



IBM Cloud
Classic Watson Assistant

Product guide

Edition notices

This PDF was created on 2023-10-03 as a supplement to *Classic Watson Assistant* in the IBM Cloud docs. It might not be a complete set of information or the latest version.

© IBM Corp. 2023

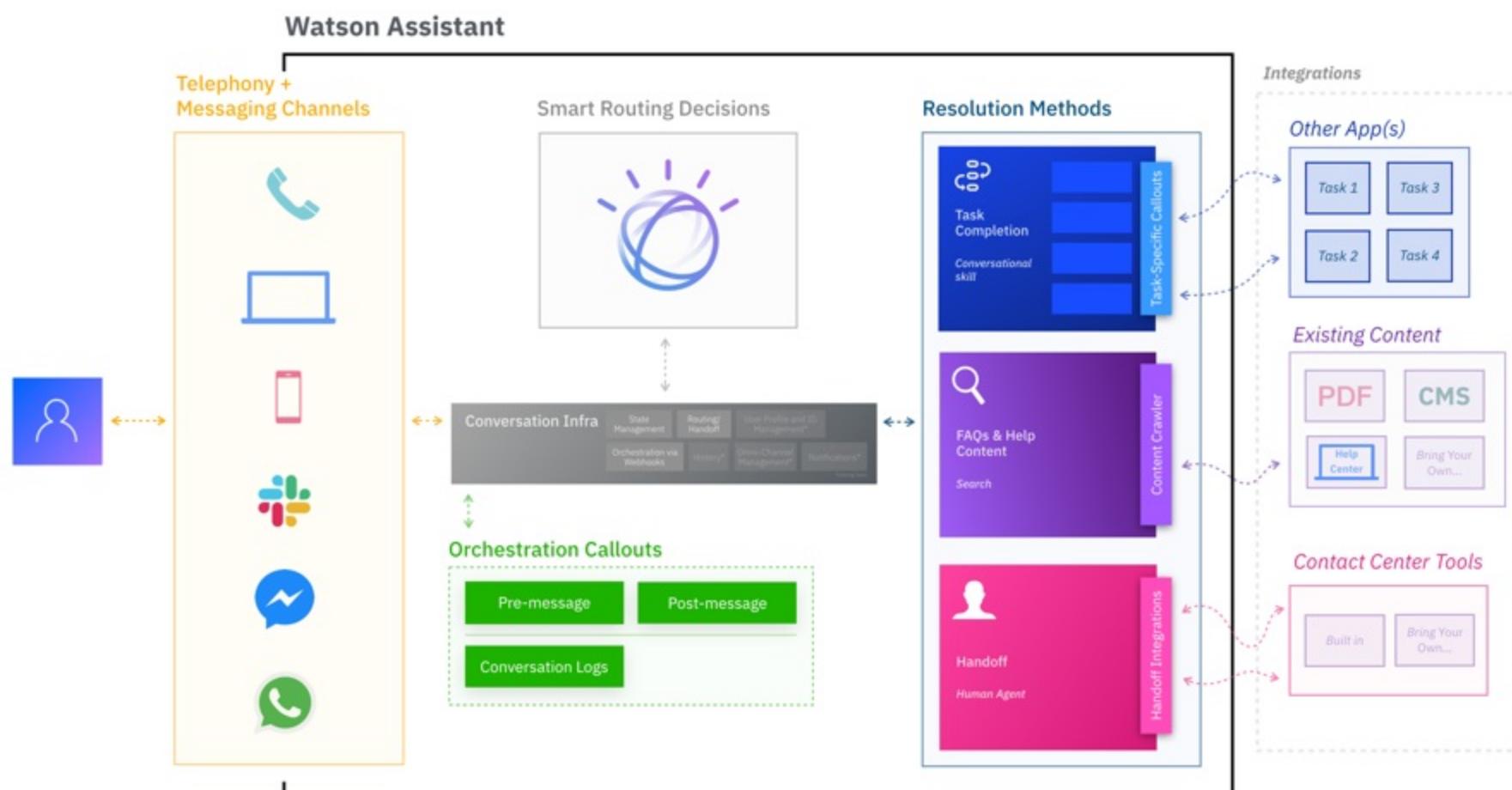
About Watson Assistant

Use IBM Watson® Assistant to build your own branded live chatbot into any device, application, or channel. Your chatbot, which is also known as an *assistant*, connects to the customer engagement resources you already use to deliver an engaging, unified problem-solving experience to your customers.

<i>Create AI-driven conversational flows</i>	Your assistant leverages industry-leading AI capabilities to understand questions that your customers ask in natural language. It uses machine learning models that are custom built from your data to deliver accurate answers in real time.
<i>Embed existing help content</i>	You already know the answers to customer questions? Put your subject matter expertise to work. Add a search skill to give your assistant access to corporate data collections that it can mine for answers.
<i>Connect to your customer service teams</i>	If customers need more help or want to discuss a topic that requires a personal touch, connect them to human agents from your existing service desk provider.
<i>Bring the assistant to your customers, where they are</i>	Configure one or more built-in integrations to quickly publish your assistant on popular social media platforms such as Slack, Facebook Messenger, Intercom, or WhatsApp. Turn the assistant into a member of your customer support call center team, where it can answer the phone and address simple requests so its human teammates can focus on more nuanced customer needs. Make your assistant the go-to help resource for customers by adding it as a chat widget to your company website. If none of the built-in integrations fit your needs, use the APIs to build your own custom app.
<i>Track customer engagement and satisfaction</i>	Use built-in metrics to analyze logs from conversations between customers and your assistant to gauge how well it's doing and identify areas for improvement.

How it works

This diagram illustrates how the product delivers an exceptional, omnichannel customer experience:



- Customers interact with the assistant through one or more of these channels:
 - An existing social media messaging platform, such as Slack, Facebook Messenger, or WhatsApp
 - A phone call or text message
 - A web chat that you embed in your company website and that can transfer complex requests to a customer support representative.
 - A custom application that you develop, such as a mobile app or a robot with a voice interface
- The **assistant** receives a message from a customer and sends it down the appropriate resolution path.

If you want to preprocess incoming messages, this is where you would use webhooks to inject logic that calls an external service that can process the messages before the assistant routes them. Likewise, you can process responses from the assistant before they are returned to the customer.

- The assistant chooses the appropriate resolution from among these options:
 - A **conversational skill** interprets the customer's message further, then directs the flow of the conversation. The skill gathers any information it needs to respond or perform a transaction on the customer's behalf.
 - A **search skill** leverages existing FAQ or other curated content that you own to find relevant answers to customer questions.
 - If a customer wants more personalized help or wants to discuss a sensitive subject, the assistant can connect the customer with someone from your support team through the web chat integration.

For more information about the architecture, read the [How to Make Chatbot Orchestration Easier](#) blog on Medium.com.

To see how Watson Assistant is helping enterprises cut costs and improve customer satisfaction today, read the [Independent study finds IBM Watson Assistant customers can accrue \\$23.9 million in benefits](#) blog on ibm.com.



Note: This documentation describes managed instances of Watson Assistant that are offered in IBM Cloud or in Cloud Pak for Data as a Service. If you are interested in on-premises or installed deployments, see [this documentation](#).

Read more about these implementation steps by following these links:

- [Creating an assistant](#)
- [Adding skills to your assistant](#)
- [Deploying your assistant](#)

Browser support

The Watson Assistant application (where you create assistants and skills) requires the same level of browser software as is required by IBM Cloud. For more information, see IBM Cloud [Prerequisites](#).

For information about the web browsers that are supported by the web chat integration, see [Browser Support](#).

Language support

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

Terms and notices

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

US Health Insurance Portability and Accountability Act (HIPAA) support is available with *Enterprise with Data Isolation* plans that are hosted in the Washington, DC location created on or after 1 April 2019. For more information, see [Enabling HIPAA support for your account](#).

To learn more about service terms and data security, read the following information:

- [Service terms](#) (Search for the Watson Assistant offering)
- [Data Processing and Protection Datasheet](#)
- [Information security](#)
- [IBM Cloud Data security and privacy](#)

Next steps

- [Get started](#) with the product.

Have questions? Contact [IBM Sales](#).

Getting started tutorial

This documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Getting started with Watson Assistant](#). To see all documentation for the new Watson Assistant, please go [Getting started with Watson Assistant](#) for a tutorial series on building in the new experience.

Getting started with a dialog skill in the classic experience

In the classic experience, you build a conversation using a dialog-based approach. If you need to learn that method, see the tutorial [Getting started with a dialog skill](#).

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Getting started with a dialog skill

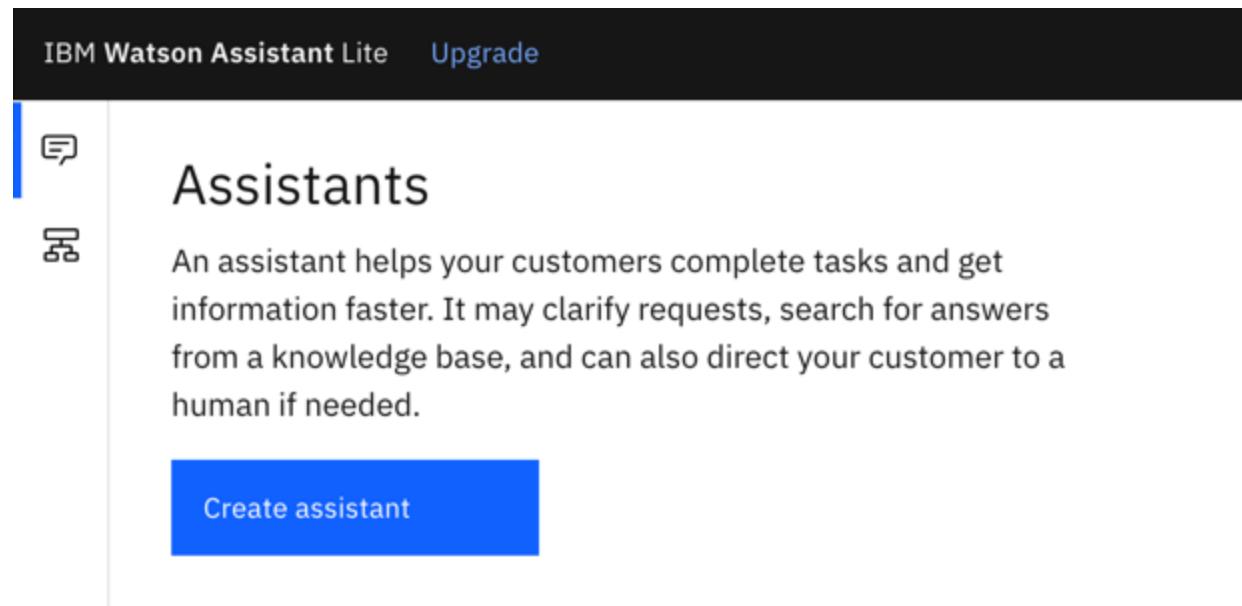
In this short tutorial, we help you use a dialog skill to build your first conversation.

A *dialog skill* uses Watson natural language processing and machine learning technologies to understand user questions and requests, and respond to them with answers that are authored by you.

Step 1: Create an assistant

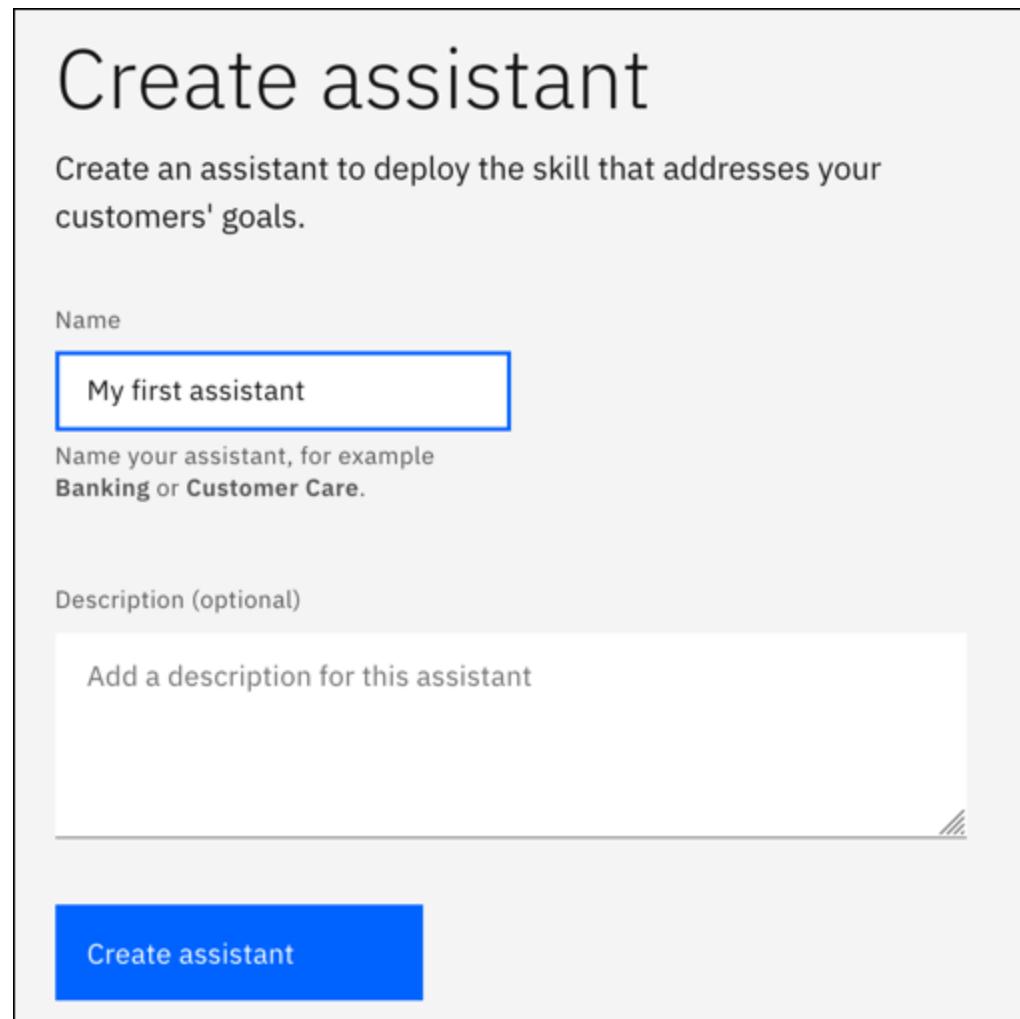
An *assistant* is a cognitive bot to which you add skills that enable it to interact with your customers in useful ways.

1. Click the **Assistants** icon , and then click **Create assistant**.



The screenshot shows the IBM Watson Assistant Lite interface. At the top, there's a dark header bar with the text "IBM Watson Assistant Lite" and "Upgrade". Below the header, on the left, is a sidebar with two icons: a speech bubble for "Assistants" and a document for "Skills". The main content area has a title "Assistants" and a descriptive paragraph: "An assistant helps your customers complete tasks and get information faster. It may clarify requests, search for answers from a knowledge base, and can also direct your customer to a human if needed." At the bottom of this section is a blue "Create assistant" button.

2. Name the assistant **My first assistant**.



The screenshot shows the "Create assistant" dialog box. The title is "Create assistant" and the subtitle is "Create an assistant to deploy the skill that addresses your customers' goals." Below this, there's a "Name" field containing "My first assistant", which is highlighted with a blue border. A note below the name field says "Name your assistant, for example Banking or Customer Care." There's also a "Description (optional)" field with a placeholder "Add a description for this assistant" and a "Create assistant" button at the bottom.

3. Click **Create assistant**.

Step 2: Create a dialog skill

A *dialog skill* is a container for the artifacts that define the flow of a conversation that your assistant can have with your customers.

1. Click **Add an actions or dialog skill**.

The screenshot shows the 'My first assistant' dashboard. In the center, there's a section titled 'Actions Beta or Dialog'. Below it, under 'Build conversations', there's a list of bullet points: 'Actions lets you have an assistant ready to chat in less time, with less effort.', 'Compose step-by-step flows for any range of simple or complex conversations.', and 'Dialog offers a set of full-feature editors that you use to define both your training data and the conversation, with greater control over the logic flow.' A blue button labeled 'Add an actions or dialog skill' is visible on the right side of this section.

2. Give your skill the name **My first skill**.

3. **Optional.** If the dialog you plan to build will use a language other than English, then choose the appropriate language from the list.
4. For skill type, choose Dialog.

The screenshot shows the 'Create an actions or dialog skill' form. At the top, it says 'Create an actions or dialog skill' and 'Add an existing skill or use the sample skill.' There are three tabs: 'Create skill' (which is selected), 'Use sample skill', and 'Upload skill'. Below the tabs, there's a 'Name' field containing 'My first skill'. A note below the field says 'Name your skill; for example, Account application or Personal banking.' There's also a 'Description (optional)' field with placeholder text 'Add a description for this skill'. Under 'Language', it says 'English (US)'. Under 'Skill type', there are two radio buttons: 'Actions' (unchecked) and 'Dialog' (checked). At the bottom is a blue 'Create skill' button.

5. Click **Create skill**.

The skill is created and appears in your assistant.

Step 3: Add intents from a content catalog

The Intents page is where you start to train your assistant. In this tutorial, you will add training data that was built by IBM to your skill. Prebuilt intents are available from the content catalog. You will give your assistant access to the **General** content catalog so your dialog can greet users, and end conversations with them.

1. Make sure your **My first skill** is open.
2. Click **Content Catalog** from the Skills menu.
3. Find **General** in the list, and then click **Add to skill**.

Category	Description	Intents	Action
Banking	Basic transactions for a banking use case.	13	Add to skill +
Bot Control	Functions that allow navigation within a conversation.	9	Add to skill +
Covid-19	Common questions about the Covid-19 virus.	23	Add to skill +
Customer Care	Understand and assist customers with information about themselves and your business.	18	Add to skill +
eCommerce	Payment, billing, and basic management tasks for orders.	14	Add to skill +
General	General conversation topics most users ask.	10	Add to skill +
Insurance	Issues related to insurance policies and claims.	12	Add to skill +

4. 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.

Intent	Description	Examples
#General_About_You	Request generic personal attrib...	20
#General_Agent_Capabilities	Request capabilities of the bot.	30
#General_Connect_to_Agent	Request a human agent.	38
#General_Ending	End the conversation.	37
#General_Greetings	Greet the bot.	27
#General_Human_or_Bot	Ask if speaking to a human or a ...	12
#General_Jokes	Request a joke.	17
#General_Negative_Feedback	Express unfavorable feedback.	20
#General_Positive_Feedback	Express positive sentiment or g...	19
#General_Security_Assurance	Express concerns about the sec...	26

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

Step 4: Build a dialog

A [dialog](#) defines the flow of your conversation in the form of a logic tree. It matches intents (what users say) to responses (what your virtual assistant says back). Each node of the tree has a condition that triggers it, based on user input.

We'll create a simple dialog that handles greeting and ending intents, each with a single node.

Adding a start node

1. From the Skills menu, click **Dialog**.

The following two dialog nodes are created for you automatically:

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

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

4. Click **X** to close the edit view.

You created a dialog node that is triggered by the **welcome** condition. (**welcome** is a special condition that functions like an intent, but does not begin with a **#**.) It is triggered when a new conversation starts. Your node specifies that when a new conversation starts, the system should respond with the welcome message that you add to the response section of this first node.

Testing the start node

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

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

Adding nodes to handle intents

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

1. Click **Add node**.

2. In the node name field, type **Greet customers**.

3. In the **If assistant recognizes** field of this node, start to type **#General_Greetings**. Then, select the **#General_Greetings** option.

4. Add the response text, **Good day to you!**

The screenshot shows the Watson Assistant interface with the title 'My first skill'. On the left, there's a sidebar with options like Intents, Entities, Dialog (which is selected), Options, Analytics, Versions, and Content Catalog. The main area shows a tree structure of nodes. The 'Greet customers' node is selected and expanded. It has three children: 'Welcome welcome', 'Greet customers #General_Greetings', and 'Anything else anything_else'. The 'Greet customers' node is highlighted with a blue border. The 'If assistant recognizes' section contains the entry '#General_Greetings'. The 'Assistant responds' section contains the text 'Good day to you!'. A note at the bottom says 'Response variations are set to sequential. Set to random | multiline'.

5. Click **X** to close the edit view.

6. Click **Add node** to create a peer node.

7. Name the peer node **Say goodbye** and specify **#General_Ending** in the **If assistant recognizes** field.

8. Add **OK. See you later.** as the response text.

The screenshot shows the Watson Assistant interface with the title 'My first skill'. The sidebar shows the 'Dialog' tab is selected. The main area shows a tree structure of nodes. The 'Say goodbye' node is selected and expanded. It has three children: 'Welcome welcome', 'Greet customers #General_Greetings', and 'Say goodbye #General_Ending'. The 'Say goodbye' node is highlighted with a blue border. The 'If assistant recognizes' section contains the entry '#General_Ending'. The 'Assistant responds' section contains the text 'OK. See you later!'. A note at the bottom says 'Response variations are set to sequential. Set to random | multiline'.

9. Click **X** to close the edit view.



My first skill

Intents

[Add node](#)[Add child node](#)[Add folder](#)

Entities

Dialog

Options

Analytics

Versions

Content Catalog

Welcome
welcome

1 Responses / 0 Context Set / Does not return

Greet customers
#General_Greetings

1 Responses / 0 Context Set / Does not return

Say goodbye
#General_Ending

1 Responses / 0 Context Set / Does not return

Anything else
anything_else

1 Responses / 0 Context Set / Does not return

Testing intent recognition

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

[Try it](#)

1. Click the [Try it](#) icon to open the "Try it out" pane. There's that reassuring welcome message.
2. In the text field, type `Hello` and then press Enter. The output indicates that the `#General_Greetings` intent was recognized, and the appropriate response (`Good day to you.`) is displayed.
3. Try the following input:
 - o `bye`
 - o `howdy`
 - o `see ya`
 - o `good morning`
 - o `sayonara`

Your browser does not support the video tag.

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

Result of building a dialog

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

Step 5: Integrate the assistant

Now that you have an assistant that can participate in a simple conversational exchange, test it.

1. Click the **Assistants** icon to open a list of your assistants.
2. Find the *My first assistant* assistant, and open it.
3. Test your assistant with *Preview*.

The screenshot shows the Watson Assistant interface. At the top left is a back arrow labeled 'Assistants'. The title 'My first assistant' is displayed, followed by the subtitle 'Built for you to explore and learn.' To the right of the title are three icons: 'Preview' (highlighted with a blue border), a right-pointing arrow, and a vertical ellipsis. Below the title is a large text area containing the text 'Build your own assistant using IBM Watson Assistant'. Underneath this is a section titled 'Assistant preview' which contains a conversation log. The log shows a user message 'hello' and a response from the assistant 'Good day to you!'. At the bottom of the preview window is a text input field with the placeholder 'Type something...' and a right-pointing arrow icon.

4. Copy the URL from *Share this link* and use it in a new tab. You can start submitting message to see how your assistant responds.



Note: With a Lite plan, you can use the service for free. With other plans, you are charged for messages that you submit from the preview link. You can review metrics about the test user conversations from the Analytics page. You are not charged for messages that you submit from the "Try it out" pane, and the exchanges you have there are not logged.

5. Type `hello` into the text field, and watch your assistant respond.

The screenshot shows the Watson Assistant interface. At the top left is a back arrow labeled 'Assistants'. The title 'My first assistant' is displayed, followed by the subtitle 'Built for you to explore and learn.' To the right of the title are three icons: 'Preview' (highlighted with a blue border), a right-pointing arrow, and a vertical ellipsis. Below the title is a large text area containing the text 'Build your own assistant using IBM Watson Assistant'. Underneath this is a section titled 'Assistant preview' which contains a conversation log. The log shows a user message 'hello' and a response from the assistant 'Good day to you!'. At the bottom of the preview window is a text input field with the placeholder 'Type something...' and a right-pointing arrow icon.

You can share the URL with others who might want to try out your assistant.

6. After testing, close the web page. Click the **X** to close the preview link integration page.

Next steps

This tutorial is built around a simple example. For a real application, you need to define some more interesting intents, some entities, and a more complex dialog that uses them both. When you have a polished version of the assistant, you can integrate it with web sites or channels, such as Slack, that your customers already use. As traffic increases between the assistant and your customers, you can use the tools that are provided in the **Analytics** page to analyze real conversations, and identify areas for improvement.

- Complete follow-on tutorials that build more advanced dialogs:
 - Add more dialog nodes to design complex conversational exchanges. See [Building a complex dialog](#).
 - Learn techniques for getting customers to share information that the assistant needs before it can provide a useful response. See [Adding a node with slots](#).



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Plan your assistant

Build an assistant that meets the needs of your customers with speed and ease.

To start, consider these factors and make key decisions up front that will keep you on track as you build.

Pick your assistant's areas of expertise

Decide what you want your assistant to specialize in. What questions or tasks will it help customers with? To make an informed decision, review any support call logs that you have access to.

Start small. Pick one or a handful of customer issues that will deliver the highest value to start.

It might be valuable for your assistant to answer a simple question that is asked all the time. Or maybe there's a task, such as scheduling appointments, that can be offloaded to the assistant to tackle 30% of all incoming customer requests.

Choose a narrow set of user goals first. After your assistant is live, you can use built-in tools to gain insights from the incoming traffic that will tell you what areas to focus on next.

Give the right type of answer to meet the need

A conversational exchange is what your assistant does best, but your assistant can do other things too. The best response to a question might be a single answer with a link somewhere else. Think about the right way to answer customer questions; don't try to fit everything into one type of conversational exchange.

The following table lists some examples.

Customer need	Best type of response
Get information about your store location	Your assistant answers with text (the store address) and an image (an area map).
Activate a credit card	Your assistant can use a conversational flow to collect information for identity verification, and then call a webhook to submit the request to activate the card on the user's behalf.
Complete a simple task that involves a complicated application	Your assistant can link them to a 2-minute video that illustrates how to complete the task.
Learn about insurance plan details after the death of a loved one	Your assistant can connect the customer directly to a person who can show empathy and patience as the matter is addressed.
Solve a problem that requires a long and involved procedure to fix	Instead of trying to walk the customer through the procedure step by step in conversation, your assistant can link to a help center that documents the full procedure in detail.
The customer calls support and your assistant answers	Let's say the assistant needs a lot of details from the customer before it can help. Instead of trying to prompt the customer for each piece of information and transcribe it properly, your assistant can switch to SMS text messaging. After years of interacting with bad interactive voice response systems, many customers are more likely to yell Agent over and over than to engage in a long exchange. But if you give them a chance to explain something in writing, they tend to do so willingly.

Example of optimal response types

How will customers find your assistant?

Deploy your assistant where customers can find it with ease. For example, you can embed the assistant in your company website or add it to a messaging platform such as Facebook, Slack, or WhatsApp.

While the conversational skill is defined in text, customers can talk with your assistant over the phone when you deploy it by using the phone integration.

For more information about ways to deploy the assistant, see [Adding integrations](#).

Understanding where you will deploy the assistant before you begin can help you author the right types of answers for a given channel or platform.

What languages does your assistant speak?

Decide whether and how you want to handle more than one spoken language. For more information about ways to approach language support, see [Adding support for global audiences](#).

Does your assistant see the glass as half full?

Is your assistant an optimist or a pessimist, a shy intellectual type, or an upbeat sidekick? Choose a personality for your assistant, and then write conversations that reflect that personality. Don't overdo it. Don't sacrifice usability for the sake of keeping your assistant in character. But strive to present a consistent tone and attitude.

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. In fact, some US states have laws that require chat bots to identify themselves as chat bots.

Who will build your assistant?

Assemble a team with people who understand your customers and their needs, people who know how to interact with customers to reach the best outcomes. These subject matter experts can focus on designing an engaging conversational flow. In fact, the actions skill is designed with this type of expert in mind. The team can simultaneously build a conversational flow by defining discrete actions.

If you have data scientists or team members with programming skills, you can take advantage of some advanced capabilities that require varying levels of development expertise. This set of users might prefer to build the conversational flow with a dialog skill because there is greater visibility into the individual components that make up the training data.

So, which conversational skill type *should* you use?

Use both. Leverage advanced capabilities that are available from a dialog skill and build individual actions to perform finite tasks that you want to support. You can call the actions skill from your dialog skill.

For more information, see [Choosing a conversational skill](#).

Do you have existing content to leverage?

The answer to many a common question is already documented somewhere in your organization's technical information collateral. If only you could find it!

Give your assistant access to this information by adding a search skill to your assistant. The search skill uses Discovery to return smart answers to natural language questions.

Expand your assistant's responsibilities

If you start small, and choose the goals with the highest impact first, you'll have room and time to grow the expertise of your assistant. The built-in metrics of active user conversations help you understand what your customers are asking about and how well your assistant is able to meet their needs.

Nothing beats real customer data. It will tell you what areas to tackle next.

Ready to start building?

See [Creating an assistant](#) to get started.

Release notes

Release notes for classic Watson Assistant

Release notes for the classic Watson Assistant experience have moved

As of 22 May 2023, these release notes are no longer being updated. All release notes for both the new Watson Assistant and the classic Watson Assistant can be found here: [Release notes for Watson Assistant](#).

22 May 2023

Changes to the date and number formats in assistant responses

Beginning on 22 May 2023, customers might see changes to the date and number formats in assistant responses.

Examples of date changes include removed or added periods, such as:

- In Spanish, **18 abr. 2021** changes to **18 abr 2021**
- In Portuguese, **18 de abr** changes to **18 de abr.**

The delimiter character changes for numbers in some languages. For example, in French, nonbreaking space (NBSP) changes to narrow no-break space (NNBSP).

These changes are the result of migrating the Watson Assistant platform to Java 17, where locale values are updated by using specifications in [CLDR 39](#).

To avoid or minimize the impact of similar changes in the future, you can use [Actions display formats](#).

18 May 2023

Differences in contextual entity detection for dialog skills with few annotations

If you have 10 to 20 examples of contextual entities in your dialog skill, you might see differences in the entities detected due to updates made to address critical vulnerabilities. The impact of these differences is limited to only newly-trained models. Existing models are unaffected. You can mitigate these differences by annotating more examples. For more information, see [Annotation-based method](#).

15 May 2023

Change to dialog skill context variables named **request**

If your dialog skill used a context variable that is named **request**, it was removed from the response payload of any **/message** calls in the V1 or V2 API, or through the Watson Assistant user interface. After 15 May 2023, this behavior changes. Watson Assistant doesn't remove context variables that are named **request** from the response payload anymore.

3 May 2023

Algorithm version **Beta** provides improved intent detection and action matching

The algorithm version **Beta** now provides improved intent detection and action matching. It includes a new foundation model that is trained using a transformer architecture to improve intent detection and action matching for English.

Improvements include:

- Improved robustness to variations in user inputs such as typos and different inflection forms
- Less training data required to reach the same level of performance compared to previous algorithms

For more information, see [Algorithm version and training](#).

16 April 2023

Autolearning beta for dialog skills removed

As of this release, the *autolearning* beta has been removed from the **Analytics** section in dialog skills. In the new experience, you can test a new and improved autolearning beta. For more information, see [Using autolearning to improve assistant responses](#).

16 March 2023

New algorithm version **Latest (20 Dec 2022)** provides improved irrelevance detection

A new algorithm version is available. The **Latest (20 Dec 2022)** version includes a new irrelevance detection implementation to improve off-topic detection accuracy.

Improvements include:

- Relevant user inputs are expected to get higher confidence, so they are less likely to be considered irrelevant or require clarification
- Irrelevance detection is improved in the presence of direct entity references
- Irrelevance detection is more stable across small changes to input
- Intent detection is more stable regarding occurrence of numerics, such as postal codes
- For German-language assistants, intent detection is more robust in the presence of umlauts

This algorithm was first introduced as the **Beta** version in June 2022. Since then, support for more languages has been added. This algorithm version was stabilized in December 2022 with minor enhancements since that time.

With this new release, the June 1, 2022 version is now labeled as **Previous (01 Jun 2022)**. The oldest release labeled as **01 Jan 2022** is no longer available for training. As of now, the new **Beta** version has the same behavior as the **Latest (20 Dec 2022)** version. Updates to the **Beta** version will be released soon.

For more information, see [Algorithm version and training](#).

6 March 2023

Improvements to algorithm version beta

Improvements to the current *Beta* algorithm version include:

- Relevant examples are expected to get higher confidence
- For Spanish-language assistants, intent detection is improved in the presence of direct entity references
- Intent detection is more stable regarding occurrence of numerics, such as postal codes
- Intent detection now accounts for fuzzy closed entity mentions
- For German-language assistants, intent detection is more robust in the presence of umlauts

For more information, see [Algorithm version](#).

26 January 2023

Intent recommendations and intent user example recommendations discontinued

As of this release, intent recommendations and intent user example recommendations are discontinued. **Intent recommendations** has been removed from the **Intents** page and **Recommended examples** has been removed from intents. In the new Watson Assistant experience, you can [use unrecognized requests to get action recommendations](#).

18 January 2023

Algorithm version stability improvement

As of this date, the **Latest (01 Jun 2022)** and **Beta** algorithm versions now have more stable behavior across retrained models, in the presence of overlapping entities (the same entity value belonging to more than one entity type). Previously, when there were overlapping entities definitions, confidences could differ across different retraining. With this improvement, you can expect to see similar confidences. For more information, see [Algorithm version](#).

11 January 2023

Algorithm version 01-Jun-2022 uses enhanced intent detection by default

As of this date, the algorithm version **Latest (01-Jun-2022)** now uses enhanced intent detection by default. Before this change, some skills that did not include a specific algorithm version selection inadvertently used **Previous (01-Jan-2022)**. You can notice small changes in intent detection behavior when changes are made to an assistant that previously didn't have enhanced intent detection enabled. For more information, see [Algorithm version](#).

6 December 2022

`String.toJson()` expression method

The new [`String.toJson\(\)`](#) method parses a string containing JSON data and returns a JSON object or array.

12 October 2022

`now(String timezone)` method output includes time zone offset

The string returned from the `now(String timezone)` method now includes the time zone offset (such as `-05:00`). The new format is `yyyy-MM-dd HH:mm:ss 'GMT'XXX` (where `XXX` represents the time zone offset). This change enables accurate time zone computations when used with other date and time methods such as `before`, `after`, and `reformatDateTime`.

If you have an existing action or dialog that depends on the previous format, you can adapt it by reformatting the output using `now(timezone).reformatDateTime('yyyy-MM-dd HH:mm:ss')`.

For more information, see [Expression language methods]((<https://cloud.ibm.com/docs/assistant?topic=assistant-dialog-methods#dialog-methods-date-time>)).

12 September 2022

Fix for fuzzy matching in German

In some cases, German closed entities were incorrectly matching shorter values over longer values. For example, suppose that you defined two entity values, `Pflege` and `Pflegegeld` in one entity. If a customer accidentally input `Pflegegelb`, the assistant would incorrectly match with `Pflege` rather than `Pflegegeld`.

With this fix, the `Pflegegelb` input value would correctly match `Pflegegeld`, not `Pflege`. For more information, see [How fuzzy matching works](#).

11 August 2022

Algorithm version options available in more languages

Algorithm version options are now available in Arabic, Czech, and Dutch. This allows you to choose which Watson Assistant algorithm to apply to your future trainings. For more information, see [Algorithm version](#).

Improved enhanced intent detection available in more languages

Previously, the exact match in enhanced intent detection was improved to better handle small differences between training examples and runtime utterances when the differences do not change the meaning of a sentence. For example, suppose in your training examples, `covid-19` is in the `#covid` intent and `@doctortype_facilitytype around Palm Beach` is in the `#find_provide_master` intent. In this example, the `@doctortype_facilitytype` direct entity reference contains entity values, including `hospital`. At run time, `covid19` is predicted as 100% confident for the `#covid` intent, and `hospital around palm beach` is predicted as 100% confident for the `#find_provide_master` intent.

This update now includes the following languages: Arabic, Czech, and Dutch. For more information, see [Accessing intents](#).

24 June 2022

Algorithm version options available in more languages

Algorithm version options are now available in Chinese (Traditional), Japanese, and Korean. This allows you to choose which Watson Assistant algorithm to apply to your future trainings. For more information, see [Algorithm version](#).

Improved enhanced intent detection available in more languages

Previously, the exact match in enhanced intent detection was improved to better handle small differences between training examples and runtime utterances when the differences do not change the meaning of a sentence. For example, suppose in your training examples, `covid-19` is in the `#covid` intent and `@doctortype_facilitytype around Palm Beach` is in the `#find_provide_master` intent. In this example, the `@doctortype_facilitytype` direct entity reference contains entity values, including `hospital`. At run time, `covid19` is predicted as 100% confident for the `#covid` intent, and `hospital around palm beach` is predicted as 100% confident for the `#find_provide_master` intent.

This update now includes the following languages: Chinese (Simplified), Chinese (Traditional), German, Japanese, Korean, and Portuguese. For more information, see [Accessing intents](#).

16 June 2022

Algorithm version

Algorithm version allows you to choose which Watson Assistant algorithm to apply to your future trainings. In the **Options** section of a dialog skill, **Algorithm Version** replaces **Intent Detection**. For more information, see [Algorithm version](#).

Algorithm beta version (2022-06-10)

Algorithm beta version (2022-06-10) includes a new irrelevance detection algorithm to improve off-topic detection accuracy. Utterances with similar meanings are expected to have more similar confidences in comparison to previous irrelevance detection algorithms. For example, in the Customer Care Sample Skill, the training utterance `please suggest route from times square` has 100% confidence at runtime. Currently in IBM Cloud, the utterance `please suggest route from central park` gets a low confidence and could be flagged as irrelevant. With beta version (2022-06-10), the same utterance is expected to be predicted correctly as #Customer_Care_Store_Location with a ~46% confidence.

19 May 2022

Sign out due to inactivity setting

Watson Assistant now uses the **Sign out due to inactivity setting** from Identity & Access Management (IAM). IBM Cloud account owners can select the time it takes before an inactive user is signed out and their credentials are required again. The default is 2 hours.

An inactive user will see two messages. The first message alerts them about an upcoming session expiration and provides a choice to renew. If they remain inactive, a second session expiration message appears and they will need to log in again.

For more information, see [Setting the sign out due to inactivity duration](#).

18 May 2022

Language support improvements

Entity recognition and intent classification for Japanese and Korean languages changed to improve the reliability of Watson Assistant. You might see minor differences in how Watson Assistant handles entity recognition and intent classification. This change affects {{site.data.keassistant_classic_shortnshort}} workspaces that are trained after May 18, 2022. Workspaces that were trained before May 18, 2022 maintain the same behavior.

Any visible changes are most likely to be seen in dictionary-based or pattern-based entity matching. For more information about defining entities, see [Defining information to look for in customer input](#). As a suggested practice, you can test your dialog skill with your current test framework to determine whether your workspace is impacted before you update your production workspace.

If entity values or synonyms that previously matched no longer match, you can update the entity and add a synonym with white space between the tokens, for example:

- Japanese: Add “お〇” as a synonym for “〇〇”
- Korean: Add “〇〇〇” as a synonym for “〇〇〇”

Enhanced intent detection available for Portuguese, German, and Simplified Chinese

Enhanced intent detection is now available for Portuguese, German, and Simplified Chinese. The enhanced intent detection model improves your assistant's ability to understand what customers want.

28 April 2022

Assistant preview link can be disabled

Assistant preview now includes a toggle to disable the preview link. This allows you to stop access to the preview link if necessary. For more information, see [Using the assistant preview to test your assistant](#).

5 April 2022

Dialog feature available in the new Watson Assistant

The dialog feature is available in the new Watson Assistant experience. If you have a dialog-based assistant that was built using the classic Watson Assistant, you can now migrate your dialog skill to the new Watson Assistant experience. For more information, see [Migrating to the new experience](#).

25 March 2022

Improved irrelevance detection for Dutch

Irrelevance detection for Dutch disregards any punctuation in an input sentence. For example, you can now expect the same confidence score for the following two inputs: `ik ben een kleine krijger?` and `ik ben een kleine krijger`. In this example, the question mark (?) doesn't affect the confidence score.

Improved enhanced intent detection

The exact match in enhanced intent detection now better handles small differences between training examples and runtime utterances when the differences do not change the meaning of a sentence.

For example, suppose in your training examples, `covid-19` is in the `#covid` intent and `@doctortype_facilitytype around Palm Beach` is in the `#find_provide_master` intent. In this example, the `@doctortype_facilitytype` direct entity reference contains entity values, including `hospital`. At run time, `covid19` is predicted as 100% confident for the `#covid` intent, and `hospital around palm beach` is predicted as 100% confident for the `#find_provide_master` intent.

This update applies to the following languages: English, French, Spanish, Italian, and the universal language model. For more information, see [Accessing intents](#).

23 March 2022

Fuzzy matching updates

Previously, an update was made so that interactions between the stemming and misspelling fuzzy matching features were not allowed. This change applied to the following languages: English, French, German, and Czech. This was updated so that this change applies only to the English language. For more information, see [How fuzzy matching works](#).

14 March 2022

Closed entity matching with accent-normalized values in French

Closed entities exact matches in French are completed using accent-normalized values or synonyms. For example, if you define a closed entity with a value or synonym with accent marks (for example, garçon or déjà), then variants without accent marks are also recognized (garcon or deja). Likewise, if a closed entity value or synonym is defined without accent marks, then user inputs with accent marks are also recognized. For more information about defining entities, see [Defining information to look for in customer input](#).

Pattern entities do not prevent spelling autocorrection

Pattern entities that match all characters and words that are usually used to count input words do not prevent spelling autocorrection. For example, if a customer defines the `^..{0,19}$` pattern entity that matches the first 20 characters of an input, then the entity match does not affect spelling autocorrection. In this example, an input of `cancl transaction` is autocorrected to `cancel transaction`.

This change applies to the following languages: English and French. For more information, see [Correcting user input](#).

1 March 2022

Enhanced irrelevance detection update

We revised the enhanced irrelevance detection classification algorithm. Now, enhanced irrelevant detection uses any provided counterexamples in training. This change does not affect workspaces without counterexamples. This update applies to the following languages: English, French, Spanish, Italian, and the universal language model. For more information, see [Defining what's irrelevant](#).

9 February 2022

All instances now default to new experience

All new instances of Watson Assistant now direct users to the new product experience by default.

Watson Assistant has been completely overhauled to simplify the end-to-end process of building and deploying a virtual assistant, reducing time to launch and enabling nontechnical authors to create virtual assistants without involving developers. For more information about the new Watson Assistant, and instructions for switching between the new and old experiences, see [Welcome to the new Watson Assistant](#).

If you would like to send us feedback on the new experience, please use [this form](#).

4 February 2022

Fuzzy matching updates

Interactions between the stemming and misspelling fuzzy matching features are not allowed. Improve fuzzy matching behavior by limiting the

interactions between different fuzzy matching features. This change applies to the following languages: English, French, German, and Czech. For more information, see [How fuzzy matching works](#).

13 January 2022

New setting for options customer response type

In actions, a new **List options** setting allows you to enable or disable the options customer response from appearing in a list. This can be useful to prevent a phone integration from reading a long list of options to the customer. As part of this change, all customer response types now have a **Settings** icon. **Allow skipping** has moved from **Edit Response** and is now found in the new settings.

24 December 2021

Apache Log4j security vulnerability updates

Watson Assistant upgraded to using Log4j version 2.17.0, which addresses all of the Critical severity and High severity Log4j CVEs, specifically CVE-2021-45105, CVE-2021-45046, and CVE-2021-44228.

10 December 2021

Channel transfer in **Phone** integration

The phone integration now supports the `channel_transfer` response type. For more information, see [Handling phone interactions](#).

3 December 2021

Configure webhook timeout

From the **Pre-message webhook** and **Post-message webhook** configuration pages, you can configure the webhook timeout length from a minimum of 1 second to a maximum of 30 seconds. For more information, see [Webhook overview](#).

27 November 2021

New API version

The current API version is now **2021-11-27**. This version introduces the following changes:

- The `output.text` object is no longer returned in `message` responses. All responses, including text responses, are returned only in the `output.generic` array.

9 November 2021

New phone response types

New response types are available for controlling the configuration and behavior of the phone integration. These response types replace most of the older `vgw` actions, which are now deprecated. (The `vgw` actions will continue to work, so existing skills do not need to be changed.) For more information, see [Handling phone interactions](#).

Rich response types

Your assistant can now send responses that include elements such as audio, video, or embedded `iframe` content. For more information, see [Rich responses](#).

4 November 2021

Actions enhancement: Add variables to links

In an actions skill, when including a link in an assistant response, you can now access and use variables. In the URL field for a link, type a dollar sign (\$) character to see a list of variables to choose from.

14 October 2021

`vgwHangUp` message no longer sent

Previously, the phone integration sent a message containing the text `vgwHangUp` to the assistant when a call was ended by the caller. This message is no longer sent.

7 October 2021

The new Watson Assistant

The new Watson Assistant is now available! This new experience, focused on using **actions** to build customer conversations, is designed to make it simple enough for *anyone* to build a virtual assistant. Building, testing, publishing, and analyzing your assistant can all now be done in one simple and intuitive interface.

- New **navigation** provides a workflow for building, previewing, publishing, and analyzing your assistant.
- Each assistant has a **home page** with a task list to help you get started.
- Build conversations with **actions**, which represent the tasks you want your assistant to help your customers with. Each action contains a series of steps that represent individual exchanges with a customer.
- A new way to **publish** lets you review and debug your work in a draft environment before going live to your customers.
- Use a new suite of **analytics** to improve your assistant. Review which actions are being completed to see what your customers want help with, determine if your assistant understands and addresses customer needs, and decide how can you make your assistant better.
- New [Top intents and top entities](#)

16 September 2021

Enhanced intent detection for French, Italian, and Spanish dialog skills

The new intent detection model improves your assistant's ability to understand what customers want. This model is now available in dialog skills using French, Italian, and Spanish.

Change to the irrelevance detection option

As of this release, new English dialog skills no longer include the option to choose between the **Enhanced** or **Existing** irrelevance detection. By default, intent detection and irrelevance detection are paired like this:

- If you use the dialog skill options to choose enhanced intent detection, it is automatically paired with enhanced irrelevance detection.
- If you use the dialog skill options to choose existing intent detection, it is automatically paired with existing irrelevance detection.

If necessary, you can use the [Update workspace API](#) to set your English-language assistant to one of the four combinations of intent and irrelevance detection:

- Enhanced intent recognition and enhanced irrelevance detection
- Enhanced intent recognition and existing irrelevance detection

- Existing intent recognition and enhanced irrelevance detection
- Existing intent recognition and existing irrelevance detection

For French, Italian, and Spanish, you can use the API to set your assistant to these combinations:

- Enhanced intent recognition and enhanced irrelevance detection
- Existing intent recognition and existing irrelevance detection

15 September 2021

Dialog skill "Try it out" improvements

The **Try it out** pane now includes these changes:

- It now includes runtime warnings in addition to runtime errors.
- For dialog skills, the **Try it out** pane now uses the [React](#) UI framework similar to the rest of the Watson Assistant user interface. You shouldn't see any change in behavior or functionality. As a part of the update, dialog skill error handling has been improved within the "Try it out" pane. This update was enabled on these dates:
 - September 9, 2021 in the Tokyo and Seoul data centers
 - September 13, 2021 in the London, Sydney, and Washington, D.C. data centers
 - September 15, 2021 in the Dallas and Frankfurt data centers

13 September 2021

Dialog skill "Try it out" improvements

For dialog skills, the **Try it out** pane now uses the [React](#) UI framework similar to the rest of the Watson Assistant user interface. You shouldn't see any change in behavior or functionality. As a part of the update, dialog skill error handling has been improved within the "Try it out" pane. This update was enabled on September 9, 2021 in the Tokyo and Seoul data centers. On September 13, 2021, the update was enabled in the London, Sydney, and Washington, D.C. data centers.

Disambiguation feature updates

The dialog skill disambiguation feature now includes improved features:

- **Increased control:** The frequency and depth of disambiguation can now be controlled by using the **sensitivity** parameter in the [workspace API](#). There are 5 levels of sensitivity:
 - `high`
 - `medium_high`
 - `medium`
 - `medium_low`
 - `low`

The default (`auto`) is `medium_high` if this option is not set.

- **More predictable:** The new disambiguation feature is more stable and predictable. The choices shown may sometimes vary slightly to enable learning and analytics, but the order and depth of disambiguation is largely stable.

These new features may affect various metrics, such as disambiguation rate and click rates, as well as influence conversation-level key performance indicators such as containment.

If the new disambiguation algorithm works differently than expected for your assistant, you can adjust it using the sensitivity parameter in the update workspace API. For more information, see [Update workspace](#).

9 September 2021

Actions skill improvements

Actions skills now include these new features:

- **Change conversation topic:** In general, an action is designed to lead a customer through a particular process without any interruptions. In real life, however, conversations almost never follow such a simple flow. In the middle of a conversation, customers might get distracted, ask questions about related issues, misunderstand something, or just change their minds about what they want to do. The **Change conversation topic** feature enables your assistant to handle these digressions, dynamically responding to the user by changing the conversation topic as needed.
- **Fallback action:** The built-in action, *Fallback*, provides a way to automatically connect customers to a human agent if they need more help. This action helps you to handle errors in the conversation, and is triggered by these conditions:
 - Step validation failed: The customer repeatedly gave answers that were not valid for the expected customer response type.
 - Agent requested: The customer directly asked to be connected to a human agent.
 - No action matches: The customer repeatedly made requests or asked questions that the assistant did not understand.

Dialog skill "Try it out" improvements

For dialog skills, the **Try it out** pane now uses the [React](#) UI framework similar to the rest of the Watson Assistant user interface. You shouldn't see any change in behavior or functionality. As a part of the update, dialog skill error handling has been improved within the "Try it out" pane. This update will be implemented incrementally, starting with service instances in the Tokyo and Seoul data centers.

2 September 2021

Deploy your assistant on the phone in minutes

We have partnered with [IntelePeer](#) to enable you to generate a phone number for free within the phone integration. Simply choose to generate a free number when following the prompts to create a phone integration, finish the setup, and a number is assigned to your assistant. These numbers are robust and ready for production.

Connect to your existing service desks

We have added step-by-step documentation for connecting to [Genesys](#) and [Twilio Flex](#) over the phone. Easily hand off to your live agents when your customers require telephony support from your service team. Watson Assistant deploys on the phone via SIP, so most phone based service desks can easily be integrated via SIP trunking standards.

23 August 2021

Intent detection updates

Intent detection for the English language has been updated with the addition of new word-piece algorithms. These algorithms improve tolerance for out-of-vocabulary words and misspelling. This change affects only English-language assistants, and only if the enhanced intent recognition model is enabled.

Automatic retraining of old skills and workspaces

As of August 23, 2021, Watson Assistant enabled automatic retraining of existing skills in order to take advantage of updated algorithms. The {{site.data.keassistant_classic_shortnshort}} service will continually monitor all ML models, and will automatically retrain those models that have not been retrained within the previous 6 months. For more information, see [Automatic retraining of old skills and workspaces](#).

19 August 2021

Actions preview now includes debug mode and variable values

When previewing your actions, you can use **debug mode** and **variable values** to ensure your assistant is working the way you expect.

Debug mode allows you to go to the corresponding step by clicking on a step locator next to each message. It shows you the confidence score of top three possible action when the input triggers an action. You can also follow the step in the action editor along with the conversation flow.

Variable values shows you a list of the variables and their values of current action and the session variables. You can check and edit variables during

the conversation flow.

17 August 2021

New service desk support reference implementation

You can use the reference implementation details to integrate the web chat with the Oracle B2C Service service desk. For more information, see [Adding service desk support](#).

29 July 2021

Salesforce and Zendesk deployment changes

The Salesforce and Zendesk integrations have been updated to use the [new chat history widget](#). The updated deployment process applies to all new deployments, including any redeployments of existing Salesforce and Zendesk connections. However, existing deployments are not affected and do not need to be modified or redeployed at this time.

Fallback value for session variables

In action skills, you can now set a fallback value for session variables. This feature lets you define a value for a session variable if a user-defined value isn't found.

16 July 2021

Logging API changes

The internal storage and processing of logs has changed. Some undocumented fields or filters might no longer be available. (Undocumented features are not officially supported and might change without notice.)

New API version

The current API version (v1 and v2) is now **2021-06-14**. The following changes were made with this version:

- The `metadata` property of entities detected at run time is deprecated. For detailed information about detected system entities, see the `interpretation` property.
- The data types of certain entity mentions are no longer automatically converted:
 - Numbers in scientific notation (such as `1E10`), which were previously converted to numbers
 - Boolean values (such as `false`), which were previously converted to booleans

These values are now returned as strings.

17 June 2021

Actions skill now generally available

As of this release, the beta program has ended, and actions skills are available for general use.

An actions skill contains actions that represent the tasks you want your assistant to help your customers with. Each action contains a series of steps that represent individual exchanges with a customer. Building the conversation that your assistant has with your customers is fundamentally about deciding which steps, or which user interactions, are required to complete an action. After you identify the list of steps, you can then focus on writing engaging content to turn each interaction into a positive experience for your customer.

Date and time response types

New to action skills, these response types allow you to collect date and time information from customers as they answer questions or make requests.

New built-in variables

Two kinds of built-in variables are now available for action skills.

- **Set by assistant** variables include the common and essential variables `Now`, `Current time`, and `Current date`.
- **Set by integration** variables are `Timezone` and `Locale` and are available to use when connected to a webhook or integration.

Universal language model now generally available

You now can build an assistant in any language you want to support. If a dedicated language model is not available for your target language, create a skill that uses the universal language model. The universal model applies a set of shared linguistic characteristics and rules from multiple languages as a starting point. It then learns from training data written in the target language that you add to it. For more information, see [Understanding the universal language model](#).

3 June 2021

Log webhook support for actions and search skills

The log webhook now supports messages exchanged with actions skills and search skills, in addition to dialog skills. For more information, see [Logging activity with a webhook](#).

27 May 2021

Change to conversation skill choices

When adding skills to new or existing assistant, the conversation skill choices have been combined, so that you pick from either an actions skill or a dialog skill.

With this change:

- New assistants can use up to two skills, either actions and search or dialog and search. Previously, new assistants could use up to three skills: actions, dialog, and search.
- Existing assistants that already use an actions skill and a dialog skill together can continue to use both.
- The ability to use actions and dialog skills together in a new assistant is planned for 2H 2021.

20 May 2021

Actions skill improvement

Actions now include a new choice, **Go to another action**, for what to do next in a step. Also called a subaction, this feature lets you call one action from another action, to switch the conversation flow to another action to perform a certain task. If you have a portion of an action that can be applied across multiple use cases you can build it once and call to it from each action. This new option is available in the **And then** section of each step.

21 April 2021

Preview button for testing your assistant

For testing your assistant, the new Preview button replaces the previous Preview tile in Integrations.

New checklist with steps to go live

Each assistant includes a checklist that you can use to ensure you're ready to go live.

Actions skill improvement

Actions now include currency and percentage response types.

Learn what's new

The *What's new* choice on the help menu opens a list of highlighting recent features.

14 April 2021

Actions skill improvement

Actions now include a free text response type, allowing you to capture special instructions or requests that a customer wants to pass along.

8 April 2021

Deploy your assistant to WhatsApp - now generally available

Make your assistant available through WhatsApp messaging so it can exchange messages with your customers where they are. This integration, which is now generally available, creates a connection between your assistant and WhatsApp by using Twilio as a provider. For more information, see [Integrating with WhatsApp](#).

Web chat home screen now generally available

Ease your customers into the conversation by adding a home screen to your web chat window. The home screen greets your customers and shows conversation starter messages that customers can click to easily start chatting with the assistant. For more information about the home screen feature, see [Configuring the home screen](#). The home screen feature is now enabled by default for all new web chat deployments. Also, you can now access context variables from the home screen. Note that initial context must be set using a `conversation_start` node. For more information, see [Starting the conversation](#).

Connect to human agent response type allows more text

In a dialog skill, the response type `Connect to human agent` now allows 320 characters in the `Response when agents are online` and `Response when no agents are online` fields. The previous limit was 100 characters.

Legacy system entities deprecated

In January 2020, a new version of the system entities was introduced. As of April 2021, only the new version of the system entities is supported for all languages. The option to switch to using the legacy version is no longer available.

6 April 2021

Service API endpoint change

As explained in [December 2019](#), as part of work done to fully support IAM authentication, the endpoint you use to access your Watson Assistant service programmatically is changing. The old endpoint URLs are deprecated and **will be retired on 26 May 2021**. Update your API calls to use the new URLs.

The pattern for the endpoint URL changes from `gateway-{location}.watsonplatform.net/assistant/api/` to `api.{location}.assistant.watson.cloud.ibm.com/`. The domain, location, and offering identifier are different in the new endpoint. For more information, see [Updating endpoint URLs from watsonplatform.net](#).

- If your service instance API credentials show the old endpoint, create a new credential and start using it today. After you update your custom applications to use the new credential, you can delete the old one.
- For a web chat integration, you might need to take action depending on when and how you created your integration.
 - If you tied your deployment to a specific web chat version by using the `clientVersion` parameter and specified a version earlier than version 3.3.0, update the parameter value to use version 3.3.0 or later. Web chat integrations that use the latest or 3.3.0 and later versions will not be impacted by the endpoint deprecation.
 - If you created your web chat integration before May 2020, check the code snippet that you embedded in your web page to see if it refers to `watsonplatform.net`. If so, you must edit the code snippet to use the new URL syntax. For example, change the following URL:

```
$ <script src="https://assistant-web.watsonplatform.net/loadWatsonAssistantChat.js"></script>
```

The correct syntax to use for the source service URL looks like this:

```
$ src="https://web-chat.global.assistant.watson.appdomain.cloud/loadWatsonAssistantChat.js"
```

- If your web chat integration connects to a Salesforce service desk, then you must edit the API call that is included in the code snippet that you added to the Visualforce Page that you created in Salesforce. From Salesforce, search for *Visualforce Pages*, and find your page. In the `<iframe>` snippet that you pasted into the page, make the following change:

Replace: `src="https://assistant-integrations-{location}.watsonplatform.net/public/salesforceweb"` with a url with this syntax:

```
src="https://integrations.{location}.assistant.watson.appdomain.cloud/public/salesforceweb/{integration-id}/agent_application?version=2020-09-24"
```

From the Web chat integration Salesforce live agent setup page, find the *Visualforce page markup* field. Look for the `src` parameter in the `<iframe>` element. It contains the full URL to use, including the appropriate `{location}` and `{integration-id}` values for your instance.

- For a Slack integration that is over 7 months old, make sure the Request URL is using the proper endpoint.
 - Go to the [Slack API](#) web page. Click *Your Apps* to find your assistant app. Click *Event Subscriptions* from the navigation pane.
 - Edit the Request URL.

For example, if the URL has the syntax: `https://assistant-slack-{location}.watsonplatform.net/public/message`, change it to have this syntax:

```
https://integrations.{location}.assistant.watson.appdomain.cloud/public/slack/{integration-id}/message?version=2020-09-24
```

Check the *Generated request URL* field in the Slack integration setup page for the full URL to use, which includes the appropriate `{location}` and `{integration-id}` values for your instance.

- For a Facebook Messenger integration that is over 7 months old, make sure the Callback URL is using the proper endpoint.
 - Go to the [Facebook for Developers](#) web page.
 - Open your app, and then select *Messenger>Settings* from the navigation pane.
 - Scroll down to the *Webhooks* section and edit the *Callback URL* field.

For example, if the URL has the syntax: `https://assistant-facebook-{location}.watsonplatform.net/public/message/`, change it to have this syntax:

```
https://integrations.{location}.assistant.watson.appdomain.cloud/public/facebook/{integration-id}/message?version=2020-09-24
```

Check the *Generated callback URL* field in the Facebook Messenger integration setup page for the full URL to use, which includes the appropriate `{location}` and `{integration-id}` values for your instance.

- For a Phone integration, if you connect to existing speech service instances, make sure those speech services use credentials that were generated with the latest endpoint syntax (a URL that starts with `https://api.{location}.speech-to-text.watson.cloud.ibm.com/`).
- For a search skill, if you connect to an existing Discovery service instance, make sure the Discovery service uses credentials that were generated with the supported syntax (a URL that starts with `https://api.{location}.discovery.watson.cloud.ibm.com/`).
- No action is required for the following integration types:
 - Intercom
 - SMS with Twilio
 - WhatsApp with Twilio
 - Zendesk service desk connection from web chat

23 March 2021

Actions skill improvement

Actions have a new toolbar making it easier to send feedback, access settings, save, and close.

17 March 2021

Channel transfer response type

Dialog skills now include a channel transfer response type. If your assistant uses multiple integrations to support different channels for interaction with users, there might be some situations when a customer begins a conversation in one channel but then needs to transfer to a different channel. The most common such situation is transferring a conversation to the web chat integration, to take advantage of web chat features such as service desk integration. For more information, see [Adding a Channel transfer response type](#).

Intercom and WhatsApp integrations now available in Lite plan

The integrations for Intercom and WhatsApp are now available in the Lite plan for Watson Assistant. For more information, see [Integrating with Intercom](#) and [Integrating with WhatsApp](#).

16 March 2021

Session history now generally available

Session history allows your web chats to maintain conversation history and context when users refresh a page or change to a different page on the same website. It is enabled by default. For more information about this feature, see [Session history](#).

Session history persists within only one browser tab, not across multiple tabs. The dialog provides an option for links to open in a new tab or the same tab. See [this example](#) for more information on how to format links to open in the same tab.

Session history saves changes that are made to messages with the [pre:receive event](#) so that messages still look the same on rerender. This data is only saved for the length of the session. If you prefer to discard the data, set `event.updateHistory = false;` so the message is rerendered without the changes that were made in the pre:receive event.

[instance.updateHistoryUserDefined\(\)](#) provides a way to save state for any message response. With the state saved, a response can be rerendered with the same state. This saved state is available in the `history.user_defined` section of the message response on reload. The data is saved during the user session. When the session expires, the data is discarded.

Two new history events, [history:begin](#) and [history:end](#) announce the beginning and end of the history of a reloaded session. These events can be used to view the messages that are being reloaded. The history:begin event allows you to edit the messages before they are displayed.

See this example for more information on saving the state of [customResponse](#) types in session history.

Channel switching

You can now create a dialog response type to functionally generate a connect-to-agent response within channels other than web chat. If a user is in a channel such as Slack or Facebook, they can trigger a channel transfer response type. The user receives a link that forwards them to your organization's website where a connection to an agent response can be started within web chat. For more information, see [Adding a Channel transfer response type](#).

11 March 2021

Actions skill improvement

Updated the page where you configure a step with an *Options* reply constraint. Now it's clearer that you have a choice to make about whether to always ask for the option value or to skip asking.

4 March 2021

Support for every language!

You now can build an assistant in any language you want to support. If a dedicated language model is not available for your target language, create a skill that uses the universal language model. The universal model applies a set of shared linguistic characteristics and rules from multiple languages as a starting point. It then learns from training data written in the target language that you add to it.

The universal model is available as a beta feature. For more information, see [Understanding the universal language model](#).

Actions skill improvement

Now you can indicate whether or not to ask for a number when you apply a number reply constraint to a step. Test how changes to this setting might help speed up a customer's interaction. Under the right circumstances, it can be useful to let a number mention be recognized and stored without having to explicitly ask the customer for it.

1 March 2021

Introducing the *Enterprise* plan!

The Enterprise plan includes all of the market differentiating features of the Plus plan, but with higher capacity limits, additional security features, custom onboarding support to get you going, and a lower overall cost at higher volumes.

To have a dedicated environment provisioned for your business, request the *Enterprise with Data Isolation* plan. To submit a request online, go to <http://ibm.biz/contact-wa-enterprise>.

The Enterprise plan is replacing the Premium plan. The Premium plan is being retired today. Existing Premium plan users are not impacted. They can continue to work in their Premium instances and create instances up to the 30-instance limit. New users do not see the Premium plan as an option when they create a service instance.

For more information, see the [Pricing](#) page.

Other plan changes

Our pricing has been revised to reflect the features we've added that help you build an assistant that functions as a powerful omnichannel SaaS application.

Starting on 1 March 2021, the Plus plan starts at \$140 per month and includes your first 1,000 monthly users. You pay \$14 for each additional 100 active users per month. Use of the voice capabilities that are provided by the *Phone* integration are available for an additional \$9 per 100 users per month.

The Plus Trial plan was renamed to Trial.

SOC 2 compliance

Watson Assistant is SOC 2 Type 2 compliant, so you know your data is secure.

The System and Organization Controls framework, developed by the American Institute of Certified Public Accountants (AICPA), is a standard for controls that protect information stored in the cloud. SOC 2 reports provide details about the nature of internal controls that are implemented to protect customer-owned data. For more information, see [IBM Cloud compliance programs](#).

25 February 2021

Search skill can emphasize the answer

You can configure the search skill to highlight text in the search result passage that Discovery determines to be the exact answer to the customer's question. For more information, see [Creating a search skill](#).

Integration changes

The following changes were made to the integrations:

- The name of *Preview link* integration changed to *Preview*.
- The *Web chat* and *Preview* integrations are no longer added automatically to every new assistant.

The integrations continue to be added to the *My first assistant* that is generated for you automatically when you first create a new service instance.

Message and log webhooks are generally available

The premessage, postmessage, and log webhooks are now generally available. For more information about them, see [Webhook overview](#).

11 February 2021

The `user_id` value is easier to access

The `user_id` property is used for billing purposes. Previously, it was available from the context object as follows:

- v2: `context.global.system.user_id`
- v1: `context.metadata.user_id`

The property is now specified at the root of the `/message` request in addition to the context object. The built-in integrations typically set this property for you. If you're using a custom application and don't specify a `user_id`, the `user_id` is set to the `session_id` (v2) or `conversation_id` (v1) value.

Digression bug fix

Fixed a bug where digression setting changes that were made to a node with slots were not being saved.

5 February 2021

Documentation update

The phone and *SMS with Twilio* deployment documentation was updated to include instructions for migrating from Voice Agent with Watson. For more information, see [Integrating with phone](#) and [Integrating with SMS with Twilio](#).

27 January 2021

German language improvements

A word decomposition function was added to the intent and entity recognition models for German-language dialog skills.

A characteristic of the German language is that some words are formed by concatenating separate words to form a single compound word. For example, "festnetznummer" (landline number) concatenates the words "festnetz" (landline) and "nummer" (number). When your customers chat with your assistant, they might write a compound word as a single word, as hyphenated words, or as separate words. Previously, the variants resulted in different intent confidence scores and different entity mention counts based on your training data. With the addition of the word decomposition function, the models now treat all compound word variants as equivalent. This update means you no longer need to add examples of every variant of the compound words to your training data.

19 January 2021

The *Phone* and *SMS with Twilio* integrations are now generally available!

For more information, see:

- [Integrating with phone](#)
- [Integrating with SMS with Twilio](#)

Preview link change

When you create a preview link, you can now test your skill from a chat window that is embedded in the page. You can also copy the URL that is provided, and open it in a web browser to see an IBM-branded web page with the web chat embedded in it. You can share the URL to the public IBM web page with others to get help with testing or for demoing purposes. For more information, see [Testing your assistant](#).

Import and export UI changes

The label on buttons for importing skills changed from *Import* to *Upload*, and the label on buttons for exporting skills changed from *Export* to *Download*.

Coverage metric change

The coverage metric now looks for nodes that were processed with a node condition that includes the `anything_else` special condition instead of

nodes that are named `Anything else`. For more information, see [Starting and ending the dialog](#).

15 January 2021

Use new webhooks to process messages!

A set of new webhooks is available as a beta feature. You can use the webhooks to perform preprocessing tasks on incoming messages and postprocessing tasks on the corresponding responses. You can use the new log webhook to log each message with an external service. For more information, see [Webhook overview](#).

New service desk support reference implementation

You can use the reference implementation details to integrate the web chat with the NICE inContact service desk. For more information, see [Adding service desk support](#).

Phone and SMS with *Twilio* integration updates

The phone integration now enables you to specify more than one phone number, and the numbers can be imported from a comma-separated values (CSV) file. The SMS with *Twilio* integration no longer requires you to add your SMS phone number to the setup page.

6 January 2021

Import and export UI changes

The label on buttons for importing intents and entities changed from *Import* to *Upload*. The label on buttons for exporting intents and entities changed from *Export* to *Download*.

4 January 2021

Dialog methods updates

Documentation and examples were added for the following supported dialog methods:

- `JSONArray.addAll(JSONArray)`
- `JSONArray.containsIgnoreCase(value)`
- `String.equals(String)`
- `String.equalsIgnoreCase(String)`

For more information, see [Expression language methods](#).

17 December 2020

Accessibility improvements

The product was updated to provide enhanced accessibility features.

14 December 2020

Increased Phone and SMS with *Twilio* integrations availability

These beta SMS and voice capabilities are now available from service instances that are hosted in Seoul, Tokyo, London, and Sydney.

Improved JSON editor

The JSON editor in the dialog skill was updated. The editor now uses JSON syntax highlighting and allows you to expand and collapse objects.

Connect to agent from actions skill

The actions skill now supports transferring a customer to an agent from within an action step.

4 December 2020

Introducing more service desk options for web chat

When you deploy your assistant by using the web chat integration, there are now reference implementations that you can use for the following service desks:

- Twilio Flex
- Genesys Cloud

Alternatively, you can bring your own service desk by using the service desk extension starter kit.

For more information, see [Adding service desk support](#).

Autolearning has been moved and improved

Go to the *Analytics>Autolearning* page to enable the feature and see visualizations that illustrate how autolearning impacts your assistant's performance over time. For more information, see [Empower your skill to learn automatically](#).

Search from actions skill

The actions skill now supports triggering a search that uses your associated search skill from within an action step.

System entities language support change

The new system entities are now used by all skills except Korean-language dialog skills. If you have a Korean skill that uses the older version of the system entities, update it. The legacy version will stop being supported for Korean skills in March 2021. For more information, see [Legacy system entities](#).

Disambiguation selection enhancement

When a customer chooses an option from a disambiguation list, the corresponding intent is submitted. With this latest release, a confidence score of 1.0 is assigned to the intent. Previously, the original confidence score of the option was used.

Skill import improvements

Importing of large skills from JSON data is now processed in the background. When you import a JSON file to create a skill, the new skill tile appears immediately. However, depending on the size of the skill, it might not be available for several minutes while the import is being processed. During this time, the skill cannot be opened for editing or added to an assistant, and the skill tile shows the text **Processing**.

23 November 2020

Deploy your assistant to WhatsApp!

Make your assistant available through WhatsApp messaging so it can exchange messages with your customers where they are. This beta integration creates a connection between your assistant and WhatsApp by using Twilio as a provider. For more information, see [Integrating with WhatsApp](#).

13 November 2020

New coverage metric and enhanced intent detection model

The following features are available in service instances hosted in all data center locations except Dallas.

Introducing the coverage metric!

Want a quick way to see how your dialog is doing at responding to customer queries? Enable the new coverage metric to find out. The coverage metric measures the rate at which your dialog is confident that it can address a customer's request per message. For conversations that are not

covered, you can review the logs to learn more about what the customer wanted. For the metric to work, you must design your dialog to include an **Anything else** node that is processed when no other dialog node intents are matched. For more information, see [Graphs and statistics](#).

Try out the enhanced intent detection model

The new model, which is being offered as a beta feature in English-language dialog and actions skills, is faster and more accurate. It combines traditional machine learning, transfer learning, and deep learning techniques in a cohesive model that is highly responsive at run time.

3 November 2020

Suggestions are now generally available

The Suggestions feature that is available for the web chat integration is generally available and is enabled by default when you create a new web chat integration. For more information, see [Showing more suggestions](#).

29 October 2020

System entity support changes

For English, Brazilian Portuguese, Czech, Dutch, French, German, Italian, and Spanish dialog skills only the new system entities API version is supported. For backward compatibility, both the **interpretation** and **metadata** attributes are included with the recognized entity object. The new system entity version is enabled automatically for dialog skills in the Arabic, Chinese, Korean, and Japanese languages. You can choose to use the legacy version of the system entities API by switching to it from the **Options>System Entities** page. This settings page is not displayed in English, Brazilian Portuguese, Czech, Dutch, French, German, Italian, and Spanish dialog skills because use of the legacy version of the API is no longer supported for those languages. For more information about the new system entities, see [System entities](#).

28 October 2020

Introducing the *actions skill*!

The actions skill is the latest step in the continuing evolution of Watson Assistant as a software as a service application. The actions skill is designed to make it simple enough for *anyone* to build a virtual assistant. We've removed the need to navigate between intents, entities, and dialog to create conversational flows. Building can all now be done in one simple and intuitive interface.

Web chat integration is created automatically

When you create a new assistant, a web chat integration is created for you automatically (in addition to the preview link integration, which was created previously). These integrations are added also to the assistant that is auto-generated (named *My first assistant*) when you create a new service instance. For more information, see [Integrating the web chat with your website](#).

Text messaging integration was renamed

The *Twilio messaging* integration was renamed to *SMS with Twilio*.

9 October 2020

Search skill update

Support was added for a new version of the Discovery API which adds the following capabilities:

- The search skill can now connect to existing Premium Discovery service instances.
- When you connect to a Box, Sharepoint, or Web crawl data collection, the result content fields are automatically populated for you. The **Title** now uses the **title** field from the source document instead of the **extracted_metadata.title** field, which provides better results.

1 October 2020

Introducing the *Phone* integration!

Your customers are calling; now your assistant can answer. Add a phone integration to enable your assistant to answer customer support calls. The integration connects to your existing Session Initiation Protocol (SIP) trunk, which routes incoming calls to your assistant. For more information, see [Integrating with phone](#).

Introducing the *Twilio messaging* integration!

Enable your assistant to receive and respond to questions that customers submit by using SMS text messaging. When you enable both new integrations, your assistant can send text messages to a customer in the context of an ongoing phone conversation. For more information, see [Integrating with Twilio messaging](#).

The *Phone* and *Twilio messaging* integrations are available as beta features in Watson Assistant service instances that are hosted in Dallas, Frankfurt, and Washington, DC.

The web chat integration is added to new assistants automatically

Much like the *Preview link* integration, the *Web chat* integration now is added to the *My first assistant* assistant that is created for new users automatically.

24 September 2020

Introducing the containment metric!

Want a quick way to see how often your assistant has to ask for help? Enable the new containment metric to find out. The containment metric measures the rate at which your assistant is able to address a customer's goal without human intervention. For conversations that are not contained, you can review the logs to understand what led customers to seek help outside of the assistant. For the metric to work, you must design your dialog to flag requests for additional support when they occur. For more information, see [Graphs and statistics](#).

Chat transfer improvements

When you add the *Connect to human agent* response type to a dialog node, you can now define messages to show to your customers during the transfer, and can specify service desk agent routing preferences. For more information, see [Adding a Connect to human agent response type](#).

22 September 2020

New API version

The current v2 API version is now `2020-09-24`. In this version, the structure of the `search` response type has changed. The `results` property has been removed and replaced with two new properties:

- `primary_results` property includes the search results that should be displayed in the initial response to a user query.
- `additional_results` property includes search results that can be displayed if the user wants to see more.

The search skill configuration determines how many search results are included in the `primary_results` and `additional_results` properties.

Search skill improvements

The following improvements were made to the search skill:

- **Control the number of search results** : You can now customize the number of search results that are shown in a response from the search skill. For more information, see [Configure the search](#).
- **FAQ extraction is available for web crawl data collections** : When you create a web crawl data collection type, you can now enable the FAQ extraction beta feature. FAQ extraction allows the Discovery service to identify question and answer pairs that it finds as it crawls the website. For more information, see [Create a data collection](#).

16 September 2020

Search skill refinement change

The search refinement beta feature that was added in [June](#) now is disabled by default. Enable the feature to refine the search results that are returned from the Discovery service. For more information, see [Configure the search](#).

25 August 2020

Give the web chat integration a try!

You can now use the web chat integration with a Lite plan. Previously, the web chat was available to Plus or higher plans only. For more information, see [Integrating the web chat with your website](#).

12 August 2020

v2 Logs API is available

If you have a Premium plan, you can use the v2 API `logs` method to list log events for an assistant. For more information, see the [API reference](#) documentation.

5 August 2020

Enable your skill to improve itself

Try the new **autolearning** beta feature to empower your skill to improve itself automatically over time. Your skill observes customer choices to understand which choices are most often the best. As its confidence grows, your skill presents better options to get the right answers to your customers with fewer clicks. For more information, see [Empower your skill to learn over time](#).

Show more of search results

When search results are returned from the search skill, the customer can now click a twistie to expand the search result card to see more of the returned text.

29 July 2020

The @sys-location and @sys-person system entities were removed

The `@sys-location` and `@sys-person` system entities are no longer listed on the *System entities* page. If your dialog uses one of these entities, a red `Entity not created` notification is displayed to inform you that the entity is not recognized.

Skill menu actions moved

The menu that was displayed in the header of the skill while you were working with a skill was removed. The actions that were available from the menu, such as import and export, are still available. Go to the Skills page, and click the menu on the skill tile.

The import skill process was updated to support overwriting an existing skill on import. For more information, see [Overwriting a skill](#).

Dialog issues were addressed

These dialog issues were addressed:

- Fixed an issue with adding a jump-to from a conditional response in one node to a conditional response in another node.
- The page now responds better when you scroll horizontally to see multiple levels of child nodes.

15 July 2020

Support ended for @sys-location and @sys-person

The person and location system entities, which were available as a beta feature in English dialog skills only, are no longer supported. You cannot enable them. If your dialog uses them, they are ignored by the service.

Use contextual entities to teach your skill to recognize the context in which such names are used. For more information about contextual entities, see [Annotation-based method](#).

For more information about how to use contextual entities to identify names of people, see the [Detecting Names And Locations With Watson Assistant](#) blog post on Medium.

How legacy numeric system entities are processed has changed

All new dialog skills use the new system entities automatically.

For existing skills that use legacy numeric system entities, how the entities are processed now differs based on the skill language.

- Arabic, Chinese, Korean, and Japanese dialog skills that use legacy numeric system entities function the same as before.
- If you choose to continue to use the legacy system entities in European-language dialog skills, a new legacy API format is used. The new legacy API format simulates the legacy system entities behavior. In particular, it returns a `metadata` object and does not stop the service from identifying multiple system entities for the same input string. In addition, it returns an `interpretation` object, which was introduced with the new version of system entities. Review the `interpretation` object to see the useful information that is returned by the new version.

Update your skills to use the new system entities from the [Options>System Entities](#) page.

Web chat security is generally available

Enable the security feature of web chat so that you can verify that messages sent to your assistant come from only your customers and can pass sensitive information to your assistant.

When configuring the JWT, you no longer need to specify the Authentication Context Class Reference (acr) claim.

1 July 2020

Salesforce support is generally available

Integrate your web chat with Salesforce so your assistant can transfer customers who asks to speak to a person to a Salesforce agent who can answer their questions. For more information, see [Integrating with Salesforce](#).

24 June 2020

Get better answers from search skill

The search skill now has a beta feature that limits the search results that are returned to include only those for which Discovery has calculated a 20% or higher confidence score. You can toggle the feature on or off from the *Refine results to return more selective answers* switch on the configuration page. You cannot change the confidence score threshold from 0.2. This beta feature is enabled by default. For more information, see [Creating a search skill](#).

3 June 2020

Zendesk support is generally available

Integrate your web chat with Zendesk so your assistant can transfer customers who asks to speak to a person to a Zendesk agent who can answer their questions. And now you can secure the connection to Zendesk. For more information, see [Adding support for transfers](#).

Pricing plan changes

We continue to revamp the overall service plan structure for Watson Assistant. In April, we announced [a new low cost entry point](#) for the Plus plan. Today, the Standard plan is being retired. Existing Standard plan users are not impacted; they can continue to work in their Standard instances. New users do not see the Standard plan as an option when they create a service instance. For more information, see the [Pricing](#) page.

27 May 2020

Full language support for new system entities

The new version of the system entities is generally available in dialog skills of all languages, including Arabic, Chinese (Simplified), Chinese (Traditional), Korean, and Japanese. For more information, see [Supported languages](#).

New system entities are enabled automatically

All new dialog skills use the new version of the system entities automatically. For more information, see [New system entities](#).

22 May 2020

Spelling correction in v2 API

The v2 `message` API now supports spelling correction options. For more information see the [API Reference](#).

21 May 2020

Preview link URL change

The URL for the preview link was changed. If you previously shared the link with teammates, provide them with the new URL.

15 May 2020

Private endpoints support is available in Plus plan

You can use private endpoints to route services over the IBM Cloud private network instead of the public network. For more information, see [Private network endpoints](#). This feature was previously available to users of Premium plans only.

14 May 2020

Get skill owner information

The email address of the person who owns the service instance that you are using is displayed from the User account menu. This information is especially helpful if you want to contact the instance owner to request access changes. For more information about access control, see [Managing access to resources](#).

System entity deprecation

As stated in the [March deprecation notice](#), the `@sys-location` and `@sys-person` system entities that were available as a beta feature are deprecated. If you are using one of these system entities in your dialog, a toggle is displayed for the entity on the *System entities* page. You can [search your dialog](#) to find out where you are currently using the entity, and remove it. Consider using a contextual entity to identify references to locations and people instead. After removing the entity from your dialog, disable the entity from the *System entities* page.

13 May 2020

Stateless v2 message API

The v2 runtime API now supports a new stateless `message` method. If you have a client application that manages its own state, you can use this new method to take advantage of [many of the benefits](#) of the v2 API without the overhead of creating sessions. For more information, see the [API](#)

30 April 2020

Web chat is generally available!

Add your assistant to your company website as a web chat widget that can help your customers with common questions and tasks. Service desk transfer support continues to be a beta feature. For more information, see [Integrating with your own website](#).

Secure your web chat

Enable the beta security feature of web chat so that you can verify that messages sent to your assistant come from only your customers and can pass sensitive information to your assistant.

27 April 2020

Add personality to your assistant in web chat

You can add an assistant image to the web chat header to brand the window. You can add an avatar image that represents your assistant or a brand logo, for example. For more information, see [Integrating with your own web site](#).

Know your plan

Now your service plan is displayed in the page header. And if you have a Plus Trial plan, you can see how many days are left in the trial.

21 April 2020

Fuzzy matching support was expanded

Added support for stemming and misspelling in French, German, and Czech dialog skills. This enhancement means that the assistant can recognize an entity value that is defined in its singular form but mentioned in its plural form in user input. It also can recognize conjugated forms of a verb that is specified as an entity value.

For example, if your French-language dialog skill has an entity value of `animal`, it recognizes the plural form of the word (`animaux`) when it is mentioned in user input. If your German-language dialog skill has the root verb `haben` as an entity value, it recognizes conjugated forms of the verb (`hast`) in user input as mentions of the entity.

2 April 2020

New and improved access control

Now, when you give other people access to your Watson Assistant resources, you have more control over the level of access they have to individual skills and assistants. You can give one person read-only access to a production skill and manager-level access to a development skill, for example. For more information, see [Managing access to resources](#).

Can't see Analytics anymore? If you cannot do things that you could do before, you might not have appropriate access. Ask the service instance owner to change your service access role. For more information, see [How to keep your access](#).

If you can't access the API Details for a skill or assistant anymore, you might not have the access role that is required to use the instance-level API credentials. You can use a personal API key instead. For more information, see [Getting API information](#).

1 April 2020

Plus plan changes

The Plus plan is now available starting at \$120/month for 1,000 users on pay-as-you-go or subscription IBM Cloud accounts. And you can subscribe without contacting Sales.

French language beta support added for contextual entities

You can add contextual entities to French-language dialog skills. For more information about contextual entities, see [Creating entities](#).

New API version

The current API version is now **2020-04-01**. The following change was made with this version:

- An `integrations` property was added to the V2 `/message` context. The service now expects the `context.integrations` property to conform to a specific schema in which the allowed values are as follows:
 - `chat`
 - `facebook`
 - `intercom`
 - `liveengage`
 - `salesforce`
 - `slack`
 - `service_desk`
 - `text.messaging`
 - `voice.telephony`
 - `zendesk`

If your app uses a `context.integrations` property that does not conform to the schema, a 400 error code will be returned.

31 March 2020

The web chat integration was updated

The update adds an `isTrackingEnabled` parameter. You can add this parameter and set it to `false` to add the `X-Watson-Learning-Opt-Out` header to each `/message` request that originates from the web chat. For more information about the header, see [Data collection](#). For more information about the parameter, see [Configuration](#).

26 March 2020

The Covid-19 content catalog is available in Brazilian Portuguese, French, and Spanish

The content catalog defines a group of intents that recognize the common types of questions people ask about the novel coronavirus. You can use the catalog to jump-start development of chatbots that can answer questions about the virus and help to minimize the anxiety and misinformation associated with it. For more information about how to add a content catalog to your skill, see [Using content catalogs](#).

19 March 2020

A Covid-19 content catalog is available

The English-only content catalog defines a group of intents that recognize the common types of questions people ask about the novel coronavirus. The World Health Organization characterized COVID-19 as a pandemic on 11 March 2020. You can use the catalog to jump-start development of chatbots that can answer questions about the virus and help to minimize the anxiety and misinformation associated with it. For more information about how to add a content catalog to your skill, see [Using content catalogs](#).

Fixed a problem with missing User Conversation data

A recent change resulted in no logs being shown in the User Conversations page unless you had a skill as the chosen data source. And the chosen skill had to be the same skill (with same skill ID) that was connected to the assistant when the user messages were submitted.

18 March 2020

Technology preview is discontinued

The technology preview user interface was replaced with the Watson Assistant standard user interface. If you used an Actions page to create actions and steps for your skill previously, you cannot access the Actions page anymore. Instead, use the Intents and Dialog pages to work with your skill.

16 March 2020

Instructions updated for Slack integrations

The steps required to set up a Slack integration have changed to reflect permission assignment changes that were made by Slack. For more information, see [Integrating with Slack](#).

Order of response types is preserved

Previously, if you included a response type of **Search skill** in a list of response types for a dialog node, the search results were displayed last despite its placement in the list. This behavior was changed to show the search results in the appropriate order, namely in the sequence in which the search skill response type is listed for the dialog node.

10 March 2020

Contextual entity support is generally available

You can add contextual entities to English-language dialog skills. For more information about contextual entities, see [Creating entities](#).

French language support added for autocorrection

Autocorrection helps your assistant understand what your customers want. It corrects misspellings in the input that customers submit before the input is evaluated. With more precise input, your assistant can more easily recognize entity mentions and understand the customer's intent. See [Correcting user input](#) for more details.

The new system entities are used by new skills

For new English, Brazilian Portuguese, Czech, Dutch, French, German, Italian, and Spanish dialog skills, the new system entities are enabled automatically. If you decide to turn on a system entity and add it to your dialog, it's the new and improved version of the system entity that is used.

For more information, see [New system entities](#).

6 March 2020

Transfer a web chat conversation to a human agent

Delight your customers with 360-degree support by integrating your web chat with a third-party service desk solution. When a customer asks to speak to a person, you can connect them to an agent through a service desk solution, such as Zendesk or Salesforce. Service desk support is a beta feature. For more information, see [Adding support for transfers](#).

2 March 2020

Known issue accessing logs

If you cannot access user logs from the Analytics page, ask the owner of the service instance for the skill to change your service level access to make you a Manager of the instance. For more information about access control, see [Managing access to resources](#).

1 March 2020 deprecation notice

March 2020 deprecation notice

To help us continue to improve and expand the capabilities of the assistants you build with Watson Assistant, we are deprecating some of the older technologies. Support for the older technologies will end in June 2020. Take action now to test and adopt the new technologies, so your skills and assistants will be ready when the old technologies stop being supported.

The following technologies are being deprecated:

- **Legacy version of numeric system entities**

We released a whole new infrastructure for our numeric system entities across all languages except Chinese, Korean, Japanese and Arabic. The updated `@sys-number`, `@sys-date`, `@sys-time`, `@sys-currency`, and `@sys-percentage` entities provide superior number recognition with higher precision. For more information about the new system entities, see [System entity details](#).

The old version of the numeric system entities will stop being supported in June 2020 for English, Brazilian Portuguese, Czech, Dutch, French, German, Italian, and Spanish dialog skills.

Action: In each dialog skill where you use numeric system entities, go to the [Options>System entities](#) page and turn on the new system entities. Take some time to test the new version of system entities with your own dialogs to make sure they continue to work as expected. As you adopt the new system entities, share your feedback about your experience with the new technology.

- **Person and location system entities**

The `@sys-person` and `@sys-location` system entities, which were available in English as a beta only, are being deprecated. Consider using contextual entities as a way to capture these types of proper nouns. Instead of trying to add a dictionary-based entity that covers every permutation of the names for people or cities, for example, you can teach your skill to recognize the context in which such names are used. For more information about contextual entities, see [Annotation-based method](#).

Action: Remove references to `@sys-person` and `@sys-location` from your dialogs. Turn off the `@sys-person` and `@sys-location` system entities to prevent yourself or others from adding them to a dialog inadvertently.

- **Irrelevance detection**

We revised the irrelevance detection classification algorithm to make it even smarter out of the box. Now, even before you begin to teach the system about irrelevant requests, it is able to recognize user input that your skill is not designed to address. For more information, see [Irrelevance detection](#).

Action: In each dialog skill, go to the [Options>Irrelevance detection](#) page and turn on the new classification model. Make sure everything works as well, if not better, than it did before. Share your feedback.

- **Old API version dates**

v1 API versions that are dated on or before `2017-02-03` are being deprecated. When you send calls to the service with earlier API version dates, they will receive properly formatted and valid responses for a time, so you can gracefully transition to using the later API versions. However, the confidence scores and other results that are sent in the response will reflect those generated by a more recent version of the API.

Action: Do some testing of calls with the latest version to verify that things work as expected. Some functionality has changed over the last few years. After testing, change the version date on any API calls that you make from your applications.

26 February 2020

Slot `Save it as` field retains your edits

When you edit what gets saved for a slot by using the JSON editor to edit the value of the context variable to be something other than what is specified in the **Check for** field, your changes are kept even if someone subsequently clicks the **Save it as** field.

20 February 2020

Access control changes are coming

Notifications are displayed in the user interface for anyone with Reader and Writer level access to a service instance. The notification explains that access control is going to change soon, and that what they can do in the instance will change unless they are given Manager service access beforehand. For more information, see [Preventing loss of access](#).

14 February 2020

More web chat color settings

You can now specify the color of more elements of the web chat integration. For example, you can define one color for the web chat window header. You can define a different color for the user message bubble. And another color for interactive components, such as the launcher button for the chat.

13 February 2020

Track API events

Premium plan users can now use the Activity Tracker service to track how users and applications interact with IBM Watson® Assistant in IBM Cloud®. See [Activity Tracker events](#).

5 February 2020

New API version

The current API version is now **2020-02-05**. The following changes were made with this version:

- When a dialog node's response type is `connect-to-agent`, the node's `title` is used as the `topic` value. Previously, `user_label` was used.
- The `alternate_intents` property is stored as a Boolean value instead of a String.

4 February 2020

Product user interface makeover

The UI has been updated to be more intuitive, responsive, and consistent across its pages. While the look and feel of the UI elements has changed, their function has not.

Requesting early access

The button you click to request participation in the early access program has moved from the Skills page to the user account menu. For more information, see [Feedback](#).

24 January 2020

New system entities are now generally available in multiple languages

The new and improved numeric system entities are now generally available in all supported languages, except Arabic, Chinese, Japanese, and Korean, where they are available as a beta feature. They are not used by your dialog skill unless you enable them from the **Options>System entities** page. For more information, see [New system entities](#).

14 January 2020

Fixed an error message that was displayed when opening an instance

An error that was displayed when you launched Watson Assistant from the IBM Cloud® dashboard has been fixed. Previously, an error message that said, `Module 'ui-router' is not available! You either misspelled the module name or forgot to load it` would sometimes be displayed.

12 December 2019

Support for private network endpoints

Users of Premium plans can create private network endpoints to connect to Watson Assistant over a private network. Connections to private network endpoints do not require public internet access. For more information, see [Protecting sensitive information](#).

Full support for IBM Cloud IAM

Watson Assistant now supports the full implementation of IBM Cloud Identity and Access Management (IAM). API keys for Watson services are no longer limited to a single service instance. You can create access policies and API keys that apply to more than one service, and you can grant access between services.

- To support this change, the API service endpoints use a different domain and include the service instance ID. The pattern is `api.{location}.{offering}.watson.cloud.ibm.com/instances/{instance_id}`.

Example URL for an instance hosted in the Dallas location: `api.us-south.assistant.watson.cloud.ibm.com/instances/6bbda3b3-d572-45e1-8c54-22d6ed9e52c2`

The previous public endpoint domain was `watsonplatform.net`.

For more information, see the [API reference](#).

These URLs do not introduce a breaking change. The new URLs work both for your existing service instances and for new instances. The original URLs continue to work on your existing service instances for at least one year (until December 2020).

- For more information, see [Authenticating to Watson services](#).

26 November 2019

Disambiguation is available to everyone

Disambiguation is now available to users of every plan type.

The following changes were made to how it functions:

- The text that you add to the dialog **node name** field now matters.
- The text in the node name field might be shown to customers. The disambiguation feature shows it to customers if the assistant needs to ask them to clarify their meaning. The text you add as the node name must identify the purpose of the node clearly and succinctly, such as *Place an order* or *Get plan information*.

If the *External node name* field exists and contains a summary of the node's purpose, then its summary is shown in the disambiguation list instead. Otherwise, the dialog node name content is shown.

- Disambiguation is enabled automatically for all nodes. You can disable it for the entire dialog or for individual dialog nodes.
- When testing, you might notice that the order of the options in the disambiguation list changes from one test run to the next. Don't worry; this new behavior is intended. As part of work being done to help the assistant learn automatically from user choices, the order of the options in the disambiguation list is being randomized on purpose. Changing the order helps to avoid bias that can be introduced by a percentage of people who always pick the first option without first reviewing their choices.

12 November 2019

Slot prompt JSON editor

You can now use the context or JSON editors for the slot response field where you define the question that your assistant asks to get information it needs from the customer. For more information about slots, see [Gathering information with slots](#).

New South Korea location

You can now create Watson Assistant instances in the Seoul location. As with other locations, the IBM Cloud Seoul location uses token-based Identity and Access Management (IAM) authentication.

Technology preview

A technology preview experience was released. A select set of new users are being presented with a new user interface that takes a different approach to building an assistant.

7 November 2019

Irrelevance detection has been added

When enabled, a supplemental model is used to help identify utterances that are irrelevant and should not be answered by the dialog skill. This new model is especially beneficial for skills that have not been trained on what subjects to ignore. This feature is available for English skills only. For more information, see [Irrelevance detection](#).

Time zone support for now() method

You can now specify the time zone for the date and time that is returned by the `now()` method. See [Now\(\)](#).

24 October 2019

Testing improvement

You can now see the top three intents that were recognized in a test user input from the "Try it out" pane. For more details, see [Testing your dialog](#).

Error message when opening an instance

When you launch Watson Assistant from the IBM Cloud® dashboard, you might see an error message that says, `Module 'ui-router' is not available! You either misspelled the module name or forgot to load it.` You can ignore the message. Refresh the web browser page to close the notification.

16 October 2019

The changes from 14 October are now available in Dallas.

14 October 2019

Deploy your assistant in minutes

Create a web chat integration to embed your assistant into a page on your website as a chat widget. See [Integrating with your own website](#).

UI changes

The main menu options of **Assistants** and **Skills** have moved from being displayed in the page header to being shown as icons on the side of the page. The tabbed pages for the tools you use to develop a dialog skill were moved to a secondary navigation bar that is displayed when you open the skill.

Rich response types are supported in a dialog node with slots

You can display a list of options for a user to choose from as the prompt for a slot, for example.

Change to switching service instances

Where you go to switch service instances has changed. See [Switching the service instance](#).

Known issue: Cannot rename search skills

You currently cannot rename a search skill after you create it.

9 October 2019

New system entities changes

The following updates have been made:

- In addition to English and German, the new numeric system entities are now available in these languages: Brazilian Portuguese, Czech, French, Italian, and Spanish.
- The `part_of_day` property of the `@sys-time` entity now returns a time range instead of a single time value.

23 September 2019

Dallas updates

The updates from 20 September are now available to service instances hosted in Dallas.

20 September 2019

Inactivity timeout increase

The maximum inactivity timeout can now be extended to up to 7 days for Premium plans. See [Changing the inactivity timeout setting](#).

Pattern entity fix

A change that was introduced in the previous release which changed all alphabetic characters to lowercase at the time an entity value was added has been fixed. The case of any alphabetic characters that are part of a pattern entity value are no longer changed when the value is added.

Dialog text response syntax fix

Fixed a bug in which the format of a dialog response reverted to an earlier version of the JSON syntax. Standard text responses were being saved as `output.text` instead of `output.generic`. For more information about the `output` object, see [Anatomy of a dialog call](#).

13 September 2019

Improved Entities and Intents page responsiveness

The Entities and Intents pages were updated to use a new JavaScript library that increases the page responsiveness. As a result, the look of some graphical user interface elements, such as buttons, changed slightly, but the function did not.

Creating contextual entities got easier

The process you use to annotate entity mentions from intent user examples was improved. You can now put the intent page into annotation mode to more easily select and label mentions. See [Adding contextual entities](#).

6 September 2019

Label character limit increase

The limit to the number of characters allowed for a label that you define for an option response type changed from 64 characters to 2,048 characters.

12 August 2019

New dialog method

The `getMatch` method was added. You can use this method to extract a specific occurrence of a regular expression pattern that recurs in user input. For more details, see the [dialog methods](#) topic.

9 August 2019

Introductory product tour

For some first-time users, a new introductory product tour is shown that the user can choose to follow to perform the initial steps of creating an assistant.

6 August 2019

- Webhook callouts and Dialog page improvements are available in Dallas.

1 August 2019

Webhook callouts are available

Add webhooks to dialog nodes to make programmatic calls to an external application as part of the conversational flow. The new Webhook support simplifies the callout implementation process. (No more `action` JSON objects required.) For more information, see [Making a programmatic call from a dialog node](#).

Improved dialog page responsiveness

In all service instances, the user interface of the Dialog page was updated to use a new JavaScript library that increases the page responsiveness. As a result, the look of some graphical user interface elements, such as buttons, changed slightly, but the function did not.

31 July 2019

Search skill and autocorrection are generally available

The search skill and spelling autocorrection features, which were previously available as beta features, are now generally available.

- Search skills can be created by users of Plus or Premium plans only.
- You can enable autocorrection for English-language dialog skills only. It is enabled automatically for new English-language dialog skills.

26 July 2019

Missing skills issue is resolved

In some cases, workspaces that were created through the API only were not being displayed when you opened the Watson Assistant user interface. This issue has been addressed. All workspaces that you create by using the API are displayed as dialog skills when you open the user interface.

23 July 2019

Dialog search is fixed

In some skills, the search function was not working in the Dialog page. The issue is now fixed.

17 July 2019

Disambiguation choice limit

You can now set the maximum number of options to show to users when the assistant asks them to clarify what they want to do. For more information about disambiguation, see [Disambiguation](#).

Dialog search issue

In some skills, the search function is not working in the Dialog page. A new user interface library, which increases the page responsiveness, is being rolled out to existing service instances in phases. This search issue affects only dialog skills for which the new library is not yet enabled.

Missing skills issue

In some cases, workspaces that were created through the API only are not being displayed when you open the Watson Assistant user interface. Normally, these workspaces are displayed as dialog skills. If you do not see your skills from the UI, don't worry; they are not gone. Contact support to report the issue, so the team can enable the workspaces to be displayed properly.

15 July 2019

Numeric system entities upgrade available in Dallas

The new system entities are now also available as a beta feature for instances that are hosted in Dallas. See [New system entities](#)

12 June 2019

Numeric system entities upgrade

New system entities are available as a beta feature that you can enable in dialog skills that are written in English or German. The revised system entities offer better date and time understanding. They can recognize date and number spans, national holiday references, and classify mentions with more precision. For example, a date such as **May 15** is recognized as a date mention(**@sys-date:2019-05-15**), and is *not* also identified as a number mention (**@sys-number:15**). See [New system entities](#)

A Plus Trial plan is available

You can use the free Plus Trial plan to try out the features of the Plus plan as you make a purchasing decision. The trial lasts for 30 days. After the trial period ends, if you do not upgrade to a Plus plan, your Plus Trial instance is converted to a Lite plan instance.

23 May 2019

Updated navigation

The home page was removed, and the order of the Assistants and Skills tabs was reversed. The new tab order encourages you to start your development work by creating an assistant, and then a skill.

Disambiguation settings have moved

The toggle to enable disambiguation, which is a feature that is available to Plus and Premium plan users only, has moved. The **Settings** button was removed from the **Dialog** page. You can now enable disambiguation and configure it from the skill's **Options** tab.

An introductory tour is now available

A short product tour is now displayed when a new service instance is created. Brand new users are also given help as they start development. A new assistant is created for them automatically. Informational popups are displayed to introduce the product user interface features, and guide the new user toward taking the key first step of creating a dialog skill.

10 April 2019

Autocorrection is now available

Autocorrection is a beta feature that helps your assistant understand what your customers want. It corrects misspellings in the input that customers submit before the input is evaluated. With more precise input, your assistant can more easily recognize entity mentions and understand the customer's intent. See [Correcting user input](#) for more details.

22 March 2019

Introducing search skill

A search skill helps you to make your assistant useful to customers faster. Customer inquiries that you did not anticipate and so have not built dialog logic to handle can be met with useful responses. Instead of saying it can't help, the assistant can query an external data source to find relevant information to share in its response. Over time, you can build dialog responses to answer customer queries that require follow-up questions to clarify the user's meaning or for which a short and clear response is suitable. And you can use search skill responses to address more open-ended customer queries that require a longer explanation. This beta feature is available to users of Premium and Plus service plans only.

See [Building a search skill](#) for more details.

14 March 2019

4 March 2019

Simplified navigation

The sidebar navigation with separate *Build*, *Improve*, and *Deploy* tabs has been removed. Now, you can get to all the tools you need to build a dialog skill from the main skill page.

Improve page is now called Analytics

The informational metrics that Watson generates from conversations between your users and your assistant moved from the *Improve* tab of the sidebar to a new tab on the main skill page called **Analytics**.

1 March 2019

28 February 2019

New API version

The current API version is now **2019-02-28**. The following changes were made with this version:

- The order in which conditions are evaluated in nodes with slots has changed. Previously, if you had a node with slots that allowed for digressions away, the `anything_else` root node was triggered before any of the slot level Not found conditions could be evaluated. The order of operations has been changed to address this behavior. Now, when a user digresses away from a node with slots, all the root nodes except the `anything_else` node are processed. Next, the slot level Not found conditions are evaluated. And, finally, the root level `anything_else` node is processed. To better understand the full order of operations for a node with slots, see [Slot usage tips](#).
- Strings that begin with a number sign (#) in the `context` or `output` objects of a message are no longer treated as intent references.

Previously, these strings were treated as intents automatically. For example, if you specified a context variable, such as `"color": "#FFFFFF"`, then the hex color code (#FFFFFF) would be treated as an intent. Your assistant would check whether an intent named #FFFFFF was detected in the user's input, and if not, would replace #FFFFFF with `false`. This replacement no longer occurs.

Similarly, if you included a number sign (#) in the text string in a node response, you used to have to escape it by preceding it with a back slash (\). For example, `We are the \#1 seller of lobster rolls in Maine.` You no longer need to escape the # symbol in a text response.

