ]  
    }  
  }  
}
```

Dialog node or response condition:

```
$nested.array.get(0).getAsString().contains('one')
```

Result: True because the nested array contains one as a value.

Response:

```
"output": {  
  "generic" : [  
    {  
      "values": [  
        {  
          "text" : "The first item in the array is <?$nested.array.get(0)?>"  
        }  
      ],  
      "response_type": "text",  
      "selection_policy": "sequential"  
    }  
  ]  
}
```

JSONArray.getRandomItem()

This method returns a random item from the input JSONArray.

For this Dialog runtime context:

```
{  
  "context": {  
    "toppings_array": ["onion", "olives", "ham"]  
  }  
}
```

Dialog node output:

```
{  
  "output": {  
    "generic" : [  
      {  
        "values": [  
          {  
            "text": "<? $toppings_array.getRandomItem() ?> is a great choice!"  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

Result: "ham is a great choice!" or "onion is a great choice!" or "olives is a great choice!"

Note: The resulting output text is randomly chosen.

JSONArray.indexOf(value)

This method returns the index number of the element in the array that matches the value you specify as a parameter or -1 if the value is not found in the array. The value can be a String ("School"), Integer(8), or Double (9.1). The value must be an exact match and is case sensitive.

For example, the following context variables contain arrays:

```
{
  "context": {
    "array1": ["Mary", "Lamb", "School"],
    "array2": [8, 9, 10],
    "array3": [8.1, 9.1, 10.1]
  }
}
```

The following expressions can be used to determine the array index at which the value is specified:

```
<? $array1.indexOf("Mary") ?> returns '0'
<? $array2.indexOf(9) ?> returns '1'
<? $array3.indexOf(10.1) ?> returns '2'
```

This method can be useful for getting the index of an element in an intents array, for example. You can apply the `indexOf` method to the array of intents generated each time user input is evaluated to determine the array index number of a specific intent.

```
intents.indexOf("General_Greetings")
```

If you want to know the confidence score for a specific intent, you can pass the expression above in as the `index` value to an expression with the syntax `intents[index].confidence`. For example:

```
intents[intents.indexOf("General_Greetings")].confidence
```

JSONArray.join(String delimiter)

This method joins all values in this array to a string. Values are converted to string and delimited by the input delimiter.

For this Dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives", "ham"]
  }
}
```

Dialog node output:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "This is the array: <? $toppings_array.join(';)') ?>"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result:

This is the array: onion;olives;ham;

If you define a variable that stores multiple values in a JSON array, you can return a subset of values from the array, and then use the `join()` method to format them properly.

Collection projection

A collection projection SpEL expression extracts a subcollection from an array that contains objects. The syntax for a collection projection is `array_that_contains_value_sets.! [value_of_interest]`.

For example, the following context variable defines a JSON array that stores flight information. There are two data points per flight, the time and flight code.

```
"flights_found": [
  {
    "time": "10:00",
    "flight_code": "OK123"
  },
  {
    "time": "12:30",
    "flight_code": "LH421"
  },
  {
    "time": "16:15",
    "flight_code": "TS4156"
  }
]
```

To return the flight codes only, you can create a collection projection expression by using the following syntax:

```
<? $flights_found.! [flight_code] ?>
```

This expression returns an array of the `flight_code` values as `["OK123", "LH421", "TS4156"]`. See the [SpEL Collection projection documentation](#) for more details.

If you apply the `join()` method to the values in the returned array, the flight codes are displayed in a comma-separated list. For example, you can use the following syntax in a response:

The flights that fit your criteria are:

```
<? T(String).join(", ", $flights_found.! [flight_code]) ?>.
```

Result: The flights that match your criteria are: OK123,LH421,TS4156.

JSONArray.joinToArray(template)

This method applies the format that you define in a template to the array, and returns an array that is formatted according to your specifications. This method is useful for applying formatting to array values that you want to return in a dialog response, for example.

The template can be specified as a String, JSON Object, or JSON Array. To reference values from the array that you are editing in the template, follow these syntax conventions:

- `%`: Represents the start or end of an element or element property that you want to return from the array being edited.
- `e`: Temporarily represents the array element to which you want to apply the formatting. This temporary variable name cannot be changed from `e`.

For example, you have a context variable that contains an array with a list of flight details for three flights.

```
"flights": [
```

```
{
  "flight": "DL1040",
  "origin": "JFK",
  "carrier": "Alitalia",
  "duration": 485,
  "destination": "FCO",
  "arrival_date": "2019-02-03",
  "arrival_time": "07:00",
  "departure_date": "2019-02-02",
  "departure_time": "16:45"
},
{
  "flight": "DL1710",
  "origin": "JFK",
  "carrier": "Delta",
  "duration": 379,
  "destination": "LAX",
  "arrival_date": "2019-02-02",
  "arrival_time": "10:19",
  "departure_date": "2019-02-02",
  "departure_time": "07:00"
},
{
  "flight": "DL4379",
  "origin": "BOS",
  "carrier": "Virgin Atlantic",
  "duration": 385,
  "destination": "LHR",
  "arrival_date": "2019-02-03",
  "arrival_time": "09:05",
  "departure_date": "2019-02-02",
  "departure_time": "21:40"
}
]
```

You want to return just the list of flight codes. To extract only the value of the `flight` element from each array and return it in a list, you can use the following expression:

The available flights are <? \$flights.joinToArray("%e.flight%"). ?>

The dialog node response is The available flights are ["DL1040", "DL1710", "DL4379"].

To display the array as text, use the `join` method in the expression like this:

The available flights are <? \$flights.joinToArray("%e.flight%").join(", "). ?>

The response is, The available flights are DL1040, DL1710, DL4379.

Complex template

To create a more complex template, instead of specifying the template details in the method parameter directly, you can create a context variable.

This template context variable contains a subset of the array elements and adds labels in front of them, so the information will be displayed in a legible list in the response:

```
"template": "<br/>Flight number: %e.flight% <br/> Airline: %e.carrier% <br/> Departure date:
```

```
%e.departure_date% <br/> Departure time: %e.departure_time% <br/> Arrival time:  
%e.arrival_time% <br/>"
```

The
 HTML tag for a line break is *not* rendered by some of the integration channels, including Facebook and Slack.

Use this expression in the dialog node response to apply the template defined in \$template to the array in \$flights.

```
The flight info is <? $flights.joinToArray($template).join(" ") ?>
```

The response looks like this:

```
The flight info is  
Flight number: DL1040  
Airline: Alitalia  
Departure date: 2019-02-02  
Departure time: 16:45  
Arrival time: 07:00
```

```
Flight number: DL1710  
Airline: Delta  
Departure date: 2019-02-02  
Departure time: 07:00  
Arrival time: 10:19
```

```
Flight number: DL4379  
Airline: Virgin Atlantic  
Departure date: 2019-02-02  
Departure time: 21:40  
Arrival time: 09:05
```

The advantage of using this method is that it doesn't matter how often the values in the array change or whether the number of elements in the array increases. As long as each array element contains at least the subset of properties that are referenced by the template, then the expression works.

JSON Object template example

In this example, the template context variable is defined as a JSON object that extracts the flight number, and arrival and departure dates and times from each of the flight elements specified in the array in the \$flights context variable. You could use this approach to apply standard formatting to flight details for flights that are managed by two different carriers, and who format flight information differently in their web services, for example.

```
"template": {  
    "departure": "Flight %e.flight% departs on %e.departure_date% at %e.departure_time%.",  
    "arrival": "Flight %e.flight% arrives on %e.arrival_date% at %e.arrival_time%."  
}
```

You might want to design your custom client application to read the objects from the returned array, and format the values properly for your chat bot's response. Your dialog node response can return the flight arrival details object as an array by using this expression:

```
<? $flights.joinToArray($template) ?>
```

This is the dialog node response:

```
[  
  {  
    "arrival": "Flight DL1040 arrives on 2019-02-03 at 07:00.",  
    "departure": "Flight DL1040 departs on 2019-02-02 at 16:45."
```

```

    },
{
  "arrival": "Flight DL1710 arrives on 2019-02-02 at 10:19.",
  "departure": "Flight DL1710 departs on 2019-02-02 at 07:00."
},
{
  "arrival": "Flight DL4379 arrives on 2019-02-03 at 09:05.",
  "departure": "Flight DL4379 departs on 2019-02-02 at 21:40."
}
]

```

Notice that the order of the `arrival` and `departure` elements is swapped in the response. The service typically reorders elements in a JSON Object. If you want the elements to be returned in a specific order, define the template by using a JSON Array or String value instead.

JSONArray.remove(Integer)

This method removes the element in the index position from the JSONArray and returns the updated JSONArray.

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"]
  }
}
```

JSONArray.removeValue(object)

This method removes the first occurrence of the value from the JSONArray and returns the updated JSONArray.

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"]  
  }  
}
```

JSONArray.set(Integer index, Object value)

This method sets the input index of the JSONArray to the input value and returns the modified JSONArray.

For this Dialog runtime context:

```
{  
  "context": {  
    "toppings_array": ["onion", "olives", "ham"]  
  }  
}
```

Dialog node output:

```
{  
  "context": {  
    "toppings_array": "<? $toppings_array.set(1,'ketchup')?>"  
  }  
}
```

Result:

```
{  
  "context": {  
    "toppings_array": ["onion", "ketchup", "ham"]  
  }  
}
```

JSONArray.size()

This method returns the size of the JSONArray as an integer.

For this Dialog runtime context:

```
{  
  "context": {  
    "toppings_array": ["onion", "olives"]  
  }  
}
```

Make this update:

```
{  
  "context": {
```

```
        "toppings_array_size": "<? $toppings_array.size() ?>"  
    }  
}
```

Result:

```
{  
    "context": {  
        "toppings_array_size": 2  
    }  
}
```

JSONArray split(String regexp)

This method splits the input string by using the input regular expression. The result is a JSONArray of strings.

For this input:

```
"bananas;apples;pears"
```

This syntax:

```
{  
    "context": {  
        "array": "<?input.text.split(\";\")?>"  
    }  
}
```

Results in this output:

```
{  
    "context": {  
        "array": [ "bananas", "apples", "pears" ]  
    }  
}
```

com.google.gson.JsonArray support

In addition to the built-in methods, you can use standard methods of the `com.google.gson.JsonArray` class.

New array

```
new JSONArray().append('value')
```

To define a new array that will be filled in with values that are provided by users, you can instantiate an array. You must also add a placeholder value to the array when you instantiate it. You can use the following syntax to do so:

```
{  
    "context":{  
        "answer": "<? output.answer?:new JSONArray().append('temp_value') ?>"  
    }  
}
```

Date and Time

Several methods are available to work with date and time.

For information about how to recognize and extract date and time information from user input, see [@sys-date and @sys-time entities](#).

.after(String date or time)

Determines whether the date/time value is after the date/time argument.

.before(String date or time)

Determines whether the date/time value is before the date/time argument.

For example:

- @sys-time.before('12:00:00')
- @sys-date.before('2016-11-21')
- If comparing different items, such as time vs. date, date vs. time, and time vs. date and time, the method returns false and an exception is printed in the response JSON log output.log_messages.

For example, @sys-date.before(@sys-time).

- If comparing date and time vs. time the method ignores the date and only compares times.

now()

Returns a string with the current date and time in the format yyyy-MM-dd HH:mm:ss.

- Static function.
- The other date/time methods can be invoked on date-time values that are returned by this function and it can be passed in as their argument.
- If the context variable \$timezone is set, this function returns dates and times in the client's time zone.

Example of a dialog node with now() used in the output field:

```
{  
  "conditions": "#what_time_is_it",  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "<? now() ?>"  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

Example of now() in node's conditions (to decide if it is still morning):

```
{  
  "conditions": "now().before('12:00:00')",  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "Morning"  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

```

"generic": [
  {
    "values": [
      {
        "text": "Good morning!"
      }
    ],
    "response_type": "text",
    "selection_policy": "sequential"
  }
]
}

```

.reformatDateTime(String format)

Formats date and time strings to the format desired for user output.

Returns a formatted string according to the specified format:

- MM/dd/yyyy for 12/31/2016
- h a for 10pm

To return the day of the week:

- E for Tuesday
- u for day index (1 = Monday, ..., 7 = Sunday)

For example, this context variable definition creates a \$time variable that saves the value 17:30:00 as *5:30 PM*.

```

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

```

Format follows the Java [SimpleDateFormat](#) rules.

Note: When trying to format time only, the date is treated as 1970-01-01.

.sameMoment(String date/time)

- Determines whether the date/time value is the same as the date/time argument.

.sameOrAfter(String date/time)

- Determines whether the date/time value is after or the same as the date/time argument.
- Analogous to `.after()`.

.sameOrBefore(String date/time)

- Determines whether the date/time value is before or the same as the date/time argument.

today()

Returns a string with the current date in the format yyyy-MM-dd.

- Static function.
- The other date methods can be invoked on date values that are returned by this function and it can be passed in as their argument.
- If the context variable \$timezone is set, this function returns dates in the client's time zone. Otherwise, the GMT time zone is used.

Example of a dialog node with today() used in the output field:

```
{  
  "conditions": "#what_day_is_it",  
  "output": [  
    {  
      "generic": [  
        {  
          "values": [  
            {  
              "text": "Today's date is <? today() ?>."  
            }  
          ],  
          "response_type": "text",  
          "selection_policy": "sequential"  
        }  
      ]  
    }  
  ]  
}
```

Result: Today's date is 2018-03-09.

Date and time calculations

Use the following methods to calculate a date.

Method	Description
<date>.minusDays(n)	Returns the date of the day n number of days before the specified date.
<date>.minusMonths(n)	Returns the date of the day n number of months before the specified date.
<date>.minusYears(n)	Returns the date of the day n number of years before the specified date.
<date>.plusDays(n)	Returns the date of the day n number of days after the specified date.
<date>.plusMonths(n)	Returns the date of the day n number of months after the specified date.
<date>.plusYears(n)	Returns the date of the day n number of years after the specified date.

where <date> is specified in the format yyyy-MM-dd or yyyy-MM-dd HH:mm:ss.

To get tomorrow's date, specify the following expression:

```
{  
  "output": [  
    {  
      "generic": [  
        {  
          "values": [  
            {  
              "text": "  
                <? today().plusDays(1) ?>  
              "  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```

        "text": "Tomorrow's date is <? today().plusDays(1) ?>."
    }
],
"response_type": "text",
"selection_policy": "sequential"
}
]
}
}

```

Result if today is March 9, 2018: Tomorrow's date is 2018-03-10.

To get the date for the day a week from today, specify the following expression:

```

{
"output": {
"generic": [
{
"values": [
{
"text": "Next week's date is <? @sys-date.plusDays(7) ?>."
}
],
"response_type": "text",
"selection_policy": "sequential"
}
]
}
}

```

Result if the date captured by the @sys-date entity is today's date, March 9, 2018: Next week's date is 2018-03-16.

To get last month's date, specify the following expression:

```

{
"output": {
"generic": [
{
"values": [
{
"text": "Last month the date was <? today().minusMonths(1) ?>."
}
],
"response_type": "text",
"selection_policy": "sequential"
}
]
}
}

```

Result if today is March 9, 2018: Last month the date was 2018-02-9.

Use the following methods to calculate time.

Method	Description
<time>.minusHours(n)	Returns the time n hours before the specified time.

Method	Description
<time>.minusMinutes(n)	Returns the time n minutes before the specified time.
<time>.minusSeconds(n)	Returns the time n seconds before the specified time.
<time>.plusHours(n)	Returns the time n hours after the specified time.
<time>.plusMinutes(n)	Returns the time n minutes after the specified time.
<time>.plusSeconds(n)	Returns the time n seconds after the specified time.

where <time> is specified in the format HH:mm:ss.

To get the time an hour from now, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "One hour from now is <? now().plusHours(1) ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if it is 8 AM: One hour from now is 09:00:00.

To get the time 30 minutes ago, specify the following expression:

```
{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "A half hour before @sys-time is <? @sys-time.minusMinutes(30) ?>."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
  }
}
```

Result if the time captured by the @sys-time entity is 8 AM: A half hour before 08:00:00 is 07:30:00.

To reformat the time that is returned, you can use the following expression:

```
{
  "output": {
```

```

"generic": [
  {
    "values": [
      {
        "text": "6 hours ago was <? now().minusHours(6).reformatDateTime('h:mm a') ?>."
      }
    ],
    "response_type": "text",
    "selection_policy": "sequential"
  }
]
}

```

Result if it is 2:19 PM: 6 hours ago was 8:19 AM.

Working with time spans

To show a response based on whether today's date falls within a certain time frame, you can use a combination of time-related methods. For example, if you run a special offer during the holiday season every year, you can check whether today's date falls between November 25 and December 24 of this year. First, define the dates of interest as context variables.

In the following start and end date context variable expressions, the date is being constructed by concatenating the dynamically-derived current year value with hard-coded month and day values.

```

"context": {
  "end_date": "<? now().reformatDateTime('Y') + '-12-24' ?>",
  "start_date": "<? now().reformatDateTime('Y') + '-11-25' ?>"
}

```

In the response condition, you can indicate that you want to show the response only if the current date falls between the start and end dates that you defined as context variables.

```
now().after($start_date) && now().before($end_date)
```

java.util.Date support

In addition to the built-in methods, you can use standard methods of the `java.util.Date` class.

To get the date of the day that falls a week from today, you can use the following syntax.

```

{
  "context": {
    "week_from_today": "<? new Date(new Date().getTime() +
      (7 * (24*60*60*1000L))) ?>"
  }
}

```

This expression first gets the current date in milliseconds (since January 1, 1970, 00:00:00 GMT). It also calculates the number of milliseconds in 7 days. (The `(24*60*60*1000L)` represents one day in milliseconds.) It then adds 7 days to the current date. The result is the full date of the day that falls a week from today. For example, `Fri Jan 26 16:30:37 UTC 2018`. Note that the time is in the UTC time zone. You can always change the 7 to a variable (`$number_of_days`, for example) that you can pass in. Just be sure that its value gets set before this expression is evaluated.

If you want to be able to subsequently compare the date with another date that is generated by the service, then you must reformat the date. System entities (@sys-date) and other built-in methods (now()) convert dates to the yyyy-MM-dd

format.

```
{  
  "context": {  
    "week_from_today": "<? new Date(new Date().getTime() +  
      (7 * (24*60*60*1000L))).format('yyyy-MM-dd') ?>"  
  }  
}
```

After reformatting the date, the result is 2018-01-26. Now, you can use an expression like `@sys-date.after($week_from_today)` in a response condition to compare a date specified in user input to the date saved in the context variable.

The following expression calculates the time 3 hours from now.

```
{  
  "context": {  
    "future_time": "<? new Date(new Date().getTime() + (3 * (60*60*1000L)) -  
      (5 * (60*60*1000L))).format('h:mm a') ?>"  
  }  
}
```

The `(60*60*1000L)` value represents an hour in milliseconds. This expression adds 3 hours to the current time. It then recalculates the time from a UTC time zone to EST time zone by subtracting 5 hours from it. It also reformats the date values to include hours and minutes AM or PM.

Numbers

These methods help you get and reformat number values.

For information about system entities that can recognize and extract numbers from user input, see [@sys-number entity](#).

If you want the service to recognize specific number formats in user input, such as order number references, consider creating a pattern entity to capture it. See [Creating entities](#) for more details.

If you want to change the decimal placement for a number, to reformat a number as a currency value, for example, see the [String format\(\) method](#).

toDouble()

Converts the object or field to the Double number type. You can call this method on any object or field. If the conversion fails, `null` is returned.

toInt()

Converts the object or field to the Integer number type. You can call this method on any object or field. If the conversion fails, `null` is returned.

toLong()

Converts the object or field to the Long number type. You can call this method on any object or field. If the conversion fails, `null` is returned.

If you specify a Long number type in a SpEL expression, you must append an L to the number to identify it as such. For example, `5000000000L`. This syntax is required for any numbers that do not fit into the 32-bit Integer type. For example, numbers that are greater than 2^{31} (2,147,483,648) or lower than -2^{31} (-2,147,483,648) are considered Long number

types. Long number types have a minimum value of -2^63 and a maximum value of 2^63-1.

Java number support

java.lang.Math()

Performs basic numeric operations.

You can use the the Class methods, including these:

- max()

```
{  
  "context": {  
    "bigger_number": "<? T(Math).max($number1,$number2) ?>"  
  },  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "The bigger number is $bigger_number."  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

- min()

```
{  
  "context": {  
    "smaller_number": "<? T(Math).min($number1,$number2) ?>"  
  },  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "The smaller number is $smaller_number."  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

- pow()

```
{  
  "context": {  
    "power_of_two": "<? T(Math).pow($base.toDouble(),2.toDouble()) ?>"  
  }  
}
```

```

},
"output": [
  "generic": [
    {
      "values": [
        {
          "text": "Your number $base to the second power is $power_of_two."
        }
      ],
      "response_type": "text",
      "selection_policy": "sequential"
    }
  ]
}
}

```

See the [java.lang.Math reference documentation](#) for information about other methods.

java.util.Random()

Returns a random number. You can use one of the following syntax options:

- To return a random boolean value (true or false), use <?new Random().nextBoolean()?>.
- To return a random double number between 0 (included) and 1 (excluded), use <?new Random().nextDouble()?>
- To return a random integer between 0 (included) and a number you specify, use <?new Random().nextInt(n)?> where n is the top of the number range you want + 1. For example, if you want to return a random number between 0 and 10, specify <?new Random().nextInt(11)?>.
- To return a random integer from the full Integer value range (-2147483648 to 2147483648), use <?new Random().nextInt()?>.

For example, you might create a dialog node that is triggered by the #random_number intent. The first response condition might look like this:

```

Condition = @sys-number
{
  "context": {
    "answer": "<? new Random().nextInt(@sys-number.numeric_value + 1) ?>"
  },
  "output": [
    "generic": [
      {
        "values": [
          {
            "text": "Here's a random number between 0 and @sys-number.literal: $answer."
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ]
}

```

See the [java.util.Random reference documentation](#) for information about other methods.

You can use standard methods of the following classes also:

- `java.lang.Byte`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Double`
- `java.lang.Short`
- `java.lang.Float`

Objects

`JSONObject.clear()`

This method clears all values from the JSON object and returns null.

For example, you want to clear the current values from the `$user` context variable.