This change does not apply to node or conditional response conditions. Any strings that begin with a number sign (#) which are specified in conditions continue to be treated as intent references. Also, you can use SpEL expression syntax to force the system to treat a string in the `context` or `output` objects of a message as an intent. For example, specify the intent as `<? #intent-name ?>`.

25 February 2019

Slack integration enhancement

You can now choose the type of event that triggers your assistant in a Slack channel. Previously, when you integrated your assistant with Slack, the

assistant interacted with users through a direct message channel. Now, you can configure the assistant to listen for mentions, and respond when it is mentioned in other channels. You can choose to use one or both event types as the mechanism through which your assistant interacts with users.

11 February 2019

Integrate with Intercom

Intercom, a leading customer service messaging platform, has partnered with IBM to add a new agent to the team, a virtual Watson Assistant. You can integrate your assistant with an Intercom application to enable the app to seamlessly pass user conversations between your assistant and human support agents. This integration is available to Plus and Premium plan users only. See [Integrating with Intercom](#) for more details.

8 February 2019

Version your skills

You can now capture a snapshot of the intents, entities, dialog, and configuration settings for a skill at key points during the development process. With versions, it's safe to get creative. You can deploy new design approaches in a test environment to validate them before you apply any updates to a production deployment of your assistant. See [Creating skill versions](#) for more details.

Arabic content catalog

Users of Arabic-language skills can now add prebuilt intents to their dialogs. See [Using content catalogs](#) for more information.

17 January 2019

Czech language support is generally available

Support for the Czech language is no longer classified as beta; it is now generally available. See [Supported languages](#) for more information.

Language support improvements

The language understanding components were updated to improve the following features:

- German and Korean system entities
- Intent classification tokenization for Arabic, Dutch, French, Italian, Japanese, Portuguese, and Spanish

4 January 2019

IBM Cloud Functions in DC and London locations

You can now make programmatic calls to IBM Cloud Functions from the dialog of an assistant in a service instance that is hosted in the London and Washington, DC data centers. See [Making programmatic calls from a dialog node](#).

New methods for working with arrays

The following SpEL expression methods are available that make it easier to work with array values in your dialog:

- **JSONArray.filter**: Filters an array by comparing each value in the array to a value that can vary based on user input.
- **JSONArray.includesIntent**: Checks whether an `intents` array contains a particular intent.
- **JSONArray.indexOf**: Gets the index number of a specific value in an array.
- **JSONArray.joinToArray**: Applies formatting to values that are returned from an array.

See the [array method documentation](#) for more details.

13 December 2018

London data center

You can now create Watson Assistant service instances that are hosted in the London data center without syndication. See [Data centers](#) for more details.

Dialog node limit changes

The dialog node limit was temporarily changed from 100,000 to 500 for new Standard plan instances. This limit change was later reversed. If you created a Standard plan instance during the time frame in which the limit was in effect, your dialogs might be impacted. The limit was in effect for skills created between 10 December and 12 December 2018. The lower limits will be removed from all impacted instances in January. If you need to have the lower limit lifted before then, open a support ticket.

1 December 2018

Determine the number of dialog nodes

To determine the number of dialog nodes in a dialog skill, do one of the following things:

- From the tool, if it is not associated with an assistant already, add the dialog skill to an assistant, and then view the skill tile from the main page of the assistant. The *trained data* section lists the number of dialog nodes.
- Send a GET request to the /dialog_nodes API endpoint, and include the `include_count=true` parameter. For example:

```
curl -u "apikey:{apikey}" "https://{{service-hostname}}/assistant/api/v1/workspaces/{{workspace_id}}/dialog_nodes?version=2018-09-20&include_count=true"
```

where {{service-hostname}} is the appropriate URL for your instance. For more details, see [Service endpoint](#).

In the response, the `total` attribute in the `pagination` object contains the number of dialog nodes.

See [Troubleshooting skill import issues](#) for information about how to edit skills that you want to continue using.

27 November 2018

A new service plan, the Plus plan, is available

The new plan offers premium-level features at a lower price point. Unlike previous plans, the Plus plan is a user-based billing plan. It measures usage by the number of unique users that interact with your assistant over a given time period. To get the most from the plan, if you build your own client application, design your app such that it defines a unique ID for each user, and passes the user ID with each /message API call. For the built-in integrations, the session ID is used to identify user interactions with the assistant. See [User-based plans](#) for more information.

Artifact	Limit
Assistants	100
Contextual entities	20
Contextual entity annotations	2,000
Dialog nodes	100,000
Entities	1,000
Entity synonyms	100,000
Entity values	100,000

Intents	2,000
Intent user examples	25,000
Integrations	100
Logs	30 days
Skills	50

Plus plan limits

User-based Premium plan

The Premium plan now bases its billing on the number of active unique users. If you choose to use this plan, design any custom applications that you build to properly identify the users who generate /message API calls. See [User-based plans](#) for more information.

Existing Premium plan service instances are not impacted by this change; they continue to use API-based billing methods. Only existing Premium plan users will see the API-based plan listed as the *Premium (API)* plan option.

See Watson Assistant [service plan options](#) for more information about all available service plans.

20 November 2018

Recommendations are discontinued

The Recommendations section on the Improve tab was removed. Recommendations was a beta feature available to Premium plan users only. It recommended actions that users could take to improve their training data. Instead of consolidating recommendations in one place, recommendations are now being made available from the parts of the tool where you make actual training data changes. For example, while adding entity synonyms, you can now opt to see a list of synonymous terms that are recommended by Watson.

9 November 2018

Major user interface revision

The Watson Assistant service has a new look and added features.

This version of the tool was evaluated by beta program participants over the past several months.

- **Skills:** What you think of as a *workspace* is now called a *skill*. A *dialog skill* is a container for the natural language processing training data and artifacts that enable your assistant to understand user questions, and respond to them.

Where are my workspaces? Any workspaces that you created previously are now listed in your service instance as skills. Click the **Skills** tab to see them. For more information, see [Adding skills to your assistant](#).

- **Assistants:** You can now publish your skill in just two steps. Add your skill to an assistant, and then set up one or more integrations with which to deploy your skill. The assistant adds a layer of function to your skill that enables Watson Assistant to orchestrate and manage the flow of information for you. See [Assistants](#).
- **Built-in integrations:** Instead of going to the **Deploy** tab to deploy your workspace, you add your dialog skill to an assistant, and add integrations to the assistant through which the skill is made available to your users. You do not need to build a custom front-end application and manage the conversation state from one call to the next. However, you can still do so if you want to. See [Adding integrations](#) for more information.
- **New major API version:** A V2 version of the API is available. This version provides access to methods you can use to interact with an assistant at run time. No more passing context with each API call; the session state is managed for you as part of the assistant layer.

What is presented in the tooling as a dialog skill is effectively a wrapper for a V1 workspace. There are currently no API methods for authoring skills and assistants with the V2 API. However, you can continue to use the V1 API for authoring workspaces. See [API Overview](#) for more details.

- **Switching data sources:** It is now easier to improve the model in one skill with user conversation logs from a different skill. You do not need to rely on deployment IDs, but can simply pick the name of the assistant to which a skill was added and deployed to use its data. See [Improving](#)

[across assistants.](#)

- **Preview links from London instances** : If your service instance is hosted in London, then you must edit the preview link URL. The URL includes a region code for the region where the instance is hosted. Because instances in London are syndicated to Dallas, you must replace the `eu-gb` reference in the URL with `us-south` for the preview web page to render properly.

8 November 2018

Japanese data center

You can now create Watson Assistant service instances that are hosted in the Tokyo data center. See [Data centers](#) for more details.

30 October 2018

New API authentication process

The Watson Assistant service transitioned from using Cloud Foundry to using token-based Identity and Access Management (IAM) authentication in the following regions:

- Dallas (us-south)
- Frankfurt (eu-de)

For new service instances, you use IAM for authentication. You can pass either a bearer token or an API key. Tokens support authenticated requests without embedding service credentials in every call. API keys use basic authentication.

For all existing service instances, you continue to use service credentials (`{username}:{password}`) for authentication.

25 October 2018

Entity synonym recommendations are available in more languages

Synonym recommendation support was added for the French, Japanese, and Spanish languages.

26 September 2018

IBM Watson® Assistant is available in IBM® Cloud Private

IBM Watson® Assistant is available in IBM® Cloud Private

21 September 2018

New API version

The current API version is now `2018-09-20`. In this version, the `errors[] .path` attribute of the error object that is returned by the API is expressed as a [JSON Pointer](#) instead of in dot notation form.

Web actions support

You can now call Cloud Functions web actions from a dialog node. See [Making programmatic calls from a dialog node](#) for more details.

15 August 2018

Entity fuzzy matching support improvements

Fuzzy matching is fully supported for English entities, and the misspelling feature is no longer a Beta-only feature for many other languages. See [Supported languages](#) for details.

6 August 2018

Intent conflict resolution

The tool can now help you to resolve conflicts when two or more user examples in separate intents are similar to one another. Non-distinct user examples can weaken the training data and make it harder for your assistant to map user input to the appropriate intent at run time. See [Resolving intent conflicts](#) for details.

Disambiguation

Enable disambiguation to allow your assistant to ask the user for help when it needs to decide between two or more viable dialog nodes to process for a response. See [Disambiguation](#) for more details.

Jump-to fix

Fixed a bug in the Dialogs tool which prevented you from being able to configure a jump-to that targets the response of a node with the `anything_else` special condition.

Digression return message

You can now specify text to display when the user returns to a node after a digression. The user will have seen the prompt for the node already. You can change the message slightly to let users know they are returning to where they left off. For example, specify a response like, `Where were we? Oh, yes...` See [Digressions](#) for more details.

12 July 2018

Rich response types

You can now add rich responses that include elements such as images or buttons in addition to text, to your dialog. See [Rich responses](#) for more information.

Contextual entities (Beta)

Contextual entities are entities that you define by labeling mentions of the entity type that occur in intent user examples. These entity types teach your assistant not only terms of interest, but also the context in which terms of interest typically appear in user utterances, enabling your assistant to recognize never-seen-before entity mentions based solely on how they are referenced in user input. For example, if you annotate the intent user example, `I want a flight to Boston` by labeling `Boston` as a `@destination` entity, then your assistant can recognize `Chicago` as a `@destination` mention in a user input that says, `I want a flight to Chicago.` This feature is currently available for English only. See [Adding contextual entities](#) for more information.

When you access the tool with an Internet Explorer web browser, you cannot label entity mentions in intent user examples nor edit user example text.

New API version

The current API version is now `2018-07-10`. This version introduces the following changes:

- The content of the /message `output` object changed from being a `text` JSON object to being a `generic` array that supports multiple rich response types, including `image`, `option`, `pause`, and `text`.
- Support for contextual entities was added.
- You can no longer add user-defined properties in `context.metadata`. However, you can add them directly to `context`.

Overview page date filter

Use the new date filters to choose the period for which data is displayed. These filters affect all data shown on the page: not just the number of conversations displayed in the graph, but also the statistics displayed along with the graph, and the lists of top intents and entities. See [Controls](#) for more information.

Pattern limit expanded

When using the **Patterns** field to [define specific patterns for an entity value](#), the pattern (regular expression) is now limited to 512 characters.

2 July 2018

Jump-tos from conditional responses

You can now configure a conditional response to jump directly to another node. See [Conditional responses](#) for more details.

21 June 2018

Language updates for system entities

Dutch and Simplified Chinese language support are now generally available. Dutch language support includes fuzzy matching for misspellings. Traditional Chinese language support includes the availability of [system entities](#) in beta release. See [Supported languages](#) for details.

14 June 2018

Washington, DC data center opens

You can now create Watson Assistant service instances that are hosted in the Washington, DC data center. See [Data centers](#) for more details.

New API authentication process

The Watson Assistant service has a new API authentication process for service instances that are hosted in the following regions:

- Washington, DC (us-east) as of 14 June 2018
- Sydney, Australia (au-syd) as of 7 May 2018

IBM Cloud® is migrating to token-based Identity and Access Management (IAM) authentication.

For new service instances in the regions listed, you use IAM for authentication. You can pass either a bearer token or an API key. Tokens support authenticated requests without embedding service credentials in every call. API keys use basic authentication.

For all new and existing service instances in other regions, you continue to use service credentials (`{username}:{password}`) for authentication.

When you use any of the Watson SDKs, you can pass the API key and let the SDK manage the lifecycle of the tokens. For more information and examples, see [Authentication](#) in the API reference.

If you are not sure which type of authentication to use, view the Watson Assistant credentials by clicking the service instance from the Services section of the [IBM Cloud Resource List](#).

25 May 2018

New sample workspace

The sample workspace that is provided for you to explore or to use as a starting point for your own workspace has changed. The **Car Dashboard** sample was replaced by a **Customer Service** sample. The new sample showcases how to use content catalog intents and other newer features to build a bot. It can answer common questions, such as inquiries about store hours and locations, and illustrates how to use a node with slots to schedule in-store appointments.

HTML rendering was added to Try it out

The "Try it out" pane now renders HTML formatting that is included in response text. Previously, if you included a hypertext link as an HTML anchor tag in a text response, you would see the HTML source in the "Try it out" pane during testing. It used to look like this:

`Contact us at ibm.com.`

Now, the hypertext link is rendered as if on a web page. It is displayed like this:

Contact us at ibm.com.

Remember, you must use the appropriate type of syntax in your responses for the client application to which you will deploy the conversation. Only use HTML syntax if your client application can interpret it properly. Other integration channels might expect other formats.

Deployment changes

The **Test in Slack** option was removed.

11 May 2018

Information security

The documentation includes some new details about data privacy. Read more in [Information security](#).

7 May 2018

Sydney, Australia data center opens

You can now create Watson Assistant service instances that are hosted in the Sydney, Australia data center. See [IBM Cloud global data centers](#) for more details.

4 April 2018

Search dialogs

You can now [search dialog nodes](#) for a given word or phrase.

15 March 2018

Introducing IBM Watson® Assistant

IBM Watson® Conversation has been renamed. It is now called IBM Watson® Assistant. The name change reflects the fact that Watson Assistant is expanding to provide prebuilt content and tools that help you more easily share the virtual assistants you build. Read [this blog post](#) for more details.

New REST APIs and SDKs are available for Watson Assistant

The new APIs are functionally identical to the existing Conversation APIs, which continue to be supported. For more information about the Watson Assistant APIs, see the [API Reference](#).

Dialog enhancements

The following features were added to the dialog tool:

- Simple variable name and value fields are now available that you can use to add context variables or update context variable values. You do not need to open the JSON editor unless you want to. See [Defining a context variable](#) for more details.
- Organize your dialog by using folders to group together related dialog nodes. See [Organizing the dialog with folders](#) for more details.
- Support was added for customizing how each dialog node participates in user-initiated digressions away from the designated dialog flow. See [Digressions](#) for more details.

Search intents and entities

A new search feature has been added that allows you to [search intents](#) for user examples, intent names, or descriptions, or to [search entity](#) values and synonyms.

Content catalogs

The new [content catalogs](#) contain a single category of prebuilt common intents and entities that you can add to your application. For example, most

applications require a general #greeting-type intent that starts a dialog with the user. You can add it from the content catalog rather than building your own.

Enhanced user metrics

The Improve component has been enhanced with additional user metrics and logging statistics. For example, the Overview page includes several new, detailed graphs that summarize interactions between users and your application, the amount of traffic for a given time period, and the intents and entities that were recognized most often in user conversations.

12 March 2018

New date and time methods

Methods were added that make it easier to perform date calculations from the dialog. See [Date and time calculations](#) for more details.

16 February 2018

Dialog node tracing

When you use the "Try it out" pane to test a dialog, a location icon is displayed next to each response. You can click the icon to highlight the path that your assistant traversed through the dialog tree to arrive at the response. See [Building a dialog](#) for details.

New API version

The current API version is now **2018-02-16**. This version introduces the following changes:

- A new `include_audit` parameter is now supported on most GET requests. This is an optional boolean parameter that specifies whether the response should include the audit properties (`created` and `updated` timestamps). The default value is `false`. (If you are using an API version earlier than **2018-02-16**, the default value is `true`.) For more information, see the [API Reference](#).
- Responses from API calls using the new version include only properties with non- `null` values.
- The `output.nodes_visited` and `output.nodes_visited_details` properties of message responses now include nodes with the following types, which were previously omitted:
 - Nodes with `type = response_condition`
 - Nodes with `type = event_handler` and `event_name = input`

9 February 2018

Dutch system entities (Beta)

Dutch language support has been enhanced to include the availability of [System entities](#) in beta release. See [Supported languages](#) for details.

29 January 2018

- The REST API now supports new request parameters:
 - Use the `append` parameter when updating a workspace to indicate whether the new workspace data should be added to the existing data, rather than replacing it. For more information, see [Update workspace](#).
 - Use the `nodes_visited_details` parameter when sending a message to indicate whether the response should include additional diagnostic information about the nodes that were visited during processing of the message. For more information, see [Send message](#).

23 January 2018

Unable to retrieve list of workspaces

If you see this or similar error messages when working in the tooling, it might mean that your session has expired. Log out by choosing **Log out** from

the **User information** icon, and then log back in.

8 December 2017

Log data access across instances (Premium users only)

If you are a Premium user, your premium instances can optionally be configured to allow access to log data from workspaces across your different premium instances.

Copy nodes

You can now duplicate a node to make a copy of it and its children. This feature is helpful if you build a node with useful logic that you want to reuse elsewhere in your dialog. See [Copying a dialog node](#) for more information.

Capture groups in pattern entities

You can identify groups in the regular expression pattern that you define for an entity. Identifying groups is useful if you want to be able to refer to a subsection of the pattern later. For example, your entity might have a regex pattern that captures US phone numbers. If you identify the area code segment of the number pattern as a group, then you can subsequently refer to that group to access just the area code segment of a phone number. See [Defining entities](#) for more information.

6 December 2017

IBM Cloud® Functions integration (Beta)

Call IBM Cloud® Functions (formerly IBM OpenWhisk) actions directly from a dialog node. This feature enables you to, for example, call an action to retrieve weather information from within a dialog node, and then condition on the returned information in the dialog response. Currently, you can call an action from a Cloud Functions instance that is hosted in the US South region from instances that are hosted in the US South region. See [Making programmatic calls from a dialog node](#) for more details.

5 December 2017

Redesigned UI for Intents and Entities

The **Intents** and **Entities** tabs have been redesigned to provide an easier, more efficient workflow when creating and editing entities and intents. See [Defining intents](#) and [Defining entities](#) for information about working with these tabs.

30 November 2017

Eastern Arabic numeral support

Eastern Arabic numerals are now supported in Arabic system entities.

29 November 2017

Improving understanding of user input across workspaces

You can now improve a workspace with utterances that were sent to other workspaces within your instance. For example, you might have multiple versions of production workspaces and development workspaces; you can use the same utterance data to improve any of these workspaces. See [Improving across workspaces](#).

20 November 2017

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 service supports this encoding, and is certified GB18030-compliant.

9 November 2017

Intent examples can directly reference entities

You can now specify an entity reference directly in an intent example. That entity reference, along with all its values or synonyms, is used by the service classifier for training the intent. For more information, see [Entity as example](#) in the [Intents](#) topic.

Currently, you can only directly reference closed entities that you define. You cannot directly reference [pattern entities](#) or [system entities](#).

8 November 2017

New connector tool

You can use the new connector tool to connect your workspace to a Slack or Facebook Messenger app that you own, making it available as a chat bot that Slack or Facebook Messenger users can interact with. This tool is available only for the IBM Cloud US South region.

3 November 2017

Dialog updates

The following updates make it easier for you to build a dialog. (See [Building a dialog](#) for details.)

- You can add a condition to a slot to make it required only if certain conditions are met. For example, you can make a slot that asks for the name of a spouse required only if a previous (required) slot that asks for marital status indicates that the user is married.
- You can now choose **Skip user input** as the next step for a node. When you choose this option, after processing the current node, your assistant jumps directly to the first child node of the current node. This option is similar to the existing *Jump to* next step option, except that it allows for more flexibility. You do not need to specify the exact node to jump to. At run time, your assistant always jumps to whichever node is the first child node, even if the child nodes are reordered or new nodes are added after the next step behavior is defined.
- You can add conditional responses for slots. For both Found and Not found responses, you can customize how your assistant responds based on whether certain conditions are met. This feature enables you to check for possible misinterpretations and correct them before saving the value provided by the user in the slot's context variable. For example, if the slot saves the user's age, and uses `@sys-number` in the *Check for* field to capture it, you can add a condition that checks for numbers over 100, and responds with something like, *Please provide a valid age in years*. See [Adding conditions to Found and Not found responses](#) for more details.
- The interface you use to add conditional responses to a node has been redesigned to make it easier to list each condition and its response. To add node-level conditional responses, click **Customize**, and then enable the **Multiple responses** option.

The **Multiple responses** toggle sets the feature on or off for the node-level response only. It does not control the ability to define conditional responses for a slot. The slot multiple response setting is controlled separately.

- To keep the page where you edit a slot simple, you now select menu options to a.) add a condition that must be met for the slot to be processed, and b.) add conditional responses for the Found and Not found conditions for a slot. Unless you choose to add this extra functionality, the slot condition and multiple responses fields are not displayed, which declutters the page and makes it easier to use.

25 October 2017

Updates to Simplified Chinese

Language support has been enhanced for Simplified Chinese. This includes intent classification improvements using character-level word embeddings, and the availability of system entities. Note that the service learning models may have been updated as part of this enhancement, and when you retrain your model any changes will be applied.

Updates to Spanish

Improvements have been made to Spanish intent classification, for very large datasets.

11 October 2017

Updates to Korean

Language support has been enhanced for Korean. Note that the service learning models may have been updated as part of this enhancement, and when you retrain your model any changes will be applied.

3 October 2017

Pattern-defined entities (Beta)

You can now define specific patterns for an entity, using regular expressions. This can help you identify entities that follow a defined pattern, for example SKU or part numbers, phone numbers, or email addresses. See [Pattern-defined entities](#) for additional details.

- You can add either synonyms or patterns for a single entity value; you cannot add both.
- For each entity value, there can be a maximum of up to 5 patterns.
- Each pattern (regular expression) is limited to 128 characters.
- Importing or exporting via a CSV file does not currently support patterns.
- The REST API does not support direct access to patterns, but you can retrieve or modify patterns using the `/values` endpoint.

Fuzzy matching filtered by dictionary (English only)

An improved version of fuzzy matching for entities is now available, for English. This improvement prevents the capturing of some common, valid English words as fuzzy matches for a given entity. For example, fuzzy matching will not match the entity value `like` to `hike` or `bike`, which are valid English words, but will continue to match examples such as `lkie` or `oike`.

27 September 2017

Condition builder updates

The control that is displayed to help you define a condition in a dialog node has been updated. Enhancements include support for listing available context variable names after you enter the \$ to begin adding a context variable.

31 August 2017

Improve section rollback

The median conversation time metric, and corresponding filters, are being temporarily removed from the Overview page of the Improve section. This removal will prevent the calculation of certain metrics from causing the median conversation time metric, and the conversations over time graph, to display inaccurate information. IBM regrets removing functionality from the tool, but is committed to ensuring that we are communicating accurate information to users.

Dialog node names

You can now assign any name to a dialog node; it does not need to be unique. And you can subsequently change the node name without impacting

how the node is referenced internally. The name you specify is saved as a title attribute of the node in the workspace JSON file and the system uses a unique ID that is stored in the name attribute to reference the node.

23 August 2017

Updates to Korean, Japanese, and Italian

Language support has been enhanced for Korean, Japanese, and Italian. Note that the service learning models may have been updated as part of this enhancement, and when you retrain your model any changes will be applied.

10 August 2017

Accent normalization

In a conversational setting, users may or may not use accents while interacting with the service. As such, an update has been made to the algorithm so that accented and non-accented versions of words are 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, your assistant 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), your assistant 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.

Note: Accent normalization is enabled for Portuguese, Spanish, French, and Czech.

Workspace opt-out flag

The Watson Assistant REST API now supports an opt-out flag for workspaces. This flag indicates that workspace training data such as intents and entities are not to be used by IBM for general service improvements. For more information, see the [API Reference](#).

7 August 2017

Next and last date interpretation

The 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. See the [system entities](#) topic for additional information.

3 August 2017

Fuzzy matching for additional languages (Beta)

Fuzzy matching for entities is now available for additional languages, as noted in the [Supported languages](#) topic.

Partial match (Beta - English only)

Fuzzy matching will now automatically suggest substring-based synonyms present in user-defined entities, and assign a lower confidence score as compared to the exact entity match. See [Fuzzy matching](#) for details.

28 July 2017

Updates

This release includes the following updates:

- When you set bidirectional preferences for the tooling, you can now specify the graphical user interface direction.
- The color scheme of the tooling was updated to be consistent with other Watson services and products.

19 July 2017

REST API now supports access to dialog nodes

The REST API now supports access to dialog nodes. For more information, see the [API Reference](#).

14 July 2017

Slots enhancement

The slots functionality of dialogs was enhanced. For example, a `slot_in_focus` property was added that you can use to define a condition that applies to a single slot only. See [Gathering information with slots](#) for details.

12 July 2017

Support for Czech

Czech language support has been introduced; please see the [Supported languages](#) topic for additional details.

11 July 2017

Test in Slack

You can use the new **Test in Slack** tool to quickly deploy your workspace as a Slack bot user for testing purposes. This tool is available only for the IBM Cloud US South region.

Updates to Arabic

Arabic language support has been enhanced to include absolute scoring per intent, and the ability to mark intents as irrelevant; please see the [Supported languages](#) topic for additional details. Note that the service learning models may have been updated as part of this enhancement, and when you retrain your model any changes will be applied.

23 June 2017

Updates to Korean

Korean language support has been enhanced; please see the [Supported languages](#) topic for additional details. Note that the service learning models may have been updated as part of this enhancement, and when you retrain your model any changes will be applied.

22 June 2017

Introducing slots

It is now easier to collect multiple pieces of information from a user in a single node by adding slots. Previously, you had to create several dialog nodes to cover all the possible combinations of ways that users might provide the information. With slots, you can configure a single node that saves any information that the user provides, and prompts for any required details that the user does not. See [Gathering information with slots](#) for more details.

Simplified dialog tree

The dialog tree has been redesigned to improve its usability. The tree view is more compact so it is easier to see where you are within it. And the links between nodes are represented in a way that makes it easier to understand the relationships between the nodes.

21 June 2017

Arabic support

Language support for Arabic is now generally available. For details, see [Configuring bidirectional languages](#).

Language updates

The service algorithms have been updated to improve overall language support. See the [Supported languages](#) topic for details.

16 June 2017

Recommendations (Beta - Premium users only)

The Improve panel also includes a **Recommendations** page that recommends ways to improve your system by analyzing the conversations that users have with your chatbot, and taking into account your system's current training data and response certainty.

14 June 2017

Fuzzy matching for additional languages (Beta)

Fuzzy matching for entities is now available for additional languages, as noted in the [Supported languages](#) topic. You can turn on fuzzy matching per entity to improve the ability of your assistant to recognize terms in user input with syntax that is similar to the entity, without requiring an exact match. 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. See [Fuzzy matching](#) for details.

13 June 2017

User conversations

The Improve panel now includes a **User conversations** page, which provides a list of user interactions with your chatbot that can be filtered by keyword, intent, entity, or number of days. You can open individual conversations to correct intents, or to add entity values or synonyms.

Regex change

The regular expressions that are supported by SpEL functions like find, matches, extract, replaceFirst, replaceAll and split have changed. A group of regular expression constructs are no longer allowed, including look-ahead, look-behind, possessive repetition and backreference constructs. This change was necessary to avoid a security exposure in the original regular expression library.

12 June 2017

Updates

This release includes the following updates:

- The maximum number of workspaces that you can create with the **Lite** plan (formerly named the Free plan) changed from 3 to 5.
- You can now assign any name to a dialog node; it does not need to be unique. And you can subsequently change the node name without impacting how the node is referenced internally. The name you specify is treated as an alias and the system uses its own internal identifier to reference the node.
- You can no longer change the language of a workspace after you create it by editing the workspace details. If you need to change the language, you can export the workspace as a JSON file, update the language property, and then import the JSON file as a new workspace.

6 June 2017

Learn

A new *Learn about* page is available that provides getting started information and links to service documentation and other useful resources. To open the page, click the icon in the page header.

Bulk export and delete

You can now simultaneously export a number of intents or entities to a CSV file, so you can then import and reuse them for another application. You can also simultaneously select a number of entities or intents for deletion in bulk.

Updates to Korean

Korean tokenizers have been updated to address informal language support. IBM continues to work on improvements to entity recognition and classification.

Emoji support

Emojis added to intent examples, or as entity values, will now be correctly classified/extracted.

Only emojis that are included in your training data will be correctly and consistently identified; emoji support may not correctly classify similar emojis with different color tones or other variations.

Entity stemming (Beta - English only)

The fuzzy matching beta feature recognizes entities and matches them based on the stem form of the entity value. For example, this feature correctly recognizes **bananas** as being similar to **banana**, and **run** being similar to **running** as they share a common stem form. For more information, see [Fuzzy matching](#).

Workspace import progress

When you import a workspace from a JSON file, a tile for the workspace is displayed immediately, in which information about the progress of the import is displayed.

Reduced training time

Multiple models are now trained in parallel, which noticeably reduces the training time for large workspaces.

26 May 2017

New API version

The current API version is now **2017-05-26**. This version introduces the following changes:

- The schema of `ErrorResponse` objects has changed. This change affects all endpoints and methods. For more information, see the [API Reference](#).
- The internal schema used to represent dialog nodes in exported workspace JSON has changed. If you use the **2017-05-26** API to import a workspace that was exported using an earlier version, some dialog nodes might not import correctly. For best results, always import a workspace using the same version that was used to export it.

25 May 2017

Manage context variables

You can now manage context variables in the "Try it out" pane. Click the **Manage context** link to open a new pane where you can set and check the values of context variables as you test the dialog. See [Testing your dialog](#) for more information.

16 May 2017

Updates

This release includes the following updates:

- A **Car Dashboard** sample workspace is now available when you open the tool. To use the sample as a starting point for your own workspace, edit the workspace. If you want to use it for multiple workspaces, then duplicate it instead. The sample workspace does not count toward your subscription workspace total unless you use it.
- It is now easier to navigate the tool. The navigation menu options are available from the side of the main page instead of the top. In the page header, breadcrumb links display that show you where you are. You can now switch between service instances from the Workspaces page. To get there quickly, click **Back to workspaces** from the navigation menu. If you have multiple service instances, the name of the current instance is displayed. You can click the **Change** link beside it to choose another instance.
- When you create a dialog, two nodes are now added to it for you: 1) a **Welcome** node at the start the dialog tree that contains the greeting to display to the user and 2) an **Anything else** node at the end of the tree that catches any user inquiries that are not recognized by other nodes in the dialog and responds to them. See [Creating a dialog](#) for more details.
- When you are testing a dialog in the "Try it out" pane, you can now find and resubmit a recent test utterance by pressing the Up key to cycle through your previous inputs.
- Experimental Korean language support for 5 system entities (`@sys-date`, `@sys-time`, `@sys-currency`, `@sys-number`, `@sys-percentage`) is now available. There are known issues for some of the numeric entities, and limited support for informal language input.
- An Overview page is available from the Improve tab. The page provides a summary of interactions with your bot. You can view the amount of traffic for a given time period, as well as the intents and entities that were recognized most often in user conversations. For additional information, see [Using the Overview page](#).

27 April 2017

System entities

The following system entities are now available as beta features in English only:

- sys-location: Recognizes references to locations, such as towns, cities, and countries, in user utterances.
- sys-person: Recognizes references to people's names, first and last, in user utterances.

For more information, see the [System entities reference](#).

Fuzzy matching for entities

Fuzzy matching for entities is a beta feature that is now available in English. You can turn on fuzzy matching per entity to improve the ability of your assistant to recognize terms in user input with syntax that is similar to the entity, without requiring an exact match. The feature is able to map user input to the appropriate corresponding entity despite the presence of misspellings or slight syntactical differences. For examples, 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. See [Defining entities](#) and search for **Fuzzy Matching** for details.

18 April 2017

Updates

This release includes the following updates:

- The REST API now supports access to the following resources:

- entities
- entity values
- entity value synonyms
- logs

For more information, see the [API Reference](#).

- The behavior of the /messages **POST** method has changed the handling of entities and intents specified as part of the message input:
 - If you specify intents on input, your assistant uses the intents you specify, but uses natural language processing to detect entities in the user input.
 - If you specify entities on input, your assistant uses the entities you specify, but uses natural language processing to detect intents in the user input.

The behavior has not changed for messages that specify both intents and entities, or for messages that specify neither.

- The option to mark user input as irrelevant is now available for all supported languages. This is a beta feature.
- A new Credentials tab provides a single place where you can find all of the information you need for connecting your application to a workspace, as well as other deployment options. To access the Credentials tab for your workspace, click the icon and select **Credentials**.

9 March 2017

REST API updates

The REST API now supports access to the following resources:

- workspaces
- intents
- examples
- counterexamples

For more information, see the [API Reference](#).

7 March 2017

Intent name restrictions

The use of `.` or `..` as an intent name causes problems and is no longer supported. You cannot rename or delete an intent with this name; to change the name, export your intents to a file, rename the intent in the file, and import the updated file into your workspace. Paying customers can contact support for a database change.

1 March 2017

System entities are now enabled in German

System entities are now enabled in German.

22 February 2017

Messages are now limited to 2,048 characters

Messages are now limited to 2,048 characters.

3 February 2017

Updates

This release includes the following updates:

- We changed how intents are scored and added the ability to mark input as irrelevant to your application. For details, see [Defining intents](#) and search for **Mark as irrelevant**.
- This release introduced a major change to the workspace. To benefit from the changes, you must manually upgrade your workspace.
- The processing of **Jump to** actions changed to prevent loops that can occur under certain conditions. Previously, if you jumped to the condition of a node and neither that node nor any of its peer nodes had a condition that was evaluated as true, the system would jump to the root-level node and look for a node whose condition matched the input. In some situations this processing created a loop, which prevented the dialog from progressing.

Under the new process, if neither the target node nor its peers is evaluated as true, the dialog turn is ended. To reimplement the old model, add a final peer node with a condition of **true**. In the response, use a **Jump to** action that targets the condition of the first node at the root level of your dialog tree.

11 January 2017

Customize node titles

In this release, you can customize node titles in dialog.

22 December 2016

Node title section

In this release, dialog nodes display a new section for **node title**. The ability to customize the **node title** is not available. When collapsed, the **node title** displays the **node condition** of the dialog node. If there is not a **node condition**, "Untitled Node" is displayed as the title.

19 December 2016

Dialog editor UI changes

Several changes make the dialog editor easier and more intuitive to use:

- A larger editing view makes it easier to view all the details of a node as you work on it.
- A node can contain multiple responses, each triggered by a separate condition. For more information see [Multiple responses](#).

5 December 2016

Updates

This release includes the following updates:

- New languages are supported, all in Experimental mode: German, Traditional Chinese, Simplified Chinese, and Dutch.
- Two new system entities are available: @sys-date and @sys-time. For details, see [System entities](#).

21 October 2016

Updates

This release includes the following updates:

- The service now provides system entities, which are common entities that can be used across any use case. For details, see [Defining entities](#) and search for **Enabling system entities**.
- You can now view a history of conversations with users on the Improve page. You can use this to understand your bot's behavior. For details, see [Improving your skill](#).
- You can now import entities from a comma-separated value (CSV) file, which helps with when you have a large number of entities. For details, see [Defining entities](#) and search for **Importing entities**.

20 September 2016

New version 2016-09-20

To take advantage of the changes in a new version, change the value of the `version` parameter to the new date. If you're not ready to update to this version, don't change your version date.

- version **2016-09-20**: `dialog_stack` changed from an array of strings to an array of JSON objects.

29 August 2016

Updates

This release includes the following updates:

- You can move dialog nodes from one branch to another, as siblings or peers. For details, see [Moving a dialog node](#).
- You can expand the JSON editor window.
- You can view chat logs of your bot's conversations to help you understand its behavior. You can filter by intents, entities, date, and time. For details, see [Improving your skill](#)

11 July 2016

General Availability

This General Availability release enables you to work with entities and dialogs to create a fully functioning bot.

18 May 2016

Experimental release

This Experimental release introduces the user interface and enables you to work with workspaces, intents, and examples.

Web chat release notes

Release notes for the classic Watson Assistant experience have moved

As of 12 June 2023, these release notes are no longer being updated. All release notes for both the new Watson Assistant and the classic Watson Assistant can be found here: [Web chat release notes](#).

7.4.0

Release date: 12 June 2023

- Added CSS variables for customizing the launcher. For more information, see [instance.updateCSSVariables](#).

7.3.0

Release date: 30 May 2023

- Released a beta of a contact center integration for Genesys Web Messenger, which currently only includes user information strings in English. For more information, see [Genesys Web Messenger](#).
- Added beta support for file sharing with custom service desk integrations, which currently only includes user information strings in English. For more information, see [Custom service desks](#).

7.2.2

Release date: 1 May 2023

- Bug fixes.

7.2.1

Release date: 24 April 2023

- Bug fixes.

7.2.0

Release date: 10 April 2023

- Added support for inline iframe responses. For more information, see [Response types reference](#).
- Redesigned the agent conversation experience.
- Bug fixes.

7.1.1

Release date: 13 February 2023

- New journey events:** The new `tour:start`, `tour:end`, and `tour:step` events provide details about the user's progress through a journey (also known as a *tour*). These events can be used to navigate to a specific page when the user starts a journey or reaches a certain step, or to show a survey after a journey ends.
- New journey instance methods:** The new `tours` object supports instance methods that provide better control over journeys. You can use these methods to start or end a journey, or to automatically navigate through a journey in response to user actions.
- Added journey strings to the language pack :** New strings for journeys have been added to the language pack. You can modify the strings in the language pack by using the `updateLanguagePack()` instance method. For more information about language packs, see [Languages](#).

For more information about the journeys beta feature, see [Guiding customers with journeys](#).

7.1.0

Release date: 17 January 2023

- Updated Zendesk agent app :** The agent app for Zendesk has been updated for compatibility with Zendesk workspaces.
- In the service desk starter kits, the instance of the web chat integration has been added to the `serviceDeskFactory` parameters to make it accessible to custom service desk implementations.
- New instance methods:** The new `elements.getMessageInput()` and `elements.getHomeScreenInput()` instance methods enable access to the input fields used by the customer to send messages. You can use these methods to change the input or to take action as the user is typing (for example, to implement for a type-ahead feature).

- **New event:** The new `agent:pre:sessionHistory` event enables you to filter potential PII from messages sent from a customer or service desk agent before the messages are sent to the assistant for storage in the session history.
- **New property in web chat state object:** In the object returned from the `getState()` instance method, the new `isDebugEnabled` property indicates whether the web chat debug flag is set to `true`.

7.0.0

Release date: 5 December 2022

- **Streamlined live agent handoff:** The live agent handoff experience has been streamlined and simplified. Instead of opening the live agent chat in a separate window, the web chat now shows the live agent entering the conversation in the same window.
Because of this change, the `updateCustomMenuOptions` instance method has changed to reflect the fact that there is now only a single view, with a single list of custom menu options. If you want to customize menu options only during a live agent chat, you can subscribe to the `agent:pre:startChat` and `agent:endChat` events to trigger your customizations.
- **agent:endChat changes:** The `agent:endChat` event now also fires if the customer cancels a live agent request before the agent has joined. If you want to show a post-chat form only after a live agent chat, you can use the new `requestCancelled` flag on the event to determine whether the request was canceled.
- **New configuration options:** The following new options are available in the configuration object:
 - `serviceDesk.availabilityTimeoutSeconds`: Specifies how long the web chat waits for an available agent before automatically canceling the live agent request.
 - `serviceDesk.disableAgentSessionHistory`: Disables storage of live agent chats in the session history. If this option is set to `true`, live agent chat history is not stored; this means that if the web chat is reloaded, the live agent chat history is lost.

For more information, see [Service desk options](#).

- **elements instance property:** A new `elements` instance property provides methods you can use to apply CSS styles to individual elements used by the web chat. (Currently, only the main window is supported.)
- **skip_card option for journeys:** The journeys beta feature has been updated to support a new `skip_card` property. You can use this property to start a journey immediately without waiting for the customer to click the introductory card, or even to start a journey from your website without opening the web chat at all. For more information, see [Guiding customers with journeys](#).
- **Path changes:** Some internal paths used for communication with the assistant have changed. If you have firewall or proxy rules that are configured to allow specific paths, you might need to update your configuration to allow the following paths:
 - `/<SUBSCRIPTION_ID>/chat/<INTEGRATION_ID>/config`
 - `/<SUBSCRIPTION_ID>/chat/<INTEGRATION_ID>/message`

6.9.0

Release date: 14 November 2022

- You can now create *journeys* to guide your customers through tasks they can already complete on your website. A journey is an interactive, multipart response that can combine text, video, and images, presented in sequence in a small window superimposed over your website.

Journeys are available as a beta feature. For more information, see [Guiding customers with journeys](#).

6.8.1

Release date: 7 November 2022

- Bug fixes.

6.8.0

Release date: 31 October 2022

- Added support for sending pre-chat information to the Salesforce integration.

6.7.0

Release date: 10 October 2022

- **New updateIsTypingCounter() method:** The new `updateIsTypingCounter()` instance method updates the counter that determines whether the typing indicator is displayed. For more information, see [instance.updateIsTypingCounter\(\)](#).
- **New updateBotUnreadIndicatorVisibility() method:** The new `updateBotUnreadIndicatorVisibility()` instance method specifies whether the unread indicator on the launcher icon is shown or hidden. For more information, see

[`instance.updateBotUnreadIndicatorVisibility\(\)`](#).

- Connect to Agent and custom cards now have rounded corners.
- Bug fixes.

6.6.2

Release date: 15 August 2022

- Bug fixes

6.6.1

Release date: 8 August 2022

- The `servers` property now supports a new `webChatScriptPrefix` option. Use this property to configure a proxy between your users' browsers and the IBM Cloud servers that host the web chat JavaScript code. For more information, see [Setting up a proxy](#).

6.6.0

Release date: 25 July 2022

- A new `servers` property is now available in the web chat configuration options. You can use this property to set up a proxy between your users' browsers and Watson Assistant. For more information, see [Setting up a proxy](#).

6.5.2

Release date: 11 July 2022

- Bug fix for `date` response type.

6.5.1

Release date: 15 June 2022

- Bug fix for the Zendesk integration.

6.5.0

Release date: 6 June 2022

- **New agent events:** New events are now fired by the web chat when messages are sent or received during a conversation with a human agent using a service desk integration. For more information, see [Agent events summary](#).
- Bug fixes.

6.4.1

Release date: 16 May 2022

- **Minimum size:** The minimum allowed size of the rendered web chat window has been reduced to satisfy the accessibility requirements defined by the [Web Content Accessibility Guidelines \(WCAG\) 2.1](#) standard.
- **Pop-up windows and tabs from iframes :** The web chat now allows pop-up windows and new tabs to be opened from content rendered inside `iframe` responses.
- **Faster responses:** Responses received from the assistant are now displayed more quickly and without a `...` typing indicator.

6.4.0

Release date: 18 April 2022

- **Date picker:** If you configure a step to collect a `Date` customer response, the step now uses the new `date` response type to request that the web chat display a graphical date picker the customer can use to select a date, as an alternative to typing the date in the input field. Existing steps do not automatically inherit this behavior; if you want to use the date picker, you must delete the existing Date response and then re-add it.
- **Skip "connect to agent" card :** A new `serviceDesk.skipConnectAgentCard` configuration option is available. If this option is enabled, the web chat immediately connects to an agent when it receives a `Connect to agent` response, without first displaying a card and waiting for the user to click.
- **Close button:** A new `showCloseAndRestartButton` configuration option specifies whether the web chat interface shows an X (Close) button in addition to the existing - (Minimize) button. A customer can click this button to close the web chat, end the conversation, and end any conversation with a human agent. The chat transcript is also cleared, but any transcript of a conversation with a human agent is preserved.

6.3.0

Release date: 24 March 2022

- **Search cards:** Search cards have a new design.
- **New `restartConversation()` method:** The new `restartConversation()` instance method restarts the conversation with the assistant by clearing the web chat transcript and starting a new session. It also fires two new events (`pre:restartConversation` and `restartConversation`).
For more information, see [instance.restartConversation\(\)](#), [pre:restartConversation](#), and [restartConversation](#).
- **New `agentEndConversation()` method:** The new `agentEndConversation()` instance method immediately ends the conversation with a human agent without requesting confirmation from the user. For more information, see [instance.agentEndConversation\(\)](#).
- Bug fixes.

6.2.0

Release date: 7 March 2022

- **Navigation:** The web chat has been updated with new navigation features. For example, new “Back” and “Minimize” buttons make it easier to navigate between the home screen, the chat view, and panels. A new customizable drop-down menu appears near the avatar in both the assistant and agent chat views. This new functionality allows you to easily add new options to the menu of the web chat (for more information, see [updateCustomMenuOptions](#)).

The experience of connecting to a human agent has also been improved to make it clearer how a user requests an agent, returns to chatting with the assistant, and ends the conversation.

The animations for the web chat panels have also been improved to make the whole experience more seamless and cohesive.

- **Launcher:** The new web chat launcher bounces on two different occasions to attract attention and encourage customer engagement. For more information, see [Launcher appearance and behavior](#).
- **Launcher:** Support has been added to control what text the launcher greets the user with, and when the greeting message is shown. For more information, see [Launcher](#).
- **Locale:** The web chat no longer sets the locale in the system context to `en-us` when no locale is configured. The locale is set in the system context only if it is configured for web chat.
- Bug fixes.

6.1.0

Release date: 7 February 2022

- Updated to support internal changes to the preview link feature.

6.0.1

Release date: 24 January 2022

- Bug fix for the disclaimer. For more information about the disclaimer, see [Configuration options object](#).

6.0.0

Release date: 19 January 2022

- **API version:** The web chat now uses the `2021-11-27` version of the Watson Assistant API. Previously it used the `2020-09-24` API version. For information about API changes that have been introduced since the `2020-09-24` version, see the release notes for [27 November 2021](#) and [16 July 2021](#).
- **Launcher:** The new web chat launcher welcomes and engages customers so they know where to find help if they need it. For more information, see [Launcher appearance and behavior](#).
- **Home screen:** The web chat home screen has been updated to have a more modern look. For more information about the home screen, see [Configuring the home screen](#).
- **Agent events:** New events are now fired by the web chat when interacting with a human agent using a service desk integration. If you are using a custom service desk integration based on the [starter kit](#), you can use these events to create a pre-chat form before the agent escalation occurs, to create a post-chat form after the agent conversation ends, or to specify what happens if an agent isn't available (like create a ticket submission form). For more information, see [Agent events summary](#).
- **Markdown support:** The web chat now fully supports common Markdown formatting in messages received from an assistant. You might need to

review existing assistant output that contains strings that might be recognized as Markdown. (For example, a line of text that begins with a greater-than (>) character is interpreted as a block quote.)

- **Time zone:** The time zone set in the context by the web chat no longer overrides any time zone set by the assistant.
- **Locale:** Any locale configured for the web chat is now sent to the assistant as part of the context.
- **Window open events:** The `window:pre:open` and `window:open` events now fire any time the chat window is opened, regardless of the reason. In previous releases, these events only fired if the window was opened by the customer clicking on the built-in launcher. Other methods of opening the chat window, such as session history or custom launchers, did not fire these events.

The event data passed to the listener has a new `reason` property that indicates the reason the window was opened. If you want to preserve the previous behavior, you can modify your handler to check this property:

```
$ instance.on({ type: "window:open", handler: event => {
  if (event.data.reason === 'default_launcher') {
    // Previous code.
  }
});
```

For more information, see [Window open reasons](#).

- **hideCloseButton property renamed:** The `hideCloseButton` property for custom panels has been renamed `hideBackButton`. The behavior of the property has not changed. For more information, see [customPanel.open\(\)](#).

5.1.2

Release date: 11 December 2021

- Bug fix for Salesforce integration.

5.1.1

Release date: 5 November 2021

- **"User is typing" support:** The web chat now supports displaying the "user is typing" message for service desks. This feature is supported for the Salesforce and Zendesk integrations, as well as any [starter kit](#) integration that implements it.
- Bug fixes.

5.1.0

Release date: 28 October 2021

- **Custom Panels:** The web chat now supports customizable panels you can use to display any custom HTML content (for example, a feedback form or a multistep process). Your code can use instance methods to dynamically populate a custom panel, as well as open and close it. For more information, see [Custom Panels](#).

5.0.2

Release date: 4 October 2021

- A [new tutorial](#) is now available that shows how to use Carbon components to customize user-defined responses and writeable elements.
- Bug fixes.

5.0.1

Release date: 20 September 2021

- Bug fixes.

5.0.0

Release date: 16 September 2021

- **New response types:** The web chat now supports the new `video`, `audio`, and `iframe` response types. For more information about these response types, see [Rich responses](#).
- **Link to start web chat:** You can now create a set of HTML links that go directly to your web chat and start conversations on specific topics. For example, you might want to send an email inviting customers to update their account information; you can include a link that opens the web chat on your site and sends the initial message `I want to update my account`. For more information, see [Creating links to web chat](#).
- **CSS improvements:** Support for CSS styles has been improved to change the way the web chat resets styles in areas where you can include your own custom content, such as user-defined responses and writeable elements. The new approach better protects custom content from accidental style

overrides. For more information about custom content and CSS classes, see [Theming & custom content](#).



Note: If you have any custom content (such as user-defined responses or writeable elements), verify that any styling is still rendering as you expect. Consider using the new [ibm-web-chat--default-styles class](#) to maintain consistency with the web chat default styles.

- **Support for Carbon components:** As part of the new styling support, you can now use [Carbon components](#) in user-defined responses and web chat writeable elements. These components will inherit any theming customizations you have made to the web chat.
- **New embedded script:** The embedded script you use to add the web chat to your website has been updated to avoid unexpected code changes when you lock on to a web chat version. (For more information about web chat versioning, see [Versioning](#).) The previous version of the script will continue to work but is now deprecated. If you want to upgrade your existing web chat deployments to use the new script, copy the updated code snippet from the **Embed** tab of the web chat integration settings. (Remember to reapply any customizations you have made.)
- **Removal of deprecated methods and events :**
 - The `error` event has been replaced by the `onError` method in the [configuration object](#).
 - The `getID` method has been removed.
- Microsoft Internet Explorer 11 is no longer a supported browser.

4.5.1

Release date: 30 August 2021

- Bug fixes for the interactive launcher beta feature. (For more information about this feature, see the `launcherBeta` configuration option at [Configuration options object](#).)

4.5.0

Release date: 29 July 2021

- A new `scrollToMessage` method is available for scrolling the web chat view to a specified message in the chat history. For more information, see [instance.scrollToMessage\(\)](#).
- A new `pre:open` event is available. This event is fired when the web chat window is opened, but before the welcome message or chat history are loaded. For more information, see [window:pre:open](#).
- A new chat history widget is available for embedding in service desk agent UIs. This new widget is based on a read-only view of the standard web chat widget. For information about using the new chat history widget in integrations built using the starter kit, see [Embedded agent application](#).

4.4.1

Release date: 6 July 2021

- Bug fixes.

4.4.0

Release date: 25 June 2021

- Bug fixes.

4.3.0

Release date: 7 June 2021

- **Search suggestions:** If a search skill is configured for your assistant, the suggestions include a new `View related content` section. This section contains search results that are relevant to the user input.
- **Focus trap:** A new `enableFocusTrap` option enables maintaining focus inside the web chat widget while it is open. For more information, see [Configuration options object](#).

4.2.1

Release date: 6 May 2021

- **Service URLs updated:** The URLs used by the web chat to communicate with the Assistant service have been updated to remove the dependency on the deprecated `watsonplatform.net` domain. This change applies retroactively to version 3.3.0 and all subsequent web chat releases. Make sure the system that hosts the web chat widget has access to the new URL; for more information, see [Deploy your assistant in production](#).

4.2.0

Release date: 27 April 2021

- **Conversation starters in suggestions :** The conversation starters you configure for the home screen are now shown as suggestions. If suggestions

are enabled, the conversation starters appear in a new section titled **People are also interested in**. This provides an easy way for customers to change the subject or start the conversation over. For more information about the suggestions feature, see [Showing more suggestions](#). For more information about the home screen, see [Configuring the home screen](#).

- **onError callback:** The new `onError` callback option in the web chat configuration enables you to specify a callback function that is called if errors occur in the web chat. This makes it possible for you to handle any errors or outages that occur with the web chat. For more information, see [Listening for errors](#).
- **Session ID available in widget state:** The state information returned by the `getState()` instance method now includes the session ID for the current conversation. For more information, see [instance.getState\(\)](#).
- **IBM watermark:** The web chat can now display a **Built with IBM Watson** watermark to users. This watermark is always enabled for any new web chat integrations on Lite plans. For more information, see [Create a web chat instance to add to your website](#).
- **Fixes to rendering of list items:** The rendering of HTML list items in the web chat widget has been updated.

4.1.0

Release date: 8 April 2021

- **Home screen now generally available:** Ease your customers into the conversation by adding a home screen to your web chat window. The home screen greets your customers and shows conversation starter messages that customers can click to easily start chatting with the assistant. For more information about the home screen feature, see [Configuring the home screen](#).
- **Home screen enabled by default:** The home screen feature is now enabled by default for all new web chat deployments.
- **Home screen context support:** You can now access context variables from the home screen. Note that initial context must be set using a `conversation_start` node. For more information, see [Starting the conversation](#).

4.0.0

Release date: 16 March 2021

- **Session history now generally available:** Session history allows your web chats to maintain conversation history and context when users refresh a page or change to a different page on the same website. It is enabled by default. For more information about this feature, see [Session history](#).

Session history persists within only one browser tab, not across multiple tabs. The dialog provides an option for links to open in a new tab or the same tab. See [this example](#) for more information on how to format links to open in the same tab.

Session history saves changes that are made to messages with the `pre:receive event` so that messages still look the same on rerender. This data is only saved for the length of the session. If you prefer to discard the data, set `event.updateHistory = false;` so the message is rerendered without the changes that were made in the pre:receive event.

`instance.updateHistoryUserDefined()` provides a way to save state for any message response. With the state saved, a response can be rerendered with the same state. This saved state is available in the `history.user_defined` section of the message response on reload. The data is saved during the user session. When the session expires, the data is discarded.

Two new history events, `history:begin` and `history:end` announce the beginning and end of the history of a reloaded session. These events can be used to view the messages that are being reloaded. The history:begin event allows you to edit the messages before they are displayed.

See this example for more information on saving the state of `customResponse` types in session history.

- **Channel switching:** You can now create a dialog response type to functionally generate a connect-to-agent response within channels other than web chat. If a user is in a channel such as Slack or Facebook, they can trigger a channel transfer response type. The user receives a link that forwards them to your organization's website where a connection to an agent response can be started within web chat. For more information, see [Adding a Channel transfer response type](#).

3.5.0

Release date: 17 February 2021

- **Session history (beta):** Web chat session history (beta) is now available. This feature makes it possible to maintain conversation history and context when customers refresh the page or navigate to a different page on the same website. For more information about this beta feature, see [Session history \(beta\)](#).

3.4.1

Release date: 2 February 2021

- Made an accessibility enhancement to the chat history. Now, you can use keys to navigate the messages by clicking the chat history, pressing Enter, and then using the arrow keys to move from one message to the next.
- The `instance.updateAssistantInputFieldVisibility()` method was added. You can use it to hide or show the text input field. For example, you might use the `pre:receive` event to check whether an options response type is being returned and if so, hide the text field so the

user is forced to pick one of the available options only.

- The `instance.getState()` method was added. You can use it to check for specific conditions, such as `isWebChatOpen`, before you perform an action that might rely on the condition being true.

For more information about the new methods, see [Instance methods](#).

3.3.2

Release date: 17 December 2020

- Addressed accessibility issues.

3.3.1

Release date: 3 December 2020

- The translated strings in the [language files](#) were revised and improved.
- An error message is shown now if a Java Web Token (JWT) that is provided with an incoming message is invalid. If the first JWT fails when the web chat opens, an error message is displayed in place of the web chat window that says, **There was an error communicating with Watson Assistant**. If the initial JWT is valid, but the token for a subsequent message is invalid, a more discreet error message is displayed in response to the insecure input.
- Bug fixes.

3.3.0

Release date: 23 November 2020

- Added support for passing contextual information to a service desk agent from web chat.
- You can now customize a `user_defined` response type. For more information, see the [Custom response type tutorial](#).
- Bug fixes.

3.2.1

Release date: 2 November 2020

- **Bug fix:** Fixing a bug that prevented the web chat integration preview from working after security was enabled.

3.2.0

Release date: 26 October 2020

- **Security improvement:** If you enable security, you no longer need to include the `identityToken` property when the web chat is loaded on a web page. If a token is not initially provided, the existing `identityTokenExpired` event will be fired when the web chat is first opened to obtain one from your handler.
- **Starter kit update:** The starter kit now allow you to customize the timeout that occurs when the web chat integration checks whether any service desk agents are online.

3.1.1

Release date: 22 October 2020

- **Accessibility improvement:** Changed how the announcement text is generated to prevent announcements from being duplicated. Announcement text is hidden text that is provided for use by screen readers to indicate when dynamic web page changes occur.

3.1.0

Release date: 8 October 2020

- **Suggestions now allow for trial and error:** If customers select a suggestion and find that the response is not helpful, they can open the suggestions list again and try a different suggestion.

3.0.0

Release date: 22 September 2020

- **Choose when a link to support is included in suggestions :** The Suggestions beta feature has moved to its own tab. Now you can enable suggestions even if your web chat is not set up to connect to a service desk solution. That's because now you can control if and when the option to connect to customer support is available from the suggestions list. For more information, see [Showing more suggestions](#).
- **Search result format change :** To support the ability to show more than 3 search results in a response, the search skill response type format changed. If you are using `pre:receive` or `receive` handlers to process search results, you might need to update your code. The `results`

property was replaced by the `primary_results` and `additional_results` properties. For more information about the new search skill response type format, see the [API reference](#).

- **Language pack key change:** Due to improvements that were made to allow you to specify separate chat transfer messages for situations where agents are available and unavailable, the [language source file](#) was updated. The `agent_chatDescription` was renamed to `default_agent_availableMessage` and another key (`default_agent_unavailableMessage`) was added. If you defined a custom string for the `agent_chatDescription` key, you must modify your code to reflect this change. For more information about the new availability messages and how they are used, see [Adding a Connect to human agent response type](#).

2.4.0

Release date: 2 September 2020

- **Add a home screen:** Ease your customers into the conversation by adding a home screen to your web chat window. The home screen greets your customers and shows conversation starter messages that customers can click to submit to the assistant. For more information about this beta feature, see [Adding a home screen](#).

2.3.0

Release date: 10 August 2020

- **Introducing Suggestions:** Enable this beta feature to show a helper icon in the chat that customers can click to see alternate topics or to connect to an agent. For more information, see [Showing more suggestions](#).
- **Search results are now expandable:** When search results are displayed in the web chat, users can click **Show more** to see more of the search result text.

2.2.0

Release date: 29 July 2020

- **Introduced the `instance.writeableElements()` method:** The `instance.writeableElements()` method gives you zones in the web chat user interface where you can embed your own content. For example, you can add content to the end of the header where it is displayed even as the chat content changes. Or you can add custom content that is displayed before the welcome node. For more information, see [Instance methods](#).
- **Gave the `Send` button a new look**

Changed the icon from  to .
- **Securing your web chat is easier:** You no longer need to specify the `acr` claim when you define the JWT. The Authentication Context Class reference is managed by the web chat automatically.
- Improved the quality of non-English translations.
- Made a few minor bug fixes.

2.1.2

Release date: 2 July 2020

- **Fixed a loading issue:** Addressed an issue that prevented the web chat from loading properly for some deployments with short JWT expiration claims.

2.1.1

Release date: 1 July 2020

- **Service desk agent initials are displayed:** When the web chat transfers a user to a service desk agent, the agent's avatar is displayed in the chat window to identify messages sent from the service desk agent. If the agent does not have an avatar, the first initial of the agent's given name is displayed instead.

2.0

Release date: 16 June 2020

- **Versioning was added to web chat:** For more information, see [Versioning](#).
- **Added bidirectional support:** You can now use the `direction` parameter to choose whether to show text and elements in the web chat in right-to-left or left-to-right order. For more information, see [Configuration](#).
- **Introduced the `instance.destroySession()` method:** The `instance.destroySession()` method removes all cookie and browser storage that is related to the current web chat session. You can use this method as part of your website logout sequence. For more information, see [Instance methods](#).

1.5.3

Release date: 14 April 2020

- **The Font family field was removed from the configuration page :** The text that is displayed in the chat window uses the fonts: `IBMPlexSans`, `Arial`, `Helvetica`, `sans-serif`. If you want to use a different set of fonts, you can customize the CSS for your web chat implementation. For more information, see [Theming](#).
- When your implementation does not specify a unique user ID, the web chat adds a first party cookie with a generated anonymous ID to use to identify the unique user. The generated cookie now expires after 45 days. For more information, see [GDPR and cookie policies](#).

1.5.2

Release date: 2 April 2020

- **Introduced the learningOptOut parameter:** You can add the `learningOptOut` parameter and set it to `true` to add the `X-Watson-Learning-Opt-Out` header to each `/message` request that originates from the Web Chat. For more information about the header, see [Data collection](#). For more information about the parameter, see [Configuration](#).

1.4.0

Release date: 20 March 2020

- **Customize the CSS theme:** For more information, see [Theming](#).
- **Shadow DOM is no longer used:** When you use custom response types or HTML in your dialog, you can apply CSS styles that are defined in your web page to the assistant's response. To override any default styling in the web chat, you must specify the `!important` modifier in your CSS. For more information, see [Rendering response types](#).



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

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 dialog skill that you created, and add nodes to the simple dialog that you built as part of the getting started exercise.

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 cancellations

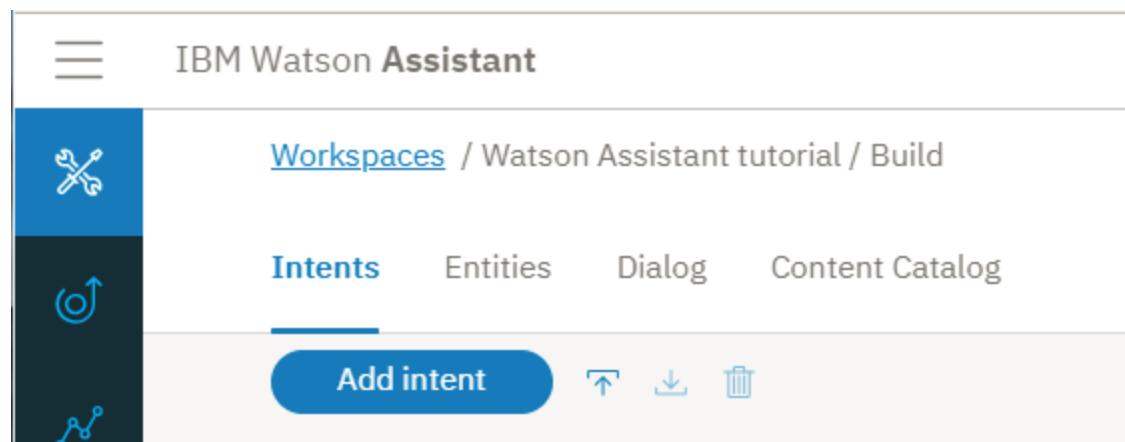
You'll start by creating intents that represent these subjects, and then build a dialog that responds to user questions about them.

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. From the **Intents** tab, click **Create intent**.



2. Enter **about_restaurant** in the *Intent name* field, and then click **Create intent**.

The screenshot shows the Watson Assistant interface for creating a new intent. The top bar has a back arrow, the title 'Create new intent', and buttons for download, delete, and 'Try it'. On the left is a sidebar with four sections: intents (selected), entities, actions, and dialog. The main area shows the 'Intent name' field containing '#about_restaurant', a 'Description' field with placeholder text, and a 'Create intent' button. Below this, a message says 'No examples yet.' and 'Train your virtual assistant with this intent by adding unique examples of what your users would say.'

3. 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?
```

4. 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 your assistant recognized the `#about_restaurant` intent from the user input.

1. Click the **Dialog** tab.
2. Find the `#General_Greetings` node in the dialog tree.
You will add a node that checks for questions about the restaurant after 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 Dialog builder interface. The top navigation bar includes 'Workspaces' (underlined), 'Watson Assistant tutorial', and 'Build'. Below the navigation are tabs for 'Intents', 'Entities', 'Dialog' (which is selected and underlined), 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, with the 'Add node below' option highlighted.

4. Start to type `#about_restaurant` into the **If assistant recognizes** field of this node. Then select the `#about_restaurant` option.
5. Add the following text as the response.

To copy the text, click the copy icon that is associated with the text block:

Truck Stop Gourmand is the brainchild 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.

[Intents](#) [Entities](#) **Dialog** [Content Catalog](#)

Name this node...

[Customize](#)#about_restaurant [-](#) [+](#)

Then respond with:

[▼](#) **Image** [▼](#)Move: [▲](#) [▼](#) [-](#)

Title (optional)

Description (optional)

[Add title text](#)[Add image description](#)

Image source

<https://www.ibmlearningcenter.com/wp-content/uploads/2018/02/IBM-Learning-Center-Food4.j>[▼](#) **Text** [▼](#)Move: [▲](#) [▼](#) [-](#)Truck Stop Gourmand is the brain child of Gloria and Fred Smith. What started out as [-](#)

Enter response variation

Response variations are set to **sequential**. Set to [random](#) | [multiline](#) [i](#)

8. Click to close the edit view.

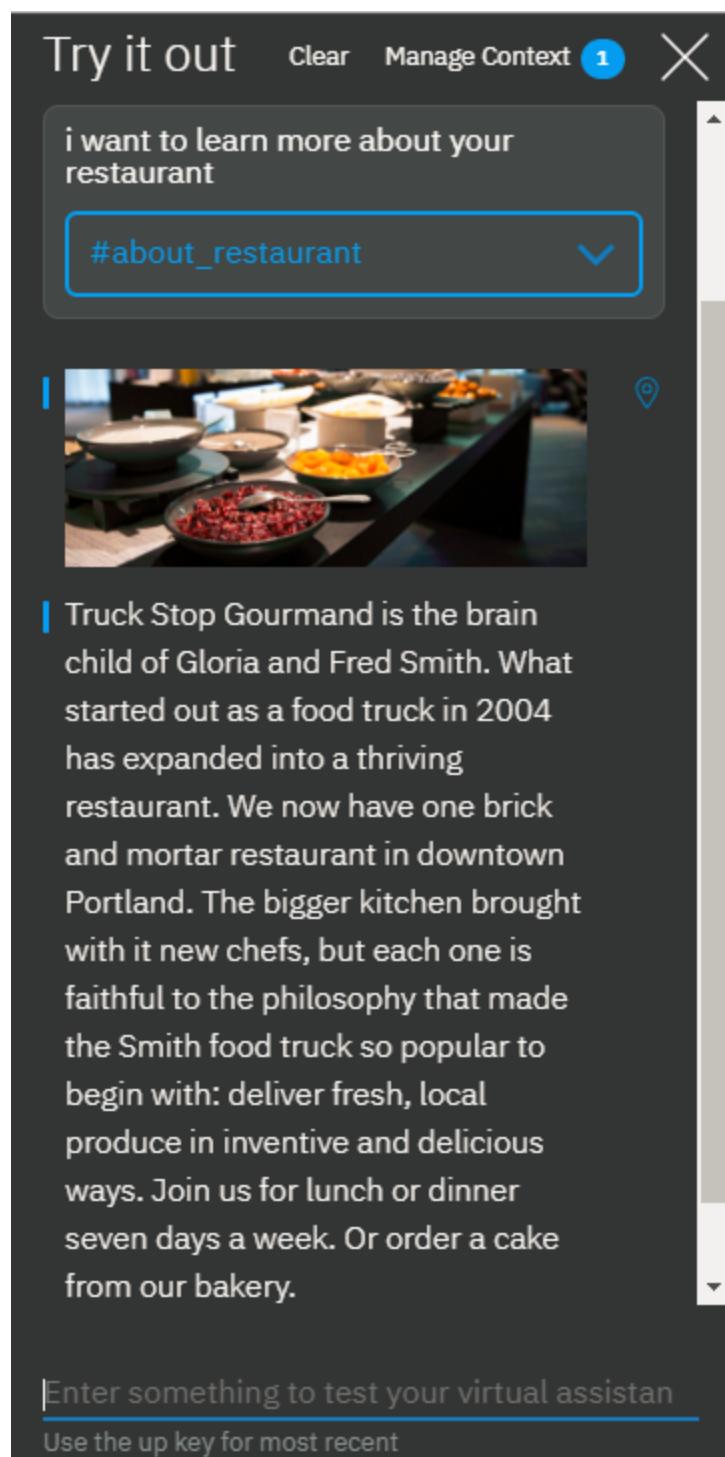
Test the #about_restaurant dialog node

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 your assistant to recognize input with an **#about_restaurant** intent.

[Try it](#)

1. Click the [Try it](#) icon to open the "Try it out" pane.
2. Enter, **I want to learn more about your restaurant.**

Your assistant indicates that the **#about_restaurant** intent is recognized, and returns a response with the image and text that you specified for the dialog node.



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 **Create intent**.

The screenshot shows the IBM Watson Assistant interface. At the top, there's a header bar with the title "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". Under the "Intents" tab, there's a sub-menu with "Add intent" and three icons: up, down, and delete. The main content area is titled "Create new intent".

3. Enter `menu` in the *Intent name* field, and then click **Create intent**.

This screenshot shows the "Create new intent" dialog box. On the left is a sidebar with icons for tools, a circular arrow, a line graph, and a document. The main area has a back arrow and the title "Create new intent". It contains two input fields: "Intent name" with "#menu" typed in and a magnifying glass icon; and "Description" with the placeholder "Add a description to this intent". Below these is a blue "Create intent" button. A message "No examples yet." is displayed, followed by a note: "Train your virtual assistant with this intent by adding unique examples of what your users would say."

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?
```

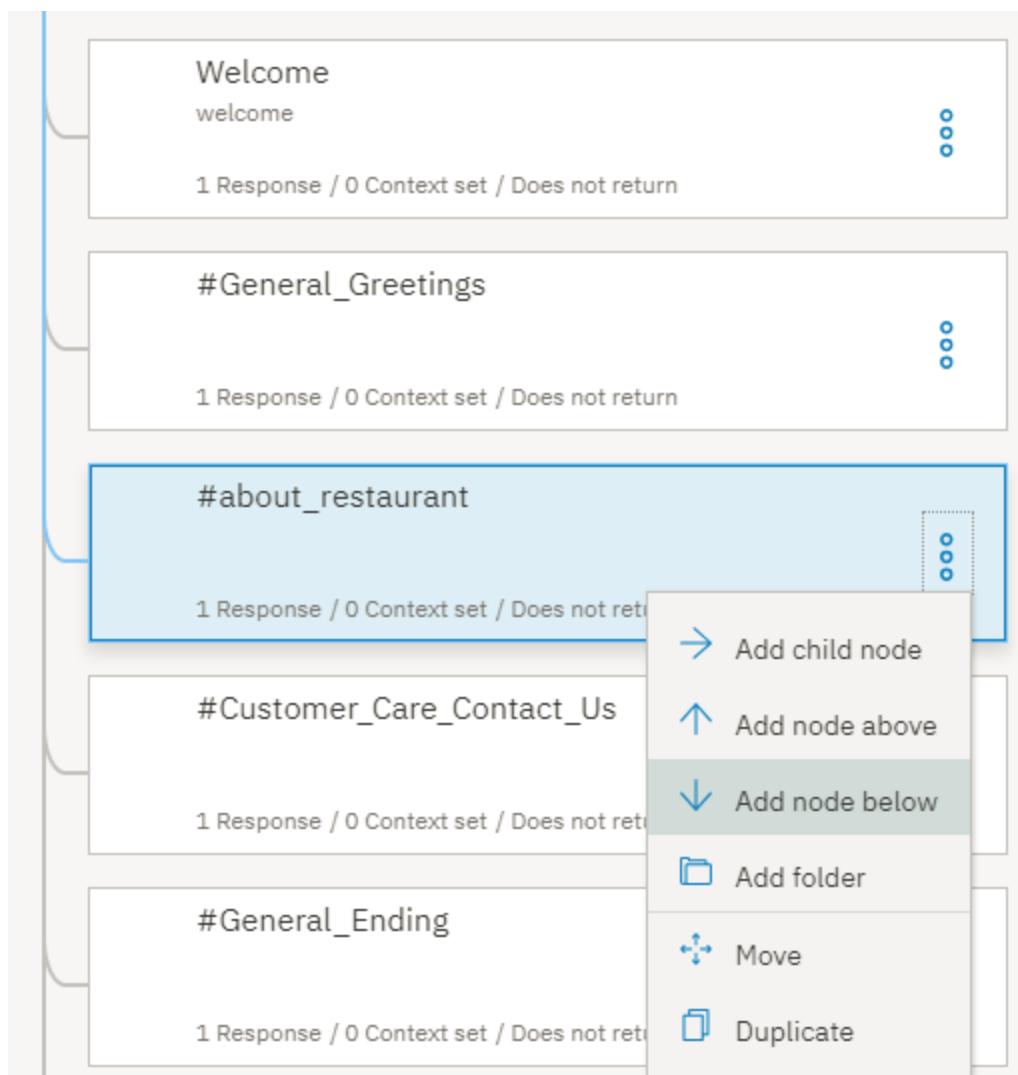
5. Click the **Close** icon to finish adding the `#menu` intent.

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 your assistant recognized the `#menu` intent from the user input.

1. Click the **Dialog** tab.
2. Find the `#about_restaurant` node in the dialog tree.

You will add a node that checks for questions about the menu after 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 **If assistant recognizes** 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 interface for creating a response. At the top, there is a dropdown menu set to "Text" with a "Move" button and a delete icon. Below this, the text "In keeping with our commitment to giving you only fresh local ingredients, our menu" is displayed with a minus sign to its right. A placeholder "Enter response variation" is present. A note at the bottom states "Response variations are set to sequential. Set to random | multiline". Below the main text area, there is a sidebar titled "+ Add response type" containing 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 Watson Assistant interface for adding an option response type. It features a "Text" response type above an "Option" response type. The "Option" response type has fields for "Title" (containing "Which menu do you want to see?") and "Description (optional)" (containing "Add description"). Below these, there are "List label" and "Value" columns, both currently empty. A red box highlights the "+ Add option" button. A note at the bottom says "No options".

8. Click **Add option**.

9. In the **List 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 your assistant 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:

List label	Value
Vegetarian	vegetarian menu
Cake shop	cake shop menu
Option response type details	

Option

Move:

Title	Description (optional)
Which menu do you want to see?	Add description
List label	Value
1 Standard	standard menu
2 Vegetarian	vegetarian menu
3 Cake shop	cake shop menu

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 among different menu types.

1. Click the **Entities** tab.

IBM Watson Assistant

Workspaces / Watson Assistant tutorial / Build

Intents Entities Dialog Content Catalog

My entities System entities

No entities yet.

An entity is a portion of the user's input that you can use to provide a different response to a particular intent. Adding values and synonyms to entities helps your virtual assistant learn and understand important details that your users mention.

Import Add entity

2. Click **Create entity**.

3. Enter `menu` into the entity name field.

Create new entity

Entity name
@menu

Create entity

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:

- o bill of fare
- o cuisine
- o carte du jour

Last modified a minute ago

Fuzzy Matching BETA

Entity name
@menu

Value name
standard

Synonyms

Synonyms	
standard menu	-
carte du jour	-
Add synonym...	+

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 checkbox for *vegan diet*.

10. Click **Add selected**.

11. Click the empty *Add synonym* field, and then add these additional synonyms:

- o vegan
- o plants-only

Last modified a few seconds ago

Fuzzy Matching BETA

Entity name
@menu

Value name
vegetarian

Synonyms

Synonyms	
vegetarian menu	-
vegan diet	-
Add synonym...	+
meatless	-
vegan	-
plants-only	-

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:

- o cake shop menu
- o dessert menu
- o bakery offerings

Entity name
@menu

Value name
cake

Synonyms

cake menu
bakery offerings

cake shop menu
dessert menu

Add value Show recommendations

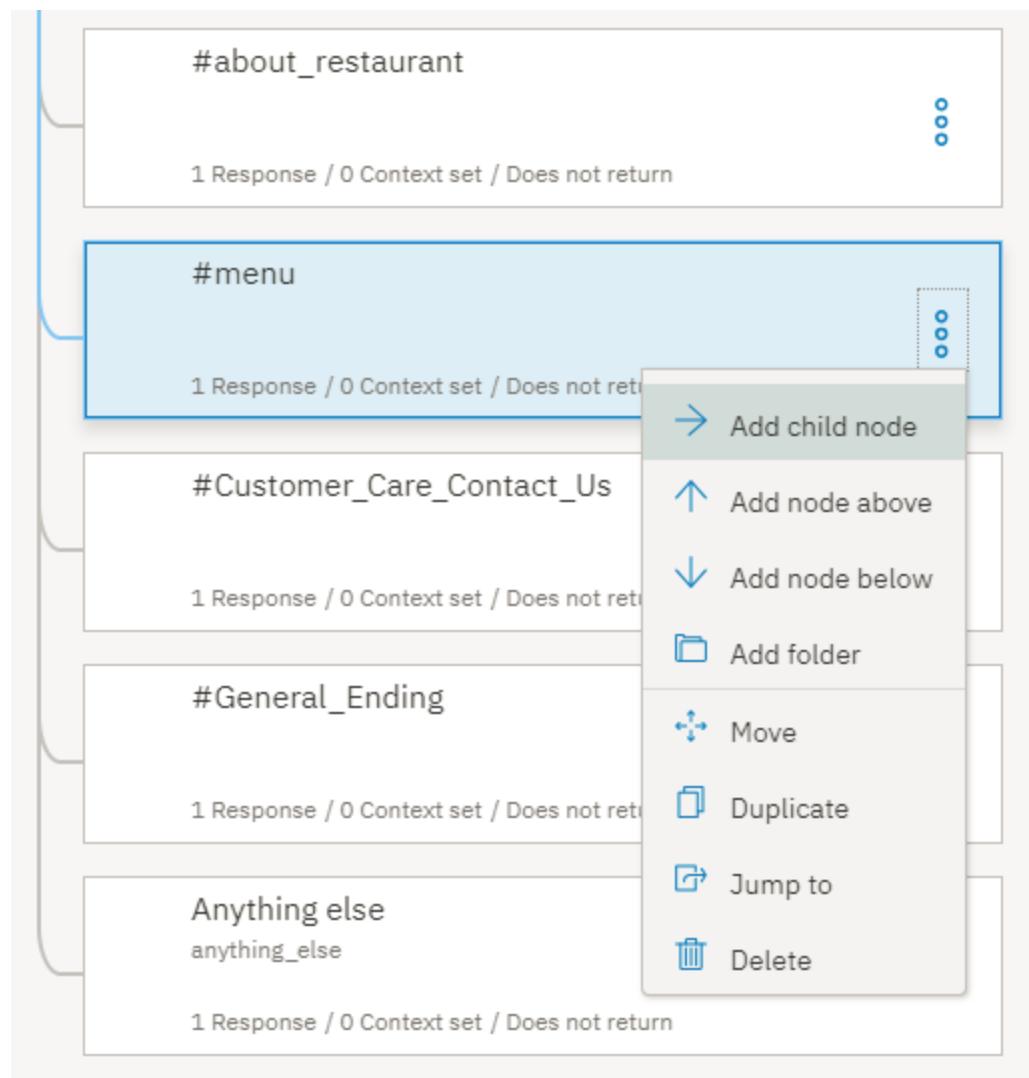
Fuzzy Matching BETA

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 **Dialog** 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 **If assistant recognizes** 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.`

The screenshot shows the Watson Assistant interface. On the left, a tree view displays nodes: '#about_restaurant' (1 Response / 0 Context set / Does not return), '#menu' (1 Response / 0 Context set / Does not return), and '@menu:standard' (1 Response / 0 Context set). The '@menu:standard' node is selected and highlighted with a blue border. On the right, the 'Then respond with:' section is visible, containing a 'Text' input field with the placeholder 'To see our menu, go to the menu page on our website.' Below it is a note: 'Enter response variation' and 'Response variations are set to sequential. Set to random | multiline'.

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 **If assistant recognizes** 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 interface. On the left, a tree view displays nodes: '#menu' (1 Response / 0 Context set / Does not return), '@menu:standard' (1 Response / 0 Context set), and '@menu:vegetarian' (1 Response / 0 Context set). The '@menu:vegetarian' node is selected and highlighted with a blue border. On the right, the 'If bot recognizes:' section is visible, containing the condition `@menu:vegetarian`. Below it is the 'Then respond with:' section, which contains a 'Text' input field with the placeholder 'To see our vegetarian menu, go to the vegetarian menu page on our website.'

10. Click 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 **If assistant recognizes** 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 editor interface. On the left is a sidebar with icons for Workspaces, Intents, Entities, Dialog, and Content Catalog. The main area has tabs for Intents, Entities, Dialog (which is selected), and Content Catalog. A tree view on the left shows 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, configuration fields are shown:

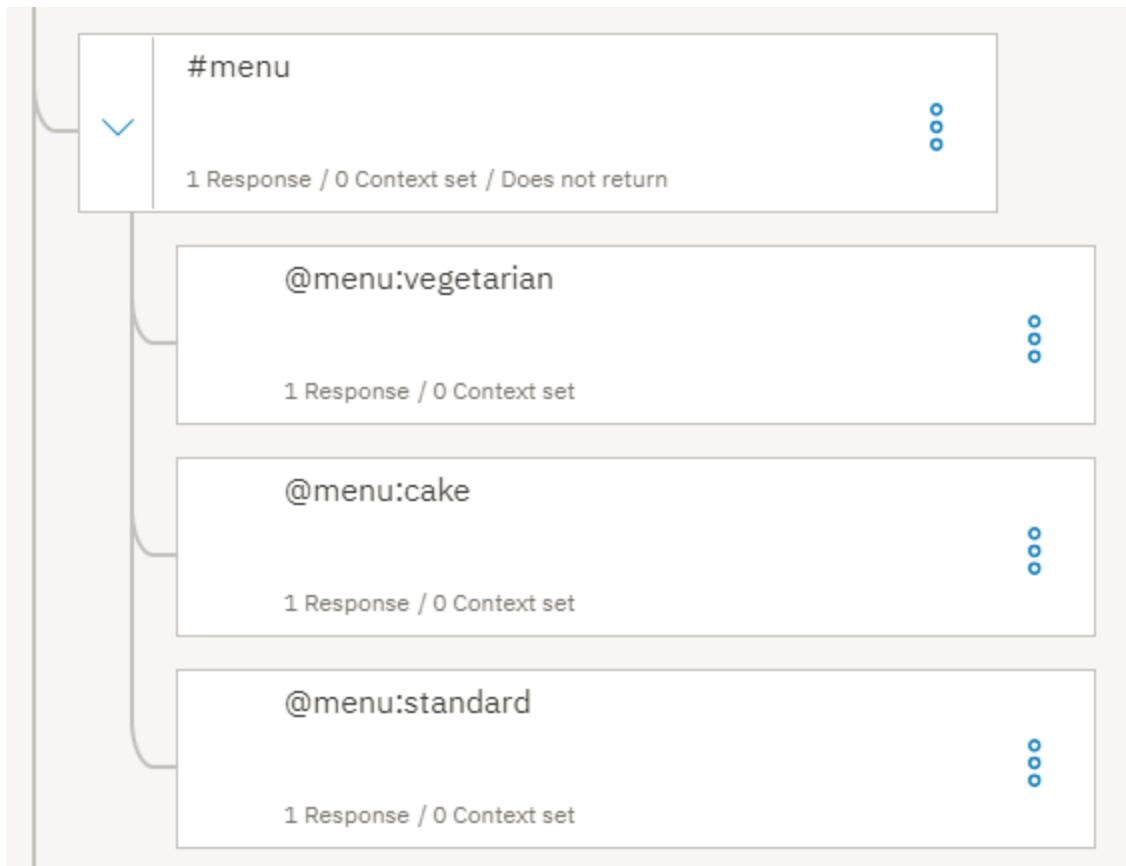
- Name this node... (with a gear icon)
- If bot recognizes: @menu:cake (with minus and plus buttons)
- Then respond with:
 - Type: Text
 - Text: To see our cake shop menu, go to the menu
 - Enter response variation

14. Click to close the edit view.
15. The standard menu is likely to be requested most often, so move it to the end 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.

Click the **More :** icon on the `@menu:standard` node, and then select **Move**.

The screenshot shows the Watson Assistant Dialog editor with the move dialog open for the `@menu:standard` node. The title bar says "Select where you want to move the node to." The tree view on the left shows the same structure as before, but the `@menu:standard` node is highlighted in blue. A context menu is open next to it, with the "Move to..." option selected. The submenu options are: As child node, Above node, and Below node. The "Below node" option is highlighted with a red box.

16. 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

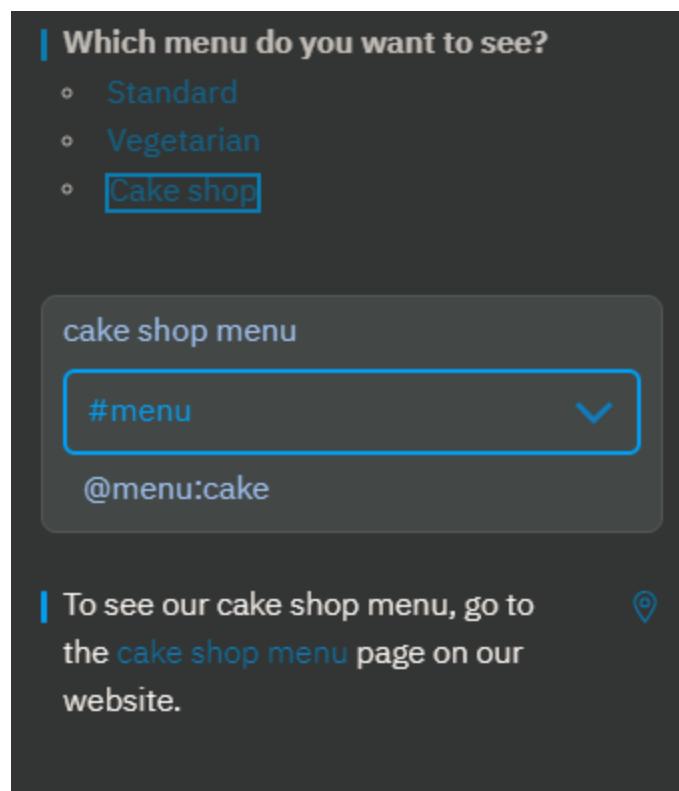
Test the dialog nodes that you added to recognize menu questions.

1. Click the  icon to open the "Try it out" pane.
2. Enter, `What type of food do you serve?`

Your assistant indicates that the `#menu` intent is recognized, and displays the list of menu options for the user to choose from.

3. Click the `Cake shop` option.

Your assistant 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.`



4. Click the *cake shop* hyperlink in the response.

A new web browser page opens and displays the example.com website.

5. Close the example.com web 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 two uppercase letters followed by 5 numbers. For example, `YR34663`. Add an entity that can recognize this character pattern.

1. Click the **Entities** tab.
2. Click **Create entity**.
3. Enter `order_number` into the entity name field.
4. Click **Create entity**.

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 Watson Assistant interface for creating a new entity. The left sidebar has icons for Home, Create, Manage, and Help. The main area has a header with a back arrow and the entity name '@order_number'. On the left, there's a sidebar with icons for Entity, Value, Pattern, and Annotation. The 'Entity' icon is highlighted. The main form fields are: Entity name (@order_number), Value name (order_syntax), and a dropdown menu showing 'Synonyms' (selected) and 'Patterns'. Below the dropdown are 'Add value' and 'Show results' buttons. At the bottom, tabs show 'Dictionary' and 'Annotation BETA'.

6. Add the following regular expression to the Pattern field: [A-Z]{2}\d{5}

The screenshot shows the entity '@order-number' created. The Entity name is '@order-number', Value name is 'order_syntax', and the Patterns dropdown is set to 'Patterns' with the value '[A-Z]{2}\d{5}' entered. A '+' button is available to add more patterns. The Fuzzy Matching toggle is off. The interface includes a 'Last modified a few seconds ago' timestamp, download, delete, and try it buttons. A note at the bottom says 'No values yet.'

7. Click **Add value**.

The screenshot shows the entity '@order-number' with an additional value added. The Entity name is '@order-number', Value name is 'order_syntax', and the Patterns dropdown is set to 'Patterns' with the value '[A-Z]{2}\d{5}' entered. A '+' button is available to add more patterns. The Fuzzy Matching toggle is off. The interface includes a 'Last modified a few seconds ago' timestamp, download, delete, and try it buttons. A note at the bottom says 'No values yet.'

8. Click the **Close** icon to finish adding the @order_number entity.

The screenshot shows the 'Entities' tab in the Watson Assistant interface. On the left is a dark sidebar with icons for Home, Intents, Entities (highlighted), Dialog, and Content Catalog. The main area has a header 'Workspaces / Watson Assistant tutorial / Build' and tabs for 'Intents', 'Entities' (selected), 'Dialog', and 'Content Catalog'. Below this is a section titled 'My entities' with a sub-section 'System entities'. A button 'Add entity' is highlighted with a dashed border. To its right are download and delete icons. A search bar with a magnifying glass icon is on the far right. The main content area lists two entities: '@menu' and '@order_number'. Each entry includes a checkbox, the entity name, its values ('vegetarian, cake, standard' for @menu, 'order_syntax' for @order_number), and a 'Modified' timestamp ('an hour ago' for @menu, 'a minute ago' for @order_number).

Add a cancel order intent

1. Click the **Intents** tab.
2. Click **Create 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.
```

The screenshot shows the creation of an intent named '#cancel_order'. The top navigation bar includes a back arrow, the intent name '#cancel_order', a 'Last modified a few seconds ago' timestamp, a download icon, a trash icon, and a 'Try it' button. The sidebar on the left is identical to the previous screenshot. The main content area starts with a 'Add user examples' section with a link 'Add user examples to this intent'. Below this is a 'User examples (8)' section with a dropdown arrow. Eight user examples are listed, each with a checkbox and a pencil icon for editing: 'Can I cancel my cake order?', 'I'd like to cancel my order', 'I need to cancel an order I just placed', 'I want to cancel my cake order', 'please cancel the birthday cake order I placed last week', 'that order i placed, i need to cancel it.', 'The party theme changed; we don't need a cake anymore', and 'There's been a change. I need to cancel my bakery order.'

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 your assistant is proposing.

1. Click the **Intents** tab.
2. Click **Create 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. The left sidebar has icons for Intents, Entities, Flows, and Dialog. The main area shows the intent name '#yes' at the top, with a note 'Last modified a few minutes ago'. Below this is a section for 'Add user examples' with a placeholder 'Add user examples to this intent' and a 'Add example' button. A list of 10 user examples is shown, each with a checkbox and a pencil icon for editing. The examples are: '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.', and '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 **If assistant recognizes** 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 Dialog builder interface. The top navigation bar includes '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'. On the left is a sidebar with icons for 'Nodes', 'Flows', 'Intents', 'Entities', and 'Variables'. The main workspace has a header 'Name this node...' with a 'Customize' button and a close 'X'. Below it, under 'If bot recognizes:', there is a list item '#cancel_order' with a minus sign and a plus sign. Under 'Then respond with:', there is a 'Text' dropdown set to 'Text' with a 'Move' section containing arrows and a trash icon. A message 'If the pickup time is more than 48 hours from now, you can cancel your order.' is listed with a minus sign.

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. You define a context variable in the context editor. From the response section of the node, click the **More** icon, and then select **Open context editor**.

The screenshot shows the Watson Assistant Dialog builder with the context editor open. The 'If bot recognizes:' section contains the item '#cancel_order'. The 'Then respond with:' section contains the message 'If the pickup time is more than 48 hours from now, you can cancel your order.' Below this, a 'More' icon is shown with a context menu open. The menu options are 'Open JSON editor' and 'Open context editor', with 'Open context editor' being highlighted. A tooltip 'Order number context variable details' is visible above the message.

6. Enter the following context variable name and value pair:

Variable	Value
\$ordernumber	<? @order_number.literal ?>

Order number context variable details

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	
\$ordernumber	<? @order_number.literal ?>	-

+ Add variable

And respond with:

Text

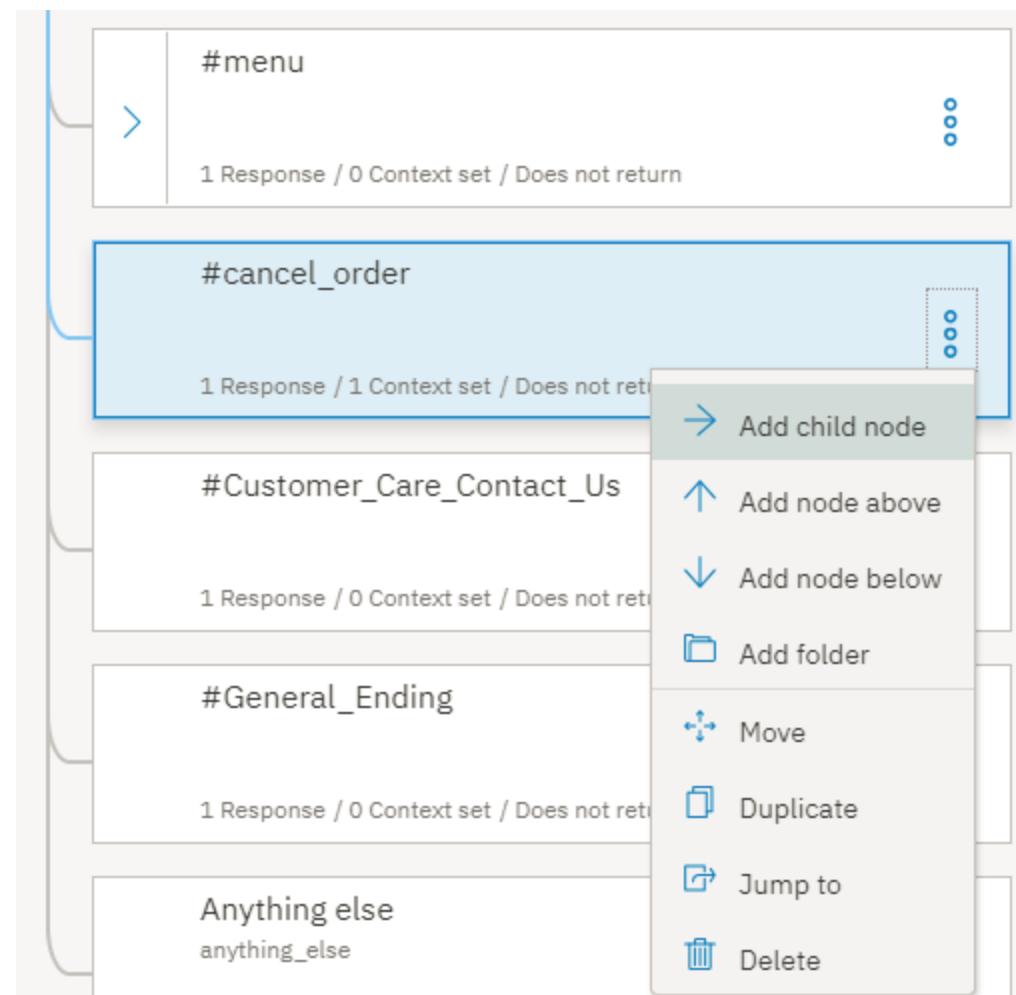
If the pickup time is more than 48 hours from now, you can cancel your order.

Move: ^ v -

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 **If assistant recognizes** field of this node.

10. Add the following message in the response text field:

What is the order number?

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 **If assistant recognizes** 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:

Variable	Value
\$ordernumber	<? @order_number.literal ?>

Order number context variable details

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 a conversational AI builder interface with a sidebar menu and a main configuration area.

Left Sidebar (Flowchart):

- #menu
 - 1 Response / 0 Context set / Does not return
- #cancel_order
 - Ask for order number
 - true
 - 1 Response / 0 Context set / Returns a value
 - @order_number
 - 1 Response / 1 Context set
- #Customer_Care_Contact_Us

Main Configuration Area:

Name this node...

Context Configuration:

@order_number

Then set context:

Variable	Value
\$ordernumber	"<? @order_number.literal ?>"

And respond with:

Text

Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a

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 **If assistant recognizes** 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 a dialog flow editor interface with the following structure:

- #cancel_order** (Root node)
 - 1 Response / 1 Context set / Does not return
- Ask for order number**
 - true
 - 1 Response / 0 Context set / Returns a value
- @order_number**
 - 1 Response / 1 Context set
- true**
 - 1 Response / 0 Context set

If bot recognizes:
true (-) (+)

Then respond with:

Text Move: ▲ ▼ 1

I need the order number to cancel the order for you. If you don't know the order num (E)

Enter response variation

Response variations are set to **sequential**. Set to **random | multiline** (i)

(+) Add response type

21. Click  to close the edit view.
 22. Add a node after 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 **If assistant recognizes** 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 flowchart on the left and configuration options on the right.

Flowchart:

```

graph TD
    A["#cancel_order  
1 Response / 1 Context set / Does not return"] --> B["Ask for order number  
true  
1 Response / 0 Context set / Return all"]
    B --> C["@order_number  
1 Response / 1 Context set"]
    C --> D["true  
1 Response / 0 Context set"]
    D --> E["Number provided  
@order_number  
1 Response / 0 Context set"]
    E --> F["#yes  
1 Response / 0 Context set"]
  
```

Configuration (right side):

- Number provided:** @order_number
- If bot recognizes:** @order_number
- Then respond with:**
 - Type:** Text
 - Text:** Just to confirm, you want to cancel order \$ordernumber?
 - Enter response variation:**
 - Response variations are set to sequential. Set to random | multiline**
 - 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 **If assistant recognizes** 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 flowchart on the left and configuration options on the right.

Flowchart:

```

graph TD
    A["#cancel_order  
1 Response / 1 Context set / Does not return"] --> B["Ask for order number  
true  
1 Response / 0 Context set / Return all"]
    B --> C["Number provided  
@order_number  
1 Response / 0 Context set / Return all"]
    C --> D["#yes  
1 Response / 0 Context set"]
  
```

Configuration (right side):

- If assistant recognizes:** #yes
- Then respond with:**
 - Type:** Text
 - Text:** Ok. The order \$ordernumber is canceled. We hope we get the opportunity to bake a cake for you sometime soon.
 - Enter response variation:**
 - Response variations are set to sequential. Set to random | multiline**
 - Add response type**

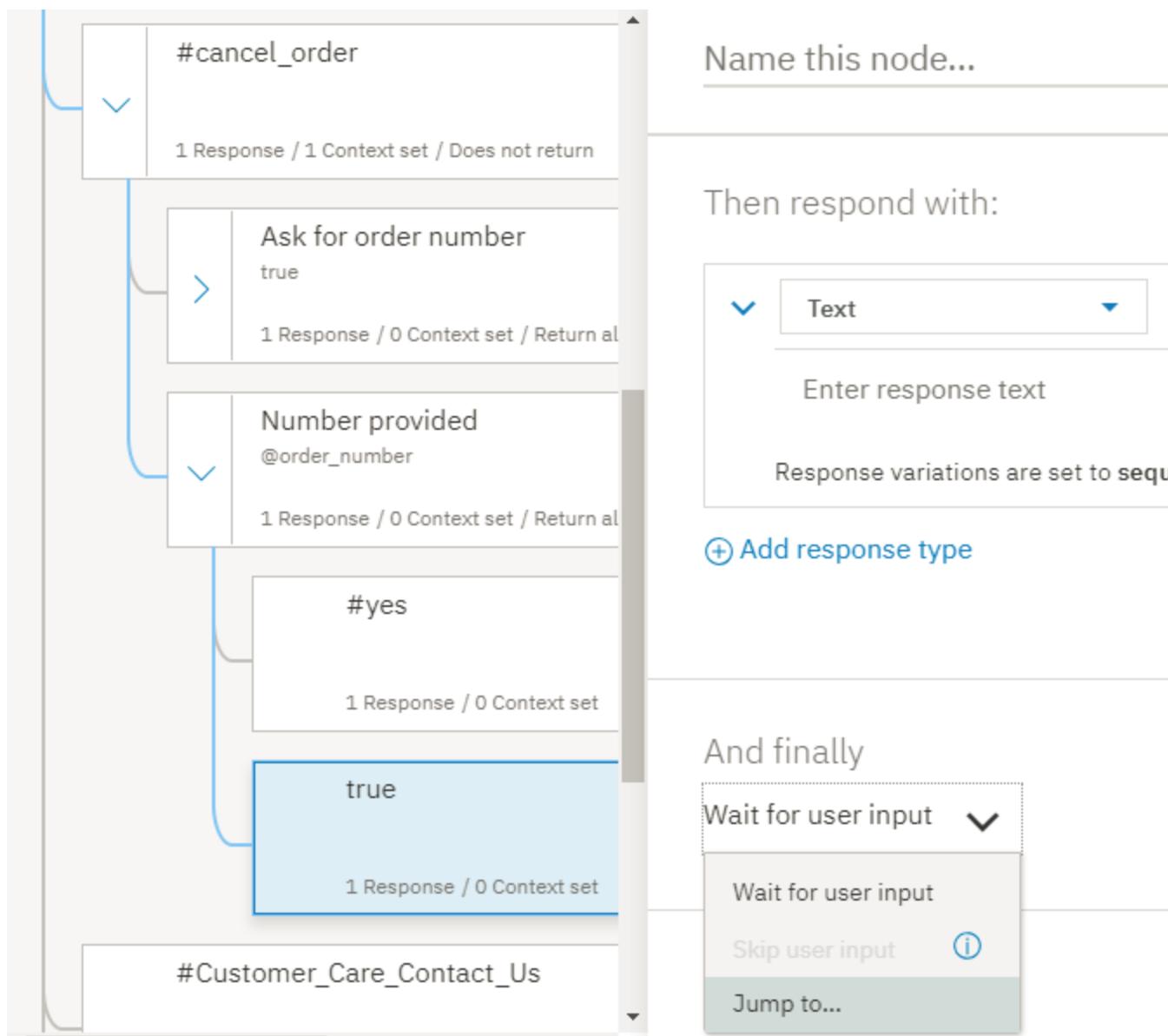
29. Click to close the edit view.

30. Click the **More :** icon on the **#yes** node, and then select **Add node below**.

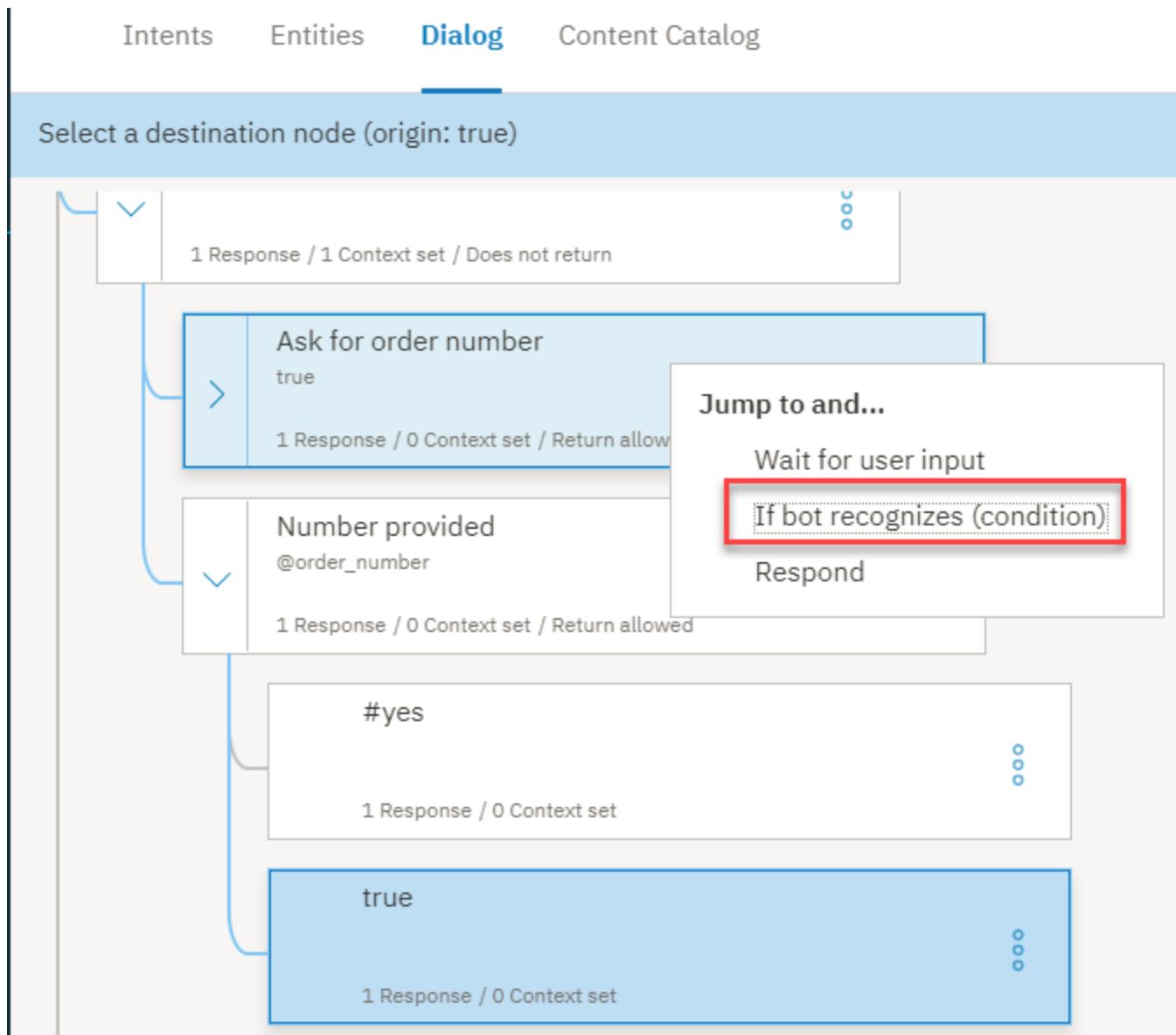
31. Type **true** into the **If assistant recognizes** field of this node.

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**.

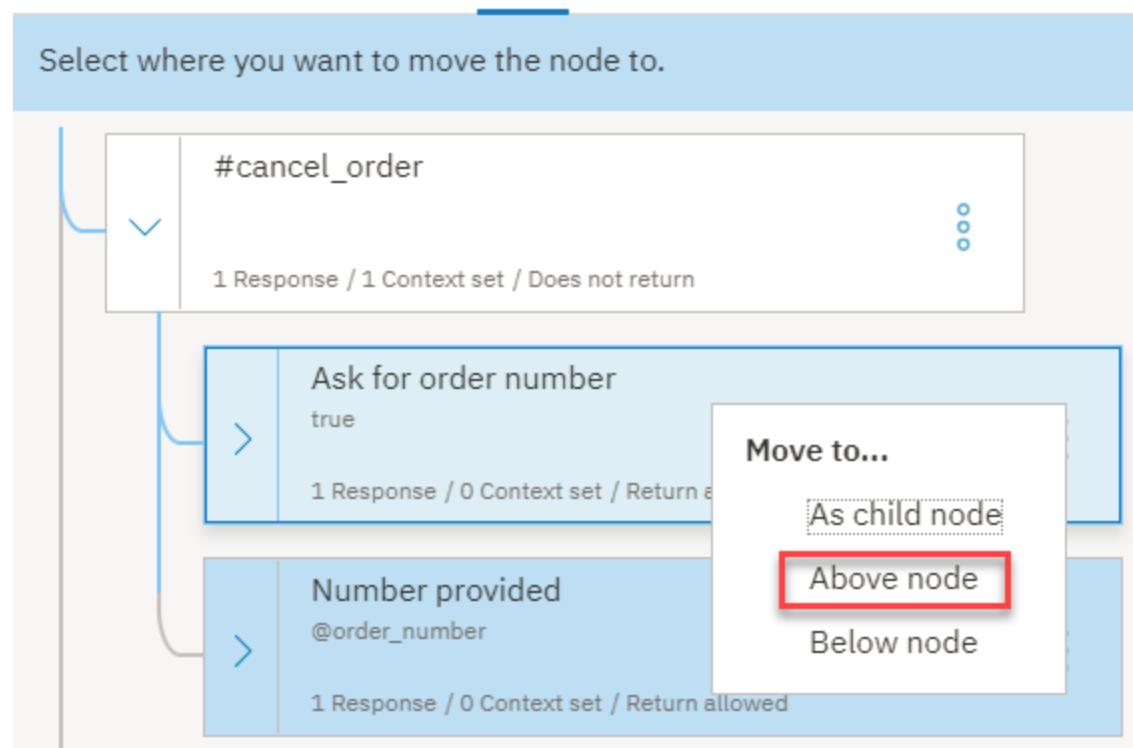


33. Select the *Ask for order number* node's condition.

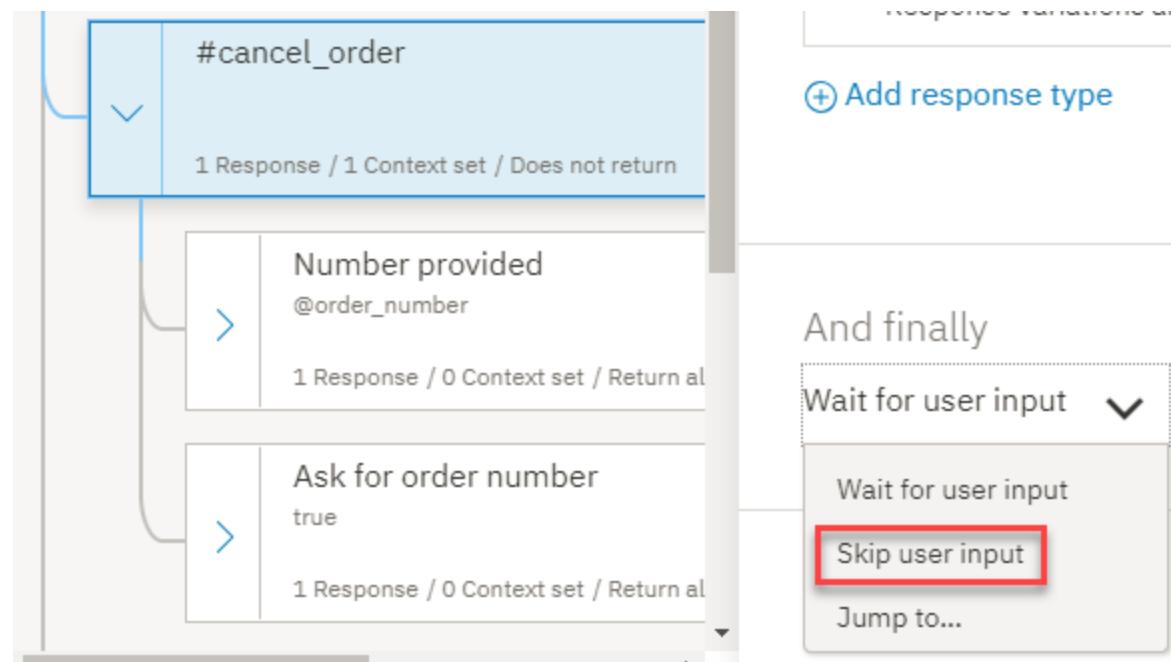


34. Click to close the edit view.

35. Move the *Number provided* node before 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**.



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 your assistant can recognize character patterns that match the pattern used for product order numbers in user input.



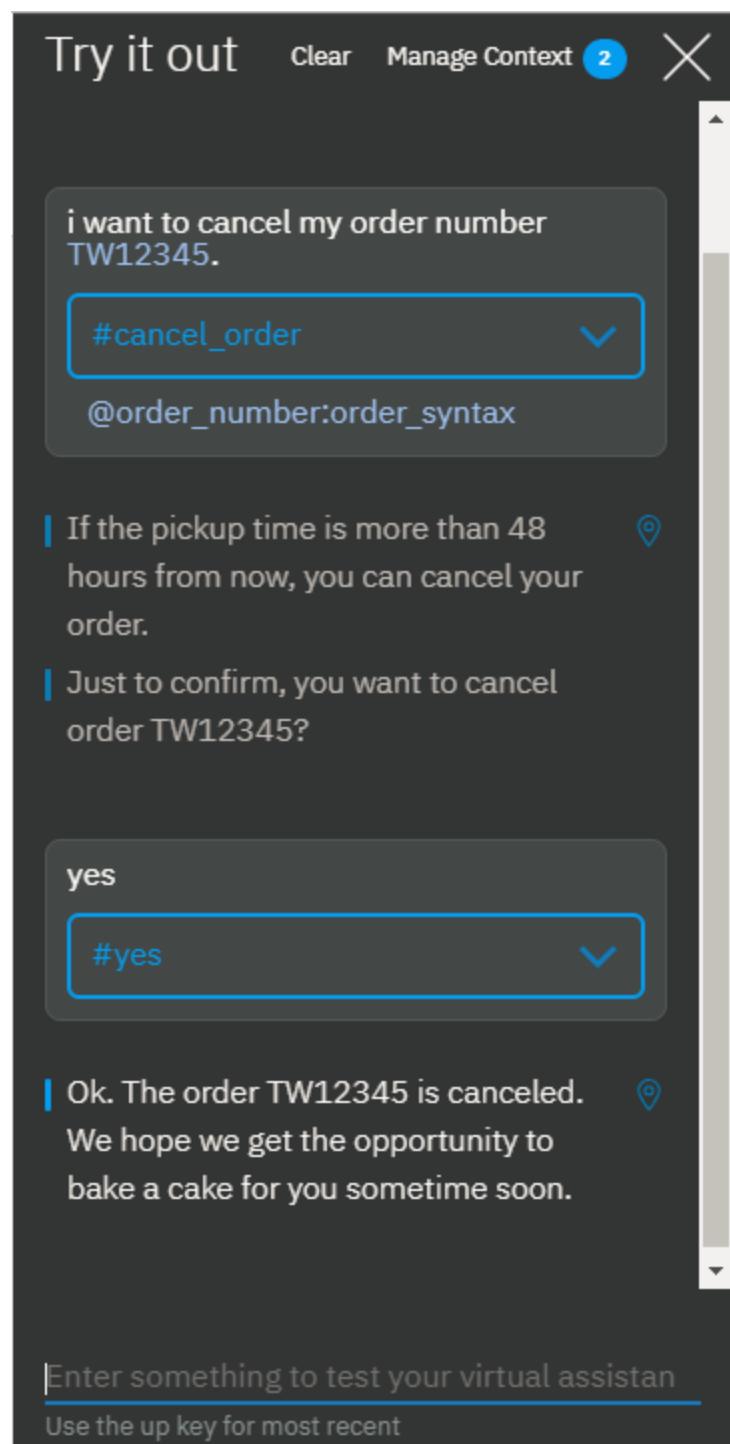
- Click the `Try it` icon to open the "Try it out" pane.

- Enter, `i want to cancel my order number TW12345.`

Your assistant 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?`

- Enter, `Yes`.

Your assistant 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.`



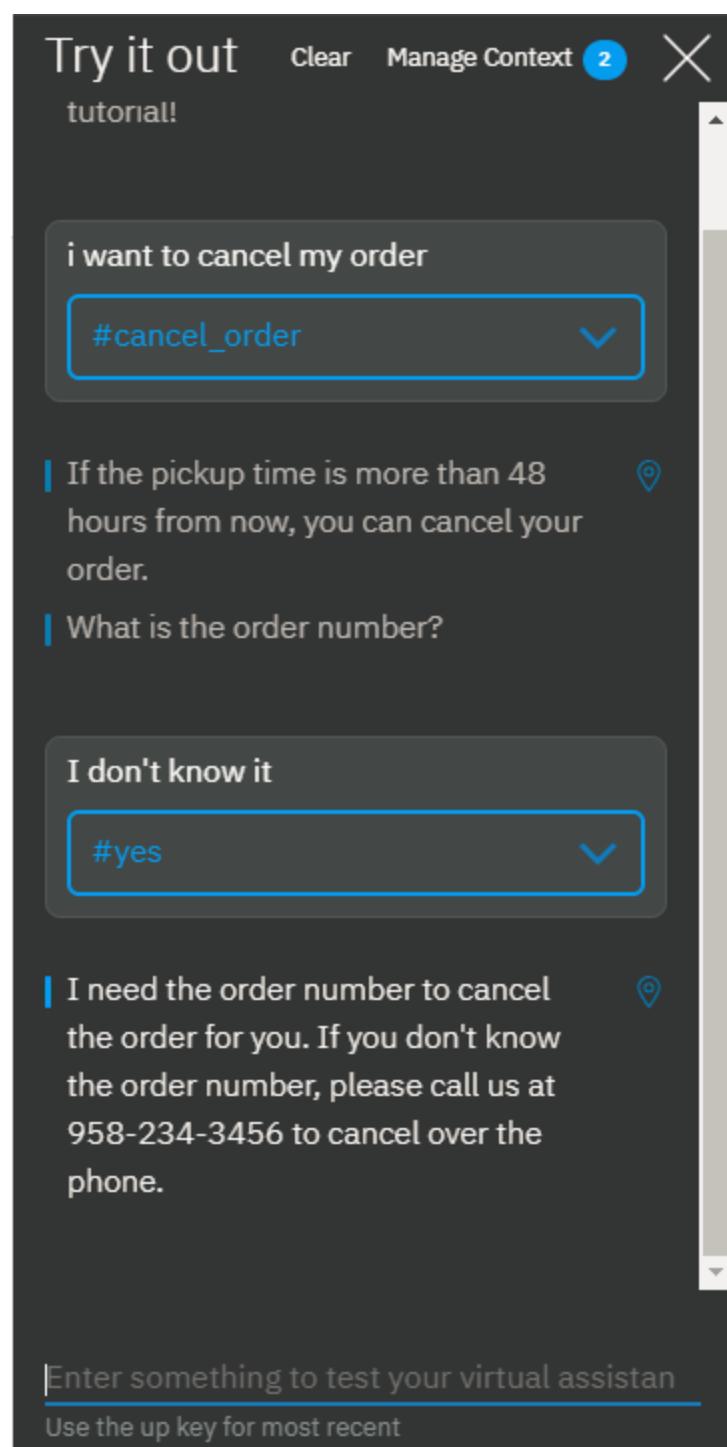
Now, try it when you don't know the order number.

4. Click **Clear** in the "Try it out" pane to start over. Enter, **I want to cancel my order.**

Your assistant 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.**

Your assistant 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.**



Add nodes to clarify order number format

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, correct? 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 uppercase letters, and the n's represent 5 numbers. Do you have an order number in that format?

5. Click **X** 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 **If assistant recognizes** field of this node.
8. Enable conditional responses by clicking **Customize**, scrolling down, and then setting the **Multiple conditioned responses** switch to **On**.
9. Click **Apply**.
10. In the newly-added *If assistant 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 assistant 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.

Intents Entities **Dialog** Content Catalog

The screenshot shows the Watson Assistant Dialog editor interface. On the left, a tree view displays a sequence of nodes: 'Ask for order number' (true), which triggers '@order_number' (1 Response / 1 Context set). This leads to 'input.text.find('\\d')' (1 Response / 0 Context set / Return allowed), which then branches into two paths: 'true' (2 Responses / 1 Context set) and 'true' (1 Response / 0 Context set). On the right, the configuration panel includes:

- Name this node...**: A text input field with a gear icon.
- If bot recognizes:**: A dropdown menu showing 'true' with minus (-) and plus (+) buttons.
- Then respond with:**: A table with two rows:

If bot recognizes	Respond with
1 @order_number	Ok. The order \$ordernumber is car
2 true	I need the order number to cancel
- + Add response**: A button to add more responses.

13. Click 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.

Tip: 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.

Add a node that handles questions about the bot

Add a dialog node that can recognize the user's interest in the bot, and respond.

1. Click the **Dialog** 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 **If assistant recognizes** 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 should I call you?

The screenshot shows the Watson Assistant dialog builder. On the left is a sidebar with icons for Workspaces, Intents, Entities, Dialog (which is selected), and Content Catalog. The main area displays a dialog tree under the heading "Watson Assistant tutorial". The tree has three nodes: "Welcome" (with sub-node "welcome"), "#General_About_You", and "#General_Greetings". Each node has a status bar indicating "1 Response / 0 Context set / Does not return". To the right of the tree is a context editor panel. It starts with a field "Name this node..." followed by "Watson Assistant tutorial". Below that is a section "If bot recognizes:" containing the condition "#General_About_You". There is a plus sign (+) icon next to it. Further down is a section "Then respond with:" which includes a dropdown menu set to "Text" and a text input field containing the message "I am a chat bot that is designed to a".

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. In the **If assistant recognizes** field of this node, enter **true**.
9. Add the following message in the response text field:

```
Hello, <? input.text ?>! 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:

Variable	Value
username	<? input.text ?>

User name context variable details

The context variable value (`<? input.text ?>`) 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.

12. 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**, scroll down, and then set the **Multiple conditioned responses** switch to **On**.

Customize "#General_Greetings"

[Customize node](#)

[Digressions](#)

Slots ⓘ

off

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 ⓘ

on

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**.

The screenshot shows the Watson Assistant Dialog builder interface. On the left, there's a sidebar with icons for intents, entities, and dialog nodes. The main area has tabs for Intents, Entities, Dialog (which is selected), and Content Catalog. In the center, there's a tree view of a dialog node named "Watson Assistant tutorial". This node has several children: "Welcome", "#General_About_You", "#General_Greetings" (which is highlighted in blue), and "#about_restaurant". To the right of the tree, there's a form for configuring the "#General_Greetings" node. It includes fields for "Name this node..." (with "Cu" next to it), "If bot recognizes:" (containing "#General_Greetings"), and "Then respond with:". Under "Then respond with:", there's a table with two columns: "If bot recognizes" and "Respond with". The first row shows "Enter an intent, entity or context va" and "Good day to you!". Below this table is a button labeled "+ Add response".

4. Click **Add response**.

5. In the *If assistant recognizes* field, type `$username`, and in the *Respond with* field, add a new response:

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!`).

Name this node...

Customize



If bot recognizes:

#General_Greetings (−) (+) Add response

Then respond with:

If bot recognizes	Respond with	
1 \$username	Good day to you, \$username!	(−) (+) Edit
2 Enter an intent, entity or context va	Good day to you!	(−) (+) Edit

(+) Add response

7. Click (X) to close the edit view.

Test personalization

Test whether your assistant 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?**

Your assistant recognizes the **#General_About_You** intent. Its response ends with the question, **What should I call you?**

4. Enter, **Jane**.

Your assistant saves **Jane** in the **\$username** variable.

5. Enter, **Hello.**

Your assistant recognizes the **#General_Greetings** intent and says, **Good day to you, Jane!** 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.

Step 6: Test the assistant from your web page integration

Now that you have built a more sophisticated version of the assistant, return to the public web page that you deployed as part of the previous tutorial, and then test the new capabilities you added.

1. Open the assistant.
2. Click **Preview**.
3. Copy and paste the URL from *Share this link* into a web browser.

An IBM-branded page is displayed with your assistant embedded in it as a chat window.

4. Repeat a few of the test utterances that you submitted to the "Try it out" pane to see how the assistant behaves in a real integration.



Note: Unlike when you send test utterances to your assistant from the "Try it out" pane, standard usage charges apply to API calls that result from utterances that are submitted to the chat widget.

Next steps

Now that you have built and tested your dialog skill, you can share it with customers. Deploy your skill by first connecting it to an assistant, and then deploying the assistant. There are several ways you can do this. See [Adding integrations](#) for more details.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

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 skill 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 dialog skill if you want. Just be sure to create the skill 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. From the **Intents** page of the tutorial skill, 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 your assistant 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.

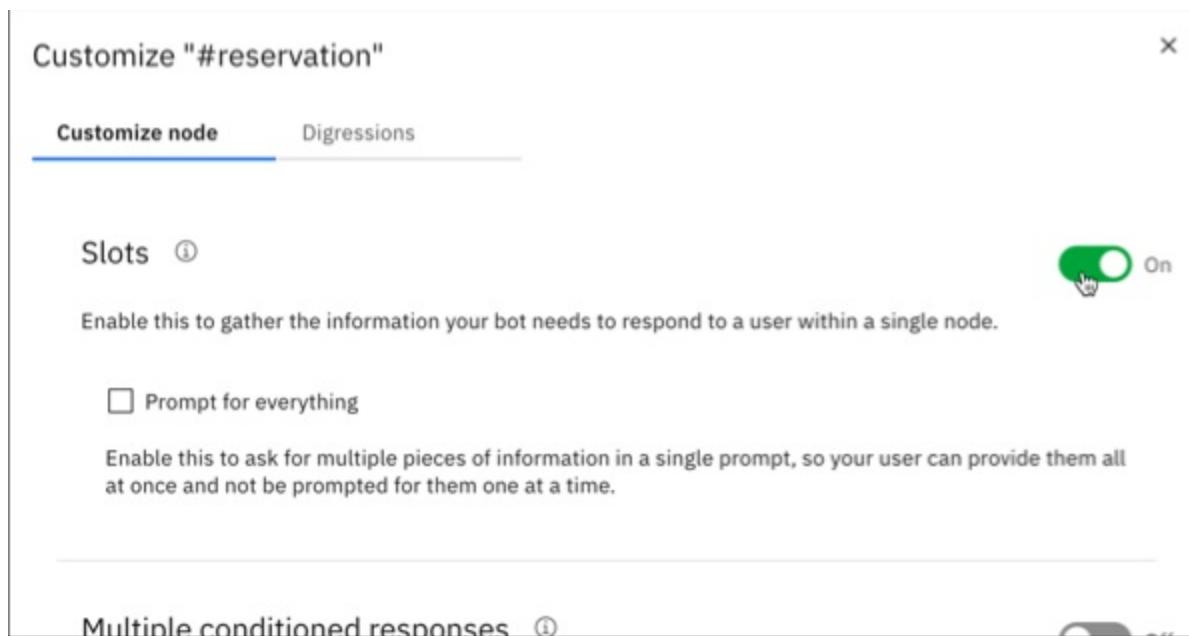
Step 3: Add a dialog node with slots

A dialog node represents the start of a thread of dialog between your assistant and the user. It contains a condition that must be met for the node to be processed by your assistant. 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 **Welcome** node, and then select **Add node below**.
3. Start typing `#reservation` in the **If assistant recognizes** field, and then select it from the list. This node will be evaluated if the user input matches the `#reservation` intent.
4. Click **Customize**, set the **Slots** switch to **On**, and then click **Apply**.



5. In the **Then check for** section, add the following slots:

Check for	Save it as	If not present, ask
@sys-date	\$date	What day would you like to come in?
@sys-time	\$time	What time do you want for the reservation?
@sys-number	\$guests	How many people will be dining?

Slot details

6. In the **Assistant responds** section, enter the text response `OK. I am making you a reservation for $guests on $date at $time.`

The screenshot shows the 'Then check for' section of a node configuration. It lists three slots:

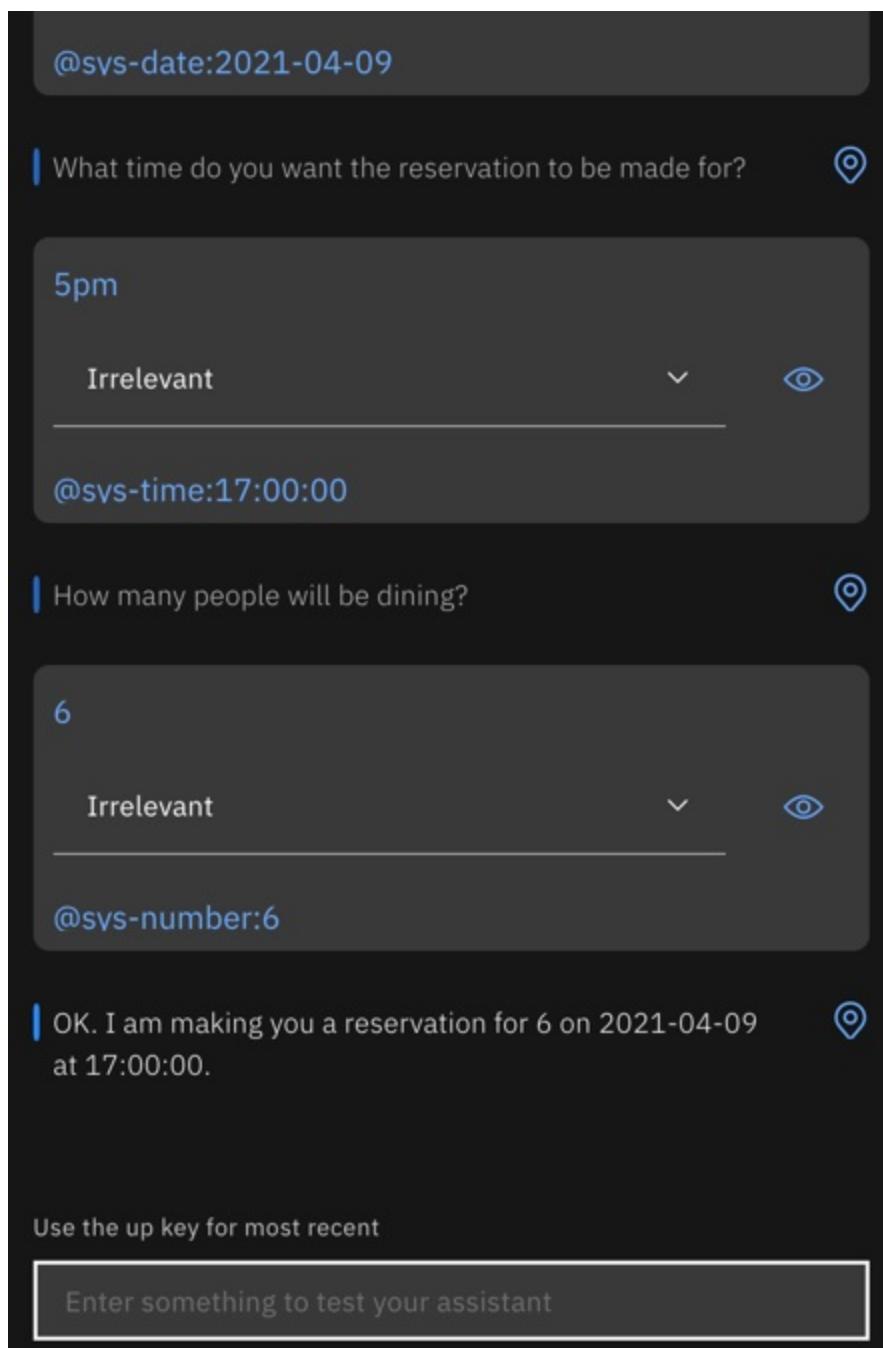
- Slot 1:** Check for `@sys-date`, Save it as `$date`, If not present, ask `What day woul`, Type Required.
- Slot 2:** Check for `@sys-time`, Save it as `$time`, If not present, ask `What time do y`, Type Required.
- Slot 3:** Check for `@sys-number`, Save it as `$guests`, If not present, ask `How many peo`, Type Required.

Below the slots, there is an 'Add slot +' button. Under 'Assistant responds', there is a 'Text' dropdown menu and a response card with the text: "OK. I am making you a reservation for \$guests on \$date at \$time." There are also buttons for moving the card up or down and deleting it.

- Click to close the node edit view.

Step 4: Test the dialog

- Click **Try it**.
- Type `i want to make a reservation`.
The assistant recognizes the #reservation intent, and it responds with the prompt for the first slot, `What day would you like to come in?`.
- Type `Friday`.
The assistant recognizes the value, and uses it to fill the `$date` context variable for the first slot. It then shows the prompt for the next slot, `What time do you want the reservation to be made for?`
- Type `5pm`.
The assistant recognizes the value, and uses it to fill the `$time` context variable for the second slot. It then shows the prompt for the next slot, `How many people will be dining?`
- Type `6`.
The assistant 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.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Improving a dialog 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.

1. To reformat the \$date context variable value, click the **Edit response**  icon for the @sys-date slot.
2. From the **More :** menu, 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 (MM), the month is shortened to Dec instead of December.

3. Click **Save**.
4. 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.
5. From the **More :** menu, 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') ?>"  
  }  
}
```

6. Click **Save**.
7. Test the node again. Open the "Try it out" pane, and click **Clear** to delete the slot context variable values that you specified when you tested the node with slots earlier. To see the impact of the changes you made, use the following script:

Speaker	Utterance
You	i want to make a reservation
Watson	What day would you like to come in?

You	Friday
Watson	What time do you want the reservation to be made for?
You	5pm
Watson	How many people will be dining?
You	6

Script details

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.

- From the main node with slots, click **Customize**.
- Select the **Prompt for everything** checkbox to enable the initial prompt, and then click **Apply**.

Customize "#reservation"

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

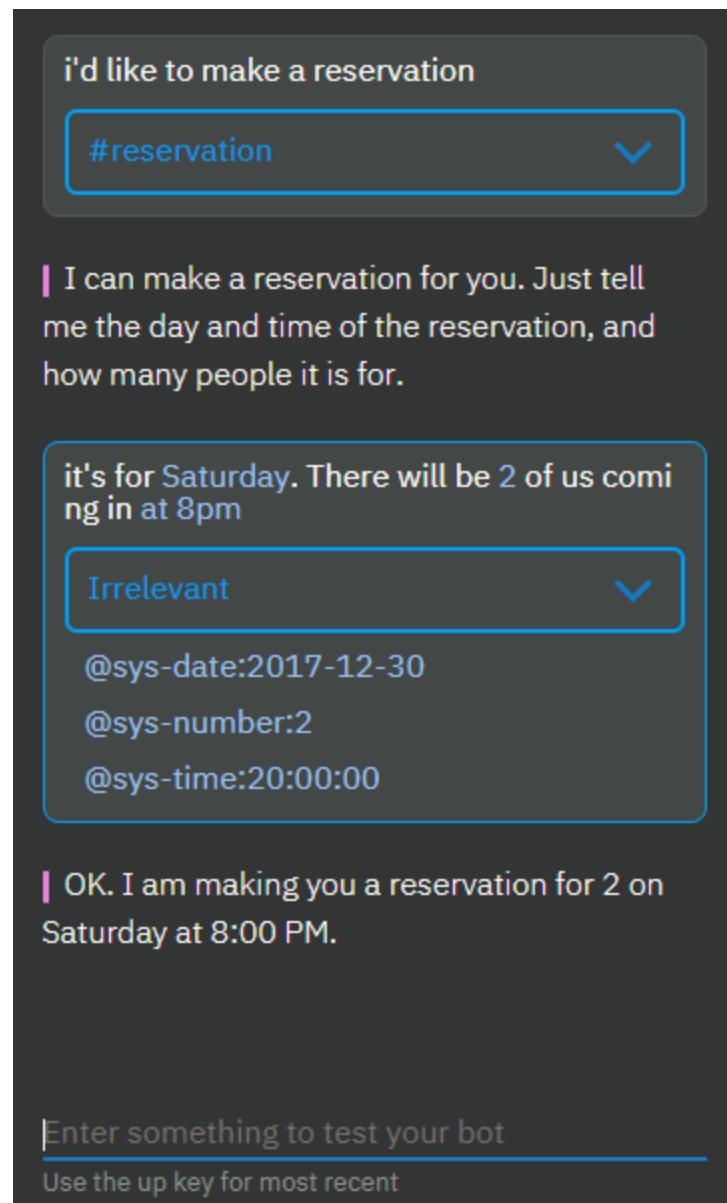
- Back in the node edit view, scroll down to the newly-added **If no slots are pre-filled, ask this first** field. Add the following initial prompt for the node, **I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for.**
- Click **X** to close the node edit view.
- Test this change from the "Try it out" pane. Open the pane, and then click **Clear** to nullify the slot context variable values from the previous test.

4. Enter `i'd like to make a reservation.`

The dialog now responds with, `I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for.`

5. 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, your assistant 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:

Speaker	Utterance
You	<code>i want to make a reservation</code>
Watson	<code>I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for.</code>
You	<code>We want to dine May 23 at 8pm. There will be 0 guests.</code>
Watson	<code>How many people will be dining?</code>
You	<code>0</code>
Watson	<code>How many people will be dining?</code>

[Script details](#)

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 >= 0`.
3. Now, you will change the context variable that is stored for the number.

The slot condition is used both to check the user input for a number value, and to store that number in a context variable. Now that you have edited the condition in the **Check for** field, you will find mentions of the number zero.

However, you want to save only the number, and store it in the context variable. Therefore, open the slot to edit it by clicking the **Edit slot**  icon. From the **Options**  menu, open the JSON editor.

4. Change the context variable value.

The value will look like this:

```
{  
  "context": {  
    "guests": "@sys-number >= 0"  
  }  
}
```

Change it to look like this:

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

5. Save your changes.

You must edit the context variable value in the JSON editor. Do not edit the value in the slot's **Check for** field. The **Check for** field must remain set to `@sys-number >= 0`. { :important }

When you edit the value in the JSON editor, you are effectively changing only what to save in the context variable. However, you do not want to change what to look for in the input. These two values will be different. That is how you want it to be.

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:

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.

Script details

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 treats zeros properly. Of course, you might not want the node to accept a 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.

- From the **Options** menu in the *Configure slot 1* header, select **Enable conditional responses**.
- In the **Found** section, add a conditional response by clicking the **Edit response** icon.
- Add the following condition and response to check whether the date that the user specifies falls before today:

Condition	Response	Action
`@sys-date.before(now())`	You cannot make a reservation for a day in the past.	Clear slot and prompt again

Slot 1 conditional response 1 details

- Add a second conditional response that is displayed if the user provides a valid date. This type of simple confirmation lets the user know that her response was understood.

Condition	Response	Action
'true'	\$date it is	Move on

Slot 1 conditional response 2 details

- From the edit view of the node with slots, click the **Edit slot** icon for the **@sys-time** slot.
- From the **Options** menu in the *Configure slot 2* header, select **Enable conditional responses**.
- In the **Found** section, add a conditional response by clicking the **Edit response** icon.
- Add the following conditions and responses to check whether the time that the user specifies falls within the allowed time window:

Condition	Response	Action
'@sys-time.after('21:00:00')'	Our last seating is at 9 PM.	Clear slot and prompt again
'@sys-time.before('09:00:00')'	Our first seating is at 9 AM.	Clear slot and prompt again

Slot 2 conditional response details

- Add a third conditional response that is displayed if the user provides a valid time that falls within the window. This type of simple confirmation lets the user know that her response was understood.

Condition	Response	Action
'true'	Ok, the reservation is for \$time.	Move on

Slot 2 conditional response 3 details

- 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.
- From the edit view of the node with slots, click the **Edit slot** icon for the **@sys-number** slot.
- From the **Options** menu in the *Configure slot 3* header, select **Enable conditional responses**.
- In the **Found** section, add a conditional response by clicking the **gear** icon, and then add the following condition and response:

Condition	Response	Action
'@sys-number == 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 ` to `.	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.

- `#yes`

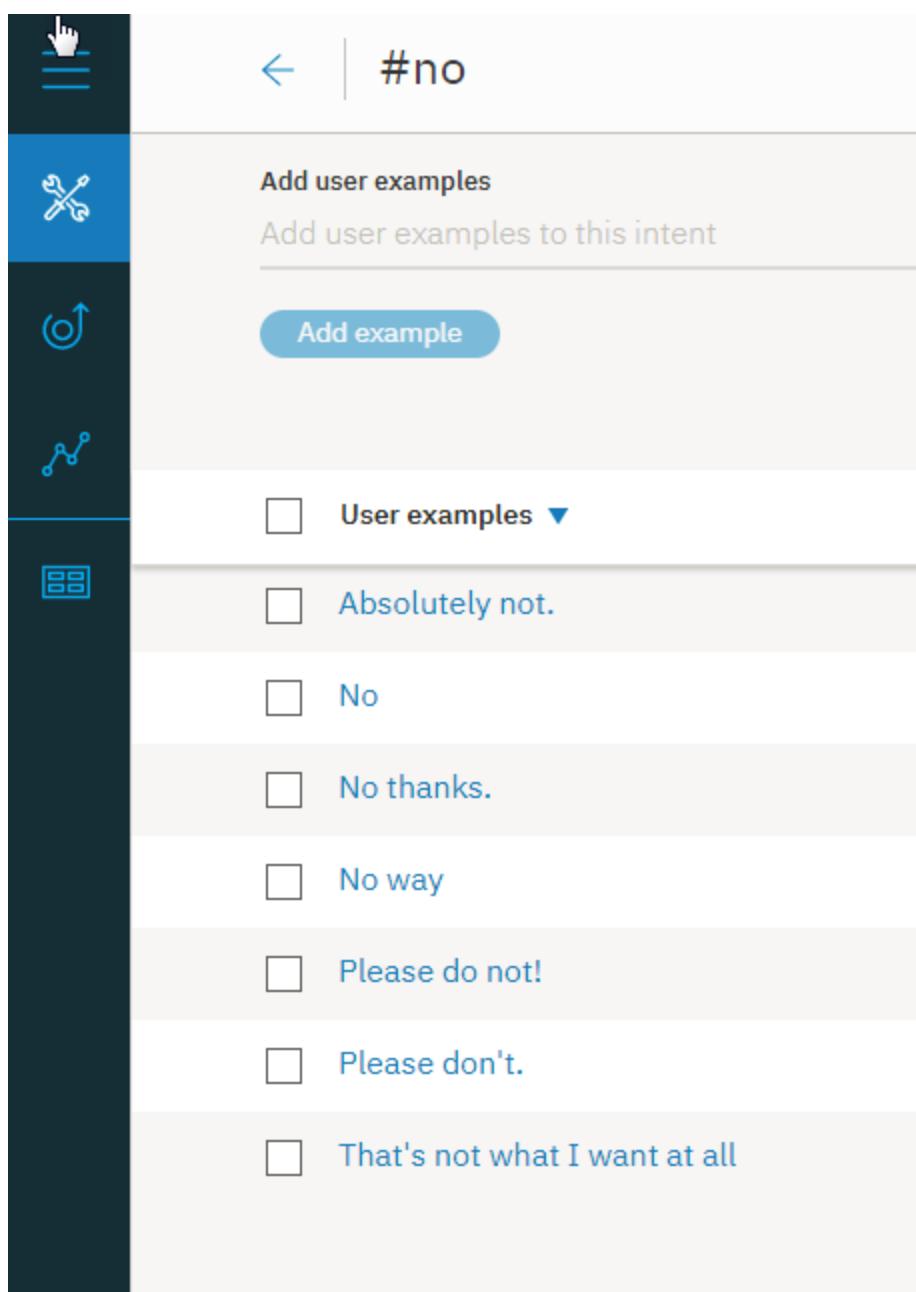
```
Yes
Sure
I'd like that
Please do
Yes please.
Ok
That sounds good.
```

The screenshot shows the Watson Assistant interface with the '#yes' intent selected. The left sidebar has icons for Home, Skills, Intents, Entities, and Flows. The main area shows the intent name '#yes' with a back arrow. Below it are buttons for 'Add user examples' and 'Add user examples to this intent'. A large blue button labeled 'Add example' is prominent. Below these are sections for 'User examples' and a list of example utterances:

- I'd like that
- Ok
- Please do
- Sure
- That sounds good.
- Yes
- Yes please.

- `#no`

```
No
No thanks.
Please don't.
Please do not!
That's not what I want at all
Absolutely not.
No way
```



1. Return to the **Dialog** tab, and then click to edit the node with slots. Click **Add slot** to add a fourth slot, and then specify the following values for it:

Check for	Save it as	If not present, ask
`(#yes #no) && slot_in_focus`	\$confirmation	I'm going to reserve you a table for \$guests on \$date at \$time. Should I go ahead?

Confirmation slot details

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.

2. Click the **Edit slot** icon. From the **Options :** menu in the *Configure slot 4* header, select **Enable conditional responses**.
3. 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.

```
{
  "output": {
    "text": {
      "values": [
        "Alright. Let's start over. I'll try to keep up this time."
      ]
    }
  },
  "context": {
    "date": null,
    "time": null,
    "guests": null
  }
}
```

4. Click **Back**, and then click **Save**.

5. 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.

Condition	Response
`true`	Respond with Yes to indicate that you want the reservation to be made as-is, or No to indicate that you do not.

Not found response details

6. Click **Save**.

7. 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.

8. Replace the **true** condition that is specified in the JSON snippet for the last conditional response in the @sys-date slot with **!(\$time && \$guests)**. For example:

Condition	Response	Action
`!(\$time && \$guests)`	\$date it is	Move on

Slot 1 conditional response 2 details

9. Replace the **true** condition that is specified in the JSON snippet for the last conditional response in the @sys-time slot with **!(\$date && \$guests)**. For example:

Condition	Response	Action
`!(\$date && \$guests)`	Ok, the reservation is for \$time.	Move on

Slot 2 conditional response 3 details

10. Replace the **true** condition that is specified in the JSON snippet for the last conditional response in the @sys-number slot with **!(\$date && \$time)**. For example:

Condition	Response	Action
`!(\$date && \$time)`	Ok. The reservation is for \$guests guests.	Move on

Slot 3 conditional response 2 details

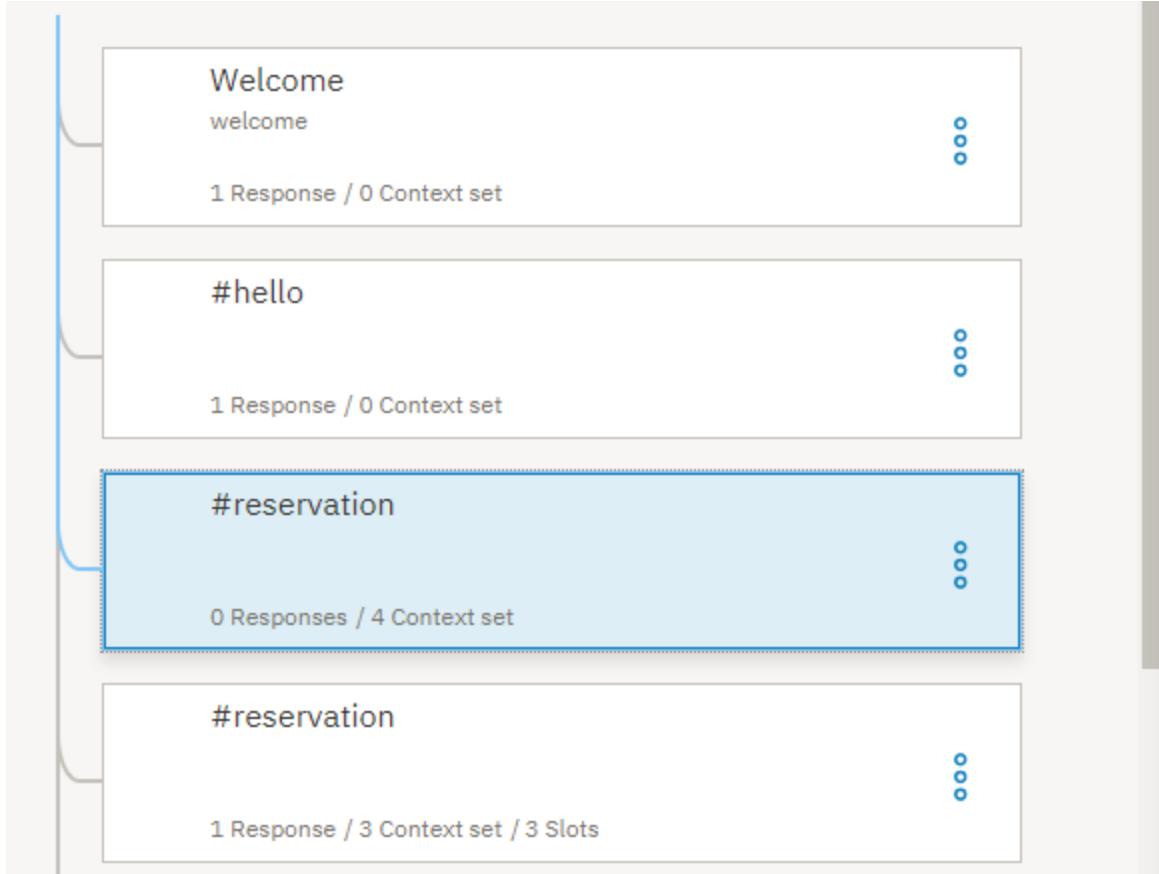
If you add more slots later, you must edit these conditions to account for the associated context variables for the additional slots. If you do not include a confirmation slot, you can specify **!all_slots_filled** only, and it would remain valid no matter how many slots you add later.

Step 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.

- From the tree view of the dialog, click the **More :** icon on the node with slots, and then select **Add node above**.
- 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.)
- 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:

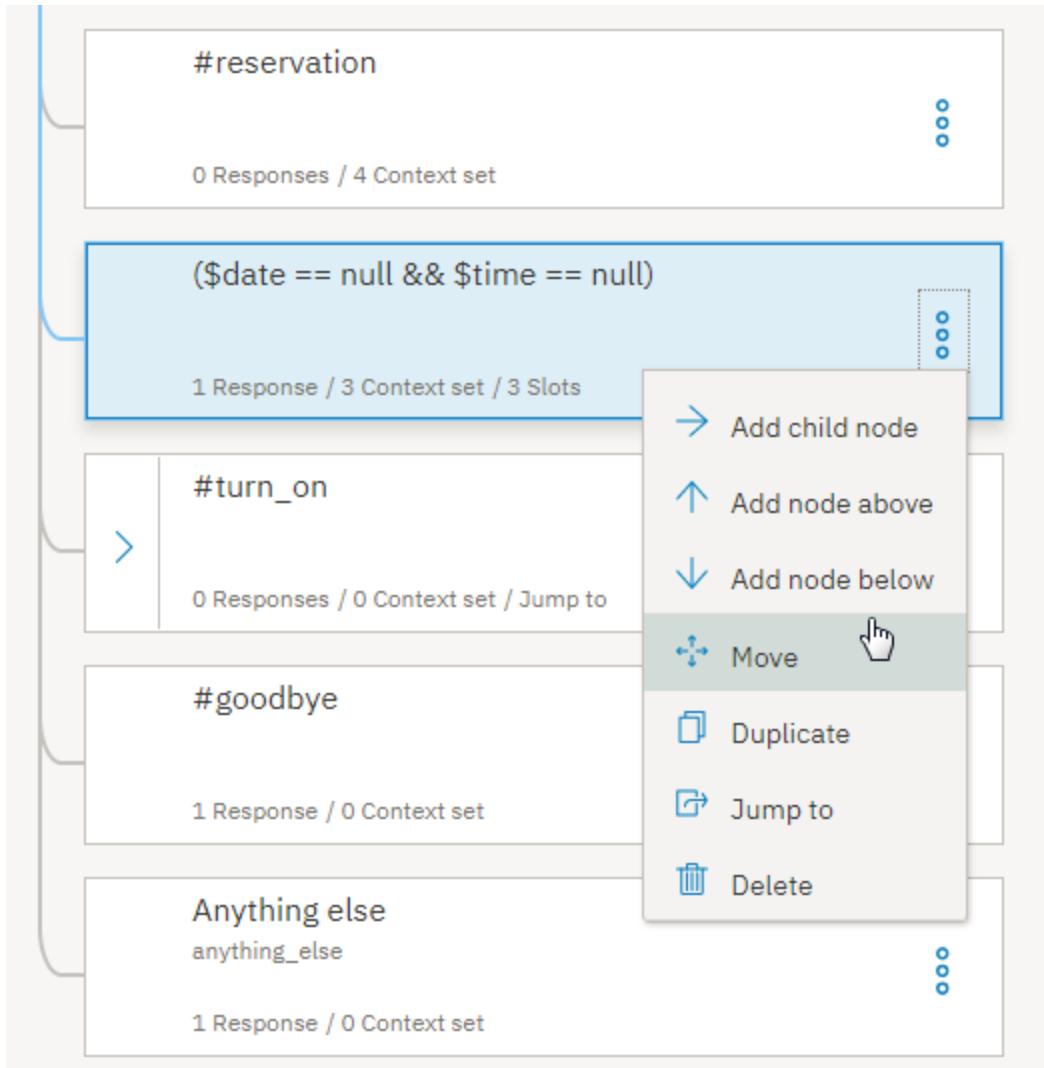
#reservation - +

Then respond with:

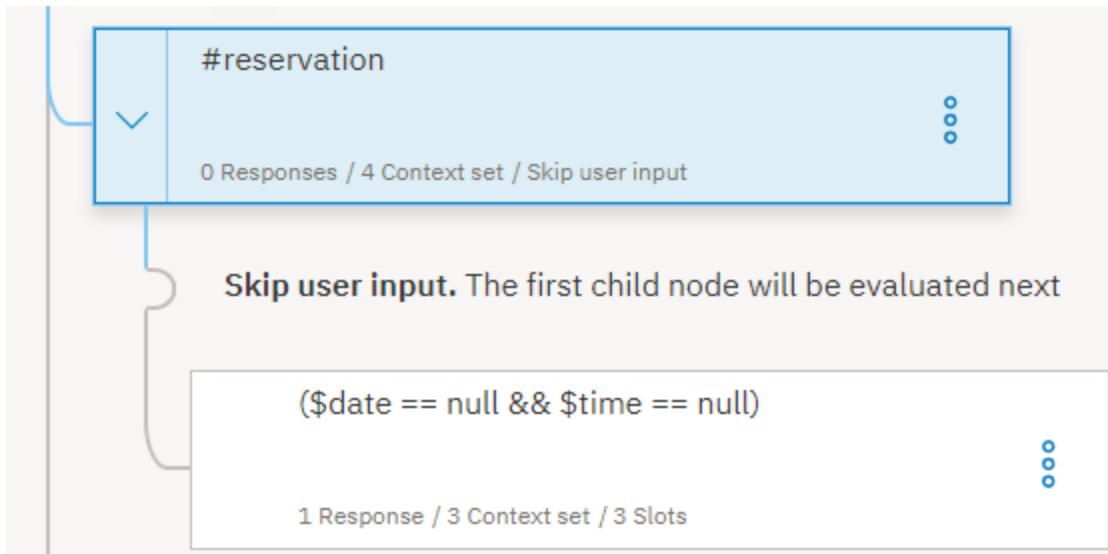
```

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 X.
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's a header with a back arrow and the intent name. Below the header, there's a section for 'Add user examples' with a button to 'Add example'. A list of examples is shown in a scrollable area, each preceded by a checkbox:

- User examples ▾
- 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:

 Manage handlers

Check for	Save it as	If not present, ask	Type		
1 <code>@sys-date</code>	<code>\$date</code>	<code>What day would you</code>	Required		
2 <code>@sys-time</code>	<code>\$time</code>	<code>What time do you w:</code>	Required		
3 <code>@sys-number</code>	<code>\$guests</code>	<code>How many people w</code>	Required		
4 <code>(#yes #no) && sl</code>	<code>\$confirmation</code>	<code>I'm going to reserve</code>	Required		

4. Add the following values to the fields.

Condition	Response	Action
<code>'#exit'</code>	Ok, we'll stop there. No reservation will be made.	Skip to response

Node-level handler details

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**, set the **Multiple conditioned responses** switch to turn it **On**, and then click **Apply**.

Customize "`($date == null && $time == null)`"

Slots 

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 

Enable multiple responses so that your bot can provide different responses to the same input, based on other conditions.

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.

Condition	Response

```
'has_skipped_slots'
```

I look forward to helping you with your next reservation. Have a good day.

Node-level conditional response details

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 your assistant 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.

Speaker	Utterance
You	i want to make a reservation
Watson	I can make a reservation for you. Just tell me the day and time of the reservation, and how many people it is for.
You	it's for 5 people
Watson	Ok. The reservation is for 5 guests. What day would you like to come in?
You	Nevermind
Watson	Ok, we'll stop there. No reservation will be made. I look forward to helping you with your next reservation. Have a good day.

Script details

Step 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 existing `"context"` block, after 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.

Condition

Response

`true`

Please specify the time that you want to eat. The restaurant seats people between 9AM and 9PM.

Not found response details

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.

Condition

Response

`\$counter > 1`

You seem to be having trouble choosing a time. I will make the reservation at 8PM for you.

Not found response details

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. See [Making programmatic calls from a dialog node](#) for more details.

In the logic that calls the reservation service, be sure to check for `has_skipped_slots` and do not continue with the reservation if it is present.

Summary

In this tutorial you tested a node with slots and made changes that optimize how it interacts with real users. For more information about this subject, see [Gathering information with slots](#).

Next steps

Deploy your dialog skill by first connecting it to an assistant, and then deploying the assistant. There are several ways you can do this. See [Adding integrations](#) for more details.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Understanding dialog skill digressions

In this tutorial, you 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 takes 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 dialog skill

First, import the *Digression showcase* dialog skill 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 dialog skill.

Step 2: Temporarily digressing away from dialog

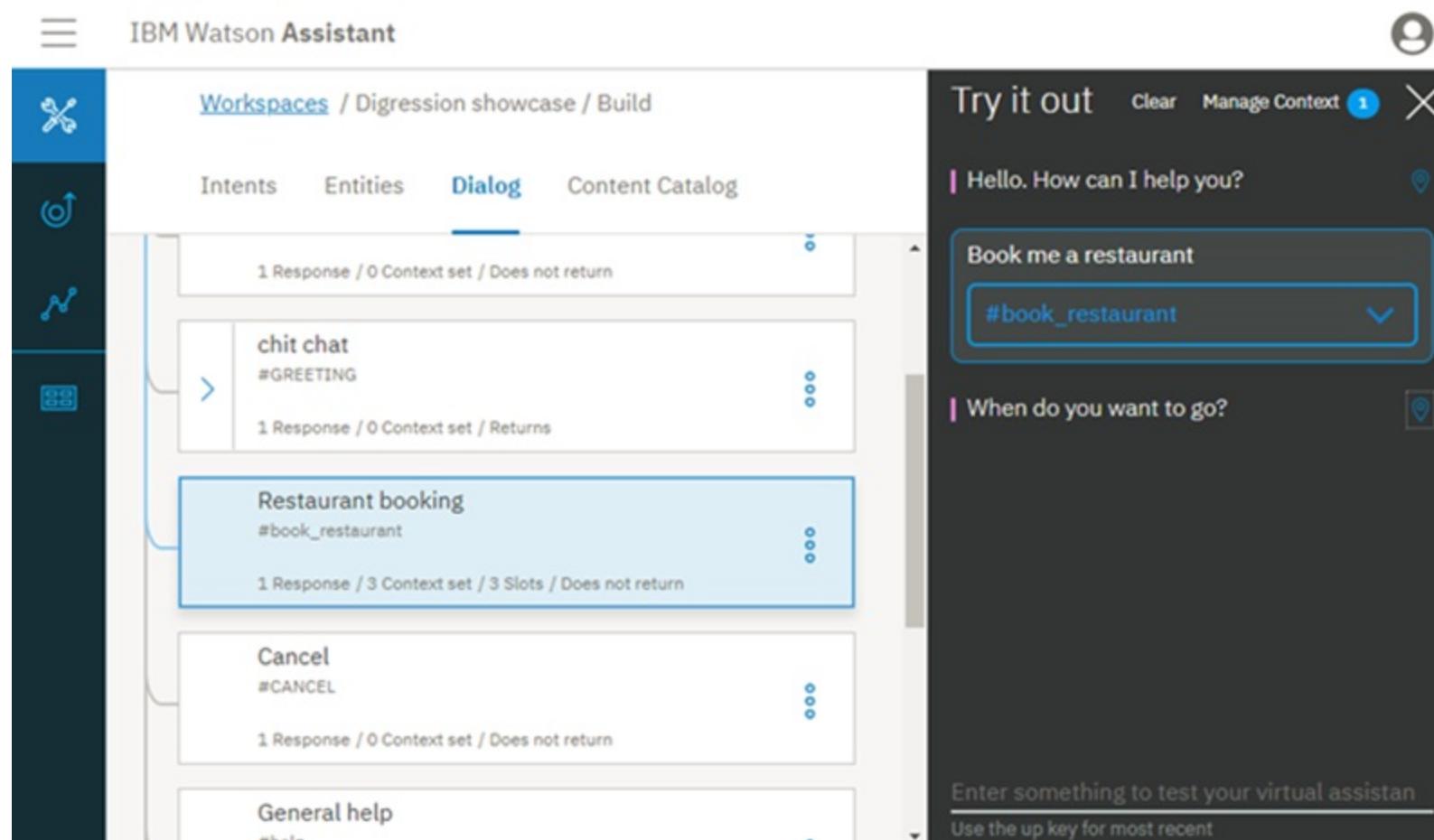
Digressions allow users to break away from a dialog branch to temporarily change the topic before they return to the original dialog flow. In this step, you will start to book a restaurant reservation, then digress away to ask for the restaurant's hours. After providing the opening hours information, your assistant 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.

Your assistant 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.



5. Type **Tomorrow**.

Your assistant 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.** Your assistant 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, there's a sidebar with icons for workspaces, intents, entities, dialog, and content catalog. The main area shows a list of nodes under the 'Dialog' tab:

- 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

To the right, a 'Try it out' panel is open, showing a conversation:

- User: What time do you want to go?
- Assistant: The restaurant is open from 8:00 AM to 10:00 PM.
- User: What time do you close?
- Assistant: #restaurant_opening_hours
- Assistant: The restaurant is open from 8:00 AM to 10:00 PM.
- User: What time do you want to go?
- Assistant: 8pm
- Assistant: Irrelevant
- Assistant: @sys-time:20:00:00

At the bottom of the panel, there's a text input field: "Enter something to test your virtual assistant" and a note: "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 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](#)

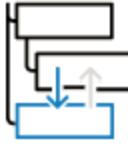
✓ Digressions can go away from this node ⓘ

 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. Set the **Allow digressions away** switch 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**.

Your assistant responds with a prompt for the day to reserve, **When do you want to go?**

7. Type **Tomorrow**.

Your assistant responds with a prompt for the time to reserve, **What time do you want to go?**

8. Ask, **What time do you close?**

Your assistant recognizes that the question triggers the `#restaurant_opening_hours` intent, but ignores it and displays the prompt that is 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 your assistant digressed away from for the current node to be processed. To demonstrate this configuration, you will change the digression setting for the restaurant hours node. In Step 2, you saw that after you digressed away from the restaurant booking node to go to the restaurant opening hours node, your assistant 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 your assistant 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**.

Your assistant responds by saying, **We are always looking for talented people to add to our team. What type of job are you interested in?**

- Instead of answering this question, ask the bot an unrelated question. Type **What time do you open?**

Your assistant digresses away from the Job opportunities node to the Restaurant opening hours node to answer your question. Your assistant 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. Your assistant does not return to the dialog that was in progress because you changed the setting on the **Restaurant opening hours** node to not return.

The screenshot shows the IBM Watson Assistant interface. On the left, the 'Dialog' tab is selected in the workspace navigation bar. A tree view of dialog nodes is displayed:

- Restaurant opening hours** (#restaurant_opening_hours): 1 Response / 0 Context set / Does not return
- Job openings** (#job_opportunities): 1 Response / 0 Context set / Returns
 - @job_role:(wait staff)**: 1 Response / 0 Context set
 - @job_role:greeter**: 1 Response / 0 Context set
 - @job_role:chef**: 1 Response / 0 Context set / Return not allowed

On the right, the 'Try it out' panel shows a conversation:

- User: I'm looking for a job
- Assistant: #job_opportunities
- User: We are always looking for talented people to add to our team. What type of job are you interested in?
- Assistant: The restaurant is open from 8:00 AM to 10:00 PM.

Congratulations! You 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](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Adding more assistants](#) and [Renaming or deleting assistants](#).

Creating an assistant

Create an assistant with the skills it needs to address the business goals of your customers.

To learn more about what an assistant is first, see [Assistants](#).

Follow these steps to create an assistant:

- Click the **Assistants** icon
- Click **Create assistant**.
- Add details about the new assistant:
 - Name**: A name no more than 100 characters in length. A name is required.
 - Description**: An optional description no more than 200 characters in length.
- Click **Create assistant**.
- Add a skill to the assistant.

Note: You can choose to add an existing skill or create a new one.

See [Adding a skill to an assistant](#).

Assistant limits

The number of assistants you can create depends on your Watson Assistant [plan type](#).

After 30 days of inactivity, an unused assistant in a Lite plan service instance might be deleted to free up space. See [Changing the inactivity timeout setting](#) for more information on the subject.

You can connect one skill of each type to your assistant. The number of skills you can build differs depending on the plan you have. See [Skill limits](#) for more details.

If you want to create an assistant in a different service instance, you can switch to another service instance. See [Switching the service instance](#).

Deleting an assistant

When you delete an assistant, any integrations that you defined for the assistant are automatically deleted also. Skills that you added to the assistant are not deleted.

To delete an assistant, follow these steps:

1. From the Assistants page, find the assistant that you want to delete.
2. Click the  icon, and then choose **Delete**. Confirm the deletion.

Renaming an assistant

You can change the name of an assistant and its associated description after you create the assistant.

To rename an assistant, follow these steps:

1. From the Assistants page, find the assistant that you want to rename.
2. Click the  icon, and then choose **Rename**.
3. Edit the name, and then click **Save**.

Changing the skill that is associated with the assistant

You can add one conversation skill (actions or dialog) and one search skill to an assistant. If you want to change a skill that your assistant is using, you can swap one conversation skill or one search skill for another skill.

1. From the Assistants page, open the assistant.
2. Click the  icon for the skill you want to swap, and then choose **Swap skill**.

To swap the current dialog skill for a different version of the skill, choose **Change skill version**.

3. Choose an existing skill to use instead or [create a skill](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Adding more assistants](#) and [Renaming or deleting assistants](#).

Creating an assistant

Create an assistant with the skills it needs to address the business goals of your customers.

To learn more about what an assistant is first, see [Assistants](#).

Follow these steps to create an assistant:

1. Click the **Assistants** icon .
2. Click **Create assistant**.
3. Add details about the new assistant:
 - **Name:** A name no more than 100 characters in length. A name is required.
 - **Description:** An optional description no more than 200 characters in length.
4. Click **Create assistant**.
5. Add a skill to the assistant.

Note: You can choose to add an existing skill or create a new one.

See [Adding a skill to an assistant](#).

Assistant limits

The number of assistants you can create depends on your Watson Assistant [plan type](#).

After 30 days of inactivity, an unused assistant in a Lite plan service instance might be deleted to free up space. See [Changing the inactivity timeout setting](#) for more information on the subject.

You can connect one skill of each type to your assistant. The number of skills you can build differs depending on the plan you have. See [Skill limits](#) for more details.

If you want to create an assistant in a different service instance, you can switch to another service instance. See [Switching the service instance](#).

Deleting an assistant

When you delete an assistant, any integrations that you defined for the assistant are automatically deleted also. Skills that you added to the assistant are not deleted.

To delete an assistant, follow these steps:

1. From the Assistants page, find the assistant that you want to delete.
2. Click the  icon, and then choose **Delete**. Confirm the deletion.

Renaming an assistant

You can change the name of an assistant and its associated description after you create the assistant.

To rename an assistant, follow these steps:

1. From the Assistants page, find the assistant that you want to rename.
2. Click the  icon, and then choose **Rename**.
3. Edit the name, and then click **Save**.

Changing the skill that is associated with the assistant

You can add one conversation skill (actions or dialog) and one search skill to an assistant. If you want to change a skill that your assistant is using, you can swap one conversation skill or one search skill for another skill.

1. From the Assistants page, open the assistant.
2. Click the  icon for the skill you want to swap, and then choose **Swap skill**.

To swap the current dialog skill for a different version of the skill, choose **Change skill version**.

3. Choose an existing skill to use instead or [create a skill](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Adding dialog](#).

Adding skills to your assistant

Customize your assistant by adding to it the skills it needs to satisfy your customers' goals.

Conversational skills return responses that are authored by you to answer common questions, while a search skill searches for and returns passages from existing self-service content.

You can add the following types of skills to your assistant:

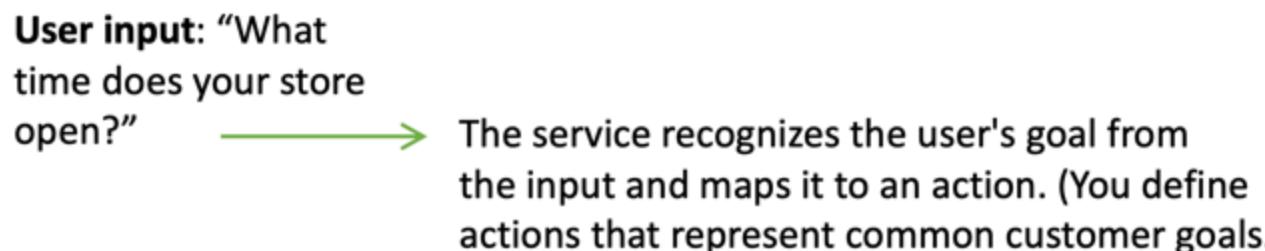
- **Conversational skills:** Understand and address questions or requests that your customers typically ask about. You provide information about the subjects or tasks that your users need help with, and how they ask about them, and the product dynamically builds a machine learning model that is tailored to understand the same and similar user requests.
 - **Actions skill:** Offers a simple interface where anyone can build a conversational flow for your assistant to follow. The complex process of training data creation occurs behind the scenes automatically. [Learn more](#)
 - **Dialog skill:** Offers a set of editors that you use to define both your training data and the conversation. The conversation is represented as a dialog tree. You use the graphical dialog editor to create a script of sorts for your assistant to read from when it interacts with your customers. The dialog keys off the common customer goals that you teach it to recognize, and provides useful responses. [Learn more](#)

- **Search skill** PlusEnterprise: Leverages information from existing corporate knowledge bases or other collections of content authored by subject matter experts to address unanticipated or more nuanced customer inquiries. For a given user query, uses the IBM Watson® Discovery service to search a data source of your self-service content and return an answer. [Learn more](#)

Actions skill

An actions skill contains actions that represent the tasks you want your assistant to help your customers with.

Each action contains a series of steps that represent individual exchanges with a customer. Building the conversation that your assistant has with your customers is fundamentally about deciding which steps, or which user interactions, are required to complete an action. After you identify the list of steps, you can then focus on writing engaging content to turn each interaction into a positive experience for your customer.



Action: Operating hours
You define an answer to questions about hours by adding a step.

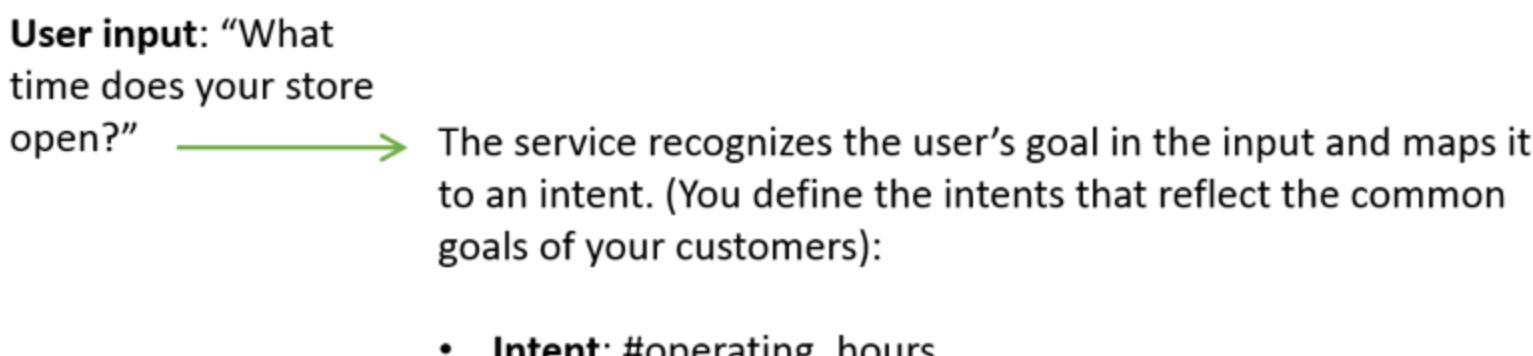
Step Text response: "Our stores are open from 9AM to 10pm."

Dialog skill

A dialog skill contains the training data and logic that enables an assistant to help your customers. It 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. The name of an intent is always prefixed with the # character. To train the dialog skill to recognize your intents, you supply lots of examples of user input and indicate which intents they map to.
- A *content catalog* is provided that contains prebuilt common intents 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 an intent that greets the user and does other useful things, like end the conversation.
- **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 editor to create conversations with users, providing responses based on the intents and entities that you recognize in their input.

Basic implementation



You define an answer to questions about hours by adding a **dialog node** like this:

Condition: #operating_hours Text response: "Our stores are open from 9am to 10pm."

To enable your dialog skill to handle more nuanced questions, define entities and reference them from your dialog.

- **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. The name of an entity is always prefixed with the @ character.

You can train the skill to recognize your entities by providing entity term values and synonyms, entity patterns, or by identifying the context in which an entity is typically used in a sentence. To fine tune your dialog, go back and add nodes that check for entity mentions in user input in addition to intents.

More complex implementation

User input: "At what time does your Boston store open?" →

The service identifies the following items in the user input:

- **Intent:** #operating_hours
- **Entity:** "Boston" is a @sys-location entity mention

You can define a more precise answer by adding a **dialog node** like this:

Condition: #operating_hours AND @sys-location == "Boston"
Text response: "Our flagship store in downtown Boston is open from 9am to 10pm."

As you add information, the skill 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.

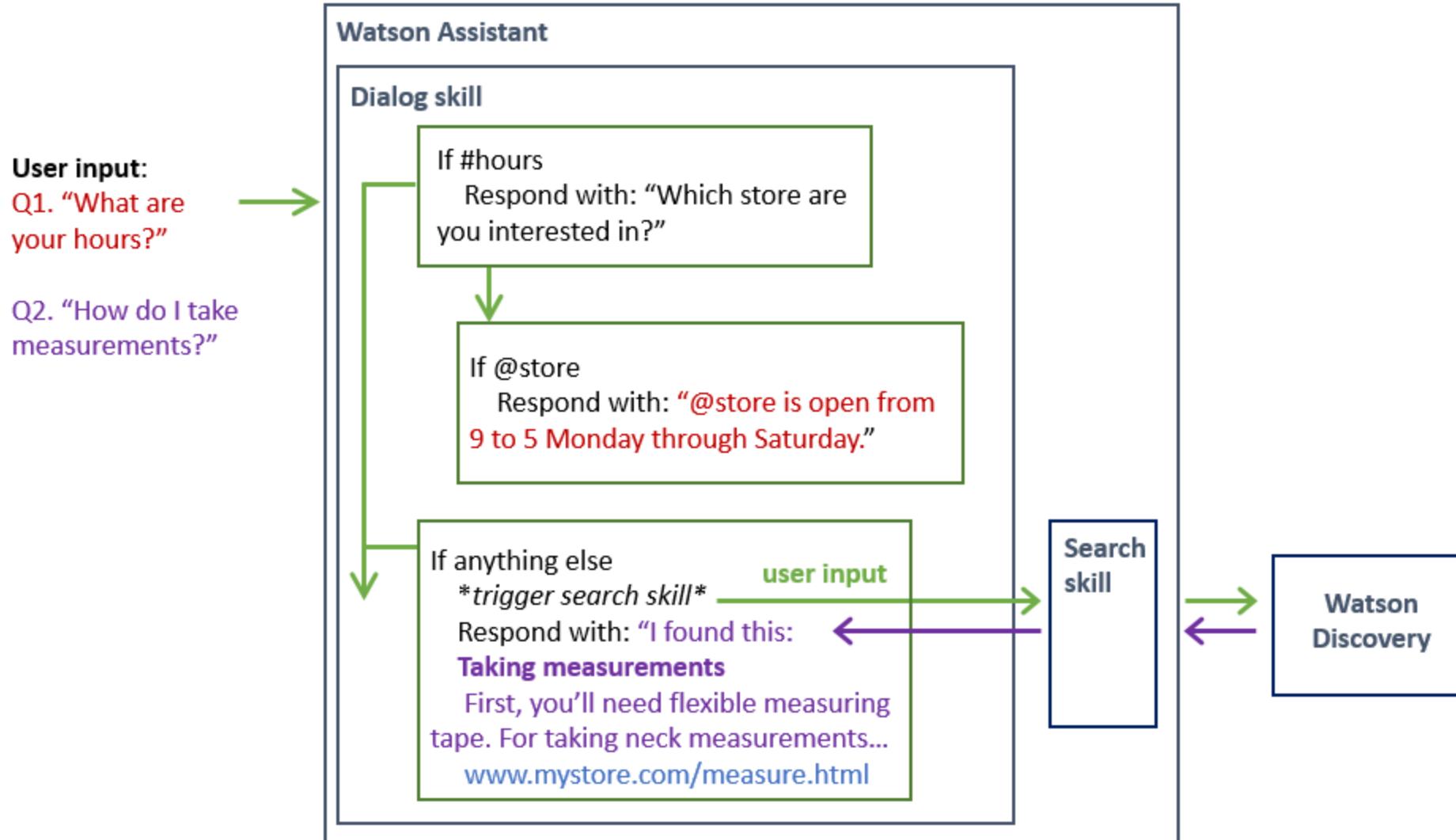
Search skill

PlusEnterprise

When Watson Assistant doesn't have an explicit solution to a problem, it routes the user question to a search skill to find an answer from across your disparate sources of self-service content. The search skill interacts with the IBM Watson® Discovery service to extract this information from a configured data collection.

If you already use the Discovery service, you can mine your existing data collections for source material that you can share with customers to address their questions.

The following diagram illustrates how user input is processed when both a dialog skill and a search skill are added to an assistant. Any questions that the dialog is not designed to answer are sent to the search skill, which finds a relevant response from a Discovery data collection.



Skill limits

The number of skills you can create depends on your Watson Assistant plan type. Any sample dialog skills that are available for you to use do not count

toward your limit unless you use them. A skill version does not count as a skill.

Plan	Maximum number of skills of each type per service instance
Enterprise	100
Premium (legacy)	100
Plus	50
Trial	50
Standard (legacy)	20
Lite	5

[Plan details](#)



Note: After 30 days of inactivity, an unused skill in a Lite plan service instance might be deleted to free up space.

Building a conversation with a dialog tree

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Adding dialog](#).

Adding a dialog skill

The natural-language processing for the Watson Assistant service is defined in a *dialog skill*, which is a container for all of the artifacts that define a conversation flow.

You can add one dialog skill to an assistant. See [Skill limits](#) for information about limits per plan.

Create the dialog skill

You can create a skill from scratch, use a sample skill that is provided by IBM, or upload a skill from a JSON file.

To add a skill, complete the following steps:

1. From the assistant where you want to add the skill, click **Add an actions or dialog skill**.
2. Do one of the following:
 - o To create a new dialog skill, remain on the *Create skill* tab.
 - o To add the dialog sample skill that is provided with the product as a starting point for your own skill or as an example to explore before you create one yourself, open the *Use sample skill* tab, and then click the sample labeled **TYPE: Dialog**. You can skip the remaining steps.
 - o To add a skill that was downloaded previously, you can upload it as a JSON file. Open the *Upload skill* tab. Drag a file or click **Drag and drop file here or click to select a file** and select the JSON file you want to upload.

 **Important:** The uploaded JSON file must use UTF-8 encoding, without byte order mark (BOM) encoding. The JSON cannot contain tabs, newlines, or carriage returns.

 **Tip:** The maximum size for a skill JSON file is 10 MB. If you need to upload a larger skill, consider using the REST API. For more information, see the [API Reference](#).

Click **Upload**.

If you have trouble uploading a skill, see [Troubleshooting skill upload issues](#).

- o If you have created a dialog skill already, the *Add existing skill* tab is displayed, and you can click to add an existing skill.
3. Specify the details for the skill:
 - o **Name:** A name no more than 64 characters in length. A name is required.
 - o **Description:** An optional description no more than 128 characters in length.
 - o **Language:** The language of the user input the skill will be trained to understand. The default value is English.
 4. For **Skill type**, choose Dialog.
 5. Click **Create skill**.

After you create the dialog skill, it appears as a tile on the Skills page. Now, you can start identifying the user goals that you want the dialog skill to address.

- To add prebuilt intents to your skill, see [Using content catalogs](#).
- To define your own intents, see [Defining intents](#).

The dialog skill cannot interact with customers until it is added to an assistant and the assistant is deployed. See [Creating an assistant](#).

Troubleshooting skill upload issues

Here are some solutions to typical upload issues:

- If you get the message, **Error. Should NOT be shorter than 1 character**, then check whether your skill has a name. If not, add one.
- The **@sys-person** and **@sys-location** system entities are no longer supported. If the skill you are uploading references them in its dialog, an error is displayed. Remove these system entities from your dialog.
- If you receive a message that says the skill contains artifacts that exceed the limits imposed by your service plan, complete the following steps to upload the skill successfully:
 1. Purchase a plan with higher artifact limits.
 2. Create a service instance in the new plan.

3. Upload the skill to the new service instance.
4. If you don't want to keep the higher-level plan, make edits to the skill such that it meets the artifact limit requirements for the plan you want to use going forward.

\$ For information about how to decrease the number of dialog nodes, see [How many nodes are in my dialog?](/docs/assistant?topic=assistant-dialog-tasks#dialog-tasks-count-nodes).

1. Download the edited skill to export it.
2. Try again to upload the edited skill into the original service instance on the plan you want.

Adding the skill to an assistant

You can add one dialog skill to an assistant. You must open the assistant tile and add the skill to the assistant from the assistant configuration page; you cannot choose the assistant that will use the skill from within the skill configuration page. One dialog skill can be used by more than one assistant.

1. From the Assistants page, click to open the tile for the assistant to which you want to add the skill.
2. Click **Add an actions or dialog skill**.
3. Click **Add existing skill**.

Click the skill that you want to add from the available skills that are displayed.

When you add a dialog skill from here, you get the development version. If you want to add a specific skill version, add it from the skill's **Versions** page instead.

Sharing a dialog skill with team members

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



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

To share a dialog skill with other people, you must give them access to the service instance that hosts the skill. Note that the person you invite will be able to access any skill or assistant in this service instance.

1. Click the User  icon in the page header.
2. Make a note of the current service instance name, and then select **Manage Users** from the drop-down.
3. From the navigation pane, click **Users**.

If you gave someone access to a service instance previously, then the person might be listed as an invited user already. To change the person's level of access to the instance, click the menu next to their name, choose **Assign access**, and then click **Assign access to resources**.

4. Click **Invite users**.
5. Add the email address of the person with whom you want to share the instance. You can add more than one person.
6. Expand the *Assign users additional resources* section, and then click **IAM Services**.
7. In the *No access* field, choose Watson Assistant.

The *Account* resource group is used unless you change it.

8. Select at least one region and at least one service instance to share with this user.

Remember, you made a note of the name of the current service instance in an earlier step. You can select it from the list of service instances if you want to give the person access to that service instance only.

9. Give this user the following assignments at a minimum:

- o **Platform access roles:** Viewer
- o **Service access roles:** Writer

For more information about roles, see [Understanding roles](#).

10. Click **Add**.

You can repeat the previous step to invite other users and apply different roles to them.

11. Click **Invite** to finish the process.

If you are editing access for an existing user, click [Assign access](#).

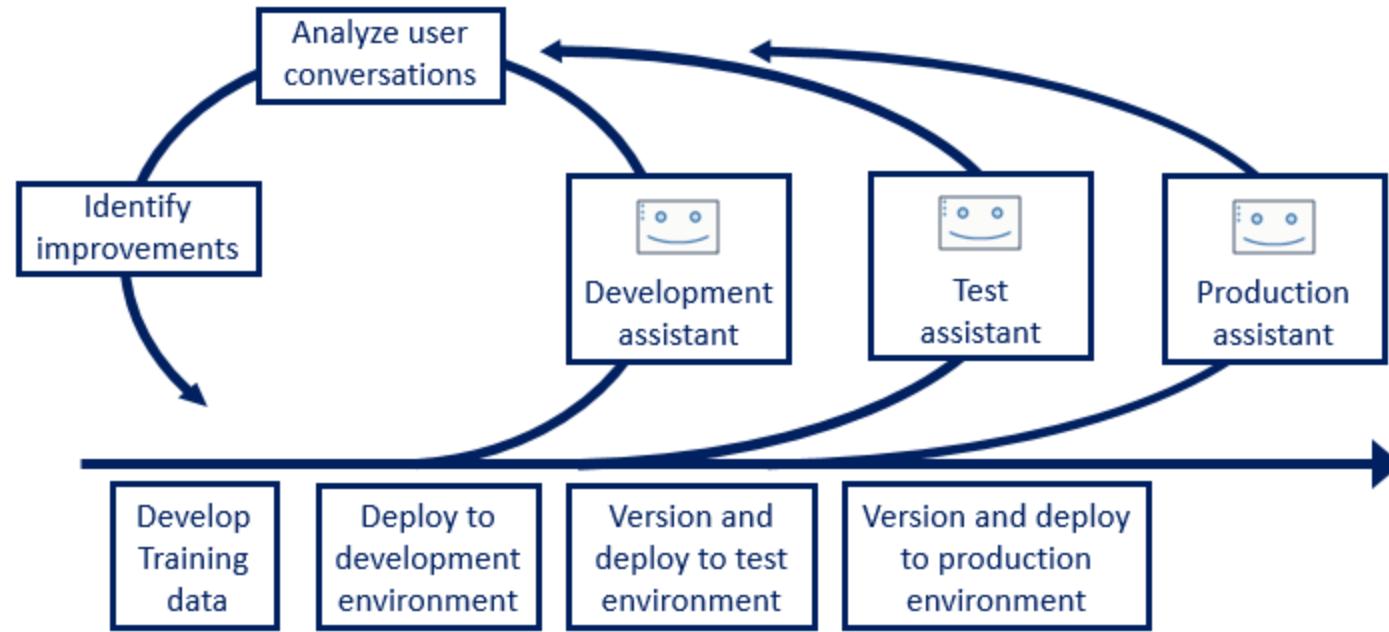
When the people you invite next log in to IBM Cloud®, your account will be included in their list of accounts. If they select your account, they can see your service instance, and open and edit your skills.

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

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Dialog creation workflow](#).

Dialog creation workflow

Use Watson Assistant to leverage AI as you build, deploy, and incrementally improve a conversational assistant.



Development process

The typical workflow for an assistant project includes the following steps:

1. Define a narrow set of key customer needs that you want the assistant to address on your behalf, including any business processes that it can initiate or complete for your customers. Start small.
2. Create intents that represent the customer needs you identified in the previous step. For example, intents such as `#about_company` or `#place_order`.
3. Build a dialog that detects the defined intents and addresses them, either with simple responses or with a dialog flow that collects more information first.
4. Define any entities that are needed to more clearly understand the user's meaning. For example, you might add an `@product` entity that you can use with the `#place_order` intent to understand what product the customer wants to buy.

Mine existing intent user examples for common entity value mentions. Using annotations to define entities captures not only the text of the entity value, but the context in which the entity value is typically used in a sentence.

5. Test each function that you add to the assistant in the "Try it" pane, incrementally, as you go.
6. When you have a working assistant that can successfully handle key tasks, add an integration that deploys the assistant to a development environment. Test the deployed assistant and make refinements.
7. After you build an effective assistant, take a snapshot of the dialog skill and save it as a version.

Saving a version when you reach a development milestone gives you something you can go back to if subsequent changes you make to the skill decrease its effectiveness. See [Creating skill versions](#).

8. Deploy the version of the assistant into a test environment, and test it.
If you use the preview, you can share the URL with others to get their help with testing.
9. Use metrics from the Analytics tab to find areas for improvement, and make adjustments.

If you need to test alternative approaches to addressing an issue, create a version for each solution, so you can deploy and test each one independently, and compare the results.

10. When you are happy with the performance of your assistant, deploy the best version of the assistant into a production environment.
11. Monitor the logs from conversations that users have with the deployed assistant.

You can view the logs for a version of a skill that is running in production from the Analytics tab of a development version of the skill. As you find misclassifications or other issues, you can correct them in the development version of the skill, and then deploy the improved version to production after testing. See [Improving across assistants](#) for more details.

The process of analyzing the logs and improving the dialog skill is ongoing. Over time, you might want to expand the tasks that the assistant can handle for you. Customer needs also change. As new needs arise, the metrics generated by your deployed assistants can help you identify and address them in subsequent iterations.

Defining goals for your assistant to address

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Creating intents](#).

Creating intents

Intents are purposes or goals that are 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.

To learn more about creating intents, watch the following 2 1/2-minute video.



[View video: Intents: Watson Assistant](#)

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 `#` that is added as a prefix to the intent name helps to clearly identify it as an intent.)

- Teach Watson about your intents.

After you decide which business requests that 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 5 examples of utterances that your customers typically use to indicate their goal. For example, `I want to make a purchase`.

Ideally, find real-world user utterance examples that you can extract from existing business processes. The user examples should be tailored to your specific business. For example, if you are an insurance company, a user example might look more like this, `I want to buy a new XYZ insurance plan`.

The examples that you provide are used by your assistant 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.

Creating intents

1. Open your dialog skill. The skill opens to the **Intents** page.
2. Select **Create intent**.
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`



Tip: A number sign `#` is prepended to the intent name automatically to help identify the term as an intent. You do not need to add it.

Keep the name as short as possible. It's easier to read in the "Try it out" pane and conversation logs if you keep the intent name short and concise.

Optionally add a description of the intent in the **Description** field.

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

← | Create intent

Intent name
Name your intent to match a customer's question or goal

Description (optional)

Create intent

5. In the **User example** field, type the text of a user example for the intent. An example can be any string up to 1,024 characters in length. The following utterances might be examples for the `#pay_bill` intent:

- `I need to pay my bill.`
- `Pay my account balance`
- `make a payment`



Tip: To learn about the impact of including references to entities in your user examples, see [How entity references are treated](#).



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

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

← | #pay_bill

Intent name
Name your intent to match a customer's question or goal

Description (optional)

User example
Add unique examples of what the user might say. (Pro tip: Add at least 5 unique examples to help Watson understand your intent.)

Add example **Show recommendations**

7. Repeat the same process to add more examples.



Important: Provide at least five examples for each intent.

8. When you are done adding examples, click to finish creating the intent.

The system begins to train itself on the intent and user examples you added.

Important:

- Intent example data should be representative and typical of data that your users 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 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.

How entity references are treated

When you include an entity mention in a user example, the machine learning model uses the information in different ways in these scenarios:

- [Referencing entity values and synonyms in intent examples](#)
- [Annotated mentions](#)
- [Directly referencing an entity name in an intent example](#)

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. It is a weak relationship, but it does inform the model.

Annotated mentions

As you define entities, you can annotate mentions of the entity directly from your existing intent user examples. A relationship that you identify in this way between the intent and the entity is *not* used by the intent classification model. However, when you add the mention to the entity, it is also added to that entity as new value. And when you add the mention to an existing entity value, it is also added to that entity value as new synonym. Intent classification does use these types of dictionary references in intent user examples to establish a weak reference between an intent and an entity.

For more information about contextual entities, see [Adding contextual entities](#).

Directly referencing an entity name in an intent example



Note: This approach is advanced. If used, it must be used consistently.

You can choose to directly reference entities in your intent examples. For instance, say that you have an entity that is called `@Phone modelName`, which contains values *Galaxy S8*, *Moto Z2*, *LG G6*, and *Google Pixel 2*. When you create an intent, for example `#order_phone`, you might then provide training data as follows:

- Can I get a `@Phone modelName`?
- Help me order a `@Phone modelName`.
- Is the `@Phone modelName` in stock?
- Add a `@Phone modelName` to my order.

[←](#) | #order_phone

Order a phone intent

User example
Add unique examples of what the user might say. (Pro tip: Add at least 5 unique examples)

Type a user example here, e.g. I want to pay my credit card bill

Add example Show recommendations

User examples (4) ↑

- Add a @PhonemodelName to my order
- Can I get a @PhonemodelName
- Help me order a @PhonemodelName
- Is the @PhonemodelName in stock?

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 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 apply 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, the change impacts your previous training. If you do choose to use `@Entity` references, you must replace all previous direct references with `@Entity` references.

Defining one example intent with an `@Entity` that has 10 values that are 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.

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. Click **Try it**.

 Try it

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.



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

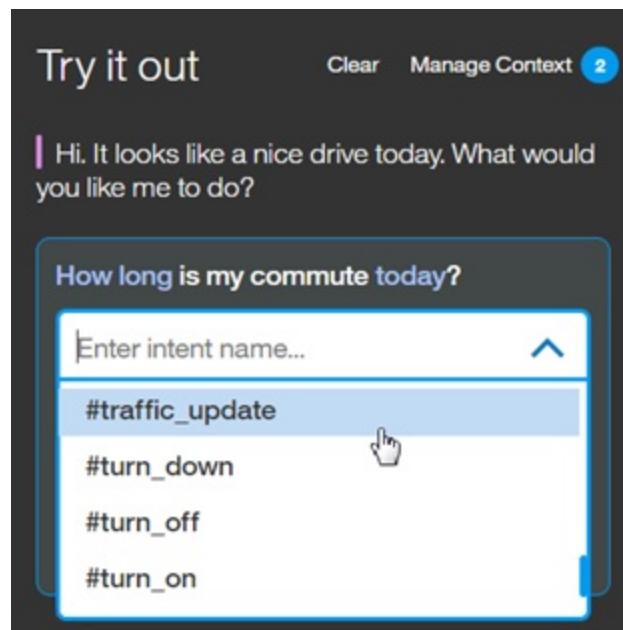
Try it out Clear Manage Context (3)

Watson is training

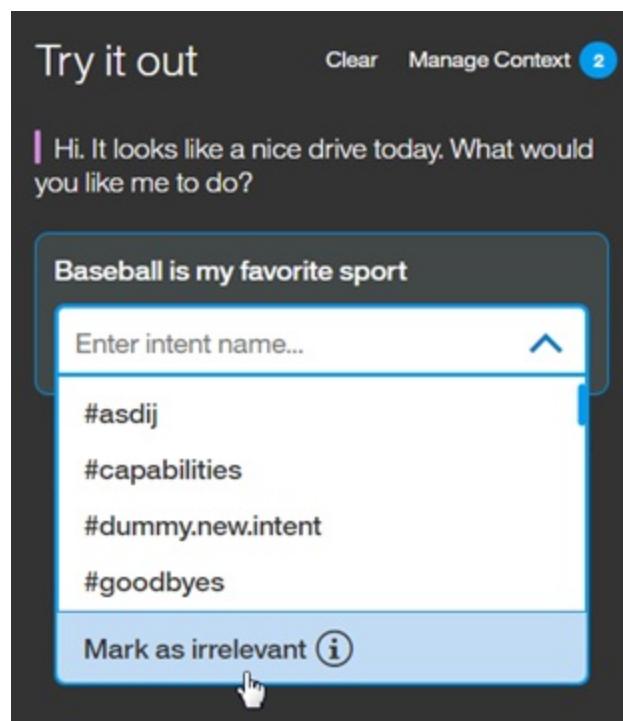
The response indicates which intent was recognized from your input.



3. If the system does 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 any of the intents in your application, you can teach your assistant that by selecting the displayed intent, and then clicking **Mark as irrelevant**.



For more information about this action, see [Teaching your assistant about topics to ignore](#).

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

The Watson Assistant service scores each intent's confidence independently, not in relation to other intents. This approach adds flexibility; multiple intents can be detected in a single user input. It also means that the system might not return an intent at all. If the top intent has a low confidence score (less than 0.2), the top intent is included in the intents array that is returned by the API, but any nodes that condition on the intent are not triggered. If you want to detect the case when no intents with good confidence scores were detected, use the `irrelevant` special condition in your dialog node. See [Special conditions](#) for more information.

As intent confidence scores change, your dialogs might need restructuring. For example, if a dialog node uses an intent in its condition, and the intent's

confidence score starts to consistently drop below 0.2, the dialog node stops being processed. If the confidence score changes, the behavior of the dialog can also change.

Intent limits

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

Plan	Intents per skill	Examples per skill
Enterprise	2,000	25,000
Premium (legacy)	2,000	25,000
Plus	2,000	25,000
Lite, Trial	100	25,000

Plan details

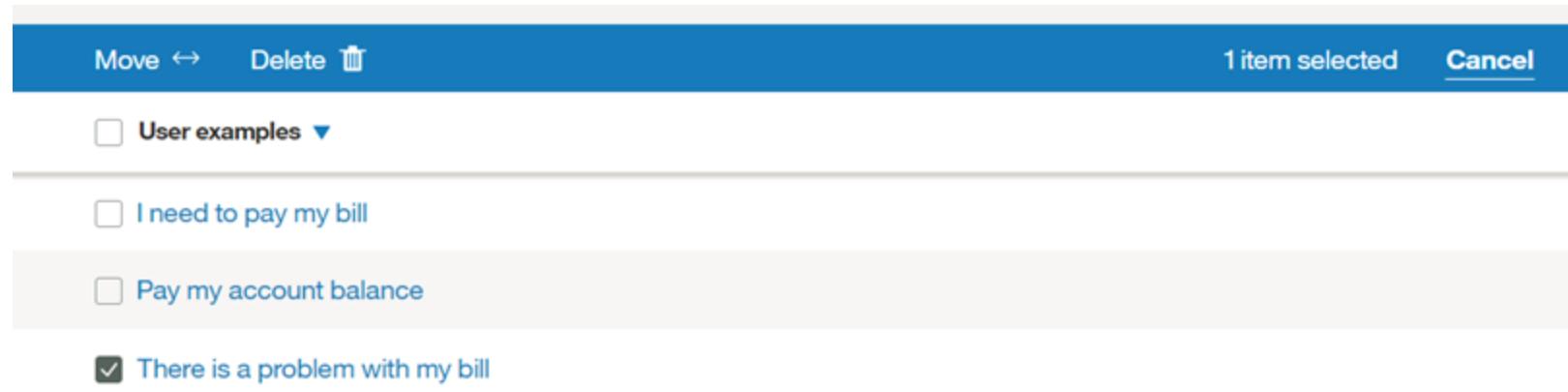
Editing intents

You can click 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.

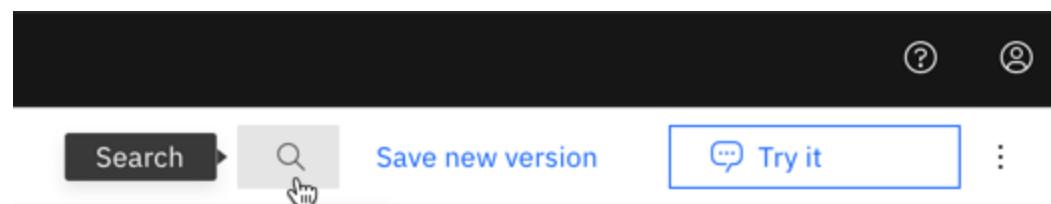
1. To move or delete an example, click the checkbox that is associated with it, and then click **Move** or **Delete**.



Searching intents

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

1. From the **Intents** page, click the *Search* icon.



2. Submit a search term or phrase.

The first time you search for something, you might get a message that says the skill is being indexed. If so, wait a minute, and then resubmit the search term.

Intents that contain your search term are displayed.

The screenshot shows the Watson Assistant interface. At the top, there are buttons for 'Save new version', 'Try it', and a menu icon. Below that is a search bar with the text 'greeting'. A message indicates 'Showing 2 intents containing: 'greeting''. The results list includes '#General_Greetings' (2 matches) and '#greeting' (1 match).

Downloading intents

You can download a number of intents to a CSV file, so you can then upload and reuse them in another Watson Assistant application.

1. Go to the **Intents** page.
 - To download all intents, meaning the intents listed on this and any additional pages, do not select any individual intents. Instead, click the *Download all intents* icon.
 - To download the intents that are listed on the current page only, select the checkbox in the header. This action selects all of the intents on the current page. Click the *Download* icon.
 - To download one or more specific intents, select the intents that you want to download, and then click the *Download* icon.
2. Specify the name and location in which to store the CSV file that is generated, and then click **Save**.

Uploading intents and examples

If you have a large number of intents and examples, you might find it easier to upload them from a comma-separated value (CSV) file than to define them one by one. Be sure to remove any personal data from the user examples that you include in the file.

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. From the **Intents** page, click the *Upload* icon .
3. Drag a file or browse to select a file from your computer.

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

4. Click **Upload**.

The file is validated and uploaded, and the system begins to train itself on the new data.

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

Resolving intent conflicts



Note: This feature is available only to users of paid plans.

The Watson Assistant application detects a conflict when two or more intent examples in *separate* intents are so similar that {{site.data.keassistant_classic_shortnshort}} is confused as to which intent to use.

To resolve conflicts:

- From the **Intents** page, review any intents with conflicts.

The screenshot shows the 'My first skill' interface with the 'Intents' tab selected. A table lists 11 intents. The 'Conflicts' column is highlighted with a red border. The data in the table is as follows:

	Intents (11) ↑	Description	Modified ↑	Conflicts ↑	Examples ↑
	#General_About_You	Request generic personal attribut...	11 days ago	20	
	#General_Agent_Capabilities	Request capabilities of the bot.	11 days ago	30	
	#General_Connect_to_Agent	Request a human agent.	11 days ago	38	
	#General_Ending	End the conversation.	11 days ago	37	
	#General_Greetings	Greet the bot.	11 days ago	1	27
	#General_Human_or_Bot	Ask if speaking to a human or a b...	11 days ago	12	
	#General_Jokes	Request a joke.	11 days ago	17	
	#General_Negative_Feedback	Express unfavorable feedback.	11 days ago	20	
	#General_Positive_Feedback	Express positive sentiment or gra...	11 days ago	19	
	#General_Security_Assurance	Express concerns about the secu...	11 days ago	26	
	#greeting		4 minutes ago	1	5

- Click an intent with a conflict to open it. Find the user example that is causing the conflict, and then click **Resolve conflicts**.

The screenshot shows the detail view for the '#greeting' intent. It includes a section for 'User example' with a text input field and a 'Show recommendations' button. Below is a table of user examples:

	User examples (5) ↑	Added ↑	Conflicts (1) ↑
	Good morning to you!	9 minutes ago	
	Hello	8 minutes ago	
	Hello there!	9 minutes ago	

A blue 'Resolve conflicts' button is located at the bottom right of the example table.

- Choose whether to delete the example from the intent or to move it to another intent.

Watson Conflict Resolution

Conflicts are detected when similar examples are found in more than one intent. Move or delete examples to clarify the meaning of each intent.

Reset

Submit

#greeting #General_Greetings

Move

Examples in Conflict ⓘ

Hello

Similar Examples ⓘ

Hello there!
 How's it going today?
 Howdy
 Good morning to you!

Examples in Conflict ⓘ

Hello

Similar Examples ⓘ

Hello Agent
 Hello I am looking for some help here
 Ok take me back
 Hi there
 Hi advisor
 Hey you
 Hey twin
 Hey there all
 Hey there
 Hey how are you doing

Similar user examples are displayed for each intent. These examples are not necessarily in conflict. They are shown to give you a quick view of the other types of user examples that are defined for each intent. It provides you with context that can help you make a more informed decision.



Tip: Keep each intent as distinct and focused on one goal as possible. If you have two intents with multiple user examples that overlap, maybe you don't need two separate intents. You can move or delete user examples that don't directly overlap into one intent, and then delete the other.

4. To move a user example, click **Move**, and then click the intent where you want to move the example.

Watson Conflict Resolution

Conflicts are detected when similar examples are found in more than one intent. Move or delete examples to clarify the meaning of each intent.

Reset

Submit

#greeting #General_Greetings Move

Examples in Conflict ⓘ

Hello

Examples in Conflict ⓘ

Hello

When deciding where to put an example, look for the intent that has synonymous, or nearly synonymous, examples.

If the exact same example is used by the other intent already, the move action only removes the example from the current intent. It does not add the same example to the other intent twice.

5. After moving or deleting the example, click **Submit** to resolve the conflict.

The screenshot shows the Watson Conflict Resolution interface. At the top, it says "Watson Conflict Resolution" and provides instructions: "Conflicts are detected when similar examples are found in more than one intent. Move or delete examples to clarify the meaning of each intent." Below this are two panels:

- #greeting** panel:
 - Examples in Conflict**: Hello there!, How's it going today?, Howdy, Good morning to you!
 - Similar Examples**: Hello, Hello Agent, Hello I am looking for some help here, Ok take me back, Hi there, Hi advisor, Hey you, Hey twin, Hey there all, Hey there, Hey how are you doing
- #General_Greetings** panel:
 - Examples in Conflict**: Hello
 - Similar Examples**: Hello Agent, Hello I am looking for some help here, Ok take me back, Hi there, Hi advisor, Hey you, Hey twin, Hey there all, Hey there, Hey how are you doing

At the bottom right of the interface is a blue "Submit" button.

The **Reset** reverts your changes. Click the **x** to close the page without submitting your changes.

- Repeat the previous steps to resolve other intents with conflicts.

Deleting intents

You can select a number of intents for deletion.

IMPORTANT: By deleting intents that 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.

- Go to the **Intents** page
 - To delete all intents, meaning the intents listed on this and any additional pages, do not select any individual intents. Instead, click the **Delete all intents** icon.
 - To delete the intents that are listed on the current page only, select the checkbox in the header. This action selects all of the intents that are listed on the current page. Click **Delete**.
 - To delete one or more specific intents, select the intents that you want to delete, and then click **Delete**.

The screenshot shows the Intents page with a modal overlay. The modal title is "1 item selected". It contains a "Delete" button and a "Cancel" button. Below the buttons is a table with the following data:

	Intents (11) ↑	Description	Modify	Conflicts ↑	Examples ↑
<input type="checkbox"/>	#General_About...	Request generi...	11 day...	20	
<input checked="" type="checkbox"/>	#General_Agent...	Request capab...	11 day...	30	

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Using built-in intents](#).

Using built-in intents

Content Catalogs provide an easy way to add common intents to your Watson Assistant dialog skill.

Intents you add from the catalog are meant to provide a starting point. Add to or edit the catalog intents to tailor them for your use case.

The latest content catalog, named **Covid-19**, is available in Brazilian Portuguese, English, French, and Spanish only. For more information about language support for the catalogs, see [Supported languages](#).

Adding a content catalog to your dialog skill

- Open your dialog skill, open the **Content Catalog** page.