```
{
  "context": {
    "user": {
      "first_name": "John",
      "last_name": "Snow"
    }
  }
}
```

Use the following expression in the output to define a field that clears the object of its values.

```
{
  "output": {
    "object_eraser": "<? $user.clear() ?>"
  }
}
```

If you subsequently reference the `$user` context variable, it returns `{}` only.

You can use the `clear()` method on the `context` or `output` JSON objects in the body of the API /message call.

Clearing context

When you use the `clear()` method to clear the `context` object, it clears **all** variables except these ones:

- `context.conversation_id`
- `context.timezone`
- `context.system`

Warning: All context variable values means:

- All default values that were set for variables in nodes that have been triggered during the current session.
- Any updates made to the default values with information provided by the user or external services during the current session.

To use the method, you can specify it in an expression in a variable that you define in the `output` object. For example:

```
{
  "output": {
    "generic": [
      {
        "values": [

```

```

    {
      "text": "Response for this node."
    }
  ],
  "response_type": "text",
  "selection_policy": "sequential"
}
],
"context_eraser": "<? context.clear() ?>"
}
}

```

Clearing output

When you use the `clear()` method to clear the `output` object, it clears all variables except the one you use to clear the `output` object and any text responses that you define in the current node. It also does not clear these variables:

- `output.nodes_visited`
- `output.nodes_visited_details`

To use the method, you can specify it in an expression in a variable that you define in the `output` object. For example:

```

{
  "output": {
    "generic": [
      {
        "values": [
          {
            "text": "Have a great day!"
          }
        ],
        "response_type": "text",
        "selection_policy": "sequential"
      }
    ],
    "output_eraser": "<? output.clear() ?>"
  }
}

```

If a node earlier in the tree defines a text response of `I'm happy to help.` and then jumps to a node with the JSON `output` object defined above, then only `Have a great day.` is displayed as the response. The `I'm happy to help.` output is not displayed, because it is cleared and replaced with the text response from the node that is calling the `clear()` method.

JSONObject.has(String)

This method returns true if the complex `JSONObject` has a property of the input name.

For this Dialog runtime context:

```
{
  "context": {
    "user": {
      "first_name": "John",
      "last_name": "Snow"
    }
  }
}
```

```
}
```

Dialog node output:

```
{  
  "conditions": "$user.has('first_name')"  
}
```

Result: The condition is true because the user object contains the property `first_name`.

JSONObject.remove(String)

This method removes a property of the name from the input `JSONObject`. The `JSONElement` that is returned by this method is the `JSONElement` that is being removed.

For this Dialog runtime context:

```
{  
  "context": {  
    "user": {  
      "first_name": "John",  
      "last_name": "Snow"  
    }  
  }  
}
```

Dialog node output:

```
{  
  "context": {  
    "attribute_removed": "<? $user.remove('first_name') ?>"  
  }  
}
```

Result:

```
{  
  "context": {  
    "user": {  
      "last_name": "Snow"  
    },  
    "attribute_removed": {  
      "first_name": "John"  
    }  
  }  
}
```

com.google.gson.JsonObject support

In addition to the built-in methods, you can use standard methods of the `com.google.gson.JsonObject` class.

Strings

These methods help you work with text.

For information about how to recognize and extract certain types of Strings, such as people names and locations, from user

input, see [System entities](#).

Note: For methods that involve regular expressions, see [RE2 Syntax reference](#) for details about the syntax to use when you specify the regular expression.

String.append(Object)

This method appends an input object to the string as a string and returns a modified string.

For this Dialog runtime context:

```
{  
  "context": {  
    "my_text": "This is a text."  
  }  
}
```

This syntax:

```
{  
  "context": {  
    "my_text": "<? $my_text.append(' More text.') ?>"  
  }  
}
```

Results in this output:

```
{  
  "context": {  
    "my_text": "This is a text. More text."  
  }  
}
```

String.contains(String)

This method returns true if the string contains the input substring.

Input: "Yes, I'd like to go."

This syntax:

```
{  
  "conditions": "input.text.contains('Yes')"  
}
```

Results: The condition is true.

String.endsWith(String)

This method returns true if the string ends with the input substring.

For this input:

"What is your name?".

This syntax:

```
{
```

```
"conditions": "input.text.endsWith('?')"  
}
```

Results: The condition is true.

String.extract(String regexp, Integer groupIndex)

This method returns a string extracted by specified group index of the input regular expression.

For this input:

```
"Hello 123456".
```

This syntax:

```
{  
  "context": {  
    "number_extract": "<? input.text.extract('[\\d]+',0) ?>"  
  }  
}
```

Important: To process \\d as the regular expression, you have to escape both the backslashes by adding another \\: \\\\d

Result:

```
{  
  "context": {  
    "number_extract": "123456"  
  }  
}
```

String.find(String regexp)

This method returns true if any segment of the string matches the input regular expression. You can call this method against a JSONArray or JSONObject element, and it will convert the array or object to a string before making the comparison.

For this input:

```
"Hello 123456".
```

This syntax:

```
{  
  "conditions": "input.text.find('^[^\\d]*[\\d]{6}[^\\d]*$')"  
}
```

Result: The condition is true because the numeric portion of the input text matches the regular expression ^[^\\d]*[\\d]{6}[^\\d]*\$.

String.isEmpty()

This method returns true if the string is an empty string, but not null.

For this Dialog runtime context:

```
{  
  "context": {
```

```
        "my_text_variable": ""  
    }  
}
```

This syntax:

```
{  
  "conditions": "$my_text_variable.isEmpty()  
}
```

Results: The condition is true.

String.length()

This method returns the character length of the string.

For this input:

"Hello"

This syntax:

```
{  
  "context": {  
    "input_length": "<? input.text.length() ?>"  
  }  
}
```

Results in this output:

```
{  
  "context": {  
    "input_length": 5  
  }  
}
```

String.matches(String regexp)

This method returns true if the string matches the input regular expression.

For this input:

"Hello".

This syntax:

```
{  
  "conditions": "input.text.matches('^Hello$')"  
}
```

Result: The condition is true because the input text matches the regular expression `\^Hello\$`.

String.startsWith(String)

This method returns true if the string starts with the input substring.

For this input:

```
"What is your name?".
```

This syntax:

```
{  
  "conditions": "input.text.startsWith('What')"  
}
```

Results: The condition is true.

String.substring(Integer beginIndex, Integer endIndex)

This method gets a substring with the character at `beginIndex` and the last character set to index before `endIndex`. The `endIndex` character is not included.

For this Dialog runtime context:

```
{  
  "context": {  
    "my_text": "This is a text."  
  }  
}
```

This syntax:

```
{  
  "context": {  
    "my_text": "<? $my_text.substring(5, $my_text.length()) ?>"  
  }  
}
```

Results in this output:

```
{  
  "context": {  
    "my_text": "is a text."  
  }  
}
```

String.toLowerCase()

This method returns the original String converted to lowercase letters.

For this input:

```
"This is A DOG!"
```

This syntax:

```
{  
  "context": {  
    "input_lower_case": "<? input.text.toLowerCase() ?>"  
  }  
}
```

Results in this output:

```
{
```

```
"context": {  
    "input_upper_case": "this is a dog!"  
}  
}
```

String.toUpperCase()

This method returns the original String converted to upper case letters.

For this input:

```
"hi there".
```

This syntax:

```
{  
    "context": {  
        "input_upper_case": "<? input.text.toUpperCase() ?>"  
    }  
}
```

Results in this output:

```
{  
    "context": {  
        "input_upper_case": "HI THERE"  
    }  
}
```

String.trim()

This method trims any spaces at the beginning and the end of the string and returns the modified string.

For this Dialog runtime context:

```
{  
    "context": {  
        "my_text": "    something is here      "  
    }  
}
```

This syntax:

```
{  
    "context": {  
        "my_text": "<? $my_text.trim() ?>"  
    }  
}
```

Results in this output:

```
{  
    "context": {  
        "my_text": "something is here"  
    }  
}
```

java.lang.String support

In addition to the built-in methods, you can use standard methods of the `java.lang.String` class.

java.lang.String.format()

You can apply the standard Java String `format()` method to text. See [java.util.Formatter reference](#) for information about the syntax to use to specify the format details.

For example, the following expression takes three decimal integers (1, 1, and 2) and adds them to a sentence.

```
{  
  "formatted String": "<? T(java.lang.String).format('%d + %d equals %d', 1, 1, 2) ?>"  
}
```

Result: 1 + 1 equals 2.

To change the decimal placement for a number, use the following syntax:

```
{  
  <? T(String).format('%.2f',<number to format>) ?>  
}
```

For example, if the `$number` variable that needs to be formatted in US dollars is `4.5`, then a response such as, `Your total is $<? T(String).format('%.2f',$number) ?>` returns `Your total is $4.50`.

Indirect data type conversion

When you include an expression within text, as part of a node response, for example, the value is rendered as a String. If you want the expression to be rendered in its original data type, then do not surround it with text.

For example, you can add this expression to a dialog node response to return the entities that are recognized in the user input in String format:

The entities are <? entities ?>.

If the user specifies `Hello now` as the input, then the `@sys-date` and `@sys-time` entities are triggered by the `now` reference. The `entities` object is an array, but because the expression is included in text, the entities are returned in String format, like this:

The entities are 2018-02-02, 14:34:56.

If you do not include text in the response, then an array is returned instead. For example, if the response is specified as an expression only, not surrounded by text.

<? entities ?>

The entity information is returned in its native data type, as an array.