Intents	Get started faster by adding existing intents from the content catalog. These intents are trained on questions that customers commonly ask.		
Entities			
Dialog			
Options			
Analytics			
Versions			
Content Catalog			
Banking	Category	Description	Intents
	Banking	Basic transactions for a banking use case.	13
	Bot Control	Functions that allow navigation within a conversation.	9
	Covid-19	Common questions about the Covid-19 virus.	10
	Customer Care	Understand and assist customers with information about themselves and your ...	18
	eCommerce	Payment, billing, and basic management tasks for orders.	14
	General	General conversation topics most users ask.	10
	Insurance	Issues related to insurance policies and claims.	12
	Mortgage	Common questions related to the mortgage industry	20
	Telco	Questions and issues related to a user's telephony service, device, and plan.	21
	Utilities	Help a user with utility emergencies and their utility service.	10

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 'Banking' content catalog selected. At the top, there is a back arrow, the title 'Banking', and an 'Add to skill +' button. Below this, there is a 'Description' section stating 'Basic transactions for a banking use case.' A table follows, listing intents with their descriptions and examples:

Intent (13)	Description	Examples
Banking_Activate_Card	Activate a 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...
Banking_Cancel_Card	Cancel a card.	Tell me the ways through which I can close my credit card Annual charges are high hence want to cancel my credit card Can I cancel a credit card I just applied for? 17 more examples...
Banking_Fee_Inquiry	Inquire about fees associated with a card.	Can you tell me about how much fee the bank charged for last month on my credit card? Where is the credit card fee information? Where can I see the cost for buying the credit card? 17 more examples...
Banking_Replace_Card	Replace a card.	What is the cost for credit card replacement? Where from can I issue another card? Can I get info on replacing my credit card? 17 more examples...

At the bottom of the table, there is a link: 'What steps can I take when I lose my credit card?'

Intents that are added from a content catalog are distinguishable from other intents by their names. Each intent name is prepended with the content catalog name.

3. Add a content catalog to your dialog skill by clicking the **Add to skill** button.

4. Go to the **Intents** page to see the intents that you added from the catalog listed.

Your skill trains itself on the new data.



Note: After you add a catalog to your skill, the intents become part of your training data. If IBM makes subsequent updates to a content catalog, the changes are not automatically applied to any intents you added from a catalog.

Editing content catalog intents

Like any other intent, after you add content catalog intents to your skill, you can make the following changes to them:

- Rename intents
- Delete intents
- Add, edit, or delete intent user examples
- Move an example to a different intent

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Algorithm version and training](#).

Algorithm version and training

The **Algorithm version** setting allows you to choose which Watson Assistant algorithm is used for training. Your assistant is trained when you update the content or settings, or through [automatic retraining](#).

There are three choices:

- **Beta:** Use this to preview and test what is coming. The capability in the beta version at any given time is likely to become a supported version later on. It's not recommended to use the beta version in a production deployment.
- **Latest:** The current supported version that's recommended for your live production assistant.
- **Previous:** The last supported before the latest version. Support for this version ends when the next version is released.

To choose an algorithm version for actions:

1. On the **Actions** page, click **Global settings** .
2. Click the **Algorithm Version** tab.
3. Choose a version, then click **Save**.

To choose an algorithm version for dialog:

1. On the **Dialog** page, open the **Options** section.
2. Click **Algorithm Version**.
3. Choose a version.

The latest and previous versions have date labels such as **Latest (01-Jun-2022)** or **Previous (01-Jan-2022)**. See the [release notes](#) for details about each algorithm version release.

Algorithm version choices are currently available for Arabic, Chinese (Simplified), Chinese (Traditional), Czech, Dutch, English, French, German, Japanese, Korean, Italian, Portuguese, and Spanish. The universal language model uses a default algorithm.

Automatic retraining

IBM Cloud

Watson Assistant was released as a service in July 2016. Since then, users have been creating and updating skills to meet their virtual assistant needs. Behind the scenes, Watson Assistant creates machine learning (ML) models to perform a variety of tasks on the user's behalf.

The primary ML models deal with action recognition, intent classification, and entity detection. For example, the model might detect what a user intends when saying **I want to open a checking account**, and what type of account the user is talking about.

These ML models rely on a sophisticated infrastructure. There are many intricate components that are responsible for analyzing what the user has said, breaking down the user's input, and processing it so the ML model can more easily predict what the user is asking.

Since Watson Assistant was first released, the product team has been making continuous updates to the algorithms that generate these sophisticated ML models. Older models have continued to function while running in the context of newer algorithms. Historically, the behavior of these existing ML model did not change unless the skill was updated, at which point the skill was retrained and a new model generated to replace the older one. This meant that many older models never benefited from improvements in our ML algorithms.

Watson Assistant uses continuous retraining. The Watson Assistant service continually monitors all ML models, and automatically retrains those models that have not been retrained in the previous 6 months. Watson Assistant retrains using the selected algorithm version. If the version you had selected is no longer supported, {{site.data.keassistant_classic_shortnshort}} retrains using the version labeled as **Previous**. This means that your assistant will automatically have the supported technologies applied. Assistants that have been modified during the previous 6 months will not be affected.



Note: In most cases, this retraining will be seamless from an end-user point of view. The same inputs will result in the same actions, intents, and entities being detected. In some cases, the retraining might cause changes in accuracy.

Instructions for Watson Assistant for IBM Cloud Pak for Data

IBM Cloud Pak for Data

When upgrading your instance of Watson Assistant for IBM Cloud Pak for Data, as long as your existing models have been trained using an algorithm version that is still supported, your models will not be retrained during or after the upgrade.

New algorithm versions will be included in major releases (for example, 4.0.0 or 5.0.0) or minor releases (for example, 4.5.0 or 4.6.0). A monthly release might include a new algorithm version if there is more than 6 months between major or minor releases. Each algorithm version supports 2 major/minor releases or a maximum time of 12 months, whichever is first. For more information, see the [IBM Cloud Pak for Data Software Support Lifecycle Addendum](#).

Each new release includes full support for the version listed as **Latest** in the most recent prior release. This version is then labeled as **Previous** after you upgrade. In addition, each new release will support running models that were trained on the version prior to that so that upgrading won't impact your runtime. For example, if you upgrade from IBM Cloud Pak for Data 4.6 to 4.7, and were using **Latest (01-Jun-2022)** that version becomes listed as **Previous (01 June 2022)** and remains your selected version.

Automatic retraining after upgrading

After your Watson Assistant for IBM Cloud Pak for Data upgrade is complete, {{site.data.keassistant_classic_shortnshort}} performs automatic retraining for any assistant models that were trained using a version that is no longer supported. In this case, {{site.dassistant_classic_shortrsationshort}}

automatically retrains your assistant to the **Latest** version. This automatic retraining is required to assure your ability to run your trained models in your next upgrade.

Best practices

It's recommended to use the **Latest** version in your production deployment of Watson Assistant for IBM Cloud Pak for Data. This is the default for newly-created assistants. During an upgrade, your settings don't automatically switch existing assistants to use the latest version. If prior to your upgrade you had selected **Latest**, your settings continue to use that version, now labeled as **Previous**. After you upgrade, it's recommended you choose **Latest** and run basic regression tests.

IBM performs robust testing on a variety of data sets to minimize impacts on existing assistants. But given the nature of machine learning models and the nuance and subtlety of natural language processing, you may find some discrepancies from version to version. If you find a major issue through your tests, you have the ability to switch your settings and use **Previous** to return to the prior behavior. In this event, we recommend you contact IBM and provide details of your test so that IBM can support you in the steps to resolve the problem.

It's also recommended that you try the **Beta** version in one of your test systems after you upgrade. This gives you early visibility to changes that are likely to be delivered in a future version, and will reduce the probability of negative impacts to your production systems. IBM values both positive and negative feedback from customers who use Beta. You will have the opportunity to shape how the algorithms function before the version is promoted to **Latest** in a future version. If you choose **Beta**, your assistant always trains on the most current beta version.

Building a conversational flow

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [How your dialog is processed](#).

How your dialog is processed

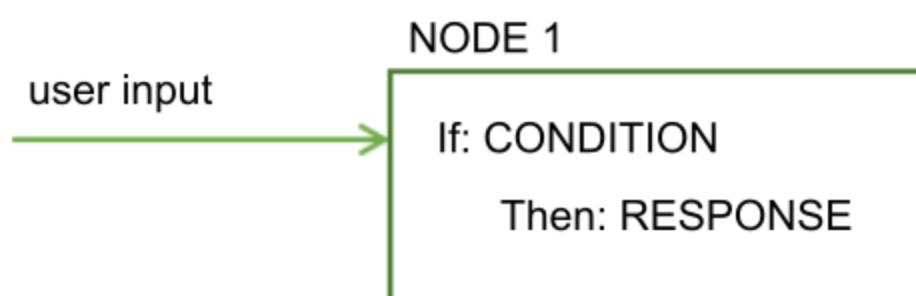
The dialog uses the intents that are identified in the user's input, plus context from the application, to interact with the user and ultimately provide a useful response.

The dialog matches intents (what users say they want to do) to responses (what the bot says back). The response might be the answer to a question such as **What are your store hours?** or the execution of a command, such as placing an order. The intent and entity might be enough information to identify the correct response, or the dialog might ask the user for more input that is needed to respond correctly. For example, if a user asks, **Where can I get some food?** you might want to clarify whether they want a restaurant or a grocery store, to dine in or take out, and so on. You can ask for more details in a text response and create one or more child nodes to process the new input.

The dialog is represented graphically in Watson Assistant 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

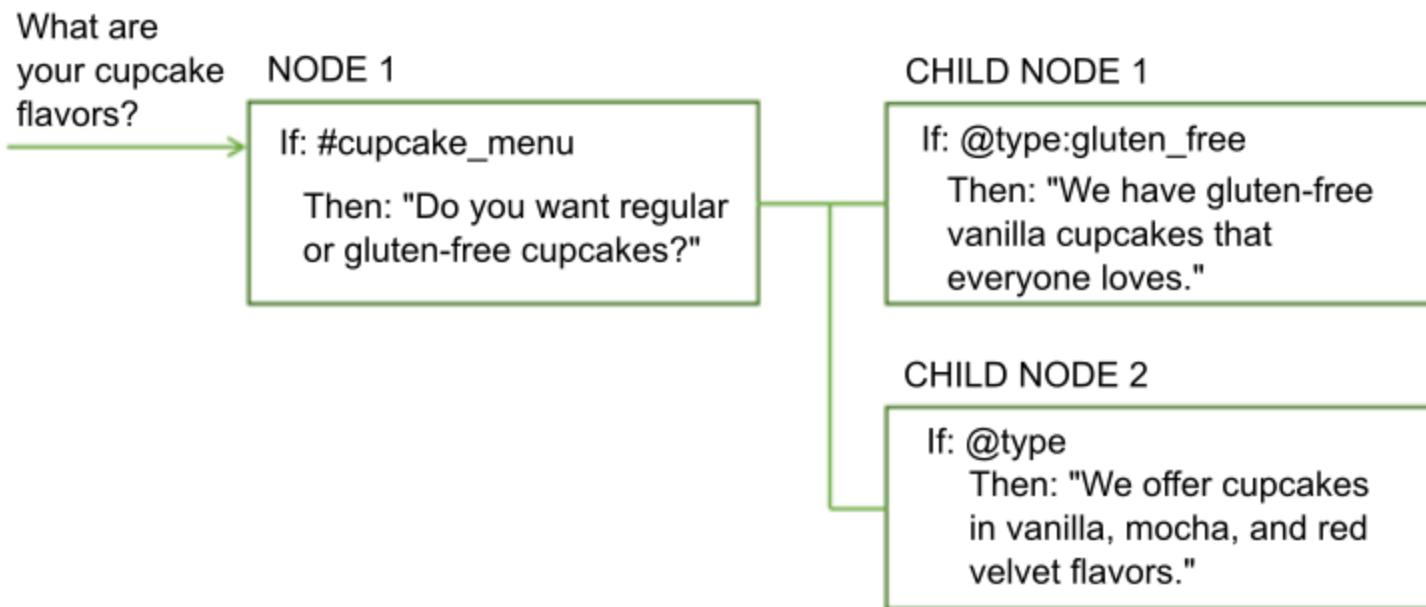
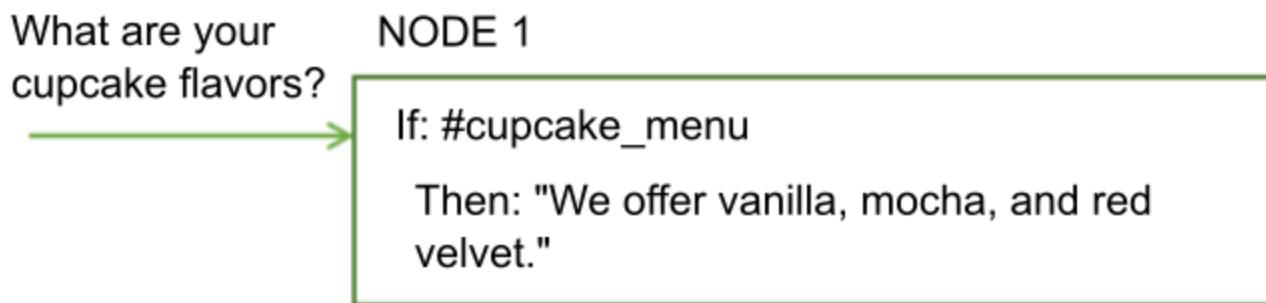
Each dialog node contains, at a minimum, a condition and a response.



- 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 your assistant 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.

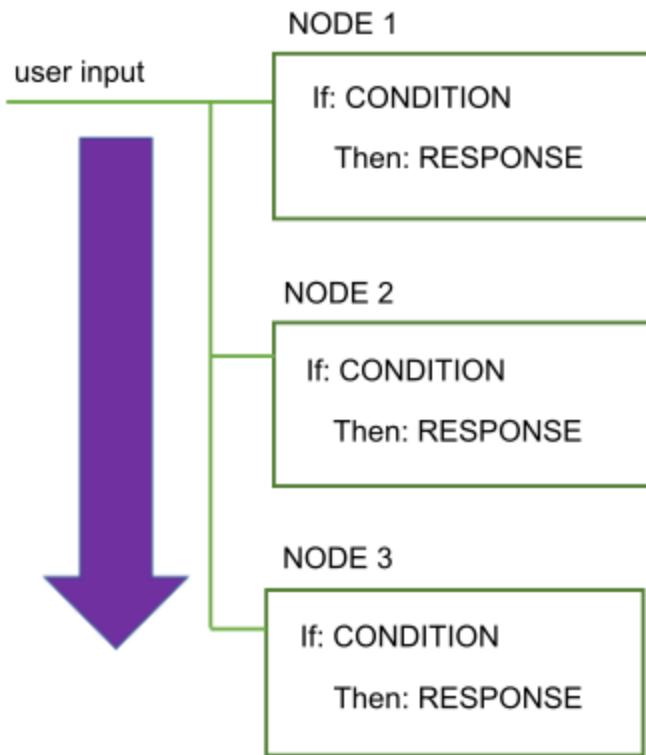
You can think of the node as having an if/then construction: if this condition is true, then return this response.

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

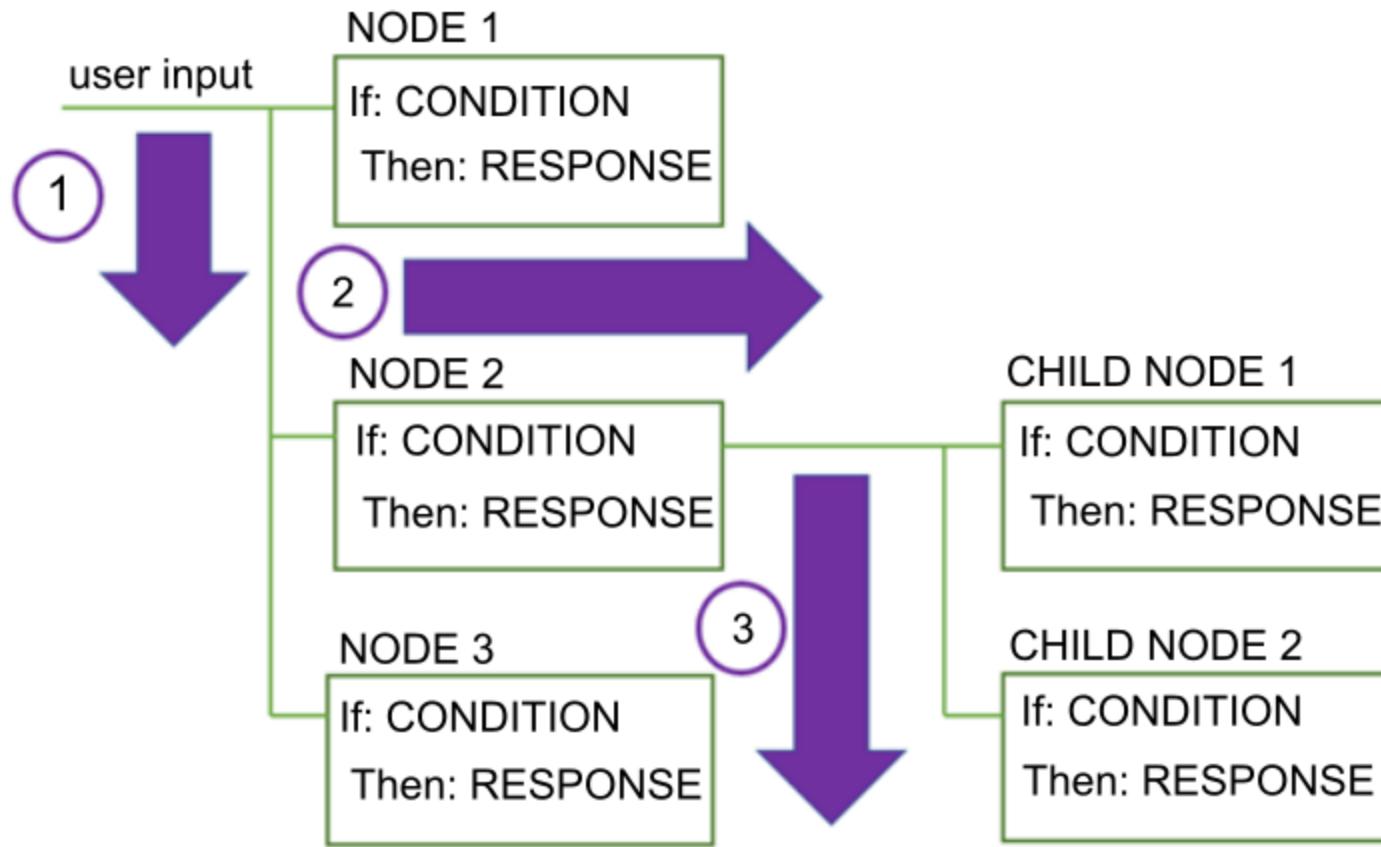


Dialog flow

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



Your assistant 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.

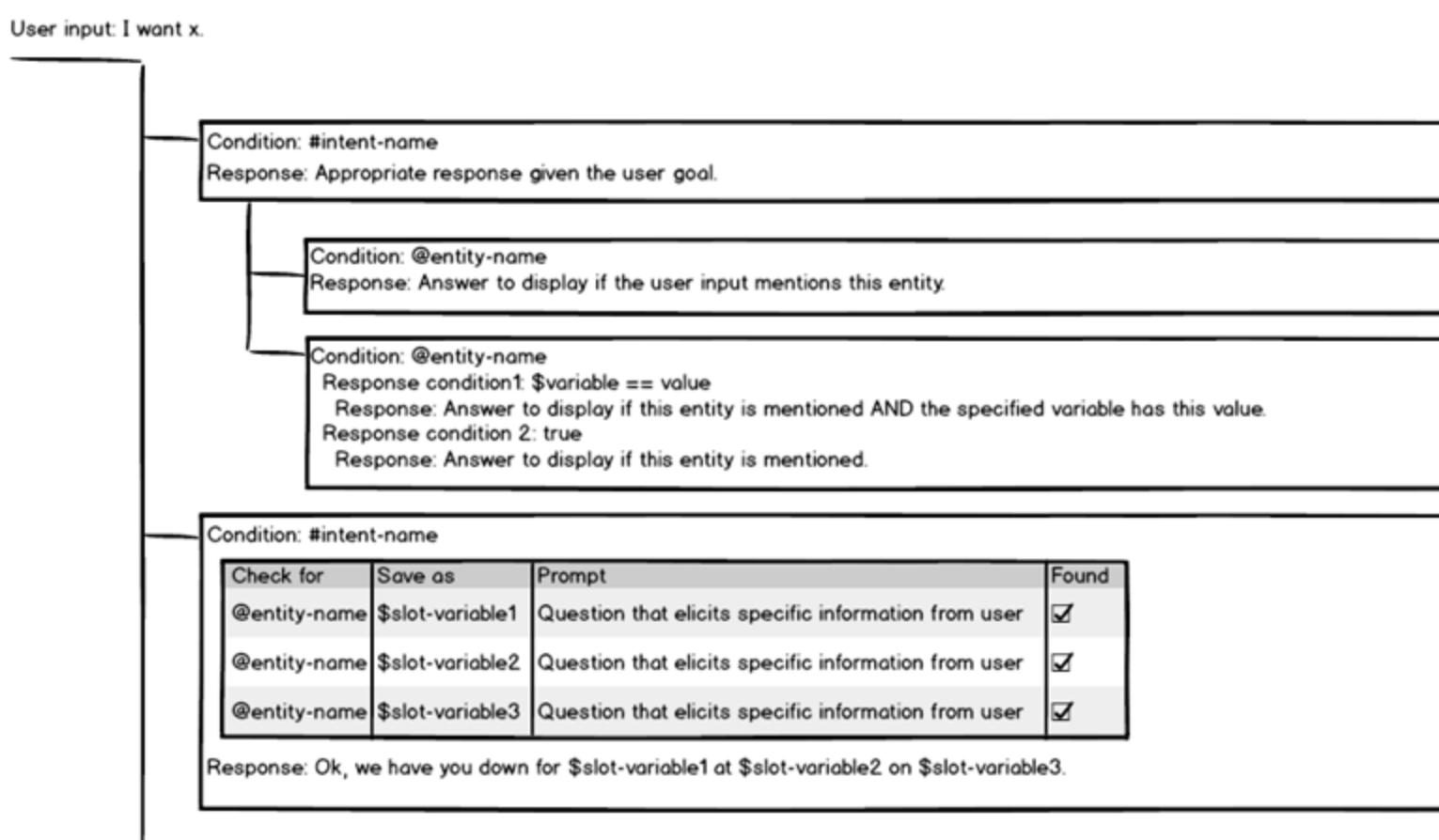
When your assistant reaches the end of a branch, or cannot find a condition that evaluates to true from the current set of child nodes it is evaluating, it jumps back out to the base of the tree. And once again, your assistant processes the root nodes from first to the last. If none of the conditions 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.

Sample dialog

This diagram shows a mockup of a dialog tree that is built with the graphical user interface dialog editor.



The dialog tree in this diagram contains two root dialog nodes. A typical dialog tree would likely have many more nodes, but this depiction provides a glimpse of what a subset of nodes might look like.

- The first root node conditions on an intent value. It has two child nodes that each condition on an entity value. The second child node defines two

responses. The first response is returned to the user if the value of the context variable matches the value specified in the condition. Otherwise, the second response is returned.

This standard type of node is useful to capture questions about a certain topic and then in the root response ask a follow-up question that is addressed by the child nodes. For example, it might recognize a user question about discounts and ask a follow-up question about whether the user is a member of any associations with which the company has special discount arrangements. And the child nodes provide different responses based on the user's answer to the question about association membership.

- The second root node is a node with slots. It also conditions on an intent value. It defines a set of slots, one for each piece of information that you want to collect from the user. Each slot asks a question to elicit the answer from the user. It looks for a specific entity value in the user's reply to the prompt, which it then saves in a slot context variable.

This type of node is useful for collecting details you might need to perform a transaction on the user's behalf. For example, if the user's intent is to book a flight, the slots can collect the origin and destination location information, travel dates, and so on.

Ready to get started?

For more information, see [Creating a dialog](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Creating a dialog](#).

Creating a dialog

The dialog defines what your assistant says in response to customers.

Creating a dialog

To create a dialog, complete the following steps:

1. From the Skills menu, click **Dialog**.

The following nodes are created for you automatically:

- **Welcome:** The first node. It contains a greeting that is displayed to your users when they first engage with your assistant. You can edit the greeting.

 **Note:** This node is not triggered in dialog flows that are initiated by users. For example, dialogs used in integrations with channels such as Facebook or Slack skip nodes with the `welcome` special condition.

- **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 your assistant to return each response that is defined in turn or return them in random order.

For more information about these built-in nodes, see [Starting and ending the dialog](#).

2. To add more nodes to the dialog tree, click **Add node**.

Your new node is added after the *Welcome* node and before the *Anything else* node.

3. Add a name to the node.

Use a short, customer-friendly description of what the node does as its name. For example, `Open an account`, `Get policy information`, or `Get a weather forecast`.

The name can be up to 512 characters in length.

 **Tip:** This node name is shown to customers or service desk personnel to express the purpose of this branch of the dialog, so take some time to add a name that is concise and descriptive.

4. In the **If assistant recognizes** field, enter a condition that, when met, triggers your assistant to process the node.

To start off, you typically want to add an intent as the condition. For example, if you add `#open_account` here, it means that you want the response that you will specify in this node to be returned to the user if the user input indicates that the user wants to open an account.

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

Character	Lists defined values for these artifact types
'#'	intents

`@`	entities
`@{entity-name}`:	{entity-name} values
`\$`	context-variables that you defined or referenced elsewhere in the dialog

Condition builder syntax

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

To define a node that triggers based on more than one condition, enter one condition, and then click the plus sign (+) icon next to it. If you want to apply an **OR** operator to the multiple conditions instead of **AND**, click the **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](#).

5. **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.

6. Enter a response.

- Add the text or multimedia elements that you want your assistant 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](#).

7. Specify what to do after the current node is processed. You can choose from the following options:

- **Wait for user input:** Your assistant pauses until new input is provided by the user.
- **Skip user input:** Your assistant jumps directly to the first child node. This option is only available if the current node has at least one child node.
- **Jump to:** Your assistant continues the dialog by processing the node you specify. You can choose whether your assistant 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.

8. **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 **external node name** field. For example, *Open an account*.



The *external node name* field is only displayed only to users of paid plans. See [Disambiguation](#) for more details.

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.

Conditions

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



Tip: 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.



Tip: 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.



Tip: 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 pattern entity groups in input](#) for more information.

- **Intent:** The simplest condition is a single intent. The node is used if, after your assistant'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 product that you can use to perform common dialog functions. See the [Special conditions](#) table in the next section for details.

Special conditions

Condition syntax	Description
<code>anything_else</code>	You can use this condition at the end of a dialog, to be processed when the user input does not match any other dialog nodes. The Anything else node is triggered by this condition. If you add a search skill to your assistant, a root node with this condition can be configured to trigger a search.
<code>conversation_start</code>	Like welcome , this condition is evaluated as true during the first dialog turn. Unlike welcome , it is true whether or not the initial request from the application contains user input.
<code>false</code>	This condition is always evaluated to false. You might use this at the start of a branch that is under development, to prevent it from being used, or as the condition for a node that provides a common function and is used only as the target of a Jump to action.
<code>irrelevant</code>	This condition will evaluate to true if the user's input is determined to be irrelevant by the Watson Assistant service.
<code>true</code>	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.
<code>welcome</code>	This condition is evaluated as true during the first dialog turn (when the conversation starts), only if the initial request from the application does not contain any user input. It is evaluated as false in all subsequent dialog turns. The Welcome node is triggered by this condition. Typically, a node with this condition is used to greet the user, for example, to display a message such as Welcome to our Pizza ordering app . This node is never processed during interactions that occur through channels such as Facebook or Slack.

Special conditions

Condition syntax details

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

- Shorthand notations to refer to intents, entities, and context variables. See [Accessing and evaluating objects](#).
- 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 your assistant 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. If you are using the JSON editor, you need to use two backslashes to escape (`\\\`). Escaping the character prevents your assistant from misinterpreting it as being one of the following artifact types:

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.

Special characters to escape in responses

The built-in integrations support the following Markdown syntax elements:

Format	Syntax	Example
Italics	We're talking about *practice*.	We're talking about <i>practice</i> .
Bold	There's **no** crying in baseball.	There's no crying in baseball.
Hypertext link	Contact us at [ibm.com](https://www.ibm.com).	Contact us at ibm.com .

Supported markdown syntax

If you don't code a link when you specify a phone number in a text response, it is not converted to a telephone link anywhere except in a web chat integration that is accessed from a mobile device.

The "Try it out" pane does not support Markdown syntax currently. For testing purposes, you can use the assistant *Preview* to see how the Markdown syntax is rendered.

The "Try it out" pane, assistant *Preview*, and *web chat* integration support HTML syntax. The *Slack* and *Facebook* integrations do not.

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:

Response variations

Hi.

How are you today?

Multiple line response

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



Note: If you are using a dialog skill that was created before support for rich response types was added to the product, 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.

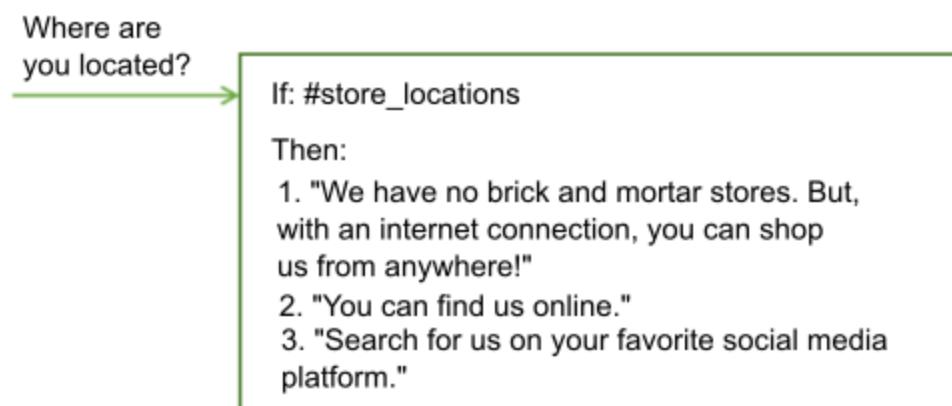
When the response is shown to the user, both response variations are displayed, one on each line, like this:

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 your assistant provides in response to questions about store locations differs from one interaction to the next:



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

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:

Response variations

Hello.

Hi.

Howdy!

Varying responses

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

- 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!

- 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:

- Connect to human agent:** The dialog calls a service that you designate, typically a service that manages human agent support ticket queues, to transfer the conversation to a person. You can optionally include a message that summarizes the user's issue to be provided to the human agent.
- Channel transfer:** The dialog requests that the conversation be transferred to a different channel (for example, from the Slack integration to the web chat integration).
- 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.
- Video:** Embeds a video player into the response. The source video must be hosted somewhere, either as a playable video on a supported video streaming service or as a video file with 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.
- Audio:** Embeds an audio clip into the response. The source audio 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.
- iframe:** Embeds content from an external website, such as a form or other interactive component, directly within the chat. The source content must

be publicly accessible using HTTP, and must be embeddable as an HTML `iframe` element.

- **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 your assistant. How options are rendered can differ depending on the number of options and where you deploy the dialog.
- **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 assistant is working on typing a response. Use this response type if you need to perform an action that might take some time.
- **Search skill:**  Searches an external data source for relevant information to return to the user. The data source that is searched is a Discovery service data collection that you configure when you add a search skill to the assistant that uses this dialog skill.



Note: This response type is available only to users of paid plans.

- **User-defined:** If you use the JSON editor to define the response, you can create your own user-defined response type. For more information, see [Defining responses using the JSON editor](#).

Different integrations have different capabilities for displaying rich responses. If you want to define different responses that are customized for different channels, you can do so by editing the response using the JSON editor. For more information, see [Targeting specific integrations](#).

To add a rich response, complete the following steps:

1. Click the dropdown menu in the **Assistant responds** field to choose a response type, and then provide any required information.

For more information, see the following sections:

- [Connect to human agent](#)
- [Channel transfer](#)
- [Image](#)
- [Option](#)
- [Pause](#)
- [Search skill](#) 



Note: This response type is only visible to users of paid plans.

- [Text](#)

2. To add another response type to the current response, click **Add response type**.

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. This means that if you define three conditioned responses for a dialog node, each conditioned response can have no more than 5 response types added to it.



Note: You cannot add more than one **Connect to human agent** or more than one **Search skill** response type to a single dialog node.



Note: Do not add more than one option response type to a single dialog node because both lists are displayed at once, but the customer can choose an option from only one of them.

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 your assistant to process them.

Adding a *Connect to human agent* response type

If your client application is able to transfer a conversation to a person, such as a customer support agent, then you can add a *Connect to human agent* response type to initiate the transfer. Some of the built-in integrations, such as web chat and Intercom, support making transfers to service desk agents. If you are using a custom application, you must program the application to recognize when this response type is triggered.



Tip: If you want to take advantage of the *containment* metric to track your assistant's success rate, add this response type to your dialog or use an alternate method to identify when customers are directed to outside support. For more information, see [Measuring containment](#).

To add a *Connect to human agent* response type, complete the following steps:

1. From the dialog node where you want to add the response type, click the dropdown menu in the **Assistant responds** field, and then choose **Connect to human agent**.
2. **Optional.** Add a message to share with the human agent to whom the conversation is transferred in the **Message to human agent** field.
3. Add a message to show to the customer to explain that they are being transferred.

You can add a message to show when agents are available and a message to show when agents are unavailable. Each message can be up to 320 characters in length.

Web chat built-in service desk integrations only: The text you add to the *Response when agents are online* and *Response when no agents are online* fields is used for transfers in web chat version 3 and later. If you don't add your own messages, the hint text (the grayed out text that is displayed as the example messages) is used.

 **Tip:** If you use this response type in multiple nodes and want to use the same custom text each time, but don't want to have to edit each node individually, you can change the default text that is used by the web chat. To change the default messages, edit the [language source file](#). Look for the `default_agent_availableMessage` and `default_agent_unavailableMessage` values. For more information about how to change web chat text, see [Languages](#).

4. **Optional:** If the channel where you deploy the assistant is integrated with a service desk, you can add initial routing information to pass with the transfer request.
 - o Pick the integration type from the **Service desk routing** field.
 - o Add routing information that is meaningful to the service desk you are using.

Service desk type	Routing information	Description
Salesforce	Button ID	Specify a valid button ID from your Salesforce deployment. For more information, see Adding routing logic for Salesforce transfers .
Zendesk	Department	Specify a valid department name from your Zendesk account. For more information, see Adding routing logic for Zendesk transfers .

Service desk routing options

The dialog transfer does not occur when you test dialog nodes with this response type in the "Try it out" pane of the dialog skill. You must access a node that uses this response type from the *Preview* button for the assistant to see how your users will experience it.

Adding a *Channel transfer* response type

If your assistant uses multiple integrations to support different channels for interaction with users, there might be some situations when a customer begins a conversation in one channel but then needs to transfer to a different channel.

The most common such situation is transferring a conversation to the web chat integration, in order to take advantage of web chat features such as service desk integration.



Note: Currently, the web chat is the only supported target for a channel transfer.

Only the following integrations can initiate a channel transfer: - Slack - Facebook Messenger - WhatsApp

Other integrations ignore the *Channel transfer* response type.

To add a *Channel transfer* response type, complete the following steps:

1. From the dialog node where you want to add the response type, click the dropdown menu in the **Assistant responds** field, and then choose **Channel transfer**.
2. **Optional.** In the **Message before link to web chat** field, edit the introductory message to display to the user (in the originating channel) before the link that initiates the transfer. By default, this message is `OK, click this link for additional help. Chat will continue on a new web page.`
3. In the **URL to web chat** field, type the URL for your website where the web chat widget is embedded.

In the integration that processes the *Channel transfer* response, the introductory message is displayed, followed by a link to the URL you specify. The user must then click the link to initiate the transfer.

When a conversation is transferred from one channel to another, the session history and context are preserved, so the destination channel can continue the

conversation from where it left off. Note that the message output that contains the *Channel transfer* response is processed first by the channel that initiates the transfer, and then by the target channel. If the output contains multiple responses (perhaps using different response types), these will be processed by both channels (before and after the transfer). If you want to target individual responses to specific channels, you can do so by editing the response using the JSON editor. For more information, see [Targeting specific integrations](#).

Adding an *Image* response type

Sometimes a picture is worth a thousand words. Include images in your response to do things like illustrate a concept, show off merchandise for sale, or maybe to show a map of your store location.

To add an *Image* response type, complete the following steps:

1. Choose **Image**.
2. 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 an **https:** 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.

To access an image that is stored in IBM Cloud® Object Storage, enable public access to the individual image storage object, and then reference it by specifying the image source with syntax like this: **https://s3.eu.cloud-object-storage.appdomain.cloud/your-bucket-name/image-name.png**.



Note: Some integration channels ignore titles or descriptions.

Adding a *Video* response type

Include videos in your response to share how-to demonstrations, promotional clips, and so forth. In the web chat, a video response renders as an embedded video player.

To add a *Video* response type, complete the following steps:

1. Choose **Video**.
2. Add the full URL to the hosted video into the **Video source** field:
 - o To link directly to a video file, specify the URL to a file in any standard format such as MPEG or AVI. In the web chat, the linked video will render as an embeddable video player.



Note: HLS (**.m3u8**) and DASH (MPD) streaming videos are not supported.

- o To link to a video hosted on a supported video hosting service, specify the URL to the video. In the web chat, the linked video will render using the embeddable player for the hosting service.



Note: Specify the URL you would use to view the video in your browser (for example, **https://www.youtube.com/watch?v=52bpMKVigGU**). You do not need to convert the URL to an embeddable form; the web chat will do this automatically.

You can embed videos hosted on the following services:

- [YouTube](#)
- [Facebook](#)
- [Vimeo](#)
- [Twitch](#)
- [Streamable](#)
- [Wistia](#)
- [Vidyard](#)

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



Note: Some integration channels ignore titles or descriptions.

If you want to scale the video to a specific display size, specify a number in the **Base height** field.

3. The **Video** response type is supported in the web chat, Facebook, WhatsApp, Slack, and SMS integrations.

Adding an *Audio* response type

Include audio clips in your response to share spoken-word or other audible content. In the web chat, a video response renders as an embedded video player. In the phone integration, an audio response plays over the phone.

To add an **Audio** response type, complete the following steps:

1. Choose **Audio**.
2. Add the full URL to the hosted audio clip into the **Audio source** field:
 - To link directly to an audio file, specify the URL to a file in any standard format such as MP3 or WAV. In the web chat, the linked audio clip will render as an embedded audio player.
 - To link to an audio clip on a supported audio hosting service, specify the URL to the audio clip. In the web chat, the linked audio clip will render using the embeddable player for the hosting service.



Note: Specify the URL you would use to access the audio file in your browser (for example, <https://soundcloud.com/ibmresearch/fallen-star-amped>). You do not need to convert the URL to an embeddable form; the web chat will do this automatically.

You can embed audio hosted on the following services:

- [SoundCloud](#)
- [Mixcloud](#)

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



Note: Some integration channels ignore titles or descriptions.

If you want the audio clip to loop indefinitely, select **On** in the **Loop** field. For example, you might want to use this option to play music while a user waits on the phone. (By default, the audio plays only once and then stops.)



Note: The **Loop** option is currently supported only by the phone integration. This option has no effect if you are using the web chat integration or any other channel.

3. The **Audio** response type is supported in the web chat, Facebook, WhatsApp, Slack, SMS, and phone integrations.

Adding an *iframe* response type

Include iframes in your response to embed content from another website directly inside the chat window as an HTML **iframe** element. An iframe response is useful if you want to enable customers to perform some interaction with an external service without leaving the chat. For example, you might use an *iframe* response to display the following within the web chat:

- An interactive map on [Google Maps](#)
- A survey using [SurveyMonkey](#)
- A form for making reservations through [OpenTable](#)
- A scheduling form using [Calendly](#)

In the web chat, an *iframe* response renders as a preview card that describes the embedded content. Customers can click this card to display the frame and interact with the content.

To add an *iframe* response type, complete the following steps:

1. Choose **iframe**.
2. Add the full URL to the external content in the **iframe source** field.

The URL must specify content that is embeddable in an HTML **iframe** element. Different sites have varying restrictions for embedding content, and different processes for generating embeddable URLs. An embeddable URL is one that can be specified as the value of the **src** attribute of the **iframe** element.

For example, to embed an interactive map using Google Maps, you can use the Google Maps Embed API. (For more information, see [The Maps Embed API overview](#).) Other sites have different processes for creating embeddable content.

For technical details about using **Content-Security-Policy: frame-src** to allow embedding of your website content, see [CSP: frame-src](#).

3. Optionally add a descriptive title in the **Title** field.

In the web chat, this title will be displayed in the preview card before the customer clicks to render the external content. (If you do not specify a title, the web chat will attempt to retrieve metadata from the specified URL and display the title of the content as specified at the source.)

4. The **iframe** response type is supported in the web chat and Facebook integrations.

 **Note:** In the "Try it out" pane, the iframe is rendered immediately, without the preview card. To see how the web chat renders your response, use the assistant [Preview](#) page.

Technical details: <iframe> sandboxing

Content loaded in an iframe by the web chat is *sandboxed*, meaning that it has restricted permissions that reduce security vulnerabilities. The web chat uses the `sandbox` attribute of the `iframe` element to grant only the following permissions:

Permission	Description
<code>allow-downloads</code>	Allows downloading files from the network, if the download is initiated by the user.
<code>allow-forms</code>	Allows submitting forms.
<code>allow-scripts</code>	Allows running scripts, but <i>not</i> opening pop-up windows.
<code>allow-same-origin</code>	Allows the content to access its own data storage (such as cookies), and allows only very limited access to JavaScript APIs.

 **Note:** A script running inside a sandboxed iframe cannot make changes to any content outside the iframe, *if* the outer page and the iframe have different origins. Be careful if you use an `iframe` response to embed content that has the same origin as the the page where your web chat widget is hosted; in this situation the embedded content can defeat the sandboxing and gain access to content outside the frame. For more information about this potential vulnerability, see the `sandbox` attribute [documentation](#).

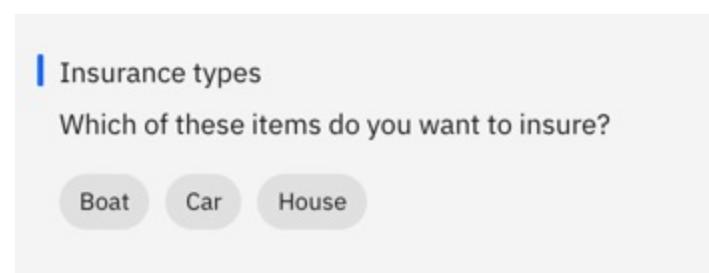
Adding an *Option* response type

Add an option response type when you want to give the customer a set of options to choose from. For example, you can construct a response like this:

List title	List description	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

Response options

Most integrations display the options as buttons if there are only a few items (4 or fewer, for example).



Otherwise, the options are displayed as a list.

To add an *Option* response type, complete the following steps:

- From the dialog node where you want to add the response type, click the dropdown menu in the **Assistant responds** field, and then choose **Option**.
- Click **Add option**.
- In the **List label** field, enter the option to display in the list.
The label must be less than 2,048 characters in length.

- In the corresponding **Value** field, enter the user input to pass to your assistant when this option is selected.

The value must be less than 2,048 characters in length.



Important: For Slack integrations where the options are displayed as a list, each value must be 75 characters or less in length.

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.

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

You can add up to 20 options.

6. Add a list introduction in the **Title** field. The title can ask the user to pick from the list of options.



Note: Some integration channels do not display the title.

7. Optionally, add additional information in the **Description** field. If specified, the description is displayed after the title and before the option list.



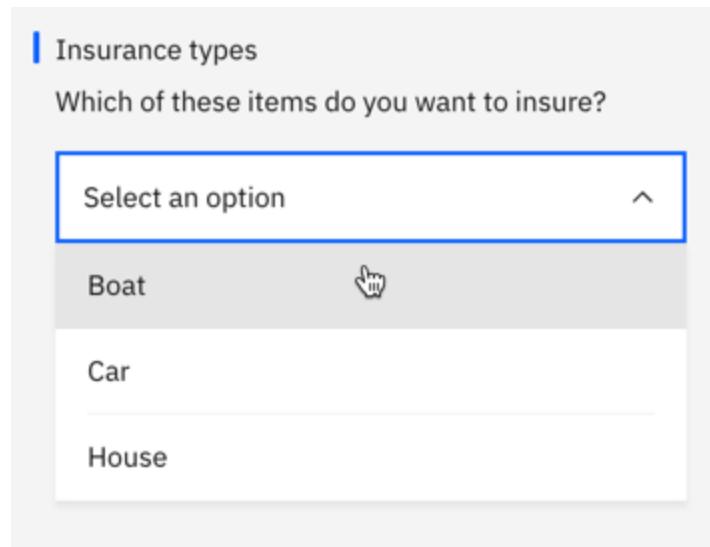
Note: Some integration channels do not display the description.

8. **Optional:** If you want to indicate a preference for how the options are displayed, as buttons or in a list, you can add a **preference** property for the response.

To do so, open the JSON editor for the response, and then add a **preference** name and value pair before the **response_type** name and value pair. You can set the preference to **dropdown** or **button**.

```
{
  "output": {
    "generic": [
      {
        "title": "Insurance types",
        "options": [
          {
            "label": "Boat",
            "value": {
              "input": {
                "text": "I want to buy boat insurance."
              }
            }
          },
          {
            "label": "Car",
            "value": {
              "input": {
                "text": "I want to buy car insurance."
              }
            }
          },
          {
            "label": "House",
            "value": {
              "input": {
                "text": "I want to buy house insurance."
              }
            }
          }
        ],
        "preference": "dropdown", //add this name and value pair
        "description": "Which of these items do you want to insure?",
        "response_type": "option"
      }
    ]
  }
}
```

When you define an options list with only 3 items, the options are typically displayed as buttons. When you add a preference property that indicates **dropdown** as the preference, for example, you can see in the "Try it out" pane that the list is displayed as a drop-down list instead.



Some integration types, such as the web chat, reflect your preference. Other integration types, such as Slack, do not reflect your preference when they render the options.

Important: Do not add more than one option response type to a single dialog node because both lists are displayed at once, but the customer can choose an option from only one of them.

If you need to be able to populate the list of options with different values based on some other factors, you can design a dynamic options list. For more information, see the [How to Dynamically Add Response Options to Dialog Nodes](#) blog post.

Adding a *Pause* response type

Add a pause response type to give the assistant time to respond. For example, you might add a pause response type to a node that calls a webhook. The pause indicates that the assistant is working on an answer, which gives the assistant time to make the webhook call and get a response. Then, you can jump to a child node to show the result.

To add a *Pause* response type, complete the following steps:

1. From the dialog node where you want to add the response type, click the dropdown menu in the **Assistant responds** field, and then choose **Pause**.
2. 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**.

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

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. For more information, see [Testing your assistant from a web page](#).

Adding a *Search skill* response type Plus

If you have existing customer-facing material, such as an FAQ, a product catalog, or sales material that can answer questions that customers often ask, put that information to use. You can trigger a search of the existing material in real time to get the latest and most up-to-date answer for your customers.

To use the search skill response type, you must create a search skill and add it to the same assistant that uses this dialog skill. For more information, see [Creating a search skill](#).

To add a *Search skill* response type, complete the following steps:

1. From the dialog node where you want to add the response type, click the dropdown menu in the **Assistant responds** field, and then choose **Search skill**.

Indicates that you want to search an external data source for a relevant response.

2. To edit the search query to pass to the Discovery service, click **Customize**, and then fill in the following fields:

- o **Query:** Optional. You can specify a specific query in natural language to pass to Discovery. If you do not add a query, then the customer's exact input text is passed as the query.

For example, you can specify `What cities do you fly to?`. This query value is passed to Discovery as a search query. Discovery uses natural language understanding to understand the query and to find an answer or relevant information about the subject in the data collection that is configured for the search skill.

You can include specific information provided by the user by referencing entities that were detected in the user's input as part of the query. For example, `Tell me about @product`. Or you can reference a context variable, such as `Do you have flights to $destination?`. Just be sure to design your dialog such that the search is not triggered unless any entities or context variables that you reference in the query have been set to valid values.

This field is equivalent to the Discovery `natural_language_query` parameter. For more information, see [Query parameters](#).

- **Filter:** Optional. Specify a text string that defines information that must be present in any of the search results that are returned.
 - To indicate that you want to return only documents with positive sentiment detected, for example, specify `enriched_text.sentiment.document.label:positive`.
 - To filter results to includes only documents that the ingestion process identified as containing the entity `Boston, MA`, specify `enriched_text.entities.text:"Boston, MA"`.
 - To filter results to includes only documents that the ingestion process identified as containing a product name provided by the customer, you can specify `enriched_text.entities.text:@product`.
 - To filter results to includes only documents that the ingestion process identified as containing a city name that you saved in a context variable named `$destination`, you can specify `enriched_text.entities.text:$destination`.

This field is equivalent to the Discovery `filter` parameter. For more information, see [Query parameters](#).

If you add both a query and a filter value, the filter parameter is applied first to filter the data collection documents and cache the results. The query parameter then ranks the cached results.

3. **Optional:** Change the query type that is used for the search.

The search skill sends a natural language query to Discovery automatically. If you want to use the Discovery query language instead, you can specify it. To do so, open the JSON editor for the node response.

Edit the JSON code snippet to replace `natural_language` with `discovery_query_language`. For example:

```
{  
  "output": {  
    "generic": [  
      {  
        "query": "",  
        "filter": "enriched_text.sentiment.document.label:positive",  
        "query_type": "discovery_query_language",  
        "response_type": "search_skill"  
      }  
    ]  
  }  
}
```

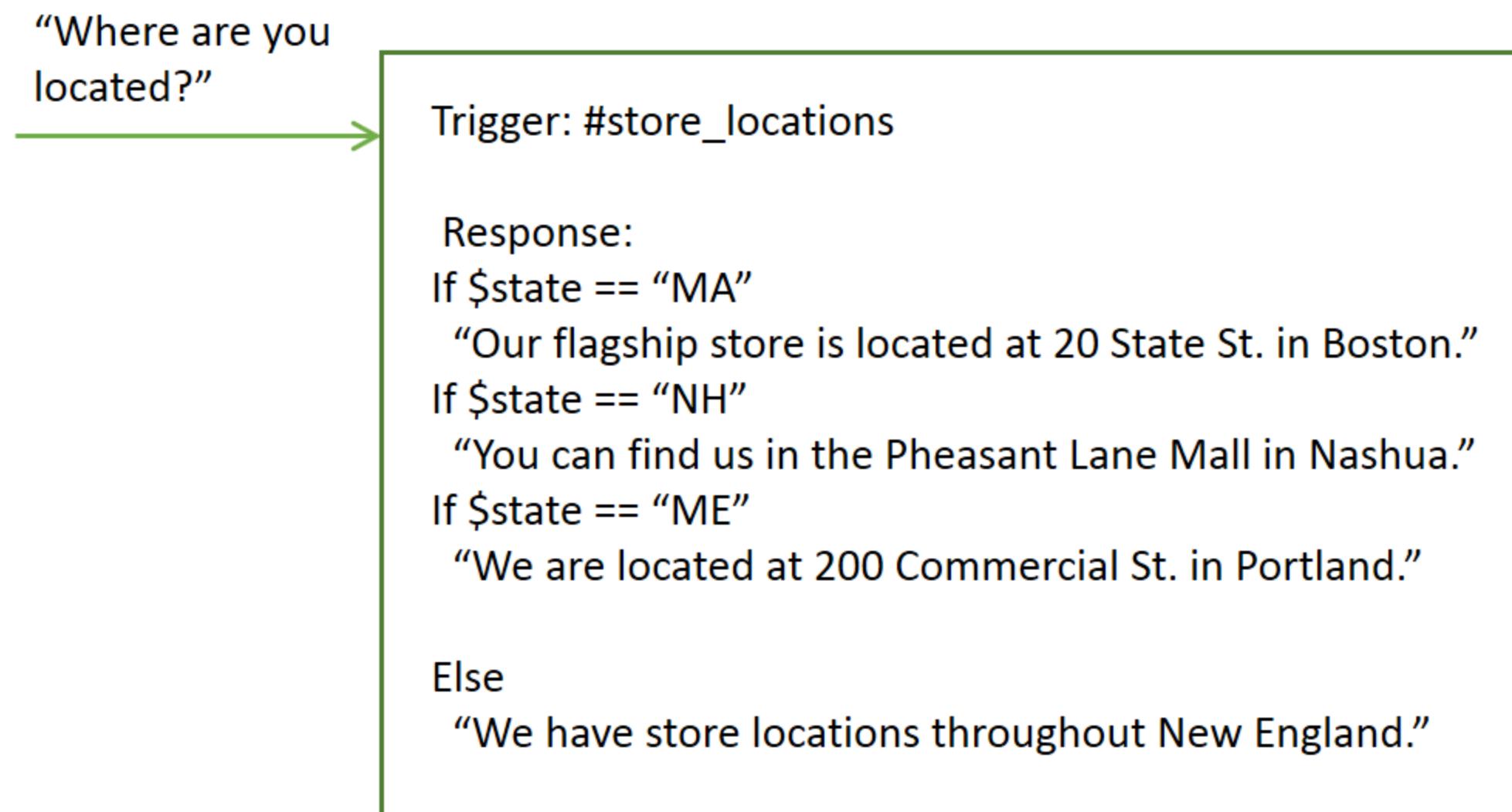
Test this response type from the assistant *Preview*. You cannot test it from the dialog skill's "Try it out" pane. For more information about testing dialog and search skills together, see [Testing your assistant from a web page](#).

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, your assistant 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 set the **Multiple conditioned responses** switch to **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 your assistant 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 your assistant 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 your assistant to do one of the following things:

- **Wait for user input**: Your assistant 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, your assistant 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 your assistant 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. Your assistant 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.

Next steps

- Be sure to test your dialog as you build it. For more information, see [Testing the dialog](#).
- For more information about ways to address common use cases, see [Dialog building tips](#).
- For more information about the expression language that you can use to improve your dialog, such as methods that reformat dates or text, see [Expression language methods](#).

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

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Starting and ending the dialog](#).

Starting and ending the dialog

Learn more about how to use the nodes that are added to your dialog automatically to start and end the conversation.

When you add a dialog to your dialog skill, the following dialog nodes are added to it automatically:

- **Welcome:** Defines how the assistant greets the user and starts the conversation.
- **Anything else:** What the assistant says when a customer's request cannot be satisfied by any of the defined intents.

Starting the conversation

The Welcome node is defined using the `welcome` special condition, which is triggered when the assistant, rather than the user, starts the conversation. This happens when the integration or client application starts the session with an empty message and then waits for the assistant to greet the user, as in the following situations:

- *Preview* for the assistant
- "Try it out" pane
- *Web chat* integration with home screen disabled

However, the Welcome node is skipped in situations when the user initiates the conversation by sending a message, such as with the *Slack* and *Facebook* integrations. It is also skipped when using the *Web chat* integration with the home screen enabled, because in this situation the home screen provides the greeting. (Note that the home screen is enabled by default.)

Unlike the `welcome` special condition, the `conversation_start` special condition is always triggered at the start of a conversation. You can use a

combination of nodes with these two special conditions (`welcome` and `conversation_start`) to manage the start of your dialog in a consistent way.

For more information, see [Special conditions](#).

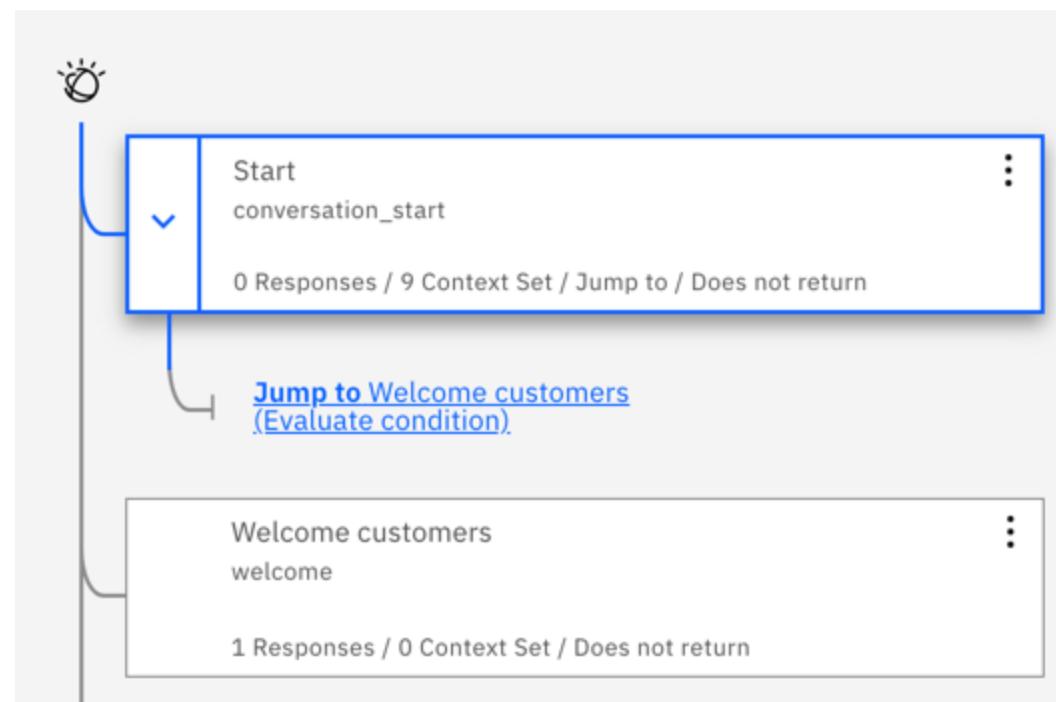
Setting initial context

If you need to set initial context variables at the beginning of each conversation, make sure you do so in a way that works with all of the integrations you plan to use. Do not use the Welcome node to set initial context variables unless you are certain your dialog will only be accessed in situations where the `welcome` special condition is triggered.

A safer and more consistent approach is to always set any initial context in a node defined using the `conversation_start` special condition, which is always triggered. You can use this node in addition to a Welcome node that displays a greeting.

To manage the start of any conversation regardless of integration, follow these steps:

1. Add a dialog node above the Welcome node that is automatically added to the top of the dialog tree when you create the dialog.
2. Set the node condition for this newly added node to `conversation_start`. This node will be reliably triggered at the beginning of any conversation.
3. In the `conversation_start` node, define any default values for context variables, and call any webhooks you need to call at the beginning of every conversation.
4. Do not define a text response for this node. Instead, configure this node to jump to the `Welcome` node directly below it in the dialog tree (or whichever other node you want to process first), and choose **If assistant recognizes (condition)**.



This design results in a dialog that works like this:

- Whatever the integration type, the `conversation_start` node is processed, which means any context variables that you define in it are initialized.
- In integrations where the assistant starts the dialog flow, the `Welcome` node is triggered and its text response is displayed.
- In integrations where the user starts the dialog flow, the user's first input is evaluated and then processed by the node that can provide the best response.

Ending the conversation gracefully

The *Anything else* node is designed to recognize the `anything_else` special condition, which understands when user input does not match any of the intents that are used as conditions in a dialog's nodes.

Don't delete the *Anything else* node.

You might not recognize its value at first, but it serves some important functions. If you did delete it, don't panic. You can add it back. Just add a dialog node to the end of your dialog tree, and add the `anything_else` special condition to its *If assistant recognizes* field.

The *Anything else* node provides the following benefits:

- It prevents your assistant from ever going silent and failing to respond at all to your customers. The *Anything else* node is what enables your assistant to (if nothing else) say, `I'm sorry, I didn't understand.` or `I can't help you with that.`
- The skill's analytics feature uses this node to learn about the topics that your dialog can't address. The *coverage metric* looks for occurrences of nodes with the `anything_else` condition being processed in the user conversation logs. It uses this information to determine the frequency with which your dialog is able to match user requests to intents that can address them. The node is registered by the metric if it conditions on `anything_else` alone or when it's used in combination with another condition, such as `anything_else && #positive_feedback`.

For more information about the coverage metric, see [Graphs and statistics](#).

- If you want your assistant to redirect queries to the search skill when the dialog is unable to address them, this node recognizes when it's time to initiate the search. It's when a customer's message reaches the `anything_else` node that the message is sent to the search skill to find a

relevant answer in your configured data collections. For more information about searching for an answer, see [Search triggers](#).



Note: Messages that trigger search in this way are still registered by the coverage metric as messages that are *not* covered.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Connecting customers with support](#).

Connecting customers with support

No matter where you deploy your assistant, give your customers a way to get additional support when they need it.

Build your dialog to recognize when customers need help that cannot be provided by the assistant. Add logic that can connect your customers to whatever type of professional support you offer. Your solutions might include:

- A toll-free phone number to a call center that is manned by human agents
- An online support ticket form that customers fill out and submit
- An integration with Intercom
- A service desk solution that is configured to work with your web chat integration or a custom client application

Design your dialog to recognize customer requests for help and address them. Add an intent that understands the customer request, and then add a dialog branch that handles the request.

For example, you might add an intent and use it in a dialog node like the intents that are shown in the table below.

Intent name	Intent user	Intent user example 2	Response from dialog node that conditions on intent
#call_support	How do I reach support?	What's your toll-free number?	Call 1-800-555-0123 to reach a call center agent at any time.
#support_ticket	How do I get help?	Who can help me with an issue I'm having?	Go to [Support Center](https://example.com/support) and open a support ticket.

Alternative support request intent examples

If you deploy your assistant with an integration that has built-in service desk support, you can use the special *Connect to human agent* response type in your dialog response to initiate a transfer.

Adding chat transfer support

Design your dialog so that it can transfer customers to human agents. Consider adding support for initiating a transfer in the following scenarios:

- Any time a user asks to speak to a person.

Create an intent that can recognize when a customer asks to speak to someone. After defining the intent, you can add a root-level dialog node that conditions on the intent. As the dialog node response, add a *Connect to human agent* response type. At run time, if the user asks to speak to someone, this node is triggered and a transfer is initiated on the user's behalf.

- When the conversation broaches a topic that is sensitive in nature, you can start a transfer.

For example, an insurance company might want questions about bereavement benefits always to be handled by a person. Or, if a customer wants to close an account, you might want to transfer the conversation to a representative who is authorized to offer incentives to keep the customer's business.

- When the assistant repeatedly fails to understand a customer's request. Instead of asking the customer to retry or rephrase the question over and over, your assistant can offer to transfer the customer to a human agent.

To design a dialog that can transfer the conversation, complete the following steps:

1. Add an intent to your skill that can recognize a user's request to speak to a human.

You can create your own intent or add the prebuilt intent named **#General_Connect_to_Agent** that is provided with the **General** content catalog.

2. Add a root node to your dialog that conditions on the intent you created in the previous step. Choose **Connect to human agent** as the response type.

For more information, see [Adding a Connect to human agent response type](#).

3. Add meaningful names to the dialog nodes in your dialog.

When a conversation is transferred to an agent, the name of the most-recently processed dialog node is sent with it. To ensure that a useful summary is provided to service desk agents when a conversation is transferred to them, add short dialog node names that reflect the node's purpose. For

example, *Find a store*.

Some older skills specify the purpose of the node in the **external node name** field instead.

And finally:

Wait for user input ▾

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

Enter external node name

 **Status:** this node will not be used for features that require displaying an external node name unless an external node name is provided.



Note: Every dialog branch can be processed by the assistant while it chats with a customer, including branches with root nodes in folders.

4. If a child node in any dialog branch conditions on a request or question that you do not want the assistant to handle, add a **Connect to human agent** response type to the child node.

At run time, if the conversation reaches this child node, the dialog is passed to a human agent at that point.

Your dialog is now ready to support transfers from your assistant to a service desk agent.

For more information about different service desk solutions, see the following resources:

- [Adding service desk support to the web chat](#)
- [Integrating with Intercom](#)

Measuring containment

From the Analytics page, you can measure conversation containment. Containment is the rate at which your assistant is able to satisfy a customer's request without human intervention per conversation.

To measure containment accurately, the metric must be able to identify when a human intervention occurs. The metric primarily uses the **Connect to human agent** response type as an indicator. If a user conversation log includes a call to a **Connect to human agent** response type, then the conversation is considered to be *not contained*.

However, not all human interventions are transacted by using a **Connect to human agent** response type. If you use an alternate method to deliver additional support, you must take additional steps to register the fact that the customer's need was not fulfilled by the assistant. For example, you might direct customers to your call center phone number or to an online support ticket form URL. If so, you need to flag these types of redirects so the containment metric is able to pick them up.

To enable the containment metric, complete the following steps:

1. For any dialog nodes that transfer the chat to a service desk agent, add a **Connect to human agent** response type.

For more information, see [Adding a Connect to human agent response type](#).

2. For any dialog nodes that redirect customers to alternate forms of customer support, add a context variable named `context_connect_to_agent` and set it to `true`.

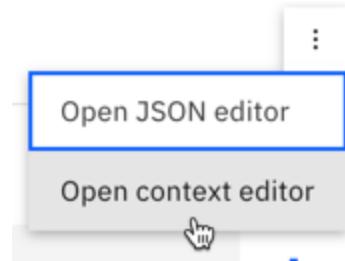
The process that calculates the containment metric looks for instances of this context variable, and registers any occurrences that it finds as interventions.

- From the dialog node, go to the *Assistant responds* section and click the menu icon .



Tip: If you're using multiple conditioned responses, you must click the *Customize response* icon  to see the menu that is associated with the response.

- Click **Open context editor**.



- Add the following variable name and value to define the context variable:

Variable	Value
`context_connect_to_agent`	`true`

Special containment context variable

- Click **X** to close the dialog node. Your change is saved automatically.

Tip: If you're using multiple conditioned responses, you must click **Save** first.

To track containment for your assistant, go to the [Analytics](#) page. For more information, see [Metrics overview](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Conversation building tips](#).

Conversation building tips

Get tips about ways to address common tasks.

Adding nodes

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

Today, you know what the node does, but months from now you might not remember. 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 your assistant 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 immediately.

- 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 skill frequently. See [Downloading a skill](#).

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 ?>.
```



Note: If autocorrection is on, and you want to return the user's original input before it was corrected, you can use `<? input.original_text ?>`. But, be sure to use a response condition that checks whether the `original_text` field exists first.

- **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 your assistant 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 correct 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.

- **Hiding data from the log:** You can prevent information from being stored in Watson logs by storing it in a context variable and nesting the context variable within the `$private` section of the message context. For example: `$private.my_info`. Storing data in the private object hides it from the logs only. The information is still stored in the underlying JSON object. Do not allow this information to be exposed to the client application.
- **Checking for personal information:** If you want to check for and prevent a user from submitting personally identifiable information (PII) to some later process, you can add a dialog node that conditions on a pattern entity. Place the node at the beginning of the dialog tree to ensure that it checks the input first. For example, the entity could check for US Social Security number patterns or email address patterns. It can then respond with something like, `Please do not submit personally identifiable information. Can you reenter your request?` You can optionally reset the context to ensure that the user-submitted information with PII is not retained.

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)
```

Your assistant 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	

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	

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	Regex engine	Dialog value	Explanation
number	value		
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 <code>((958) (555))</code> .
groups[2]	(958)	958	Match against the group that is included as the first operand in the OR expression <code>((958) (555))</code>
groups[3]	(555)	null	No match against the group that is included as the second operand in the OR expression <code>((958) (555))</code>
groups[4]	(\d{3})	234	String that matches the regular expression that is defined for the group.
groups[5]	(\d{4})	3456	String that matches the regular expression that is defined for the group.

Group details

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

Variable	Value
array_of_matched_groups	

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

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

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

Array elements

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	

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	

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

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Personalizing the dialog with context](#).

Personalizing the dialog with context

To personalize the conversation, your assistant can collect information from the customer and then refer back to it later in the conversation.

Retaining information across dialog turns

The dialog in a dialog skill is stateless, meaning that it does not retain information from one interaction with the user to the next. When you add a dialog skill to an assistant and deploy it, the assistant saves the context from one message call and then re-submits it on the next request throughout the current session. The current session lasts for as long a user interacts with the assistant plus the designated session inactivity time frame. The maximum session inactivity time allowed ranges from 5 minutes to 7 days, depending on your plan type. If you do not add the dialog skill to an assistant, it is your responsibility as the custom application developer to maintain any continuing information that the application needs.

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.

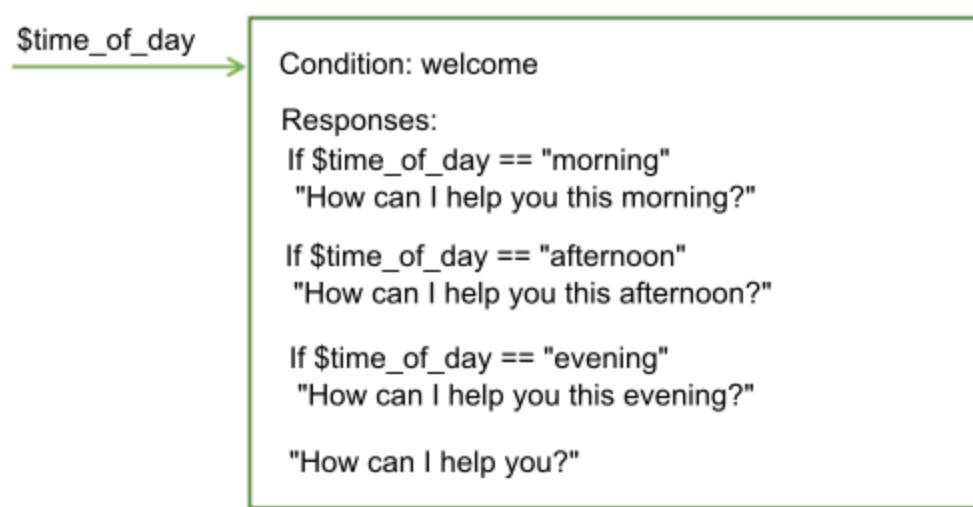
Learn more:

- [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.

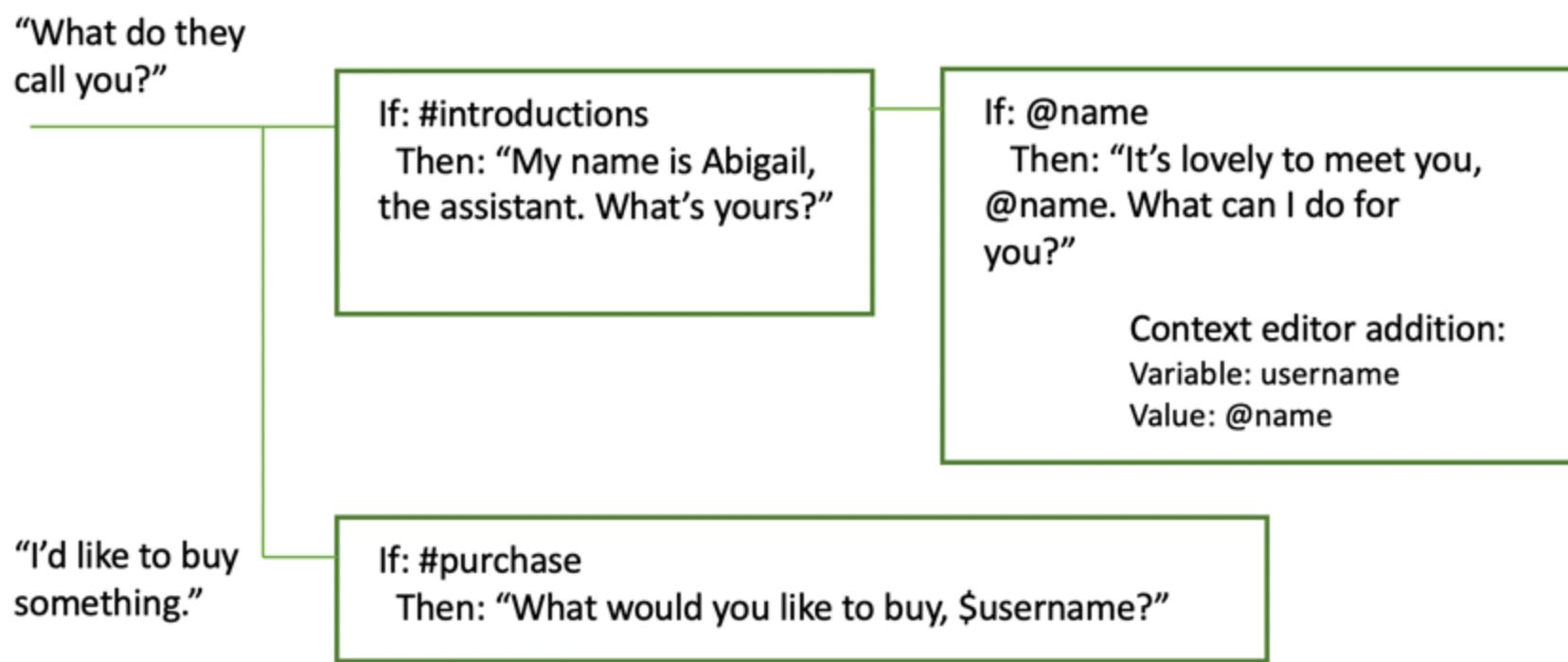


In this example, the dialog knows that the application sets the variable to one of these values: *morning*, *afternoon*, or *evening*. It can check for each value, and depending on which value is present, return the appropriate greeting. If the variable is not passed or has a value that does not match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

 **Tip:** If you're using multiple conditioned responses, you must click the *Customize response* icon  to see the menu that is associated with the response.

3. Click **Open context editor**.



4. Add the variable name and value pair to the **Variable** and **Value** fields.
 - o The **name** can contain any upper- and lowercase alphabetic characters, numeric characters (0-9), and underscores.

 **Tip:** 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.

 - o 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	<?input.text?>

Capturing user input

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	@place

Capturing an entity mention

For example, the user input is, `I want to go to Paris.` If your `@place` entity recognizes `Paris`, then your assistant 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	<?input.text.extract('[\d]+',0)?>

Using a String method

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

Variable	Value
email	<? @email.literal ?>

Capturing a pattern entity value

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
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 your assistant 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 your assistant 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 your assistant 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 your assistant 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 your assistant. Your assistant 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	<code><? input.text ?></code>
contains_yes	<code><? \$user_input.contains('Yes') ?></code>

Using two context variables to check for a value in user input

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	<code><? input.text.contains('Yes') ?></code>

Using a single context variable

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.
 - o To add a context variable that is processed after a response condition for a slot is met, perform the following steps:
 1. Click the *Edit slot* icon .
 2. Click the *Options* icon , and then select **Enable conditional responses**.
 3. Click the *Edit response* icon  next to the response with which you want to associate the context variable.
 4. Click the *Options* icon  in the response section, and then click **Open context editor**.
 5. Add the variable name and value pair to the **Variable** and **Value** fields.

Condition: #reservation

Slot	Prompt
Guests	How many people are dining? 
Restaurant	Where would
Date	What day is t
Time	What time do

Configure slot 1

Check for: @sys-number Save it as: \$guests

If @sys-number is not present, then ask:
How many people are dining?

When user responds, if @sys-number is Found:

If bot recognizes
1. true \$guests it is. 

Not found:

If bot recognizes	Respond with
1. true	What number of people need seats at the table?

Configure slot > "Found" Response 1

If bot recognizes: true

Then respond with:

1. \$guests it is.	 Open JSON editor
	Open context editor

- 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](#).

Condition: #reservation

Slot	Prompt
Guests	How many people are dining? 
Restaurant	Where would
Date	What day is t
Time	What time do

Configure slot 1

Check for: @sys-number Save it as: \$guests

If @sys-number is not present, then ask:
How many people are dining?

When user responds, if @sys-number is Found:

\$guests it is.

Not found:

What number of people need seats at the table?

 Open 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.

Tip: 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:

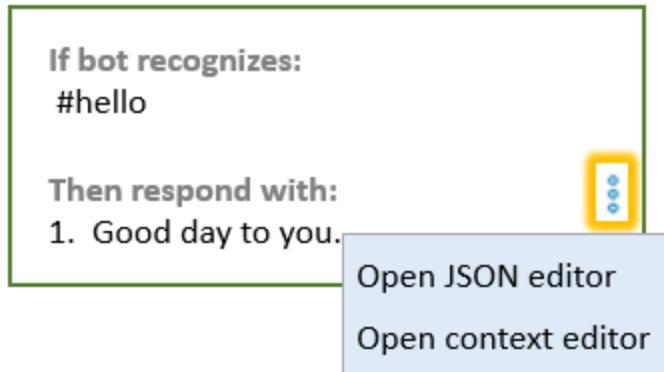
```
{
  "context": {
    "dessert": "cake",
    "toppings_array": [
      "onions",
      "olives"
    ],
    "age": 18,
    "full_name": {
      "first": "Jane",
      "last": "Doe"
    }
  },
  "output": {}
}
```

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

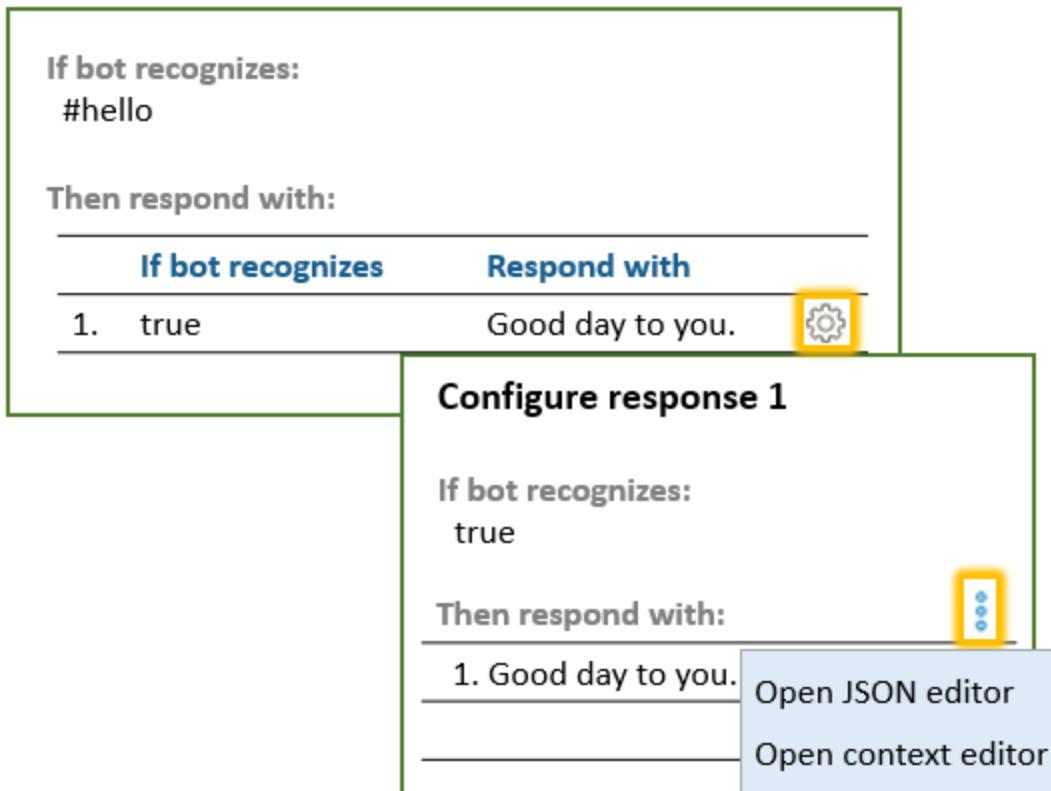
- Click to open the dialog node to which you want to add the context variable.

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.

- 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.



- 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:

```
{  
  "context": {  
    "toppings_array": ["onion", "olives"]  
  }  
}
```

Make this update:

```
{  
  "context": {  
    "toppings_array": ["ketchup", "tomatoes"]  
  }  
}
```

Result:

```
{  
  "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) ?>"  
  }  
}
```

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Improving your conversation](#).

Improving your conversation

Test your dialog, and organize your dialog nodes.

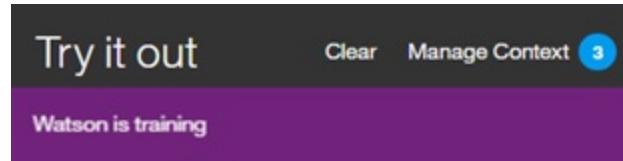
Testing your dialog

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

Queries you submit through the "Try it out" pane generate `/message` API calls, but they are not logged and do not incur charges.

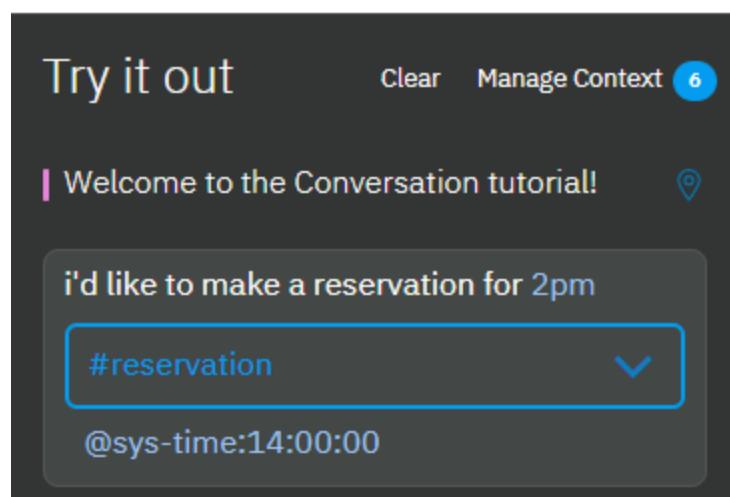
1. From the Dialog page, click the  icon.
2. In the chat pane, type some text and then press Enter.

 **Tip:** 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:

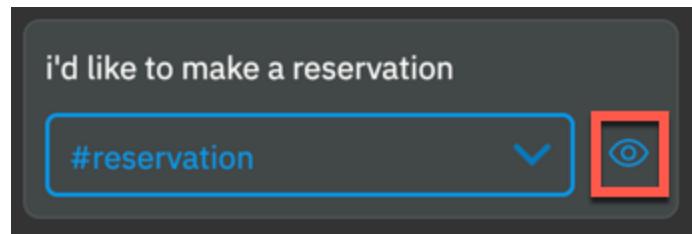


3. Check the response to see if the dialog correctly interpreted your input and chose the appropriate response.

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



4. To see the top intents that were recognized in the test input along with their confidence scores, hover over the eye icon that is displayed next to the intent with the highest confidence score.



5. If the response is not what you expected it to be, you can take the following actions from the "Try it out" pane:
 - If you want to edit an entity that is recognized in the input, click the entity name to open it in the Entities page.
 - If the wrong intent is recognized, you can click the arrow next to the intent name to correct it or mark the topic as irrelevant. For more information, see [Making training data improvements](#).
6. If you want to know which node in the dialog tree triggered a response, click the **Location** icon next to it.



Tip: If you are not already on the Dialog page, open it.

The source node is given focus and the route that your assistant 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.

7. 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, the following context variables are listed:

- **\$timezone**: The *Try it out* pane 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.
- **\$metadata**: This context variable contains a user ID name and value pair. Because no **user_id** is specified by default in the *Try it out* pane, the **conversation_id** is used as the **user_id** value. At run time, the ID value is typically passed to the assistant from whatever integration you are using. If you design a custom application, you set this value yourself.

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.

8. 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.

What to do next

Make changes to the dialog to address issues you see when testing:

- 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.

If you are ready to put the conversation to work helping your users, integrate your assistant with a messaging platform or custom application. See [Adding integrations](#).

Searching your dialog

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

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



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

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

```
$ ![Intent search return](images/search_dialog.png)
```

Copying a dialog node

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

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

1. On the node you want to copy, click the **More :** 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 your assistant processes the nodes within the folder directly. If a condition is specified, your assistant 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 your assistant 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 your assistant, 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 after the selected node.

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

You can add a folder after 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.

Tip: 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 node limits

The number of dialog nodes you can create per skill depends on your plan type.

Plan	Dialog nodes per skill
Enterprise	100,000
Premium (legacy)	100,000
Plus	100,000
Trial	25,000
Lite	25,000

Plan details

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

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

Finding a dialog node by its node ID

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

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

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

1. From the Dialog page, 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 syntax similar to the following:

```
https://{{location}}.assistant.watson.cloud.ibm.com/{{location}}/{{instance-id}}/skills/{{skill-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 page 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.



Tip: If you still cannot find the node, you can export the dialog skill and use a JSON editor to search the skill JSON file.

How many nodes are in my dialog?

To see the number of dialog nodes in a dialog skill, do one of the following things:

- If it is not associated with an assistant already, add the dialog skill to an assistant, and then view the skill tile from the main page of the assistant. The *trained data* section lists the number of dialog nodes.
- Send a GET request to the /dialog_nodes API endpoint, and include the `include_count=true` parameter. For example:

```
$ curl -u "apikey:{apikey}" "{url}/v1/workspaces/{workspace_id}/dialog_nodes?version=2018-09-20&include_count=true"
```

where {url} is the appropriate URL for your instance. For more details, see [Service endpoint](#).

In the response, the `total` attribute in the `pagination` object contains the number of dialog nodes.

If the total seems larger than you expected, it might be because the dialog that you build from the application is translated into a JSON object. Some fields that appear to be part of a single node are actually structured as separate dialog nodes in the underlying JSON object.

```
$ - Each node and folder is represented as its own node.  
- Each conditional response that is associated with a single dialog node is represented as an individual node.  
- For a node with slots, each slot, slot found response, slot not found response, slot handler, and if set, the "prompt for everything" response is an individual node. In effect, one node with three slots might be equivalent to eleven dialog nodes.
```

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Controlling the conversational flow](#).

Controlling the conversational flow

Understand how your dialog is processed when a person interacts with your assistant at run time. Learn to use the conversation flow to your advantage and alter the flow if necessary.

You can impact the flow of a conversation with the following features:

- [Disambiguation](#)
- [Digressions](#)

Disambiguation

Disambiguation instructs your assistant to ask the customer for help when more than one dialog node can respond to a customer's 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.



Disambiguation is triggered when the following conditions are met:

- The confidence scores of the runner-up intents that are detected in the user input are close in value to 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.

- A description of the node's purpose is provided for the node in the node name field. (Alternatively, a description can be included in the external node name field.)

A node with a boolean node condition that evaluates to true is likely to be included in the disambiguation list. For example, if the node checks for an entity type and the entity is mentioned in the user input, it is eligible to be included in the list. Such nodes do not trigger disambiguation, but if disambiguation is triggered, they are likely to be included in the resulting disambiguation list.

Learn more

- [Disambiguation example](#)
- [Editing the disambiguation configuration](#)
- [Choosing nodes to disable](#)
- [Disabling disambiguation](#)
- [Handling none of the above](#)
- [Testing disambiguation](#)

For information about how disambiguation works with actions skills, see [Disambiguation](#).

Disambiguation example

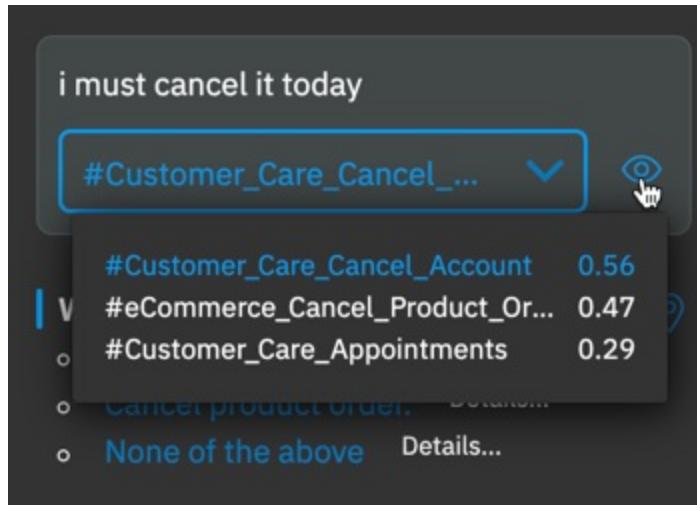
For example, you have a dialog that has two nodes with intent conditions that address cancellation requests. The conditions are:

- 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":0.5602024316787719},
  {"intent":"eCommerce_Cancel_Product_Order","confidence":0.46903514862060547},
  {"intent":"Customer_Care_Appointments","confidence":0.29033891558647157},
  {"intent":"General_Greetings","confidence":0.2894785046577454},
```

In fact, if you test from the "Try it out" pane, you can hover over the eye icon to see the top three intents that were recognized in the test input.

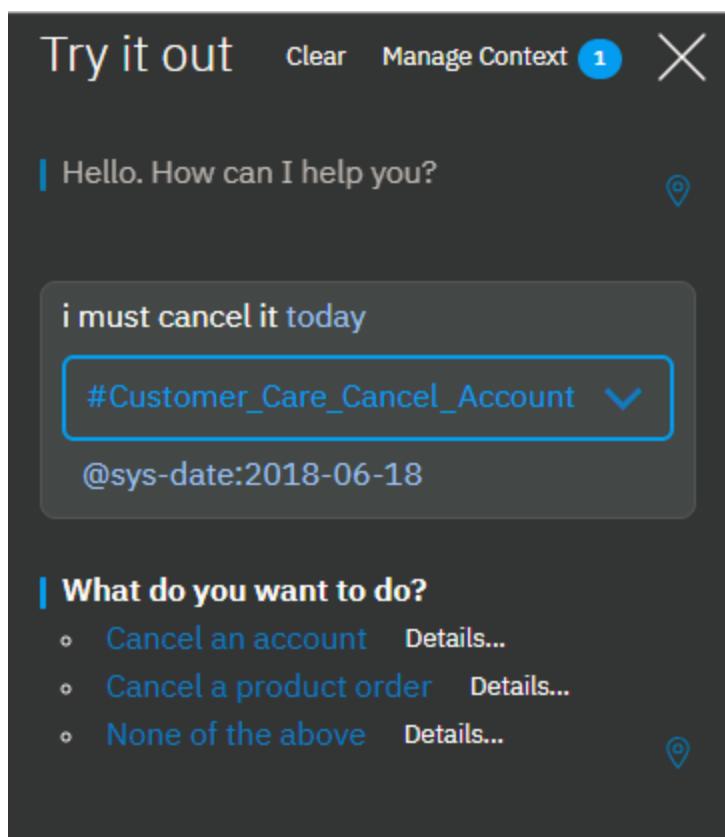


Your assistant is **0.5618281841278076** (56%) confident that the user goal matches the `#Customer_Care_Cancel_Account` intent. If another intent has a confidence score that is close to the score of this top intent, then disambiguation is triggered. In this example, the `#eCommerce_Cancel_Product_Order` intent has a close confidence score of 46%.

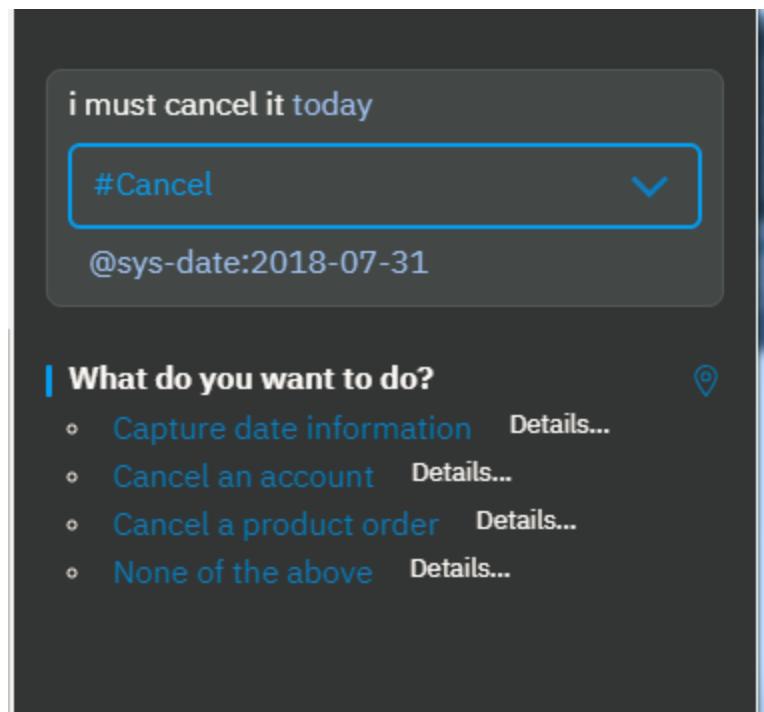
As a result, when the user input is `i must cancel it today`, both dialog nodes are likely to be considered viable candidates to respond. To determine which dialog node to process, the assistant asks the user to pick one. And to help the user choose between them, the assistant provides a short summary of what each node does. The summary text is extracted directly from the node's *name* field. If present and if a description is added to it, then the text is taken from the *external node name* field instead.



Note: The description that is displayed in the disambiguation list comes from the name (or external node name) of the last node that is processed in the branch where the intent match occurs. For more information, see [Disable jumped-to utility nodes](#).



Notice that your assistant 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 conditions on the **@sys-date** entity, then it is also likely to be included in the list of disambiguation choices. This image shows it included in the list as the *Capture date information* option.



Editing the disambiguation configuration

Disambiguation is enabled automatically for all new dialog skills. You can change the settings that are applied automatically to disambiguation from the **Options** page.

To edit the disambiguation settings, complete the following steps:

1. From the Skills menu, click **Options**.
2. Click **Disambiguation**.
3. In the **Disambiguation message** field, add text to show before the list of dialog node options. For example, *What do you want to do?*
4. In the **Anything else** field, add text to display as an additional option that users can pick if none of the other dialog node options 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 your assistant does if a user chooses this option, see [Handling none of the above](#).

5. If you want to limit the number of disambiguation options that can be displayed to a user, then in the **Maximum number of suggestions** field, specify a number between 2 and 5.

Your changes are automatically saved.

Tip: You can use the API to modify additional disambiguation settings. These settings include the disambiguation sensitivity, which affects how often disambiguation is triggered and how many choices are included. For more information, see the [API Reference](#).

Next, you must decide which dialog nodes you want to make eligible for disambiguation. From the Skills menu, click **Dialog**.

Choosing nodes to not show as disambiguation options

All nodes are eligible to be included in the disambiguation list.

\$ - Nodes at any level of the tree hierarchy are included.
- Nodes that condition on intents, entities, special conditions, context variables, or any combination of these values are included.

Consider hiding some nodes from the disambiguation list.

- **Hide root nodes with `welcome` and `anything_else` conditions**

Unless you added extra functionality to these nodes, they typically are not useful options to include in a disambiguation list.

- **Hide jumped-to utility nodes**

 **Important:** The text that is displayed in the disambiguation list is populated from the node name (or external node name) of the *last node that is processed* in the branch where the node condition is matched.

You do not want the name of a utility node, such as one that thanks the user, or says goodbye, or asks for feedback on answer quality to be shown in the disambiguation list instead of a phrase that describes the purpose of the matched root node.

For example, maybe a root node with the matching intent condition of `#store-location` jumps to a node that asks users if they are satisfied with the response. If the `#check_satisfaction` node has a node name and has disambiguation enabled, then the name for that jumped-to node is displayed in the disambiguation list. As a result, `Check satisfaction` is displayed in the disambiguation list to represent the `#store-location` branch instead of the `Get store location` name from the root node.

- **Hide root nodes that condition on an entity or context variable only**

Only a node with a matched intent can trigger disambiguation. However, once it's triggered, any node with a condition that matches is included in the disambiguation list. When your dialog includes nodes that condition on entities, for example, the order of nodes in the tree hierarchy can become significant in unexpected ways.

- The order of nodes 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 your assistant 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.

- The order of nodes 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 might not be added as a disambiguation option because your assistant never reaches it. If it matched against the earlier node and was omitted, your assistant might not process the later node.

Hiding nodes from disambiguation altogether

You can prevent every node in a dialog or an individual dialog node from being included in the disambiguation list.

- To disable disambiguation entirely:
 - From the Skills menu, click **Options**.
 - On the *Disambiguation* page, set the switch to **Off**.
- To prevent a single dialog node from being included in the disambiguation list:
 - From the Skills menu, click **Dialog**. Click the node to open it in the edit view.


Cancel order

Node name will be shown to customers for disambiguation so use something descriptive. [Settings](#)

- Click **Settings**.

[Cancel order](#)

Node name will be shown to customers for disambiguation so use something descriptive.

Disambiguation

Choose whether to include this node in the list of options shown to customers when the assistant asks customer to clarify their meaning. [Learn more](#)

Show node name

Off  On

[Hide settings](#)

- Set the **Show node name** switch to **Off**.
-  If you added a node summary description to the **external node name** field instead of the **name** field, remove it.

The *external node name* field serves two purposes. It provides information about the node to customers when it is included in a disambiguation list. It also describes the node in a chat summary that is shared with service desk agents when a conversation is transferred to a person. The *external node name* field is only visible in skills that are part of a paid plan instance. If the *external node name* field contains text, its text is used, whether or not there is text in the *name* field.

Optionally, add a node purpose summary that can be displayed to users. 

Enter external node name

 **Status:** this node will not be available for selection by human agents unless an external node name is provided.

For each node, 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. See [Testing disambiguation](#).

 **Tip:** Watson Assistant can recognize intent conflicts, which occur when two or more intents have user examples that overlap. [Resolve any such conflicts](#) first to ensure that the intents themselves are as unique as possible, which helps your assistant attain better intent confidence scores.

Prioritizing a node over disambiguation

Some nodes are important enough to your customer's satisfaction that you want them to be returned on their own, outside of a disambiguation list, when the assistant is confident enough that the node will meet a customer need.

For example, you might have a node that matches the `#stolen_card` intent. Whenever an incoming message suggests that a customer wants to report a stolen credit card, you don't want your assistant's response to be lost in a disambiguation list of options. You want the assistant to respond with a single answer that assures the customer that it can address this urgent matter.

To design your dialog to prioritize a single node over disambiguation, complete the following steps:

1. In the node that conditions on the intent, enable multiple conditioned responses (Customize > Multiple conditioned responses).
2. Add a conditioned response with the following condition:

`intent.confidence > n`

where `n` is a confidence score that makes sense for your training data. For example:

`intent.confidence > 0.7`

3. Move the response up to be first in the list of conditioned responses.
4. Click the gear icon to customize the conditioned response.
5. From the *Assistant responds* section, open the context editor.
6. Add the following context variable:

Variable	Value
<code>'system'</code>	<code>'{"prevent_disambiguation":true}'</code>

Context variable details

7. Click **Save**.

Alternatively, you can add a root-level node with a condition such as the following:

```
#stolen_card && intent.confidence > 0.7
```

Place this node higher in the tree than the node that conditions on `#stolen_card`, which allows the node to be included in a disambiguation list.

8. Test your dialog. Make sure that the node's response is returned instead of a disambiguation list when the appropriate confidence score threshold is met.

Handling none of the above

When a user clicks the *None of the above* option, your assistant 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.

 **Tip:** 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 your assistant 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 will be displayed instead.

Testing disambiguation

 **Important:** When testing, the order in which the options are listed might change from test run to test run. In fact, the options themselves that are included in the disambiguation list might change from one test run to the next.

This behavior is intended. As part of development that is in progress to help the assistant learn automatically from user choices, the choices included and their order in the disambiguation list is being randomized on purpose. Changing the order helps to avoid bias that can be introduced by a percentage of people who always pick the first option without carefully reviewing all of their choices beforehand.

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 name (or external node name) 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.
 - To see the confidence scores of the top three intents that were detected in the input, hover over the eye icon in the "Try it out" pane.
 - To see the confidence scores of all the intents that are detected in the 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 your assistant 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 ?>`.

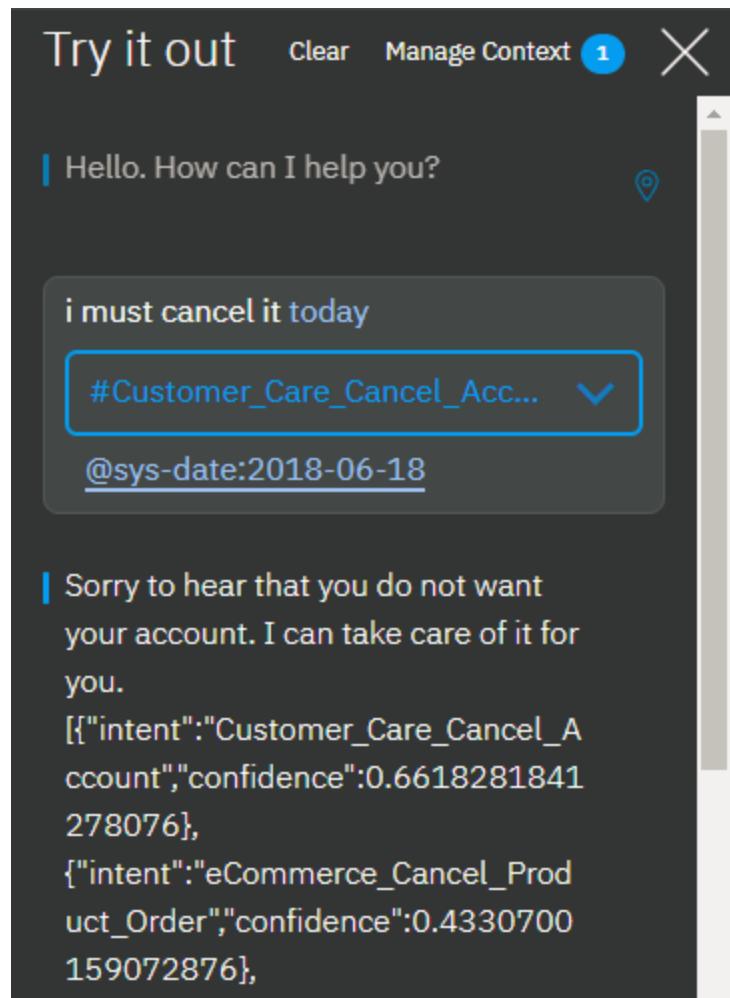
This SpEL expression shows the entities that were detected in the user input as an array. The array includes the entity name, location of the entity mention within the user input string, the entity mention string, and the level of confidence that your assistant has that the term is a mention of the entity 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 `name` field (or `external node name` field) for at least one of the nodes that you anticipate will be listed as a disambiguation option.

- 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 your assistant 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 *name* or *external node name* fields from which you removed text.

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.

- [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** switch to **No** to prevent the dialog from returning to the current node and continuing to process its branch.

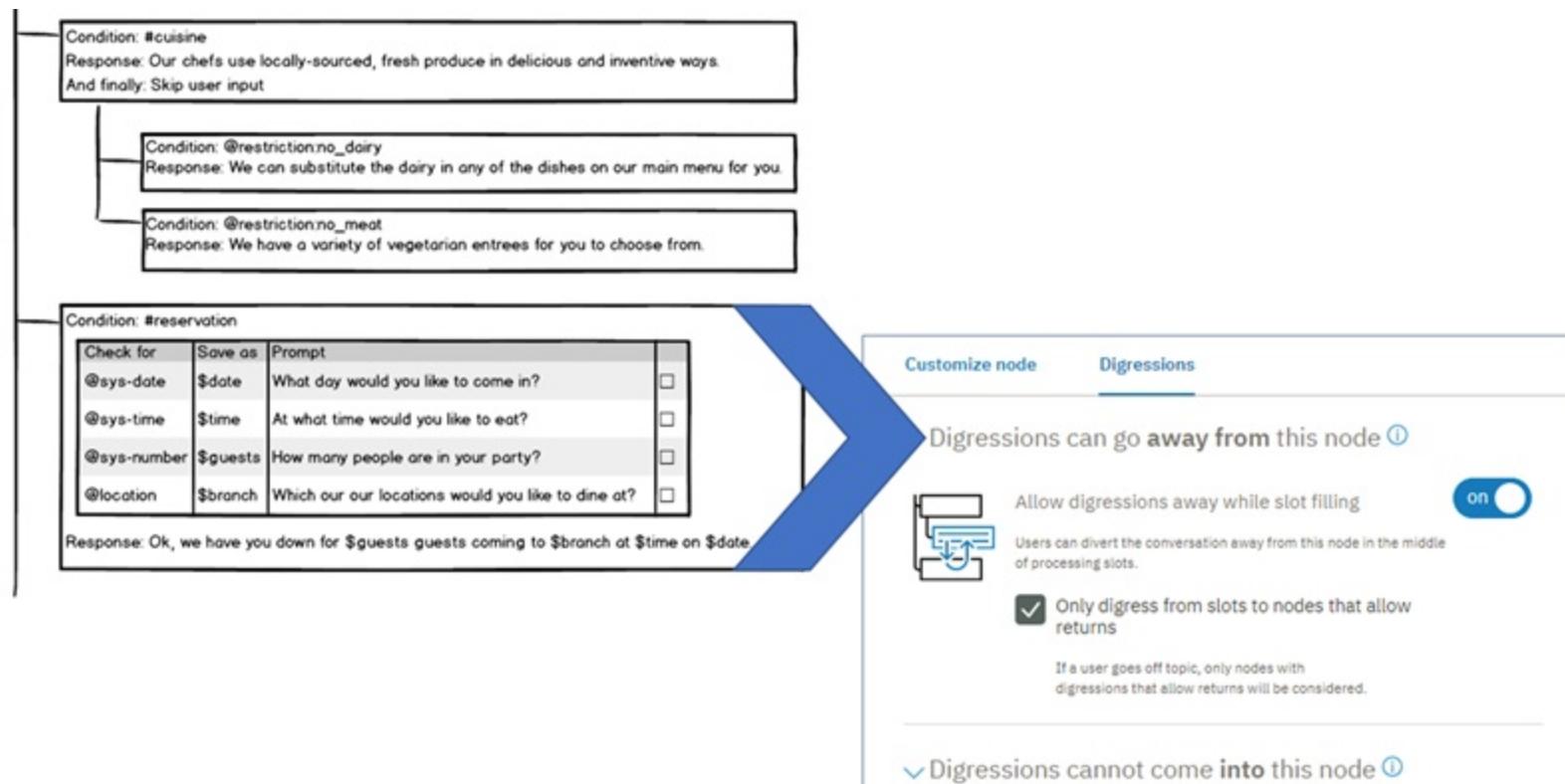
For example, if the user asks, **Do you sell cupcakes?** and the response, **We offer cupcakes in a variety of flavors and sizes** is displayed before the user changes subjects, you might not want the dialog to return to where it left off. Especially, if the child nodes only address possible follow-up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

Customize "#cuisine"

Digressions

This node has **edited** digressions settings ⓘ

> **Digressions cannot go away from this node** ⓘ

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.

Customize node **Digressions**

Cancel **Apply**

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.

Customize "#cuisine"

Digressions

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.

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.

Customize "#reservation"

Digressions

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.

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** switch 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** switch 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, your assistant 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.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Autocorrecting user input](#).

Autocorrecting user input

Autocorrection fixes misspellings that users make in their requests. The corrected words are used to match to an action or an intent.

Autocorrection corrects user input in the following way:

- Original input: `letme applt for a memberdhip`
- Corrected input: `let me apply for a membership`

When your assistant evaluates whether to correct the spelling of a word, it does not rely on a simple dictionary lookup process. Instead, it uses a combination of natural language processing and probabilistic models to assess whether a term is, in fact, misspelled and should be corrected.

Autocorrection is enabled automatically for all English-language assistants. It is also available in French-language assistants, but is disabled by default. Autocorrection isn't available for any other languages.

Disabling autocorrection

If necessary, you can disable autocorrection for your assistant.



Note: If you find that a domain-specific term is being corrected that shouldn't be, you can prevent the correction from happening by adding the term or phrase to your training data. For more details, see [Autocorrection rules](#).

If you are using actions in your assistant, follow these steps to disable autocorrection:

1. On the **Actions** page, click **Global settings** ! .
2. Click the **Autocorrection** tab.
3. Set the switch to **Off**, then click **Save**.

If you are using dialog in your assistant, follow these steps to disable autocorrection:

1. In the **Options** section, click **Autocorrection**.
2. Set the switch to **Off**.

Testing autocorrection in dialog

If you are using dialog, you can test autocorrection using **Try it out**.

1. In **Try it out**, enter a request that includes some misspelled words.

If words in your input are misspelled, they are corrected automatically, and an  icon is displayed. The corrected utterance is underlined.

2. Hover over the underlined utterance to see the original wording.

If there are misspelled terms that you expected your assistant to correct, but it did not, then review the rules that your assistant uses to decide whether to correct a word to see if the word falls into the category of words that your assistant intentionally does not change.

Autocorrection rules

To avoid overcorrection, your assistant does not correct the spelling of the following types of input:

- Capitalized words
- Emojis
- Locations, such as states and street addresses
- Numbers and units of measurement or time
- Proper nouns, such as common first names or company names
- Text within quotation marks
- Words containing special characters, such as hyphens (-), asterisks (*), ampersands (&), or at signs (@), including those used in email addresses or URLs.
- Words that *belong*, meaning words that have implied significance because they occur in your action steps or dialog entity values, entity synonyms, or intent user examples.

How is spelling autocorrection related to fuzzy matching?

In dialog, *fuzzy matching* helps your assistant recognize dictionary-based entity mentions in user input. It uses a dictionary lookup approach to match a word from the user input to an existing entity value or synonym in the skill's training data. For example, if the user enters **boook**, and your training data contains a **@reading_material** entity with a **book** value, then fuzzy matching recognizes that the two terms (**boook** and **book**) mean the same thing.

In dialog, when you enable both autocorrection and fuzzy matching, the fuzzy matching function runs before autocorrection is triggered. If it finds a term that it can match to an existing dictionary entity value or synonym, it adds the term to the list of words that *belong* to the skill, and does not correct it.

For example, if a user enters a sentence like **I wnt to buy a boook**, fuzzy matching recognizes that the term **boook** means the same thing as your entity value **book**, and adds it to the protected words list. Your assistant corrects the input to be, **I want to buy a boook**. Notice that it corrects **wnt** but does *not* correct the spelling of **boook**. If you see this type of result when you are testing your dialog, you might think your assistant is misbehaving. However, your assistant is not. Thanks to fuzzy matching, it correctly identifies **boook** as a **@reading_material** entity mention. And thanks to autocorrection revising the term to **want**, your assistant is able to map the input to your **#buy_something** intent. Each feature does its part to help your assistant understand the meaning of the user input.

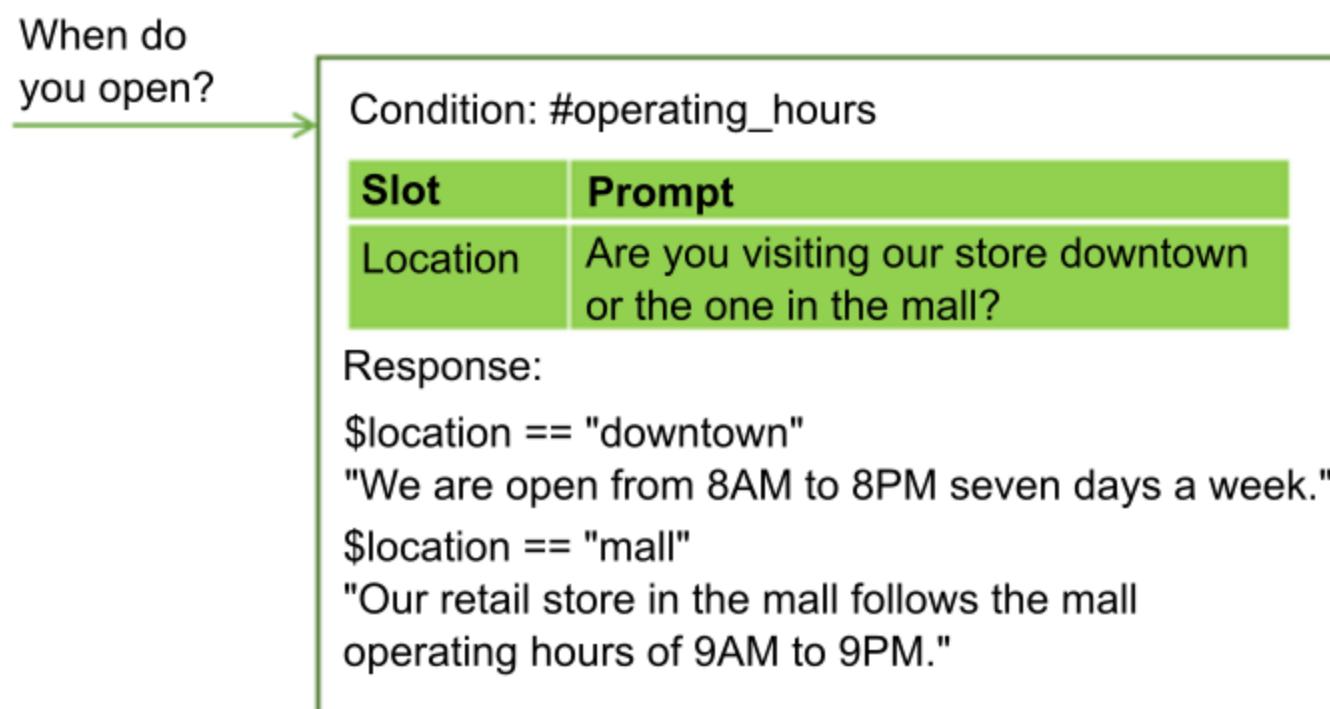
Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Gathering information with slots](#).

Gathering information with slots

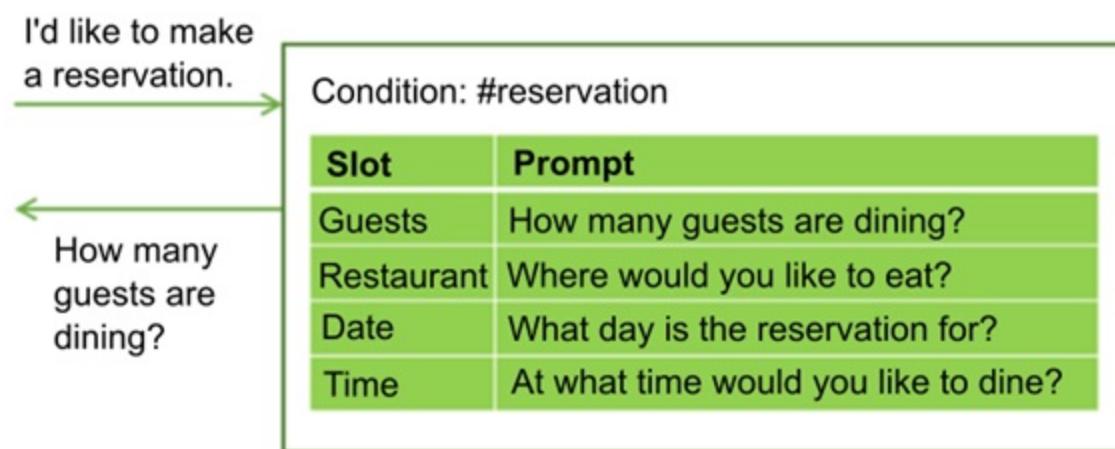
Add slots to a dialog node to gather multiple pieces of information from a user within that node. Slots collect information at the user's pace. Details that a user provides up front are saved, and your assistant asks only for the missing details it needs to fulfill the request.

Why add slots?

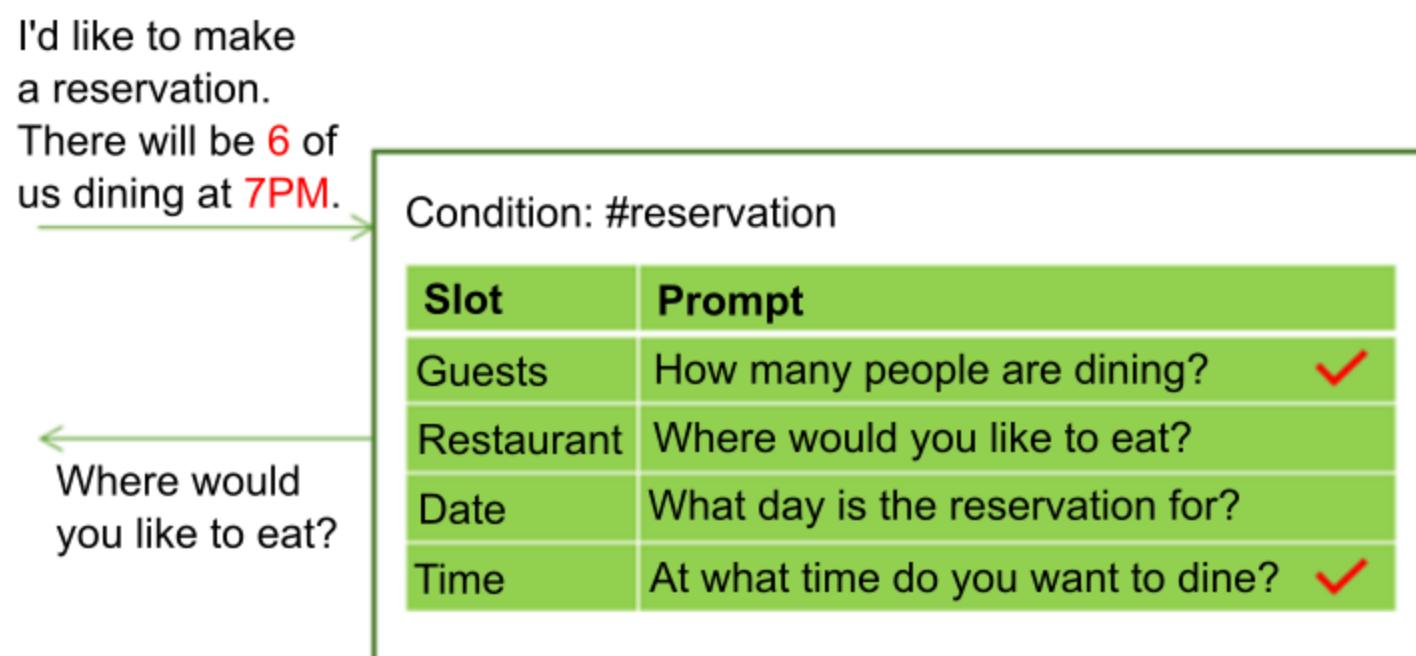
Use slots to get the information you need before you can respond accurately to the user. For example, if users ask about operating hours, but the hours differ by store location, you could ask a follow-up question about which store location they plan to visit before you answer. You can then add response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



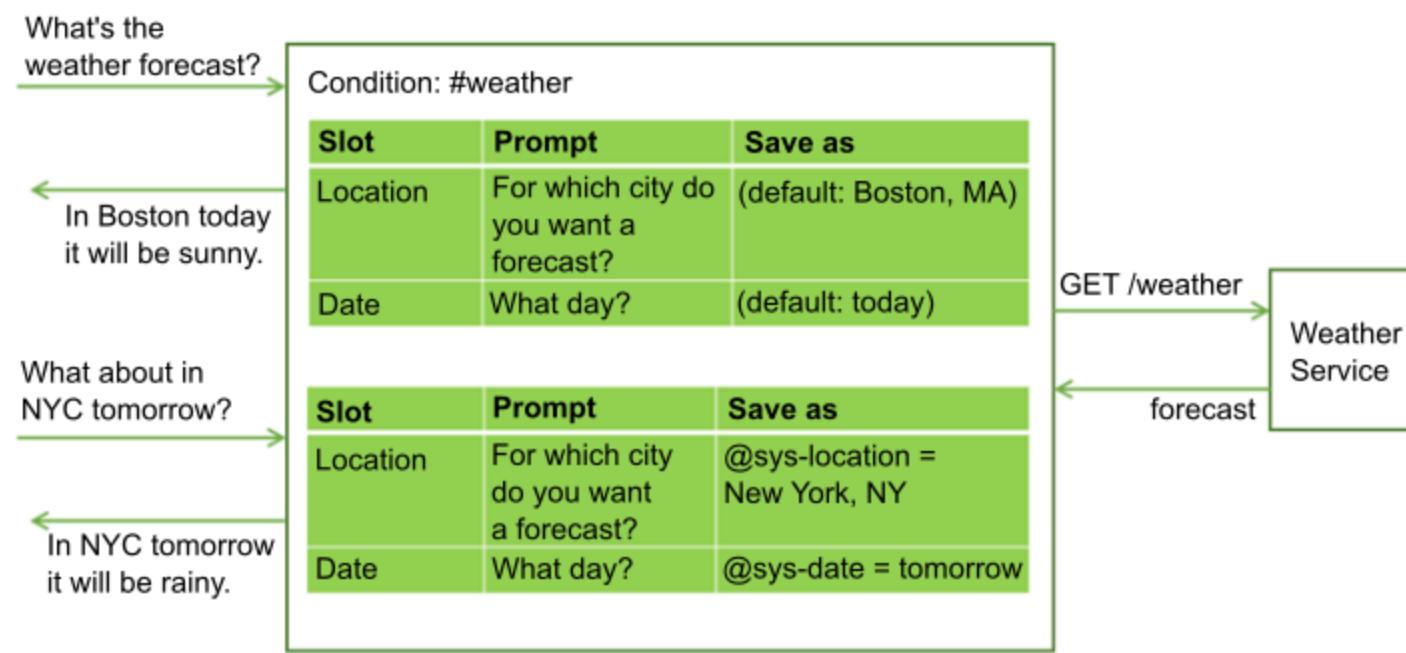
Slots can help you to collect multiple pieces of information that you need to complete a complex task for a user, such as making a dinner reservation.



The user might provide values for 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. Your assistant recognizes and stores both of them, each one in its corresponding slot. It then displays the prompt that is associated with the next empty slot.



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



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

Adding slots

- 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
- If you have not started to create a dialog, follow the instructions in [Creating a dialog](#) to create one.
- From the dialog node edit view, click **Customize**, and then set the **Slots** switch to **On**.

For more information about the **Prompt for everything** checkbox, see [Asking for everything at once](#).

- 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 it as* field. It specifies both **what to check for** and **what to save**. If you want to change how the value is saved, then add the expression that reformats the value to the *Save it as* field.

For example, if the entity is a pattern entity, such as `@email`, then after adding the entity name, append `.literal` to it. Adding `.literal` indicates 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.

In some cases, you might want to use an expression to capture the value, but not apply the expression to what is saved. In such cases, you can use one value in the *Check for* field to capture the value, and then open the JSON editor to change the value of the context variable, so it saves something else.

Note: Any edit you make to a slot's context variable value in the JSON editor is not reflected in the **Check for** field after you exit the JSON editor. You must reopen the JSON editor to see what will be saved as the context variable value.

Avoid checking for context variable values in the *Check for* field. Because the value you check for is also the value that is saved, using a context variable in the condition can lead to unexpected behavior.

- Save it 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 reuse a context variable that is used elsewhere in the dialog. If the context variable has a value already, then the slot's prompt is not displayed. 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 your assistant waits for the user to respond.

If you want the prompt to be something other than a text response, you can change the response type by clicking the **Customize slot** icon. Click **Text** to choose a different response type.

Response type options:

- [Connect to human agent](#)
- [Image](#)
- [Option](#)
- [Pause](#)
- [Search skill](#) 

 **Note:** This response type is visible only to users of paid plans.

- [Text](#)

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

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

Example slots for pizza order

5. **Add slot value validation**: 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 **Customize slot**  icon) and define the follow-up statements:

Found: Displayed after the user provides the expected information.

Not found: Displayed only if the information provided by the user is not understood, which means all of the following are true:

- None of the active slots in the node are filled successfully
- No slot handlers are understood
- If digressions away are enabled for the node, no top-level dialog nodes are triggered as a digression from slot filling

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](#).

6. **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. Your assistant 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 information in case the user specifies any. However, you don't want to ask all users for dietary information since it is irrelevant in most cases.

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

 **Tip:** If you make a slot optional, only reference its context variable in the node-level response text if you can word it such that it makes sense even if no value is provided for the slot. For example, you might word a summary statement like this, **I am ordering a \$size dietary pizza for delivery at \$time.** The resulting text makes sense whether the dietary restriction information, such as **gluten-free** or **dairy-free**, is provided or not. The result is either, **I am ordering a large gluten-free pizza for delivery at 3:00PM.** or **I am ordering a large pizza for delivery at 3:00PM.**

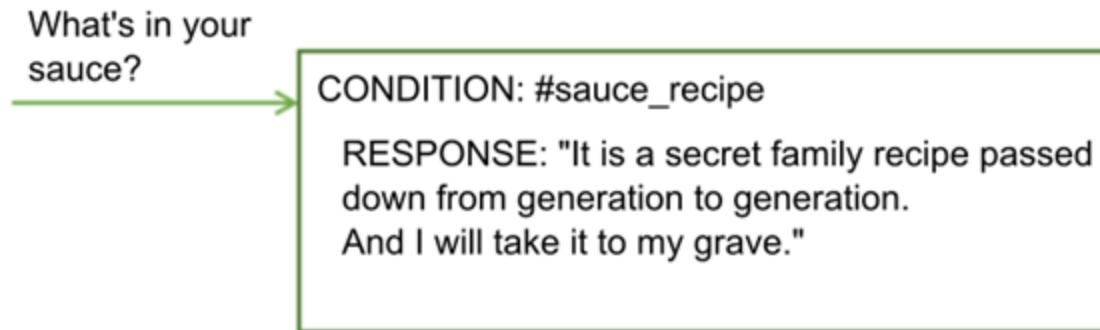
- **Conditional**: If you want a slot to be enabled only under certain conditions, then you can add a condition to it. For example, if slot 1 asks for a meeting start time, slot 2 captures the meeting duration, and slot 3 captures the end time, then you might want to enable slot 3 (and ask for the meeting end time) only if a value for slot 2 is not provided. To make a slot conditional, edit the slot, and then from the **More**  menu, select **Enable condition**. Define the condition that must be met for the slot to be enabled.

 **Tip:** 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.

7. **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.

8. **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!**

You can alternatively show an image or list of options as a response instead of a text response. See [Response type options](#).

If you want to define different responses based on certain conditions, click **Customize**, and then set the **Multiple responses** switch to **On**. For information about conditional responses, see [Conditional responses](#).

9. **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 your assistant 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.

Property name	Description
<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.
<code>slot_in_focus</code>	Forces the slot condition to be applied to the current slot only. See Getting confirmation for more details. You can use this property to collect and store the exact words that are submitted by a customer. See Collecting summary information from the customer .

Slot properties

Consider using these approaches for handling common tasks.

- [Asking for everything at once](#)

- [Capturing multiple values](#)
- [Reformatting values](#)
- [Dealing with zeros](#)
- [Getting confirmation](#)
- [Collecting summary information from the customer](#)
- [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 your assistant analyzes the input, it recognizes **large** as the pizza size and fills the **Size** slot with the value provided. Because one of the slots is filled, it skips displaying the initial prompt to avoid asking for the pizza size information again. Instead, it displays the prompts for any remaining slots with missing information.

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

Capturing multiple values

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

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

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

Multiple value slot

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

Reformatting values

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

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

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

See [Expression language methods](#) 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 your assistant prompts the user for a number again. To prevent this behavior, check for an **@sys-number** mention that is greater than or equal to zero in the slot condition.

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

1. Add **@sys-number >= 0** to the slot condition field, and then provide the context variable name and text prompt.

What you check for in the input is also what is saved in the slot context variable. However, in this case, you want only the number (such as **5**) to be saved. You do not want to save **5 > = 0**. To change what is saved, you must edit the value of the context variable.

2. Open the slot to edit it by clicking the **Customize slot** icon. From the **Options** menu, open the JSON editor.
3. Change the context variable value.

The value will look like this:

```
{  
  "context": {  
    "number": "@sys-number >= 0"  
  }  
}
```

Change it to look like this:

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

4. Save your changes.

 **Note:** The change you made to the context variable value is not reflected in the Check for field. You must reopen the JSON editor see what will be saved.

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 it as	Prompt	Follow-up if found	Follow-up if not found
#yes	#no	\$confirmation	I'm going to order you a \$size pizza for delivery at \$time. Should I go ahead?	Confirmation slot

Complex response Because users might include affirmative or negative statements at other times during the dialog (*Oh yes, we want the pizza delivered at 5pm*) or (*no guests tonight, let's make it a small*), use the **slot_in_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
```

 **Tip:** 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 check 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
}
}

```

Collecting summary information from the customer

You might want to prompt a user to supply free form text in a dialog node with slots that you can save and refer to later. To do so, follow these steps:

1. In the *Check for* field, add the following special property: `slot_in_focus`.
2. Optionally, change the context variable name for the slot in the *Save it as* field. For example, you might want to change it to something like `summary`.
3. In the *If not present, ask* field, ask the user to provide open-ended information. For example, `Can you summarize the problem?`
4. To store the input in the customer's exact words, edit what is saved by using the JSON editor.
5. Open the slot to edit it by clicking the **Customize slot**  icon. From the **Options**  menu, open the JSON editor.
6. Change the context variable value.

The value will look like this:

```
{
  "context": {
    "summary": "slot_in_focus"
  }
}
```

Change the value that is saved in the `$summary` context variable. You want to capture and store whatever text the user submits. To do so, use a SpEL expression that captures the input text.

```
{
  "context": {
    "summary": "<? input.text ?>"
  }
}
```

7. Save your changes.



Note: The change you made to the context variable value is not reflected in the *Check for* field. You must reopen the JSON editor to see what will be saved.

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 slot filling confusion

When a user input is evaluated, the slot with the first slot condition to match it is filled only. Test for the following possible causes of misinterpretation, and address them:

- **Problem:** The same entity is used in more than one slot.

For example, **@sys-date** is used to capture the departure date in one slot and arrival date in another.

Solution: Use slot found conditions that get clarification from the user about which date you are saving in a slot before you save it.

- **Problem:** A term fully or partially matches the entities in more than one slot condition.

For example, if one slot captures a product ID (**@id**) with a syntax like **GR1234** and another slot captures a number (**@number**), such as **1234**, then user input that contains an ID, such as **BR3344** might get claimed by the **@number** slot as a number reference and fill the **\$number** context variable with **3344**.

Solution: Place the slot with the entity condition that captures the longer pattern (@id) higher in the list of slots than the condition that captures the shorter pattern (@number).

- **Problem:** A term is recognized as more than one system entity type.

For example, if the user enters **May 2**, then your assistant recognizes both the **@sys-date** (2017-05-02) and **@sys-number** (2) entities.

Solution: In logic that is unique to the slots feature, when two system entities are recognized in a single user input, the one with the larger span is used. Therefore, even though your assistant recognizes both system entities in the text, only the system entity with the longer span (**@sys-date** with **2017-05-02**) is registered and applied to the slot.



Note: This workaround is not necessary if you are using the revised system entities. With the updated entities, a date reference is considered to be a **@sys-date** mention only, and is not also treated as a **@sys-number** mention. For more details, see [System entities](#).

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 **Customize 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: A 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 **@sys-number** entity for the number of pizzas in a takeout order. The Not found condition of **true** might be used to display a message prompting the user, just in case a number isn't provided in the conversation (for example, **We need to know how many pizzas you want.**).

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 your assistant 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 your assistant 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, your assistant 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.

Tip: 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 your assistant 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 your assistant 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. Your assistant 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, your assistant 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 it 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 {:#dialog-slots-stifle-found-responses}
```

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 {:#dialog-slots-node-level-handler}
```

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 handler that conditions on the #cancel intent, which recognizes utterances such as, <q>Forget it. I changed my mind.</q>

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.`

1. Choose what action you want your assistant 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 your assistant 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.

1. 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 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.

```
```json {:#codeblock}
{
 "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:

- Slot handlers in the order they are listed.
- First slot condition.
- Current slot level Found conditions.
- If digressions away are allowed, root level node conditions are checked for a match (except the final `anything else` node in the dialog tree root or a root folder).
- Current slot level Not found conditions.
- Final `anything else` node condition.

**Tip:** 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 [conversation repo](#) in GitHub.

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

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Defining information to look for in customer input](#).

## Defining information to look for in customer input

**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.

## Entity evaluation overview

Your assistant detects entities in the user input by using one of the following evaluation methods:

### Dictionary-based method

Your assistant looks for terms in the user input that match the values, synonyms, or patterns you define for the entity.

- **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, your assistant recognizes terms in the user input that exactly 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 specify a regular expression that defines a `text@text.com` pattern.

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

- **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.

## Annotation-based method

When you define an annotation-based entity, which is also referred to as a contextual entity, a model is trained on both the *annotated term* and the *context* in which the term is used in the sentence you annotate. This new contextual entity model enables your assistant 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.

- **Contextual entity:** First, you define a category of terms as an entity (`product`). Next, you go to the *Intents* page and mine your existing intent user examples to find any mentions of the entity, 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` entity.

For training purposes, the term you annotated, `Coach bag`, is added as a value of the `@product` entity.

At run time, your assistant evaluates terms based on the context in which they are used in the sentence only. If the structure of a user request that mentions the term matches the structure of an intent user example in which a mention is labeled, then your assistant interprets the term to be a mention of that entity type. 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`), your assistant recognizes `Gucci bag` as a `@product` entity mention.

When a contextual entity model is used for an entity, your assistant does *not* look for exact text or pattern matches for the entity in the user input, but focuses instead on the context of the sentence in which the entity is mentioned.

If you choose to define entity values by using annotations, add at least 10 annotations per entity to give the contextual entity model enough data to be reliable.

To learn more about contextual entities, [read this blog post](#).

To learn about how to use contextual entities to identify names and locations, see the [Detecting Names And Locations With Watson Assistant](#) blog post on Medium.

## Creating entities

1. Open your dialog skill and then click the **Entities** tab.

2. Click **Create entity**.

You can also click **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.



**Tip:** The at sign `@` is prepended to the entity name automatically to identify the term as an entity. You do not need to add it.

4. Click **Create entity**.

**Entity name**

Name your entity to match the category of values that it will detect.

@meeting

Create entity

- For this entity, choose whether you want your assistant to use a dictionary-based or annotation-based approach to find mentions of it, and then follow the appropriate procedure.

**For each entity that you create, choose one entity type to use only.** As soon as you add an annotation for an entity, the contextual model is initialized and becomes the primary approach for analyzing user input to find mentions of that entity. The context in which the mention is used in the user input takes precedence over any exact matches that might be present. See [Entity evaluation overview](#) for more information about how each type is evaluated.

- Dictionary-based entities
- Annotation-based entities

## Adding dictionary-based entities

Dictionary-based entities are those for which you define specific terms, synonyms, or patterns. At run time, your assistant finds entity mentions only when a term in the user input exactly matches (or closely matches if fuzzy matching is enabled) the value or one of its synonyms.

- In the **Value name** field, type a value. For example, for the `@city` entity, you might type `New York City`.

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.

- Add synonyms for the value. For example, you might add `NYC` and `The Big Apple` as synonyms for `New York City`.

A synonym can be any string up to 64 characters in length.

If you want to define a pattern for your assistant to look for in user input, such as a product order number or email address, define a pattern value instead. See [Adding entities that recognize patterns](#) for more details.

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

- If you want your assistant to recognize terms with syntax that is similar to the entity value and synonyms you specify, but without requiring an exact match, set the **Fuzzy Matching** switch to **On**.

For example, if you add `apple` as a value for a `@fruit` entity, and a user enters `apples` or `appel`, if fuzzy matching is enabled, your assistant will recognize the word as a `@fruit` mention. For more information, see [How fuzzy matching works](#).

- Click **Add value** and repeat the process to add more entity values.



**Tip:** If you are adding many values, one after another, press **Shift+Enter** to finish adding the current value, and keep focus in the value field so you can add the next value.

- After you add the entity values, click to finish creating the entity.

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

## Adding entities that recognize patterns

You can create an entity that looks for patterns in user input. For example, you can look for mentions of an email address by looking for occurrences of the pattern `{word}+@+{word}+.com`. Or, you might have product order numbers that follow a very specific format, such as `TWEX3433JKL`. You can create a pattern to look for strings with that syntax in the user utterance.

To add an entity that recognizes a pattern:

- Follow the standard procedure to create a dictionary-based entity, but select **Patterns** from the *Type* drop-down menu instead of *Synonyms*.

## [←](#) | @ContactInfo

### Entity name

Name your entity to match the category of values that it will detect.

@ContactInfo

### Value

email

### Synonyms

#### Synonyms

#### Patterns

Add value

Recommend synonyms



2. Add a regular expression that defines the pattern you want to look for.

- 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 Watson Assistant interface for creating a new entity. The entity name is '@ContactInfo'. Under the 'Value' section, 'localPhone' is listed with a pattern '(\\d{3})-(\\d{4})'. Below this, four additional patterns are defined for other value types: 'email' with pattern '\\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}', 'fullUSphone' with pattern '(\\d{3})-(\\d{3})-(\\d{4})', 'internationalPhone' with pattern '^((\\(?\\+?[0-9]\*\\)?)([0-9\_\\-\\ ]\*))\$', and 'website' with pattern '(https?:\\/\\/)?([\\da-zA-Z\\-.]+)\\.'. Each pattern has an 'Add patterns...' button next to it.

Follow these syntax rules:

- 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. You will see an error if you try to upload an unsupported pattern, either by using the API or from within the Watson Assistant user interface.

For example, for entity `ContactInfo`, the patterns for phone, email, and website values can be defined as follows:

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

3. Click **Add value** and repeat the process to add more entity values.

When you use pattern entities to find patterns in user input, you often need a way to store the part of the user input text that matches the pattern. To do so, you can use a context variable. For more information, see [Defining a context variable](#).

For example, your dialog might ask users for their email addresses. The dialog node condition will contain a condition similar to `@contactInfo:email`. You can use the following syntax in the dialog node's response section to define a context variable that captures and stores the user's email address text:

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

Saving a pattern

This syntax indicates that you want to find the part of the user input that matches the email pattern and save that subset of text into a context variable named `email`.

## 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 `@ContactInfo` has a pattern value named `fullUSPhone` that contains three captured groups:

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

Grouping can be helpful if, for example, you want your assistant to ask a user for a phone number, and then use only the area code of the provided number in a response.

To assign the user-entered area code as a context variable, use the following syntax in the dialog node's response section to capture the group match:

Variable	Value
area_code	<code>&lt;? @ContactInfo.groups[1] ?&gt;</code>

Saving a capture group

For more information about using capture groups in your dialog, see [Storing and recognizing entity pattern groups in input](#).

## How fuzzy matching works

Fuzzy matching is available for languages noted in the [Supported languages](#) topic.

Fuzzy matching has these components:

- *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:** The partial match component is supported only in English-language dialog skills.

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`.

**⚠ Important:** Interactions between the stemming and misspelling fuzzy matching features are not allowed. Specifically, if either an entity or the input is stemmed, misspelling fuzzy matching does not work. For example, assume that the entity is `@lending` and the input word is `pending`. During entity stemming, `@lending` produces `lend`. During input stemming, `pending` produces `pend`. In this case, `lend` does not match to `pend` because the entity and input were stemmed. This change applies to only the English language.

## Adding contextual entities

Annotation-based entities are those for which you annotate occurrences of the entity in sample sentences to teach your assistant about the context in which the entity is typically used.

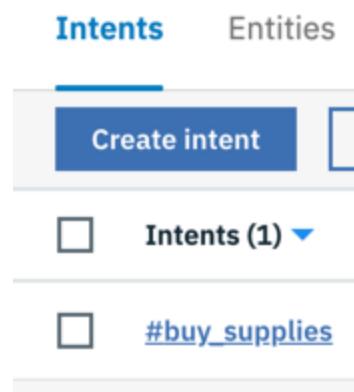
In order to train a contextual entity model, you can take advantage of your intent examples, which provide readily-available sentences to annotate.

 **Note:** This feature is generally available in English-language dialog skills and is available as a beta feature in French-language dialog skills. For more information about language support, see [Supported languages](#).

 **Note:** Using an intent's user examples to define contextual entities does not affect the classification of that intent. However, entity mentions that you label are also added to that entity as synonyms. And intent classification does use synonym mentions in intent user examples to establish a weak reference between an intent and an entity.

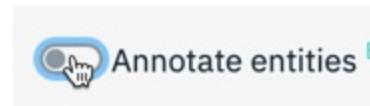
1. From your dialog skill, click the **Intents** tab.
2. Click an intent to open it.

For this example, the intent **#buy\_supplies** defines the order function for an online retailer.



The screenshot shows the 'Intents' tab selected in the top navigation bar. Below it, a list of intents is displayed. The intent '#buy\_supplies' is highlighted with a blue background and white text, indicating it is currently selected or being annotated.

3. Click **Annotate entities**, and then review the intent examples for potential entity mentions.



The screenshot shows the 'Annotate entities' button, which is a blue button with a white circular icon containing a hand cursor and the text 'Annotate entities'.

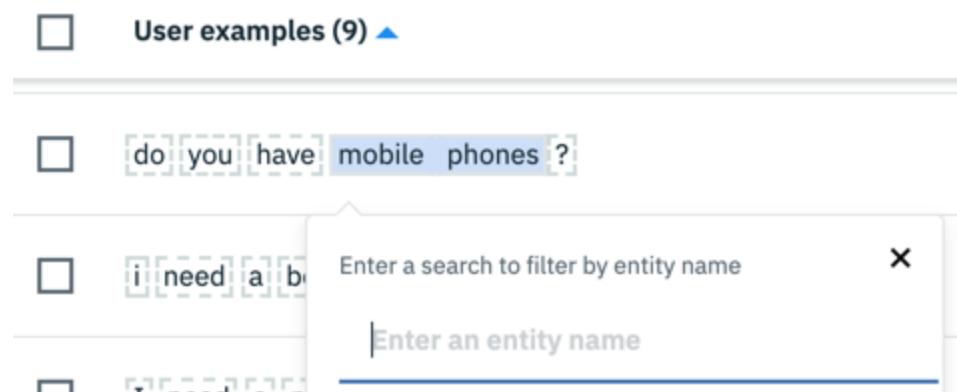
4. Click any word, words, or punctuation that is part of a single entity mention from the intent examples.

In this example, **mobile phones** is the entity mention.



The screenshot shows the annotated intent examples. The phrase 'mobile phones' is highlighted with a blue selection bar, indicating it has been identified as an entity mention.

A search box opens that you can use to search for the entity that the highlighted word or phrase is a mention of.



The screenshot shows a search interface for entity annotations. It includes a search bar with placeholder text 'Enter a search to filter by entity name' and a text input field with placeholder text 'Enter an entity name'.

5. Enter the entity name to search for. You do not need to include the starting '@' symbol.

Do one of the following things:

- o If the entity has any existing entity values, they are displayed for informational purposes only. You are adding the annotation to the entity, not to any specific entity value.
- o If you want to teach the model that the mention is synonymous with an existing entity value, add a colon ( ':') after the entity name to show a list of entity values. Choose an entity value from the list that is displayed. For example, **@product:device**.

## User examples (8) ▲

do you have mobile phones ?

i need a b

Enter a search to filter by entity name

product: device

I need a n

@product:device

i need print

@product:paper

6. Select the entity or entity and value to which you want to add the annotation.

In this example, **mobile phones** is being added as an annotation for the **@product** entity value and as a synonym for the **@product:device** entity value.



**Important:** Create *at least* 10 annotations for each contextual entity; more annotations are recommended for production use.

7. If none of the entities are appropriate, you can create a new entity by adding its name. Then choose the **{entity\_name}(create new entity)** option from the list.

## User examples (9) ▲

can i order a box of black ink pens

Do you have dry erasers in your Boston store ?

do you have mobile phones ?

Enter a search to filter by entity name

@location

i need a box of plain white co

@location (create new entity)

8. Repeat this process for each entity mention that you want to annotate.



**Important:** Be sure to annotate every mention of an entity type that occurs in any user examples that you edit. See [What you don't annotate matters](#) for more details.

9. Now, click the annotation you just created. A box is displayed that says, **Go to: {entity-name}**. Clicking that link takes you directly to the entity.

## User examples (9) ▲

do you have mobile phones ?

i need a b

Enter a search to filter by entity name

product device

I need a n

[Remove @product](#)

i need print

[Go to @product](#)

The annotation is added to the entity you associated it with, and the system begins to train itself on the new data.



**Important:** The term you annotated is added to the entity as a new dictionary value. If you associated the annotated term with an existing

entity value, then the term is added as a synonym of that entity value instead of as an independent entity value.

The screenshot shows the Entity configuration page for the entity '@product'. At the top, there is a back arrow and the entity name '@product'. To the right, there is a 'Last updated' timestamp. Below the header, there is a section for 'Entity name' with a placeholder 'Name your entity to match the category of values that it will detect.' followed by the value '@product'. There are two input fields: 'Value' (placeholder 'Type value here, e.g. Checking') and 'Synonyms' (placeholder 'Type synonym here, e.g. Deposit'). Below these fields are two buttons: 'Add value' and 'Recommend synonyms'. Under the 'Annotation' tab, there are two sections: 'Dictionary (2)' and 'Annotation (1) BETA'. The 'Annotation (1) BETA' section contains one entry: 'Values (2) ▲' with a checkbox next to it, followed by 'device' and its synonyms 'mobile phones, ipad, surface'. There are also 'Type' and 'Intent' columns for this entry.

10. To see all of the mentions you annotated for a particular entity, from the entity's configuration page, click the **Annotation** tab.

The screenshot shows the Annotation view for the entity '@product'. It displays the 'Annotation (1) BETA' tab, which lists one user example: 'do you have mobile phones?' with the intent '#buy\_supplies'. There is also a 'User Examples (1)' section with a checkbox next to it.

**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 then click **Delete**.

**Tutorial:** To walk through a tutorial that shows you how to define contextual entities before you add your own, go to [Tutorial: Defining contextual entities](#).

**Note:** The contextual entities tutorial shows a slightly older version of the user interface. But the concepts it covers are the same, so it is still a worthwhile exercise.

## What you don't annotate matters

If you have an intent example with an annotation, and another word in that example matches the value or a synonym of the same entity, but the value is *not* annotated, that omission has impact. The model also learns from the context of the term you did not annotate. Therefore, if you label one term as a mention of an entity in a user example, be sure to label any other applicable mentions also.

1. The `#Customer_Care_Appointments` intent includes two intent examples with the word `visit`.

**User example**

Add unique examples of what the user might say. (Pro tip: Add at least 5 unique examples to help Watson understand)

Type a user example here, e.g. I want to pay my credit card bill

**Add example**

Show recommendations

**User examples (20) ▲**

I would like to make an appointment to visit the nearest store to my location.

Is it possible to set a date?

Make an appointment

Set up an appt

Store appointment

Want to change my visit

What time can I meet the staff?

2. In the second occurrence of the word, 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*.

Want to change my visit

What time can I r

When can I meet



3. In the first occurrence, the word **visit** is being used as a verb. It has a different meaning from 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. The model learns from the fact that the word **visit** in the same example is not annotated.

The screenshot shows a list of intents on the left and a search modal on the right. The intents include: "I would like to make an appointment to visit the nearest store to my location.", "Is it possible to set a [redacted]", "Make an appointment", "Set up an appt", "Store appointment", and "Want to change my visit". The search modal has a search bar with placeholder text "Enter a search to filter by entity name" and a close button. Below the search bar is a list of suggestions: "@fruit (add new value)", "@meeting (add new value)" (which is highlighted with a blue background and a cursor icon), and "@product (add new value)".

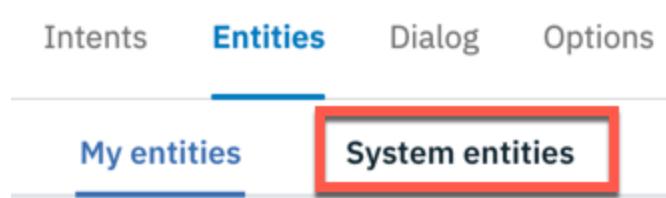
## Enabling system entities

Watson Assistant 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 skill 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 page, click **System entities**.



2. Browse through the list of system entities to choose the ones that are useful for your application.
  - To see more information about a system entity, including examples of matching input, click the entity in the list.
  - For details about the available system entities, see [System entities](#).
3. Set the switch for each system entity that you want to use to **On**.

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

## Entity limits

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

Plan	Entities per skill	Entity values per skill	Entity synonyms per skill
Enterprise	1,000	100,000	100,000
Premium (legacy)	1,000	100,000	100,000
Plus	1,000	100,000	100,000
Lite, Trial	100	100,000	100,000

[Plan details](#)

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

Plan	Contextual entities and annotations
Enterprise	150 contextual entities with 3000 annotations

Premium (legacy)	150 contextual entities with 3000 annotations
Plus	100 contextual entities with 2000 annotations
Lite, Trial	10 contextual entities with 1000 annotations

Plan details continued

## Editing entities

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

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

## Searching entities

Use the Search feature to find entity names, values, and synonyms.

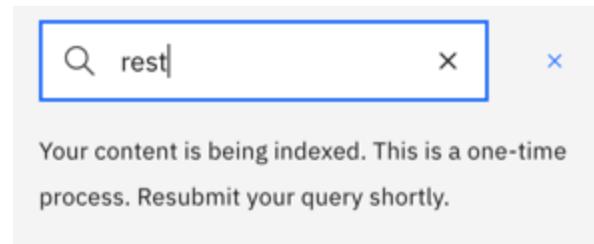
- From the **Entities** page, click the Search icon.



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

- Enter a search term or phrase.

The first time you search for something, you might get a message that says the skill is being indexed. If so, wait a minute, and then resubmit the search term.



Entities containing your search term, with corresponding examples, are shown.

Showing 1 entity containing:  
'rest'

- ^ @amenity  
2 matches
  - value [restaurant](#)
  - value [restroom](#)

## Downloading entities

You can download a number of entities to a CSV file, so you can then upload and reuse them in another Watson Assistant application.

- Pattern information is included in the downloaded CSV file. Any string wrapped with `/` will be considered a pattern (as opposed to a synonym).
- Annotations associated with contextual entities are not downloaded. You must download the entire dialog skill to capture both the entity value and any associated annotations.

- Go to the **Entities** page

- To download all entities, meaning the entities that are listed on this and any additional pages, do not select any individual entities. Instead, click the *Download* icon .
- To download the entities that are listed on the current page only, select the checkbox in the header. This action selects all of the entities on the page.

current page. Click the *Download* icon .

- To download one or more specific entities, select the entities that you want to download, and then click the *Download* icon .

2. Specify the name and location in which to store the CSV file that is generated, and then click **Save**.

## Uploading entities

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



**Note:** Entity annotations are not included in the upload of an entity CSV file. You must upload the entire dialog skill to retain the associated annotations for a contextual entity in that skill. If you download and upload entities only, then any contextual entities that you downloaded are treated as dictionary-based entities after you upload them.

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
```

Uploading 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_\\-\\()]*$/
ContactInfo,email,/\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.\\[A-Za-z]{2,}\\b/
ContactInfo,website,/\\(https?:\\//)?([\\da-zA-Z\\.-]+)\\.(\\[a-zA-Z\\]{2,6})([\\w\\.-]*)*/?$/
```



**Tip:** 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 uploading them separately. Open your dialog skill and then click the **Entities** tab.

2. Click the upload icon .

3. Drag a file, or browse to select a file from your computer, and then click **Upload**.

The file is validated and uploaded, and the system begins to train itself on the new data.

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

## Deleting entities

You can select a number of entities for deletion.



**Important:** When you delete an entity, you remove any values, synonyms, patterns, or annotations that are associated with the entity. This data 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. Go to the **Entities** page.

- To delete all entities, meaning the entities listed on this and any additional pages, do not select any individual entities. Instead, click the *Delete all entities* icon .
- To delete the entities that are listed on the current page only, select the checkbox in the header. This action selects all of the entities that are listed on the current page. Click **Delete**.

- To delete one or more specific entities, select the entities that you want to delete, and then click **Delete**.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Managing workflow with versions](#).

## Managing workflow with versions

Versions help you manage the workflow of a dialog skill development project.

Create a skill version to capture a snapshot of the training data (intents and entities) and dialog in the skill at key points during the development process. Being able to save an in-progress skill at a specific point in time is especially useful as you start to fine tune your assistant. You often need to make a change and see the impact of the change in real time before you can determine whether or not the change improves or lessens the effectiveness of the assistant. Based on your findings from a test environment deployment, you can make an informed decision about whether to deploy a given change to an assistant that is deployed in a production environment.



**Note:** If you have a free (Lite) plan, you cannot create skill versions.

To learn more about how versions can improve the workflow you use to build an assistant, [read this blog post](#).

### Creating a version

You can edit only one version of the dialog skill at a time. The in-progress version is called the *development* version.

When you save a version, any skill settings that you applied to the development version are saved also.

To create a dialog skill version, follow these steps:

1. From the header of the skill, click **Save new version**, and then describe the current state of the skill.

Adding a good description will help you to distinguish multiple versions from one another later.



**Tip:** Add the date you deploy the version to its description to make it easier to filter logs by version from the Analytics page later. For more information, see [Picking a data source](#).

2. Click **Save**.

A snapshot is taken of the current skill and saved as a new version. You remain in the development version of the skill. Any changes you make continue to be applied to the development version, not the version you saved. To access the version you saved, go to the **Versions** page.

If you have trouble creating the version, check that your skill does not have any entities with large numbers of values (such as 10,000 or more synonyms for a single entity). If it does, try to break the entity into many, more categorized entities, or consider using a contextual entity instead.

### Deploying a skill version

1. From the skill menu, click **Versions**.
2. Click the icon from the version you want to deploy, and then choose **Assign**.

A list of assistants to which you can link this version is displayed. The list is limited to those assistants that don't have any skills associated with them, or that are associated with a different version of this skill.

3. Click the checkbox for one or more of the assistants, and then click **Assign**.



**Important:** Keep track of when this version is deployed to an assistant and for how long. It is likely that you will want to analyze user conversations that occur between users and this specific version of the skill. You can get this information from the **Analytics** page. However, when you pick a data source, versions are not listed. You must choose the name of the assistant to which you deployed this version. You can then filter the metrics data to show only those conversations that occurred between the start and end dates of the time frame during which this skill version was deployed to the assistant.

### Skill version limits

The number of versions you can create for a single skill depends on your Watson Assistant plan.

Service plan	Versions per skill
Enterprise	50

Premium (legacy)	50
Plus	10
Trial	10
Lite	0

#### Service plan details

## Swapping skills

If you find that an earlier version of the skill did a better job of recognizing and addressing customer needs than a later version, you can swap the skill that is linked to the assistant to use the earlier version instead.

Follow the same steps you use to [deploy a skill version](#) to change the version that is linked to an assistant.

## Accessing a saved version

The only way to view a saved version is to overwrite the in-progress development version of the skill with the saved version. (But not before you have saved any work you have done in the current development version.)

You cannot edit a saved version. To achieve the same goal, you can use a saved version as the basis for a new version in which you incorporate any changes you want to make. To start development work from a saved version, overwrite the in-progress development version of the skill with the saved version.

1. Save any changes you made to the skill since the last time you created a version.



**Important:** Save a version of the skill now. Otherwise, your work will be lost when you follow these steps.

2. From the version you want to edit, click the **Skill actions** icon, and then choose **Revert to this version** and confirm the action.

The page refreshes to revert to the state that the skill was in when the version was created.

If you want to save any changes that you make to this version, you must save the skill as a new version. You cannot apply changes to an already-saved version.



**Important:** When you open a skill by clicking the skill tile from the assistant page, the development version of the skill is displayed. Even if you associated a later version with the assistant, when you access the skill, its development version opens.

## Downloading a skill version

You can download a dialog skill version in JSON format. You might want to download a skill version if you want to use a specific version of a dialog skill in a different instance of the Watson Assistant service, for example. You can download the version from one instance and import it to another instance as a new dialog skill.

To download a dialog skill version, complete the following steps:

1. From the skill menu, click **Versions**.
2. Click the icon from the version you want to download, and then choose **Export**.
3. Specify a name for the JSON file and where to save it, and then click **Save**.

For more information about how to replace a skill, see [Overwriting a skill](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [IBM Watson® Discovery search integration setup](#).

## Use a search skill to embed existing help content Plus

Put your subject matter expertise to work by adding a search skill. The search skill gives your assistant access to corporate data collections that it can mine for answers.

When a *search skill* is added, your assistant can route complex customer inquiries to the IBM Watson® Discovery service. Discovery treats the user input as a search query. It finds information that is relevant to the query from an external data source and returns it to the assistant.



**Note:** This feature is available only to paid plan users.

Add a search skill to your assistant to prevent the assistant from having to say things like, **I'm sorry. I can't help you with that**. Instead, the assistant can query existing company documents or data to see whether any useful information can be found and shared with the customer.

The screenshot shows the Watson Assistant interface in 'Preview' mode. A user message bubble contains the question "how many bitcoins can be created?". Below it, a response message bubble begins with "I searched my knowledge base and found this information which might be useful:". The first result is "21 million". The text continues: "Bitcoin is unique in that only **21 million** bitcoins will ever be created. However, this will never be a limitation because transactions can be denominated in smaller sub-units of a bitcoin, such as...". There is a "Show more" link and a snippet of additional text starting with "Won't the finite amount of bitcoins be a limit...". At the bottom of the response bubble, there is a "Show more results" link. The input field at the bottom is labeled "Type something..." with a right-pointing arrow.



**Tip:** To show the exact answer highlighted in bold font, enable the *Emphasize the answer* feature that is available with Discovery v2 instances.

Watch a 4-minute video that provides an overview of the search skill:



**View video:** [Search Integration: Watson Assistant](#)

To learn more about how search skill can benefit your business, [read this blog post](#).

Find out how to keep your assistant data current by reading the blog post, [COVID-19: Are Your Virtual Assistant's Answers Up-To-Date?](#).

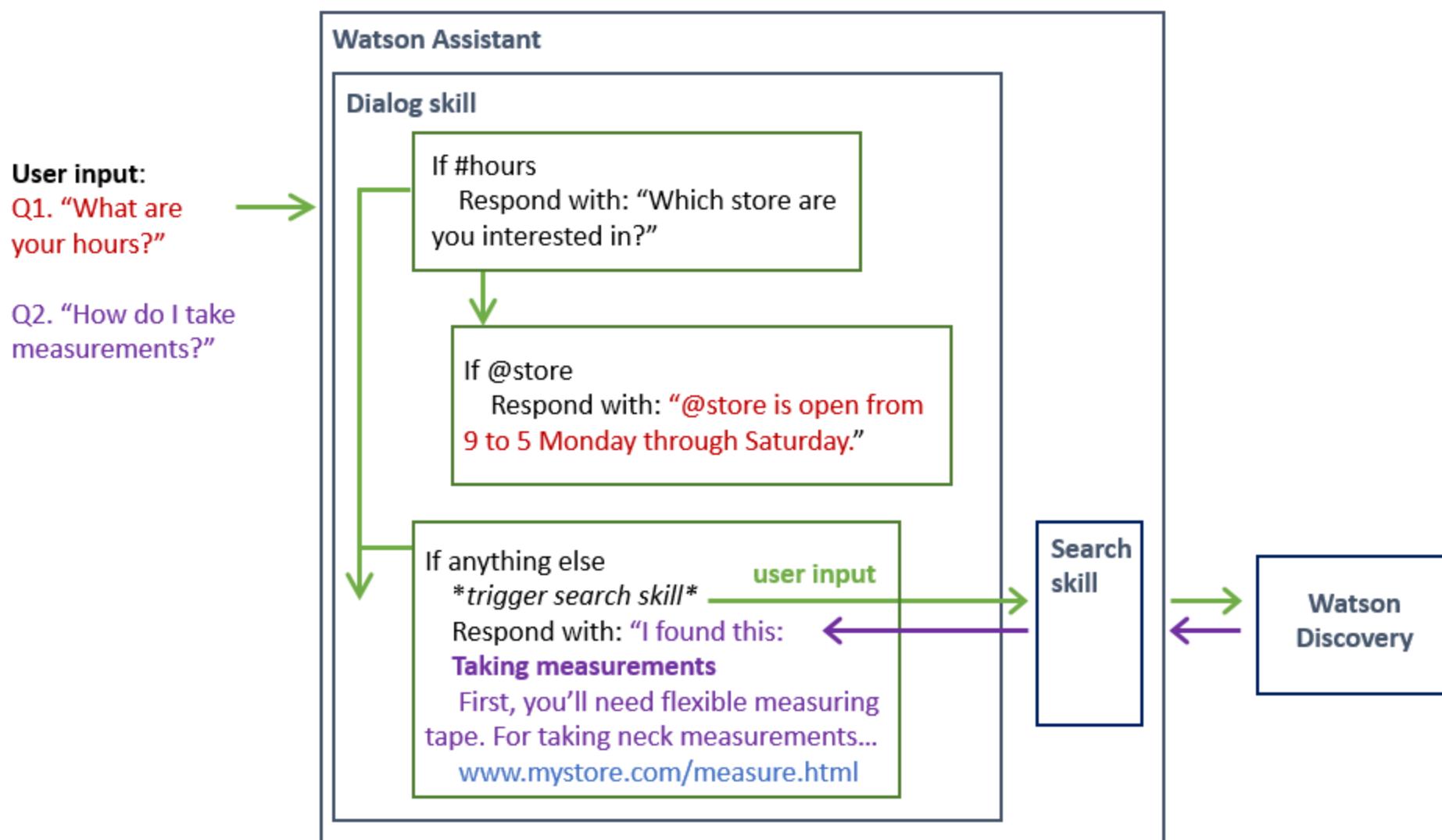
## How it works

The search skill searches for information from a data collection that you create by using the Discovery service.

Discovery is a service that crawls, converts, and normalizes your unstructured data. The product applies data analysis and cognitive intuition to enrich your data such that you can more easily find and retrieve meaningful information from it later. To read more about Discovery, see the [product documentation](#).

Typically, the type of data collection you add to Discovery and access from your assistant contains information that is owned by your company. This proprietary information can include FAQs, sales collateral, technical manuals, or papers written by subject matter experts. Mine this dense collection of proprietary information to find answers to customer questions quickly.

The following diagram illustrates how user input is processed when both a dialog skill and a search skill are added to an assistant.



For a step-by-step tutorial, read [Help your chatbot get answers from existing content](#) in the [Watson Discovery documentation](#).

## Before you begin

1. Before you begin, you must set up a Discovery instance.

You can do this at no cost by using a Plus plan, which offers a free 30-day trial. However, to create a Plus plan instance of the service, you must have a paid account (where you provide credit card details).

2. Create a Plus plan Discovery service instance.

Go to the [Discovery](#) page in the IBM Cloud catalog and create a Plus plan service instance.

**Important:** If you decide not to continue using the Plus plan and don't want to pay for it, delete the Plus plan service instance before the 30-day trial period ends.

## Create the search skill

1. From the assistant where you want to add the skill, click **Add search skill**.



**Note:** You can only add a search skill if you are a user with a paid plan.

2. Take one of the following actions:

- To create a new search skill, stay on the *Create skill* tab.
- If you have created a search skill already, the *Add existing skill* tab is displayed, and you can click to add an existing skill.

3. Specify the details for the new skill:

- **Name:** A name no more than 64 characters in length. A name is required.
- **Description:** An optional description no more than 128 characters in length.

Next, you will connect to a Discovery service instance.

4. Click **Continue**.

## Connect to an existing Watson Discovery instance

1. Choose the Discovery instance that you want to connect to.

 **Note:** If you see a warning that some of your Discovery service instances do not have credentials set, it means that you can access at least one instance that you never opened from the IBM Cloud dashboard directly yourself. You must access a service instance for credentials to be created for it, and credentials must exist before Watson Assistant can establish a connection to the Discovery service instance on your behalf. If you think a Discovery service instance is missing from the list, open the instance from the IBM Cloud® dashboard directly to generate credentials for it.

2. Indicate the collection to use, by doing one of the following things:

- Choose an existing project.

You can click the *Open Discovery* icon to review the configuration of a project before you decide which one to use.

Go to [Configure the search](#) and follow the remaining steps.

- If you do not have a project or do not want to use any of the projects that are listed, click **Create a new project** to add one. Follow the steps in [Create a project](#).

 **Note:** The **Create a new project** button is not displayed if you have reached the limits based on your Discovery service plan. See [Discovery pricing plans](#) for plan limit details.

## Create a project

1. On the **OK, where is your data?** page, select your data source, then click **Next**. Example choices include Salesforce, SharePoint, Box, IBM Cloud Object Storage, web crawl, and upload data.
2. On the **Let's create a collection for your data** pages, enter the information on how to connect to your data and configure your collection. The information you need to enter is different depending on the data source. For example, you need to enter your authentication credentials for services such as Salesforce, Sharepoint, and Box. For web crawl, you specify the website where your existing information resides.
3. Click **Finish**. Give Discovery a few minutes to start creating documents. You can use the **Manage collections** page within the project to see the progress.

After the digestion process is completed, a summary page is displayed in Discovery, which is hosted in a separate web browser tab.

4. Wait for the collection to be fully ingested, then click **Back to Watson Assistant**.

## Configure the search

1. On the Watson Assistant search skill page, ensure the Discovery project you want to use is selected, then click **Next**.
2. In the **Configure result content** section, review the Discovery fields and examples that are used in the search results shown to your customers. You can accept the defaults, or customize them as you want.

The appropriate collection fields to extract data from vary depending on your collection's data source and how the data source was enriched. After you choose a data collection type, the collection field values are prepopulated with source fields that are considered most likely to contain useful information given the collection's data source type. However, you know your data better than anyone. You can change the source fields to ones that contain the best information to meet your needs.

To learn more about the structure of the documents in your collection, including the names of fields that contain information you might want to extract, open the collection in Discovery, and then use the **Identity fields** and **Manage fields** tabs.

Each search result can consist of the following sections:

- **Title:** Search result title. Use the title, name, or similar type of field from the collection as the search result title.  
You must select something for the title or no search result response is displayed in the Facebook and Slack integrations.
- **Body:** Search result description. Use an abstract, summary, or highlight field from the collection as the search result body.  
You must select something for the body or no search result response is displayed in the Facebook and Slack integrations.
- **URL:** This field can be populated with any footer content that you want to include at the end of the search result.

For example, you might want to include a hypertext link to the original data object in its native data source. Most online data sources provide self-referencing public URLs for objects in the store to support direct access. If you add a URL, it must be valid and reachable. If it is not, the Slack integration will not include the URL in its response and the Facebook integration will not return any response.

The Facebook and Slack integrations can successfully display the search result response when the URL field is empty.



**Important:** You must use a field for at least one of the search results.

If no options are available from the drop-down fields, give Discovery more time to finish creating the collection. After waiting, if the collection is not created, then your collection might not contain any documents or might have ingestion errors that you need to address first.

The screenshot shows the 'Search Skill/Personal banking' configuration page. At the top, there are 'Cancel' and 'Create' buttons. To the right, it says 'Discovery instance: Discovery-dev' and 'Collection: Web Crawl'. Below this, the 'Configure result content' section is shown, with 'Title', 'Body', and 'URL' fields mapped to specific fields. Under 'Message', there are tabs for 'No results found' and 'Connectivity issue'. A 'Text to display' section contains a message about searching the knowledge base. On the right, a 'Preview' window shows a list of search results related to the Internal Revenue Service.

As you add field mappings, a preview of the search result is displayed with information from the corresponding fields of your data collection. This preview shows you what gets included in the search result response that is returned to users.

To get help with configuring the search, see [Troubleshooting](#).

3. Use the **Message**, **No results found** and **Connectivity issue** tabs to customize different messages to share with users based on the successfulness of the search.

Tab	Scenario	Example message
Message	Search results are returned	I found this information that might be helpful:
No results found	No search results are found	I searched my knowledge base for information that might address your query, but did not find anything useful to share.
Connectivity issue	I was unable to complete the search for some reason	I might have information that could help address your query, but am unable to search my knowledge base at the moment.

#### Search result messages

4. Choose whether to enable **Emphasize the answer**.



**Note:** This option is available only if your Discovery instance uses the v2 Discovery API.

When you enable this feature, the sentence that is determined by Discovery to be the exact answer to the customer's question is highlighted in the block of text that is displayed to the customer as the search result.

5. In the **Adjust result quantity** section, specify the number of results to return.

The top three results are returned automatically. You can choose to show fewer or more (up to 10) results in the response.

By default, customers can choose to see more results. If you don't want to give customers this choice, clear the **Include link for customers to view up to 10 results** checkbox.

6. In the **Set result selectivity** section, decide whether to be more selective with the answers that are returned. By increasing result selectivity, Search returns fewer but more accurate results. In most cases, Search is accurate enough that the default setting (off) is sufficient.

7. Click **Preview** to open the Preview pane for testing. Enter a test message to see the results that are returned when your configuration choices are applied to the search. Make adjustments as necessary.

8. Click **Create**.

If you want to change the configuration of the search result card later, open the search skill again, and make edits. You do not need to save changes as you make them; they are automatically applied. When you are happy with the search results, click **Save** to finish configuring the search skill.



**Important:** If you decide you want to connect to a different Discovery service instance or data collection, then create a new search skill and configure it to connect to the other instance. You *cannot* change the service instance or data collection details for a search skill after you create it.

## Troubleshooting

Review this information for help with performing common tasks.

- **Creating a Web crawl data collection :** Things to know when you create a web crawl data source:
  - To increase the number of documents that are available to the data collection, click add a URL group where you can list the URLs for pages that you want to crawl but that are not linked to from the initial seed URL.
  - To decrease the number of documents that are available to the data collection, specify a subdomain of the base URL. Or, in the web crawl settings, limit the number of hops that Watson can make from the original page. You can specify subdomains to explicitly exclude from the crawl also.
  - If no documents are listed after a few minutes and a page refresh, then make sure that the content you want to ingest is available from the URL's page source. Some web page content is dynamically generated and therefore cannot be crawled.
- **Configuring search results for uploaded documents :** If you are using a collection of uploaded documents and cannot get the correct search results or the results are not concise enough, consider using *Smart Document Understanding* when you create the data collection.

This feature enables you to annotate documents based on text formatting. For example, you can teach Discovery that any text in 28-point bold font is a document title. If you apply this information to the collection when you ingest it, you can later use the *title* field as the source for the title section of your search result.

You can also use Smart Document Understanding to split up large documents into segments that are easier to search. For more information, see the the [Using Smart Document Understanding](#) topic in the Discovery documentation.

- **Improve search results:** If you don't like the results you are seeing, call the search skill from a dialog node, and specify filter details.

From a dialog node search skill response, you can specify a full Discovery query syntax filter to help narrow the results.

For example, you can define a filter that filters out any documents in the data collection that do not mention an intent in the document title or some other metadata field. Or the filter can filter out documents that do not identify an entity as a known entity in the data collection's metadata or that don't mention the entity anywhere in the full text of the document. For details about how to add a search skill response type, see [Adding a Search skill response type](#).

For more tips about improving results, read the [Improve your natural language query results from Watson Discovery](#) blog post.

- **My response text is surrounded by brackets :** If you notice that your response text is surrounded by brackets and quotation marks ( `[ "My response text" ]` ) when you test it from the Preview, for example, you might need to change the source field that you're using in the configuration. The unexpected formatting indicates that the value is stored in the source document as an array. Any field that you extract text from must contain a value with a String data type, not an Array data type. When the chat integration shows a response that is extracted from a field that stores the data as an array, it does a straight conversion of the array value into a string, which produces a response that includes the array syntax.

For example, maybe the field in the source document contains an array with a single text value as its only array element:

```
"title": ["a single array element"]
```

The array value is converted by the Watson Assistant into this string value:

```
"title": "[\"a single array element\"]"
```

As a result, the string is returned in this format in the chat; the surrounding brackets and quotation marks are displayed:

```
["a single array element"]
```

If you see this happening, consider choosing a different collection field from which to extract search results.



**Note:** The Discovery document `highlight` field stores values in an array.

## Next steps

After you create the skill, it appears as a tile on the Skills page.

The search skill cannot interact with customers until it is added to an assistant and the assistant is deployed. See [Creating assistants](#).

## Adding the skill to an assistant

Open the assistant tile and add the skill to the assistant from there. You cannot choose the assistant that will use the skill from within the skill configuration page.

One search skill can be used by more than one assistant.

1. From the Assistants page, click to open the tile for the assistant to which you want to add the skill.
2. Click **Add search skill**.
3. Click **Add existing skill**, then click the skill that you want to add from the available skills that are displayed.

## Search triggers

The search skill is triggered in the following ways:

- **From a specific dialog node or action step**: This approach is useful if you want to narrow down a user query before you trigger a search.

For example, the conversational flow might collect information about the type of device a customer wants to buy. When you know the device model, you can then send a model keyword in the query that is submitted to the search skill to get better results.

Trigger the search only at a specific point in a conversation in one of the following ways:

- Dialog skill: Add a *search skill* response type to a dialog node. When the node is processed, your assistant retrieves a passage from an external data source and returns it as the response to a particular question. This type of search occurs only when the individual dialog node is processed. For more information, see [Adding a search skill response type](#).
- Actions skill: In the *And then* field of the step where you want the search to be triggered, choose **Search for the answer**.

- **From the dialog skill's Anything else node**: If the assistant has a dialog skill and a search skill, any user input triggers the dialog skill first. The dialog addresses user input that it has a high confidence it can answer correctly. Any queries that would normally trigger the **anything\_else** node in the dialog tree are sent to the search skill instead.

For example, instead of showing a standard message, such as **I don't know how to help you with that.** the assistant can say, **Maybe this information can help:**. The assistant passes the user input as the query to your search skill, and returns the search results as the response.



**Note:** You can prevent the search from being triggered from the **anything\_else** node by following the steps in [Disabling search](#).

- **When only a search skill is used**: If only a search skill is linked to an assistant, and no conversational skill is configured, then a search query is sent to the Discovery service when any user input is received from one of the assistant's integration channels.

## Test the search skill

After you configure the search, you can send test queries to see the search results that get returned from Discovery by using the Preview pane of the search skill.

To test the full experience that customers will have when they ask questions that are either answered by the dialog or trigger a search, use the **Preview** button for your assistant.



**Important:** You cannot test the full end-to-end user experience from the dialog "Try it out" pane. The search skill is configured separately and attached to an assistant. The dialog skill has no way of knowing the details of the search, and therefore cannot show search results in its "Try it out" pane.

Configure at least one integration channel to test the search skill. In the channel, enter queries that trigger the search. If you initiate any type of search from your dialog, test the dialog to ensure that the search is triggered as expected. If you are not using search response types, test that a search is triggered only when no existing dialog nodes can address the user input. And any time a search is triggered, ensure that it returns meaningful results.

## Sending more requests to the search skill

If you want the dialog skill to respond less often and to send more queries to the search skill instead, you can configure the dialog to do so.

You must add both a dialog skill and search skill to your assistant for this approach to work.

Follow this procedure to make it less likely that the dialog will respond by resetting the confidence level threshold from the default setting of 0.2 to 0.5. Changing the confidence level threshold to 0.5 instructs your assistant to not respond with an answer from the dialog unless the assistant is more than 50% confident that the dialog can understand the user's intent and can address it.

1. From the *Dialog* page of your dialog skill, make sure that the last node in the dialog tree has an `anything_else` condition.

Whenever this node is processed, the search skill is triggered.

2. Add a folder to the dialog. Position the folder before the first dialog node that you want to de-emphasize. Add the following condition to the folder:

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

This condition is applied to all of the nodes in the folder. The condition tells your assistant to process the nodes in the folder only if your assistant is at least 50% confident that it knows the user's intent.

3. Move any dialog nodes that you do not want your assistant to process often into the folder.

After changing the dialog, test the assistant to make sure the search skill is triggered as often as you want it to be.

An alternative approach is to teach the dialog about topics to ignore. To do so, you can add utterances that you want the assistant to send to the search skill immediately as test utterances in the dialog skill's "Try it out" pane. You can then select the **Mark as irrelevant** option within the "Try it out" pane to teach the dialog not to respond to this utterance or others like it. For more information, see [Teaching your assistant about topics to ignore](#).

## Disabling search

---

You can disable the search skill from being triggered.

You might want to do so temporarily, while you are setting up the integration. Or you might want to only ever trigger a search for specific user queries that you can identify within the dialog, and use a search skill response type to answer.

To prevent the search skill from being triggered, complete the following steps:

1. From the **Assistants** page, click the menu for your assistant, and then choose **Settings**.
2. Open the *Search skill* page, and then set the switch to **Disabled**.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

# Working with skills

Perform common tasks, such as renaming or deleting a skill.

## Renaming a skill

You can change the name of a skill and its associated description after you create the skill.

To rename a skill, follow these steps:

1. From the Skills page, find the skill that you want to rename.
2. Click the  icon, and then choose **Rename**.
3. Edit the name, and then click **Save**.

## Duplicating a skill

You can duplicate a skill to make a copy of it.

To duplicate a skill, follow these steps:

1. From the Skills page, find the skill that you want to duplicate.
2. Click the  icon, and then choose **Duplicate**. The copied skill has the word **copy** added to the end of its name.

## Deleting a skill

You can delete any skill that you can access, unless it is being used by an assistant. If it is in use, you must remove it from the assistant that is using it before you can delete it.



**Tip:** Be sure to check with anyone else who might be using the skill before you delete it.

To delete a skill, complete the following steps:

1. Find out whether the skill is being used by any assistants. From the Skills page, find the tile for the skill that you want to delete. The **Assistants** field lists the assistants that currently use the skill.
2. If the skill you want to delete is associated with an assistant, then remove it from the assistant by completing the following steps:
  - o Check with the owner of the assistant that is using the skill before you remove the skill from it.
  - o Open the Assistants page, and then click to open the assistant tile.
  - o Find the tile for the skill that you want to delete. Click the  icon, and then choose **Remove**.
  - o Repeat the previous steps for any other assistants that use the skill.
  - o Return to the Skills page and find the tile for the skill that you want to delete.
3. Click the  icon, and then choose **Delete**. Confirm the deletion.

## Downloading a skill

You can download a dialog or actions skill in JSON format. You might want to download a skill if you want to use the same skill in a different instance of the Watson Assistant service. You can download a dialog skill from one instance and upload it to another instance as a new skill, for example.

You cannot download a search skill.

To download a skill, complete the following steps:

1. Find the skill tile on the Skills page or on the configuration page of an assistant that uses the skill.
2. Click the  icon, and then choose **Download**.
3. Specify a name for the JSON file and where to save it, and then click **Save**.

For dialog skills only:

- You can download a dialog skill by using the API also. Include the `export=true` parameter with the request. See the [API reference](#) for more details.
- For information about how to download a specific dialog skill version, see [Downloading a skill version](#).

## Overwriting a skill

---

To overwrite or replace an existing skill, upload the new version of the skill as a JSON file into the existing skill.

You can overwrite a dialog or actions skill. You cannot overwrite a search skill.

To overwrite a skill, complete the following steps:

1. Click the **Skills** icon  to open the Skills page.
2. Find the skill that you want to replace.
3. Click the  icon, and then choose **Upload**.
4. Drag a file or click **Drag and drop file here or click to select a file** and select the JSON file you want to overwrite your skill with.

 **Important:** The uploaded JSON file must use UTF-8 encoding, without byte order mark (BOM) encoding. The JSON cannot contain tabs, newlines, or carriage returns.

 **Tip:** For dialog skills only: The maximum size for a skill JSON file is 10 MB. If you need to upload a larger skill, consider using the REST API. For more information, see the [API Reference](#).

5. Click **Upload and overwrite**.

If you have trouble uploading a dialog skill, see [Troubleshooting skill upload issues](#).

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Testing your assistant

Use the assistant preview to test your assistant.

The assistant preview renders your assistant as a web chat embedded in a preview web page. You can test the skills that you added to the assistant by entering text into the chat window. You can copy and paste the web page URL into a web browser and share the page with others to enlist help in testing and getting feedback about the assistant.

Unlike when you test a skill using the "Try it out" pane, any API calls that result from your interactions with the assistant hosted by the assistant preview might incur charges.

## Using the assistant preview to test your assistant

To test the assistant from a web-hosted chat widget, complete the following steps:

1. From the Assistants page, click to open the assistant tile that you want to test.
2. Click **Preview**.



**Note:** For environments where private endpoints are in use, keep in mind that the preview sends traffic over the internet. For more information, see [Private network endpoints](#).

3. In the "Assistant preview" pane, submit test utterances to see how the assistant responds.
4. You can click the **Restart conversation** button to start a conversation over.

This action is useful if you want to test different routes through a single conversational flow. Or if you set or change context variable values in a dialog skill and want to see how specifying different values affects the conversation.

The conversation is restarted after the assistant's inactivity period elapses. The inactivity period can range from 5 minutes to 7 days depending on what you configure for your service plan type. After the inactivity time frame passes, any context variable values that were set during the conversation are set to null or back to their default values. For more information, see [Changing the inactivity timeout setting](#).

5. To test the assistant from a separate web page, copy and paste the *Share this link* URL into a web browser.

A simple IBM-branded web page is displayed that contains a chat window where you can interact with your assistant. You can share the page with others by sending them the *Share this link* URL. If necessary, you can click the toggle on the Preview page to disable the link.

**Important:** Previewing your assistant within the Watson Assistant user interface does not incur any charges. If you use the *Share this link* URL to preview the assistant from elsewhere, you might be charged for messages that you submit, according to the terms of your plan. You can review metrics about the test user conversations from the Analytics page.

6. Click **X** to close the Preview page.

## Dialog considerations

The rich responses that you add to a dialog are displayed in the web-hosted chat widget as expected, with the following exceptions:

- **Connect to human agent**: This response type is ignored.
- **Pause**: This response type pauses the assistant's activity in the chat widget. However, activity does not resume after the pause until another response is triggered. Whenever you include a **pause** response type, add another, different response type, such as **text**, after it.

See [Rich responses](#) for more information about response types.

**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Working with your assistant

Learn how to find and open assistants and skills that you created or can access.

Assistants and skills are created within a Watson Assistant service instance. To continue working with a skill or assistant, open the service instance that contains the skill or assistant. If you can't remember the service instance name, you can switch between instances from within the Watson Assistant user interface.

1. Go to the [IBM Cloud Resource list](#).

2. Log in.

A list of the service instances that you own or were given access to is displayed.

3. Click a service instance to open it.
4. Click **Launch Watson Assistant** from the service instance details page to open the product in a new browser tab or window.
5. Click the appropriate icon from the navigation pane to see a list of your assistants or skills.

-  Assistants
-  Skills

If you do not see the skill or assistant you are looking for, you can look for it in a different service instance.

## Switching the service instance

To switch to a different Watson Assistant service instance, complete the following steps:



1. From the header of any page in the current instance, click the User icon.
2. Choose **Switch instance** from the menu.
3. A list of the available service instances is displayed. Click a different service instance to open it.

## Getting API information

Get information, such as the assistant ID and skill ID and credentials that you need before you can use the API.

- For an assistant, click the overflow menu, and then choose **Settings**. Click **API Details**.
- For a skill, click the overflow menu, and then choose **View API Details**.



**Note:** You can find the ID of the workspace that is associated with a dialog skill on the *Skill API Details* page also. It is included in the Legacy v1 workspace URL, after the `/workspaces/` segment.

If you cannot view the API details or service credentials, it is likely that you do not have Manager access to the service instance in which the resource was created. Only people with Manager service access to the instance can use the service credentials.

If you have a Writer or Reader service access role to the instance and want to use the API, you can create a personal API key. Go to the [IBM Cloud API keys](#) page to create an IBM Cloud API key to use to authenticate your API requests. For more information, see [Managing user API keys](#). With the personal key, you can make API requests that you have the appropriate privileges to complete. For more information about API privileges per access role, see [Resource-level role impact on available actions](#).



**Tip:** If the assistant ID is not displayed, you can copy it from the URL of the assistant web page. The assistant ID is listed just after `/assistants/{assistantID}`. Similarly, you can get the skill ID from the URL of the skill page. The skill ID is listed just after `/skills/{skillID}`.

## Changing the inactivity timeout setting

When a user interacts with your assistant through one of the built-in integrations, the chat session ends after a specific period of time passes in which the user does not interact with the assistant. You can specify the amount of time to wait before a chat session is reset by changing the inactivity timeout setting.

When a chat session is reset, the dialog loses any contextual information that it saved during the previous exchange with the user. For example, if the dialog asks for the user's name and then calls the user by that name throughout the rest of the conversation, then after the chat session ends and a new one begins, the dialog will start by asking for the user's name again.

To change the timeout, complete the following steps:

1. Click the menu for your assistant , and then choose **Settings**.

2. Change the time specified in the **Timeout limit** field. The choices and maximum timeout vary depending on your plan.

3. Close the settings page. Your change is saved automatically.

## Session limits

---

The length allowed for an inactivity timeout differs by service instance plan type. The following table lists the limits.

Service plan	Chat session default inactivity period	Chat session maximum inactivity period
Enterprise	1 hour	168 hours (7 days)
Premium (legacy)	1 hour	168 hours (7 days)
Plus	1 hour	24 hours
Trial	5 minutes	5 minutes
Lite	5 minutes	5 minutes

[Service plan details](#)



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Adding support for global audiences

Your customers come from all around the globe. You need an assistant that can talk to them in their own language and in a familiar style. Choose the approach that best fits your business needs.

- **Quickest solution:** The simplest way to add language support is to author the conversational skill in a single language. You can translate each message that is sent to your assistant from the customer's local language to the skill language. Later you can translate each response from the skill language back to the customer's local language.

This approach simplifies the process of authoring and maintaining the conversational skill. You can build one skill and use it for all languages. However, the intention and meaning of the customer message can be lost in the translation.

For more information about webhooks you can use for translation, see [Webhook overview](#).

- **Most precise solution:** If you have the time and resources, the best user experience can be achieved when you build multiple conversational skills, one for each language that you want to support. Watson Assistant has built-in support for all languages. Use one of 13 language-specific models or the universal model, which adapts to any other language you want to support.

When you build a skill that is dedicated to a language, a language-specific classifier model is used by the skill. The precision of the model means that your assistant can better understand and recognize the goals of even the most colloquial message from a customer.

Use the new universal language model to create an assistant that is fluent even in languages that Watson Assistant doesn't support with built-in models.

To deploy, attach each language skill to its own assistant that you can deploy in a way that optimizes its use by your target audience.

For example, use the web chat integration with your French-speaking assistant to deploy to a French-language page on your website. Deploy your German-speaking assistant to the German page of your website. Maybe you have a support phone number for French customers. You can configure your French-speaking assistant to answer those calls, and configure another phone number that German customers can use.

## Understanding the universal language model

A skill that uses the universal language model applies a set of shared linguistic characteristics and rules from multiple languages as a starting point. It then learns from the training data that you add to it.

The universal language classifier can adapt to a single language per skill. It cannot be used to support multiple languages within a single skill. However, you can use the universal language model in one skill to support one language, such as Russian, and in another skill to support another language, such as Hindi. The key is to add enough training examples or intent user examples in your target language to teach the model about the unique syntactic and grammatical rules of the language.

Use the universal language model when you want to create a conversation in a language for which no dedicated language model is available, and which is unique enough that an existing model is insufficient.

For more information about feature support in the universal language model, see [Supported languages](#).

## Creating a skill that uses the universal language model

To create a skill that uses the universal language model, complete the following steps:

1. From the **Skills** page, click **Create skill**.
  2. Choose to create either a dialog or actions skill, and then click **Next**.
- For more information, see [Choosing a conversational skill](#).
3. Name your skill, and optionally add a description. From the *Language* field, choose **Another language**.

 **Tip:** Remember, if the language you want to support is listed individually, choose it instead of using the universal model. The built-in language models provide optimal language support.

4. Create the skill.
5. If your skill will support a language with right-to-left directional text, such as Hebrew, configure the bidirectional capabilities.

Click the options icon from the skill tile , and then select **Language preferences**. Click the *Enable bidirectional capabilities* switch. Specify any settings that you want to configure.

For more information, see [Configuring bidirectional languages](#).

Next, start building your conversation.

As you follow the normal steps to design a conversational flow, you teach the universal language model about the language you want your skill to support.

It is by adding training data that is written in the target language that the universal model is constructed. For an actions skill, add actions. For a dialog skill, add intents, intent user examples, and entities. The universal language model adapts to understand and support your language of choice.

## Integration considerations

---

When your skill is ready, you can add it to an assistant and deploy it. Keep these tips in mind:

- **Phone integration:** If you want to deploy an assistant that uses the universal language model with the phone integration, you must connect to custom Speech service language models that can understand the language you're using. For more information about supported language models, see the [Speech to Text](#) and [Text to Speech](#) documentation.
- **Search skill:** If you build an assistant that specializes in a single language, be sure to connect it to data collections that are written in that language. For more information about the languages that are supported by Discovery, see [Language support](#).
- **Web chat:** Web chat has some hardcoded strings that you can customize to reflect your target language. For more information, see [Global audience support](#).

# Deploying

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Deploying your assistant](#).

## Deploying your assistant

After you have built and tested your assistant, you can make it available for customers to use.

There are multiple options for deploying your assistant, depending on how you want your customers to interact with it. A *channel* is a communication platform through which you want to make your assistant available, such as a website or a telephone system; an assistant can be deployed to one or more channels. To deploy to a channel, you use an *integration*, which is essentially an adapter or interface that enables the assistant to communicate through a channel.

In most cases, an assistant is deployed using one of these integrations:

- [Web chat integration](#): The web chat integration provides a secure and highly customizable widget you can add to your website. You can configure how and where the web chat widget appears, and you can use theming to align it with your branding and website design. If a customer needs help from a person, the web chat integration can transfer the conversation to an agent.
- [Phone integration](#): The phone integration enables your assistant to converse with customers on the phone, using the IBM Watson Text to Speech and Speech to Text services. If your customer asks to speak to a person, the phone integration can transfer the call to an agent.

## Other channels

In addition to the web chat and phone integrations, Watson Assistant provides integrations you can use to deploy your assistant to numerous other channels:

- [Facebook Messenger](#)
- [Intercom](#)
- [Slack](#)
- [SMS with Twilio](#)
- [WhatsApp](#)
- A [custom client application](#)

## Deploying to your website



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Adding the web chat to your website

Add your assistant to your company website as a web chat widget that can help your customers with common questions and tasks, and can transfer customers to human agents.

When you create a web chat integration, code is generated that calls a script that is written in JavaScript. The script instantiates a unique instance of your assistant. You can then copy and paste the HTML `script` element into any page or pages on your website where you want users to be able to ask your assistant for help.

To learn more about how web chat can help your business, read [this Medium blog post](#).

### Create a web chat instance to add to your website

To add the assistant to a web page on your company website, complete the following steps:

1. From the Assistants page, click to open the assistant tile that you want to deploy to your site.
2. From the *Integrations* section, either click **Integrate web chat**, or click **Add integration**, and then choose the **Web chat** tile.

The web chat integration is added to your first assistant automatically. If you're using the *My first assistant*, click the **Web chat** tile to open the integration that was added for you.

3. **Optional:** Change the web chat integration name from *Web chat* to something more descriptive.



**Tip:** A preview pane is displayed that shows you what the web chat looks like when it is embedded in a web page. If a message is displayed that starts with, `There is an error`, it means you haven't added a conversational skill to your assistant yet. After you add a skill, you can test the conversation from the preview pane.

4. Click **Create** to create a web chat instance.

You can skip this step if the web chat integration was created for you automatically.



**Note:**

For environments where private endpoints are in use, keep in mind that the web chat integration sends traffic over the internet. For more information, see [Private network endpoints](#).

5. **Optional:** Customize the style of the chat window. You can make the following changes:

- **Assistant's name as known by customers:** The name by which the assistant is known to users. This name is displayed in the header of the chat window. The name can be up to 64 characters in length.

- **Primary color:** The color of the web chat header.

Click the white dot to open a color switcher where you can choose a color. The color is saved as an HTML color code, such as **#FF33FC** for pink and **#329A1D** for green. Alternatively, you can add an HTML color code directly to the field to set the color.

- **Secondary color:** The color of the user input message bubble.

- **Accent color:** The color of interactive elements, including:

- Chat launcher button that is embedded in your web page
- Send button associated with the input text field
- Input text field border when in focus
- Marker that shows the start of the assistant's response
- Border of a button after it is clicked
- Border of the drop-down list field as the user chooses an option
- Typing indicator that is shown to represent a pause response

Style changes you make are immediately applied to the preview that is shown on the page, so you can see how your choices impact the style of the chat UI.

- **IBM Watermark:** Enables or disables the **Built with IBM Watson** watermark displayed in the web chat window. The watermark is always enabled for any new web chat integrations on Lite plans.

6. If you want to add an image that represents your assistant or organization to the web chat header, click **Add an avatar image**.

Specify the URL for a publicly accessible hosted image, such as a company or brand logo or an assistant avatar. The image file must be between 64 x 64 and 100 x 100 pixels in size.

Click **Save**.

7. Configure the home screen for the chat window.

The home screen helps to ease your customers into a conversation with your assistant. You can add a greeting and a list of quick conversation starter questions for customers to click. For more information, see [Configuring the home screen](#).

If you don't want to use a home screen, go to the **Home screen** tab and toggle the switch to **Off**.

8. **Optional:** To configure support for transferring conversations to a service desk agent, click the **Live agent** tab. For more information, see [Adding service desk support](#).

9. **Optional:** The web chat gives your customers a way to reset the conversation if they get stuck by showing them a list of suggestions. Suggestions are enabled automatically. You can control how often suggestions are displayed and what they include. Click the **Suggestions** tab. For more information, see [Showing more suggestions](#).

10. **Optional:** To secure the web chat, click the **Security** tab. For more information, see [Securing the web chat](#).

11. Click the **Embed** tab.

A code snippet is displayed that defines the chat window implementation. You will add this code snippet to your web page. The code snippet contains an HTML script element. The script calls JavaScript code that is hosted on an IBM site. The code creates an instance of a widget that communicates with the assistant. The generated code includes a region and unique integration ID. Do not change these parameter values.

12. Copy the **script** HTML element. You add this script to your website in the next section, [Deploy your assistant in production](#).

13. If you made any customizations, click **Save and exit**. Otherwise, click **Close**.

The web chat instance is created as soon as you click the *Create* button, and does not need to be saved.

## Deploy your assistant in production

1. If the system that hosts your website has limited Internet access (for example, if you use a proxy or firewall), make sure the following URLs are accessible:

- **https://web-chat.global.assistant.watson.appdomain.cloud**: Hosts the code for the web chat widget, and is referenced by the script you embed on your website.

- o `https://integrations.{location}.assistant.watson.appdomain.cloud`: Hosts the web chat server, which handles communication with your assistant. Replace `{location}` with the location of the data center where your service instance is located, which is part of the service endpoint URL. For more information, see [Finding and updating the endpoint URL](#).

2. Open the HTML source for a web page on your website where you want the chat window to be displayed. Paste the code snippet into the page.



**Tip:** Paste the code as close to the closing `</body>` tag as possible to ensure that your page renders faster.

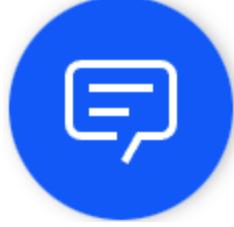
The following HTML snippet is the source for a test page that you can copy and save as a file with a .html extension for testing purposes. You would replace the script element block here with the script elements you copied from the web chat integration setup page.

```
<html>
<head></head>
<body>
 <title>My Test Page</title>
 <p>The body of my page.</p>

 </body>
</html>
```

3. Refresh the web page.

For information about the web browsers that are supported by the web chat, see [Browser Support](#).



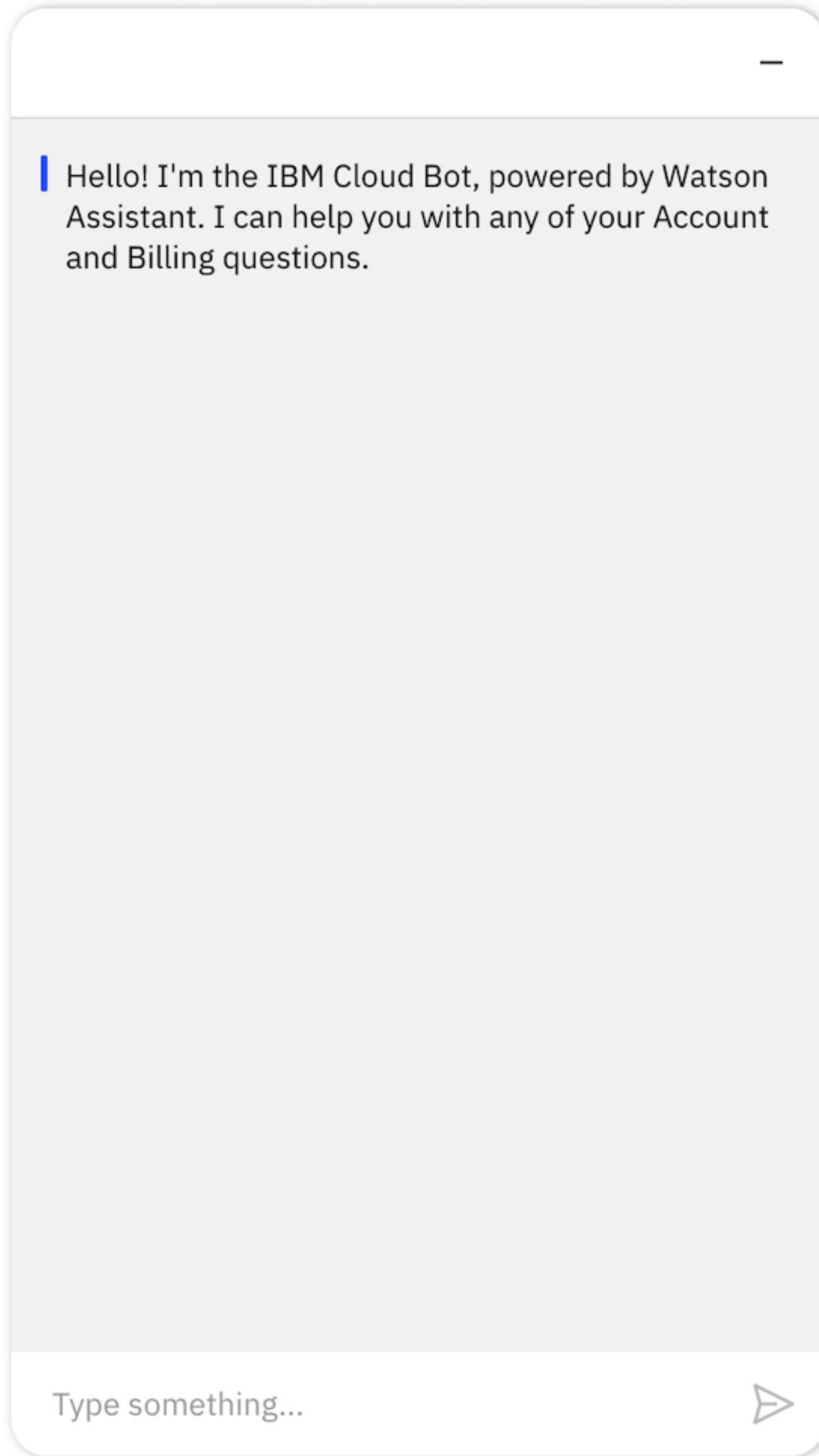
The web chat launcher icon is displayed at the end of the page. The icon is blue unless you customize the accent color.



**Important:** The placement of the web chat icon is always the same regardless of where you paste the script element into the web page source. The chat window is represented by a `div` HTML element.

A developer can make more involved style changes, such as changing the launcher icon and its placement or the size and position of the chat window. For more information, see [Applying advanced customizations](#).

4. Click the icon to open the chat window and talk to your assistant.



5. Paste the code snippet into each web page where you want the assistant to be available to your customers.

 **Tip:** You can paste the same script tag into as many pages on your website as you want. Add it anywhere where you want users to be able to reach your assistant for help. However, be sure to add it only one time per page.

6. Submit test utterances from the chat widget that is displayed on your web page to see how the assistant responds.

No responses are returned until after you create a dialog skill and add it to the assistant.

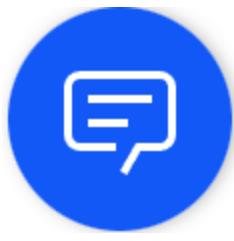
 **Note:** If the **Connect to agent** button is displayed and you don't have human agent support configured, you can hide it by changing the *Suggestions* configuration. For more information, see [Showing more suggestions](#).

If you don't extend the session timeout setting for the assistant, the dialog flow for the current session is restarted after 60 minutes of inactivity. This means that if a user stops interacting with the assistant, after 60 minutes, any context variable values that were set during the previous conversation are set to null or back to their initial values.

A developer can use APIs to apply more advanced customizations to the style of the web chat. For more information, see [Applying advanced customizations](#).

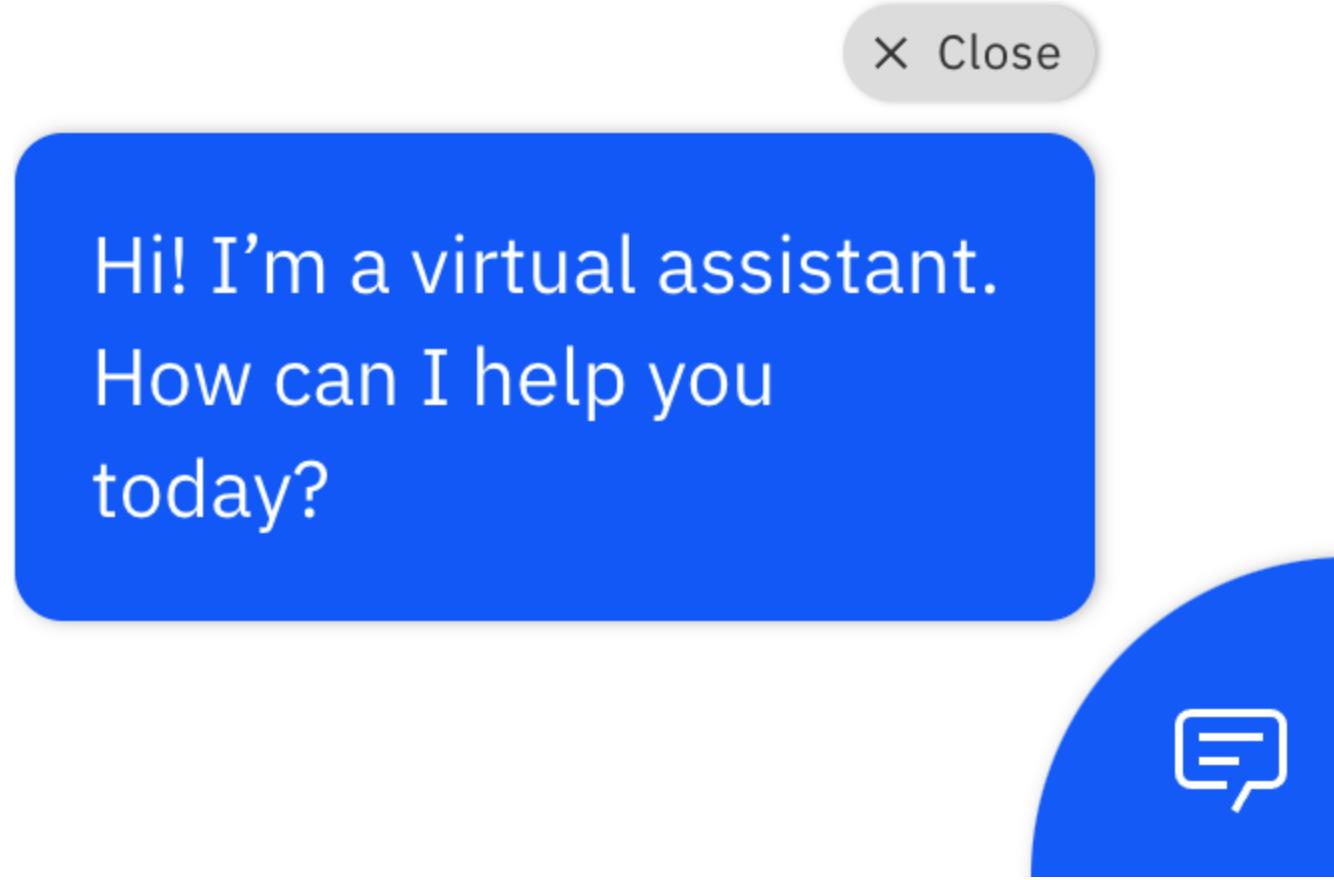
## Launcher appearance and behavior

The web chat launcher welcomes and engages customers so they know where to find help if they need it. By default, the web chat launcher appears in a small initial state as a circle in the bottom right corner:



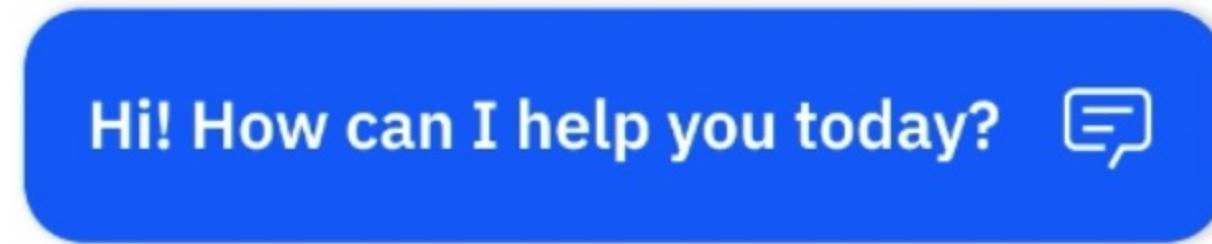
After 15 seconds, the launcher expands to show a greeting message to the user. In this expanded state, a customer can still click the launcher to open the web chat. (If the customer reloads the page or navigates to a different page before the launcher has expanded, the 15-second timer restarts.) There are two slightly different appearances for this expanded state, depending on whether the user is using a desktop browser or a mobile browser.

- For desktop browsers, the expanded launcher shows two primary buttons the customer can click to open the web chat, and a **Close** button that closes the launcher:



The expanded launcher remains in its expanded state even if the customer reloads the page or navigates to a different page. It stays in its expanded state until the customer either opens it by clicking on either of the two primary buttons, or closes it, at which point it returns to its initial small state for the rest of the session.

- For mobile browsers, the launcher shows only a single primary button:



The customer can close the launcher by scrolling on the page, swiping right on the expanded launcher, or waiting 10 seconds, at which point the expanded launcher shrinks back to its initial small state automatically. If the user reloads the page or navigates to a different page while the launcher is expanded, it stays in its expanded state, and the 10-second timer restarts.

The color of the launcher is specified by the **Accent color** field on the **Style** tab of the web chat settings. To change the color, specify a new color using a standard hexadecimal RGB value.