```
[  
  {  
    "entity": "sys-date", "location": [6,9], "value": "2018-02-02", "confidence": 1, "metadata":  
    {"calendar_type": "GREGORIAN", "timezone": "America/New_York"}  
  },  
  {  
    "entity": "sys-time", "location": [6,9], "value": "14:33:22", "confidence": 1, "metadata":  
    {"calendar_type": "GREGORIAN", "timezone": "America/New_York"}  
}
```

```
}
```

As another example, the following \$array context variable is an array, but the \$string_array context variable is a string.

```
{
  "context": {
    "array": [
      "one",
      "two"
    ],
    "array_in_string": "this is my array: $array"
  }
}
```

If you check the values of these context variables in the Try it out pane, you will see their values specified as follows:

```
$array: ["one", "two"]
```

```
$array_in_string: "this is my array: [\"one\", \"two\"]"
```

You can subsequently perform array methods on the \$array variable, such as <? \$array.removeValue('two') ?>, but not the \$array_in_string variable.

Expressions for accessing objects

You can write expressions that access objects and properties of objects by using the Spring Expression (SpEL) language. For more information, see [Spring Expression Language \(SpEL\)](#).

Evaluation syntax

To expand variable values inside other variables or invoke methods on properties and global objects, use the <? expression ?> expression syntax. For example:

- **Expanding a property**
 - "output": {"generic": [{"values": [{"text": "Your name is <? context.userName ?>"}]}]}
- **Invoking methods on properties of global objects**
 - "context": {"email": "<? @email.literal ?>"}

Shorthand syntax

Learn how to quickly reference the following objects by using the SpEL shorthand syntax:

- [Context variables](#)
- [Entities](#)
- [Intents](#)

Shorthand syntax for context variables

The following table shows examples of the shorthand syntax that you can use to write context variables in condition expressions.

Shorthand syntax	Full syntax in SpEL
\$card_type	context['card_type']
\$(card-type)	context['card-type']
\$card_type:VISA	context['card_type'] == 'VISA'
\$card_type:(MASTER CARD)	context['card_type'] == 'MASTER CARD'

You can include special characters, such as hyphens or periods, in context variable names. However, doing so can lead to problems when the SpEL expression is evaluated. The hyphen could be interpreted as a minus sign, for example. To avoid such problems, reference the variable by using either the full expression syntax or the shorthand syntax \$(variable-name) and do not use the following special characters in the name:

- Parentheses ()
- More than one apostrophe ''
- Quotation marks "

Shorthand syntax for entities

The following table shows examples of the shorthand syntax that you can use when referring to entities.

Shorthand syntax	Full syntax in SpEL
@year	entities['year']?.value
@year == 2016	entities['year']?.value == 2016
@year != 2016	entities['year']?.value != 2016
@city == 'Boston'	entities['city']?.value == 'Boston'
@city:Boston	entities['city']?.contains('Boston')
@city:(New York)	entities['city']?.contains('New York')

In SpEL, the question mark (?) prevents a null pointer exception from being triggered when an entity object is null.

If the entity value that you want to check for contains a space character, you cannot use the : operator for comparison. For example, if you want to check whether the city entity is Dublin (Ohio), you must use @city == 'Dublin (Ohio)' instead of @city:(Dublin (Ohio)).

Shorthand syntax for intents

The following table shows examples of the shorthand syntax that you can use when referring to intents.

Shorthand syntax	Full syntax in SpEL
#help	intent == 'help'
! #help	intent != 'help'
NOT #help	intent != 'help'
#help or #i_am_lost	(intent == 'help' intent == 'I_am_lost')

Built-in global variables

You can use the expression language to extract property information for the following global variables:

Global variable	Definition
context	JSON object part of the processed message.
entities[]	List of entities that supports default access to 1st element.
input	JSON object part of the processed message.
intents[]	List of intents that supports default access to first element.
output	JSON object part of the processed message.

Accessing entities

The entities array contains one or more entities that were recognized in user input.

While testing your dialog, you can see details of the entities that are recognized in user input by specifying this expression in

a dialog node response:

```
<? entities ?>
```

For the user input, *Hello now*, the service recognizes the @sys-date and @sys-time system entities, so the response contains these entity objects:

```
[  
{"entity": "sys-date", "location": [6, 9], "value": "2017-08-07",  
 "confidence": 1, "metadata": {"calendar_type": "GREGORIAN",  
 "timezone": "America/New_York"}},  
 {"entity": "sys-time", "location": [6, 9], "value": "15:01:00",  
 "confidence": 1, "metadata": {"calendar_type": "GREGORIAN",  
 "timezone": "America/New_York"}}  
]
```

When placement of entities in the input matters

When you use the shorthand expression, `@city.contains('Boston')`, in a condition, the dialog node returns true **only if** Boston is the first entity detected in the user input. Only use this syntax if the placement of entities in the input matters to you and you want to check the first mention only.

Use the full SpEL expression if you want the condition to return true any time the term is mentioned in the user input, regardless of the order in which the entities are mentioned. The condition, `entities['city']?.contains('Boston')` returns true when at least one 'Boston' city entity is found in all the @city entities, regardless of placement.

For example, a user submits "I want to go from Toronto to Boston." The @city:Toronto and @city:Boston entities are detected and are represented in the array that is returned as follows:

- `entities.city[0].value = 'Toronto'`
- `entities.city[1].value = 'Boston'`

Note: The order of entities in the array that is returned matches the order in which they are mentioned in the user input.

Entity properties

Each entity has a set of properties associated with it. You can access information about an entity through its properties.

Property	Definition	Usage tips
<code>confidence</code>	A decimal percentage that represents the service's confidence in the recognized entity. The confidence of an entity is either 0 or 1, unless you have activated fuzzy matching of entities. When fuzzy matching is enabled, the default confidence level threshold is 0.3. Whether or not fuzzy matching is enabled, system entities always have a confidence level of 1.0.	You can use this property in a condition to have it return false if the confidence level is not higher than a percent you specify.
<code>location</code>	A zero-based character offsets that indicates where the detected entity values begin and end in the input text.	Use <code>.literal</code> to extract the span of text between start and end index values that are stored in the location property.

Property	Definition	Usage tips
<code>value</code>	The entity value identified in the input.	This property returns the entity value as defined in the training data, even if the match was made against one of its associated synonyms. You can use <code>.values</code> to capture multiple occurrences of an entity that might be present in user input.

Entity property usage examples

In the following examples, the workspace contains an airport entity that includes a value of JFK, and the synonym 'Kennedy Airport'. The user input is *I want to go to Kennedy Aiport*.

- To return a specific response if the 'JFK' entity is recognized in the user input, you could add this expression to the response condition: `entities.airport[0].value == 'JFK'` or `@airport = "JFK"`
- To return the entity name as it was specified by the user in the dialog response, use the `.literal` property: `So you want to go to <?entities.airport[0].literal?>...` or `So you want to go to @airport.literal`
...

Both formats evaluate to 'So you want to go to Kennedy Airport...' in the response.

- Expressions like `@airport:(JFK)` or `@airport.contains('JFK')` always refer to the **value** of the entity (JFK in this example).
- To be more restrictive about which terms are identified as airports in the input when fuzzy matching is enabled, you can specify this expression in a node condition, for example: `@airport && @airport.confidence > 0.7`. The node will only execute if the service is 70% confident that the input text contains an airport reference.

In this example, the user input is *Are there places to exchange currency at JFK, Logan, and O'Hare?*

- To capture multiple occurrences of an entity type in user input, use syntax like this:

```
"context":{  
  "airports":"@airport.values"  
}
```

To later refer to the captured list in a dialog response, use this syntax: `You asked about these airports: <? $airports.join(', ') ?>`. It is displayed like this: You asked about these airports: JFK, Logan, O'Hare.

Accessing intents

The intents array contains one or more intents that were recognized in the user input, sorted in descending order of confidence.

Each intent has one property only: the `confidence` property. The `confidence` property is a decimal percentage that represents the service's confidence in the recognized intent.

While testing your dialog, you can see details of the intents that are recognized in user input by specifying this expression in a dialog node response:

```
<? intents ?>
```

For the user input, *Hello now*, the service finds an exact match with the `#greeting` intent. Therefore, it lists the `#greeting`

intent object details first. The response also includes the top 10 other intents that are defined in the skill regardless of their confidence score. (In this example, its confidence in the other intents is set to 0 because the first intent is an exact match.) The top 10 intents are returned because the "Try it out" pane sends the alternate_intents:true parameter with its request. If you are using the API directly and want to see the top 10 results, be sure to specify this parameter in your call. If alternate_intents is false, which is the default value, only intents with a confidence above 0.2 are returned in the array.