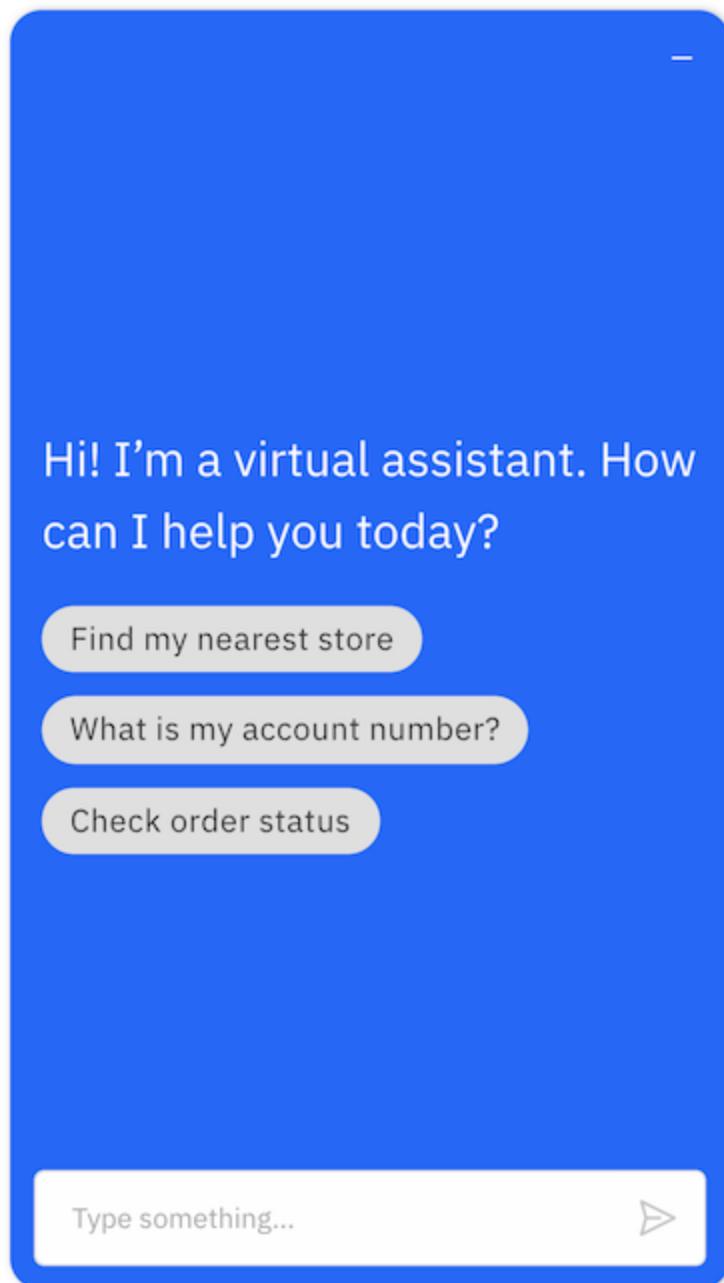
You can customize the greeting message displayed by the launcher on the **Launcher** tab of the web chat settings. The settings include separate greeting messages for the desktop and mobile versions of the launcher.



**Note:** The language of the default text shown within the launcher depends on the locale configured for the web chat. For more information, see [Languages](#). If you customize the greeting text, the text you provide is used regardless of the locale settings.

## Configuring the home screen

By default, the web chat window shows a home screen that can welcome users and tell them how to interact with the assistant. The home screen replaces any greeting that would otherwise be sent by a Welcome node in a dialog skill, or by the *Greet customer* system action in an actions skill. (If you prefer to use a Welcome node or a *Greet customer* system action instead of the home screen, you can disable the home screen on the **Home screen** tab.)



1. Add a greeting that is engaging and invites the user to interact with your assistant. A greeting is required.
2. Add three conversation starter messages.

These messages are displayed in the web chat as examples of the types of questions that customers can ask. Customers can click one of them to submit it to the assistant.

You must test each message that you add as a conversation starter. Use only questions that the assistant understands and knows how to answer well.

If your assistant has multiple skills attached to it, the dialog skill orchestrates the incoming messages. If you want an action that you created to respond to a conversation starter message, make sure your dialog is set up to call the action. For more information, see [Calling an actions skill from a dialog](#).

All three conversation starters are required.

A developer can customize the home screen even more:

- A **Get started** heading is displayed before the list of conversation starter messages. A developer can change the heading text by replacing the `homeScreen_conversationStarterLabel` in the web chat language strings file. For more information, see the [instance.updateLanguagePack\(\) method](#) documentation.
- You can use the web chat API to add other elements to the home screen page. For more information, see the [instance.writeableElements\(\) method](#) documentation.
- For information about CSS helper classes that you can use to change the home screen style, see the [prebuilt templates](#) documentation.

## Showing more suggestions

*Suggestions* give your customers a way to try something else when the current exchange with the assistant isn't delivering what they expect. A question mark icon  is displayed in the web chat that customers can click at any time to see other topics that might be of interest or, if configured, to request support. Customers can click a suggested topic to submit it as input or click the **X** icon to close the suggestions list.

Starting with web chat version 3.1, if customers select a suggestion and the response is not helpful, they can open the suggestions list again to try a different suggestion. The input generated by the first choice is submitted and recorded as part of the conversation. However, any contextual information that is generated by the initial suggestion is reset when the subsequent suggestion is submitted.

The suggestions are shown automatically in situations where the customer might otherwise become frustrated. For example, if a customer uses different wording to ask the same question multiple times in succession, and the same dialog node is triggered each time, then related topic suggestions are shown in addition to the triggered node's response. The suggestions that are offered give the customer a quick way to get the conversation back on track.

The suggestions list is populated with dialog nodes that condition on intents that are related in some way to the matched intent. The intents are ones that the AI model considered to be possible alternatives, but that didn't meet the high confidence threshold that is required for a node to be listed as a

disambiguation option. Any dialog node with a node name (or external node name) can be shown as a suggestion, unless its **Show node name** setting is set to **Off**.

To customize suggestions, complete the following steps:

1. Open the *Suggestions* tab.

The Suggestions feature is enabled automatically for new web chat integrations. If it's not enabled, set the Suggestions switch to **On**.

The *Include a connection to support* section is displayed where you can configure whether and how to give customers the ability to connect with support.

2. Decide when you want an option to connect with support to be shown in the suggestions list. The choices are:

- **Always**: Always shows the option to get support in the list of suggestions.
- **Never**: Never shows the option to get support in the list of suggestions.
- **After one failed attempt**: Adds the option to the list only if the customer reached a node with an anything\_else condition in the previous conversation turn or reaches the same dialog node for a second time in succession.

3. In the **Option label** field, add a label for the option.

The text in the **Option label** field has two functions:

- The text is shown in the suggestions list as an option for customers to select.
- When selected by a customer, the text is sent to your assistant as a new message. The label must be able to function as input that your dialog understands and knows how to handle.

By default, the option label **Connect with agent** is used. If your web chat is integrated with a service desk, this message initiates a conversation transfer, as long as your dialog is designed to handle transfer requests.

If your web chat is not integrated with a service desk, you can change the option label to a message that helps your customers reach whatever form of support you do offer. If you offer a toll-free support line, you might add **Get the support line phone number**. Or if you offer an online support request form, you might add **Open a support ticket**.

Whether you use the default option label or add your own, make sure your dialog is designed to recognize the message and respond to it appropriately. For more information, see [Connecting customers with support](#).



**Note:** With web chat version 2.4 only, you cannot enable suggestions unless your web chat is connected to [a service desk solution](#). Later releases allow you to enable suggestions whether or not you have human agent support configured.

## Dialog considerations

The rich responses that you add to a dialog are displayed in the web chat as expected, with the following exceptions:

- **Connect to human agent**: If service desk support is enabled for the web chat, this response type triggers a chat transfer. If service desk support is not configured, this response type is ignored.
- **Option**: If your option list contains up to four choices, they are displayed as buttons. If your list contains five or more options, then they are displayed in a drop-down list.
- **Pause**: This response type pauses the assistant's activity in the chat. However, activity does not resume after the pause until another response is triggered. Whenever you include a **pause** response type, add another, different response type, such as **text**, after it.

For more information about rich response types, see [Rich responses](#).

If you want to use the same dialog for an assistant that you deploy to many different platforms, add custom responses per integration type. You can add a conditioned response that tells the assistant to show the response only when the web chat integration is being used. For more information, see [Building integration-specific responses](#).

## Web chat security

Configure the web chat to authenticate users and send private data from your embedded web chat. For more information, see [Securing the web chat](#).

## Adding service desk support

Delight your customers with 360-degree support by integrating your web chat with a third-party service desk solution.

The following service desk offerings are supported:

- [Zendesk](#)
- [Salesforce](#)

Fully functional reference implementations are provided for the following service desks:

- [Genesys Cloud](#)
- [NICE inContact](#)

- [Twilio Flex](#)
- [Oracle B2C Service](#)

**⚠️ Important:** The starter kit reference implementations, while functional, are examples only, and have not been vetted for production use. You should perform robust testing before deploying these integrations in production.

- [Bring your own](#): A web chat service desk extension starter kit that enables you to develop your own service desk integrations.

After you set up the service desk integration, you must update your dialog to ensure it understands user requests to speak to someone, and can transfer the conversation properly. For more information, see [Adding chat transfer support](#).

## Web chat integration limits

The usage is measured differently depending on the plan type. For Lite plans, usage is measured by the number of `/message` calls (API) are sent to the assistant from the web chat integration. For all other plans, usage is measured by the number of monthly active users (MAU) that the web chat interacts with. The maximum number of allowed MAUs differs depending on your Watson Assistant plan type.

Plan	Maximum usage
Enterprise	Unlimited MAU
Premium (legacy)	Unlimited MAU
Plus	Unlimited MAU
Trial	5,000 MAU
Lite	10,000 API (approximately 1,000 MAU)

### Plan details

For more information about how the web chat widget tracks MAUs, see [Billing](#).

**⚠️ Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with Salesforce

Integrate your web chat with a Salesforce service desk solution so your customers always get the help they need.

Integrate with a Salesforce service desk by deploying your assistant with the web chat integration. The web chat serves as the client interface for your assistant. If, in the course of a conversation with your assistant, a customer asks to speak to a person, you can transfer the conversation directly to a Salesforce agent.

Salesforce is a customer relationship management solution that brings companies and customers together. It is one integrated CRM platform that gives all your departments, including marketing, sales, commerce, and service, a single, shared view of every customer.

## Before you begin

To connect to a Salesforce service desk, your organization must have a Salesforce Service Cloud plan that supports Live Agent Chat. Chat support is available in Salesforce Service Cloud Unlimited and Enterprise plans. It is also available with Performance or Developer plans that were created after 14 June 2012.

Your organization must have a [Salesforce chat app](#) with the following characteristics:

- Console navigation
- Navigation items: Cases, Chat sessions, Chat transcripts
- User profiles: Apply the appropriate profiles to ensure that agents can access the app and view chat history information. You can limit access to this page later. See [Profiles](#).
- A [chat deployment](#).
- A [chat button deployment](#).
- Routing must be configured for the chat button. See [Chat routing options](#).
- If you choose omni-channel routing, be sure to include omni-channel as a utility in the chat app. See [Omni-Channel](#).

You must have a level of access to your Salesforce service desk deployment that allows you to do the following things:

- Edit the chat app

- Get chat deployment and button code details
- Add custom fields to layout objects
- Create Visualforce pages

If you don't, ask someone with the appropriate level of access to perform this procedure for you.

## Setting up the Salesforce service desk connection

To set up a Salesforce service desk integration, complete the following steps:

1. Create a web chat integration. For more information, see [Integrating the web chat with your website](#).
2. From the web chat integration page in Watson Assistant, set the **Allow transfers to live agents** switch to **On**, and then choose **Salesforce** as the service desk type. Click **Next**.
3. For Watson Assistant to connect to a Salesforce service desk, it needs information about your organization's Salesforce chat deployment and button implementations. Specifically, it needs the API endpoint, organization ID, deployment ID, and button ID. The service can derive the values that it needs from code snippets that you copy and paste to this configuration page.

 **Tip:** In a separate browser tab or window, open your Salesforce account settings page. Log in with a user ID that has administrative privileges. You must switch back and forth between your Salesforce and Watson Assistant web chat integration setup pages. It's easier to do so if you have both pages open at once.

- Get the deployment code for your Salesforce Agent Configuration chat deployment.

Go to the Salesforce **Feature Settings>Service>Chat>Deployments** page. Find your organization's deployment. Scroll to the end of the chat deployment configuration page and copy the *Deployment Code* snippet.

- Paste the deployment code snippet into the **Deployment code** field in the Watson Assistant Salesforce configuration page.
- Get the Chat Button code.

Go to the Salesforce **Feature Settings>Service>Chat>Chat Buttons & Invitations** page. Find your organization's button implementation. Scroll to the end of the page, and then copy the *Chat Button Code* snippet.

- Paste the chat button code snippet into the **Chat button code** field in the Watson Assistant Salesforce configuration page, and then click **Next**.

4. Add a chat app that enables the Salesforce agent to see a history of the chat. To do so, create a Visualforce page, and then add a chat app to the page.
5. Add custom fields to the Salesforce chat transcript layout.



**Note:** This is a one-time task. If the fields already exist for your organization, you can skip this step.

These custom fields are referenced from the Visualforce page code that you will use in the next step.

See [Create Custom Fields](#).

From the Salesforce **Data>Objects and Fields>Object Manager>Chat Transcript>Fields & Relationships** page, create the following custom fields:

- **Token:** Stores a Watson Assistant authentication token that secures the communication between Salesforce and your assistant.
  - **Data Type:** Text Area (Long)
  - **Field Label:** `x-watson-assistant-key`
  - **Field Length:** Specify the maximum length allowed to ensure it can hold a token that might contain over 100,000 characters.

6. Create a Visualforce page.

Visualforce pages are the mechanism that Salesforce provides for you to customize a live agent's console by adding your own pages to it. A Visualforce page is similar to a standard web page, but it provides ways for you to access, display, and update your organization's data. Pages can be referenced and invoked by using a unique URL, just as HTML pages on a traditional web server can be. See [Create Visualforce Pages](#).

- From the web chat integration page in Watson Assistant, copy the code snippet from the Visualforce page markup field.
- Switch to your Salesforce web page. Search for **Visualforce Pages**. Create a page. Add a label and name to the page. Select the **Available for Lightning Experience, Lightning Communities, and the mobile app** checkbox. Paste the code snippet that you copied in the previous step into the page markup field.

7. Add the Visualforce page that you created to the Salesforce chat app.

To ensure the Salesforce agents can see history of the chat between the customer and your assistant, you must add the page that you created earlier into the console that they use to keep track of their work. See [Create and Configure Lightning Experience Record Pages](#).

- From the Salesforce App Launcher, open the chat app that you created for your agents to talk to customers.

- Open the *Chat Transcripts* object, and then select a transcript page.
- Click the *Setup* icon, and then select *Edit Page*.
- Drag the Visualforce component and drop it into the Chat Transcript Record page layout where you want the chat window to be displayed.
- In the component editor, select the Visualforce page that you created earlier, make any adjustments to the component height that you want, and then click *Save*.

**⚠ Important:** If you do make changes, make sure the height of the Visualforce page is 20 px smaller than the height of the component that you add it to. By default, the height of the component is 300 px and the height of the Visualforce page is 280 px. (The height of the Visualforce page is specified in the `height` attribute of the `iframe` HTML element in the code snippet that you copy and paste.)

- Click *Activation*, and then click the APP, RECORD TYPE, AND PROFILE tab.
- Select the apps to which you want to apply the page layout, and then click *Next*.
- Select the appropriate record type, such as Main, and then click *Next*.
- Select user profiles to give the appropriate set of users access to the page. Limit the group to include only those who you want to be able to view chat history information in the page.
- Click *Next*, and then click *Save*.

#### 8. From the Salesforce configuration page in Watson Assistant, click **Save and exit** to finish setting up the connection.

When you test the service desk integration, make sure there is at least one agent with **Available** status.

Watch a 5-minute video that provides an overview of setting up a connection to a Salesforce service desk:



[View video: Salesforce Integration: Watson Assistant](#)

## Adding transfer support to your dialog

Update your dialog to make sure it understands when users request to speak to a person, and can transfer the conversation properly. For more information, see [Adding chat transfer support](#).

## Adding routing logic for transfers

When you enable transfers to the Salesforce service desk, the default routing preference that you specify is used. However, there might be times when you want to route a customer to a different Salesforce agent queue. For example, your dialog might have a root dialog node that conditions on a `#close_account` intent. For that branch of the conversation only, you want to transfer customers to agents in the sales queue who are authorized to offer incentives as a way to retain customers. You can direct transfers to specific agent queues by adding routing logic to your dialog.

You can specify alternate routing preferences based on:

- browser information
- the current topic of conversation

## Routing based on browser information

When a customer interacts with the web chat, information about the current web browser session is collected. For example, the URL of the current page is collected. You can use this information to add custom routing rules to your dialog. For example, if the customer is on the Products page when a transfer to a human is requested, you might want to route the chat transfer to a queue with agents who are experts in your product portfolio. If the customer is on the Returns page, you might want to route the chat transfer to a queue with agents who know how to help customers return merchandise.

For more information, see [Web chat: Accessing browser information](#).

## Routing by topic

When you enable transfers to the Salesforce service desk, you copy and paste code snippets from Salesforce into the service desk transfer setup page. These code snippets define how transferred conversations are handled within Salesforce. Routing rules are included in the initial transfer configuration. The routing rules identify the queue of agents to which messages from the assistant are transferred by default.

The code that you add to the setup page when you configure the service desk integration shares the following required information with Watson Assistant:

- **organization\_id**: Unique ID of the organization. A company can have more than one organization set up in Salesforce.
- **chat\_api\_endpoint**: Salesforce API endpoint that is used by the integration to communicate with Salesforce.
- **deployment\_id**: Unique ID of the deployment. An organization can have multiple deployments.
- **button\_id**: Unique ID of a button, which defines the specific routing rules for incoming messages. Each deployment can have multiple buttons associated with it.

To override the default routing rules, you must specify a new value for the **button\_id**. Before you perform this procedure, find out the ID of the Salesforce button implementation with the alternative routing rules that you want to use.

To add custom routing logic, complete the following steps:

1. From the *Dialog* page, find the root dialog node for the branch of the conversation that you want to route to a specific group of Salesforce agents which is distinct from the default group.
2. Find the dialog node in the branch where you want the transfer to take place, and then add the *Connect to human agent* response type as the dialog node response type.  
For more information, see [Adding a Connect to human agent response type](#).
3. After you add the response type and customize the transfer messages, select *Salesforce* from the **Service desk routing** field.
4. In the **Button ID** field, add the **button\_id** value for the alternate routing destination that you want the assistant to use for conversations about only this topic. For example, **5733i0000008yGz**.



**Note:** Be sure to specify the exact right syntax for the **button\_id** value. The value is not validated by the service as you add it to your dialog.

If the special routing that you apply fails to deliver the message for any reason, the routing preference that is specified in the Salesforce service desk setup page is used. If both routing preferences fail, the transferred message is treated like a new message in Salesforce. The standard Salesforce routing rules are followed.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with Zendesk

Integrate your web chat with a Zendesk service desk solution so your customers always get the help they need.

Integrate with a Zendesk service desk by deploying your assistant with the web chat integration. The web chat serves as the client interface for your assistant. If, in the course of a conversation with your assistant, a customer asks to speak to a person, you can transfer the conversation directly to a Zendesk agent.

Zendesk Chat lets you help customers in real time, which increases customer satisfaction. And satisfied customers are happier customers. To learn more about this service desk solution, see the [Zendesk website](#).

Zendesk Chat is an add-on to Zendesk Support. Zendesk Support puts all your customer support interactions in one place, so communication is seamless, personal, and efficient, which means more productive agents and satisfied customers.

### Before you begin

1. You must have a Zendesk account. If not, create one.



**Important:** A Zendesk Chat Enterprise plan is required.

2. Decide whether you want to enable security.

If you choose to enable security in Zendesk, you must collect the name and email address of each user. This information must be passed to the web chat so it can be provided to Zendesk when the conversation is transferred.

## Setting up the Zendesk service desk connection

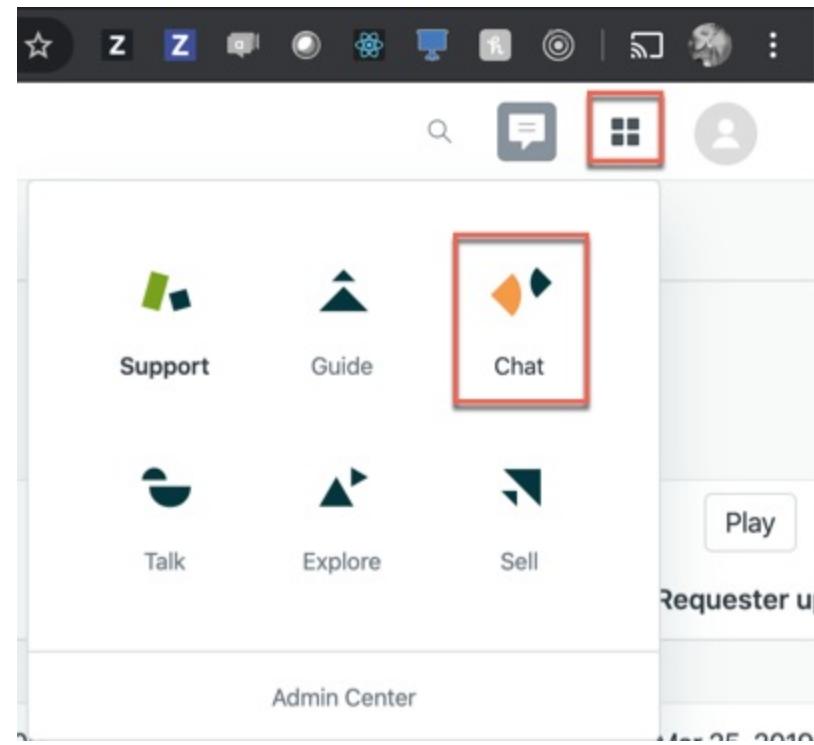
To set up a Zendesk service desk integration, complete the following steps:

1. Create a web chat integration. For more information, see [Integrating the web chat with your website](#).
2. From the web chat integration page in Watson Assistant, set the **Allow transfers to live agents** switch to **On**, and then choose **Zendesk** as the service desk type, and then click **Next**.
3. Add the account key for your Zendesk account, and then click **Next**.

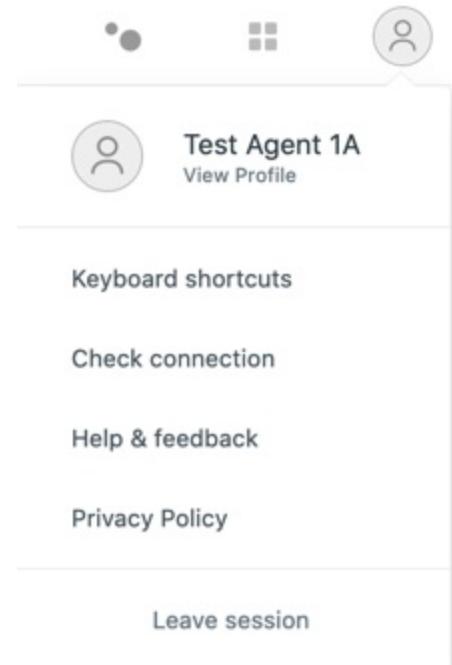
To get the account key for your Zendesk account, follow these steps:

- Log in to your Zendesk subdomain.
- Open the Zendesk Chat Dashboard.

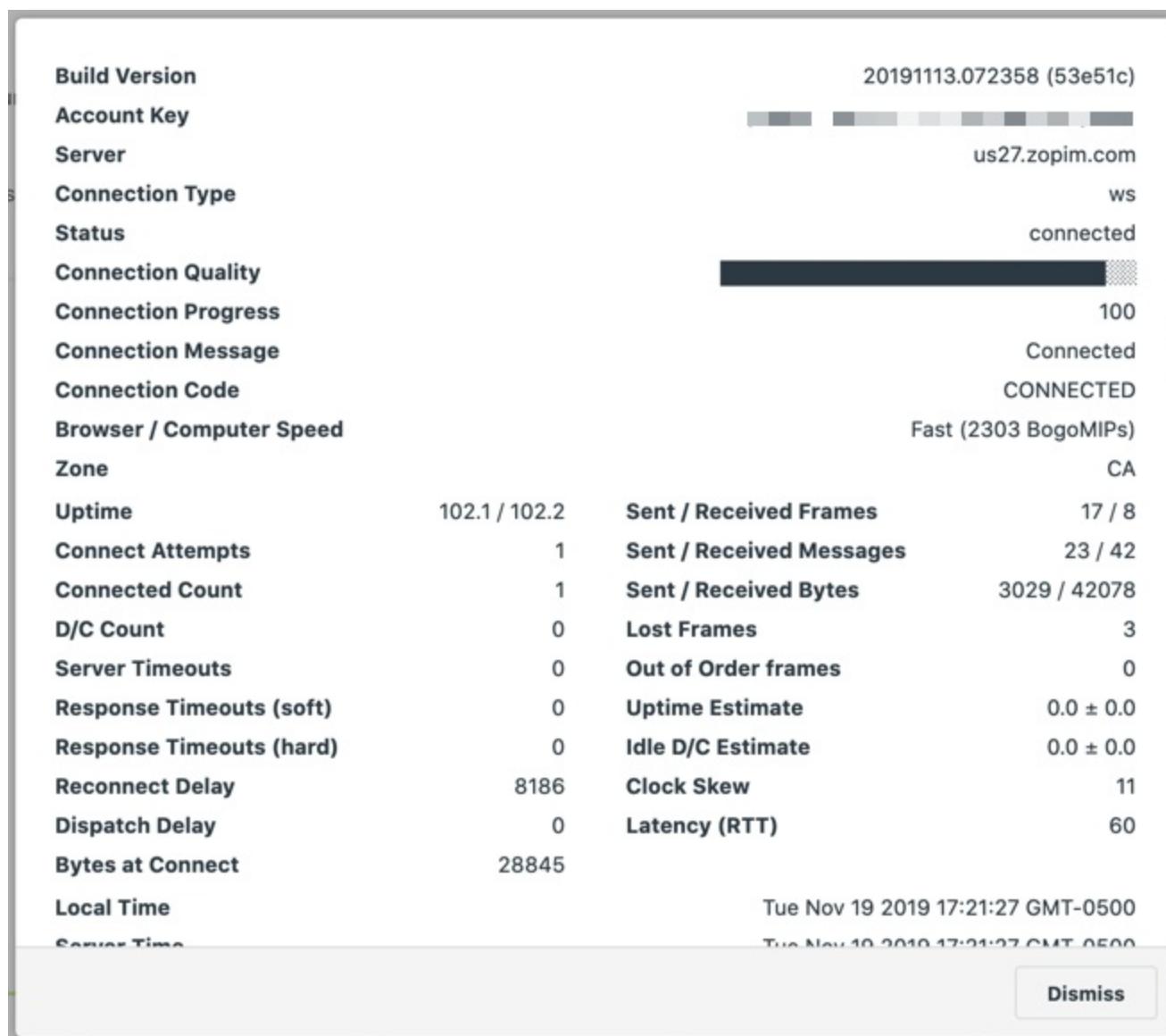
From the Zendesk Support dashboard, you can click the *Zendesk Products* icon in the header, and then click the *Chat* icon.



- Click your profile, and then click *Check Connection*.



- Copy the account key value.



- Return to the setup page in Watson Assistant, and then paste the key into the field.

#### 4. Install the Watson Assistant private application in your Zendesk Chat subdomain.

When you create a Zendesk Chat account, you specify a subdomain. Afterward, your Zendesk console is available from a URL with the syntax: **<subdomain>.zendesk.com**. For example, **ibm.zendesk.com**.

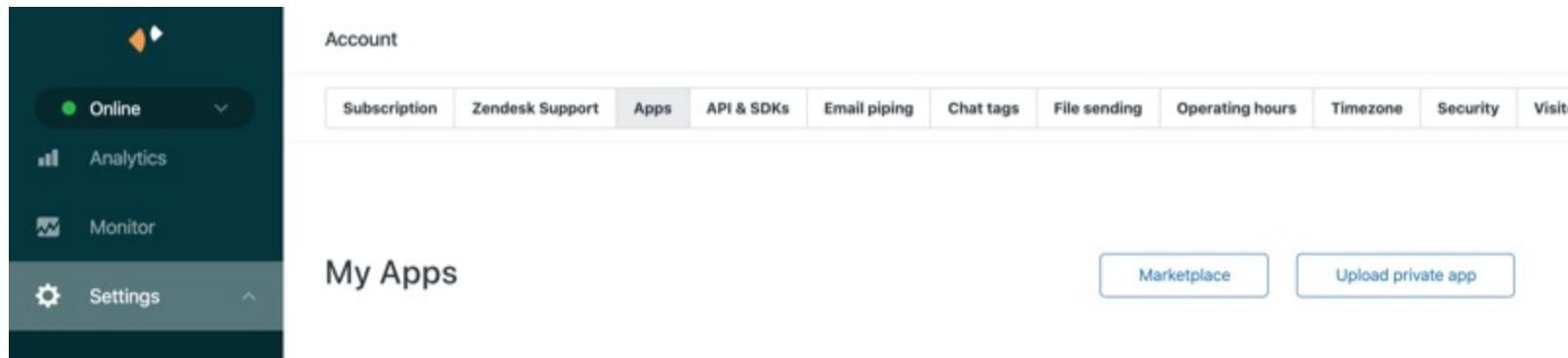
IBM provides an application that you can install in your Zendesk Chat domain. When a customer asks to speak to a person, your assistant will share a chat summary for the transferred conversation with the Zendesk agent by using this private app.

- Download the Watson Assistant Zendesk application from the Zendesk Chat setup page in Watson Assistant.



**Note:** On Safari, the application files are extracted from the ZIP file into a folder. To keep the file archived as a .zip file, so you can upload it later, edit the Safari preferences. Clear the *Open safe files after downloading* checkbox.

- Log in to Zendesk with a user ID that has administrative privileges.
- Install the Watson Assistant Zendesk app to your Zendesk Chat subdomain as a new private app.
  - First, make sure the Zendesk Agent Workspace is not enabled for your account: From the Zendesk navigation pane, go to *Settings*, and then click *Agents*. Deselect the *Enable the Zendesk Agent Workspace* checkbox.
  - Then download the app. From the Chat dashboard navigation pane, expand *Settings*, and then click *Account*.
  - Open the *App* tab.
  - Click *Upload private app*, and then browse for the application file that you downloaded earlier.



For more information, see [Uploading and installing a private app in Zendesk Chat](#).

#### 5. Click **Save and exit** to finish setting up the connection to the Zendesk Chat service desk.

When you test the service desk integration, make sure there is at least one agent with **Online** status. Agent status is set to **Invisible** unless it is explicitly changed.

Watch a 4-minute video that provides an overview of setting up a connection to a Zendesk service desk:



[View video: Zendesk Integration: Watson Assistant](#)

 **Note:** The product user interface is slightly different from the interface that is shown in the video. However, the main steps are the same.

## Securing the transfer to Zendesk

When you add security to your Zendesk integration, you ensure that the visitors you are helping are legitimate customers. Enabling visitor authentication also enables support for cross-domain traffic and cross-browser identification. For more information, see the [Zendesk documentation](#).

Before you can secure the Zendesk connection, complete the following required tasks:

1. Secure the web chat. For more information see [Securing the web chat](#).
2. Encrypt sensitive information that you pass to the web chat.

When you enable security in Zendesk, you must provide the name and email address of the current user with each request. Configure the web chat to pass this information in the payload.

Specify the information by using the following syntax. Use the exact names (`name` and `email`) for the two name and value pairs.

```
{
 user_payload : {
 name: '#{customerName}',
 email: '#{customerEmail}'
 }
}
```

For more information, see [Passing sensitive data](#).

Zendesk also expects `iat` and `external_id` name and value pairs. However, there's no need for you to provide this information. IBM automatically provides a JWT that contains these values.

For example:

```
const userPayload = {
 "name" : "Cade Jones",
 "email" : "cade@example.com",
}
```

```
// Sample NodeJS code on your server.
const jwt = require('jsonwebtoken');
const RSA = require('node-rsa');

const rsaKey = new RSA(process.env.PUBLIC_IBM_RSA_KEY);

/**
 * Returns a signed JWT. Optionally, adds an encrypted user_payload in stringified JSON.
 */
function mockLogin(userID, userPayload) {
 const payload = {
 sub: userID, // Required
 iss: 'www.ibm.com', // Required
 acr: 'loa1' // Required
 // A short-lived exp claim is automatically added by the jsonwebtoken library.
 };
 if (userPayload) {
 // If there is a user payload, it is encrypted in base64 format using the IBM public key.
 payload.user_payload = rsaKey.encrypt(userPayload, 'base64');
 }
 const token = jwt.sign(payload, process.env.YOUR_PRIVATE_RSA_KEY, { algorithm: 'RS256', expiresIn: '10000ms' });
 return token;
}
```

3. From the Zendesk application, enable visitor authentication.
  - From the Chat dashboard navigation pane, expand *Settings*, and then click *Widget*.

- Open the *Widget security* tab.
- In the *Visitor Authentication* section, click the *Generate* button.

For more information, see [Enabling authenticated visitors in the Chat widget](#). You do not need to follow the steps to create a JWT. The Assistant service generates a JSON Web Token for you.

#### 4. Copy the shared secret from Zendesk.

To secure the Zendesk connection, complete the following steps:

1. In the *Authenticate users* section, set the switch to **On**.
2. Paste the secret that you copied from the Zendesk setup page into the **Zendesk shared secret** field.
3. Decide whether to allow unidentified users to access Zendesk.

The web chat integration allows anonymous users to initiate chats. However, as soon as you enable visitor authentication, Zendesk requires that the name and email of each user be provided. If you try to connect without passing the required information, the connection will be refused.

If you want to allow anonymous users to connect to Zendesk, you can provide fictitious name and email data. Write a function to populate the two fields with fictitious name and email values.

For example, your function must check whether you know the name and email of the current user, and if not, add canned values for them:

```
const userPayload = {
 "name" : "Jane Doe1",
 "email" : "jdoe1@example.com",
}
```

After writing a function that ensures that name and email values are always provided, set the *Authenticate anonymous user chat transfers* switch to **On**.

## Adding transfer support to your dialog

Update your dialog to make sure it understands when users request to speak to a person, and can transfer the conversation properly. For more information, see [Adding chat transfer support](#).

## Adding routing logic for transfers

When you enable transfers to the Zendesk service desk, no routing preferences are specified. The conversation is sent to any available agent. However, there might be times when you want to route a customer to a specific Zendesk department. For example, your dialog might have a root dialog node that conditions on a **#close\_account** intent. For that branch of the conversation only, you want to transfer customers to agents in the Sales department who are authorized to offer incentives as a way to retain customers. You can direct transfers to specific departments by adding routing logic to your dialog.

You can specify alternate routing preferences based on:

- browser information
- the current topic of conversation

## Routing based on browser information

When a customer interacts with the web chat, information about the current web browser session is collected. For example, the URL of the current page is collected. You can use this information to add custom routing rules to your dialog. For example, if the customer is on the Products page when a transfer to a human is requested, you might want to route the chat transfer to agents who are experts in your product portfolio. If the customer is on the Returns page, you might want to route the chat transfer to agents who know how to help customers return merchandise.

For more information, see [Web chat: Accessing browser information](#).

## Routing by topic

You can specify a routing preference for specific topics of conversation in your dialog. When specified, the chat is transferred to the department that you designate. You can choose a department that you know has agents who are best able to address the topic.

Before you perform this procedure, determine which department you want users to be routed to.

To add custom routing logic, complete the following steps:

1. From the *Dialog* page, find the root dialog node for the branch of the conversation that you want to route to a specific Zendesk department.
2. Find the dialog node in the branch where you want the transfer to take place, and then add the *Connect to human agent* response type as the dialog node response type.

For more information, see [Adding a Connect to human agent response type](#).

3. After you add the response type and customize the transfer messages, select **Zendesk** from the **Service desk routing** field.
4. In the **Department** field, add the department to which you want the assistant to transfer customers who want to discuss this topic. For example, **sales**.



**Note:** Be sure to specify the exact right syntax for the department name. The value is not validated by the service as you add it to your dialog.

## Deploying for phone



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Integrating with phone

Plus

By adding the phone integration to your assistant, you can make your assistant available to customers over the phone.

When you add the phone integration to your assistant, you can automatically generate a working phone number that is automatically connected to your assistant. Or, if you prefer, you can connect the assistant to your existing infrastructure by configuring an existing Session Initiation Protocol (SIP) trunk.

A SIP trunk is equivalent to an analog telephone line, except it uses Voice over Internet Protocol (VoIP) to transmit voice data and can support multiple concurrent calls. The trunk can connect to the public switched telephone network (PSTN) or your company's on-premises private branch exchange (PBX). If you choose to generate a free phone number for your assistant, a SIP trunk is automatically provisioned from IntelePeer. You can also choose to use an existing SIP trunk from a provider such as IntelePeer, Genesys, or Twilio.



**Note:** Generating a free phone number is available only with new phone integrations. If you have an existing phone integration and you want to switch to a free phone number, you must delete the existing integration and create a new one.

When your customer makes a phone call using the telephone number connected to your assistant, the phone integration answers. The integration converts output from your assistant into voice audio by using the IBM Watson® Text to Speech service. The audio is sent to the telephone network through the SIP trunk. When the customer replies, the voice input is converted into text by using the IBM Watson® Speech to Text service.

This feature is available only to Plus or Enterprise plan users. Note that Speech to Text and Text to Speech charges are included in the cost of a [monthly active user](#) (MAU).



**Tip:** Depending on the architecture of your existing telephony infrastructure, there are multiple ways you might integrate it with Watson Assistant. For more information about common integration patterns, read the blog post [Hey Watson, can I have your number?](#) on Medium.

## Set up the integration



**Note:** You must have Manager role access to the instance and Viewer role access to the resource group. For more information about access levels, see [Managing access to resources](#).

To set up the integration, complete the following steps:

1. In the **Integrations** section on the main page for your assistant, click **Add integration**.
2. On the **Add integration** page, click **Phone**.
3. Click **Create**.
4. Choose whether you want to generate a free phone number for your assistant or connect to an existing SIP trunk:
  - To generate a free phone number for your assistant, click **Generate a free phone number**.



**Note:** Generating a free phone number is supported only for Watson Assistant instances in the Dallas and Washington DC data centers.

- To use an existing phone number you have already configured with a [SIP trunk provider](#), click **Use an existing phone number with an external provider**.

Click **Next**.

5. If you are using an existing phone number, follow the instructions to configure the SIP trunk. (If you are generating a free phone number, skip this step).

1. On the **Bring your own SIP trunk** page, copy the SIP URI and assign it to your SIP trunk. Click **Next**.

2. On the **Phone number** page, specify the phone number of the SIP trunk. Specify the number by using the international phone number format: **+1 958 555 0123**. Do not surround the area code with parentheses.

Currently, only one primary phone number can be added during initial setup of the phone integration. You can add more phone numbers in the phone integration settings later.

Click **Next**.

6. On the **Speech to Text** page, select the instance of the Speech to Text service you want to use for the phone integration.

- If you have existing Speech to Text instances, select the instance you want to use from the list.
- If you do not have any existing Speech to Text instances, click **Create new instance** to create a new Plus instance.

7. In the **Choose your Speech to Text language model** field, select the language model you want to use.

The list of language models is automatically filtered to use the same language as your assistant. To see all language models, toggle the **Filter models based on assistant language** switch to **Off**.



**Note:** If you created specialized custom models that you want your assistant to use, choose the Speech to Text service instance that hosts the custom models now, and you can configure your assistant to use them later. The Speech to Text service instance must be hosted in the same location as your Watson Assistant service instance. For more information, see [Using a custom language model](#).

For more information about language models, see [Languages and models](#) in the Speech to Text documentation.

Click **Next**.

8. On the **Text to Speech** page, select the instance of the Text to Speech service you want to use for the phone integration.

- If you have existing Text to Speech instances, select the instance you want to use from the list.
- If you do not have any existing Text to Speech instances, click **Create new instance** to create a new Standard instance.

9. In the **Choose your Text to Speech voice** field, select the voice you want to use.

The list of voices is automatically filtered to use the same language as your assistant. To see all voices, toggle the **Filter voices based on assistant language** switch to **Off**.

For more information about voice options, and to listen to audio samples, see [Languages and voices](#) in the Text to Speech documentation.

Click **Next**.



**Important:** Any speech service charges that are incurred by the phone integration are billed with the Watson Assistant service plan as *voice add-on* charges. After the instances are created, you can access them directly from the IBM Cloud dashboard. Any use of the speech instances that occurs outside of your assistant are charged separately as speech service usage costs.

The phone integration setup is now complete. On the **Phone** page, you can click the tabs to view or edit the phone integration.



**Note:** If you chose to generate a free telephone number, your new number is displayed on the **Phone number** tab immediately. However, provisioning the new number so it is ready to use might take several minutes.

## Adding more phone numbers

If you are using existing phone numbers you configured using a SIP trunk provider, you can add multiple numbers to the same phone integration.



**Note:** If you generated a free phone number, you cannot add more numbers.

To add more phone numbers:

1. In the phone integration settings, go to the **Phone number** tab.

2. Use one of the following methods to add phone numbers:

- To add phone numbers one by one, type each number in the table, along with an optional description. Click the checkmark icon to save each number.
- To import a set of phone numbers that are stored in a comma-separated values (CSV) file, click the *Upload a CSV file* icon () , and then find the CSV file that contains the list of phone numbers.

The phone numbers you upload will replace any existing numbers in the table.

## Advanced configuration options

Click the *Advanced options* tab to make any of the following customizations to the call behavior:

### Handle call and transfer failures

You can configure the phone integration to transfer the caller to a human agent if the phone connection fails for any reason. To transfer the caller to a human agent automatically, make the following configuration selections:

- **SIP target when a call fails**: Add the SIP endpoint for your support agent service. Specify a SIP or telephone URI for a general call queue that can redirect requests to other queues. For more information, see [Configuring backup support](#).
- **Call failure message**: Add the message to say to a caller before you transfer them to a human agent.

If, after you transfer the caller to a human agent, the connection to the human agent fails for any reason, you can configure what to do.

- **Transfer failure message**: Add the message to stream to callers if the transfer to a human agent fails. The message can be up to 1,024 characters in length.
- **Disconnect call on transfer failure**: Choose whether to disconnect the call after sharing the failure message. This option is enabled by default. If disabled, when a call transfer fails, your assistant can disconnect or process a different dialog node.

### Secure the phone connection

You can add security to the phone connection by selecting one or both of the following configuration options:

- **Force secure trunking**: Select this option to use Secure Real-Time Transfer Protocol (SRTP) to secure the audio that is transmitted over the phone. For more information about RTP, see [Call routing details](#).
- **Enable SIP authentication**: Select this option if you want to require SIP digest authentication.

When SIP authentication is required, all inbound traffic (meaning requests from the SIP provider to your assistant) is authenticated using SIP digest authentication, and must be sent using Transport Layer Security (TLS). If this option is selected, the SIP digest user name and password must be configured, and the SIP trunk being used to connect to Assistant must be configured to use only TLS.



**Important:** If you use Twilio as your SIP trunk provider, you cannot enable SIP authentication for outbound SIP trunks to Watson Assistant.

### Apply advanced SIP trunk configuration settings

- **SIP INVITE headers to extract**: List headers that you want to use in your dialog.

The SIP request often sends INVITE headers with information about the request that is used by the SIP network. For example, many companies use Interactive Voice Response (IVR) systems that pass information about an incoming call by using headers. If you want to make use of any of these headers, list the header names here.

The specified headers, if present in the request, are stored in the context variable `sip_custom_invite_headers`. This variable is an array in which each key/value pair represents a header from the request, as in this example:

```
$ "sip_custom_invite_headers": {
 "X-customer-name": "my_name",
 "X-account-number": "12345"
}
```

You can then reference these headers in your dialog. For example, you might check the header value in a dialog node condition to determine whether to process a branch. You can also use these headers when searching the Assistant logs; for example, you might search for a custom header to find all the messages associated with particular account.

- **Disable the ring that callers will hear while the assistant is contacted**: Choose whether you want the caller to hear a signal that indicates that the assistant is being contacted.

A `180 Ringing` response is sent from the assistant back to the SIP trunk provider while your assistant processes the incoming call invitation. The ringing response is sent by default.

- **Don't place callers on hold while transferring to a live agent**: Choose whether the phone integration puts the caller on hold.

If your SIP trunk provider manages holds, disable this feature. For example, some SIP trunk providers prefer to have the assistant send a SIP REFER request, so they can put the call on hold themselves.

For more information about the SIP protocol, see [RFC 3261](#) and about the RTP protocol, see [RFC 3550](#).

### Configuring backup call center support

When you use the phone integration as the first line of assistance for customers, it's a good idea to have human backup available. You can design your assistant to be able to transfer a call to a human in case the phone connection fails, or a user asks to speak to someone.

Your company might already have one or more phone numbers that connect to an automatic call dispatcher (ACD) that can queue callers until an appropriate agent is available. If not, choose a call center service to use as your backup.

A conversation cannot be transferred from one integration type to another. If you use the web chat integration with service desk support, there's no way to transfer the phone call to the existing service desk that is set up for the web chat, for example.

For whichever call center service you use, you will need to provide the call center SIP URI. You must specify this information in your dialog when you enable a call transfer from a dialog node. For more information, see [Transfer a call to a human agent](#).

## Optimize your dialog for voice

For the best customer experience, design your dialog with the capabilities of the phone integration in mind:

- Do not include HTML elements in your dialog text responses. To add formatting, use Markdown. For more information, see [Simple text response](#).
- Use the *Connect to human agent* response type to initiate a transfer to a human agent. For more information, see [Transferring a call to a human agent](#).
- Use the *Channel transfer* response type to initiate a transfer to the web chat integration. For more information, see [Transferring the caller to the web chat integration](#).
- The *pause* response type is not supported. If you want to add a pause, use the `turn_settings.timeout_count` context variable (for more information, see [Context variables that are set by your dialog or actions](#)).
- You can include search skill response types in dialog nodes that the phone integration will read. The introductory message (*I searched my knowledge base* and so on), and then the body of only the first search result is read.

The search skill response (meaning the introductory message plus the body of the first search result) must be less than 5,000 characters long or the response will not be read at all. Be sure to test the search results that are returned and curate the data collection that you use as necessary.

For more information about dialog response types, see [Rich responses](#).

For more information about how to implement common actions from your dialog, see [Handling phone integrations](#).

If you want to use the same dialog for an assistant that you deploy to many different platforms, you can add custom responses per integration type. Add a conditioned response that tells the assistant to show the response only when the phone integration is being used. For more information, see [Building integration-specific responses](#).

## Setting up a SIP trunk

You are responsible for setting up the SIP trunk that will be used by the phone integration. Find a provider and create a SIP trunk account. Your account will be charged for the phone integration's use of the SIP trunk.

You can set up a SIP trunk in the following ways:

- [Create a Twilio SIP trunk](#)
- [Use other third-party providers](#)
- [Bring your own SIP trunk](#)
- [Migrate from Voice Agent with Watson](#)

## Creating a Twilio SIP trunk

To set up a Twilio SIP trunk, complete the following steps:

1. Create a Twilio account on the [Twilio website](#).
2. From the Twilio website, go to the *Elastic SIP Trunking* dashboard.
3. Select *Trunks* from the navigation bar and create a SIP trunk. If you already have a SIP trunk, click the plus sign (+). Enter a name for your SIP trunk and click *Create*.
4. From the Elastic SIP Trunks page, select your SIP trunk.
5. Select *Origination* from the navigation bar for your SIP trunk and configure the origination SIP URI.

You can get the SIP URI for your phone integration from the phone integration configuration page.

6. If you plan to support call transfers, enable Call Transfer (SIP REFER) in your SIP trunk. If you expect to transfer calls to the public switched telephone network (PSTN), also enable PSTN Transfer on your trunk.
7. Select *Numbers* from the navigation bar for your SIP trunk, and then do one of the following things:
  - Click *Buy a Number*.
  - If you already have a number, you can click the plus sign (+) to provision a new phone number in your region.
8. Assign the number to the SIP trunk you created by going back to the SIP trunk and clicking the number sign (#) icon.

If you use a Lite or Trial Twilio account for testing purposes, then be sure to verify the transfer target.

You cannot enable SIP authentication if you choose Twilio as your SIP trunk provider. Twilio doesn't support SIPS for originating calls.

## Using other third-party providers

You can ask for help setting up an account with another SIP trunk provider by opening a support request.

IBM has established relationships with the following SIP trunk providers:

- [Five9](#)
- [Genesys](#)
- [Vonage](#)
- [Voximplant](#)

The SIP trunk provider sets up a SIP trunk for your voice traffic, and manages access from allowed IP addresses. Most of the major SIP trunk providers have existing relationships with IBM. Therefore, the network configuration that is required to support the SIP trunk connection typically can be handled for you with minimal effort.

1. Create a [IBM Cloud case](#).
2. Click **Customer success** as the case type.
3. For **Subject**, enter **SIP trunk provider setup for Watson Assistant**.
4. Include the following information in the description:
  - Company Name
  - Your IBM Cloud account ID
  - Your Watson Assistant service name
  - Network diagram with IP address or SIP trunk provider information

## Bring your own SIP trunk

If you choose to use a SIP trunk carrier that IBM does not have an established relationship with, you can do so.

The following table lists the fully qualified domain names that are used for SIP connections.

Location	Domain name
Dallas	public.0001.voip.us-south.assistant.watson.cloud.ibm.com
Dallas	public.0002.voip.us-south.assistant.watson.cloud.ibm.com
Dallas	public.0003.voip.us-south.assistant.watson.cloud.ibm.com
Frankfurt	public.0001.voip.eu-de.assistant.watson.cloud.ibm.com
Frankfurt	public.0002.voip.eu-de.assistant.watson.cloud.ibm.com
Frankfurt	public.0003.voip.eu-de.assistant.watson.cloud.ibm.com
London	public.0001.voip.eu-gb.assistant.watson.cloud.ibm.com
London	public.0002.voip.eu-gb.assistant.watson.cloud.ibm.com
London	public.0003.voip.eu-gb.assistant.watson.cloud.ibm.com
Seoul	public.0001.voip.kr-seo.assistant.watson.cloud.ibm.com
Sydney	public.0001.voip.au-syd.assistant.watson.cloud.ibm.com
Sydney	public.0002.voip.au-syd.assistant.watson.cloud.ibm.com
Sydney	public.0003.voip.au-syd.assistant.watson.cloud.ibm.com

Tokyo	public.0001.voip.jp-tok.assistant.watson.cloud.ibm.com
Tokyo	public.0002.voip.jp-tok.assistant.watson.cloud.ibm.com
Tokyo	public.0003.voip.jp-tok.assistant.watson.cloud.ibm.com
Washington, DC	public.0001.voip.us-east.assistant.watson.cloud.ibm.com
Washington, DC	public.0002.voip.us-east.assistant.watson.cloud.ibm.com
Washington, DC	public.0003.voip.us-east.assistant.watson.cloud.ibm.com

#### SIP network information

## Migrating from Voice Agent with Watson

If you created an IBM® Voice Agent with Watson service instance in IBM Cloud to enable customers to connect to an assistant over the phone, use the phone integration instead. You can use the same SIP account and phone number that you configured for use with Voice Agent with Watson in the phone integration.

The phone integration provides a more seamless integration with your assistant. However, the integration currently does not support the following functions:

- Outbound calling
- Configuring backup locations
- Event forwarding to save call detail reports in the IBM Cloudant for IBM Cloud database service
- Reviewing the usage summary page. Use IBM Log Analysis instead. For more information, see [Viewing logs](#).

To migrate from Voice Agent with Watson to the Watson Assistant phone integration, complete the following steps:

1. From the Voice Agent with Watson page, copy the phone number or numbers that you used for your SIP account.
2. When you set up the Watson Assistant phone integration, add the phone number or set of numbers that you copied in the previous step.
3. From the phone integration setup page, copy the *SIP uniform resource identifier (URI)*.
4. In your SIP trunk account, replace the Voice Agent with Watson URI that you specified previously with the URI that you copied from the phone integration setup page in the previous step.

For example, if you use a Twilio SIP trunk, you would add the assistant's *SIP uniform resource identifier (URI)* to the Twilio *Origination SIP URI* field.

## Call routing details

Incoming calls to your assistant follow this path:

1. A customer calls the customer support phone number that is managed by your Session Initiation Protocol (SIP) trunk provider.
2. The SIP trunk service sends a SIP **INVITE** HTTP request to your assistant's phone integration to establish a connection.
3. The phone integration connects to the speech services that are required to support the interaction.
4. After the services are ready, the connection is established, and audio is sent over the Real-time Transport Protocol (RTP).

RTP is a network protocol for delivering audio and video over IP networks.

5. The welcome node of the dialog is processed. The response text is sent to the Text to Speech service to be converted to audio and the audio is sent to the caller.
6. When the customer says something, the audio is converted to text by the Speech to Text service and is sent to your assistant's dialog skill for evaluation.
7. The dialog processes the input and calculates the best response. The response text from the dialog node is sent to the Text to Speech service to be converted to audio and the audio is sent back to the caller over the existing connection.
8. If the caller asks to speak to a person, the assistant can transfer the person to a call center. A SIP **REFER** request is sent to the SIP trunk provider so it can transfer the call to the call center SIP URI that is specified in the dialog node where the transfer action is configured.
9. When one of the participants of the call hangs up, a SIP **BYE** HTTP request is sent to the other participant.

## Phone integration limits

Any speech service charges that are incurred by the phone integration are included as *Voice add-on* charges in your Watson Assistant service plan usage. The Voice add-on use is charged separately and in addition to your service plan charges.

Plan usage is measured based on the number of monthly active users, where a user is identified by the caller's unique phone number. An MD5 hash is applied to the phone number and the 128-bit hash value is used for billing purposes.

The number of concurrent calls that your assistant can participate in at one time depends on your plan type.

Plan	Concurrent calls
Enterprise	1,000
Plus	100
Trial	5

#### Plan details

## Troubleshooting the phone integration

Find solutions to problems that you might encounter while using the integration.

- If you get a *Forbidden* message, it means the phone number that you specified when you configured the integration cannot be verified. Make sure the number fully matches the SIP trunk phone number.

## Viewing logs

The log events that occur in the components that are used by the phone integration are written to IBM Log Analysis. To check the logs, create an instance and configure the platform logs to observe the region where your service instance is hosted.

For more information about setting up an instance, see [Provisioning an instance](#).



**Note:** Currently, the Phone and SMS with Twilio integrations are the only components of your assistant that write logs to the IBM Log Analysis dashboard.

After you create the instance, get log information by completing the following steps:

1. Go to the [IBM Cloud Logging](#) page.
2. Click **Options**, then choose **Edit platform**.
3. Select the region and instance, and then click **Select**.
4. To open the IBM Log Analysis console, click **Open Dashboard**.
5. The source name of the log events is *Watson*.

You can apply filters or search the logs by values such as a phone number or instance ID.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with phone and Twilio Flex

You can use the phone integration to help your customers over the phone and transfer them to live agents inside of Twilio Flex. If, in the course of a conversation with your assistant, a customer asks to speak to a person, you can transfer the conversation directly to a Twilio Flex agent.

### Before you begin

To use this integration pattern, make sure you have the following:

- Watson Assistant Plus or Enterprise Plan (required for phone integration)
- A Twilio account with the following products:
  - Twilio Flex
  - Twilio Voice with Programmable Voice API
  - Twilio Studio

### Adding the Watson Assistant phone integration

If you have not already added the phone integration to your assistant, follow these steps:

1. From the Assistants page, click to open the assistant tile that you want to deploy.
2. From the Integrations section, click **Add integration**.
3. Click **Phone**.
4. Click **Create**.
5. Click **Close**.

For now, this is all you need to do. For more information about configuring the phone integration, see [Integrating with phone](#).

## Adding the Twilio Flex Project

If you don't already have a Twilio Flex project, you can create one by following these steps:

1. From the project drop-down menu, click **Create New Project**. Specify a name for the project and verify your account information.
2. On the welcome page, select Flex as the Twilio product for your new project. Fill out the questionnaire and then click **Get Started with Twilio**.  
After your Flex project has been provisioned, return to the [Twilio console](#). Make sure you have selected the correct project from the drop-down list.
3. In the navigation menu, click the **All Products & Services** icon.
4. Click **Twilio Programmable Voice > Settings > General**.
5. Under **Enhanced Programmable SIP Features**, toggle the switch to **Enabled**.

## Creating the call flow

After you have your phone integration and Twilio Flex project configured, you must create a call flow with Twilio Studio and provision (or port) the phone number you want your assistant to work with.

To create the call flow:

1. In the navigation menu, click the **All Products & Services** icon.
2. Click **Studio**.
3. Click **+** to create a new flow.
4. Name the new flow and then click **Next**.
5. Select **Start From Scratch** and then click **Next**.
6. At this point you should have a **Trigger** widget at the top of your flow canvas.
7. Click the **Trigger** widget.
8. Make note of the value from the **WEBHOOK URL** field. You will need this value in a subsequent step.

## Configuring the phone number

1. In the navigation menu, click the **All Products & Services** icon.
2. Click **Phone Numbers**.
3. Under **Manage Numbers**, configure the phone number you want your assistant to use. Select **Buy a Number** to buy a new number, or **Port & Host** to port an existing phone number.
4. In the **Active Numbers** list, click the new phone number.
5. Under **Voice and Fax**, configure the following settings:
  - For **CONFIGURE WITH** field, select **Webhook, TwiML Bins, Functions, Studio, or Proxy**.
  - For **A CALL COMES IN**, select **Studio Flow**. Select your flow from the drop-down list.
  - For **PRIMARY HANDLER FAILS**, select **Studio Flow**. Select your flow from the drop-down list.
6. Go to the Watson Assistant user interface, open the phone integration settings for your assistant.
7. In the **Phone number** field, type the phone number you configured in Flex Studio.
8. Click **Save and exit**.

## Test your phone number

You can now test that your phone number is connected to your flow by triggering a **Say/Play** widget in the Twilio Flex Flow editor.

1. Drag a **Say/Play** widget onto your flow canvas.
2. Configure the Say/Play widget with a simple phrase like `I'm alive.`.
3. Connect the **Incoming call** node on your **Trigger** widget to your **Say/Play** widget.
4. Call your phone number. You should hear your Twilio flow respond with your test phrase.
5. Delete the **Say/Play** widget and continue to the next step.
6. If this test did not work as expected, double check your phone number configuration to make sure its attached to your flow.

## Creating a Twilio function to handle incoming calls

Now we need to configure the call flow to direct inbound calls to the assistant using a Twilio function. Follow these steps:

1. In the navigation menu, click the **All Products & Services** icon.
2. Click **Services**.
3. Click **Create Service**. Specify a service name and then click **Next**.
4. Click **Add > Add Function** to add a new function to your service. Name the new function `/receive-call`.
5. Replace the template in your `/receive-call` function with the following code:

```
$ exports.handler = function(context, event, callback) {
 const VoiceResponse = require('twilio').twiml.VoiceResponse;
 const response = new VoiceResponse();
 const dial = response.dial({
 answerOnBridge: "true",
 referUrl: "/refer-handler"
 });
 const calledPhoneNumber = event.Called;
 dial.sip(`sip:${calledPhoneNumber}@{sip_uri_hostname};secure=true`);
 return callback(null, response);
}
```

- o Replace `{sip_uri_hostname}` with the hostname portion of your assistant's phone integration SIP URI (everything that comes after `sip:`). Note that Twilio does not support **SIPS** URIs, but does support secure SIP trunking by appending `;secure=true` to the SIP URI.

6. Click **Save**.
7. Click **Deploy All**.

## Redirecting to the incoming call handler

In this section you will use a TwiML **Redirect**\*\* widget in your Studio Flow editor to call out to the `/call-recvieve` function created in the previous section.

1. Add a **TwiML Redirect** widget to your Studio Flow canvas.
2. Connect the Incoming Call trigger to your **TwiML Redirect** widget.
3. Configure the **TwiML Redirect** widget with the URL for the `/receive-call` function you created in the previous section.
4. Your flow should now redirect to Watson Assistant when receiving an inbound call.
5. If the redirect fails, make sure you deployed your `/receive-call` function.

## Creating a Twilio function to handle transfers from assistant

We also need to configure the call flow to handle calls being transferred from the assistant back to Twilio Flex, for cases when customers ask to speak to an agent. To show this, we will use a **Say/Play** after the **TwiML Redirect** widget to show that the call is transferred back to the flow from Watson Assistant. Note that there are many things like queuing the call for a live agent that can happen at this point. These will be discussed below.

1. Add a new **Say/Play** widget to your canvas and configure it with a phrase like `Transfer from Watson complete.`.
2. Connect the **Return** node on the **TwiML Redirect** widget to your **Say/Play** widget.
3. Click the **Trigger** widget.
4. Copy the value from the **WEBHOOK URL** field. You will need this value in a subsequent step.

5. On the Twilio Functions page, click **Add > Add Function** to add another new function to your service. Name this new function `/refer-handler`.

6. Replace the template in your `/refer-handler` function with the following code:

```
$ exports.handler = function(context, event, callback) {
 // This function handler will handle the SIP REFER back from the Watson Assistant Phone Integration.
 // Before handing the call back to Twilio, it will extract the session history key from the
 // User-to-User header that's part of the SIP REFER Refer-To header. This session history key
 // is a string that is used to load the agent application in order to share the transcripts of the caller
 // with Watson Assistant to the agent.
 // See https://github.com/watson-developer-cloud/assistant-web-chat-service-desk-starter/blob/main/docs/AGENT_APP.md
 const VoiceResponse = require('twilio').twiml.VoiceResponse;

 const STUDIO_WEBHOOK_URL = '{webhook_url}';

 let studioWebhookReturnUrl = `${STUDIO_WEBHOOK_URL}?FlowEvent=return`;

 const response = new VoiceResponse();
 console.log("ReferTransferTarget: " + event.ReferTransferTarget);

 const referToSipUriHeaders = event.ReferTransferTarget.split("?")[1];
 console.log(referToSipUriHeaders);
 if (referToSipUriHeaders) {
 const sanitizedReferToSipUriHeaders = referToSipUriHeaders.replace(">", "");
 console.log("Custom Headers: " + sanitizedReferToSipUriHeaders);

 const sipHeadersList = sanitizedReferToSipUriHeaders.split("&");

 const sipHeaders = {};
 for (const sipHeaderSet of sipHeadersList) {
 const [name, value] = sipHeaderSet.split('=');
 sipHeaders[name] = value;
 }

 const USER_TO_USER_HEADER = 'User-to-User';

 // Extracts the User-to-User header value
 const uuidData = sipHeaders[USER_TO_USER_HEADER];

 if (uuidData) {
 const decodedUUIData = decodeURIComponent(uuidData);
 const sessionHistoryKey = decodedUUIData.split(';')[0];
 // Passes the session history key back to Twilio Studio through a query parameter.
 studioWebhookReturnUrl = `${studioWebhookReturnUrl}&SessionHistoryKey=${sessionHistoryKey}`;
 }
 }

 response.redirect(
 { method: 'POST' },
 studioWebhookReturnUrl
);

 // This callback is what is returned in response to this function being invoked.
 // It's really important! E.g. you might respond with TWiML here for a voice or SMS response.
 // Or you might return JSON data to a studio flow. Don't forget it!
 return callback(null, response);
}
```

Replace `{webhook_url}` with the **WEBHOOK URL** value you copied from the **Trigger** widget in your Studio Flow.

7. Click **Save**.

8. Click **Deploy All**.

9. After you create this refer-handler, copy the function URL back into the `/receive-call` handler's **referUrl** field.

## Configuring the assistant to transfer calls to Twilio Flex

Now we need to configure the assistant to transfer calls to Twilio Flex when a customer asks to speak to an agent. Follow these steps:

1. In the Watson Assistant user interface, open the dialog skill of your assistant.
2. Add a node with the condition you want to trigger your assistant to transfer customers to an agent.
3. Add a text response to the node, and specify the text you want your assistant to say to your customers before it transfers them to an agent.
4. Open the JSON editor for the response.

5. In the JSON editor, add a [connect\\_to\\_agent response](#), specifying your phone number as the `sip.uri` (replace `{phone_number}` with the phone number of your SIP trunk):

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "connect_to_agent",
 "transfer_info": {
 "target": {
 "service_desk": {
 "sip": {
 "uri": "sip:{phone_number}@flex.twilio.com",
 "transfer_headers_send_method": "refer_to_header"
 }
 }
 }
 },
 "agent_available": {
 "message": "Ok, I'm transferring you to an agent"
 },
 "agent_unavailable": {
 "message": ""
 }
 }
]
 }
}
```

Note that this example does not show how to use the context passed from Watson Assistant to Twilio Flex. You can reference the User-to-User information from within the Twilio Flex flow as follows:

```
$ {
 "context": {
 "widgets": {
 "redirect_1": {
 "User-to-User": "value"
 }
 }
 }
}
```

where `redirect_1` is the name of your redirect widget. For example, if you set up multiple queues, you might want to use a Twilio Split widget to pick a queue based on the returned context.

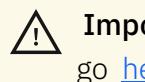
## Test your assistant

Your assistant should now be able to answer phone calls to your phone number and transfer calls back to your Twilio Flex flow. To test your assistant:

1. Call your phone number. When the assistant responds, ask for an agent.
2. At this point you should hear the phrase configured in the **Say/Play** widget (such as "Transfer from Watson complete").
3. If the transfer fails, use the console log to follow the flow of the call as it moves from the flow to the `/call-receive` handler, to Watson Assistant, to the refer-handler and back to your Twilio Flex flow.

## Share the conversation history with service desk agents

To enable the service desk agent to get a quick view of the conversation history between the visitor and the assistant, set up the Watson Assistant Agent App app for your Twilio Flex environment. For more information, see the documentation for the [Twilio Flex Watson Assistant Agent App](#).



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with phone and Genesys Cloud

You can use the phone integration to help your customers over the phone and transfer them to live agents inside of Genesys Cloud. If, in the course of a conversation with your assistant, a customer asks to speak to a person, you can transfer the conversation directly to a Genesys Cloud agent.

### Before you begin

To use this integration pattern, make sure you have the following:

- Watson Assistant Plus or Enterprise Plan (required for phone integration)
- A working assistant you are ready to deploy
- A Genesys Cloud account

## Integrating with Genesys Cloud

To integrate your assistant with Genesys Cloud, follow these steps:

1. Log into the [Genesys Cloud console](#).
2. Click **Admin**.
3. On the **Telephony** tab, click **Trunks**.
4. In the **External Trunks** section, click **Create new**. Specify the following information:
  - In the **External Trunk Name** field, type a descriptive name (for example, **Watson**).
  - In the **Type** field, select **BYOC Carrier** and then **Generic BYOC Carrier**.
  - In the **Inbound SIP Termination Identifier** field, specify any name you want to use (for example, **Watson**). This value will not be used for now, but it is required by Genesys Cloud.
  - In the **Protocol** field, select **TLS**.



Topology

Metrics

**Trunks**

Sites

Edge Groups

Edges

Phone Management

Certificate Authorities

DID Numbers

Extensions

**External Trunk Name**

WatsonDoc

**Type**

BYOC Carrier

Generic BYOC Carrier

**Managed By**

Everyone

Provider Only

**Trunk State**

In Service

**Inbound / Termination****Inbound SIP Termination Identifier**

WatsonDoc

**DNIS Replacement Routing**

Disabled

5. Under **Outbound**, scroll to the **SIP Servers or Proxies** section. Specify the following information:

- In the **Hostname or IP Address** field, type the SIP URI (not including `sips:`) from your Watson Assistant phone integration settings.
- In the **Port** field, type `5061`.

Click the **+** button.



**Note:** Currently, SIPS and digest authentication are supported.



Topology

Metrics

**Trunks**

Sites

Edge Groups

Edges

Phone Management

Certificate Authorities

DID Numbers

Extensions

**External Trunk Name**

WatsonDoc

**Type**

BYOC Carrier

Generic BYOC Carrier

**Managed By**

Everyone

Provider Only

**Trunk State**

In Service

**Inbound / Termination****Inbound SIP Termination Identifier**

WatsonDoc

**DNIS Replacement Routing**

Disabled

- Under **SIP Access Control**, add the IP addresses for the data center where your assistant is located:

Data center

IP addresses

US-South	67.228.108.82 169.63.5.162 150.239.30.146
US-East	52.216.100.158 169.61.70.162 169.59.136.194
EU-DE	161.156.178.162 169.50.56.146 149.81.86.82
EU-GB	158.176.120.162 141.125.102.34 158.175.99.34
AU-SYD	168.1.47.2 135.90.86.50 168.1.106.130
JP-TOK	165.192.69.82 128.168.105.178 161.202.149.162

7. Under **Identity**, specify the following information:

- Toggle the **Address Omit + Prefix** switch to **Disabled**.

### Calling

#### Address Transformation

Match Regular Expression

Format Regular Expression

No Transformations

Match Regular Expression

Format Regular Expression



#### Address Digits Length

0

#### Address Omit + Prefix

Disabled

8. Under **Media**, remove **Opus** from the **Preferred Codec List**. Click **Select a Codec** and then select **g729** to add it to the list. Leave **PCMU** as the first item in the list.

Media

DSCP Value <a href="#">?</a>	Media Method <a href="#">?</a>
2E (46, 101110) EF	Normal
<b>⚠ G.729 is not recommended over WAN links that don't guarantee QoS. (ex: the Internet)</b>	
<b>Preferred Codec List <a href="#">?</a></b>	<b>SRTP Cipher Suite List <a href="#">?</a></b>
audio/PCMU audio/PCMA audio/g729	AES_CM_128_HMAC_SHA1_80
Select a Codec	Select a Cipher Suite

9. Under **Protocol**

- In the **Header/Invite** section, toggle the **Conversation Headers** switch to **Enabled**
- Enable **Take Back and Transfer**.

10. Click **Save External Trunk**.

11. Under **Sites**, select the existing site you want to use this trunk with. To create a new site, specify a name and location and click **Create**.

12. Click **Number plans**. Create a new number plan and specify the following information:

- In the **Number Plan Name** field, type a descriptive name (for example, **Watson**).
- For **Match type**, select **E. 164 Number List**.
- In the **Numbers** field, type a number in the **Start** and **End** fields. This does not need to be a real number; you can make up any number to use as an identifier to assign to Watson. Specify the same number in both fields.



**Note:** To create a PSTN number you can give to your clients, you must create a Direct Inward Dialing (DID) or Bring Your Own Carrier (BYOC) number. For more information about how to do this, see the Genesys documentation.

- In the **Classification** field, type a classification name (for example, **Watson**).

Click **Save Number Plans**.

[+ New Number Plan](#)

 [Delete Number Plan](#)

**WatsonDoc**

**Emergency**

**Extension**

**National**

**International**

**Network**

**Number Plan Name**  
WatsonDoc

**Match Type**  
E.164 Number List

**Numbers**

+1 408-981-3165 → +1 408-981-3165 

**Start Number** → **End Number** 

**Classification**   
Watson

**Save Number Plans** **Cancel**

13. Click **Outbound Routes**. You can either edit the default outbound route or create a new one. Specify the following information:

- In the **External Trunks** field, click **Select External Trunks**. Select the trunk you created for Watson Assistant.
- In the **Classifications** field, add the applicable classifications. This should at least include **National** and the classification you created for Watson Assistant earlier. (The **National** route is used only to simulate the call in order to make sure the trunk is operational.)
- Toggle the **State** switch to **Enabled**.

## **+ New Outbound Route**

**Default Outbound Route**

**Outbound Route Name**

Default Outbound Route

**Description**

**State**

**Enabled**

**Classifications**

Emergency 

National 

International

Network 

Watson 

**Save Outbound Routes**

**Cancel**

14. Click **Save Outbound Routes**.

15. Go to the **Simulate Call** tab and click the **Simulate Call** button. The trunk should be shown as operational. (No actual call is made during simulation.)

- i** Simulate call will use settings from the "General", "Number Plans", and "Outbound" tabs. It is recommended that you test before applying the changes.

+14089813165

Simulate Call

✓ Success

**Normalized Number** ?

✓ tel:+14089813165

**Number Plan** ?

✓ WatsonDoc

**Classification** ?

✓ Watson

**Outbound Route** ?

✓ Default Outbound Route

**External Trunks** ?

WatsonDoc

✓ This Trunk is operational

**Preferred Edges** ?

None

**Additional Edges**

- virtual-edge-i-0e13
- virtual-edge-i-00a8

16. Go to **Phone Management** and click **Create new**. Specify the following information:

- In the **Phone Name** field, specify a descriptive name.
- In the **Base Settings** field, select **WebRTCPhone**.
- In the **Site** field, select the site you want to use.
- In the **Person** field, select yourself.

17. In the Watson Assistant user interface, create a new phone integration. Specify the following information:

- When prompted, select **Use an existing phone number with an external provider**.
- Specify the phone number you assigned in the Genesys **Number Plans** setting. (Remember, this is not necessarily a real phone number; it is just the identifier you assigned.)
- Complete the phone integration setup process. (For more information, see [Integrating with phone](#).)
- After the phone integration is setup, go to the **SIP trunk** tab and uncheck the **Don't place callers on hold while transferring to a live agent** option.

18. In the Genesys Cloud console, click the circle in the upper left corner. Select **Phone**, and then choose the phone you created in the **Phone management** section. Set yourself as available. The phone icon on the left should now be active.
19. Click **+** to start a new call. Specify the number you assigned to Watson Assistant and then click **Dial**. You should now hear your assistant speak.!



**Note:** If you encounter any errors, click **Performance -> Interactions** and view the PCAP file to read the diagnostics.

## Transferring to an agent

Now that your Genesys Cloud environment can connect to Watson Assistant, you can set up the ability for your assistant to transfer calls back to your human agents. To do so, follow these steps:

1. In the Genesys Cloud console, go to **DID Numbers -> DID Ranges** and create a new range. Specify the following information:
  - o In the **DID Start** and **DID End** fields, specify a phone number. (Once again, you do not need to use a real phone number; you can just make up an identifier for your Genesys environment, such as **1-888-888-1234**.)

Provider	Comments	
		<span style="font-size: 2em;">□</span>

Create Range

DID Start

USA +1 +18888881234

DID End

USA +1 +18888881234

Service Provider

Watson

Comments

1 – 1 of 1 DID Ranges

Save Cancel

- In the **Service Provider** field, type a descriptive name (for example, **Watson**).
2. If you have not already set up a queue to enable callers to wait for available agents, follow these steps to create a simple one now:
    1. Click **Admin**.
    2. Under **Contact Center**, click **Queues**.
    3. Create a new queue and give it a descriptive name.
    4. Add yourself as a member.
    5. Click **Save**.
  3. Create a simple call flow. (Your business might already have something more complex for routing.)
    1. Click **Admin**.

2. Click **Architect**.

3. In the **Flows: Inbound Call** section, click **+** to create a new flow. Give it a descriptive name (for example, **Escalate to Agent**).

The screenshot shows the Watson Assistant Architect interface. At the top, there's a toolbar with options like Save, Version 1.0, Validate, Print, and Publish. Below the toolbar is a navigation bar with sections: Starting Menu, Toolbox, and Main Menu. The Starting Menu section shows a tree view with '10 Main Menu' expanded, containing 'Disconnect'. The Toolbox section shows icons for Dial By Extension, Disconnect, Menu, Task, and Transfer. The Main Menu section on the right is titled '10 Main Menu' and contains several configuration panels: 'Initial Greeting' (Hello, this is the initial greeting), 'Menu Prompt' (You are at the Main Menu, press 9 to c), 'Default Menu Choice' (None ( disconnect the interaction )), 'Menu Options' (link), and 'Speech Recognition Options' (link). A sidebar on the left lists Settings (Actions, Event Handling, Menus, Supported Languages, Speech Recognition) and Resources (Data, Prompts, Dependencies). A Reusable Menus section at the bottom has a placeholder 'Add reusable menu here'.

1. In the toolbox, click **Task** and drag it into **Reusable Tasks**.

Starting Menu

- 10 Main Menu
  - 11 Disconnect

Settings

- Actions
- Event Handling
- Menus
- Supported Languages
- Speech Recognition

Resources

- Data
- Prompts
- Dependencies

Reusable Menus

Add reusable menu here

Reusable Tasks

12 New Task 1

Add reusable task here

Toolbox

- Audio
- Call Common Module
- Data
- Dial By Extension
- Disconnect
- External Contacts
- Find
- Flow
- Logical
- Loop
- Menu
- Task
  - Task

Jump to Reusable

12 New Task 1

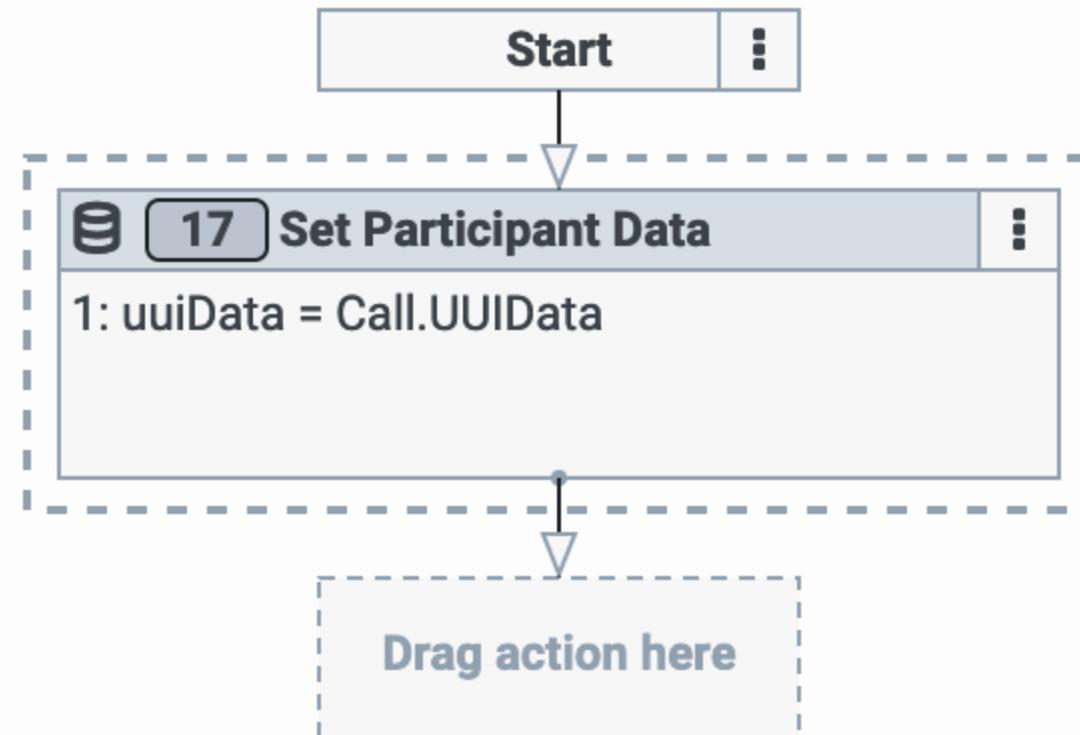
- From your toolbox, under **Data**, drag a **Set Participant Data** widget into the first action. Click on the widget and specify an attribute named **uuiData**. For the value to assign, choose the **Expression** type and specify **Call.UUIData**.



## 12 New Task 1 ?

### Initial Greeting

Enter text to speech or prompt



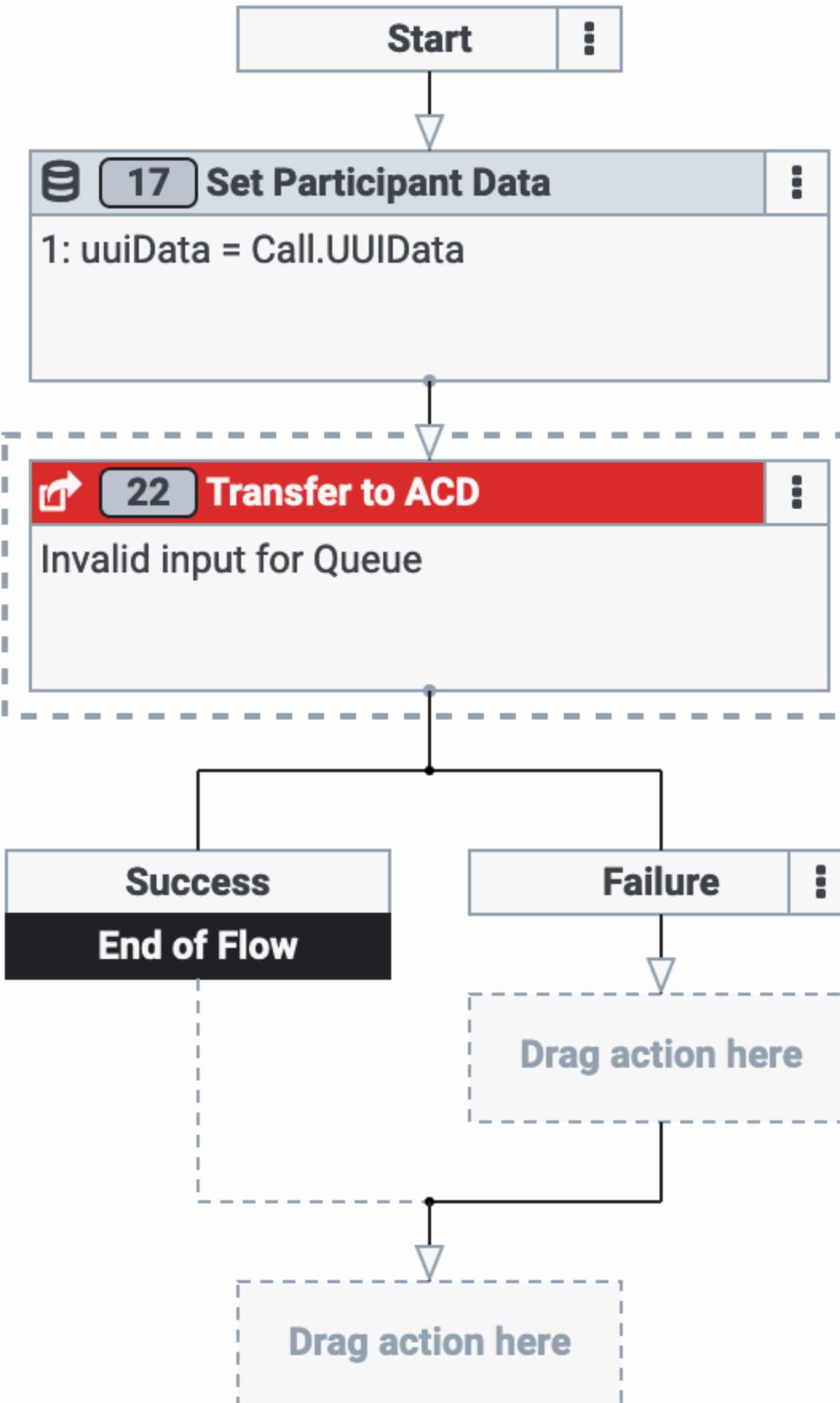
1. From your toolbox, under **Transfer**, drag the **Transfer to ACD** widget into the first action.



## 12 New Task 1 ?

### Initial Greeting

Enter text to speech or prompt



1. Select the queue you want to use.
2. In the toolbox, click the **Disconnect** widget and drag it into the action after **Failure**. (This disconnects the call if the transfer fails.)
3. Click the three dots under **Reusable tasks** and click **Set this as the starting task**. You can remove the **Initial Greeting**, because your assistant will speak before handing off the call.
4. Click **Publish** in the menu bar to make this transfer live.
5. Return to the main Genesys Cloud console.
6. Click **Admin** and then navigate to **Call Routing** in the **Routing** section.

7. Give the route a descriptive name (for example, [Escalate to Agent](#)).
  8. Under **Regular Routing**, for all calls, select your new flow.
  9. Assign the DID number you previously created.
  10. Click **Save**.
4. Make sure your assistant is configured with at least one dialog node that transfers calls to an agent. For more information about how to do this, see [Transferring a call to a human agent](#).

For the `transferTarget` parameter, use the DID number you created in Genesys Cloud, as well as the inbound SIP URI from your Genesys trunk. Use the following format:

```
sip:+18883334444\\@example.com
```

 **Note:** Make sure you use the `\\" escape characters so Watson Assistant does not misinterpret the @ as part of the entity shorthand syntax.`

5. Make a test call and say something that initiates a transfer to an agent. In your Genesys Cloud console, you should see the transfer take place.

## Share the conversation history with service desk agents

To enable the service desk agent to get a quick view of the conversation history between the visitor and the assistant, set up the Watson Assistant Agent App app for your Genesys Pure Cloud Environment. For more information, see the documentation for the [Genesys starter kit](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Handling phone interactions](#).

## Handling phone interactions

If your assistant uses the phone integration, you can use various response types to customize the behavior of the integration or manage the flow of conversations that your assistant has with customers over the telephone.

You can use response types to perform the following phone-specific actions:

- [Apply advanced settings to the Speech to Text service](#)
- [Apply advanced settings to the Text to Speech service](#)
- [Transfer a call to a human agent](#)
- [Play hold music or a voice recording](#)
- [Enable keypad entry](#)
- [Transfer the conversation to the web chat integration](#)
- [End the call](#)
- [Send a text message during a phone conversation](#)

In some cases, you might want to combine response types to perform multiple actions. For example, you might want to implement two-factor authentication by requesting phone keypad entry and sending a text message from the same dialog node or step. For more information, see [Defining a sequence of phone actions](#).

For reference information about phone-specific repsonse types and related context variables, see [Phone context variables](#).

## Adding phone-specific responses to your dialog or actions

To initiate a voice-specific interaction from a dialog node or a step in an action, add a response within the `output.generic` array using the appropriate response type.

 **Note:** Although many response types can be specified using the Watson Assistant user interface, phone-specific response types must currently be added using the JSON editor.

For more information about using the JSON editor to add responses, see [Defining responses using the JSON editor](#).

## Applying advanced settings to the Speech to Text service

Use the `speech_to_text` response type to send configuration commands to the Speech to Text service instance used by the phone integration. By sending a `speech_to_text` response from a dialog node or action step, you can dynamically change the Speech to Text configuration during a conversation.

By default, any Speech to Text configuration changes you make persist for the remainder of the conversation, or until you update them again. You can change this behavior by specifying the `update_strategy` property of the `parameters` object.

The format of the `speech_to_text` response type is as follows:

```
{
```

```

"output": {
 "generic": [
 {
 "response_type": "speech_to_text",
 "command_info": {
 "type": "<command type>",
 "parameters": {
 "parameter name1": "parameter value",
 "parameter name2": "parameter value"
 }
 }
 }
]
}

```

Each command type along with its related parameters are described in the following sections.

#### `command_info.type : configure`

Dynamically reconfigures the Speech to Text service by applying a set of configuration parameters, which can be based on the dialog or action flow. For example, you might want to choose a particular customization ID or grammar at a specific point in the conversation.

parameter	description	required	default
<code>narrowband_recognize</code>	The Speech to Text service configuration to use for narrowband codecs (such as PCMU and PCMA, which are sampled at 8 kHz). The parameters defined by this object are used when connecting to the Speech to Text service for speech recognition requests. For more information about these parameters, see the <a href="#">Speech to Text API documentation</a> .	no	Current Speech to Text configuration
<code>broadband_recognize</code>	The Speech to Text service configuration to use for broadband codecs (such as G722, which is sampled at 8 kHz). The parameters defined by this object are used when connecting to the Speech to Text service for speech recognition requests. For more information about these parameters, see the <a href="#">Speech to Text API documentation</a> .	no	Current Speech to Text configuration
<code>band_preference</code>	Specifies which audio band ( <code>narrowband</code> or <code>broadband</code> ) is preferred when negotiating audio codecs for the session. Set to <code>broadband</code> to use broadband audio when possible.	no	<code>narrowband</code>
<code>update_strategy</code>	Specifies the update strategy to use when setting the speech configuration. Possible values include: <ul style="list-style-type: none"> <li>• <code>replace</code>: Replaces the configuration for the rest of the session. Any root-level fields in the new configuration completely overwrite the previous configuration.</li> <li>• <code>replace_once</code>: Replaces the configuration only for the next turn of the conversation. Subsequently, the previous configuration is used.</li> <li>• <code>merge</code>: Merges the new configuration with the existing configuration for the rest of the session. Only changed parameters are overwritten; any other configuration parameters are unchanged.</li> <li>• <code>merge_once</code>: Merges the new configuration with the existing configuration only for the next turn of the conversation. Subsequently, the previous configuration is used.</li> </ul>	no	<code>replace</code>

The parameters that you can set for `narrowband_recognize` and `broadband_recognize` reflect the parameters that are made available by the Speech to Text WebSocket interface. The WebSocket API sends two types of parameters: query parameters, which are sent when the phone integration connects to the service, and message parameters, which are sent as part of the JSON data in the request body. For example, `model` is a query parameter, and `smart_formatting` is a WebSocket message parameter. For a full list of parameters, see the [Speech to Text API documentation](#).

You can define the following query parameters for the phone integration's connection to the Speech to Text service. Any other parameter that you define for `narrowband` or `broadband` is passed through as part of the WebSocket message request.

- `model`
- `acoustic_customization_id`
- `version`
- `x-watson-learning-opt-out`
- `base_model_version`

- `language_customization_id`

The following parameters from the Speech to Text service can't be modified because they have fixed values that are used by the phone integration.

- `action`
- `content-type`
- `interim_results`
- `continuous`
- `inactivity_timeout`



**Note:** When configuring dynamically from Watson Assistant using the `configure` command, note that only the root level fields, such as `narrowband` or `broadband`, are updated. If these fields are omitted from the command, the original configuration settings persist. You can use the `update_strategy` values `merge` and `merge_once` to merge configuration parameters with the existing configuration.

## Using a custom language model

When you set up the phone integration, you can configure the integration to use a custom language model all the time. However, you might want to use a standard language model most of the time, and specify a custom language model to use only for specific topics that your assistant is designed to help customers with. For example, you might want to use a custom model that specializes in medical terms for a dialog branch that helps with medical bills only. You can apply a custom language model for a specific dialog node or action step.

For more information, about custom language models, see [Creating a custom language model](#).

To apply a custom language model to a specific dialog node or action step, use the `speech_to_text` command.

```
{
 "output": {
 "generic": [
 {
 "response_type": "speech_to_text",
 "command_info": {
 "type": "configure",
 "parameters": {
 "narrowband_recognize": {
 "x-watson-learning-opt-out": true,
 "model": "en-US_NarrowbandModel",
 "profanity_filter": true,
 "smart_formatting": true,
 "language_customization_id": "81d3630-ba58-11e7-aa4b-41bcd3f6f24d",
 "acoustic_customization_id": "e4766090-ba51-11e7-be33-99bd3ac8fa93"
 }
 }
 }
 }
]
 }
}
```

You can also apply an acoustic model that you might have trained to deal with background noise, accents, or other things that are associated with the quality or noise of the signal.

## Using a custom grammar

The Speech to Text service supports the use of grammars. A grammar allows you to configure the audio to match specific characteristics only.

You can think of it this way:

- A custom language model expands the service's base vocabulary.
- A grammar restricts the words that the service can recognize from that vocabulary.

When you use a grammar with a custom language model for speech recognition, the service can recognize only words, phrases, and strings that are recognized by the grammar. For example, maybe you want to accept only a yes or no response. You can define a grammar that allows only those options.

For more information, see [Using grammars with custom language models](#).

This example shows how to specify a custom grammar during the conversation:

```
{
 "output": {
 "generic": [
 {
 "response_type": "speech_to_text",
 "command_info": {

```

```
 "type": "configure",
 "parameters": {
 "update_strategy": "merge_once",
 "narrowband_recognize": {
 "x-watson-learning-opt-out": true,
 "grammar_name": "names-abnf",
 "language_customization_id": "81d3630-ba58-11e7-aa4b-41bcd3f6f24d"
 }
 }
 }
}
```

## Examples

The following examples illustrate how to use the `speech_to_text` response type to send configuration commands to the Speech to Text service.

## Example: Setting the language model

In this example, the language model is switched to Spanish ( `es-ES_NarrowbandModel` ), and smart formatting is enabled.

```
{
 "output": {
 "generic": [
 {
 "response_type": "speech_to_text",
 "command_info": {
 "type": "configure",
 "parameters": {
 "narrowband_recognize": {
 "model": "es-ES_NarrowbandModel",
 "smart_formatting": true
 }
 }
 }
 }
]
 }
}
```

**Example: Updating the `recognizeBody` property for one conversation turn**

The following example shows how to specify the use of a custom language model for a single turn of the conversation turn. This is done by setting `update_strategy` to `merge_once` and specifying the ID of the custom language model in the configuration parameters.

```
{
 "output": {
 "generic": [
 {
 "response_type": "speech_to_text",
 "command_info": {
 "type": "configure",
 "parameters": {
 "update_strategy": "merge_once",
 "narrowband_recognize": {
 "language_customization_id": "ao45vohgFuxyOQRgztu-02I10ut7aJcM-AdInT-VWgj3V
 }
 }
 }
 }
]
 }
}
```

## Applying advanced settings to the Text to Speech service

Use the `text_to_speech` response type to send configuration commands to the Text to Speech service instance used by the phone integration. By sending a `text_to_speech` response from a dialog node or action step, you can dynamically change the Text to Speech configuration during a

conversation.

By default, any Text to Speech configuration changes you make persist for the remainder of the conversation, or until you update them again. You can change this behavior by specifying the `update_strategy` property of the `parameters` object.

The format of the `speech_to_text` response type is as follows:

```
$
{
 "output": {
 "generic": [
 {
 "response_type": "text_to_speech",
 "command_info": {
 "type": "<command type>",
 "parameters": {
 "parameter name": "parameter value",
 "parameter name": "parameter value"
 }
 }
 }
]
 }
}
```

Each command type along with its related parameters are described in the following sections.

#### `command_info.type : configure`

Dynamically reconfigures the Text to Speech service by applying a set of configuration parameters, which can be based on the dialog or action flow. For example, you might want to choose a particular voice at a specific point in the conversation.

parameter	description	required	default
<code>synthesize</code>	The Text to Speech service configuration to use when synthesizing audio. The parameters defined by this object are used when connecting to the Text to Speech service for speech synthesis requests. For more information about these parameters, see the <a href="#">Text to Speech API documentation</a> .	yes	Current Text to Speech configuration
<code>update_strategy</code>	Specifies the update strategy to use when setting the speech configuration. Possible values include: <ul style="list-style-type: none"><li>• <code>replace</code>: Replaces the configuration for the rest of the session. Any root-level fields in the new configuration completely overwrite the previous configuration.</li><li>• <code>replace_once</code>: Replaces the configuration only for the next turn of the conversation. Subsequently, the previous configuration is used.</li><li>• <code>merge</code>: Merges the new configuration with the existing configuration for the rest of the session. Only changed parameters are overwritten; any other configuration parameters are unchanged.</li><li>• <code>merge_once</code>: Merges the new configuration with the existing configuration only for the next turn of the conversation. Subsequently, the previous configuration is used.</li></ul>	no	<code>replace</code>

The parameters that you can set for `synthesize` reflect the parameters that are made available by the Text to Speech WebSocket interface. The WebSocket API sends two types of parameters: query parameters, which are sent when phone integration connects to the service, and message parameters, which are sent as part of the JSON data in the request body. For a full list of parameters, see the [Text to Speech API documentation](#).

#### `command_info.type : disable_barge_in`

Disables speech barge-in so that playback isn't interrupted when the caller speaks while audio is being played back.

No parameters.

#### `command_info.type : enable_barge_in`

Enables speech barge-in so that callers can interrupt playback by speaking.

No parameters.

## Changing the assistant's voice

You can change the voice of your assistant when it covers certain topics in the conversation that warrant it. For example, you might want to use a voice with a British accent for a branch of the conversation that applies only to customers in the UK.

This example shows how to specify a voice during the conversation:

```
{
 "output": {
 "generic": [
 {
 "response_type": "text_to_speech",
 "command_info": {
 "type": "configure",
 "parameters": {
 "synthesize": {
 "voice": "en-GB_KateV3Voice"
 }
 }
 }
 }
]
 }
}
```

In the `voice` parameter, specify the voice model that you want to use. For more information about voice model options, see [Supported languages and voices](#).



**Note:** The model you specify must be one that is supported by the Text to Speech service instance that is configured for use with the integration.

## Transferring a call to a human agent

When you configure the phone integration, you can optionally set up backup call center support, which makes it possible for the assistant to transfer a call to a human agent. You can use the *Connect to human agent* response type in a dialog node or action step to initiate a transfer to a human agent at a specific point in the conversation. When a *Connect to human agent* response is sent to the phone integration, a SIP transfer is initiated using the SIP `REFER` message, as defined by [RFC 5589](#).

For more information about initiating a transfer to a human agent during the conversation, see the following documentation:

- [Adding Connect to human agent response type](#) (dialog node)
- [Deciding what to do next](#) (action step)

The phone integration supports additional parameters for the *Connect to human agent* response type. You can add these phone-specific parameters to the `connect_to_agent` response type using the JSON editor.

The `connect_to_agent` response type supports the ability to specify the target transfer information under the `transfer_info` parameter.

The following example shows a transfer that uses all of the configurable parameters:

```
{
 "output": {
 "generic": [
 {
 "response_type": "connect_to_agent",
 "transfer_info": {
 "target": {
 "service_desk": {
 "sip": {
 "uri": "sip:user\\@domain.com",
 "transfer_headers": [
 {
 "name": "Customer-Header1",
 "value": "Some-Custom-Info"
 },
 {
 "name": "User-to-User",
 "value": "XXXXXX"
 }
],
 "transfer_headers_send_method": "refer_to_header"
 }
 }
 }
 }
 }
]
 }
}
```

```

 "agent_available": {
 "message": "I'll transfer you to an agent"
 },
 "agent_unavailable": {
 "message": "Sorry, I could not find an agent."
 },
 "message_to_human_agent": "The caller needs help resetting their password"
 }
}
}

```

The `connect_to_agent` response type supports the following phone-specific properties.

Parameter	Default	Description
<code>service_desk.sip.uri</code>	N/A	The SIP or telephone URI to transfer the call to, such as <code>sip:12345556789\@myhost.com</code> or <code>tel:+18883334444</code>
<code>service_desk.sip.transfer_headers</code>	N/A	A list of custom header field name/value pairs to be added to a transfer request
<code>service_desk.sip.transfer_headers_send_method</code>	<code>custom_header</code>	<p>The method by which the SIP transfer headers are sent:</p> <ul style="list-style-type: none"> <li>• <code>custom_header</code>: Sends the transfer headers as part of the SIP message. This is the default value.</li> <li>• <code>contact_header</code>: Sends the transfer headers in the <code>Contact</code> header.</li> <li>• <code>refer_to_header</code>: Sends the transfer headers in the <code>Refer-To</code> header.</li> </ul>

If you define a SIP URI as the transfer target, escape the at sign (`@`) in the URI by adding two backslashes (`\\"`) in front of it. This is to prevent the string from being recognized as part of the entity shorthand syntax.

```
"uri": "sip:12345556789\\@myhost.com"
```

## Passing Watson Assistant Metadata in SIP Signaling

To support loading the conversational history between the caller and Watson Assistant, the phone integration specifies a value for the `User-to-User` header as a key that can be used with the web chat integration. If `User-to-User` is specified in the `transfer_headers` list, the session history key is sent in the `X-Watson-Assistant-Session-History-Key` header.

The value of the SIP header is limited to 1024 bytes.

How this data is presented in the SIP `REFER` message also depends on the value of `transfer_headers_send_method` (as defined in [Generic Service Desk SIP Parameters](#)).

The following example shows the data included as headers:

```

REFER sip:b@atlanta.example.com SIP/2.0
Via: SIP/2.0/UDP agenta.atlanta.example.com;branch=z9hG4bK2293940223
To: <sip:b@atlanta.example.com>
From: <sip:a@atlanta.example.com>;tag=193402342
Call-ID: 898234234@agenta.atlanta.example.com
CSeq: 23 REFER
Max-Forwards: 7
Refer-To: sip:user@domain.com
X-Watson-Assistant-Token: 8f817472-8c57-4117-850d-fdf4fd23ba7
User-to-User: dev::latest::212033::0a64c30d-c558-4055-85ad-ef75ad6cc29d::978f1fd7-4e24-47d8-adb0-
24a8a6eff69e::b5ffd6c2-902f-4658-b586-e3fc170a6cf3::7ad616a350cc48078f17e3ee3df551de;encoding=ascii
Contact: sip:a@atlanta.example.com
Content-Length: 0

```

If a custom `User-to-User` header is specified, then the session history key is set in the `X-Watson-Assistant-Session-History-Key` header:

```

REFER sip:b@atlanta.example.com SIP/2.0
Via: SIP/2.0/UDP agenta.atlanta.example.com;branch=z9hG4bK2293940223
To: <sip:b@atlanta.example.com>
From: <sip:a@atlanta.example.com>;tag=193402342
Call-ID: 898234234@agenta.atlanta.example.com

```

```
CSeq: 93809823 REFER
Max-Forwards: 70
Refer-To: sip:user@domain.com
User-to-User: 637573746f6d2d757365722d746f2d75736572;encoding=hex;
X-Watson-Assistant-Session-History-Key: dev::latest::212033::0a64c30d-c558-4055-85ad-ef75ad6cc29d::978f1fd7-4e24-47d8-adb0-24a8a6eff69e::b5ffd6c2-902f-4658-b586-e3fc170a6cf3::7ad616a350cc48078f17e3ee3df551de
Contact: sip:a@atlanta.example.com
Content-Length: 0
```

This example shows the metadata passed into the `Refer-To` header as query parameters (as defined by [SIP RFC 3261](#)).

```
REFER sip:b@atlanta.example.com SIP/2.0
Via: SIP/2.0/UDP agenta.atlanta.example.com;branch=z9hG4bK2293940223
To: <sip:b@atlanta.example.com>
From: <sip:a@atlanta.example.com>;tag=193402342
Call-ID: 898234234@agenta.atlanta.example.com
CSeq: 23 REFER
Max-Forwards: 70
Refer-To: sip:user@domain.com?User-to-User=dev::latest::893499::dff9c274-adc4-4f63-93de-781166760bf8::978f1fd7-4e24-47d8-adb0-24a8a6eff69e::b5ffd6c2-902f-4658-b586-e3fc170a6cf3::7ad616a350cc48078f17e3ee3df551de%3Bencoding%3Dascii
Contact: sip:a@atlanta.example.com
Content-Length: 0
```

If a custom `User-to-User` header is specified, then the session history key is set in the `X-Watson-Assistant-Session-History-Key` header.

```
REFER sip:b@atlanta.example.com SIP/2.0
Via: SIP/2.0/UDP agenta.atlanta.example.com;branch=z9hG4bK2293940223
To: <sip:b@atlanta.example.com>
From: <sip:a@atlanta.example.com>;tag=193402342
Call-ID: 898234234@agenta.atlanta.example.com
CSeq: 93809823 REFER
Max-Forwards: 70
Refer-To: sip:user@domain.com?User-to-User=637573746f6d2d757365722d746f2d75736572%3Bencoding%3Dhe&X-Watson-Assistant-Session-History-Key=dev::latest::893499::dff9c274-adc4-4f63-93de-781166760bf8::978f1fd7-4e24-47d8-adb0-24a8a6eff69e::b5ffd6c2-902f-4658-b586-e3fc170a6cf3::7ad616a350cc48078f17e3ee3df551de
Contact: sip:a@atlanta.example.com
Content-Length: 0
```

## Playing hold music or a voice recording

To play hold music or to play a recorded message, use the `audio` response type. This response type can be specified in a dialog node or action step using the JSON editor.

You cannot play hold music during a call transfer. However, you might want to play hold music if your assistant needs time to perform processing of some kind, such as calling a client-side action or making a call to a webhook.

The phone integration supports the following properties for the `audio` response type:

Property	Description
<code>source</code>	The URL of a publicly-accessible <code>.wav</code> audio file. The audio file must be single channel (mono) and PCM-encoded, and must have an 8,000 Hz sampling rate with 16 bits per sample.
<code>channel_options.voice_telephony.loop</code>	Whether to repeatedly restart the audio playback after it finishes. The default value is <code>false</code> .

If you set `channel_options.voice_telephony.loop` to `true`, add a user-defined response with the `vgwActForceNoInputTurn` command. This command instructs the phone integration to initiate a turn with a `vgwNoInputTurn` text without waiting for an input from the caller. In the `vgwNoInputTurn` turn you can initiate a transaction while the caller is on hold. When the `vgwNoInputTurn` turn completes, the looped audio stops.

The following example shows an `audio` response with `loop=true`, and a `user_defined` response specifying the `vgwActForceNoInputTurn` command.

```
{
 "output": {
 "generic": [
 {
 "user_defined": {
 "vgwAction": {
 "command": "vgwActForceNoInputTurn"
 }
 },
],
],
 },
}
```

```

 "response_type": "user_defined"
 },
 {
 "source": "https://upload.wikimedia.org/wikipedia/commons/d/d8/Random_composition3.wav",
 "response_type": "audio",
 "channel_options": {
 "voice_telephony": {
 "loop": true
 }
 }
 }
]
}

```

## Enabling keypad entry

If you want customers to be able to send information by typing it on their phone keypad instead of speaking, you can add support for phone keypad entry. The best way to implement this type of support is to enable dual-tone multifrequency (DTMF) signaling. DTMF is a protocol used to transmit tones that are generated when a user presses keys on a push-button phone. The tones have a specific frequency and duration that can be interpreted by the phone network.

To start listening for tones as the user presses phone keys, use the `dtmf` response type in a dialog node or action step. This response type can be added using the JSON editor.

```

{
 "output": {
 "generic": [
 {
 "response_type": "dtmf",
 "command_info": {
 "type": "<command type>",
 "parameters": {
 "parameter name": "parameter value",
 "parameter name": "parameter value"
 }
 },
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}

```

The `command_info` property specifies a DTMF command for the phone integration. The supported commands and their related parameters are as follows.

`command_info.type : collect`

Instructs the phone integration to collect dual-tone multi-frequency signaling (DTMF) input from a user. This command supports the following parameters:

parameter name	description	required	default
<code>termination_key</code>	The DTMF termination key, which signals the end of DTMF input (for example, <code>#</code> ).	no	n/a
<code>count</code>	The number of DTMF digits to collect. This must be a positive integer no larger than 100.	Required if <code>termination_key</code> , or <code>minimum_count</code> and <code>maximum_count</code> , are not defined	n/a
<code>minimum_count</code>	The minimum number of DTMF digits to collect. This property is used along with <code>maximum_count</code> to define a range for the number of digits to collect. This value must be a positive integer with a minimum value of 1 and a maximum value less than <code>maximum_count</code> .	Required if <code>termination_key</code> and <code>count</code> are not defined.	n/a

<code>maximum_count</code>	The maximum number of DTMF digits to collect. This property is used along with <code>minimum_count</code> to define a range for the number of digits to collect. When this number of digits is collected, a conversation turn is initiated. This value must be a positive integer no greater than 100.	Required if <code>termination_key</code> and <code>count</code> are not defined.	n/a
<code>inter_digit_timeout_count</code>	The amount of time (in milliseconds) to wait for a new DTMF digit after a DTMF digit is received. During an active DTMF collection, this timeout activates when the first DTMF collection is received. When the inter-digit timeout is active, it deactivates the post-response timeout timer. If the <code>inter_digit_timeout_count</code> parameter is not specified, the post-response timer resets after receiving each DTMF digit, and it stays active until either the post-response timeout count is met or the collection completes. This value is a positive integer no higher than 100,000 (or 100 seconds).	no	n/a
<code>ignore_speech</code>	Whether to disable speech recognition during collection of DTMF digits, until either the collection completes or a timeout occurs. If this parameter is <code>true</code> , speech recognition is disabled automatically when the first DTMF signal is received.	no	false
<code>stop_after_collection</code>	Whether to stop DTMF input when the DTMF collection completes. After this command, all DTMF input is ignored until it is reenabled using the <code>start</code> response type.	no	false

#### `command_info.type : disable_barge_in`

Disables DTMF barge-in so that playback from the phone integration is not interrupted when callers press keys. If `disable_barge_in` is enabled, keys pressed during playback are ignored.

This command has no parameters.

#### `command_info.type : enable_barge_in`

Enables DTMF barge-in so that callers can interrupt playback from the phone integration by pressing a key.

This command has no parameters.

#### `command_info.type : send`

Sends DTMF signals using the phone integration.

This command supports the following parameters:

parameter	description	required	default
<code>digits</code>	An array of JSON objects where each element represents a DTMF tone to be sent to a caller.	yes	n/a
<code>digits[] . code</code>	The event code to send. In addition to the digits 0 through 9, you can specify the following codes:	yes	n/a
	<ul style="list-style-type: none"> <li>• 10: *</li> <li>• 11: #</li> <li>• 12: A</li> <li>• 13: B</li> <li>• 14: C</li> <li>• 15: D</li> </ul>		
<code>digits[] . duration</code>	The duration (in milliseconds) of the event.	no	200
<code>digits[] . volume</code>	The power level of the tone, in dBm0. The supported range is 0 to -63 dBm0.	no	0
<code>send_interval</code>	An interval (in milliseconds) to wait before sending the next DTMF tone in the list.	no	200

## Examples

This example shows the `dtmf` response type with the `collect` command, used to collect DTMF input.

```
{
 "output": {
 "generic": [
 {
 "response_type": "dtmf",
 "command_info": {
 "type": "collect",
 "parameters": {
 "termination_key": "#",
 "count": 16,
 "ignore_speech": true
 }
 },
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

This example shows the `dtmf` response type with the `send` command, used to send DTMF signals.

```
{
 "output": {
 "generic": [
 {
 "response_type": "dtmf",
 "command_info": {
 "type": "send",
 "parameters": {
 "digits": [
 {
 "code": "9",
 "volume": -8
 },
 {
 "code": "11"
 }
],
 "send_interval": 100
 }
 },
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

## Transferring the conversation to the web chat integration

You can transfer the caller from the current phone call to a [web chat](#) session by using the `channel_transfer` response type. The assistant sends an SMS message to the caller that includes a URL that the caller can tap to load the web chat widget in the phone's browser. The web chat session displays the history of the phone call and can start the process of collecting information needed to complete the transaction. This can be useful in situations where the customer can provide information more easily in writing than by speaking (for example, changing an address).

After the transfer has successfully completed, the caller can hang up the phone and continue the conversation using the web chat. The `channel_transfer` response type can be used with the phone integration only if the [SMS with Twilio](#) integration is also configured for the assistant.

```
{
 "output": {
 "generic": [
 {
 "response_type": "text",
 "values": [
 {
 "text": "I will send you a text message now with a link to our website."
 }
]
 }
]
 }
}
```

```

 }
],
 "selection_policy": "sequential"
},
{
 "response_type": "channel_transfer",
 "message_to_user": "Click the link to connect with an agent using our website.",
 "transfer_info": {
 "target": {
 "chat": {
 "url": "https://example.com/webchat"
 }
 }
 }
}
]
}
}

```

## Ending the call

You can instruct your assistant end a phone call by using the `end_session` response type, as shown in this example.

```
{
 "output": {
 "generic": [
 {
 "response_type": "end_session"
 }
]
 }
}
```

You can optionally include custom headers to include with the SIP `BYE` request that is generated when the phone integration receives this response type.

This example shows the `end_session` response type with custom SIP headers:

```
{
 "output": {
 "generic": [
 {
 "response_type": "end_session",
 "channel_options": {
 "voice_telephony": {
 "sip": {
 "headers": [
 {
 "name": "Customer-Header1",
 "value": "Some-Custom-Info"
 },
 {
 "name": "User-to-User",
 "value": "XXXXXX"
 }
]
 }
 }
 }
 }
]
 }
}
```

## Sending a text message during a phone conversation

There are some situations when it is useful to be able to send a text message during an ongoing voice. For example, you might want the customer to specify a street address, which is easier to communicate accurately in writing than by transcribing voice input.



**Note:** Before you can send SMS messages during a phone call, you must set up the `SMS with Twilio` integration. For more information, see [Integrating with SMS with Twilio](#).

When you exchange a text with a customer during a conversation, the dialog initiates the SMS message exchange. It sends a text message to the user and asks for the user to respond to it.

To send a specific message from a dialog node or action step, use the `user_defined` response type with the `vgwActSendSMS` command:

```
{
 "output": {
 "generic": [
 {
 "response_type": "text",
 "values": [
 {
 "text": "I will send you a text message now."
 }
],
 "selection_policy": "sequential"
 },
 {
 "response_type": "user_defined",
 "user_defined": {
 "vgwAction": {
 "command": "vgwActSendSMS",
 "parameters": {
 "message": "Hey, this is Watson Assistant. To send me your street address, respond to this text message with your address."
 }
 }
 }
 }
]
 }
}
```

You can specify any of the following parameters in the `parameters` object:

Parameter	Type	Description
message	string	The text of the SMS message to send. Required.
mediaURL	list	A list of URLs for media files to be sent with the message as MMS attachments. Optional.
tenantPhoneNumber	string	The phone number that is associated with the tenant. The format of the number must match the format that is required by the SMS provider. If no <code>tenantPhoneNumber</code> value is provided, the tenant ID from the phone integration configuration for the active call is used. Optional.
userPhoneNumber	string	The phone number to send the SMS message to. The format of the number must match the format that is required by the SMS provider. If no <code>userPhoneNumber</code> value is provided, the voice caller's phone number from <code>From</code> header of the incoming SIP <code>INVITE</code> request is used. Optional.

If your *SMS with Twilio* integration supports more than one SMS phone number, or you are using a non-Twilio SIP trunk, be sure to specify the phone number that you want to use to send the text message. Otherwise, the text is sent using the same phone number that was called.

After the assistant receives an SMS message, a new conversation turn is initiated with the text input `vgwSMSMessage`. This input indicates that a message was received from the caller. The text of the customer's message is included as the value of the `vgwSMSMessage` context variable.

If the assistant is unable to send an SMS message to the caller, a new turn is initiated with the text input `vgwSMSFailed`. This input indicates that an SMS message could not be sent to the caller. You can design your dialog or actions to handle such a failure by creating intents or actions that are triggered by the input text `vgwSMSFailed`.

```
{
 "input": {
 "message_type": "text",
 "text": "vgwSMSMessage"
 },
 "context": {
 "skills": {
 "main skill": {
 "user_defined": {
 "vgwSMSMessage": "1545 Lexington Ave."
 }
 }
 }
 }
}
```

## Defining a sequence of phone commands

If you want to run more than one command in succession, include multiple responses in the `output.generic` array. These commands are processed in the order in which they are specified in the array.

This example shows two responses: first a text response, followed by an `end_session` response to end the call.

```
{
 "output": {
 "generic": [
 {
 "values": [
 {
 "text": "Goodbye."
 }
],
 "response_type": "text",
 "selection_policy": "sequential"
 },
 {
 "response_type": "end_session"
 }
]
 }
}
```

## Deploying to other platforms



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Adding integrations

To deploy your skill, add it to an assistant, and then add integrations to the assistant that publish your bot to the channels where your customers go for help.

### Add an integration

Follow these steps to add integrations to your assistant:

1. Click the **Assistants** icon .
2. Click to open the tile for the assistant that you want to deploy.
3. Go to the Integrations section.

**Why do I have the Web chat integration?** This integration is provisioned and added automatically to your first assistant only.

The web chat is a chat window that you can embed in one or more pages on your website to share your assistant with your customers.

4. Click **Add integration**.
5. Click the tile for the channel with which you want to integrate the assistant. The options include:

- o [Web chat](#)
- o [Facebook Messenger](#)
- o [Slack](#)
- o [Phone](#)
- o [SMS with Twilio](#)
- o [WhatsApp with Twilio](#)
- o [Custom application](#)

Built-in service desk integrations:

- o [Intercom](#)
- o [Web chat with Salesforce support](#)
- o [Web chat with Zendesk support](#)

Web chat service desk reference implementations:

- o [Genesys Cloud](#)
- o [NICE inContact](#)
- o [Twilio Flex](#)
- o [Bring your own \(starter kit\)](#)

- Follow the instructions that are provided on the screen to complete the integration process.

After you integrate the assistant, test it from the target channel to ensure that the assistant works as expected.



**Note:** For environments where private endpoints are in use, keep in mind that these integrations send traffic over the internet. For more information, see [Private network endpoints](#).

## How service desk platform integrations work

Watch a 4-minute video about integrating your assistant with a live agent integration, such as Zendesk:



**View video:** [Zendesk Integration: Watson Assistant](#)

To learn about how service desk integrations with your assistant can benefit your business, [read this blog post](#).

## Integration limits

The number of integrations you can create in a single service instance depends on your Watson Assistant plan.

Service plan	Integrations per assistant
Enterprise	100
Premium (legacy)	100
Plus	100
Trial	100
Lite	100

### Service plan details



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with Facebook Messenger

Facebook Messenger is a mobile messaging application that helps businesses and customers communicate directly with one another.

After you configure a dialog skill and add it to an assistant, you can integrate the assistant with Facebook Messenger.



**Important:** There is currently no mechanism for identifying users who interact with the assistant through Facebook Messenger, which means that there is no way to identify or delete data associated with a specific user. Do not use this integration method for deployments that must be GDPR compliant. See [Information security](#) for more details.

- From the Assistants page, click to open the assistant tile that you want to deploy.
- From the Integrations section, click **Add integration**.
- Click **Facebook Messenger**.
- Follow the instructions that are provided on the screen to complete the integration process.

## Dialog considerations

The rich responses that you add to the dialog are displayed in a Facebook app as expected, with the following exceptions:

- **Connect to human agent**: This response type is ignored.
- **Image**: This response type embeds an image in the response. A title and description are not displayed before the image, whether or not you specify them.
- **Audio**: This response type embeds audio from various file formats in the response. A title and description are not displayed, whether or not you specify them.
- **Video**: This response type embeds a native video from various file formats in the response. A title and description are not displayed, whether or not you specify them.
- **iframe**: This response type inserts an embeddable third-party link in the response that displays interactive content (such as forms or maps). A title is displayed in the preview card. A description is not displayed, whether or not you specify it.
- **Option**: This response type shows a list of options that the user can choose from.
  - A title is **required** and is displayed before the list of options.
  - A description is not displayed, whether you specify one or not.
  - After a user clicks one of the buttons, the button choices disappear and are replaced by the user input that is generated by the user's choice. If the assistant or the user enters new input, then the button-generated input disappears. Therefore, if you include multiple response types in a single response, position the option response type last. Otherwise, content from subsequent responses, such as text from a text response type, will replace the button-generated text.
- **Pause**: This response type pauses the assistant's activity in the Messenger. However, activity does not resume after the pause unless another response type is triggered after it. Whenever you include this response type, add another, different response type, such as a text response, and position it after this one.

See [Rich responses](#) for more information about response types.

## Chatting with the assistant

To start a chat with the assistant, complete the following steps:

1. Open Facebook Messenger.
2. Type the name of the page you created earlier.
3. After the page comes up, click it, and then start chatting with the assistant.



**Note:** The Welcome node of your dialog is not processed by the Facebook Messenger integration. The welcome message is not displayed in the Facebook chat like it is in the "Try it out" pane or in the assistant preview. It is not triggered from here because nodes with the `welcome` special condition are skipped in dialog flows that are started by users. Facebook Messenger waits for the user to initiate the conversation. For more information about how to set context variable values consistently at the start of a conversation, see [Starting the dialog](#).

You can upload media files or even documents to the chat. Files shared in the chat can be intercepted and processed by a configured premessage webhook. For more details, see [Processing input attachments](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Integrating with Intercom](#).

## Integrating with Intercom

Intercom is a customer messaging platform that helps drive business growth through better relationships throughout the customer lifecycle.

Intercom has partnered with IBM to add a new agent to the customer support team, a virtual Watson Assistant. You can integrate your assistant with an Intercom application to enable the app to seamlessly pass user conversations between your assistant and human agents. Read the [Watson blog post](#) to learn more about the integration.

If you integrate the assistant with Intercom, the Intercom application becomes the client-facing application for your skill. All interactions with users are initiated through and managed by Intercom.

## One-time agent creation

You or someone in your organization must complete these one-time prerequisite steps before you add the Intercom integration to your assistant.

1. Create a functional email account for your assistant.  
Each assistant must have a valid, unique email address before it can be added to a team in Intercom.
2. From your Intercom workspace, add the assistant to your team as a new agent.

Go to the teammate settings page in your Intercom workspace, invite the assistant as a new agent by adding the email you created in the previous step to the invite field.

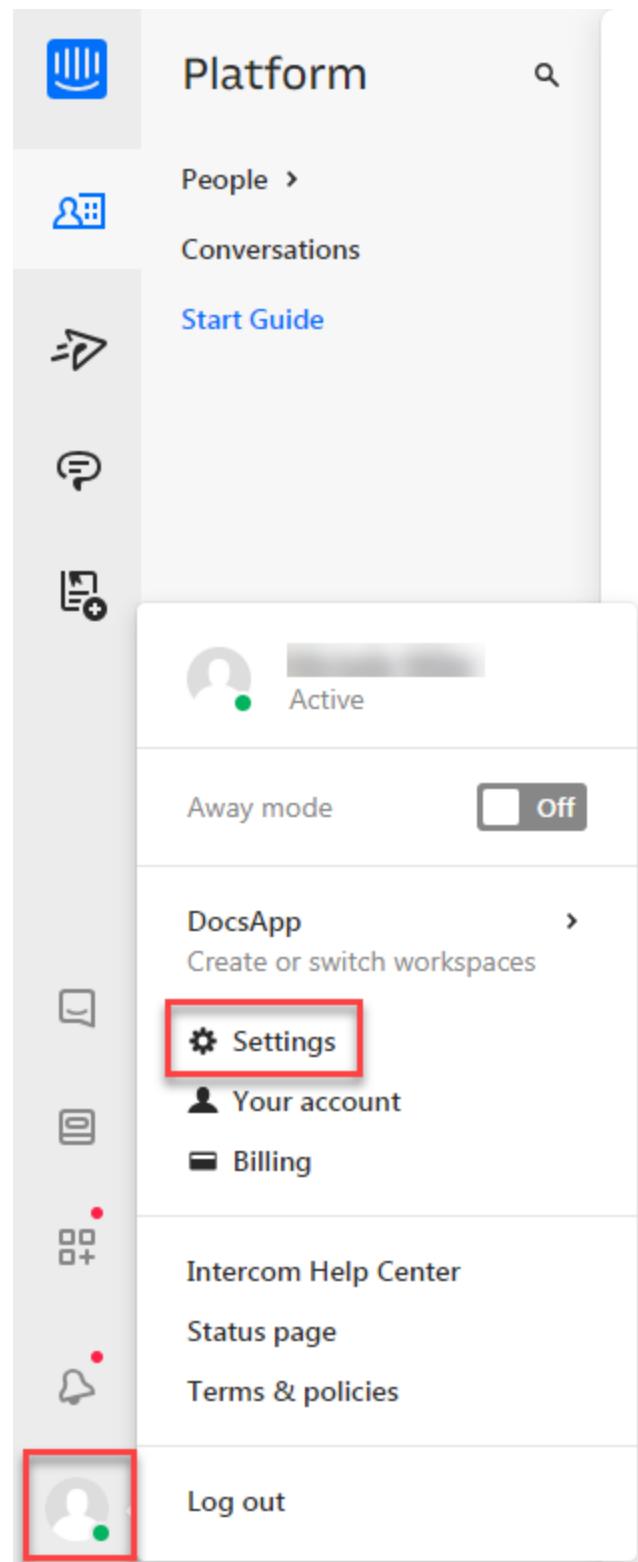
If you don't have an Intercom workspace set up yet, create one at [www.intercom.com](http://www.intercom.com). At a minimum, you need a subscription to the *Inbox* product from Intercom to be able to create a workspace.

3. From the assistant email account you created earlier, find the invitation from Intercom. Click the link in the email to join the team. Sign up using the assistant's functional email address, and then join the team.

4. **Optional:** Update the profile for your assistant.

You can edit the name and profile picture for your assistant. This profile represents the assistant in private agent communications within your workspace, and in public interactions with customers through your Intercom apps. Create a profile that reflects your brand.

Click the Intercom profile icon in the navigation bar to access profile and workspace settings.



## Preparing the dialog

If you do not have a dialog skill associated with your assistant, create one or add one to your assistant now. See [Building a dialog](#) for more details.



**Note:** Triggering a search through a search skill is not currently supported from an Intercom integration.

Complete these steps in your dialog skill so the assistant can handle user requests, and can pass the conversation to a human agent when a customer asks for one.

1. Add an intent to your skill that can recognize a user's request to speak to a human.

You can create your own intent or add the prebuilt intent named `#General_Connect_to_Agent` that is provided with the **General** content catalog developed by IBM.

2. Add a root node to your dialog that conditions on the intent that you created in the previous step. Choose **Connect to human agent** as the response type.
3. Prepare each dialog branch to be triggered by the assistant from the Intercom app.

Every root dialog node in the dialog can be processed by the assistant while it is functioning as an Intercom teammate, including root nodes in folders. You will specify the action that you want the assistant to take for each dialog branch later when you configure the Intercom integration. Therefore, although you cannot hide a node from Intercom, you can configure the assistant to do nothing when the node is triggered.

Fill in the following fields of the root node of each dialog branch:

- **Node name:** Give the node a name. This name is how you will identify the node when you configure interactions for it later. If you don't add a name, you will have to choose the node based on its node ID instead.

- **External node name:** Add a summary of the purpose of the dialog branch. For example, *Find a store*.

This information is shown to other agents on the assistant's team when the assistant offers to answer a user query. If there is more than one dialog node that can address the query, the assistant shares a list of response options with human agent teammates to get their advice about which response to use.

And finally:

**Wait for user input**

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

Enter external node name

 **Status:** this node will not be used for features that require displaying an external node name unless an external node name is provided.

 **Tip:** Do **not** add an external node name to the root node that you created in Step 2. When an escalation occurs, your assistant looks at the external node name of the last processed node to learn which user goal was not met successfully. If you include an external node name in the node with the connect to human agent intent, then you will prevent your assistant from learning the last real, goal-oriented node that the user interacted with before escalating the issue.

4. If a child node in the branch conditions on a follow-up request or question that you do not want the assistant to handle, add a **Connect to human agent** response type to the node.

For example, you might want to add this response type to nodes that cover sensitive issues only a human should handle or that track when an assistant repeatedly fails to understand a user.

At run time, if the conversation reaches this child node, the dialog is passed to a human agent at that point. Later, when you set up the Intercom integration, you can choose a human agent as a backup for each branch.

Your dialog is now ready to support your assistant in Intercom.

## Dialog considerations

Some rich responses that you add to a dialog are displayed differently within the "Try it out" pane from how they are displayed to Intercom users. The following table describes how the response types are treated by Intercom.

Response type	How displayed to Intercom users
Option	The options are displayed as a numbered list. In the <b>title</b> or <b>description</b> field, provide instructions that explain to the user how to choose an option from the list.
Image	The image <b>title</b> , <b>description</b> , and the image itself are rendered.
Pause	Whether or not you enable it, a typing indicator is not displayed during the pause.

See [Rich responses](#) for more information about response types.

## Adding an Intercom integration

1. From the Assistants page, click to open the assistant tile that you want to deploy.
2. From the Integrations section, click **Add integration**.
3. Click **Intercom**.

Follow the instructions that are provided on the screen. The following sections help you with the steps.

## Connecting the assistant to Intercom

As soon as you give Intercom permission to use the assistant, the assistant becomes a viable member of the Intercom team.

Human agents can assign messages to the assistant by using Intercom's assignment rules. Messages can be assigned to the assistant in the following ways:

- Automatic assignment of inbound conversations to a teammate or team inbox based on some criteria
- Manual reassignment made by a human agent at run time.

See the [Intercom documentation](#) for more details.

1. When your dialog is ready, click **Connect now**.
2. Click **Access Intercom** to be redirected to the Intercom site.

**⚠️ Important:** Log in to Intercom using the assistant's functional email address and password, not your own. You want to establish the connection to a functional ID that is shared by more than one person in your organization.

3. Click **Authorize access**.
4. Click **Back to overview**.

**⚠️ Important:** The assistant is now available to receive assignments from Intercom teammates. If you haven't yet, [set up your dialog](#).

## Configuring message routing

Assign human teammates as backups for the assistant in case the assistant needs to transfer an in-progress conversation to a human. You can choose a different team or teammate to be the backup contact for each dialog branch.

To set up routing assignments for escalations from the assistant to a human, complete the following steps:

1. From the Intercom integration page, click **My Dialog Skill is ready** to confirm that you have prepared your dialog.

**⚠️ Important:** Only click this button if you have completed the [Preparing the dialog](#) procedure.

2. In the *Settings* section, click **Manage rules**.

If you make no changes, the backup contact for all of the nodes remains unassigned.

3. Click **New rule**.

4. From the *Choose node* drop-down list, choose the node for the dialog branch you want to configure.

Remember, branches are identified by their node name. If you did not specify a node name, then the node's ID is displayed instead.

5. Choose the team or human agent teammate to be the backup contact for this dialog branch. The user query will be escalated to this person if the assistant cannot answer the query or hits a child node with a *Connect to human agent* response type, indicating that it should be answered only by a human.

6. To define routing rules for other dialog branches, click **New rule** again, and repeat the previous steps.

Don't forget to set up an assignment for any root nodes that have a *Connect to human agent* response type in a child node in their branch. If you do not transfer the associated root node to a specific person or team, a sensitive matter can be transferred to the *Unassigned* inbox.

7. After adding rules, click **Return to overview** to exit the page.

## Give the assistant permission to monitor and answer user queries

When you want the assistant to start monitoring an Intercom inbox, and answering messages on its own, turn on monitoring.

Your assistant watches user inquiries as they are logged in Intercom. When the assistant is confident that it knows how to answer a user query, the assistant responds to the user directly. (The assistant is confident when the top intent identified by your assistant has a confidence score of 0.75 or higher.)

If you do not want the assistant to answer certain types of user queries, then you can add rules to specify other actions for the assistant to take per dialog branch. For example, you might want to start incorporating the assistant into the Intercom team more conservatively, allowing the assistant only to suggest responses as it transfers user messages to other teammates for them to answer. Over time, after the assistant proves itself, you can give it more responsibility.

To configure how you want the assistant to handle specific dialog branches, define rules.

1. From the Intercom integration page, in the *Enable your assistant to monitor an inbox* section, switch monitoring **On**.

2. In **Settings**, click **Manage rules**.

If you do not define rules, the assistant is configured to monitor the *Unassigned* inbox and automatically answer user inquiries that it is confident it can address.

3. From the *Monitor Intercom inbox* field, choose the Intercom inbox that you want the assistant to monitor.

4. Click **New rule** to define a unique interaction pattern for a specific dialog branch.

5. From the *Choose node* drop-down list, choose the node for the branch you want to configure.

Remember, branches are identified by their node name. If you did not specify a node name, then the node's ID is displayed instead.

6. Pick the type of action that you want the assistant to perform when this dialog node is triggered. The action type options are these:

- **Do nothing**: The assistant is not involved in the response; the user's message remains in the inbox for someone else to address.
- **Send to team or teammate**: The assistant evaluates the user input to determine its goal, and then transfers it to the appropriate teammate.
- **Suggest to team or teammate**: The assistant provides the teammate with suggestions for how to respond by sharing notes with the human agent through the internal Intercom app.
  - If the user input triggers a dialog branch, meaning a root dialog node with child nodes that represents a comprehensive interaction, then the assistant indicates that it is capable of addressing the request, and offers to do so.
  - If the user input triggers a root node with no children, then the assistant simply shares the programmed response from the node with the human agent, but does not respond directly to the user.
  - If the input triggers more than one dialog node with high confidence, the assistant shows the human teammate a list of possible responses and asks the teammate to choose the best response.

In every case, the human agent decides whether to let the assistant take over.

- **Answer**: The assistant responds to the user directly, without conferring with any other teammates.

Teammate involvement is required for any nodes to which you assign the *Send to team or teammate* or *Suggest to team or teammate* action types. Be sure to go back and add a rule that assigns the right person or team as the backup for this dialog node in particular.

7. To define unique interaction settings for other dialog branches, click **New rule** again, and repeat the previous steps.

8. After adding rules, click **Return to overview** to exit the page.

As your dialog changes, you will likely return to the Intercom integration page to make incremental changes to these rules.

## Testing the integration

To effectively test your Intercom integration from end-to-end, you must have access to an Intercom end-user application. You already created or edited an Intercom workspace. The workspace must have an associated user interface client. If it does not, see [How do I create an app?](#) for help with setting one up.

Submit test user queries through a client application that is associated with your Intercom workspace to see how the messages are handled by Intercom. Verify that messages that are meant to be answered by the assistant are generating the appropriate responses, and that the assistant is not responding to messages that it is not configured to answer.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with Slack

Slack is a cloud-based messaging application that helps people collaborate with one another.

After you configure a dialog skill and add it to an assistant, you can integrate the assistant with Slack.

When integrated, depending on the events that you configure the assistant to support, your assistant can respond to questions that are asked in direct messages or in channels where the assistant is directly mentioned.

## Adding the Slack integration

1. From the Assistants page, click to open the assistant tile that you want to deploy.

2. From the Integrations section, click **Add integration**.

3. Click **Slack**.

4. You need to have a Slack app to connect to.

If you don't have a Slack app, create one now. See [Starting with Slack apps](#).

5. Go to the [Your Apps](#) page on the Slack website, and then click the app you want to use.



**Tip:** Open the Slack app in a new browser tab, so you can easily switch back and forth between the Slack app settings page and Watson Assistant Slack integration configuration page.

6. From the settings page for your Slack app, open the [App Home](#) page.

7. Add access scopes for your Slack app.

The button label might be *Review Scopes to Add* or *Update scopes* depending on whether you are creating a new app or editing an app that you created before February 2020.



**Note:** The method for Slack access changed. For more information about it, read the [Slack blog post](#) about it.

8. Assign bot token scopes to your Slack app. At a minimum, apply the following scopes:

- `app_mentions:read`
- `chat:write`
- `im:history`
- `im:read`
- `im:write`

9. Click *Install App to Workspace*, and then allow the installation when prompted.

If you are editing scopes for an existing application, reinstall it.

10. From the Slack settings App Home page, enable the *Always Show My Bot As Online* setting.

11. Go to the *OAuth and Permissions* page in Slack, copy the *Bot User OAuth Access Token*.

12. From the Watson Assistant Slack integration configuration page, paste the token that you copied in the previous step into both the **OAuth access token** and **Bot user OAuth access token** fields.

13. On the Slack app settings page, go to the *Basic Information* page, and then find the *App Credentials* section. Copy the app credential verification token.

14. From the Watson Assistant Slack integration configuration page, paste the verification token that you copied in the previous step into the **Verification token** field.

15. Click **Generate request URL**, and then copy the generated request URL.

16. Return to the Slack app settings page. Open the *Event Subscriptions* page, and then turn on *Enable Events*. Paste the request URL that you copied in the previous step into the field.

17. On the *Event Subscriptions* page in Slack, find the *Subscribe to Bot Events* section. Click *Add Bot User Event*, and then select the event types you want to subscribe to. You must select at least one of the following types:

- `message.im`: Listens for message events that are posted in a direct message channel.
- `app_mention`: Listens for only message events that mention your app or bot.



**Note:** Choose the `app_mention` entry in normal font, *not* the `app_mention` entry that is in bold font.

18. Click *Save Changes*.

19. Optional: To add support for showing buttons, menus, and disambiguation options in the Slack app, go to the *Interactive Components* tab and enable the feature. Paste your request URL in the provided text entry field, and then click *Enable Interactive Components*.

## Dialog considerations

The rich responses that you add to a dialog are displayed in a Slack channel as expected, with the following exceptions:

- **Connect to human agent**: This response type is ignored.
- **Image**: This response type embeds an image in the response. A title and description are displayed before the image. Slack will automatically show a preview of the image.
- **Audio**: This response type embeds audio from various file formats in the response. A title and description are displayed before the attachment. Slack will automatically show a preview of the content for supported formats.

- **Video:** This response type embeds a native video from various file formats in the response. A title and description are displayed before the attachment. Slack will automatically show a preview of the content for supported formats.
- **iframe** This response type is not supported.
- **Option:** This response type shows a list of options that the user can choose from.
  - After a user clicks one of the options, the choices disappear and are replaced by the user input that is generated by the user's choice. If you include multiple response types in a single response, position the option response type last. Otherwise, the output might contain a mix of responses and user inputs that can confuse the user.
  - If the options are displayed in a drop-down list, then each option value must be 75 characters or fewer in length. When a list includes 5 or more options, it is displayed in a drop-down list automatically.
- **Pause:** This response type pauses the assistant's activity in the Slack channel. However, no visible indicator is shown to indicate that the assistant is paused, whether you choose to show typing or not.
- **Search skill:** The response type is supported, but you must test and curate your search results to ensure that each result is as concise as possible. The combined length of the title, body, and URL of the search result must be less than or equal to 2,990 characters.

See [Rich responses](#) for more information about response types.

## Chatting with the assistant

To start a chat with the assistant, complete the following steps:

1. Open Slack, and go to the workspace associated with your app.
2. Click the application that you created from the Apps section.
3. Chat with the assistant.



**Note:** The Welcome node of your dialog is not processed by the Slack integration. The welcome message is not displayed in the Slack channel like it is in the "Try it out" pane or in the assistant preview. It is not triggered from here because nodes with the `welcome` special condition are skipped in dialog flows that are started by users. Slack waits for the user to initiate the conversation. For more information about how to set context variable values consistently at the start of a conversation, see [Starting the dialog](#).

You can upload media files or even documents to the chat. Files shared in the chat can be intercepted and processed by a configured premessage webhook. For more details, see [Processing input attachments](#).

The dialog flow for the current session is restarted after 60 minutes of inactivity (5 minutes for Lite and Standard plans). This means that if a user stops interacting with the assistant, after 60 (or 5) minutes, any context variable values that were set during the previous conversation are set to null or back to their default values.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with SMS with Twilio

Add a text messaging integration so your assistant can exchange messages with your customers.

The Short Messaging Service (SMS) supports text-only messages. Typically, SMS restricts the text message length to 160 characters. The Multimedia Messaging Service (MMS) supports sending images and text messages that are over 160 characters in length. When you create a phone number with Twilio, MMS message support is included automatically.

Customers send text messages to your Twilio-hosted phone number. Twilio uses a messaging webhook that you set up to send a POST request with the text message body to your assistant. Each response from the assistant is sent back to Twilio to be converted to an outbound SMS message that is sent to the customer. The responses are sent to the Twilio API for processing. You provide your Twilio account SID and project authentication token information, which serve as your Twilio API access credentials.



**Note:** This feature is available only to Plus plan users.

## Before you begin

If you don't have a text messaging phone number, set up a SMS with Twilio account and get a phone number.

1. Go to the [Twilio website](#).
2. Create an account or start a free trial.
3. From the *All Products and Services* menu , click *Phone numbers*.
4. Follow the instructions to get a phone number.

When you get a Twilio phone number, it supports voice, SMS, and MMS automatically. Your new phone number is listed as an active number.

 **Tip:** Keep the Twilio web page open in a web browser tab so you can refer to it again later.

## Migrating from Voice Agent with Watson

If you created an IBM® Voice Agent with Watson service instance in IBM Cloud to enable customers to exchange text messages with an assistant, use the *SMS with Twilio* integration instead.

The *SMS with Twilio* integration provides a more seamless integration with your assistant and supports as many Twilio phone numbers as needed. However, the integration currently does not support the following functions:

- Starting an SMS-only interaction with an outgoing text
- Configuring backup locations
- Reviewing the usage summary page. Use IBM Log Analysis instead. For more information, see [Viewing logs](#).

To migrate from Voice Agent with Watson to the Watson Assistant *SMS with Twilio* integration, complete the following step:

1. Do one of the following things:
  - If your Voice Agent with Watson service instance uses an SMS service provider other than Twilio, you cannot continue to use it. You must create an SMS account with Twilio first. Complete the [Before you begin](#) steps to create the account. Next, set up the integration.
  - If your Voice Agent with Watson service instance uses Twilio as its SMS provider, you can go directly to setting up the integration.

## Set up the integration

To set up the integration, complete the following steps:

1. From the Assistants page, click to open the assistant tile that you want to deploy.
2. From the Integrations section, click **Add integration**.
3. Click **SMS with Twilio**.
4. Click **Create**.
5. From the Twilio site, click the home icon to go to your project dashboard.

Copy the following values and store them temporarily, so you can paste them into the *SMS with Twilio* integration setup page in the next step.

- Account SID
- Auth token

6. Return to the *SMS with Twilio* integration setup page.

Paste the values that you copied in the previous step into the fields with the corresponding names in the *Twilio account information* section.

- **Account SID**
- **Auth token**

7. Scroll to the *Setup instructions* section, and then copy the value from the **Webhook URI (uniform resource identifier)** field.

You will add this URI to the webhook configuration in Twilio. If you want to support more than one phone number, you must add the URI to the webhook for each phone number separately.

8. Go to your Twilio account web page. From the *All Products and Services* menu, click *Phone Numbers*.
9. From the *Active Numbers* page, click one of your phone numbers. Scroll to the *Messaging* section, and then find the *Webhook* field that defines what to do when *a message comes in*.

Paste the value that you copied from the *Webhook URI* field into it.

10. If you want to support multiple phone numbers, repeat the previous step for each phone number that you want to use.
11. Click **Save and exit**.

To watch a video that walks through the setup process, see [Phone and SMS Integration](#) in the *IBM Watson Apps Community*.

If you want your assistant to be able to switch between voice and text during a customer interaction, enable both the phone and text messaging integrations. The integrations do not need to use the same third-party service provider. For more information, see [Integrating with phone](#).

## Advanced configuration options

Click the *Advanced options* tab to make any of the following customizations to the messaging behavior:

- **Initiate conversation from inbound messages:** Disable this option if you want to limit messaging support to allow messages that are sent in the context of an ongoing phone integration conversation only, and not allow customers to start a message exchange with the assistant outside of a phone call.
- **Default failure message:** Add a message to send to the customer if the SMS connection fails.
- **Base URL:** This URL is the REST API endpoint for the SMS service you are using.

## Optimize your dialog for messaging

For the best customer experience, design your dialog with the capabilities of the Twilio integration in mind:

- Do not include HTML elements in your text responses.
- You can send media files or even documents in your response. These files can be intercepted and processed by a configured premessage webhook. For more details, see [Processing input attachments](#).
- The SMS with Twilio integration does not support chat transfers that are initiated with the `Connect to human agent` response type.
- The **pause** response type is ignored. If you want to add a pause, use a `vgwConversationResponseTimeout` context variable instead.
- **Image, Audio, Video** response types allow sending a message containing media. A title and description are sent along with the attachment. Note that depending on the carrier and device of the end user these messages may not be successfully received. For a list of the supported content types, see [Accepted Content Types for Media](#).
- You can include search skill response types in dialog nodes that the phone integration will send as a message. The message includes the introductory text (*I searched my knowledge base* and so on), and then the body of only the first search result.

If you want to use the same dialog for an assistant that you deploy to many different platforms, add custom responses per integration type. You can add a conditioned response that tells the assistant to show the response only when the SMS with Twilio integration is being used. For more information, see [Building integration-specific responses](#).

For reference documentation, see [Handling SMS with Twilio interactions](#).

## Troubleshooting

Find solutions to problems that you might encounter while using the integration.

- If you get a *Forbidden* message, it means the phone number that you specified when you configured the integration cannot be verified. Make sure the number fully matches the SMS phone number.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Integrating with WhatsApp](#). To see all documentation for the new {{site.data.keassistant\_classic\_shortnshort}}, please go [here](#).

## Integrating with WhatsApp

Integrate with WhatsApp messaging so your assistant can exchange messages with your customers where they are.

Many customers use WhatsApp because it provides fast, simple, secure messaging for free, and is available on phones all over the world. WhatsApp uses the phone Internet connection to send messages so customers can avoid SMS fees.

This integration creates a connection between your assistant and WhatsApp by using Twilio as a provider.

### Before you begin

If you don't have one, set up a Twilio messaging account and get a phone number.

1. Go to the [Twilio website](#).
2. Create an account.
3. From the *All Products and Services* menu , click *Phone numbers*.
4. Follow the instructions to get a phone number.

When you get a Twilio phone number, it supports voice, SMS, and MMS automatically. Your new phone number is listed as an active number.

### Ask WhatsApp for permission to enable your Twilio number for WhatsApp.

WhatsApp has a rigorous process that they use to review all businesses that want to interact with customers over their network. WhatsApp, which is owned by Facebook, requires that you register your business with the Facebook business directory.

1. To register, go to the [Facebook Business Manager](#) website, and click *Create account* and follow the instructions to create an account.
2. Get your Facebook Business Manager ID. In [Settings](#), click the **Business info** tab. The Facebook Business Manager ID is at the top of the page.
3. Submit the *Request to enable your Twilio numbers for WhatsApp* form from the [Twilio API for WhatsApp](#) web page.

Tips for specifying the following values:

- **Phone Number:** Specify the Twilio phone number that you created earlier.

**Tip:** Consider provisioning more than one phone number and going through the process of getting permission for the numbers in parallel. If your number was used by a different business previously (because Twilio assigned you a number that was used before, for example), WhatsApp will reject it.

- Are you working with an ISV : No
- Twilio Account SID: From the Twilio site, click the home icon to go to your project dashboard to find the SID.
- Facebook Business Manager ID: Add the ID for the account that you created in the previous step.

#### 4. Click Request Now.

Give WhatsApp time to evaluate and approve your request. It can take up to 7 days for your request to be approved.

## Set up the integration

To set up the integration, complete the following steps:

1. From the Assistants page, click to open the assistant tile that you want to deploy.
2. From the Integrations section, click **Add integration**.
3. Click **WhatsApp with Twilio**.
4. Click **Create**.
5. From the Twilio site, click the home icon to go to your project dashboard.

Copy the following values and store them temporarily, so you can paste them into the phone integration setup page in the next step.

- Account SID
  - Auth token
6. Return to the WhatsApp integration setup page.

Paste the values that you copied in the previous step into the fields with the corresponding names in the *Twilio account information* section.

- **Account SID**
  - **Auth token**
7. Click **Sync account**.
  8. Add the Twilio messaging phone number that you created previously to the **Company phone number** field.  
Specify the number by using the international phone number format: **+1 958 555 0123**. Do not include parentheses (**(958)**).  
The phone number must be unique per WhatsApp integration.
  9. Copy the value from the **WhatsApp Webhook** field.
  10. Click **Save and exit**.

## Testing the integration

While you wait for WhatsApp to approve your request, you can test the integration by using the Twilio sandbox. With the sandbox, you can send and receive preapproved template messages to numbers that join your sandbox, using a shared Twilio test number.

Do not use the Twilio sandbox in production. Sandbox sessions expire after 3 days.

1. To create a sandbox, go to the [Twilio Try WhatsApp](#) web page. An *Activate your sandbox* prompt is displayed. Agree to have a sandbox created, and confirm your choice.
2. Follow the instructions to create the sandbox.
3. Connect to the sandbox by sending a WhatsApp message from your device to the Sandbox phone number.
4. From the *Programmable Messaging* menu, expand *Settings*, and then click *WhatsApp Sandbox Settings*.
5. In the *Sandbox Configuration* section, paste the webhook URI that you copied earlier into the *when a message comes in* field, and then save the configuration.
6. You can test the integration by sending a message from WhatsApp to the shared phone number that is assigned to your Twilio sandbox.

## Finish the product integration

After WhatsApp grants permission for your Twilio phone number or number to access the WhatsApp network, update the integration to use your dedicated

Twilio phone number instead of the sandbox number.

1. From the *WhatsApp with Twilio* integration setup page, scroll to the *Setup instructions* section, and then copy the value from the **Webhook URI (uniform resource identifier)** field.
2. Go to your Twilio account web page and add the webhook you copied to the Twilio configuration to complete the connection to the WhatsApp integration in Twilio.

## Give your customers fast access to your assistant

You can add an icon to your web page that customers can click to start a conversation over WhatsApp with your assistant.

To add an icon to your web page, complete the following steps:

1. From the *WhatsApp with Twilio* integration setup page, click the **Click to chat** tab.
2. In the **Prefilled message** field, add text that you want WhatsApp to send to your assistant on the customer's behalf to get the conversation started.  
Specify a message that you know your assistant can answer in a useful way.
3. Copy the embed link and add it to your web page. Consider adding text in front of the icon that explains what the icon does. For example, you might add a `<span>` HTML tag in front of the icon's `<span>` element that says `Have a question? Ask Watson Assistant for help.`

When a user clicks the icon in your web page, it opens a WhatsApp messaging session that is connected to your assistant, and adds the text you specify into the user's text field, ready to be submitted.

## Dialog considerations

The rich responses that you add to the dialog are displayed in WhatsApp as expected, with the following exceptions:

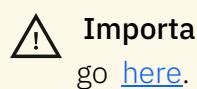
- **Image:** This response type embeds an image in the response. A title and description are displayed before the image.
- **Audio:** This response type embeds audio from various file formats in the response. A title and description are not displayed, whether or not you specify them. WhatsApp automatically shows a preview of content on supported platforms (such as SoundCloud and Mixcloud).
- **Video:** This response type embeds a native video from various file formats in the response. A title and description are not displayed, whether or not you specify them. WhatsApp automatically shows a preview of content on supported platforms (such as YouTube and Vimeo).
- The **iframe** response type is not supported.

For the best customer experience, design your dialog with the capabilities of the WhatsApp integration in mind:

- A text response that contains more than 1,600 characters is broken up into multiple responses.
- You can upload media files or even documents to the chat. Files shared in the chat can be intercepted and processed by a configured premessage webhook. For more details, see [Processing input attachments](#).
- Do not include HTML elements in your text responses.
- The WhatsApp with Twilio integration does not support chat transfers that are initiated with the *Connect to human agent* response type.
- If you use Markdown syntax, see the *Supported Markdown syntax* table.
- To include a hypertext link in a text response, specify the url directly. Do not use markdown syntax for links. For example, specify `Contact us at https://www.ibm.com.`

Format	Syntax	Example
Italics	<code>We're talking about _practice_.</code>	We're talking about <i>practice</i> .
Bold	<code>There's *no* crying in baseball.</code>	There's <b>no</b> crying in baseball.

### Supported Markdown syntax



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Integrating with a custom application

If the available integration channels do not meet your needs, you can build your own client application as the interface between the assistant and your users.

See [API overview](#) for more information.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Deleting an integration

To stop an assistant from being available to users from a certain channel, delete the integration to that channel.

1. From the Assistants page, click to open the assistant tile that you want to edit.
2. From the Integrations section, find the tile for the integration that you want to delete.
3. Click the  icon, and then choose **Delete**. Confirm the deletion.

For Slack and Facebook Messenger integrations, the app that you created to connect to the assistant will no longer function properly. Be sure to update the app to connect to another assistant or delete it altogether.

## Analytics



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Metrics overview

The overview page provides a summary of the interactions between users and your assistant.

Analytics help you to understand the following things:

- *What do customers want help with today?*
- *Is your assistant understanding and addressing customer needs?*
- *How can you make your assistant better?*

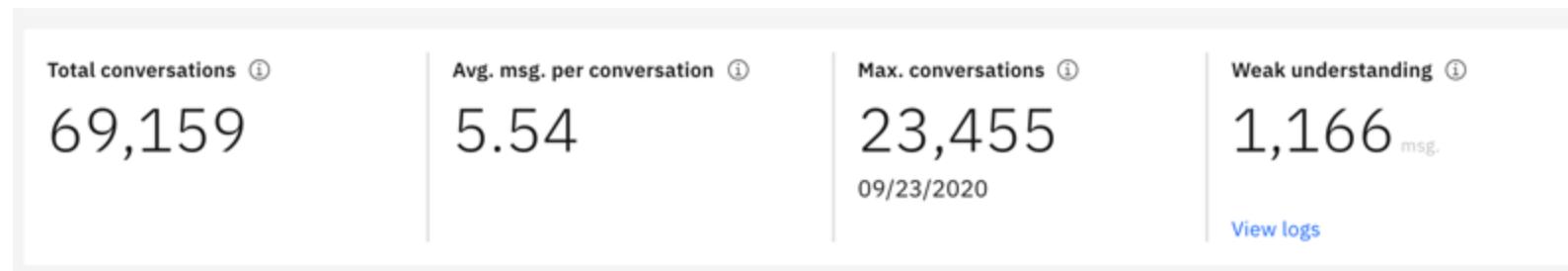
To see metrics information, select **Overview** in the navigation bar. The information in Analytics is not immediately updated after a user interacts with your assistant. Watson Assistant prepares the data for Analytics in the background.



**Note:** After the numbers in metrics hit the 3,000 and above range, the totals are approximated to prevent performance lags in the page.

### Scorecards

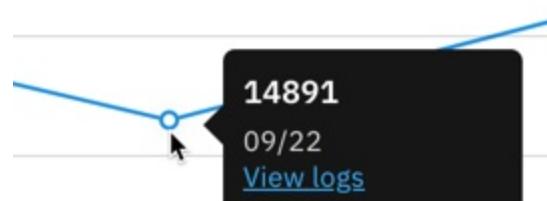
The scorecards give you a quick view of your metrics. Scroll to see full interactive graphs later in the page.



- **Total conversations:** The total number of conversations between active users and your assistant that occur during the selected time period. The total conversations metric and the *User conversations* page only include conversations in which both the assistant and customer participate. Conversations that only have welcome messages from the assistant, or that only have user messages of zero length, are not included. This metric is not used for billing purposes. An exchange with a user is not considered a billable conversation until the customer submits a message.
- **Avg. msg. per conversation:** The total messages received during the selected time period divided by the total conversations during the selected time period, as shown in the corresponding graph.
- **Max. conversations:** The maximum number of conversations for a single data point within the selected time period.
- **Weak understanding:** The number of individual messages with weak understanding. These messages are not classified by an intent, and do not contain any known entities. Reviewing unrecognized messages can help you to identify potential dialog problems.

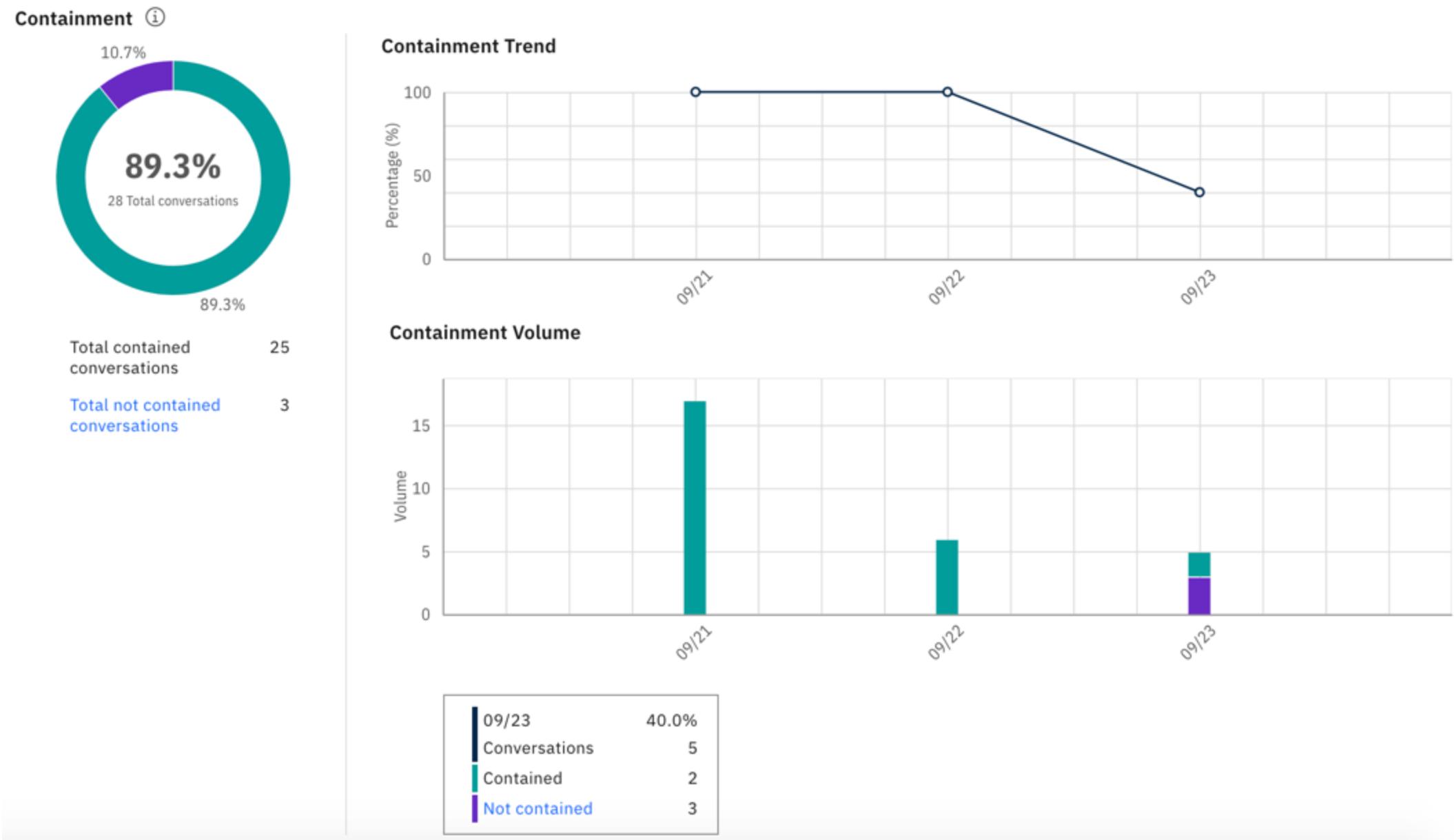
### Graphs and statistics

Detailed graphs provide additional information. Click a data point on the graphs to see more detail.



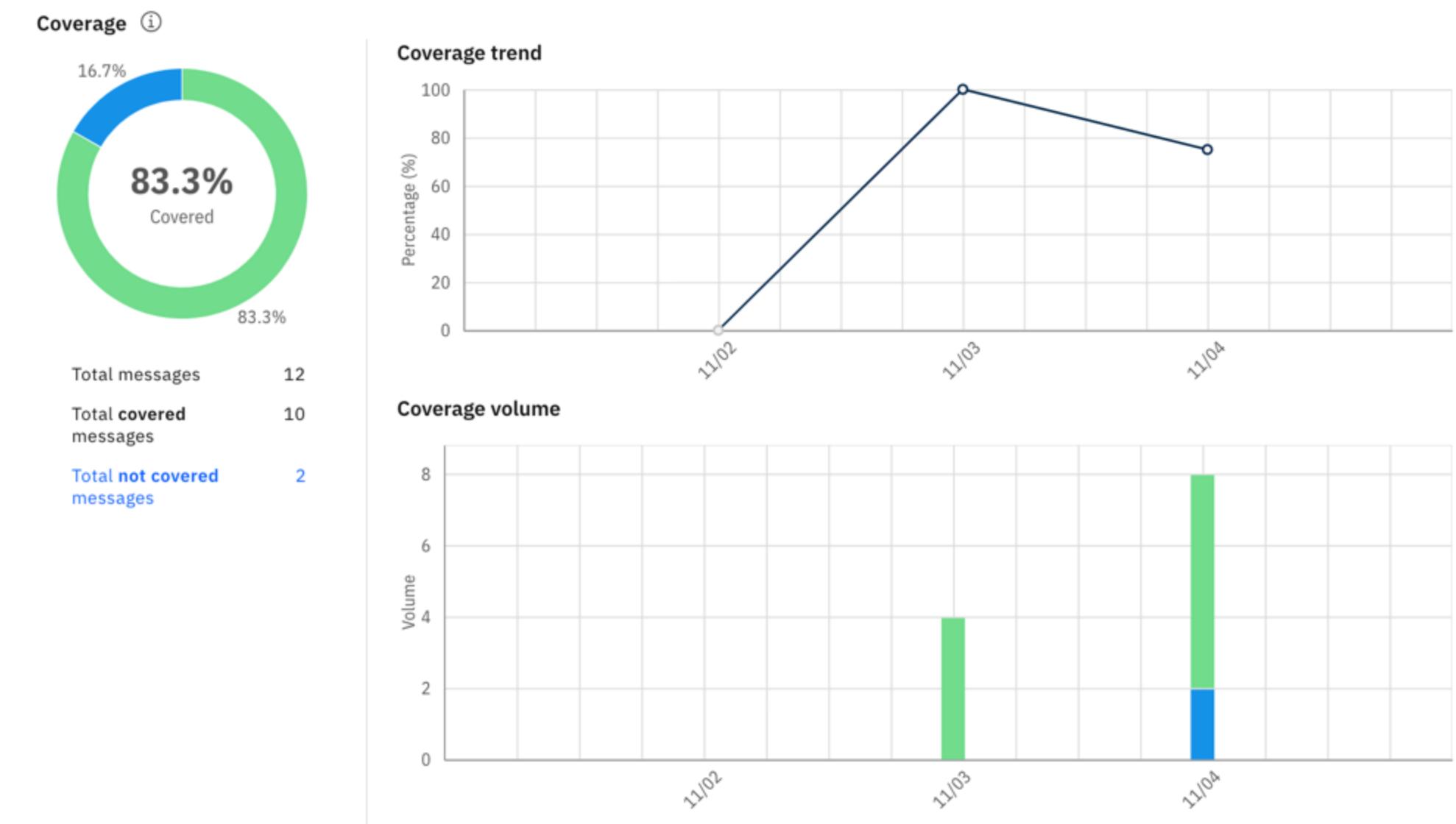
- **Containment:** Number of conversations in which the assistant is able to satisfy the customer's request without human intervention.
  - The volume graph shows the total number of conversations per day and how many of the conversations were contained and not contained.

- The trend graph shows the percentage of daily conversations that were contained. This graph helps you to see if the assistant is getting better or worse at containing conversations over time.



The containment metric requires that your dialog flag requests for external support when they occur. For more information, see [Measuring containment](#).

- Coverage:** Number of conversations in which the assistant is confident that it can address a customer's request.
- The volume graph shows the total number of conversations per day and how many of the conversations were covered (meaning intents in your dialog understood user requests and were able to address them), and not covered (meaning the input did not match an intent in the dialog and was processed by the *Anything else* node instead).
- The trend graph shows the percentage of daily conversations that were covered. This graph helps you to see if your dialog is getting better or worse at covering conversations over time.



The coverage metric requires that your dialog contain an *Anything else* node. For more information, see [Ending the conversation gracefully](#).



**Note:** The containment and coverage metrics are available to Plus or Enterprise plan users.

- *Total conversations*: The total number of conversations between active users and your assistant during the selected time period. This number counts exchanges in which the welcome message is displayed to a user, even if the user doesn't respond.
- *Average messages per conversation* - The total messages received during the selected time period divided by the total conversations during the selected time period.
- *Total messages* - The total number of messages received from active users over the selected time period.
- *Active users* - The number of unique users who have engaged with your assistant within the selected time period.
- *Average conversations per user* - The total conversations divided by the total number of unique users during the selected time period.



**Important:** Statistics for *Active users* and *Average conversations per user* require a unique `user_id` parameter to be specified with the messages. This value is typically specified by all integrations because it is used for billing purposes. For more information, see [User-based plans explained](#).

## Controls

You can use the following controls to filter the information:

- *Time period control* - Use this control to choose the period for which data is displayed. This control affects all data shown on the page: not just the number of conversations displayed in the graph, but also the statistics displayed along with the graph, and the lists of top intents and entities.



**Note:** The statistics can cover a longer time period than the period for which logs of conversations are retained.

Past 30 days ▼ 24 Aug 20 to 23 Sep 20 by day ▼

You can choose whether to view data for a single day, a week, a month, or a quarter. In each case, the data points on the graph adjust to an appropriate measurement period. For example, when viewing a graph for a day, the data is presented in hourly values, but when viewing a graph for a week, the data is shown by day. A week always runs from Sunday through Saturday.

You can create custom time periods also, such as a week that runs from Thursday to the following Wednesday, or a month that begins on any date other than the first.

The time shown for each conversation is localized to reflect the time zone of your browser. However, API log calls are always shown in UTC time. As a result, if you choose a single day view, for example, the time shown in the visualization might differ from the timestamp specified in the log for the same conversation.

Max. conversations ①

3,945

09/24 (05:00 AM - 06:00 AM)

- *Intents and Entities filters* - Use either of these drop-down filters to show data for a specific intent or entity in your skill.



**Important:** The intent and entities filters are populated by the intents and entities in the skill, and not what is in the data source. If you have [selected a data source](#) other than the skill, you might not see an intent or entity from your data source logs as an option in the filters, unless those intents and entities are also in the skill.

- *Refresh data*: Select **Refresh data** to refresh the data that is used in the page metrics.

The statistics show traffic from customers who interact with your assistant; they do not include interactions from the *Try it out* pane.

## Top intents and top entities

You can also view the intents and entities that were recognized most often during the specified time period.

- *Top intents* - Intents are shown in a simple list. In addition to seeing the number of times an intent was recognized, you can select an intent to open the **User conversations** page with the date range filtered to match the data you are viewing, and the intent filtered to match the selected intent.
- *Top entities* are also shown in a list. You can select an entity to open the **User conversations** page with the date range filtered to match the data you are viewing, and the entity filtered to match the selected entity.

See [Improve your skill](#) for tips on how to edit intents and entities based on discoveries you make by reviewing the intents and entities that your assistant recognizes.

**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Improve your skill

The Analytics page of Watson Assistant provides a history of conversations between users and a deployed assistant. You can use this history to improve how your assistants understand and respond to user requests.

To open a list of individual messages between customers and the assistant that uses this dialog skill, select **User conversations** in the navigation bar.

When you open the **User conversations** page, the default view lists inputs that were submitted to the assistant for the last day, with the newest results first. The top intent (#intent) and any recognized entity (@entity) values used in a message, and the message text are available. For intents that are not recognized, the value shown is *Irrelevant*. If an entity is not recognized, or has not been provided, the value shown is *No entities found*.

The screenshot shows the Watson Assistant User conversations page. At the top, there are three tabs: Overview, User conversations (which is selected), and Recommendations BETA. On the right, it says "Data source: HelpDes". Below the tabs are two dropdown menus: "Intents" and "Entities", both with a downward arrow icon. To the right of these is a search bar with the placeholder "Search user statements...". Above the search bar are buttons for "Refresh data" (with a clock icon) and "Last updated: 9:35 AM | February 2018". There are also icons for a calendar and a back arrow. The main area displays a list of messages. One message is shown in detail: "Please do something - I am freezing!" from "07/05/2017 @ 11:33 AM". To the right of the message are icons for a pen and a speech bubble, followed by "#weather" and "@appliance:ac @appliance:heater". Below the message is a link "Open conversation". At the bottom left of the main area, it says "Showing 1 through 50 of 450 results".

**Important:** The User conversations page displays the total number of *messages* between customers and your assistant. A message is a single utterance that a user sends to the assistant. A conversation typically consists of multiple messages. Therefore, the number of results on the **User conversations** page is different from the number of conversations that are shown on the **Overview** page.

## Log limits

The length of time for which messages are retained depends on your Watson Assistant service plan:

\$ Service plan	Chat message retention
Enterprise with Data Isolation	Last 90 days
Enterprise	Last 30 days
Premium (legacy)	Last 90 days
Plus	Last 30 days
Trial	Last 30 days
Lite	Last 7 days

## Filtering messages

You can filter messages by *Search user statements*, *Intents*, *Entities*, and *Last n days*.

*Search user statements* - Type a word in the search bar. This searches the users' inputs, but not your assistant's replies.

*Intents* - Select the drop-down menu and type an intent in the input field, or choose from the populated list. You can select more than one intent, which filters the results using any of the selected intents, including *Irrelevant*.

The screenshot shows a search interface for intents. At the top is a dropdown menu labeled "Intents". Below it is a search bar with the placeholder "Search intents". A list of intents is displayed as checkboxes:

- #traffic\_update
- #turn\_down
- #turn\_off
- #turn\_on
- #turn\_up
- #weather

*Entities* - Select the drop-down menu and type an entity name in the input field, or choose from the populated list. You can select more than one entity, which filters the results by any of the selected entities. If you filter by intent *and* entity, your results will include the messages that have both values. You can also filter for results with *No entities found*.

The screenshot shows a search interface for entities. At the top is a dropdown menu labeled "Entities". Below it is a search bar with the placeholder "Search entities". A list of entities is displayed as checkboxes:

- @genre
- @genre\_bad
- @greet
- @option
- @phone
- @radio

## Viewing individual messages

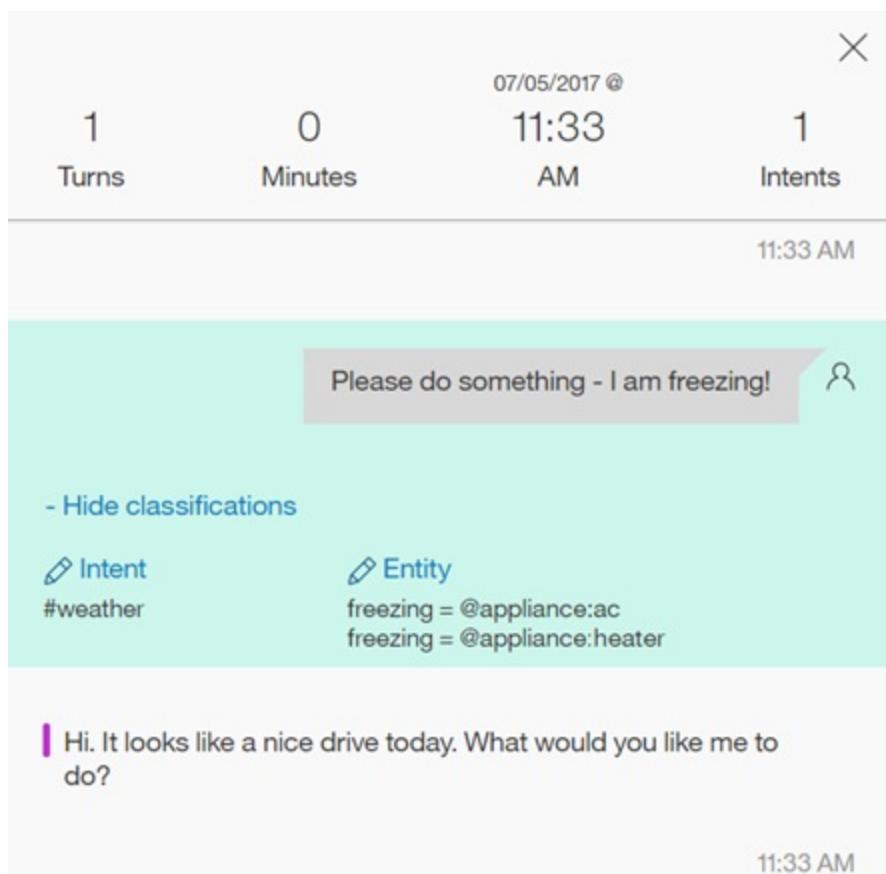
For any user input entry, click **Open conversation** to see the user input and the response made to it by the assistant within the context of the full conversation.

The time that is shown for each conversation is localized to reflect the time zone of your browser. This time might differ from the timestamp shown if you review the same conversation log via an API call; API log calls are always shown in UTC.

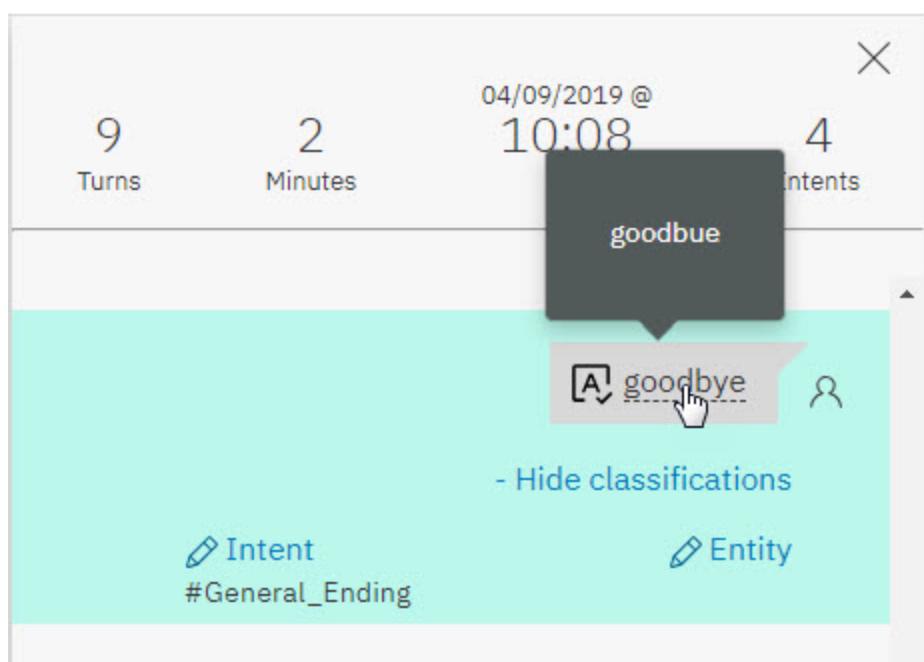
The screenshot shows a message conversation interface. At the top, there are summary statistics: 1 Turn, 0 Minutes, 11:33 AM, and 1 Intent. The date is 07/05/2017. Below this is a timestamp of 11:33 AM. The conversation history consists of two messages:

- User message: "Please do something - I am freezing!" (with a copy icon)
- Assistant response: "Hi. It looks like a nice drive today. What would you like me to do?" (timestamped 11:33 AM)

You can then choose to show the classification(s) for the message you selected.



If the spell check feature is enabled for the skill, then any user utterances that were corrected are highlighted by the Auto-correct icon. The term that was corrected is underlined. You can hover over the underlined term to see the user's original input.



## Improving across assistants

Creating a dialog skill is an iterative process. While you develop your skill, you use the *Try it out* pane to verify that your assistant recognizes the correct intents and entities in test inputs, and to make corrections as needed.

From the User conversations page, you can analyze actual interactions between the assistant you used to deploy the skill and your users. Based on those interactions, you can make corrections to improve the accuracy with which intents and entities are recognized by your dialog skill. It is difficult to know exactly *how* your users will ask questions, or what random messages they might submit, so it is important to frequently analyze real conversations to improve your dialog skills.

For a Watson Assistant instance that includes multiple assistants, there might be times when it is useful to use message data from the dialog skill of one assistant to improve the dialog skill used by another assistant within that same instance.

As an example, say you have a Watson Assistant instance named *HelpDesk*. You might have both a Production assistant and a Development assistant in your HelpDesk instance. When working in the dialog skill for the Development assistant, you can use logs from the Production assistant messages to improve the Development assistant's dialog skill.

Any edits you then make within the dialog skill for the Development assistant will only affect the Development assistant's dialog skill, even though you're using data from messages sent to the Production assistant.

Similarly, if you create multiple versions of a skill, you might want to use message data from one version to improve the training data of another version.

You cannot access log data from assistants that were created in other service instances.

## Picking a data source

The term *data source* refers to the logs compiled from conversations between customers and the assistant or custom application by which a dialog skill was deployed.

When you open the *Analytics* page, metrics are shown that were generated by user interactions with the current dialog skill. No metrics are shown if the current skill has not been deployed and used by customers.

To populate the metrics with message data from a dialog skill or skill version that was added to a different assistant or custom application, one that has interacted with customers, complete these steps:

1. Click the **Data source** field to see a list of assistants with log data that you might want to use.

The list includes assistants that have been deployed and to which you have access. Or you can choose to show a list of other deployments. For more information about other types of deployments, see [Show deployment IDs explained](#).

2. Choose a data source.

Statistical information for the selected data source is displayed.

Notice the list does not include skill versions. To get data that is associated with a specific skill version, you must know the time frame during which a specific skill version was used by a deployed assistant. You can select the assistant as the data source, and then filter the metrics by the appropriate dates to see only log data that was generated by the assistant while it was using the skill version.

## Show deployment IDs explained

Applications that use the older v1 runtime API must specify a deployment ID in each messages sent using the `/message` API. This ID identifies the deployed app that the call was made from. The Analytics page can use this deployment ID to retrieve and display logs that are associated with a specific live application.

For assistants or custom apps that use the v2 version of the API, your assistant automatically includes an assistant ID with each `/message` call, so you can choose a data source by assistant name instead of using a deployment ID.

To add the deployment ID, v1 API users include the deployment property inside the metadata of the `context`, as in this example:

```
"context" : {
 "metadata" : {
 "deployment": "HelpDesk-Production"
 }
}
```

For an Enterprise with Data Isolation plan, you can ask IBM to configure your instances to allow you to access log data from deployed applications across different instances. Each instance must the v1 API and specify a deployment ID with each `/message` call. (If your instances use the v2 API, you cannot get log data from across different instances.)



**Note:** If log sharing is enabled, only someone with Manager service access to the current instance can view logs. The person can view logs from all of the instances that are being shared. Logs from across all of the participating instances are displayed, regardless of the current user's service level access to the other instances. Similarly, when someone with Manager service access to an instance sends a GET request to the v1 `/logs` API endpoint, logs from all instances that participate in log sharing are returned, regardless of the user's service level access to each instance.

## Making training data improvements

Use insights from real user conversations to correct the model associated with your dialog skill.

If you use data from another data source, any improvements you make to the model are applied to the current dialog skill only. The **Data source** field shows the source of the messages you are using to improve this dialog skill, and the header of the page shows the dialog skill you are applying changes to.

## Correcting an intent

1. To correct an intent, select the edit icon for the chosen #intent.
2. From the list provided, select the correct intent for this input.
  - o Begin typing in the entry field and the list of intents is filtered.
  - o You can also choose **Mark as irrelevant** from this menu. (For more information, see [Teaching your assistant about topics to ignore](#).) Or, you can choose **Do not train on intent**, which does not save this message as an example for training.

**Intent**

Select the intent that best describes the user's goal and train Watson to improve its understanding

Do not train on intent

#weather Show Examples

#turn\_on tur

#turn\_down  
#turn\_off  
#turn\_on  
#turn\_up

**Message will not be saved as an example**

3. Select **Save**.

**Evaluate and train Watson**

**Please do something - I am freezing!**

[Open conversation](#)

**Entities** 1 entities recognized

Review and assign the entities that Watson recognized

**Intent**

Select the intent that best describes the user's goal and train Watson to improve its understanding

Do not train on intent

#weather Show Examples

#turn\_on tur On Save, Watson will train on #turn\_on

Mark as irrelevant ? Show Examples

**Tip:** The Watson Assistant service supports adding user input as an example to an intent *as-is*. If you are using @entity references as examples in your intent training data, and a user message that you want to save contains an entity value or synonym from your training data, then you must edit the message later. After you save it, edit the message from the Intents page to replace the entity that it references. For more information, see [Directly referencing an @Entity as an intent example](#).

## Adding an entity value or synonym

- To add an entity value or synonym, select the  edit icon for the chosen @entity.
- Select **Add entity**.

## Please do something - I am freezing!

[Open conversation](#)

▼ Entities

1 entities recognized

Watson recognized these entities; no further training is necessary

freezing = @appliance:ac

If there are additional entities that you want Watson to recognize, select and assign them.

[+ Add entity](#) ←

- Now, select a word or phrase in the underlined user input.

## Please do **something** - I am freezing!

[Open conversation](#)

▼ Entities

1 entities recognized

Watson recognized these entities; no further training is necessary

freezing = @appliance:ac

If there are additional entities that you want Watson to recognize, select and assign them.

Choose a word or words in the user message to add as an entity, value, or synonym. 

- Choose an entity to which the highlighted phrase will be added as a value.

- Begin typing in the entry field and the list of entities and values is filtered.
- To add the highlighted phrase as a synonym for an existing value, choose the `@entity:value` from the drop-down list.

## Please do **something** - I am freezing!

[Open conversation](#)

▼ Entities

1 entities recognized

Watson recognized these entities:

@app
@appliance
@appliance:AC
@appliance:fan
@appliance:music
@appliance:volume
@appliance:wipers

freezing = @appliance:ac

If there are additional entities that you want Watson to recognize, select and assign them.

Choose a word or words in the user message to add as an entity, value, or synonym. 

- Select **Save**.

## Please do something - I am freezing!

[Open conversation](#)

▼ Entities

1 entities recognized

Watson recognized these entities; no further training is necessary

freezing = @appliance:ac

If there are additional entities that you want Watson to recognize, select and assign them.

something = @appliance:fan 

Choose a word or words in the user message to add as an entity, value, or synonym. 

### Teaching your assistant about topics to ignore

It is important to help your assistant stay focused on the types of customer questions and business transactions you have designed it to handle. You can use information collected from real customer conversations to highlight subjects that you do not want your assistant to even attempt to address.

For more information, see [Defining what's irrelevant](#).



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Defining what's irrelevant

Teach your dialog skill to recognize when a user asks about topics that it is not designed to answer.

To teach your assistant about subjects it should ignore, you can review your user conversation logs to mark utterances that discuss off-topic subjects as irrelevant.

Intents that are marked as irrelevant are saved as counterexamples in the JSON workspace, and are included as part of the training data. They teach your assistant to explicitly not answer utterances of this type.

While testing your dialog, you can mark an intent as irrelevant directly from the *Try it out* pane.

The screenshot shows the 'Try it out' pane with the following interface elements:

- Try it out** button
- Clear** and **Manage Context** buttons
- Enter intent name...** input field
- A list of intents: #asdij, #capabilities, #dummy.new.intent, #goodbyes
- Mark as irrelevant** button with an info icon

The input example "Baseball is my favorite sport" is highlighted in blue, indicating it is selected for marking as irrelevant.

Be certain before you designate an input as irrelevant.

- There is no way to access or change the inputs from the user interface later.
- The only way to reverse the identification of an input as being irrelevant is to use the same input in a test integration channel, and then explicitly assign it to an intent.

Often there are subjects that you expect customers to ask and that you want your assistant to address eventually, but that you aren't ready to fully implement yet. Instead of adding those topics as counterexamples which can be hard to find later, capture the customer input examples as new intents.

But don't link dialog nodes to the intents until you're ready. If customers ask about one of these topics in the meantime, the anything\_else node is triggered to explain that the assistant can't help with the current request, but can help them with other things.

## Irrelevance detection

The *irrelevance detection* feature helps your dialog skill recognize subjects that you do not want it to address, even if you haven't explicitly taught it about what to ignore. This feature helps your skill recognize irrelevant inputs earlier in the development process.

To test irrelevance detection in the "Try it out" pane, submit one or more utterances that have absolutely nothing to do with your training data. Irrelevance detection helps your skill to correctly recognize that the test utterances do not map to any of the intents defined in your training data, and classifies them as being **Irrelevant**.

The algorithmic models that help your assistant understand what your users say are built from two key pieces of information:

- Subjects you want the assistant to address. For example, questions about order shipments for an assistant that manages product orders.  
You teach your assistant about these subjects by defining intents and providing lots of sample user utterances that articulate the intents so your assistant can recognize these and similar requests as examples of input for it to handle.
- Subjects you want the assistant to ignore. For example, questions about politics for an assistant that makes pet grooming appointments exclusively.  
You teach your assistant about subjects to ignore by marking utterances that discuss subjects which are out of scope for your application as being irrelevant. Such utterances become counterexamples for the model.

The best way to build an assistant that understands your domain and the specific needs of your customers is to take the time to build good training data.

## Counterexample limits

The maximum number of counterexamples that you can create for any plan type is 25,000.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Advanced analysis and log-related tasks

Learn about APIs and other tools you can use to access and analyze log data.

### Using Jupyter notebooks for analysis

IBM created Jupyter notebooks that you can use to analyze the behavior of your assistant. A Jupyter notebook is a web-based environment for interactive computing. You can run small pieces of code that process your data, and you can immediately view the results of your computation.



**Note:** You can use the notebooks with English-language skills only.

### Analysis notebooks

There is a set of analysis notebooks that you can use with standard Python tools and a set that is designed for optimal use with IBM Watson® Studio.

Watson Studio is a product that provides an environment in which you can pick and choose the tools you need to analyze and visualize data, to cleanse and shape data, to ingest streaming data, or to create, train, and deploy machine learning models. See the [product documentation](#) for more details.

The [Watson Assistant Continuous Improvement Best Practices Guide](#) describes how to get the most out of these notebooks.

### Using the notebooks with Watson Studio

The following notebooks are available:

- [Dialog skill analysis for Watson Assistant](#)
- [Measure Watson Assistant Performance](#)
- [Analyze Watson Assistant Effectiveness](#)
- [Dialog Flow Analysis for Watson Assistant](#)

If you choose to use the notebooks that are designed for use with Watson Studio, the steps are roughly these:

1. Create a Watson Studio account, [create a project](#), and add a Cloud Object Storage account to it.
2. From the Watson Studio community, choose a notebook.

Early in the development process, use the **Dialog skill analysis for Watson Assistant** notebook to help you get started. It offers the following types of insights:

- Examines the terms that are correlated with each intent in your training data to find anomalies that might identify problems that you can investigate further.
- Uses a blind test set that you provide to calculate performance on statistical metrics like Accuracy, Precision, Recall & F1.
- Offers advanced features that you can use to find the causes of common issues such as why some sentences are often misidentified.

To learn more about how this notebook can help you improve your dialog, read this [Medium.com blog post](#).

3. After you deploy a version of the assistant, and have some conversation log data collected, run the **Measure Watson Assistant Performance** notebook.

4. Follow the step-by-step instructions provided with the notebooks to analyze a subset of the dialog exchanges from the logs.

Run the following notebook first:

- **Measure:** Gathers metrics that focus on coverage (how often the assistant is confident enough to respond to users) and effectiveness (when the assistant does respond, whether the responses are satisfying user needs).

The insights are visualized in ways that make it easier to understand areas for improvement in your assistant.

5. Export a sample set of the logs from ineffective conversations, and then analyze and annotate them.

For example, indicate whether a response is correct. If correct, mark whether it is helpful. If a response is incorrect, then identify the root cause, the wrong intent or entity was detected, for example, or the wrong dialog node was triggered. After identifying the root cause, indicate what the correct choice would have been.

6. Feed the annotated spreadsheet to the **Analyze Watson Assistant Effectiveness** notebook.

- **Effectiveness:** Performs a deeper analysis of your logs to help you understand the steps you can take to improve your assistant.

7. Use the **Dialog Flow Analysis for Watson Assistant** notebook to review your dialog. The notebook can help you pinpoint the dialog nodes where customers most frequently abandon the conversation.

For more information about how this notebook can help you analyze and assess abandonment, read this [Medium.com blog post](#).

This process helps you to understand the steps you can take to improve your assistant.

## Using the notebooks with standard Python tools

If you choose to use standard Python tools to run the notebooks, you can get the notebooks from GitHub.

- [Dialog Skill Analysis for Watson Assistant](#)
- [Watson Assistant Recommendation notebooks \(Measure and Analyze Effectiveness\)](#)
- [Watson Assistant Dialog Flow Analysis notebook](#)

Again, the [Watson Assistant Continuous Improvement Best Practices Guide](#) outlines which notebook to use at each stage of your improvement process.

## Using the logs API

You can use the `/logs` API to list events from the transcripts of conversations that occurred between your users and your assistant. For conversations created by using the v2 `/message` API, use the instance-level endpoint to [list log events in all workspaces](#), and then filter by Assistant ID. For more information about filtering logs, see [Filter query reference](#).



**Important:** The API logs messages that are exchanged in conversations that are defined by a dialog skill only.

The number of days that logs are stored differs by service plan type. See [Log limits](#) for details.

For a Python script you can run to export logs and convert them to CSV format, download the `export_logs_py.py` file from the [Watson Assistant GitHub](#) repository.

## Understanding logs-related terminology

First, review the definitions of terms that are associated with Watson Assistant logs:

- **Assistant:** An application - sometimes referred to as a 'chat bot' - that implements your Watson Assistant content.
- **Assistant ID:** The unique identifier of an assistant.
- **Conversation:** A set of messages consisting of the messages that an individual user sends to your assistant, and the messages your assistant sends back.
- **Conversation ID:** Unique identifier that is added to individual message calls to link related message exchanges together. App developers using the V1 version of the Watson Assistant API add this value to the message calls in a conversation by including the ID in the metadata of the context object.
- **Customer ID:** A unique ID that can be used to label customer data such that it can be subsequently deleted if the customer requests the removal of their data.
- **Deployment ID:** A unique label that app developers using the V1 version of the Watson Assistant API pass with each user message to help identify the deployment environment that produced the message.

- **Instance:** Your deployment of Watson Assistant, accessible with unique credentials. A {{site.data.keassistant\_classic\_shortnshort}} instance might contain multiple assistants.
- **Message:** A message is a single utterance a user sends to the assistant.
- **Skill ID:** The unique identifier of a skill.
- **User:** A user is anyone who interacts with your assistant; often these are your customers.
- **User ID:** A unique label that is used to track the level of service usage of a specific user.
- **Workspace ID:** The unique identifier of a workspace. Although any workspaces that you created before November 9 are shown as skills in the product user interface, a skill and a workspace are not the same thing. A skill is effectively a wrapper for a V1 workspace.

**Important:** The **User ID** property is *not* equivalent to the **Customer ID** property, though both can be passed with the message. The **User ID** field is used to track levels of usage for billing purposes, whereas the **Customer ID** field is used to support the labeling and subsequent deletion of messages that are associated with end users. Customer ID is used consistently across all Watson services and is specified in the **X-Watson-Metadata** header. User ID is used exclusively by the Watson Assistant service and is passed in the context object of each /message API call.

## Associating message data with a user for deletion

There might come a time when you want to completely remove a set of your user's data from a Watson Assistant instance. When the delete feature is used, then the Overview metrics will no longer reflect those deleted messages; for example, they will have fewer Total Conversations.

### Before you begin

To delete messages for one or more individuals, you first need to associate a message with a unique **Customer ID** for each individual. To specify the **Customer ID** for any message sent using the **/message** API, include the **X-Watson-Metadata: customer\_id** property in your header. You can pass multiple **Customer ID** entries with semicolon separated **field=value** pairs, using **customer\_id**, as in the following example:

```
$ curl -X POST -u "apikey:3Df... ...Y7Pc9" \
--header \
"Content-Type: application/json" \
"X-Watson-Metadata: customer_id={first-customer-ID};customer_id={second-customer-ID}" \
--data "{\"input\":{\"text\":\"hello\"}}" \
"{url}/v2/assistants/{assistant_id}/sessions/{session_id}/message?version=2019-02-28"
```

where {url} is the appropriate URL for your instance. For more details, see [Service endpoint](#).



**Note:** The **customer\_id** string cannot include the semicolon ( ; ) or equal sign ( = ) characters. You are responsible for ensuring that each **Customer ID** parameter is unique across your customers.

To delete messages using **customer\_id** values, see the [Information security](#) topic.

## Administrative tasks

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Backing up and restoring data](#).

### Backing up and restoring data

Back up and restore your data by downloading, and then uploading the data.

You can download the following data from a Watson Assistant service instance:

- Dialog skill training data (intents and entities)
- Dialog skill dialog
- Actions skill

You cannot download the following data:

- Search skill
- Assistant, including any configured integrations

### Retaining logs

If you want to store logs of conversations that users have had with your assistant, you can use the **/logs** API to export your log data. See [API reference](#) for details.



**Tip:** To get the workspace ID for a skill, from the skill tile, click the icon, and then choose **View API Details**.

Logs are stored for a different amount of time depending on your service plan. For example, Lite plans provide logs from the past 7 days only. See [Log limits](#) for more information.

You cannot import logs from one skill into another skill.

## Downloading a skill

To back up actions or dialog skill data, download the skill as a JSON file, and store the JSON file.

1. Find the actions or dialog skill tile on the Skills page or on the configuration page of an assistant that uses the skill.
2. Click the  icon, and then choose **Download**.
3. Specify a name for the JSON file and where to save it, and then click **Save**.

Alternatively, you can use the `/workspaces` API to export a dialog skill. Include the `export=true` parameter with the GET workspace request. See the [API reference](#) for more details.

## Uploading a skill

To reinstate a backup copy of an actions or dialog skill that you exported from another service instance or environment, create a new skill by importing the JSON file of the skill you exported.

 **Important:** If the Watson Assistant service changes between the time you export the skill and import it, due to functional updates which are regularly applied to instances in cloud-hosted continuous delivery environments, your imported skill might function differently than it did before.

1. Click the **Skills** icon .
2. Click **Create skill**.
3. Choose to create either an actions or dialog skill, then click **Next**.
4. Select the JSON file you want to import.

 **Important:** The imported JSON file must use UTF-8 encoding, without byte order mark (BOM) encoding. The JSON file cannot contain tabs, newlines, or carriage returns.

 **Tip:** The maximum size for a skill JSON file is 10MB. If you need to import a larger skill, consider using the REST API. For more information, see the [API Reference](#).

5. Click **Upload**.
- If you have trouble uploading a skill, see [Troubleshooting skill import issues](#).

6. Specify the details for the skill:
  - **Name:** A name no more than 100 characters in length. A name is required.
  - **Description:** An optional description no more than 200 characters in length.
  - **Language:** The language of the user input the skill will be trained to understand. The default value is English.

After you create the skill, it appears as a tile on the Skills page.

## Re-creating your assistant

You can now re-create your assistant. You can then link your uploaded skills to the assistant, and configure integrations for it.

See [Creating an assistant](#) and [Adding integrations](#) for more details.

## Update your client applications

When you import a dialog skill that you exported, a new skill is created. The new skill has a new workspace ID. If you have existing client applications that use the v1 API to access this skill, then you must update any workspace ID references to use the new workspace ID instead.

When you re-create your assistant, it is given a new assistant ID. If you have existing client applications that use the v2 API to access the assistant, then you must update any assistant ID references to use the new assistant ID instead.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Upgrading your plan](#).

## Upgrading

Learn how to upgrade your service plan.

## Upgrading your plan

You can explore the Watson Assistant [service plan options](#) to decide which plan is best for you.

The page header shows the plan you are using today.

IBM Watson Assistant Plus trial | 29 days left [Upgrade](#)

To upgrade your plan, complete these steps:

1. Do one of the following things:

- **Trial plan only:** The number of days that are left in your trial is displayed in the page header. To upgrade your plan, click **Upgrade** from the page header before your trial period ends.

- For all other plan types, click the  User icon from the page header, and then choose **Upgrade Plan** from the menu.

2. From here, you can see other available plan options. For most plan types, you can step through the upgrade process yourself.

- If you upgrade to an Enterprise with Data Isolation plan, you cannot do an in-place upgrade of your service instance. An Enterprise with Data Isolation plan instance must be provisioned for you first. After the service instance is provisioned, you must export your dialog skills from the old plan instance, and import them into the new plan instance. For more information, see [Backing up and restoring data](#).
- When you upgrade from a legacy Standard plan, you change the metrics that are used for billing purposes. Instead of basing billing on API usage, the Plus plan bases billing on the number of monthly active users. If you built a custom app to deploy your assistant, you might need to update the app. Ensure that the API calls from the app include user ID information. For more information, see [User-based plans explained](#).
- You cannot change from a Trial plan to a Lite plan.

For answers to common questions about subscriptions, see the [How you're charged](#).

Still have questions? [Fill out this form](#) to schedule a consultation with IBM.

## Enhancing security

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Managing access

You can give other people access to your Watson Assistant resources, and control the level of access they get.

Maybe you want one development team to have access to a test assistant and another development team to have access to a production assistant. And you want data scientists to be able to view analytics for user conversation logs from both assistants. And maybe you want a writer to be able to author the dialogue that is used by your assistant to converse with your customers. To manage who can do what with your skills and assistants, you can assign different access roles to different people.

### Before you grant access to others

For each person to whom you grant access to your Watson Assistant service instance, decide whether you want to give the person a role with instance-level or resource-level access. Instance-level access applies to all of the assistants and skills in a single service instance. Resource-level access applies to individual skills and assistants within a service instance only.

### Granting users access to your resources

1. If you plan to give a user access to a single skill or assistant in your service instance, get the ID for the skill or assistant. You need to provide the ID in a later step.

- To get the assistant ID, go to the Assistants page. Click the overflow menu for the assistant, and then click **Settings > API Details**. Copy the assistant ID and paste it somewhere that you can access it from later.
- To get the skill ID, go to the Skills page. Click the overflow menu for the skill, and then click **View API Details**. Copy the skill ID and paste it somewhere that you can access it from later.

2. Click the User  icon in the page header.

3. Make a note of the current instance name, and then click **Manage Users**.

You are directed to the IBM Cloud® Identity and Access Management (IAM) page.

4. Click **Invite users**, and then enter the email addresses of the people to whom you want to grant access.



**Tip:** If you plan to grant specific resource-level access to different team members, then add one person at a time.

5. Expand **Assign users additional access**, and then click **IAM services**.
  6. In the **Which service do you want to assign access to?** field, choose **Watson Assistant**.
  7. Optionally select a specific region and service instance to share with this user.
- Otherwise, the user access you define is applied to all of your services instances in all data center locations where you have instances.
- Remember, you made a note of the name of the current service instance in an earlier step.
8. Optionally, indicate that you want to give this user access to a specific skill or assistant only by selecting an option from the **Resource Type** field. And then paste the associated ID (that you copied in an earlier step) into the **Assistant or Skill ID** field.
  9. Assign the person to the following role types for the service instance:

- Platform
- Service

For help, see [Popular role assignments](#). For a description of all your options, see [Understanding roles](#).

10. **Optional:** Select a single resource, such as a skill or assistant, and then assign the person to the appropriate service role for the resource.

For a description of your resource-level service options, see [Resource-level access roles](#).

11. Click **Add**.

12. Click **Invite** to finish the process.



**Tip:** When more people contribute to the development of a dialog skill, unintended changes can occur, including skill deletions. Only one person can edit an intent, entity, or a dialog node at a time. If multiple people work on the same item at the same time, then the changes that are made by the person who saves their changes last are the only changes applied. Changes that are made at the same time by someone else and that are saved first are not retained. Coordinate the updates that you plan to make with your team members to prevent anyone from losing their work. Also, consider creating backup copies of your dialog skill regularly, so you can roll back to an earlier version if necessary. To create a backup, [download the skill as a JSON file](#). If you use an actions skill, each team member can work on a single action at a time.

## Popular role assignments

To get you started quickly, consider applying these roles at first.

- To give someone the same level of access to a Watson Assistant service instance as you have, assign the user to the following roles:
  - Instance platform role: **Administrator**
  - Instance service role: **Manager**
- To give someone manager access to a specific set of resources, but read-only access to everything else, assign the user to the following roles:
  - Instance platform role: **Viewer**
  - Instance service role: **Reader**

Then, for each assistant or skill that you want the person to be able to manage, add another role assignment like this:

- Instance platform role: **Viewer**
- For an individual assistant or skill, assign the **Manager** role.

## Understanding roles

Access to Watson Assistant, its service instances and all of the resources that are used by the service is managed by IBM Cloud®. To share a service instance with other users and control access to the instance, you must use the IBM Cloud Identity and Access Management user interface.

IBM Cloud breaks down access definitions into the following role types:

- Instance-level platform roles
- Instance-level service roles
- Resource-level service roles

An instance-level access role is a role that applies to a service instance and all of the resources that are created as part of the instance. A resource-level service role gives you the ability to apply a finer-tuned level of access control to a single skill or assistant in an instance.

## Platform roles

A platform role controls a person's ability to access a service instance in IBM Cloud. Choose a platform role to assign to the user.

Role	Privileges of the users in this role
------	--------------------------------------

<b>Administrator</b>	Access and change all resources for the instance, and invite other people to access the instance.
<b>Viewer</b>	View the service instance, but cannot invite others to access the instance.

Table 1. Global platform role details

The **Editor** and **Operator** platform roles are equivalent to the Administrator role.



**Important:** At a minimum, you must give someone *Viewer* platform access to a service instance or they cannot access anything.

## Service roles

A service role controls what a person can do in Watson Assistant.

When you assign a service role, you can decide whether to apply the service role to an instance, which effectively applies to every assistant and skill in the instance, or to a specific resource, such as one skill or one assistant.

The highest-level role assignment wins. You can give someone Reader access to everything in an instance, and Manager access to a single skill in that instance. But, you cannot do the opposite. You cannot give a person Manager access to all but one skill. If you assign someone to the Manager role at the instance level, you cannot limit the person's access to a resource in that instance by assigning the person to a Reader level access for a single skill, for example.

## Instance-level service roles

Choose a service role for the user that will apply to all of the skills and assistants in the instance.

Role	Privileges of the users in this role
<b>Reader</b>	Open and read all assistants and skills in the service instance, but not edit them.
<b>Writer</b>	Read, edit, delete, or create assistants and skill in the service instance.
<b>Manager</b>	Read, edit, delete, or create assistants and skills in the service instance; view all conversation logs; access the v1 API for all skills in the instance.

Table 2. Global service role details

## Resource-level service roles

To fine tune a person's level of access to individual skills and assistants within the instance, assign one or more resource-level service roles to the user.

You do not need to assign resource-level roles to people who already have access to all the resources based on their global service role. But, if you want to tailor a bit more who can read, edit, or add to specific assistants or skills within the instance, then apply resource-level roles.



**Important:** You cannot assign someone to a resource-level role that has fewer privileges than the instance-level role to which the person is assigned. If the person has an instance-level Manager access role, for example, you cannot give them read-only access to a specific assistant. The higher-level role assignment wins.

Role	Privileges of the users in this role
Reader	Open and read the assistant or skill, but not edit it.
Writer	Read, edit, delete, or create an assistant or skill.
Manager	Read, edit, delete, or create an assistant or skill, and view conversation logs.

Table 3. Resource service role details

Anyone who creates an assistant or skill is automatically granted the Manager service role to that resource.

## Service access examples

- Reader

The following screen capture shows you what someone with Reader service access to a resource sees on the Dialog page.

This screenshot shows the IBM Watson Assistant interface for a skill named "My first skill". The navigation bar at the top includes "IBM Watson Assistant", a user icon, and three question mark icons. The main area displays the "Dialog" section for the skill. On the left, a sidebar lists "Intents", "Entities", "Dialog" (which is selected), "Options", "Versions", and "Content Catalog". In the center, there are two dialog nodes: "Welcome" (with intent "welcome") and "Anything else" (with intent "anything\_else"). Each node has a status bar below it indicating "1 Responses / 0 Context Set / Does not return". At the bottom of the central area, there are buttons for "Add node", "Add child node", and "Add folder". To the right, a "Try it out" pane shows a message input field with "Hello. How can I help you?" and a location pin icon. Below the input field is a button labeled "Enter something to test your assistant". A note at the bottom of the pane says "Use the up key for most recent". At the very bottom of the interface, a dark bar indicates "Permission: Read Only".

**Note:** Most buttons, such as **Add node**, **Add folder**, and **Save new version** are disabled. There is no Analytics option in the navigation. The "Try it out" pane is available.

- Writer

This screen capture shows you what someone with Writer service access to a resource sees on the same page.

This screenshot shows the same IBM Watson Assistant interface as the previous one, but with Writer service access. The "Add node" button in the central toolbar is now highlighted in blue, indicating it is enabled. All other buttons ("Add child node" and "Add folder") remain greyed out. The rest of the interface, including the sidebar, dialog nodes, and "Try it out" pane, appears identical to the Reader access view.

**Note:** All of the buttons are available. There is no Analytics option in the navigation.

## Common role assignments

Goal	Instance-level platform role	Instance-level service roles	Resource-level service roles
Make someone a service instance co-owner	Administrator	Manager	N/A
Give someone reader access to only one skill or assistant in an instance	Viewer, Reader	Reader	Reader

Allow someone to edit and view logs of all the skills and assistants in the service instance	Viewer	Manager	N/A
Allow someone to edit a skill or assistant in the service instance, but not view logs	Viewer, Reader	Writer	Writer
Give someone full access to only one skill or assistant in an instance	Viewer, Reader	Reader	Manager

Table 4. Common role assignments

N/A stands for no assignment, meaning no role of the type is assigned.

## Resource-level role impact on available actions

The Watson Assistant user interface and API comply with the access roles that are defined for a service instance. When someone logs in to the user interface, it adjusts to show only what the current user can access, and it disables functions that the user does not have permissions to do. (For example, a person who is assigned to the Reader role for a dialog skill cannot create or edit entities, intents, or dialog nodes in the skill because the **Create {resource}** and **Edit** functions are disabled.) Similarly, the API allows access only to resources and methods that are permitted for the role that is associated with the specified API key.



**Note:** If you cannot access the API Details page for a skill or assistant, you might not have the access that is required to use the instance-level API credentials. You can use a personal API key instead. For more information, see [Getting API information](#).

The following table shows the UI and API actions that can be performed by different resource-level service roles.

Action	Reader	Writer	Manager
Open and view an assistant or skill	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
View API details for an assistant or skill			<input type="checkbox"/>
Create an assistant or skill	<input type="checkbox"/>		<input type="checkbox"/>
Rename, edit, delete an assistant or skill	<input type="checkbox"/>	<input type="checkbox"/>	
Open, view, create, edit, or delete an integration	<input type="checkbox"/>	<input type="checkbox"/>	
Add, swap, or remove skills	<input type="checkbox"/>	<input type="checkbox"/>	
Swap skill versions for an assistant	<input type="checkbox"/>		<input type="checkbox"/>
Revert to a previous dialog skill version	<input type="checkbox"/>		<input type="checkbox"/>
Save or delete dialog skill versions	<input type="checkbox"/>		<input type="checkbox"/>
Export a skill or dialog skill version	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Duplicate a skill	<input type="checkbox"/>		<input type="checkbox"/>
Change bidirectional preferences	<input type="checkbox"/>		<input type="checkbox"/>
Open and view intents and intent examples	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Import, add (including from content catalog), edit, or delete intents or intent examples		<input type="checkbox"/>	<input type="checkbox"/>
Export intents	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Manage intent conflicts	<input type="checkbox"/>		<input type="checkbox"/>
Open and view entities	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Import, add, edit, or delete entities	<input type="checkbox"/>	<input type="checkbox"/>
Export entities	<input type="checkbox"/>	<input type="checkbox"/>
Enable system entities and fuzzy matching	<input type="checkbox"/>	<input type="checkbox"/>
Add contextual entities	<input type="checkbox"/>	<input type="checkbox"/>
Import, add, edit, or delete actions or action examples	<input type="checkbox"/>	<input type="checkbox"/>
Edit system actions	<input type="checkbox"/>	<input type="checkbox"/>
Open and view dialog nodes or action steps	<input type="checkbox"/>	<input type="checkbox"/>
Add, move, edit, or delete dialog nodes, dialog folders, or action steps	<input type="checkbox"/>	<input type="checkbox"/>
Change dialog node or action step settings	<input type="checkbox"/>	<input type="checkbox"/>
Test and set context in dialog skill "Try it out" pane	<input type="checkbox"/>	<input type="checkbox"/>
Test in actions skill Preview pane	<input type="checkbox"/>	<input type="checkbox"/>
Test in search skill "Try it out" pane	<input type="checkbox"/>	<input type="checkbox"/>
Add, edit, or delete context variables in dialog skills or session variables in actions skills	<input type="checkbox"/>	<input type="checkbox"/>
Improve data from dialog skill "Try it out" pane	<input type="checkbox"/>	<input type="checkbox"/>
View analytics	<input type="checkbox"/>	<input type="checkbox"/>
View intent and intent example recommendations	<input type="checkbox"/>	
v1 runtime API ( <code>/message</code> endpoint)	<input type="checkbox"/>	<input type="checkbox"/>
v2 runtime API ( <code>session</code> and <code>/message</code> endpoints)	<input type="checkbox"/>	<input type="checkbox"/>
v1 authoring API (all but <code>/message</code> endpoint)	<input type="checkbox"/>	
v1 and v2 logs API	<input type="checkbox"/>	

Table 5. Action privileges per service role

## Access control improvements

Before April 2020, the service access role assignments that were defined for a Watson Assistant service instance in the IBM Cloud dashboard were not applied. Users with either Reader- or Writer-level service access to an instance effectively had Manager-level access. This behavior changed for all data centers as of 2 April 2020. With the access control improvements released, service-level access role assignments are recognized by the product.

## How to keep your access

Now that service level access control is supported for instances in your data center, if you have the wrong service level access role to an instance, you might be unable to do things you could do before.

- If you have Manager service access to a Watson Assistant service, then no action is required. You have the same privileges and can do the same actions within the product now that service-level access control is enabled.
- If you have a Reader or Writer service access role, then what you can do has changed.

## I'm a Reader

With a Reader role, you cannot take any actions in the Watson Assistant application user interface. You can view only the assistants and skills pages. You cannot edit, create, or delete anything, and cannot view analytics.

## I'm a Writer

With a Writer role, you can view, edit, create, and delete resources, such as assistants or skills. However, you cannot view analytics or user conversation log information. Only people with a Manager service access role can view the analytics page.

Act now to get the service instance owner to change your access level.

1. Contact the owner of the service instance.



**Tip:** The email address of the owner of the service instance is displayed in the User account menu. Click the from the page header.



2. Ask the owner of the service instance to change your service access role assignment.

You can copy, edit, and then paste the following text into an email or message that you send to the instance owner.