```
[{"intent": "greeting", "confidence": 1},  
 {"intent": "yes", "confidence": 0},  
 {"intent": "pizza-order", "confidence": 0}]
```

The following examples show how to check for an intent value:

- `intents[0] == 'Help'`
- `intent == 'Help'`

`intent == 'help'` differs from `intents[0] == 'help'` because `intent == 'help'` does not throw an exception if no intent is detected. It is evaluated as true only if the intent confidence exceeds a threshold. If you want to, you can specify a custom confidence level for a condition, for example, `intents.size() > 0 && intents[0] == 'help' && intents[0].confidence > 0.1`

Accessing input

The input JSON object contains one property only: the `text` property. The `text` property represents the text of the user input.

Input property usage examples

The following example shows how to access input:

- To execute a node if the user input is "Yes", add this expression to the node condition: `input.text == 'Yes'`

You can use any of the [String methods](#) to evaluate or manipulate text from the user input. For example:

- To check whether the user input contains "Yes", use: `input.text.contains('Yes')`.
- Returns true if the user input is a number: `input.text.matches('[0-9]+')`.
- To check whether the input string contains ten characters, use: `input.text.length() == 10`.

Sample apps

Explore our sample applications to understand what you can develop with Watson Assistant.

Simple chat app

The Node.js app shows how Watson Assistant uses intents in a simple chat interface. It shows the conversation with an end user and the JSON responses at each turn of the conversation.

[See the demo](#) or [fork the code](#) .

Watson Assistant and car dashboard UI

This Node.js app is a fully developed example of the type of app you can build with Watson Assistant that uses intents, entities, and dialog.

[View the demo](#) .

Watson Assistant and Discovery

This app demonstrates the combination of Watson Assistant and Discovery. First, users pose questions to the Watson Assistant service. If Watson Assistant is not able to answer confidently, Watson Assistant Enhanced executes a call to Discovery to provide the user with a list of helpful answers.

[See the demo](#) or [fork the code](#) .

Watson Assistant and Tone Analyzer

IBM Watson™ Tone Analyzer uses linguistic analysis to detect three types of tones from written text: emotions, social tendencies, and writing style. The Watson Food Coach app is an example of Tone Analyzer integrated with Watson Assistant. In this app, depending on the kind of food the user ate and how he or she is feeling about it, the automated agent provides an appropriate coaching response to encourage the user to make healthy choices.

Note: The demo refers to Watson Assistant by its former name, Watson Conversation.

[View the demo](#) or [fork the code](#) .

Supported languages

The Watson Assistant service supports the languages listed. Individual features of the service are supported to a greater or lesser extent for each language.

In the following tables, the level of language and feature support is indicated by these codes:

- **GA** - The feature is generally available and supported for this language. Note that features may continue to be updated even after they are generally available.
- **Beta** - The feature is supported only as a Beta release, and is still undergoing testing before it is made generally available in this language.
- **NA** - Indicates that a feature is not available in this language.

The first table shows the level of support for all features, except those related to entities which are shown in the second table.

Table 1. Feature support details

Language	Feature support details			
	Defining intents	dialog	Absolute scoring	'Mark as irrelevant'
English (en)	GA		GA	GA
Arabic (ar)	GA		Beta	GA
Chinese (Simplified) (zh-cn)	GA		GA	NA
Chinese (Traditional) (zh-tw)	Beta		Beta	NA
Czech (cs)	GA		GA	NA
Dutch (nl)	GA		GA	NA
French (fr)	GA		GA	GA
German (de)	GA		GA	GA
Italian (it)	GA		GA	GA
Japanese (ja)	GA		GA	GA
Korean (ko)	GA		GA	NA
Portuguese (Brazilian) (pt-br)	GA		GA	GA
Spanish (es)	GA		GA	GA

Table 2. Entity feature support details

Language	Entity feature support details			
	Defining entities	System entities (number, currency, percentage, date, time)	Entity fuzzy matching	Contextual entities
English (en)	GA	GA	Beta (Stemming, misspelling, and partial match)	Beta

<u>Language</u>	<u>Defining entities</u>	<u>System entities (number, currency, percentage, date, time)</u>	<u>Entity fuzzy matching</u>	<u>Contextual entities</u>
Arabic (ar)	GA	Beta	Beta (Misspelling only)	NA
Chinese (Simplified) (zh-cn)	GA	GA	Beta (Misspelling only)	NA
Chinese (Traditional) (zh-tw)	Beta	Beta	Beta	NA
Czech (cs)	Beta	Beta	Beta (Misspelling only)	NA
Dutch (nl)	GA	GA	Beta (Misspelling only)	NA
French (fr)	GA	GA	Beta (Misspelling only)	NA
German (de)	GA	GA	Beta (Misspelling only)	NA
Italian (it)	GA	GA	Beta (Misspelling only)	NA
Japanese (ja)	GA	GA	Beta (Misspelling only)	NA
Korean (ko)	GA	GA	Beta (Misspelling only)	NA
Portuguese (Brazilian) (pt-br)	GA	GA	Beta (Misspelling only)	NA
Spanish (es)	GA	GA	Beta (Misspelling only)	NA

Note: The Watson Assistant service supports multiple languages as noted, but the tooling interface itself (descriptions, labels, etc.) is in English. All supported languages can be input and trained through the English interface.

GB18030 compliance: GB18030 is a Chinese standard that specifies an extended code page for use in the Chinese market. This code page standard is important for the software industry because the China National Information Technology Standardization Technical Committee has mandated that any software application that is released for the Chinese market after September 1, 2001, be enabled for GB18030. The Watson Assistant service supports this encoding, and is certified GB18030-compliant

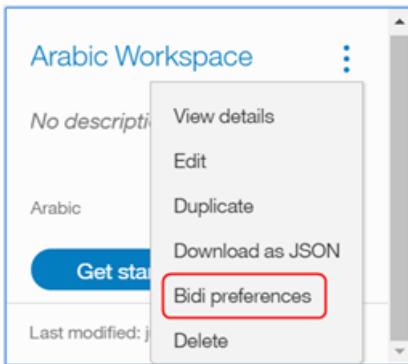
Changing a workspace language

Once a workspace has been created, its language cannot be modified. If it is necessary to change the supported language of a workspace, the user should download the workspace. Then, edit the resulting JSON file in a text editor, searching for a JSON property called `language`.

The `language` property should be set to the original language of the workspace; for example, English would be `en`. Modify the value of this property, changing it to the desired language (`fr` for French, `de` for German, etc.). Save the changes to the JSON file, and import the modified file into your Watson Assistant service instance.

Configuring bi-directional languages

For bi-directional languages, for example Arabic, you can change your workspace preferences accordingly. From your workspace tab, select the `Actions` drop-down menu, and select **Bidi preferences** (this option is only available for workspaces set to a bi-directional language):



Select from the following options for your workspace:

- **GUI Direction:** Specifies the layout direction of elements, such as buttons or menus, in the graphical user interface. Choose LTR (left-to-right) or RTL (right-to-left). If not specified, the tool follows the web browser GUI direction setting.
- **Text Direction:** Specifies the direction of typed text. Choose LTR (left-to-right) or RTL (right-to-left), or select Auto which will automatically choose the text direction based on your system settings. The None option will display left-to-right text.
- **Numeric Shaping:** Specifies which form of numerals to use when presenting regular digits. Choose from Nominal, Arabic-Indic, or Arabic-European. The None option will display Western numerals.
- **Calendar Type:** Specifies how you choose filtering dates in the workspace UI. Choose Islamic-Civil, Islamic-Tabular, Islamic-Umm al-Qura, or Gregorian. **Note:** This setting does not apply to the "Try it out" panel.

X

Edit Bidi Preferences

GUI Direction

Select a GUI direction ▾

Text Direction

Auto ▾

Numeric Shaping

Arabic Indic ▾

Calendar Type

Islamic-Civil ▾

A modal dialog titled 'Edit Bidi Preferences'. It contains four dropdown menus: 'GUI Direction' (set to 'Select a GUI direction'), 'Text Direction' (set to 'Auto'), 'Numeric Shaping' (set to 'Arabic Indic'), and 'Calendar Type' (set to 'Islamic-Civil'). At the bottom are 'Cancel' and 'Update' buttons, with 'Update' being the primary action button.

When finished making selections, click **Update** to save and return to the workspace tab.

Working with accented characters

In a conversational setting, users may or may not use accents while interacting with the Watson Assistant service. As such, both accented and non-accented versions of words may be treated the same for intent detection and entity recognition.

However for some languages, like Spanish, some accents can alter the meaning of the entity. Thus, for entity detection, although the original entity may implicitly have an accent, the service can also match the non-accented version of the same entity, but with a slightly lower confidence score.

For example, for the word "barrió", which has an accent and corresponds to the past tense of the verb "barrer" (to sweep), the service can also match the word "barrio" (neighborhood), but with a slightly lower confidence.

The system will provide the highest confidence scores in entities with exact matches. For example, `barrio` will not be detected if `barrió` is in the training set; and `barrió` will not be detected if `barrio` is in the training set.

You are expected to train the system with the proper characters and accents. For example, if you are expecting `barrió` as a response, then you should put `barrió` into the training set.

Although not an accent mark, the same applies to words using, for example, the Spanish letter ñ vs. the letter n, such as "uña" vs. "una". In this case the letter ñ is not simply an n with an accent; it is a unique, Spanish-specific letter.

You can enable fuzzy matching if you think your customers will not use the appropriate accents, or misspell words (including, for example, putting a n instead of a ñ), or you can explicitly include them in the training examples.

Information security

IBM is committed to providing our clients and partners with innovative data privacy, security and governance solutions.

Notice: Clients are responsible for ensuring their own compliance with various laws and regulations, including the European Union General Data Protection Regulation (GDPR). Clients are solely responsible for obtaining advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulations that may affect the clients' business and any actions the clients may need to take to comply with such laws and regulations.

The products, services, and other capabilities described herein are not suitable for all client situations and may have restricted availability. IBM does not provide legal, accounting or auditing advice or represent or warrant that its services or products will ensure that clients are in compliance with any law or regulation.

If you need to request GDPR support for IBM Cloud™ Watson resources that are created

- In the European Union, see [Requesting support for IBM Cloud Watson resources created in the European Union](#) [REDACTED].
- Outside the European Union, see [Requesting support for resources outside the European Union](#) [REDACTED].

European Union General Data Protection Regulation (GDPR)

IBM is committed to providing our clients and partners with innovative data privacy, security and governance solutions to assist them on their journey to GDPR compliance.

Learn more about IBM's own GDPR readiness journey and our GDPR capabilities and offerings to support your compliance journey [here](#) [REDACTED].

Data in Watson Assistant

Do not add personal data to the training data (entities and intents, including user examples) that you create.

In the IBM Cloud Private environment, the tool does not include metrics for analyzing conversations that your assistant is having with your users. Therefore, no log data is stored currently.

System entities

System entity details

This reference section provides complete information about the available system entities. For more information about system entities and how to use them, refer to [Defining entities](#) and search for "Enabling system entities".

System entities are available for languages noted in the [Supported languages](#) topic.

@sys-currency entity

The @sys-currency system entity detects monetary currency values that are expressed in an utterance with a currency symbol or currency-specific terms. A numeric value is returned.

Recognized formats

- 20 cents
- Five dollars
- \$10

Metadata

- `.numeric_value`: the canonical numeric value as an integer or a double, in base units
- `.unit`: the base unit currency code (for example, 'USD' or 'EUR')

Returns

For the input twenty dollars or \$1,234.56, @sys-currency returns these values:

Attribute	Type	Returned for twenty dollars	Returned for \$1,234.56
@sys-currency	string	20	1234.56
@sys-currency.literal	string	twenty dollars	\$1,234.56
@sys-currency.numeric_value	number	20	1234.56
@sys-currency.location	array	[0,14]	[0,9]
@sys-currency.unit	string	USD*	USD

*@sys-currency.unit always returns the 3-letter ISO currency code.

For the input veinte euro or €1.234,56, in Spanish, @sys-currency returns these values:

Attribute	Type	Returned for veinte euro	Returned for €1.234,56
@sys-currency	string	20	1234.56
@sys-currency.literal	string	veinte euro	€1.234,56
@sys-currency.numeric_value	number	20	1234.56

Attribute	Type	Returned for veinte euro	Returned for €1.234,56
@sys-currency.location	array	[0,11]	[0,9]
@sys-currency.unit	string	EUR*	EUR

*@sys-currency.unit always returns the 3-letter ISO currency code.

You get equivalent results for other supported languages and national currencies.

@system-currency usage tips

- Currency values are recognized as instances of @sys-number entities as well. If you are using separate conditions to check for both currency values and numbers, place the condition that checks for currency above the one that checks for a number.
- If you use the @sys-currency entity as a node condition and the user specifies \$0 as the value, the value is recognized as a currency properly, but the condition is evaluated to the number zero, not the currency zero. As a result, it does not return the expected response. To check for currency values in a way that handles zeros properly, use the full SpEL expression syntax entities['sys-currency']?.value in the node condition instead.

@sys-date and @sys-time entities

The @sys-date system entity extracts mentions such as Friday, today, or November 1. The value of this entity stores the corresponding inferred date as a string in the format "yyyy-MM-dd" e.g. "2016-11-21". The system augments missing elements of a date (such as the year for "November 21") with the current date values.

Note: - For English locale only, the default system behavior for date input is MM/DD/YYYY. This will change to DD/MM/YYYY only if the first two numbers are greater than 12. The value stored will still be in the format "yyyy-MM-dd".

The @sys-time system entity extracts mentions such as 2pm, at 4, or 15:30. The value of this entity stores the time as a string in the format "HH:mm:ss". For example, "13:00:00."

Date-time mentions

Mentions of date and time, such as now or two hours from now are extracted as two separate entity mentions - one @sys-date and one @sys-time. These two mentions are not linked to one another except that they share the same literal string that spans the complete date-time mention.

Multi-word date-time mentions such as on Monday at 4pm are also extracted as two @sys-date and @sys-time mentions. When mentioned together consecutively they also share a single literal string that spans the complete date-time mention.

Date and time ranges

Mentions of a date range such as the weekend, next week, or from Monday to Friday are extracted as a pair of @sys-date entity mentions that show the start and end of the range. Similarly, mentions of time ranges such as from 2 to 3 are extracted as two @sys-time entities, showing the start and end times. The two entities in the pair share a literal string that corresponds to the full date or time range mention.

Last and Next dates and times

In some locales, a phrase like "last Monday" is used to specify the Monday of the previous week only. In contrast, other locales use "last Monday" to specify the last day which was a Monday, but which may have been either in the same week or the previous week.

As an example, for Friday June 16, in some locales "last Monday" could refer to either June 12 or to June 5, while in other locales it refers only to June 5 (the previous week). This same logic holds true for a phrase like "next Monday".

The Watson Assistant service treats "last" and "next" dates as referring to the most immediate last or next day referenced, which may be in either the same or a previous week.

For time phrases like "for the last 3 days" or "in the next 4 hours", the logic is equivalent. For example, in the case of "in the next 4 hours", this results in two @sys-time entities: one of the current time, and one of the time four hours later than the current time.

Time zones

Mentions of a date or time that are relative to the current time are resolved with respect to a chosen time zone. By default, this is UTC (GMT). This means that by default, REST API clients located in time zones different from UTC will observe the value of now extracted according to the current UTC time.

Optionally, the REST API client can add the local timezone as the context variable \$timezone. This context variable should be sent with every client request. For example, the \$timezone value should be America/Los_Angeles, EST, or UTC. For a full list of supported time zones, see [Supported time zones](#).

When the \$timezone variable is provided, the values of relative @sys-date and @sys-time mentions are computed based on the client time zone instead of UTC.

Examples of mentions relative to time zones

- now
- in two hours
- today
- tomorrow
- 2 days from now

Recognized formats

- November 21
- 10:30
- at 6 pm
- this weekend

Returns

For the input November 21 @sys-date returns these values:

Attribute	Type	Returned for November 21
@sys-date.literal	string	November 21
@sys-date	string	20xx-11-21 *
@sys-date.location	array	[0,11]
@sys-date.calendar_type	string	GREGORIAN

- @sys-date always returns the date in this format: yyyy-MM-dd.

- * Returns the next matching date. If that date has already passed this year, this returns next year's date.

For the input at 6 pm @sys-time returns these values:

Attribute	Type	Returned for at 6 pm
@sys-time.literal	string	at 6 pm
@sys-time	string	18:00:00
@sys-time.location	array	[0,7]
@sys-time.calendar_type	string	GREGORIAN

- @sys-time always returns the time in this format: HH:mm:ss.

For information about processing date and time values, see the [Date and time](#) method reference.

@sys-number entity

The @sys-number system entity detects numbers that are written using either numerals or words. In either case, a numeric value is returned.

Recognized formats

- 21
- twenty one
- 3.13

Metadata

- .numeric_value - the canonical numeric value as an integer or a double

Returns

For the input twenty or 1,234.56, @sys-number returns these values:

Attribute	Type	Returned for twenty	Returned for 1,234.56
@sys-number	string	20	1234.56
@sys-number.literal	string	twenty	1,234.56
@sys-number.location	array	[0,6]	[0,8]
@sys-number.numeric_value	number	20	1234.56

For the input veinte or 1.234,56, in Spanish, @sys-number returns these values:

Attribute	Type	Returned for veinte	Returned for 1.234,56
@sys-number	string	20	1234.56
@sys-number.literal	string	veinte	1.234,56

Attribute	Type	Returned for veinte	Returned for 1.234,56
@sys-number.location	array	[0,6]	[0,8]
@sys-number.numeric_value	number	20	1234.56

You get equivalent results for other supported languages.

@system-number usage tips

- If you use the @sys-number entity as a node condition and the user specifies zero as the value, the 0 value is recognized properly as a number, but the condition is evaluated to false and cannot return the associated response properly. To check for numbers in a way that handles zeros properly, use the full SpEL expression syntax `entities['sys-number']?.value` in the node condition instead.
- If you use @sys-number to compare number values in a condition, be sure to separately include a check for the presence of a number itself. If no number is found, @sys-number evaluates to null, which might result in your comparison evaluating to true even when no number is present.

For example, do not use `@sys-number<4` alone because if no number is found, @sys-number evaluates to null. Because null is less than 4, the condition evaluates to true even though no number is present.

Use `@sys-number AND @sys-number<4` instead. If no number is present, the first condition evaluates to false, which appropriately results in the whole condition evaluating to false.

For information about processing number values, see the [Numbers](#) method reference.

@sys-percentage entity

The @sys-percentage system entity detects percentages that are expressed in an utterance with the percent symbol or written out using the word **percent**. In either case, a numeric value is returned.

Recognized formats

- 15%
- 10 percent

Metadata

.numeric_value: the canonical numeric value as an integer or a double

Returns

For the input 1,234.56%, @sys-percentage returns these values:

Attribute	Type	Returned for 1,234.56%
@sys-percentage	string	1234.56
@sys-percentage.literal	string	1,234.56%
@sys-percentage.location	array	[0,9]
@sys-percentage.numeric_value	number	1234.56

For the input `1.234,56%`, in Spanish, `@sys-currency` returns these values:

Attribute	Type	Returned for <code>1.234,56%</code>
<code>@sys-percentage</code>	string	<code>1234.56</code>
<code>@sys-percentage.literal</code>	string	<code>1.234,56%</code>
<code>@sys-percentage.location</code>	array	<code>[0,9]</code>
<code>@sys-percentage.numeric_value</code>	number	<code>1234.56</code>

You get equivalent results for other supported languages.

`@system-percentage` usage tips

- Percentage values are recognized as instances of `@sys-number` entities as well. If you are using separate conditions to check for both percentage values and numbers, place the condition that checks for a percentage above the one that checks for a number.
- If you use the `@sys-percentage` entity as a node condition and the user specifies `0%` as the value, the value is recognized as a percentage properly, but the condition is evaluated to the number zero not the percentage `0%`. Therefore, it does not return the expected response. To check for percentages in a way that handles zero percentages properly, use the full SpEL expression syntax `entities['sys-percentage']?.value` in the node condition instead.
- If you input a value like `1-2%`, the values `1%` and `2%` are returned as system entities. The index will be the whole range between `1%` and `2%`, and both entities will have the same index.

Time zones supported by system entities

This list of supported time zones can be used with the time zone functions related to the [@sys-date and @sys-time entities](#).

Time zone	Time zone
<code>Africa/Abidjan</code>	<code>Africa/Accra</code>
<code>Africa/Addis_Ababa</code>	<code>Africa/Algiers</code>
<code>Africa/Asmara</code>	<code>Africa/Asmera</code>
<code>Africa/Bamako</code>	<code>Africa/Bangui</code>
<code>Africa/Banjul</code>	<code>Africa/Bissau</code>
<code>Africa/Blantyre</code>	<code>Africa/Brazzaville</code>
<code>Africa/Bujumbura</code>	<code>Africa/Cairo</code>
<code>Africa/Casablanca</code>	<code>Africa/Ceuta</code>
<code>Africa/Conakry</code>	<code>Africa/Dakar</code>
<code>Africa/Dar_es_Salaam</code>	<code>Africa/Djibouti</code>
<code>Africa/Douala</code>	<code>Africa/El_Aaiun</code>

Time zone	Time zone
Africa/Freetown	Africa/Gaborone
Africa/Harare	Africa/Johannesburg
Africa/Juba	Africa/Kampala
Africa/Khartoum	Africa/Kigali
Africa/Kinshasa	Africa/Lagos
Africa/Libreville	Africa/Lome
Africa/Luanda	Africa/Lubumbashi
Africa/Lusaka	Africa/Malabo
Africa/Maputo	Africa/Maseru
Africa/Mbabane	Africa/Mogadishu
Africa/Monrovia	Africa/Nairobi
Africa/Ndjamena	Africa/Niamey
Africa/Nouakchott	Africa/Ouagadougou
Africa/Porto-Novo	Africa/Sao_Tome
Africa/Timbuktu	Africa/Tripoli
Africa/Tunis	Africa/Windhoek
America/Adak	America/Anchorage
America/Anguilla	America/Antigua
America/Araguaina	America/Argentina/Buenos_Aires
America/Argentina/Catamarca	America/Argentina/ComodRivadavia
America/Argentina/Cordoba	America/Argentina/Jujuy
America/Argentina/La_Rioja	America/Argentina/Mendoza
America/Argentina/Rio_Gallegos	America/Argentina/Salta
America/Argentina/San_Juan	America/Argentina/San_Luis
America/Argentina/Tucuman	America/Argentina/Ushuaia
America/Aruba	America/Asuncion
America/Atikokan	America/Atka
America/Bahia	America/Bahia_Banderas

Time zone	Time zone
America/Barbados	America/Belem
America/Belize	America/Blanc-Sablon
America/Boa_Vista	America/Bogota
America/Boise	America/Buenos_Aires
America/Cambridge_Bay	America/Campo_Grande
America/Cancun	America/Caracas
America/Catamarca	America/Cayenne
America/Cayman	America/Chicago
America/Chihuahua	America/Coral_Harbour
America/Cordoba	America/Costa_Rica
America/Creston	America/Cuiaba
America/Curacao	America/Danmarkshavn
America/Dawson	America/Dawson_Creek
America/Denver	America/Detroit
America/Dominica	America/Edmonton
America/Eirunepe	America/El_Salvador
America/Ensenada	America/Fort_Nelson
America/Fort_Wayne	America/Fortaleza
America/Glace_Bay	America/Godthab
America/Goose_Bay	America/Grand_Turk
America/Grenada	America/Guadeloupe
America/Guatemala	America/Guayaquil
America/Guyana	America/Halifax
America/Havana	America/Hermosillo
America/Indiana/Indianapolis	America/Indiana/Knox
America/Indiana/Marengo	America/Indiana/Petersburg
America/Indiana/Tell_City	America/Indiana/Vevay
America/Indiana/Vincennes	America/Indiana/Winamac

Time zone	Time zone
America/Indianapolis	America/Inuvik
America/Iqaluit	America/Jamaica
America/Jujuy	America/Juneau
America/Kentucky/Louisville	America/Kentucky/Monticello
America/Knox_IN	America/Kralendijk
America/La_Paz	America/Lima
America/Los_Angeles	America/Louisville
America/Lower_Princes	America/Maceio
America/Managua	America/Manaus
America/Marigot	America/Martinique
America/Matamoros	America/Mazatlan
America/Mendoza	America/Menominee
America/Merida	America/Metlakatla
America/Mexico_City	America/Miquelon
America/Moncton	America/Monterrey
America/Montevideo	America/Montreal
America/Montserrat	America/Nassau
America/New_York	America/Nipigon
America/Nome	America/Noronha
America/North_Dakota/Beulah	America/North_Dakota/Center
America/North_Dakota/New_Salem	America/Ojinaga
America/Panama	America/Pangnirtung
America/Paramaribo	America/Phoenix
America/Port-au-Prince	America/Port_of_Spain
America/Porto_Acre	America/Porto_Velho
America/Puerto_Rico	America/Rainy_River
America/Rankin_Inlet	America/Recife
America/Regina	America/Resolute

Time zone	Time zone
America/Rio_Branco	America/Rosario
America/Santa_Isabel	America/Santarem
America/Santiago	America/Santo_Domingo
America/Sao_Paulo	America/Scoresbysund
America/Shiprock	America/Sitka
America/St_Barthelemy	America/St_Johns
America/St_Kitts	America/St_Lucia
America/St_Thomas	America/St_Vincent
America/Swift_Current	America/Tegucigalpa
America/Thule	America/Thunder_Bay
America/Tijuana	America/Toronto
America/Tortola	America/Vancouver
America/Virgin	America/Whitehorse
America/Winnipeg	America/Yakutat
America/Yellowknife	Antarctica/Casey
Antarctica/Davis	Antarctica/DumontDUrville
Antarctica/Macquarie	Antarctica/Mawson
Antarctica/Mcmurdo	Antarctica/Palmer
Antarctica/Rothera	Antarctica/South_Pole
Antarctica/Syowa	Antarctica/Troll
Antarctica/Vostok	Arctic/Longyearbyen
Asia/Aden	Asia/Almaty
Asia/Amman	Asia/Anadyr
Asia/Aqtau	Asia/Aqtobe
Asia/Ashgabat	Asia/Ashkhabad
Asia/Baghdad	Asia/Bahrain
Asia/Baku	Asia/Bangkok
Asia/Beirut	Asia/Bishkek

Time zone	Time zone
Asia/Brunei	Asia/Calcutta
Asia/Chita	Asia/Choibalsan
Asia/Chongqing	Asia/Chungking
Asia/Colombo	Asia/Dacca
Asia/Damascus	Asia/Dhaka
Asia/Dili	Asia/Dubai
Asia/Dushanbe	Asia/Gaza
Asia/Harbin	Asia/Hebron
Asia/Ho_Chi_Minh	Asia/Hong_Kong
Asia/Hovd	Asia/Irkutsk
Asia/Istanbul	Asia/Jakarta
Asia/Jayapura	Asia/Jerusalem
Asia/Kabul	Asia/Kamchatka
Asia/Karachi	Asia/Kashgar
Asia/Kathmandu	Asia/Katmandu
Asia/Khandyga	Asia/Kolkata
Asia/Krasnoyarsk	Asia/Kuala_Lumpur
Asia/Kuching	Asia/Kuwait
Asia/Macao	Asia/Macau
Asia/Magadan	Asia/Makassar
Asia/Manila	Asia/Muscat
Asia/Nicosia	Asia/Novokuznetsk
Asia/Novosibirsk	Asia/Omsk
Asia/Oral	Asia/Phnom_Penh
Asia/Pontianak	Asia/Pyongyang
Asia/Qatar	Asia/Qyzylorda
Asia/Rangoon	Asia/Riyadh
Asia/Saigon	Asia/Sakhalin

Time zone	Time zone
Asia/Samarkand	Asia/Seoul
Asia/Shanghai	Asia/Singapore
Asia/Srednekolymsk	Asia/Taipei
Asia/Tashkent	Asia/Tbilisi
Asia/Tehran	Asia/Tel_Aviv
Asia/Thimbu	Asia/Thimphu
Asia/Tokyo	Asia/Ujung_Pandang
Asia/Ulaanbaatar	Asia/Ulan_Bator
Asia/Urumqi	Asia/Ust-Nera
Asia/Vientiane	Asia/Vladivostok
Asia/Yakutsk	Asia/Yekaterinburg
Asia/Yerevan	Atlantic/Azores
Atlantic/Bermuda	Atlantic/Canary
Atlantic/Cape_Verde	Atlantic/Faeroe
Atlantic/Faroe	Atlantic/Jan_Mayen
Atlantic/Madeira	Atlantic/Reykjavik
Atlantic/South_Georgia	Atlantic/St_Helena
Atlantic/Stanley	Australia/ACT
Australia/Adelaide	Australia/Brisbane
Australia/Broken_Hill	Australia/Canberra
Australia/Currie	Australia/Darwin
Australia/Eucla	Australia/Hobart
Australia/LHI	Australia/Lindeman
Australia/Lord_Howe	Australia/Melbourne
Australia/NSW	Australia/North
Australia/Perth	Australia/Queensland
Australia/South	Australia/Sydney
Australia/Tasmania	Australia/Victoria

Time zone	Time zone
Australia/West	Australia/Yancowinna
Brazil/Acre	Brazil/DeNoronha
Brazil/East	Brazil/West
CET	CST6CDT
Canada/Atlantic	Canada/Central
Canada/East-Saskatchewan	Canada/Eastern
Canada/Mountain	Canada/Newfoundland
Canada/Pacific	Canada/Saskatchewan
Canada/Yukon	Chile/Continental
Chile/EasterIsland	Cuba
EET	EST5EDT
Egypt	Eire
Etc/GMT	Etc/GMT+0
Etc/GMT+1	Etc/GMT+10
Etc/GMT+11	Etc/GMT+12
Etc/GMT+2	Etc/GMT+3
Etc/GMT+4	Etc/GMT+5
Etc/GMT+6	Etc/GMT+7
Etc/GMT+8	Etc/GMT+9
Etc/GMT-0	Etc/GMT-1
Etc/GMT-10	Etc/GMT-11
Etc/GMT-12	Etc/GMT-13
Etc/GMT-14	Etc/GMT-2
Etc/GMT-3	Etc/GMT-4
Etc/GMT-5	Etc/GMT-6
Etc/GMT-7	Etc/GMT-8
Etc/GMT-9	Etc/GMT0
Etc/Greenwich	Etc/UCT

Time zone	Time zone
Etc/UTC	Etc/Universal
Etc/Zulu	Europe/Amsterdam
Europe/Andorra	Europe/Athens
Europe/Belfast	Europe/Belgrade
Europe/Berlin	Europe/Bratislava
Europe/Brussels	Europe/Bucharest
Europe/Budapest	Europe/Busingen
Europe/Chisinau	Europe/Copenhagen
Europe/Dublin	Europe/Gibraltar
Europe/Guernsey	Europe/Helsinki
Europe/Isle_of_Man	Europe/Istanbul
Europe/Jersey	Europe/Kaliningrad
Europe/Kiev	Europe/Lisbon
Europe/Ljubljana	Europe/London
Europe/Luxembourg	Europe/Madrid
Europe/Malta	Europe/Mariehamn
Europe/Minsk	Europe/Monaco
Europe/Moscow	Europe/Nicosia
Europe/Oslo	Europe/Paris
Europe/Podgorica	Europe/Prague
Europe/Riga	Europe/Rome
Europe/Samara	Europe/San_Marino
Europe/Sarajevo	Europe/Simferopol
Europe/Skopje	Europe/Sofia
Europe/Stockholm	Europe/Tallinn
Europe/Tirane	Europe/Tiraspol
Europe/Uzhgorod	Europe/Vaduz
Europe/Vatican	Europe/Vienna

Time zone	Time zone
Europe/Vilnius	Europe/Volgograd
Europe/Warsaw	Europe/Zagreb
Europe/Zaporozhye	Europe/Zurich
GB	GB-Eire
GMT	GMT0
Greenwich	Hongkong
Iceland	Indian/Antananarivo
Indian/Chagos	Indian/Christmas
Indian/Cocos	Indian/Comoro
Indian/Kerguelen	Indian/Mahe
Indian/Maldives	Indian/Mauritius
Indian/Mayotte	Indian/Reunion
Iran	Israel
Jamaica	Japan
Kwajalein	Libya
MET	MST7MDT
Mexico/BajaNorte	Mexico/BajaSur
Mexico/General	NZ
NZ-CHAT	Navajo
PRC	PST8PDT
Pacific/Apia	Pacific/Auckland
Pacific/Bougainville	Pacific/Chatham
Pacific/Chuuk	Pacific/Easter
Pacific/Efate	Pacific/Enderbury
Pacific/Fakaofo	Pacific/Fiji
Pacific/Funafuti	Pacific/Galapagos
Pacific/Gambier	Pacific/Guadalcanal
Pacific/Guam	Pacific/Honolulu

Time zone	Time zone
Pacific/Johnston	Pacific/Kiritimati
Pacific/Kosrae	Pacific/Kwajalein
Pacific/Majuro	Pacific/Marquesas
Pacific/Midway	Pacific/Nauru
Pacific/Niue	Pacific/Norfolk
Pacific/Noumea	Pacific/Pago_Pago
Pacific/Palau	Pacific/Pitcairn
Pacific/Pohnpei	Pacific/Ponape
Pacific/Port_Moresby	Pacific/Rarotonga
Pacific/Saipan	Pacific/Samoa
Pacific/Tahiti	Pacific/Tarawa
Pacific/Tongatapu	Pacific/Truk
Pacific/Wake	Pacific/Wallis
Pacific/Yap	Poland
Portugal	ROK
Singapore	SystemV/AST4
SystemV/AST4ADT	SystemV/CST6
SystemV/CST6CDT	SystemV/EST5
SystemV/EST5EDT	SystemV/HST10
SystemV/MST7	SystemV/MST7MDT
SystemV/PST8	SystemV/PST8PDT
SystemV/YST9	SystemV/YST9YDT
Turkey	UCT
US/Alaska	US/Aleutian
US/Arizona	US/Central
US/East-Indiana	US/Eastern
US/Hawaii	US/Indiana-Starke
US/Michigan	US/Mountain

Time zone	Time zone
US/Pacific	US/Pacific-New
US/Samoa	UTC
Universal	W-SU
WET	Zulu
EST	HST
MST	ACT
AET	AGT
ART	AST
BET	BST
CAT	CNT
CST	CTT
EAT	ECT
IET	IST
JST	MIT
NET	NST
PLT	PNT
PRT	PST
SST	VST

Watson SDKs

SDKs abstract much of the complexity associated with application development. By providing programming interfaces in languages that you already know, they can help you get up and running quickly with IBM Watson services.

Supported SDKs

The following Watson SDKs are supported by IBM:

- [Android SDK](#)
- [Go SDK](#)
- [Java SDK](#)
- [Node.js SDK](#)
- [Python SDK](#)
- [Ruby SDK](#)
- [.NET SDK](#)
- [Salesforce SDK](#)
- [Swift SDK](#)
- [Unity SDK](#)

The [API reference](#) for each service includes information and examples for many of the SDKs, including Java, Node.js, Python, Go, Ruby, and Swift.

Community SDKs

The following SDKs are available from the Watson community of developers:

- [PHP SDK](#)
- [Scala SDK](#)

Release notes

Service API Versioning

API requests require a version parameter that takes a date in the format `version=YYYY-MM-DD`. Whenever we change the API in a backwards-incompatible way, we release a new minor version of the API.

Send the version parameter with every API request. The service uses the API version for the date you specify, or the most recent version before that date. Don't default to the current date. Instead, specify a date that matches a version that is compatible with your app, and don't change it until your app is ready for a later version.

- The current version is 2018-09-20.
- The "Try it out" pane in the Watson Assistant tooling is using version 2018-07-10.

Beta features

IBM releases services, features, and language support for your evaluation that are classified as beta. These features might be unstable, might change frequently, and might be discontinued with short notice. Beta features also might not provide the same level of performance or compatibility that generally available features provide and are not intended for use in a production environment. Beta features are supported only on the [IBM Developer Answers](#).

Change log

21 February 2019

IBM Watson™ Assistant for IBM® Cloud Private version 1.1 is available. : The Watson Assistant tool now works with IBM Cloud Private 3.1.0. It does not work with IBM Cloud Private 2.1.0.3. See [Upgrading](#).

- Watson Assistant for IBM Cloud Private version 1.1 is compatible with IBM® Cloud Private for Data version 1.2.
- The number of required Virtual Private CPUs has decreased from its previous number (of 60 VPCs). See [VPC requirements](#) for more details.
- Language support has been improved, which means you do not need as many additional resources when you add support for more languages. See [Language considerations](#) for more details.

23 November 2018

- A revised Helm chart (version 1.0.1) was published, which improves the Helm chart and packaging.
- New configuration settings were added that allow you to specify domain names and IP addresses for the master and proxy nodes of the IBM® Cloud Private cluster. A new checkbox is visible for enabling recommendations; however, do not select it as the feature is not fully supported yet.
- The resources required for a development deployment changed for Minio from one 20 GB replica to four 5 GB replicas. This change means you need to create 13 persistent volumes instead of 10 to support the deployment.

5 October 2018

- A revised Helm chart (version 1.0.0.1) was published, which improves the installation process.

26 September 2018

- **IBM Watson™ Assistant for IBM® Cloud Private 1.0.0 is available.**

The IBM Watson™ Assistant tool includes a Build tab that offers pre-built intents you can add to your workspace from a content catalog, the ability to define your own intents and entities, and has a graphical user interface you can use to build a dialog. The following key features are also available:

- Dialog: Digression and disambiguation support, nodes with slots, rich responses (including *Connect to human agent*)
- Entities: Contextual entities, system entities for currency, date, number, percentage, and time.
- Intents: Content catalog

These features are not available from IBM® Cloud Private, but are available in the public IBM Cloud instance at the time of this release:

- There are no metrics or analytics capabilities. Therefore, the *Improve* tab is not included in the tool.
- There are no deployment connectors or built-in integrations available. You must build a custom client application that can host the assistant. As a result, the *Deploy* tab is not included in the tool.
- You cannot search within the tool.
- The @sys-person and @sys-location system entities are not supported.
- You cannot make programmatic calls to Cloud Functions actions from the dialog.
- No entity synonym recommendations are available.
- No intent conflict detection is available.

Service information

The assistant that you build is hosted by IBM® Cloud Private.

Limits by artifact

For information about artifact limits per plan, see these topics:

- [Workspaces](#)
- [Dialog nodes](#)
- [Intents](#)
- [Entities](#)

Service API Versioning

API requests require a version parameter that takes a date in the format `version=YYYY-MM-DD`. Whenever we change the API in a backwards-incompatible way, we release a new minor version of the API.

Send the version parameter with every API request. The service uses the API version for the date you specify, or the most recent version before that date. Don't default to the current date. Instead, specify a date that matches a version that is compatible with your app, and don't change it until your app is ready for a later version.

The current version is 2018-09-20.

The "Try it out" pane in the Watson Assistant tooling is using version 2018-07-10.

Authenticating API calls on private cloud version 3.1.0

The authentication mechanism used by your service instance impacts how you must provide credentials when making an API call.

Note: The following instructions describe how to authenticate calls when using IBM® Cloud Private version 3.1.0. If you are using IBM® Cloud Private version 2.1.0.3, then see [Authenticating API calls on private cloud version 2.1.0.3](#) instead.

1. Get the service credentials by using the Kubernetes command line interface.

1. You should have already installed the Kubernetes CLI (`kubectl()`), and configured access to your cluster. If not, see [Accessing your cluster from the kubectl CLI](#).

2. Log in to IBM Cloud Private.

```
cloudctl login -a https://{{icp_url}}:8443
```

3. To get the API key for the secret, run the following command:

```
kubectl -n [namespace-name] get secret wcs-{{release-name}}-serviceid-secret -o go-template='{{ index .data "api_key" | base64decode }}'
```

The key is returned. For example: `icp-CXTvuAA2QwXZbETadG3zIpvqmi3djUmGBBBzV4803C6D`.

4. Copy the key.

2. Use these credentials in your API call.

- The base URL uses the syntax `https://{{global.icp.proxyHostname}}{{global.icp.ingress.path}}/api`. For example: `https://myproxy/myrelease/assistant/api`.

- Provide the API key when you call the service. The following example shows an API key being used.

```
curl -k -H "apikey:{API_KEY}" https://{{icp_url}}/assistant/api/v1/workspaces?
version=2018-09-20
```

To get a workspace ID, go to the **Workspaces** tab of the tool, find the workspace you want to access programmatically, and then from the menu, choose **View details**.

See the [IBM® Cloud Private overview](#) [] for more information about IBM Cloud Private.

Authenticating API calls on private cloud version 2.1.0.3

The authentication mechanism used by your service instance impacts how you must provide credentials when making an API call.

Note: The following instructions describe how to authenticate calls when using IBM® Cloud Private version 2.1.0.3. If you are using IBM® Cloud Private version 3.1.0, then see [Authenticating API calls on private cloud version 3.1.0](#) instead.

1. Get the service credentials.

- From a new tab in the web browser, go to a URL with the following syntax:

```
https://{{icp-url}}:8443/console/configuration/secrets
```

2. Search for -serviceid-secret

The search result includes a secret with a name that has the following syntax:

```
wcs-{release-name}-serviceid-secret
```

3. Click the secret.

- Click the unlock icon  for api_key.
- Copy the key.

2. Use these credentials in your API call.

- The base URL uses the syntax `http://{{ global.icp.proxyHostname }}/{{ ingress.config.backendService.ingressPath }}`

where the default value for `ingress.config.backendService.ingressPath` is `/assistant/ap1`. For example, `https://mycluster.icp/assistant/api`.

- Provide the API key when you call the service. The following example shows an API key being used.

```
curl -X GET \
-u "apikey:{api_key}" \
'https://myicp.net/assistant/api/v1/workspaces?version=2018-09-20' \
```

To get a workspace ID, go to the **Workspaces** tab of the tool, find the workspace you want to access programmatically, and then from the menu, choose **View details**.

See the [IBM® Cloud Private overview](#) [] for more information about IBM Cloud Private.

Feedback

We value your opinion and want to hear it.

- To share ideas or suggest new features for the IBM Watson™ Assistant service, go to the [IBM Watson Ideas Portal](#).
- To provide feedback about the documentation, click the **FEEDBACK** button that is displayed along the edge of the page you are reading and want to comment about.

Note: The **FEEDBACK** button is not available when you access the documentation site from mobile devices.

Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

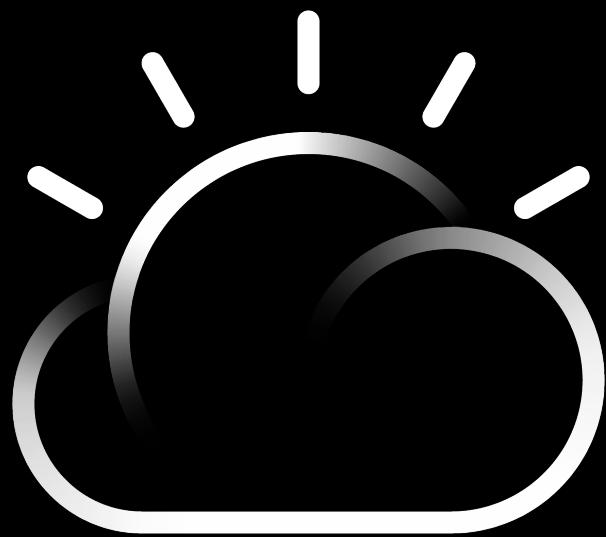
Accessibility features in the product documentation

Accessibility features help people with a physical disability, such as restricted mobility or limited vision, or with other special needs, use information technology products successfully.

The accessibility features in this product documentation allow users to do the following:

- Use screen-reader software and digital speech synthesizers to hear what is displayed on the screen. Consult the product documentation of the assistive technology for details on using assistive technologies with HTML-based information.
- Use screen magnifiers to magnify what is displayed on the screen.
- Operate specific or equivalent features by using only the keyboard.

The documentation content is published in the IBM Cloud Docs site. For information about the accessibility of the site, see [Accessibility features for IBM Cloud](#).



IBM Cloud