```
You are the owner of the {service instance name} {{site.data.keyword.assistant_classic_short}} service instance.
I am currently a {role-you-have} of the instance.
Please change my service access role to {role-you-want}.
For instructions, see https://cloud.ibm.com/docs/assistant?topic=assistant-access-control#access-control-admin-prep.
```

## Prevent your collaborators from losing their access

When you create a Watson Assistant service instance, you are assigned to a Manager service access role automatically. Your access has not changed.

If you invited people to your service instance and assigned them to a Reader or Writer service access role, then perform the following steps now.

To change a collaborator's service access role:

1. Click the User icon in the page header.
2. Make a note of the current service instance name, and then select **Manage Users** from the drop-down.
3. From the navigation pane, click **Users**.

People that you invited to collaborate with you on any service instance are listed.

4. To see the person's level of access to an instance, click the user's email address, and then click the **Access policies** tab.

A list of your instances that this user can access is displayed.

The screenshot shows the 'Access policies' tab of a user profile. The tab bar includes 'User details', 'Access groups', 'Access policies' (which is active), and 'Cloud Foundry'. A note at the top says, 'Based on your assigned role, you can click the role to view or edit the policy.' Below is a table with columns 'Role', 'Policy Details', and a 'More' column with 'Edit' and 'Remove' buttons. The table rows show:

Role	Policy Details	More
Operator	Watson Assistant service	⋮
Operator, Writer	Watson Assistant service Region string equals us-east, Service Instance string equals Watson Assistant East	⋮
Viewer	default resource group resourceType string equals resource-group, resource string equals default	⋮

5. Find the instance for which you want to change this person's access. (The service instance name is referred to as the `serviceInstance string`.)

The user's platform access role is listed first. The user's service access role is listed second.



**Note:** If only one role is displayed, then you might not have assigned a service access role to the user when you invited them to the instance. Complete the next step to assign a service access role to the user now.

6. To change the user's role, click the *list of actions* icon for the entry, and then click **Edit**.
7. Click the appropriate service access role checkbox.

Region

Washington DC

Service Instance

string equals Watson Assistant East (6a)

Resource Type

string equals Resource Type

Assistant or Skill ID

string equals

Viewer      12 As a viewer, you can view service instances, but you can't modify them.

Operator      20 As an operator, you can perform platform actions required to configure and operate service instances, such as viewing a service's dashboard.

Editor      29 As an editor, you can perform all platform actions except for managing the account and assigning access policies.

Administrator      38 As an administrator, you can perform all platform actions based on the resource this role is being assigned, including assigning access policies to other users.

Reader      11 As a reader, you can perform read-only actions within a service such as viewing service-specific resources.

Writer      13 As a writer, you have permissions beyond the reader role, including creating and editing service-specific resources.

Manager      15 As a manager, you have permissions beyond the writer role to complete privileged actions as defined by the service. In addition, you can manage other users and their access to the service.

8. Click **Save**.

The page closes and a confirmation message is displayed to indicate that the policy change was successfully saved.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Securing your assistant](#).

## Securing your assistant

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](#).
- Outside the European Union, see [Requesting support for resources outside the European Union](#).

## 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](#).

## Health Insurance Portability and Accountability Act (HIPAA)

US Health Insurance Portability and Accountability Act (HIPAA) support is available for Enterprise plans that are hosted in the Washington, DC or Dallas locations. For more information, see [Enabling HIPAA support for your account](#).

Do not add personal health information (PHI) to the training data (entities and intents, including user examples) that you create. In particular, be sure to remove any PHI from files that contain real user utterances that you upload to mine for intent or intent user example recommendations.

## Opting out of log data use

IBM uses log data, Enterprise plan data excluded, to continually learn from and improve the Watson Assistant product. The logged data is not shared or made public.

To prevent IBM from using your log data for general service improvements, complete one of the following tasks:

- If you are using a custom application, for each API `/message` request, set the `X-Watson-Learning-Opt-Out` header parameter to `true`.  
For more information, see [Data collection](#).
- If you are using the web chat integration, add the `learningOptOut` parameter to the script that you embed in your web page, and set it to `true`.  
For more information, see [Configuration](#).

## Labeling and deleting data in Watson Assistant

Do not add personal data to the training data (actions and steps, including user examples) that you create. In particular, be sure to remove any personally-identifiable information from files that contain real user utterances that you upload to mine for user example recommendations.

Experimental and beta features are not intended for use with a production environment and therefore are not guaranteed to function as expected when labeling and deleting data. Experimental and beta features should not be used when implementing a solution that requires the labeling and deletion of data.

If you need to remove a customer's message data from a Watson Assistant instance, you can do so based on the customer ID of the client, as long as you associate the message with a customer ID when the message is sent to Watson Assistant.



**Note:** Removing message data must be an occasional event only for individual customer IDs. To disable analytics logs, you can upgrade to an Enterprise with Data Isolation plan.

- The assistant preview and automatic Facebook integration do not support the labeling and therefore deletion of data based on customer ID. They should not be used in a solution that must support the ability to delete data based on a customer ID.
- For Intercom, the `customer_id` is the `user_id` prepended with `intercom_`. The Intercom `user_id` property is the `id` of the `author` message object in the Conversation Model that is defined by Intercom.
  - To get the ID, open the channel from a web browser. Open the web developer tools to view the console. Look for `author`.

The full customer ID looks like this: `customer_id=intercom_5c499e5535ddf5c7fa2d72b3`.

- For Slack, the `customer_id` is the `user_id` prepended with `slack_`. The Slack `user_id` property is a concatenation of the team ID, such as `T09LVDR7Y`, and the member ID of the user, such has `W4F8K9JNF`. For example: `T09LVDR7Yw4F8K9JNF`.
  - To get the team ID, open the channel from a web browser. Open the web developer tools to view the console. Look for `[BOOT] Initial team ID`.
  - You can copy the member ID from the user's Slack profile.
  - To get the IDs programmatically, use the Slack API. For more information, see [Overview](#). The full customer ID looks like this: `customer_id=slack_T09LVDR7Yw4F8K9JNF`.
- For the web chat integration, the service takes the `user_id` that is passed in and adds it as the `customer_id` parameter value to the `X-Watson-Metadata` header with each request.

## Before you begin

To be able to delete message data associated with a specific user, you must first associate all messages with a unique `customer ID` for each user. To specify the `customer ID` for any messages sent using the `/message` API, include the `X-Watson-Metadata: customer_id` property in your header. For example:

```
$ curl -X POST -u "apikey:3Df... ...Y7Pc9"
--header
'Content-Type: application/json'
'X-Watson-Metadata: customer_id=abc'
--data
'{"input":{"text":"hello"}}
'{url}/v2/assistants/{assistant_id}/sessions/{session_id}/message?version=2019-02-28'
```

where `{url}` is the appropriate URL for your instance. For more details, see [Service endpoint](#).



**Note:** The `customer_id` string cannot include the semicolon (`;`) or equal sign (`=`) characters. You are responsible for ensuring that each

**customer ID** property is unique across your customers.

Only the first **customer ID** value that is passed in the **X-Watson-Metadata** header is used as the **customer\_id** string for the message log. This **customer ID** value can be deleted with **DELETE /user\_data** v1 API calls.

If you add search to an assistant, user input that is submitted to the assistant is passed to the Discovery service as a search query. If the Watson Assistant integration provides a customer ID, then the resulting **/message** API request includes the customer ID in the header, and the ID is passed through to the Discovery **/query** API request. To delete any query data that is associated with a specific customer, you must send a separate delete request directly to the Discovery service instance that is linked to your the assistant. See the Discovery [information security](#) topic for details.

## Querying user data

Use the v1 **/logs** method **filter** parameter to search an application log for specific user data. For example, to search for data specific to a **customer\_id** that matches **my\_best\_customer**, the query might be:

```
$ curl -X GET -u "apikey:3Df... ...Y7Pc9" \
"{url}/v2/assistants/{assistant_id}/logs?version=2020-04-01&filter=customer_id::my_best_customer"
```

where {url} is the appropriate URL for your instance. For more details, see [Service endpoint](#).

See the [Filter query reference](#) for additional details.

## Deleting data

To delete any message log data associated with a specific user that your assistant might have stored, use the **DELETE /user\_data** v1 API method. Specify the customer ID of the user by passing a **customer\_id** parameter with the request.

Only data that was added by using the **POST /message** API endpoint with an associated customer ID can be deleted using this delete method. Data that was added by other methods cannot be deleted based on customer ID. For example, entities and intents that were added from customer conversations, cannot be deleted in this way. Personal Data is not supported for those methods.

**IMPORTANT:** Specifying a **customer\_id** will delete *all* messages with that **customer\_id** that were received before the delete request, across your entire Watson Assistant instance, not just within one skill.

As an example, to delete any message data associated with a user that has the customer ID **abc** from your Watson Assistant instance, send the following cURL command:

```
$ curl -X DELETE -u "apikey:3Df... ...Y7Pc9" \
"{url}/v2/user_data?customer_id=abc&version=2020-04-01"
```

where {url} is the appropriate URL for your instance. For more details, see [Service endpoint](#).

An empty JSON object **{}** is returned.

For more information, see the [API reference](#).

**Note:** Delete requests are processed in batches and may take up to 24 hours to complete.

## Web chat usage data

The Watson Assistant web chat sends limited usage data to the [Amplitude service](#). When the web chat widget is being interacted with by a user, we track the features that are being used, and events such as how many times the widget is opened and how many users start conversations. This information does not include Assistant training data or the content of any chat interactions. The information being sent to Amplitude is not Content as defined in the Cloud Service Agreement (CSA); it is Account Usage Information as described in Section 9.d of the CSA and is handled accordingly as described in the [IBM Privacy Statement](#). The purpose of this information gathering is limited to establishing statistics about use and effectiveness of the web chat and making general improvements.

## Private network endpoints

You can set up a private network for Watson Assistant instances that are part of a Plus or Enterprise service plan. Using a private network prevents data from being transferred over the public internet, and ensures greater data isolation.

**Plus** This feature is available only to users of paid plans.

Private network endpoints support routing services over the IBM Cloud private network instead of the public network. A private network endpoint provides a unique IP address that is accessible to you without a VPN connection.

For implementation details, see [Public and private network endpoints](#).

## Important private network endpoint notes

The integrations that are provided with the product require endpoints that are available on the public internet. Therefore, any built-in integrations you add to your assistant will have public endpoints. If you only want to connect to a client application or messaging channel over the private network, then you must build your own custom client application or channel integration.

## Related topics

- [Security architecture](#): Describes the security components that are needed for secure cloud development, deployment, and operations.
- [IBM Cloud compliance programs](#): Describes how to manage regulatory compliance and internal governance requirements with IBM Cloud services.

## Advanced development



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Advanced development tasks

A developer can extend the capabilities of your assistant by using tools that are provided within the product or the product APIs or a combination of both.

- [Using webhooks](#)
- [Dialog skill development](#)
- [Web chat development](#)
- [Custom user interface development](#)

## Using webhooks

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Extending your assistant with webhooks](#).

### Webhook overview

Make a call to an external service or application during a conversation.

Watson Assistant supports the following types of webhooks:

- Skill-level webhooks

The following type of webhook can be set up for use from a dialog skill:

- Assistant-level webhooks
  - [Dialog](#)

The following types of webhooks can be set up for an assistant:

- [Logs](#)
- [Premessage](#)
- [Postmessage](#)

### Which type of webhook should I use?

The skill-level webhook is different from the assistant-level webhooks in the following ways:

- Frequency with which the webhooks are called

The dialog webhook is called on the rare occasion that the dialog node from which it is triggered is processed.

The message processing webhooks are called with every single exchange in a conversation between the customer and the assistant.

The log webhook is called with each message and its corresponding response.

- Where the condition is defined

For a dialog webhook, the condition to meet before an action is taken is defined in the dialog skill. If the node condition is not met, then the dialog webhook is never called.

For the message processing webhooks, the condition to check for before taking an action must be defined in the external application code. For example, even if your webhook performs a simple language translation, you'd want to use a condition to check the language of the incoming message before sending the text to the translation service.

You don't need to define a condition for the log webhook unless you want to filter the messages somehow. In most cases, the goal is to write out every message that is submitted, so the messages can be stored for as long as you want, and analyzed by an external application or service.

- Where you configure them

You configure the dialog webhook from the the **Options>Webhook** page of the dialog skill. You then initialize it from one or more dialog nodes by customizing the node.

You configure the assistant-level webhooks from the **Settings>Webhooks** page for the assistant.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Making a programmatic call from dialog](#).

## Making a programmatic call from dialog

To make a programmatic call, define a webhook that sends a POST request callout to an external application that performs a programmatic function. You can then invoke the webhook from one or more dialog nodes.

A webhook is a mechanism that allows you to call out to an external program based on something happening in your program. When used in a dialog skill, a webhook is triggered when the assistant processes a node that has a webhook enabled. The webhook collects data that you specify or that you collect from the user during the conversation and save in context variables. It sends the data as part of a HTTP POST request to the URL that you specify as part of your webhook definition. The URL that receives the webhook is the listener. It performs a predefined action using the information that you pass to it as specified in the webhook definition, and can optionally return a response.

You can use a webhook to do the following types of things:

- Validate information that you collected from the user.
- Interact with an external web service to get information. For example, you might check on the expected arrival time for a flight from an air traffic service or get a forecast from a weather service.
- Send requests to an external application, such as a restaurant reservation site, to complete a simple transaction on the user's behalf.
- Trigger a SMS notification.
- Trigger an IBM Cloud® Functions web action.



**Important:** You cannot use a webhook to call a Cloud Functions action that uses token-based Identity and Access Management (IAM) authentication. However, you can make a call to a secured Cloud Functions web action. For more information, see [Calling an IBM Cloud Functions web action](#).

For information about how to call a client application, see [Calling a client application from a dialog node](#).



**Note:** For environments where private endpoints are in use, keep in mind that a webhook sends traffic over the internet. For more information, see [Private network endpoints](#).

## Defining the webhook

You can define one webhook URL for a dialog skill, and then call the webhook from one or more dialog nodes.

The programmatic call to the external service must meet these requirements:

- The call must be a POST HTTP request.
- The request body must be a JSON object (**Content-Type: application/json**).
- The response must be a JSON object (**Accept: application/json**).
- The call must return in **8 seconds or less**. If invoked more than once in a single message call through [dialog nodes](#), all of those invocations must return in 8 seconds or less.



**Tip:** If your external service supports only GET requests, or if you need to specify URL parameters dynamically at run time, consider creating an intermediate service that accepts a POST request with a JSON payload containing any runtime values. The intermediate service can then make a request to the target service, passing these values as URL parameters, and then return the response to the dialog.



**Tip:** If you need to call a service that might not return within 8 seconds, you can manage the call through a custom client application and pass the information to the dialog as a separate step. For more information, see [Calling a client application from a dialog node](#).

To add the webhook details, complete the following steps:

1. From the skill where you want to add the webhook, click the **Options** tab.
2. Click **Webhooks**.
3. In the **URL** field, add the URL for the external application to which you want to send HTTP POST request callouts.

For example, to call the Language Translator service, specify the URL for your service instance.

```
https://api.us-south.language-translator.watson.cloud.ibm.com/v3/translate?version=2018-05-01
```

If the external application that you call returns a response, it must be able to send back a response in JSON format. For the Language Translator

service, for example, you must specify the format in which you want the result to be returned. You can do so by passing a header to the service.

4. In the Headers section, add any headers that you want to pass to the service one at a time by clicking **Add header**.

For example, this header indicates that the request is in JSON format.

Header name	Header value
Content-Type	application/json
Header example	

5. If the external service requires that you pass basic authentication credentials with the request, then provide them. Click **Add authorization**, add your credentials to the **User name** and **Password** fields, and then click **Save**.

The product creates a base-64 encoded ASCII string from the credentials and generates a header that it adds to the page for you.

Header name	Header value
Authorization	Basic ``
Header example	

 **Tip:** If you use the web chat integration and enable security, you can use the same token you use to secure the web chat in the Authorization header. For more information, see [Web chat: Reusing the JWT for webhook authentication](#).

Your webhook details are saved automatically.

## Adding a webhook callout to a dialog node

To use a webhook from a dialog node, you must enable webhooks on the node, and then add details for the callout.

1. Click the **Dialog** tab.
2. Find the dialog node where you want to add a callout. The callout to the webhook will occur whenever this node is triggered during a conversation with a user.  
For example, you might want to send a callout to the webhook from the `#General_Greetings` node.
3. Click to open the dialog node, and then click **Customize**.
4. Scroll down to the webhook section. Set the **Callout to webhooks/actions skill** switch to **On**.
5. Select **Call a webhook**, and then click **Apply**.

 **Note:** If you did not have it enabled already, the **Multiple conditioned responses** switch is set to **On** automatically and you cannot disable it. This setting is enabled to support adding different responses depending on the success or failure of the Webhook call. If you had a response specified for the node already, it becomes the first conditional response.

6. Add any data that you want to pass to the external application as key and value pairs in the **Parameters** section.

 **Note:** Parameters are passed as request body properties. You cannot specify query parameters or URL parameters in a dialog node. These parameters can only be configured with static values as part of the webhook definition. For more information, see [Defining the webhook](#).

For example, if you call the Language Translator service, you must provide values for the following parameters:

Key	Value	Description
model_id	"en-es"	Identifies the input and output languages. In this example, the request is for text in English (en) to be translated into Spanish (es).
text	"How are you?"	This parameter contains the text string that you want the service to translate. You can hard code this value, pass a context variable, such as \$saved_text, or pass the user input to the service directly, by specifying `` as this value.
Parameter example		

In more complex use cases, you might collect information during a conversation with a user about their travel plans, for example. You can collect dates and destination information and save it in context variables that you can pass to an external application as parameters.

Key	Value
depart_date	\$departure
arrive_date	\$arrival
origin	\$origin
destination	\$destination

#### Travel parameters example

- Any response made by the callout is saved to the return variable. You can rename the variable that is automatically added to the **Return variable** field for you. If the callout results in an error, this variable is set to `null`.

The generated variable name has the syntax `webhook_result_n`, where the appended `n` is incremented each time you add a webhook callout to a dialog node. This naming convention ensures that context variable names are unique across the dialog skill. If you change the name, be sure to use a unique name.

- In the conditional responses section, two response conditions are added automatically, one response to show when the webhook callout is successful and a return variable is sent back. And one response to show when the callout fails. You can edit these responses, and add more conditional responses to the node.

- If the callout returns a response, and you know the format of the JSON response, then you can edit the dialog node response to include only the section of the response that you want to share with users.

For example, the Language Translator service returns an object like this:

```
{
 "translations": [
 {"translation": "¿Cómo estás?"}
],
 "word_count": 3,
 "character_count": 12
}
```

Use a SpEL expression that extracts only the translated text value.

Condition	Response
\$webhook_result_1	Your words in Spanish: .
anything_else	The call to the external application failed. Please try again later.

#### Conditional responses example

If you use the recommended format for the response and the translation response shown earlier is returned, the assistant's response to the user would be: `Your words in Spanish: ¿Cómo estás?`

- If you want to provide a specific response if the callout returns an empty string, meaning the call is successful, but the value returned is an empty string, you can add a conditional response that has a condition with syntax like this:

```
$webhook_result_1.size() == 0
```

- When you are done, click the X to close the node. Your changes are automatically saved.

## Testing webhooks

When you first add a webhook callout, it can be useful to see exactly what is returned in the response from the external application, the data and its format. To do this, add this expression as the text response for the successful callout conditional response: `$webhook_result_n` where `n` is the appropriate number for the webhook you are testing.

This response returns the full body of the return variable, so you can see what the callout is sending back and decide what to share with the user. You can then use the methods documented in [Expression language methods](#) to extract only the information you care about from the response.

Test whether certain user inputs can generate errors in the callout, and build in ways to handle those situations. Errors generated by the external application are stored in `output.webhook_error.<result_variable>`. You can use a conditional response like this while you are testing to capture such errors:

Condition	Response
output.webhook_error	The callout generated this error: .

For example, you might not be authenticating the request properly (401), or you might be trying to pass a parameter with a name that is already in use by the external application. Test the webhook to discover and fix these types of errors before you deploy the webhook.

## Removing a webhook

If you decide you do not want to make a webhook call from a dialog node, open the node's *Customize* page, and then switch Webhooks **Off**.

The *Parameters* section and the **Return variable** field are removed from the dialog node editor. However, any conditional responses that were added for you or that you added yourself remain.

The *Multiple conditioned responses* section is editable again. You can choose to switch the feature off. If you do so, then only the first conditional response is saved as the node's only text response.

To change the external service that you call from dialog nodes, edit the webhook details defined on the Webhooks page of the **Options** tab. If the new service expects different parameters to be passed to it, be sure to update any dialog nodes that call it.

## Calling an IBM Cloud Functions web action

Typically, web actions can be run without requiring the caller to authenticate first. However, you can secure a web action that requires any callers to pass an ID with the request. For more information about how to secure a web action, see [Securing web actions](#).

The following tips will help you call a Cloud Functions web action from your dialog.

1. Open the **Options** page for the skill, and then click **Webhooks**.
2. In the **URL** field, add the URL for the external application to which you want to send HTTP POST request callouts.

For example, to call a Cloud Functions web action, specify the URL for the public web action. For example:

```
https://us-south.functions.cloud.ibm.com/api/v1/web/my_org_dev/default>Hello%20World.json
```

If the external application that you call returns a response, it must be able to send back a response in JSON format.

 **Tip:** Notice the request URL in this example ends in `.json`. By specifying this extension, you take advantage of a feature of web actions that lets you specify the desired content type of the response. Specifying this extension type ensures that, if the web actions can return responses in more than one format, a JSON response will be returned. See [Extra features](#) for more details.

3. If the web action is secured, specify any headers, such as `X-Require-Whisk-Auth`, that are required to call the web action.

Header name	Header value
'X-Require-Whisk-Auth'	'{my-secret}'

Header example

4. Click the **Dialog** tab.
5. Click to open the dialog node from which you want to call the web action, and then click **Customize**.
6. Scroll down to the webhook section. Set the **Callout to webhooks** switch to **On**, and then click **Apply**.
7. Add any data that you want to pass to the external application as key and value pairs in the *Parameters* section.

For example, if you call the Hello World Cloud Functions web action, you might want to add the following information to pass in the message parameter that is accepted by that application:

Key	Value
message	"hello"

Parameter example

 **Note:** When calling a Cloud Functions web action, you cannot pass parameters with the same key as parameters that are defined as part of the web action. See [Protected parameters](#) for more details.

8. You can edit the dialog node response to include only the section of the response that you want to display to users.

The Hello World Cloud Functions web action includes a message name and value pair in its response, along with other information. To show only the content of the message, you can use `<return-variable>.message` syntax to extract the message section only from the response object.

Condition	Response
\$webhook_result_1	The application returned "\$webhook_result_1.message".
anything_else	The call to the external application failed. Please try again later.

#### Conditional responses example

9. When you are done, click the X to close the node. Your changes are automatically saved.

## Updating output.generic with a webhook

You can use a webhook to update `output.generic` and provide dynamic responses. The blog article [How to Dynamically Add Response Options to Dialog Nodes in Watson Assistant](#) shows you how.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Making a call before processing a message](#).

## Making a call before processing a message

Make a call to an external service or application every time a customer submits input. The external service can process the message before it is processed by your assistant.

A webhook is a mechanism that allows you to call out to an external program based on events in your program. Add a premessage webhook to your assistant if you want the webhook to be triggered before each incoming message is processed by your assistant.

 **Important:** The premessage webhook works with the v2 `/message` API only (stateless and stateful). For more information, see the [API reference](#).

You can use a premessage webhook to do the following types of things:

- Translate the customer's input to the language that is used by your skill.
- Check for and remove any personally identifiable information, such as an email address or social security number that a customer might submit.

You can use this webhook in coordination with the postmessage webhook. For example, the postmessage webhook can do things like translate the response back into the customer's native language or add back information that was removed for privacy reasons, if necessary. For more information, see [Making a call after processing a message](#).

If you want to perform a one-time action when certain conditions are met during a conversation, use a dialog webhook instead. For more information about the dialog webhook, see [Making a programmatic call from dialog](#).

 **Note:** For environments where private endpoints are in use, keep in mind that a webhook sends traffic over the internet.

## Defining the webhook

You can define one webhook URL to use for preprocessing every incoming message.

The programmatic call to the external service must meet these requirements:

- The call must be a POST HTTP request.
- The request body must be a JSON object (`Content-Type: application/json`).
- The call must return in 30 seconds or less.

 **Tip:** If your external service supports only GET requests, or if you need to specify URL parameters dynamically at run time, consider creating an intermediate service that accepts a POST request with a JSON payload containing any runtime values. The intermediate service can then make a request to the target service, passing these values as URL parameters, and then return the response to the dialog.

 **Important:** Do not set up and test your webhook in a production environment where the assistant is deployed and is interacting with customers.

To add the webhook details, complete the following steps:

1. From your assistant, click the  icon, and then choose **Settings**.
2. Click **Webhooks > Pre-message webhook**.

- Set the *Pre-message webhook* switch to **Enabled**.
- Decide whether to return an error if the webhook call fails.



**Important:** When enabled, everything stops until the preprocessing step is completed successfully.

- If you have a critical preprocessing step that must be taken before you want to allow the message to be processed by the assistant, enable this setting.

Take steps to test the process that you are calling on a regular basis so you will know if it's down, and can change this setting to prevent all of your message calls from failing.

- When this setting is disabled, the assistant ignores any errors it encounters and continues to process the incoming message without taking the preprocessing step. If the preprocessing step is helpful but not critical, consider keeping this setting disabled.

- In the **URL** field, add the URL for the external application to which you want to send HTTP POST request callouts.

For example, you might write a Cloud Functions web action that checks whether a message is in a language other than English, and if so, send it to the Language Translator service to convert it to English. Specify the URL for your web action.

`https://us-south.functions.cloud.ibm.com/api/v1/web/my_org_dev/default/translateToEnglish.json`

You must specify a URL that uses the SSL protocol, so specify a URL that begins with `https`.



**Important:** You cannot use a webhook to call a Cloud Functions action that uses token-based Identity and Access Management (IAM) authentication. However, you can make a call to a Cloud Functions web action or a secured web action.

- In the **Secret** field, add a private key to pass with the request that can be used to authenticate with the external service.

The key must be specified as a text string, such as `purple unicorn`. Maximum length is 1,024 characters. You cannot specify a context variable.

It is the responsibility of the external service to check for and verify the secret. If the external service does not require a token, specify any string you want. You cannot leave this field empty.



**Note:** If you want to see the secret as you enter it, click on the **Show password** icon before you start typing. After you save the secret, the string is replaced by asterisks and can't be viewed again.

For more information about how this field is used, see [Webhook security](#).

- In the **Timeout** field, specify the length of time (in seconds) you want the assistant to wait for a response from the webhook before returning an error. The timeout duration cannot be shorter than 1 second or longer than 30 seconds.
- In the Headers section, add any headers that you want to pass to the service one at a time by clicking **Add header**.

For example, if the external application that you call returns a response, it might be able to send a response in multiple different formats. The webhook requires that the response is formatted in JSON. The following table illustrates how to add a header that indicates that you want the resulting value to be returned in JSON format.

Header name	Header value
<code>Content-Type</code>	<code>application/json</code>
Header example	

The service automatically sends an `Authorization` header with a JWT; you do not need to add one. If you want to handle authorization yourself, add your own authorization header and it will be used instead.



**Note:** After you save the header value, the string is replaced by asterisks and can't be viewed again.

Your webhook details are saved automatically.

## Testing the webhook



**Important:** Do extensive testing of your webhook before you enable it for an assistant that is being used in a production environment.

Your assistant must have a skill added to it before your webhook can do anything useful. The webhook is triggered when a message is sent to your assistant to be processed by the associated skill.

If you enable the setting that returns an error when the webhook call fails, the processing of the assistant is halted entirely if the webhook encounters any issues. Take steps to test the process that you are calling on a regular basis so you will be alerted if the external service is down, and can take actions to prevent all of the incoming messages from failing to be received.

**Tip:** If you call an Cloud Functions web action, you can use the logging capability in Cloud Functions to help you troubleshoot your code. You can [download the command line interface](#), and then enable logging with the [activation polling command](#).

## Troubleshooting the webhook

The following error codes can help you track down the cause of issues you might encounter. If you have a web chat integration, for example, you will know that your webhook has an issue if every test message you submit returns a message such as **There is an error with the message you just sent, but feel free to ask me something else.** If this message is displayed, use a REST API tool, such as cURL, to send a test **/message** API request, so you can see the error code and full message that is returned.

Error code and message	Description
422 Webhook responded with invalid JSON body	The webhook's HTTP response body could not be parsed as JSON.
422 Error validating webhook response	The webhook's HTTP response body was not a valid <b>/message</b> body.
422 Webhook responded with <b>[500]</b> status code	There's a problem with the external service you called. The code failed or the external server refused the request.
500 Processor Exception : <b>[connections to all backends failing]</b>	An error occurred in the webhook microservice. It could not connect to backend services.

### Error code details

## Webhook security

To authenticate the webhook request, verify the JSON Web Token (JWT) that is sent with the request. The webhook microservice automatically generates a JWT and sends it in the **Authorization** header with each webhook call. It is your responsibility to add code to the external service that verifies the JWT.

For example, if you specify **purple unicorn** in the **Secret** field, you might add code similar to this:

```
const jwt = require('jsonwebtoken');
...
const token = request.headers.authentication; // grab the "Authentication" header
try {
 const decoded = jwt.verify(token, 'purple unicorn');
} catch(err) {
 // error thrown if token is invalid
}
```

## Request body

It is useful to know the format of the request body of the premessage webhook so that your external code can process it.

The payload contains the request body of the **/message** (stateful and stateless) v2 API request. The event name **message\_received** indicates that the request is generated by the premessage webhook. For more information about the message request body, see the [API reference](#).

```
{
 "payload" : { Copy of request body sent to /message }
 "event": {
 "name": "message_received"
 }
}
```

## Response body

The service that receives the POST request from the webhook must return a JSON object (**Accept: application/json**).

The response body must have the following structure:

```
{
 "payload": {
 ...
 }
}
```

```
}
```

The **payload** object in the response should contain the **payload** object that was received in the request body. Your code can modify property values in the message payload it received (for example, to update property values, or to add or remove context variables); but the message payload returned to the service must conform to the schema for a request to the **message** method. For more information, see the [API reference](#).

## Example 1

This example shows you how to check the language of the input text, and append the language info to the input text string.

In the premessage webhook configuration page, the following values are specified:

- **URL:** `https://us-south.functions.appdomain.cloud/api/v1/web/e97d2516-5ce4-4fd9-9d05-acc3dd8ennn/default/check_language`
- **Secret:** none
- **Header name:** Content-Type
- **Header value:** application/json

The premessage webhook calls an IBM Cloud Functions web action name **check\_language**.

The node.js code in the **check\_language** web action looks as follows.

```
let rp = require("request-promise");

function main(params) {
 console.log(JSON.stringify(params))
 if (params.payload.input.text !== "") {
 // Send a request to the Watson Language Translator service to check the language of the input text.
 const options = { method: 'POST',
 url: 'https://api.us-south.language-translator.watson.cloud.ibm.com/instances/572b37be-09f4-4704-b693-3bc63869nnnn/v3/identify?version=2018-05-01',
 auth: {
 'username': 'apikey',
 'password': 'nnn'
 },
 headers: {
 "Content-Type": "text/plain"
 },
 body: [
 params.payload.input.text
],
 json: true,
 };
 return rp(options)
 .then(res => {
 params.payload.context.skills["main skill"].user_defined["language"] = res.languages[0].language;
 console.log(JSON.stringify(params))
 //Append "in" plus "the language code" to the input text, surrounded by parentheses.
 const response = {
 body : {
 payload : {
 input : {
 text : params.payload.input.text + '' + '(in ' + res.languages[0].language + ')'
 },
 },
 };
 return response;
 })
 }
 return {
 body : params
 };
}
```

To test the webhook, click **Preview**. In the preview panel, submit the text **Buenos días**. The assistant probably won't understand the input, and will return the response from your *Anything else* node. However, if you go to the Analytics page of your skill and open the User conversations page, you can see what was submitted. Check the most recent user conversation. The log will show that the user input is **Buenos días (in es)**. The **es** in parentheses represents the language code for Spanish, so the webhook worked and recognized that the submitted text was a Spanish phrase.

The screenshot shows the Watson Assistant interface for a skill named 'My first skill'. The 'User conversations' tab is active, displaying 19 messages. One message from a user is highlighted with a yellow oval, showing the text 'Buenas dias (in es)'. The timestamp for this message is 01/20/2021 @ 4:39 PM.

## Example 2

This example shows you how to check the language of the incoming message, and if it's not English, translate it into English before submitting it to the assistant. Doing so can be useful if the assistant's conversational skill is in English.

Define a sequence of web actions in IBM Cloud Functions. The first action in the sequence checks the language of the incoming text. The second action in the sequence translates the text from its original language into English.

In the premessage webhook configuration page, the following values are specified:

- **URL:** [https://us-south.functions.appdomain.cloud/api/v1/web/e97d2516-5ce4-4fd9-9d05-acc3dd8ennn/default/translation\\_sequence](https://us-south.functions.appdomain.cloud/api/v1/web/e97d2516-5ce4-4fd9-9d05-acc3dd8ennn/default/translation_sequence)
- **Secret:** none
- **Header name:** Content-Type
- **Header value:** application/json

The node.js code for the first web action in your sequence looks as follows:

```
let rp = require("request-promise");

function main(params) {
 console.log(JSON.stringify(params))
 if (params.payload.input.text !== "") {
 const options = { method: 'POST',
 url: 'https://api.us-south.language-translator.watson.cloud.ibm.com/instances/572b37be-09f4-4704-b693-3bc63869nnnn/v3/identify?version=2018-05-01',
 auth: {
 'username': 'apikey',
 'password': 'nnn'
 },
 headers: {
 "Content-Type": "text/plain"
 },
 body: [
 params.payload.input.text
],
 json: true,
 };
 return rp(options)
 .then(res => {
 //Set the language property of the incoming message to the language that was identified by Watson Language Translator.
 params.payload.context.skills["main skill"].user_defined["language"] = res.languages[0].language;
 console.log(JSON.stringify(params))
 return params;
 })
 }
 else {
 params.payload.context.skills["main skill"].user_defined["language"] = 'none'
 return params
 }
};
```

The second web action in the sequence sends the text to the Watson Language Translator service to translate the input text from the language that was

identified in the previous web action into English. The translated string is then sent to your assistant instead of the original text.

The node.js code for the second action in your sequence looks as follows:

```
let rp = require("request-promise");

function main(params) {
 console.log(JSON.stringify(params))
 //If the incoming message is not null and is not English, translate it.
 if ((params.payload.context.skills["main skill"].user_defined.language !== 'en') && (params.payload.context.skills["main skill"].user_defined.language !== 'none')) {
 const options = { method: 'POST',
 url: 'https://api.us-south.language-translator.watson.cloud.ibm.com/instances/572b37be-09f4-4704-b693-3bc63869nnnn/v3/translate?',
 version=2018-05-01',
 auth: {
 'username': 'apikey',
 'password': 'nnn'
 },
 headers: {
 "Content-Type": "application/json"
 },
 //The body includes the parameters that are required by the Language Translator service, the text to translate and the target language to
 //translate it into.
 body: {
 text: [
 params.payload.input.text
],
 target: 'en'
 },
 json: true
 };
 return rp(options)
 .then(res => {
 params.payload.context.skills["main skill"].user_defined["original_input"] = params.payload.input.text;
 const response = {
 body : {
 payload : {
 "context" : params.payload.context,
 "input" : {
 "text" : res.translations[0].translation,
 "options" : {
 "export" : true
 }
 }
 },
 },
 };
 return response
 })
 }
 return {
 body : params
 };
}
```

When you test the webhook in the preview panel, you can submit **Buenos días**, and the assistant responds as if you said **Good morning** in English. In fact, when you check the *Analytics>User conversations* page, the log shows that the user input was **Good morning**.

You can add a postmessage webhook to translate the message's response back into the customer's native language before it is displayed. For more information see [Example 2](#).

## Removing the webhook

If you decide you do not want to preprocess customer input, complete the following steps:

1. From the assistant overview page, click the **:** icon, and then choose **Settings**.
2. Click **Webhooks > Pre-message webhook**.
3. Do one of the following things:
  - o To change the webhook that you want to call, click **Delete webhook** to delete the currently specified URL and secret. You can then add a new URL and other details.
  - o To stop calling a webhook to process every incoming message, click the **Pre-message webhook** switch to disable the webhook altogether.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Making a call after processing a message](#).

## Making a call after processing a message

Make a call to an external service or application every time a response is rendered by the assistant.

A webhook is a mechanism that allows you to call out to an external program based on events in your program. You can add a postmessage webhook to your assistant if you want the webhook to be triggered before each message response is returned to the customer.



**Important:** The postmessage webhook works with the v2 `/message` API only (stateless and stateful). For more information, see the [API reference](#).

You can use a postmessage webhook to do things like extract custom responses from an external content repository. For example, you can define a conversational skill with custom IDs in the response instead of text. The postmessage webhook can pass the ID to an external database to retrieve a stored text response.

You can use this webhook in coordination with the premessage webhook. For example, if you used the premessage webhook to strip personally identifiable information from the customer's input, you can use the postmessage webhook to add it back. Or if you use the premessage webhook to translate the customer's input to the language of the skill, you can use the postmessage webhook to translate the response back into the customer's native language before it is returned. For more information, see [Making a call before processing a message](#).

If you want to perform a one-time action when certain conditions are met during a conversation, use a dialog webhook instead. For more information about the dialog webhook, see [Making a programmatic call from dialog](#).



**Note:** For environments where private endpoints are in use, keep in mind that a webhook sends traffic over the internet.

## Defining the webhook

You can define one webhook URL to use for processing every message response before it is shared with the customer.

The programmatic call to the external service must meet these requirements:

- The call must be a POST HTTP request.
- The call must be completed in 30 seconds or less.
- The format of the request and response must be in JSON. For example: `Content-Type: application/json`.



**Important:** Do not set up and test your webhook in a production environment where the assistant is deployed and is interacting with customers.

To add the webhook details, complete the following steps:

1. From your assistant, click the `:` icon, and then choose **Settings**.
2. Click **Webhooks > Post-message webhook**.
3. Set the *Post-message webhook* switch to **Enabled**.
4. Decide whether to return an error if the webhook call fails.



**Important:** When enabled, everything stops until the processing step is completed successfully.

- If you have a critical postprocessing step that must be taken before you want to allow the response to be sent to the customer, then enable this setting.
- When this setting is disabled, the assistant ignores any errors it encounters and continues to process the response without taking the processing step. If the postprocessing step is helpful but not critical, consider keeping this setting disabled.

5. In the **URL** field, add the URL for the external application to which you want to send HTTP POST request callouts.

For example, maybe you store your assistant's responses in a separate content management system. When the skill understands the input, the processed dialog node returns a unique ID that corresponds to a response in your CMS. To call a service that retrieves a response from your CMS for a given unique ID, specify the URL for your service instance.

`https://my-webhook.com/get_answer`

You must specify a URL that uses the SSL protocol, so specify a URL that begins with `https`.



**Important:** You cannot use a webhook to call a Cloud Functions action that uses token-based Identity and Access Management (IAM) authentication. However, you can make a call to a Cloud Functions web action or a secured web action.

6. In the **Secret** field, add a private key to pass with the request that can be used to authenticate with the external service.

The key must be specified as a text string, such as `purple unicorn`. Maximum length is 1,024 characters. You cannot specify a context variable.

It is the responsibility of the external service to check for and verify the secret. If the external service does not require a token, specify any string you want. You cannot leave this field empty.



**Note:** If you want to see the secret as you enter it, click on the **Show password** icon before you start typing. After you save the secret, the string is replaced by asterisks and can't be viewed again.

7. In the **Timeout** field, specify the length of time (in seconds) you want the assistant to wait for a response from the webhook before returning an error. The timeout duration cannot be shorter than 1 second or longer than 30 seconds.
8. In the Headers section, add any headers that you want to pass to the service one at a time by clicking **Add header**.

For example, if the external application that you call returns a response, it might be able to send a response in multiple different formats. The webhook requires that the response is formatted in JSON. The following table illustrates how to add a header that indicates that you want the resulting value to be returned in JSON format.

Header name	Header value
Content-Type	application/json
Header example	

The service automatically sends an `Authorization` header with a JWT; you do not need to add one. If you want to handle authorization yourself, add your own authorization header and it will be used instead.



**Note:** After you save the header value, the string is replaced by asterisks and can't be viewed again.

Your webhook details are saved automatically.

## Testing the webhook



**Important:** Do extensive testing of your webhook before you enable it for an assistant that is being used in a production environment.

Your assistant must have a skill added to it before your webhook can do anything useful. The webhook is triggered only when a message has been processed by your skill, and a response is ready to be returned.

If you enable the setting that returns an error when the webhook call fails, the processing of the assistant is halted entirely if the webhook encounters any issues. Take steps to test the process that you are calling regularly so you are alerted if the external service is down, and can take actions to prevent all of the message responses from failing to be returned.



**Tip:** If you call an Cloud Functions web action, you can use the logging capability in Cloud Functions to help you troubleshoot your code. You can [download the command line interface](#), and then enable logging with the [activation polling command](#).

## Troubleshooting the webhook

The following error codes can help you track down the cause of issues you might encounter. If you have a web chat integration, for example, you will know that your webhook has an issue if every test message you submit returns a message such as, `There is an error with the message you just sent, but feel free to ask me something else.` If this message is displayed, use a REST API tool, such as cURL, to send a test `/message` API request, so you can see the error code and full message that is returned.

Error code and message	Description
422 Webhook responded with invalid JSON body	The webhook's HTTP response body could not be parsed as JSON.
422 Webhook responded with [500] status code	There's a problem with the external service you called. The code failed or the external server refused the request.
500 Processor Exception : <code>[connections to all backends failing]</code>	An error occurred in the webhook microservice. It could not connect to backend services.
Error code details	

## Webhook security

To authenticate the webhook request, verify the JSON Web Token (JWT) that is sent with the request. The webhook microservice automatically generates a JWT and sends it in the **Authorization** header with each webhook call. It is your responsibility to add code to the external service that verifies the JWT.

For example, if you specify **purple unicorn** in the **Secret** field, you might add code similar to this:

```
const jwt = require('jsonwebtoken');
...
const token = request.headers.authentication; // grab the "Authentication" header
try {
 const decoded = jwt.verify(token, 'purple unicorn');
} catch(err) {
 // error thrown if token is invalid
}
```

## Example request body

It is useful to know the format of the request postmessage webhook body so that your external code can process it.

The payload contains the response body that is returned by your assistant for the v2 **/message** (stateful and stateless) API call. The event name **message\_processed** indicates that the request is generated by the postmessage webhook. For more information about the message request body, see the [API reference](#).

The following sample shows how a simple request body is formatted.

```
{
 "event": {
 "name": "message_processed"
 },
 "options": {},
 "payload": {
 "output": {
 "intents": [
 {
 "intent": "General_Greetings",
 "confidence": 1
 }
],
 "entities": [],
 "generic": [
 {
 "response_type": "text",
 "text": "Hello. Good evening"
 }
]
 },
 "user_id": "test user",
 "context": {
 "global": {
 "system": {
 "user_id": "test user",
 "turn_count": 11
 },
 "session_id": "sxxx"
 },
 "skills": {
 "main skill": {
 "user_defined": {
 "var": "anthony"
 },
 "system": {
 "state": "nnn"
 }
 }
 }
 }
 }
}
```

## Example 1

This example shows you how to add **y'all** to the end of each response from the assistant.

In the postmessage webhook configuration page, the following values are specified:

- **URL:** `https://us-south.functions.appdomain.cloud/api/v1/web/e97d2516-5ce4-4fd9-9d05-acc3dd8ennn/southernize/add_southern_charm`

- **Secret:** none
- **Header name:** Content-Type
- **Header value:** application/json

The postmessage webhook calls an IBM Cloud Functions web action name `add_southern_charm`.

The node.js code in the `add_southern_charm` web action looks as follows:

```
function main(params) {
 console.log(JSON.stringify(params))
 if (params.payload.output.generic[0].text !== "") {
 //Get the length of the input text
 var length = params.payload.output.generic[0].text.length;
 //create a substring that removes the last character from the input string, which is typically punctuation.
 var revision = params.payload.output.generic[0].text.substring(0,length-1);
 const response = {
 body : {
 payload : {
 output : {
 generic : [
 {
 //Replace the input text with your shortened revision and append y'all to it.
 "response_type": "text",
 "text": revision + ',' + 'y'all.'
 }
],
 },
 },
 };
 return response;
 }
 else {
 return {
 body : params
 }
 }
}
```

## Example 2

This example shows you how to translate a message response back to the customer's native language. It only works if you perform the steps in [Example 2](#) to define a premessage webhook that translates the original message into English.

Define a sequence of web actions in IBM Cloud Functions. The first action in the sequence checks for the language of the original incoming text, which you stored in a context variable named `original_input` in the premessage webhook code. The second action in the sequence translates the dialog response text from English into the original language that was used by the customer.

In the premessage webhook configuration page, the following values are specified:

- **URL:** `https://us-south.functions.appdomain.cloud/api/v1/web/e97d2516-5ce4-4fd9-9d05-acc3dd8ennn/default/response-translation_sequence`
- **Secret:** none
- **Header name:** Content-Type
- **Header value:** application/json

The node.js code for the first web action in your sequence looks as follows:

```
let rp = require("request-promise");

function main(params) {
 console.log(JSON.stringify(params))

 if (params.payload.output.generic[0].text !== "") {
 const options = { method: 'POST',
 url: 'https://api.us-south.language-translator.watson.cloud.ibm.com/instances/572b37be-09f4-4704-b693-3bc63869nnnn/v3/identify?version=2018-05-01',
 auth: {
 'username': 'apikey',
 'password': 'nnnn'
 },
 headers: {
 "Content-Type": "text/plain"
 },
 body: [
 params.payload.context.skills['main skill'].user_defined.original_input
],
 }
```

```

 json: true,
);
 return rp(options)
 .then(res => {
 //Set the language property of the incoming message to the language that was identified by Watson Language Translator.
 params.payload.context.skills['main skill'].user_defined['language'] = res.languages[0].language;
 console.log(JSON.stringify(params))
 return params;
 })
}
else {
 params.payload.context.skills['main skill'].user_defined['language'] = 'none';
 return params
}
};

```

The second web action in the sequence looks as follows:

```

let rp = require("request-promise");

function main(params) {
 console.log(JSON.stringify(params))
 if ((params.payload.context.skills["main skill"].user_defined.language !== 'en') && (params.payload.context.skills["main skill"].user_defined.language !== 'none')) {
 const options = { method: 'POST',
 url: 'https://api.us-south.language-translator.watson.cloud.ibm.com/instances/572b37be-09f4-4704-b693-3bc63869nnnn/v3/translate?version=2018-05-01',
 auth: {
 'username': 'apikey',
 'password': 'nnn'
 },
 body: {
 text: [
 params.payload.output.generic[0].text
],
 target: params.payload.context.skills["main skill"].user_defined.language
 },
 json: true
 };
 return rp(options)
 .then(res => {
 params.payload.context.skills["main skill"].user_defined["original_output"] = params.payload.output.generic[0].text;
 params.payload.output.generic[0].text = res.translations[0].translation;
 return {
 body : params
 }
 })
 }
 return {
 body : params
 }
};

```

## Removing the webhook

If you decide you do not want to process message responses, complete the following steps:

1. From the assistant overview page, click the  icon, and then choose **Settings**.
2. Click **Webhooks > Post-message webhook**.
3. Do one of the following things:
  - o To change the webhook that you want to call, click **Delete webhook** to delete the currently specified URL and secret. You can then add a new URL and other details.
  - o To stop calling a webhook to process every message response, click the *Post-message webhook* switch to disable the webhook altogether.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Logging activity with a webhook](#).

## Logging activity with a webhook

PlusEnterprise

You can log activity by making a call to an external service or application every time a customer submits input to the assistant.

A webhook is a mechanism that allows you to call out to an external program based on events in your program.

This feature is available only to Plus and Enterprise plan users.

Add a log webhook to your assistant if you want to use an external service to log Watson Assistant activity. You can log two kinds of activity:

- **Messages and responses:** The log webhook is triggered each time the assistant responds to user input. You can use this option as an alternative to the built-in analytics feature to handle logging yourself. (For more information about the built-in analytics support, see [Metrics overview](#).)



**Note:** The log webhook is not supported for API clients that use the v1 `/message` method.

- **Call detail records (CDRs):** The log webhook is triggered after each telephone call a user makes to your assistant using the phone integration. A Call Detail Record (CDR) is a summary report that documents the details of a telephone call, including phone numbers, call length, latency, and other diagnostic information. CDR records are only available for assistants that use the phone integration.

The log webhook does not return anything to your assistant.



**Note:** For environments where private endpoints are in use, keep in mind that a webhook sends traffic over the internet. For more information, see [Private network endpoints](#).

## Defining the webhook

You can define one webhook URL to use for logging every incoming message or CDR event.

The programmatic call to the external service must meet these requirements:

- The call must be a POST HTTP request.

To add the webhook details, complete the following steps:

1. From your assistant, click the icon, and then choose **Settings**.
2. Click **Webhooks > Log webhook**.
3. Set the *Log webhook* switch to **Enabled**.

If you cannot enable the webhook, you might need to upgrade your service plan.

4. In the **URL** field, add the URL for the external application to which you want to send HTTP POST request callouts.

For example, to write the message to the

`https://www.mycompany.com/my_log_service`

You must specify a URL that uses the SSL protocol, so specify a URL that begins with `https`.

5. In the **Secret** field, add a token to pass with the request that can be used to authenticate with the external service.

The secret must be specified as a text string, such as `purple unicorn`. Maximum length is 1,024 characters. You cannot specify a context variable.

It is the responsibility of the external service to check for and verify the secret. If the external service does not require a secret, you can leave this field empty.

It is the responsibility of the external service to check for and verify the secret. If the external service does not require a token, specify any string you want. You cannot leave this field empty.



**Note:** If you want to see the secret as you enter it, click on the **Show password** icon before you start typing. After you save the secret, the string is replaced by asterisks and can't be viewed again.

6. Click the appropriate checkboxes to select which kinds of activity you want to log:

- To log messages and responses, select **Subscribe to conversation logs**.
- To log CDR events for the phone integration, select **Subscribe to CDR (Call Detail Records)**.

7. In the Headers section, add any headers that you want to pass to the service one at a time by clicking **Add header**.

The service automatically sends an **Authorization** header with a JWT; you do not need to add one. If you want to handle authorization yourself, add your own authorization header and it will be used instead.



**Note:** After you save the header value, the string is replaced by asterisks and can't be viewed again.

Your webhook details are saved automatically.

## Removing the webhook

If you decide you do not want to log messages programmatically, complete the following steps:

1. From the assistant overview page, click the  icon, and then choose **Settings**.
2. Click **Webhooks > Log webhook**.
3. Do one of the following things:
  - o To change the webhook that you want to call, click **Delete webhook** to delete the currently specified URL and secret. You can then add a new URL and other details.
  - o To stop calling a webhook to log every message and response, click the *Log webhook* switch to disable the webhook altogether.

## Webhook security

To authenticate the webhook request, verify the JSON Web Token (JWT) that is sent with the request. The webhook microservice automatically generates a JWT and sends it in the **Authorization** header with each webhook call. It is your responsibility to add code to the external service that verifies the JWT.

For example, if you specify **purple unicorn** in the **Secret** field, you might add code similar to this:

```
const jwt = require('jsonwebtoken');
...
const token = request.headers.authentication; // grab the "Authentication" header
try {
 const decoded = jwt.verify(token, 'purple unicorn');
} catch(err) {
 // error thrown if token is invalid
}
```

## Webhook request body

The request body the webhook sends to the external service is a JSON object with the following structure:

```
{
 "event": {
 "name": "{event_type}"
 },
 "payload": {
 ...
 }
}
```

where **{event\_type}** is either **message\_logged** (for messages and responses) or **cdr\_logged** (for CDR events).

The **payload** object contains the event data to be logged. The structure of the **payload** object depends on the type of event.

### Message event payload

For **message\_logged** events, the **payload** object contains data about a message request sent to the assistant and the message response returned to the integration or client application. For more information about the fields that are part of message requests and responses, see the [API reference](#).

 **Important:** The log webhook payload might include data that is not currently supported by the API (for example, data returned from actions skills). Any fields that are not defined in the API reference documentation are subject to change.

### CDR event payload

For **cdr\_logged** events, the **payload** object contains data about a Call Detail Record (CDR) event that was handled by the phone integration. The structure of the **payload** object for a CDR event is as shown by this example:

```
{
 "primary_phone_number": "+18005550123",
 "global_session_id": "9caa8bad-aaa8-4a5a-a4b5-62bcc703d15",
 "failure_occurred": false,
 "transfer_occurred": false,
 "active_calls": 0,
 "warnings_and_errors": [
 {
 "code": "CWSMR0033W",
 "message": "CWSMR0033W: The inbound RTP audio stream jitter of 43 ms exceeds the maximum jitter threshold of 30 ms."
 },
 {
 }
```

```

 "code": "CWSMR0070W",
 "message": "CWSMR0070W: A request to the Watson Speech To Text service failed for the following reason = Unexpected server response: 403, response headers = {"strict-transport-security": "max-age=31536000; includeSubDomains; content-length": "157", "content-type": "application/json", "x-dp-watson-tran-id": "23860083-88b6-41d7-9130-30bbfebe647e", "x-request-id": "23860083-88b6-41d7-9130-30bbfebe647e", "x-global-transaction-id": "6c764df3-81db-41bb-a14f-62384facffca", "server": "watson-gateway", "x-edgeconnect-midmile-rtt": "1", "x-edgeconnect-origin-mex-latency": "28", "date": "Thu, 13 May 2021 20:31:12 GMT", "connection": "keep-alive"}, response body = {"code": 403, "trace": "23860083-88b6-41d7-9130-30bbfebe647e", "error": "Forbidden", "more_info": "https://cloud.ibm.com/docs/watson?topic=watson-forbidden-error"}, x-global-transaction-id = 6c764df3-81db-41bb-a14f-62384facffca. The Media Relay will reattempt to send the request."
 },
],
"realtime_transport_network_summary": {
 "inbound_stream": {
 "average_jitter": 4,
 "canonical_name": "b74f3689-1ae8-4a0a-bde3-adf5b488553e",
 "maximum_jitter": 18,
 "packets_lost": 0,
 "packets_transmitted": 952,
 "tool_name": ""
 },
 "outbound_stream": {
 "average_jitter": 0,
 "canonical_name": "voice.gateway",
 "maximum_jitter": 0,
 "packets_lost": 0,
 "packets_transmitted": 838,
 "tool_name": "IBM Voice Gateway/1.0.7.0"
 }
},
"call": {
 "start_timestamp": "2021-10-12T20:54:02.591Z",
 "stop_timestamp": "2021-10-12T20:54:20.375Z",
 "milliseconds_elapsed": 17784,
 "outbound": false,
 "end_reason": "assistant_hangup",
 "security": {
 "media_encrypted": false,
 "signaling_encrypted": false,
 "sip_authenticated": false
 }
},
"session_initiation_protocol": {
 "invite_arrival_time": "2021-10-12T20:54:00.565Z",
 "setup_milliseconds": 2026,
 "headers": {
 "call_id": "17465345_115257202@10.90.150.99",
 "from_uri": "sip:+18885550456@pstn.twilio.com",
 "to_uri": "sip:+18005550123@public.voip.us-south.assistant.test.watson.cloud.ibm.com"
 }
},
"max_response_milliseconds": {
 "assistant": 339,
 "text_to_speech": 535,
 "speech_to_text": 0
},
"assistant_interaction_summaries": [
{
 "session_id": "7874ec3a-1330-4180-afe1-46bfb220af5b",
 "assistant_id": "97f16ba4-ad94-41af-aa6c-33cd56ad5e7e",
 "turns": [
 {
 "assistant": {
 "log_id": "58bebfd1-0118-419b-a555-b152a1efbbe8",
 "response_milliseconds": 339,
 "start_timestamp": "2021-10-12T20:54:00.722Z"
 },
 "request": {
 "type": "start"
 },
 "response": [
 {
 "barge_in_occurred": true,
 "streaming_statistics": {
 "response_milliseconds": 301,
 "start_timestamp": "2021-10-12T20:54:00.722Z",
 "stop_timestamp": "2021-10-12T20:54:01.023Z",
 "transaction_id": "3dce431c-fb2f-4b62-9fce-585f4e06fe00"
 },
 ...
 }
]
 }
]
}
]
}

```

```

 "type": "text_to_speech"
 }
]
},
{
 "assistant": {
 "log_id": "38f36bfb-c2aa-4600-9418-6ab422664e31",
 "response_milliseconds": 158,
 "start_timestamp": "2021-10-12T20:54:05.621Z"
 },
 "request": {
 "type": "dtmf"
 },
 "response": [
 {
 "type": "disable_speech_barge_in"
 },
 {
 "type": "text_to_speech",
 "barge_in_occurred": false,
 "streaming_statistics": {
 "transaction_id": "af4c47c3-5cc4-43c8-9b9c-81d6f997c52f",
 "start_timestamp": "2021-10-12T20:54:06.321Z",
 "stop_timestamp": "2021-10-12T20:54:14.338Z",
 "response_milliseconds": 535
 }
 },
 {
 "type": "enable_speech_barge_in"
 },
 {
 "type": "text_to_speech",
 "barge_in_occurred": true,
 "streaming_statistics": {
 "transaction_id": "eafdd846-2829-4e1a-8068-b1035510b1e1",
 "start_timestamp": "2021-10-12T20:54:14.795Z",
 "stop_timestamp": "2021-10-12T20:54:20.388Z",
 "response_milliseconds": 447
 }
 }
]
},
{
 "assistant": {
 "log_id": "07d74b35-0205-43e4-923c-1e43e1cb429c",
 "response_milliseconds": 0,
 "start_timestamp": "2021-10-12T20:54:20.377Z"
 },
 "request": {
 "type": "hangup"
 },
 "response": []
}
]
}
]
}
}

```

The **payload** JSON object for a call detail record event contains the following keys:

Key	Type	Description
<code>primary_phone_number</code>	string	The phone number that was called.
<code>global_session_id</code>	string	The unique session identifier.
<code>failure_occurred</code>	Boolean	Indicates whether a failure occurred during the call.
<code>failure_details</code>	string	Details about a failure.
<code>transfer_occurred</code>	Boolean	Indicates whether an attempt was made to transfer a call

<code>active_calls</code>	Number	Number of active calls when the call started.
<code>x-global-sip-trunk-call-id</code>	string	The value of the SIP trunk call ID header extracted from the initial SIP <code>INVITE</code> request. The following SIP trunk call ID headers are supported: <code>X-Twilio-CallSid</code> <code>X-SID</code> <code>X-Global-SIP-Trunk-Call-ID</code>
<code>call</code>	JSON object	Contains information about the call.
<code>session_initiation_protocol</code>	JSON object	SIP protocol related details.
<code>max_response_milliseconds</code>	JSON object	Maximum latency for various services used during the call.
<code>assistant_interaction_summaries</code>	JSON array	Details about the Watson Assistant transactions that took place during the call.
<code>injected_custom_data</code>	JSON object	A JSON object that contains a set of key/value pairs. Extracted from the <code>cdr_custom_data</code> context variable.
<code>warnings_and_errors</code>	JSON array	An array of warnings and errors that were logged during the call.
<code>realtime_transport_network_summary</code>	JSON object	When RTCP is enabled, the <code>realtime_transport_network_summary</code> object provides statistics for the inbound stream in the <code>inbound_stream</code> object and statistics for the outbound stream in the <code>outbound_stream</code> object.

#### Keys for the payload object

The `call` object contains the following keys:

Key	Type	Description
<code>start_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the call started.
<code>stop_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the call ended.
<code>milliseconds_elapsed</code>	number	Length of the call in milliseconds.
<code>end_reason</code>	string	Call end reason: <code>assistant_transfer</code> <code>assistant_hangup</code> <code>caller_hangup</code> <code>failed</code>
<code>security.media_encrypted</code>	Boolean	Indicates whether the media was encrypted.
<code>security.signalng_encrypted</code>	Boolean	Indicates whether the SIP signaling was encrypted.
<code>security.sip_authenticated</code>	Boolean	Indicates whether SIP authentication was used to challenge the authentication of the caller.

#### Keys for the call object

The `session_initiation_protocol` object contains the following keys:

Key	Type	Description
-----	------	-------------

<code>invite_arrival_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the <b>INVITE</b> request arrived.
<code>setup_milliseconds</code>	number	Time it took to set up the call in milliseconds. Specifically, the field shows the time between when the initial SIP <b>INVITE</b> request was received and when the final SIP <b>ACK</b> request was received.
<code>headers.call_id</code>	string	The SIP <b>Call-ID</b> header field pulled from the SIP <b>INVITE</b> related to the call.
<code>headers.from_uri</code>	string	SIP URI from the initial SIP INVITE <b>From</b> field
<code>headers.to_uri</code>	string	SIP URI from the initial SIP INVITE <b>To</b> field

#### Keys for the session\_initiation\_protocol object

The `assistant_interaction_summaries` object contains the following keys:

Key	Type	Description
<code>assistant_id</code>	string	The unique identifier of the assistant.
<code>session_id</code>	string	The unique identifier of the session.
<code>turns</code>	JSON array	An array of the Watson Assistant transactions that took place during the conversation.

#### Keys for the assistant\_interaction\_summaries object

The `turn` object contains the following keys:

Key	Type	Description
<code>assistant.log_id</code>	string	A unique identifier for the logged transaction. Can be used to correlate between message logs and CDR events.
<code>assistant.start_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the request was sent to Watson Assistant.
<code>assistant.response_milliseconds</code>	number	Time between when the request was sent and when the response was received from Watson Assistant.
<code>request</code>	JSON object	A request sent to Watson Assistant.
<code>response</code>	JSON array	An array of the <code>response</code> objects associated with the request.

#### Keys for the turn object

The `request` object contains the following keys:

Key	Type	Description
-----	------	-------------

<b>type</b>	string	The request type: <i>start</i> - an initial request to Watson Assistant <i>speech_to_text</i> - a request is triggered on speech recognition <i>dtmf</i> - a request is triggered when DTMF collection completes <i>sms</i> - a request is triggered when a SMS message is received from the caller <i>post_response_timeout</i> - a request is triggered when the post response timer expires <i>redirect</i> - a request is triggered when a call is redirected <i>transfer</i> - a request is triggered when a call is transferred <i>transfer_failed</i> - a request is triggered when a call transfer fails <i>final_utterance_timeout</i> - a request is triggered when the final utterance timer expires <i>no_input_turn</i> - a request is triggered when <b>no_inout_turn</b> is enabled <i>sms_failure</i> - a request is triggered when a SMS message can't be sent to the caller <i>speech_to_text_result_filtered</i> - a request is triggered when an utterance is filtered due to low confidence level <i>mrcp_recognition_unsuccessful</i> - a request is triggered when the MRCP recognition completes without a final utterance <i>network_warning</i> - a request is triggered when a network error is detected <i>media_capability_change</i> - a request is triggered when media capabilities change in the middle of a call
<b>streaming_statistics</b>	JSON object	Contains information and statistics related to the Speech to Text recognition.

#### Keys for the request object

The `request.streaming_statistics` object contains the following keys:

Key	Type	Description
<code>transaction_id</code>	string	A unique identifier of the transaction.
<code>start_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the transaction started.
<code>stop_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the transaction ended.
<code>response_milliseconds</code>	number	Latency in milliseconds between when silence is detected in the caller's speech and a final result from Speech to Text is received.
<code>echo_detected</code>	Boolean	Indicates whether an echo was detected. The value can be <code>true</code> or <code>false</code> .
<code>confidence</code>	number	The confidence score of the final utterance.

#### Keys for the `request.streaming_statistics` object

The `response` object contains the following keys:

Key	Type	Description
-----	------	-------------

<b>type</b>	string	The response type: <i>text_to_speech</i> - a command to play an utterance to the caller <i>sms</i> - a command to send a SMS to the caller <i>url</i> - a command to play an audio file to the caller <i>transfer</i> - a command to transfer a call <i>text_to_speech_config</i> - a command to change Text to Speech settings <i>speech_to_text_config</i> - a command to change the Speech to Text settings <i>pause_speech_to_text</i> - a command to stop speech recognition <i>unpause_speech_to_text</i> - a command to start speech recognition <i>pause_dtmf</i> - a command to stop DTMF recognition <i>unpause_dtmf</i> - a command to start speech recognition <i>enable_speech_barge_in</i> - a command to enable speech barge-in so that callers can interrupt playback by speaking <i>disable_speech_barge_in</i> - a command to disable speech barge-in so that playback isn't interrupted when callers speak over top of the played back audio <i>enable_dtmf_barge_in</i> - a command to enables DTMF barge-in so that callers can interrupt playback from the phone integration by pressing a key. <i>disable_dtmf_barge_in</i> - a command to disable DTMF barge-in so that playback from the phone integration isn't interrupted when callers press keys <i>dtmf</i> - a command to send DTMFs to the caller <i>hangup</i> - a command to disconnect a call
-------------	--------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<b>barge_in_occurred</b>	Boolean	Indicates whether barge-in occurred during the turn.
<b>streaming_statistics</b>	JSON object	Contains information and statistics related to the Text to Speech synthesis and playback.

#### Keys for the response object

The `response.streaming_statistics` object contains the following keys:

Key	Type	Description
<code>transaction_id</code>	string	A unique identifier of the transaction.
<code>start_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the transaction started.
<code>stop_timestamp</code>	string. Time in the ISO format <code>yyyy-MM-ddTHH:mm:ss.SSSZ</code>	Time when the transaction ended.
<code>response_milliseconds</code>	number	Time in milliseconds between when a text utterance is sent to the Text to Speech service and when the phone integration receives the first packet of synthesized audio.

#### Keys for the `response.streaming_statistics` object

The `max_response_milliseconds` object contains the following keys:

Key	Type	Description
<code>assistant</code>	number	Maximum round-trip latency in milliseconds, calculated from all Watson Assistant requests related to the call.
<code>text_to_speech</code>	number	Maximum time in milliseconds between when a text utterance is sent to the Text to Speech service and when the phone integration receives the first packet of synthesized audio. Calculated from all the Text to Speech requests related to this call.
<code>speech_to_text</code>	number	Maximum latency in milliseconds between when silence is detected in the user's speech and a final result from Speech to Text is received. This value is calculated from all the Speech to Text recognition results related to this call.

#### Keys for the `max_response_milliseconds` object

## Mapping between CDR and Watson Assistant response types

CDR response type	Watson Assistant response type
text_to_speech	text
url	audio
dtmf	dtmf, command_info.type: send
sms	user_defined, vgwAction.command: vgwActSendSMS
transfer	connect_to_agent
text_to_speech_config	text_to_speech, command_info.type: configure
speech_to_text_config	speech_to_text, command_info.type: configure
unpause_speech_to_text	start_activities, type:speech_to_text_recognition
pause_speech_to_text	stop_activities, type:speech_to_text_recognition
unpause_dtmf	start_activities, type:dtmf_collection
pause_dtmf	stop_activities, type:dtmf_collection
enable_speech_barge_in	text_to_speech, command_info.type: enable_barge_in
disable_speech_barge_in	text_to_speech, command_info.type: disable_barge_in
enable_dtmf_barge_in	dtmf, command_info.type: enable_barge_in
disable_dtmf_barge_in	dtmf, command_info.type: disable_barge_in
hangup	end_session

Mapping between CDR and Watson Assistant response types

## Warning details

The `warnings_and_errors` object contains warnings and errors that were logged during the call, listed in order of occurrence. Warnings for the following conditions are included:

- Messages when utterances are filtered out by the confidence score threshold.
- Text to Speech underflows, which is when Text to Speech synthesis can't keep up with the phone integration streaming rate and audio might skip.
- RTP network warnings, such as high packet loss or high average jitter, if RTCP is enabled.

```
$
"warnings_and_errors": [
 {
 "message": "CWSMR0032W: A Watson Speech to Text final utterance has a confidence score of 0.1, which does not meet the confidence score threshold of 0.2. The utterance will be ignored.",
 "id": "CWSMR0032W"
 },
 {
 "message": "CWSMR0031W: The synthesis stream from the Watson Text To Speech service can't keep up with the playback rate to the caller, so audio might skip. transaction ID=a1b2c3d4e5",
 "id": "CWSMR0031W"
 }
]
```

The object for each warning contains the following keys:

Key	Type	Description

<code>message</code>	string	The text of the warning message that was logged.
<code>id</code>	string	The unique message identifier.

Keys for the `warnings_and_errors` object

## RTP network summary details

When RTCP is enabled, each `realtime_transport_network_summary` object provides statistics for the inbound stream in the `inbound_stream` object and statistics for the outbound stream in the `outbound_stream` object.

```
$ "realtime_transport_network_summary": {
 "inbound_stream": {
 "maximum_jitter": 5,
 "average_jitter": 1,
 "packets_lost": 0,
 "packets_transmitted": 1000,
 "canonical_name": "user@example.com",
 "tool_name": "User SIP Phone"
 },
 "outbound_stream": {
 "maximum_jitter": 5,
 "average_jitter": 1,
 "packets_lost": 0,
 "packets_transmitted": 2000,
 "canonical_name": "voice.gateway@127.0.0.1",
 "tool_name": "IBM Voice Gateway/1.0.0.5"
 }
}
```

The objects for each stream contain the following keys:

Key	Type	Description
<code>maximum_jitter</code>	Number	Maximum jitter during the call
<code>average_jitter</code>	Number	Average jitter, calculated over the call duration
<code>packets_lost</code>	Number	An estimate of the number of packets that were lost during the call
<code>packets_transmitted</code>	Number	An estimate of the total number of packets that were transmitted during the call
<code>canonical_name</code>	string	A unique identifier for the sender of the stream, typically in <code>@</code> format
<code>tool_name</code>	string	The name of the application or tool where the stream originated. For Voice Gateway, the default is <code>IBM Voice Gateway/</code> .

Keys for RTP network summary details

## Injecting custom values into call detail record events

By including the `cdr_custom_data` context variable in your Watson Assistant dialog or actions, you can add custom data to your Call Detail Records. Each time this object is received it will merge with any previously received `cdr_custom_data`. You can record data related to various activities that are happening during a call. This includes ways to identify completion of specific tasks.

```
"context": {
 "integrations": {
 "voice_telephony": {
 "cdr_custom_data": {
 "key1": "value1",
 "key2": "value2"
 }
 }
 }
}
```

When generating a CDR report, the custom data is included in the `injected_custom_data` field. For example, in the following CDR report, the `key1` and `key2` are added to the `injected_custom_data` field.

```
{
 "payload": {
 ...
 "injected_custom_data": {
 "key1": "value1",
 "key2": "value2"
 }
 ...
 }
}
```

Merging CDR data

During a call, if the `cdr_custom_data` context variable is received multiple times, the data is merged in the `injected_custom_data` field.

In the first call, values for `key1` and `key2` are defined.

```
"context": {
 "integrations": {
 "voice_telephony": {
 "cdr_custom_data": {
 "key1": "value1",
 "key2": "value2"
 }
 }
 }
}
```

In the second call, values for `key1` and `key3` are defined.

```
"context": {
 "integrations": {
 "voice_telephony": {
 "cdr_custom_data": {
 "key1": "value11",
 "key3": "valu3"
 }
 }
 }
}
```

After these calls, the following code example shows how the custom CDR data is merged in the `injected_custom_data` field.

```
{
 "payload": {
 ...
 "injected_custom_data": {
 "key1": "value11",
 "key2": "value2",
 "key3": "value3"
 }
 ...
 }
}
```

Removing custom CDR data

To remove a key from the CDR custom data, set the `cdr_custom_data` context variable with that key set to `""`.

The following example shows that to remove the `key1` field from the custom CDR data, you would overwrite the `key1` value with a blank, `""`.

```
"context": {
 "integrations": {
 "voice_telephony": {
 "cdr_custom_data": {
 "key1": ""
 }
 }
 }
}
```

## Dialog skill development

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Advanced skill development

A developer can extend the capabilities of a skill.

Learn more about what's possible in the following sections.

### Dialog skill development

- [Adding custom dialog flow for integrations](#)
- [Defining responses using the JSON editor](#)
- [Handling phone interactions](#)
- [Handling SMS with Twilio interactions](#)
- [Making a programmatic call from dialog](#)
- [Requesting client actions](#)

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Anatomy of a message

A single `/message` API call is equivalent to a single turn in a conversation, which consists of a message that is submitted by a customer and a corresponding response from your assistant.

Each reply that a customer makes in response to a prompt from the assistant is passed as an independent `/message` API call.

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

- `context`: Contains variables that are meant to be persisted. For the dialog to subsequently reference information that is submitted by the user, you must store the information in the context object. For example, the dialog can collect the user's name and then refer to the user by name in subsequent nodes. The following example shows how the context object is represented in the dialog JSON editor:

```
{
 "context": {
 "user_name": "<? @name.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. The following example shows how the `input` object is represented in the dialog JSON editor:

```
{
 "input": {
 "text": "Where's your nearest store?"
 }
}
```

- `output`: The dialog response to return to the user. The following example shows how the output object is represented in the dialog JSON editor:

```
{
 "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.

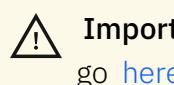
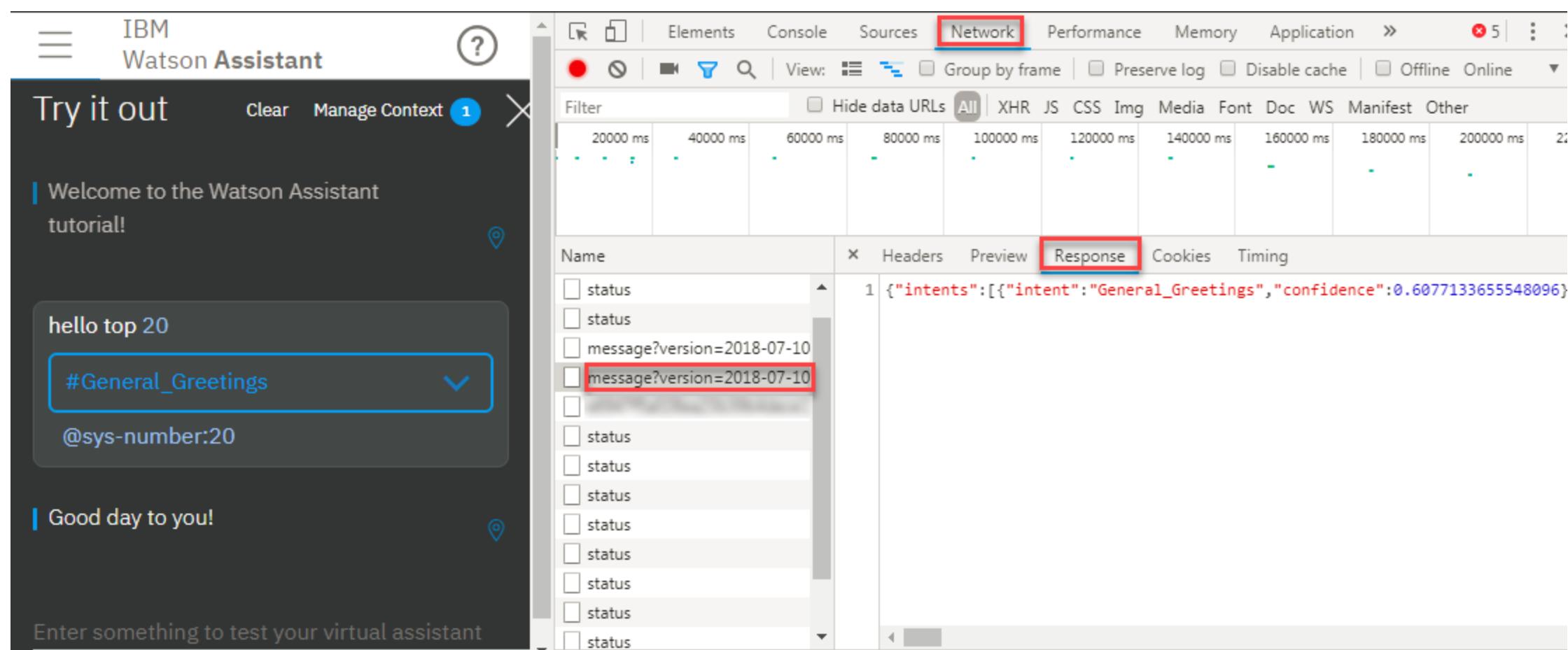
For more information about the `/message` API call, see the [API reference](#).

For information about how to refer to these message objects within a conversation, see [Expressions for accessing objects](#).

## Viewing API call details

As you test your conversation, 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. Open 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. It also lists the values of context variables at the time of the call. To view the response body in structured format, click the Preview column.



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Adding custom dialog flows for integrations

Use the JSON editor in dialog to access information that is submitted from the web chat integration.

Starting with API version `2020-04-01`, the `context` object that is passed as part of the v2 `/message` API request contains an `integrations` object. This object makes it possible to pass information that is specific to a single integration type in the context. For more information about context variables, see [Context variables](#).



**Important:** The `integrations` object is available from the v2 API in version `2020-04-01` or later only.

To take advantage of the `context.integrations` object, you can create context variables that are named as follows to get and set values for different integrations:

Integration type	Context variable syntax
Phone	<code>\$integrations.voice_telephony</code>

Salesforce service desk from web chat	<code>\$integrations.salesforce</code>
SMS with Twilio	<code>\$integrations.text.messaging</code>
Web chat (and assistant preview)	<code>\$integrations.chat</code>
Zendesk service desk from web chat	<code>\$integrations.zendesk</code>

#### Integration-specific context variables

The following sections describe how to use integration-specific context variables to do common tasks.

## Building integration-specific dialog flow

Create a single dialog that is optimized to use the best features offered by each channel or client interface in which it is deployed.

You can customize the conversation in the following ways:

- To add an entire dialog branch that is only processed by a specific integration type, add the appropriate integration type context variable, such as `$integrations.chat`, to the *If assistant recognizes* field of the dialog root node.
- To add a single dialog node that is only processed by a specific integration type, add the appropriate integration type context variable, such as `$integrations.zendesk`, to the *If assistant recognizes* field of the dialog child node.
- To add slightly different responses for a single dialog node based on the integration type, complete the following steps:
  - From the node's edit view, click **Customize** and then set the *Multiple conditioned responses* switch to **On**. Click **Apply**.
  - In the dialog node response section, add the appropriate condition and corresponding response for each custom response type.

The following examples show how to specify a hypertext link in the best format for the integration where the text response will be displayed. For the *Web chat* integration, which supports Markdown formatting, you can include a link label in the response text to make the response look nicer. For the *SMS with Twilio* integration, you can skip the formatting that makes sense in a web page, and add the straight URL.

Integration type	Condition	Sample text response
SMS with Twilio	<code>'\$integrations.text.messaging'</code>	<code>'For more information, go to https://www.ibm.com.'</code>
Web chat	<code>'\$integrations.chat'</code>	<code>'For more information, go to [the ibm.com site](https://www.ibm.com).'</code>
Response to show if no other conditions are met.	<code>'true'</code>	<code>'For more information, go to ibm.com.'</code>

#### Custom conditioned responses

The rich response types often behave differently when they are displayed in different built-in integrations or in the assistant preview. For more information about these unique behaviors, see the following topics:

- [Web chat](#)
- [Phone](#)
- [SMS with Twilio](#)
- [Assistant preview](#)



**Note:** If you need to provide customized responses for different channels, and you do not need to modify your dialog flow based on which integration is in use, you can also use the `channels` array to target your responses to specific integrations. For more information, see [Targeting specific integrations](#).

## Web chat: Accessing sensitive data

If you enable security for the web chat, you can configure your web chat implementation to send encrypted data to the dialog. Payload data that is sent from web chat is stored in a private context variable named `context.integrations.chat.private.user_payload`. No private variables are sent from the dialog to any integrations. For more information about how to pass data, see [Passing sensitive data](#).

To access the payload data, you can reference the `context.integrations.chat.private.user_payload` object from the dialog node condition.



**Note:** You must know the structure of the JSON object that is sent in the payload.

For example, if you passed the value `"mvp:true"` in the JSON payload, you can add a dialog flow that checks for this value to define a response that is meant for VIP customers only. Add a dialog node with a condition like this:

Field	Value
If assistant recognizes	<code>\$integrations.chat.private.user_payload.mvp</code>
Assistant responds	<code>I can help you reserve box seats at the upcoming conference!</code>

Private variable as node condition

## Web chat: Accessing web browser information

When you use the web chat integration, information about the web browser that your customer is using to access the web chat is automatically collected and stored. The information is stored in the `context.integrations.chat.browser_info` object.

You can design your dialog to take advantage of details about the web browser in use. The following properties are taken from the `window` object that represents the window in which the web chat is running:

- `browser_name`: The browser name, such as `chrome`, `edge`, or `firefox`.
- `browser_version`: The browser version, such as `80.0.0`.
- `browser_OS`: The operating system of the customer's computer, such as `Mac OS`.
- `language`: The default locale code of the browser, such as `en-US`.
- `page_url`: Full URL of the web page in which the web chat is embedded. For example: `https://www.example.com/products`
- `screen_resolution`: Specifies the height and width of the browser window in which the web page is displayed. For example: `width: 1440, height: 900`
- `user_agent`: Content from the User-Agent request header. For example: `Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101 Firefox/80.0`
- `client_ip_address`: IP address of the customer's computer. For example: `183.49.92.42`

## Example: Using page URL information in your dialog

You might embed the web chat in multiple pages on your website. Let's say you want to route chat transfers from the web chat to different Salesforce agents based on the page from which the customer asks to speak to someone. If the customer is on the insurance plans page of your website (`https://www.example.com/insurance.html`), you want to route them to agents who are experts in your company's insurance plans. If the customer is on the investments web page (`https://www.example.com/invest.html`), you want to route them to agents who are experts in your company's investment opportunities. You can add multiple conditioned responses to the dialog node where the transfer takes place and add logic like this:

### Conditioned response 1

- Condition: `$integrations.chat.browser_info.page_url.contains('insurance.html')`
- Response type: *Connect to human agent*
- Button ID: `Z23453e25vv` (The button that routes to the insurance experts agent queue)

### Conditioned response 2

- Condition: `$integrations.chat.browser_info.page_url.contains('invest.html')`
- Response type: *Connect to human agent*
- Button ID: `Z23453j24ty` (The button that routes to the investment experts agent queue)

For more information about implementing chat transfers, see [Adding a Connect to human agent response type](#). For more information about the `contains()` SpEL expression method, see [Expression language methods](#).

## Web chat: Reusing the JWT for webhook authentication

You can use the same JSON Web Token (JWT) that was used to secure your web chat to authenticate webhook calls. If you specify a token in the `identityToken` property when you add the web chat to your web page, the token is stored in a private variable named `context.integrations.chat.private.jwt`. For more information about passing a JWT, see [Enable security](#).

1. From your dialog skill, open the **Options>Webhooks** page.
2. Click **Add authentication**.
3. Without adding any values to the fields in the *Basic authorization* modal that is displayed, click **Save**.

An Authorization header is added for you.
4. In the **Header value** field, add `$integrations.chat.private.jwt`.

**!** **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Handling SMS with Twilio interactions

Learn about common actions you can use to manage the flow of conversations that your assistant has with customers by using SMS text messaging.

Before you add customizations to your dialog that support SMS messaging interactions, you must set up the SMS with Twilio integration. For more information, see [Integrating with SMS with Twilio](#).

You can perform the following types of actions:

- [Sending multimedia content over text](#)
- [Customizing lists in a text message](#)
- [Sending a text message during a phone conversation](#)

For command reference documentation, see [SMS with Twilio integration commands reference](#).

## Adding SMS-based actions to your dialog

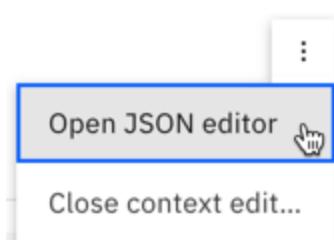
When calling messaging-specific actions from a dialog, follow these guidelines:

- Define the action within the `context` object, not the `output` object of the dialog node JSON snippet.
- Define only one action or one sequence per conversation turn.
- Do not jump from a dialog node with an action configured for it to another dialog node with an action configured for it.

To enable text messaging-specific actions, you must add a JSON code block to the dialog node where you want the action to trigger.

To add a JSON code block to a dialog node, complete the following steps:

1. From your dialog skill, go to the *Dialog* page.
2. Open the dialog node where you want to call the action.
3. From the *Assistant responds* section, click the menu , and then choose **Open JSON editor**.



4. Add the action command JSON code block to the `context` object. (If no `context` object exists, add one. The `context` object is a peer to the `output` object.)

For example:

```
{
 "output": {
 "generic": [
]
 },
 "context": {
 "smsAction": {
 "command": "<command-name>",
 "parameters": {
 "<first-parameter>": "<parameter-value>"
 }
 }
 }
}
```

## Sending multimedia content over text

To allow multimedia content, such as an image, to be sent in a text message, use the `smsActSendMedia` command.

```
{
 "output": {
 "generic": [
]
 },
 "context": {
 "smsAction": {
 "command": "smsActSendMedia",
 "parameters": {
 "media": "image:
 <image URL>"
 }
 }
 }
}
```

```

" smsAction": {
 "command": "smsActSendMedia",
 "parameters": {
 "mediaURL": [
 "https://upload.wikimedia.org/wikipedia/commons/3/39/We_love_kids_but.jpg"
]
 }
}
}

```

You can specify the following parameter value for the `vsmsActSendMedia` command:

- `mediaURL`: A JSON array of one or more publicly-accessible media URLs for images or videos.

## Customizing lists

You can customize how these lists are displayed and handled.

- [Options list](#)
- [Disambiguation](#)

### Options list

The dialog supports an `option` response type, which shows the customer multiple options to choose from. You can customize how the options that are defined for an option response type are shown to customers and the ways in which a customer can select an option by adding the `vgwActSetOptionsConfig` action command.

The following example shows how to customize the option response type.

```

{
 "output": {
 "generic": [
 {
 "title": "Which of these items do you want to insure?",
 "options": [
 {
 "label": "Boat",
 "value": {
 "input": {
 "text": "I want to buy boat insurance."
 }
 },
 "label": "Car",
 "value": {
 "input": {
 "text": "I want to buy car insurance."
 }
 },
 "label": "House",
 "value": {
 "input": {
 "text": "I want to buy house insurance."
 }
 }
 }
],
 "description": "Insurance types.",
 "response_type": "option"
 }
],
 "context": {
 "smsAction": {
 "command": "smsActSetOptionsConfig",
 "parameters": {
 "prefixText": "%s."
 }
 }
 }
 }
}

```

First the value specified in the `title` attribute is displayed to the user. Then, the text specified in each `label` attribute. For example, `Which of these items do you want to insure? 1.Boat 2.Car 3.House`

To configure what the assistant shows before each option, edit the `prefixText` parameter. Use `%s` to represent the number corresponding to the

option; it is replaced with the actual number at run time.

```
"smsAction": {
 "command": "smsActSetOptionsConfig",
 "parameters": {
 "prefixText": "Enter %s for "
 }
}
```

For example, `Which of these items do you want to insure? Enter 1 for Boat Enter 2 for Car Enter 3 for House`

## Disambiguation list

When the dialog is confident that more than one dialog node might be the right one to process in response to a customer query, disambiguation is triggered. Disambiguation asks the customer to clarify which path they want to follow to get an answer. For more information, see [Disambiguation](#).

You can customize how the disambiguation list choices are displayed and how a customer can select a disambiguation choice by adding the `smsActSetDisambiguationConfig` action command.

You might want to define the customization in the welcome node or another node that is triggered early in the conversation so it is applied any time disambiguation is triggered.

```
{
 "output": {
 "generic": [
]
 },
 "context": {
 "smsAction": {
 "command": "smsActSetDisambiguationConfig",
 "parameters": {
 "prefixText": "%s."
 }
 }
 }
}
```

When displayed, the assistant shows the introductory text that is configured for disambiguation, such as `Did you mean?`. Then it lists the choices from the disambiguation list as numbered choices.

The `prefixText` parameter prepends a number to the text specified in a `label`. The list choices are numbered sequentially and are displayed to the user in the order in which they appear in the list. The user can type a number to pick one of the choices.

For example, if label is configured as follows:

```
"label": "I'd like to order a drink."
```

The assistant sends this message to the user:

1. I'd like to order a drink.

You can configure the text that will be prepended to each label. In the `prefixText` attribute, `%s` represents the number corresponding to the suggestion; it is replaced with the actual number at run time.

```
"context": {
 "smsAction": {
 "command": "smsActSetDisambiguationConfig",
 "parameters": {
 "prefixText": "Enter %s for:"
 }
 }
}
```

When `prefixText` is set to `Enter %s for:`, the following output is sent to the customer:

```
Enter 1 for: I'd like to order a drink.
```



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Requesting client actions

Beta

Add a client action to your dialog node to request that a client-side process run and return a result to the dialog.

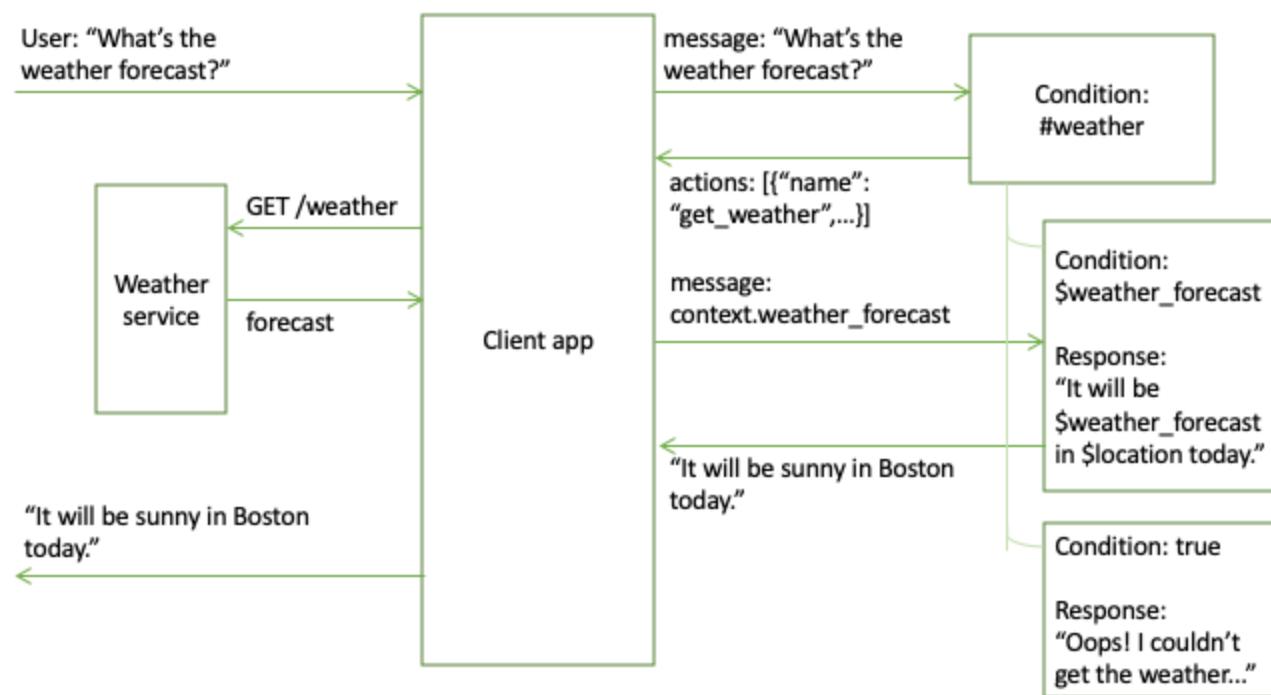
If your assistant is integrated with a custom client using the API, you can use client actions to invoke functions that the client application completes. A client action defines a request for a client-side process, in a standardized format that your external client application can understand. Your external client application must use the provided information to carry out the requested action, which can be a local programmatic function, a call to an external service, or any other action the client can perform. The client then returns the result from the action to the dialog, which can process it further, display it to the user, or use it as a condition to determine subsequent flow.

Unlike other action types, a client action does not make a direct call itself. Instead, it simply makes a request for the client application to do something, in the form of an action object that is included in a response from the dialog (along with any output text). This object includes a name identifying the requested action, along with any required parameters and the name of a context variable where the result should be stored. It is the responsibility of the client application to carry out the requested action, store the result in the context, and send it back to the dialog with the next message.

You might call a client application to do the following types of things:

- Validate information that you collected from the user.
- Do calculations or string manipulations on user input that are too complex for supported SpEL expression methods to handle.
- Get data from another application or service.

The following diagram illustrates how client actions work, using the example of an action to get a weather forecast.



The flow of requests and responses follows this pattern:

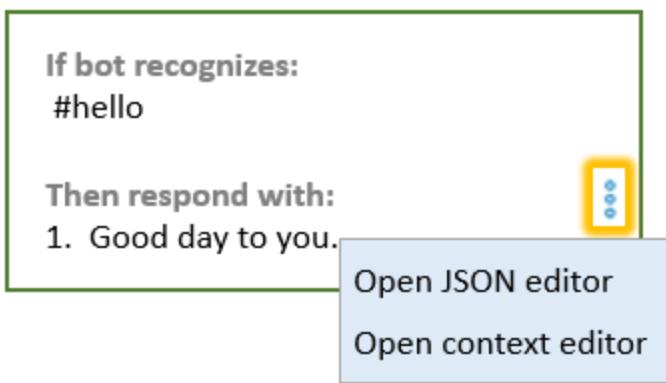
1. The client application sends a message containing user input that asks for a weather forecast (using the `message` or `message_stateless` method).
2. The user input triggers a dialog node conditioned on a `#weather` intent. In its response to the client, this node specifies the `get_weather` client action, which is a name that the client application recognizes. (This is in addition to any text response, such as `Checking the weather forecast...`.)
3. When it receives this response, the client application recognizes that the `get_weather` action is being requested. It calls an external web service (`/weather`) to get the actual forecast information, passing any specified parameters (such as the user's location). The client application then stores the returned information in context variable specified in the action request (`$weather_forecast`).
4. The client application sends another message to the service, including the updated context that contains the weather forecast information. From the dialog's perspective, this message is effectively the next round of user input, although the actual input text is blank.
5. A child node of the `#weather` node is triggered by the presence of the `$weather_forecast` context variable. This node responds with text output that includes the weather forecast, which the client application displays to the user. (If the `$weather_forecast` variable is not set, another child node can handle this case and report an error.)

**Tip:** It is also possible to call an external Web service directly from a dialog node, without involving the client application, by defining a webhook. For more information about how to call an external service using a webhook, see [Making a programmatic call from a dialog node](#).

## Procedure

To request a client action from a dialog node, complete the following steps:

1. In the dialog node from which you want to request the client action, open the JSON editor for the node response.



2. Use the following syntax to define the client action you want to request.

```
{
 "context": {
 "variable_name" : "variable_value"
 },
 "actions": [
 {
 "name":<actionName>,
 "type": "client",
 "parameters": {
 "<parameter_name>": "<parameter_value>",
 "<parameter_name>": "<parameter_value>"
 },
 "result_variable": "<result_variable_name>"
 }
],
 "output": {
 "text": "response text"
 }
}
```

The `actions` array specifies the actions you want the client application to carry out. It can define up to five separate actions. Specify the following name and value pairs in the JSON array:

- `<actionName>` : Required. The name of the action you want the client application to perform. This name can be in any format (for example, `calculateRate` or `get_balance`), but it must be a name that your client application will recognize and know how to handle. The name cannot be longer than 256 characters.
- `<type>` : Indicates the type of call to make. For a client action, this property is optional (`client` is the default value).
- `<action_parameters>` : Any parameters that are required for the client action, which are specified as a JSON object. If the action does not require any parameters, omit this property.
- `<result_variable_name>` : The name of the context variable you want to use to store the JSON object that is returned by the client action. The client application is expected to use the specified variable to send the return value in the context of the next `/message` input, and subsequent dialog nodes can then access it. You can specify the `result_variable_name` by using the following syntax:

- `my_result`
- `$my_result`

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

You can optionally specify a `context.` location keyword prefix for this variable (for example, `context.my_result`). However, this is not necessary, because all context variables are stored in this location by default.

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

- `context.weather.today`
- `context.weather.tomorrow`

The results (in this example, `temp` and `rain` properties) are stored in the context in this structure:

```
{
 "weather": {
 "today": {
 "temp": "20",
 "rain": "30"
 },
 "tomorrow": {
 "temp": "23",
 "rain": "80"
 }
}
```

```
 }
}
}
```

If multiple actions in a single JSON action array add the result of their programmatic call to the same context variable, then the order in which the context is updated matters. Per action type, the order in which the actions are defined in the array determines the order in which the context variable's value is set. The context variable value returned by the last action in the array overwrites the values calculated by any other actions.

The `result_variable` property is required. If the client action does not return any result, specify `null` as the value.

## Client action example

The following example shows what a request for a call to an external weather service might look like. It is added to the JSON editor that is associated with the node response. By the time the node-level response is triggered, slots have collected and stored the date and location information from the user. This example assumes that the client action is named `weather_forecast`, that it takes a `location` parameter, and that the results are to be stored in the `weather_forecast` context variable.

```
{
 "actions": [
 {
 "name": "get_weather",
 "type": "client",
 "parameters": {
 "location": "$location"
 },
 "result_variable": "weather_forecast"
 }
]
}
```

The client application must check for the presence of any client actions in the responses to messages it sends to the assistant. When it recognizes a request for the `get_weather` action, it executes the action (calling the external weather service), and it stores the result in the specified context variable (`weather_forecast`). It then sends a message to the service, including the updated context.

To handle this message in your dialog, create a child node following the node that requested the action. You can condition this child node on the special condition `true` to ensure that it is always triggered by the message that the client sends after completing the requested action. In this child node, add the response to show the user, reading the stored action result from the `$my_forecast` context variable:

```
{
 "output": {
 "text": {
 "values": [
 "It will be $weather_forecast in $location.literal today."
]
 }
 }
}
```



**Tip:** For more information about how to implement the client side of this exchange, see [Implementing app actions](#).



**Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## 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 integration or custom client.

### Generic JSON format

The generic JSON format for responses is used to specify responses that are intended for any integration or custom client. This format can accommodate various response types that are supported by multiple 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.)



**Note:** You can also specify responses using the native format used by the target channel. For more information, see [Native JSON format](#).

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": "https://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 an integration, the response is converted at run time into the format expected by the channel. If the response contains multiple media types or attachments, the generic response is converted into a series of separate message payloads as needed. These are sent to the channel in separate messages.



**Note:** When a response is split into multiple messages, the integration 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.

If you are building your own client application, your app must implement each response type as appropriate. For more information, see [Implementing responses](#).

## Targeting specific integrations

If you plan to use integrations to deploy your assistant to multiple channels, you might want to send different responses to different integrations. The **channels** property of the generic response object provides a way to do this.

This mechanism is useful if your dialog flow does not change based on the integration in use, and if you cannot know in advance what integration the response will be sent to at run time. By using **channels**, you can define a single dialog node that supports all integrations, while still customizing the output for each channel. For example, you might want to customize the text formatting, or even send different response types, based on what the channel

supports.

Using `channels` is particularly useful in conjunction with the `channel_transfer` response type. Because the message output is processed both by the channel initiating the `transfer` and by the target channel, you can use `channels` to define responses that will only be displayed by one or the other. (For more information, and an example, see [Channel transfer](#).)

To specify the integrations for which a response is intended, include the optional `channels` array as part of the response object. All response types support the `channels` array. This array contains one or more objects using the following syntax:

```
$ {
 "channel": "<channel_name>"
}
```

The value of `<channel_name>` can be any of the following strings:

- `chat`: Web chat
- `voice_telephony`: Phone
- `text.messaging`: SMS with Twilio
- `slack`: Slack
- `facebook`: Facebook Messenger
- `intercom`: Intercom
- `whatsapp`: WhatsApp

The following example shows dialog node output that contains two responses: one intended for the web chat integration and one intended for the Slack and Facebook integrations.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "text",
 "channels": [
 {
 "channel": "chat"
 }
],
 "text" : "This output is intended for the web chat."
 },
 {
 "response_type": "text",
 "channels": [
 {
 "channel": "slack"
 },
 {
 "channel": "facebook"
 }
],
 "text" : "This output is intended for either Slack or Facebook."
 }
]
 }
}
```

If the `channels` array is present, it must contain at least one channel object. Any integration that is not listed ignores the response. If the `channels` array is absent, all integrations handle the response.

**Note:** If you need to change the logic of your dialog flow based on which integration is in use, or based on context data that is specific to a particular integration, see [Adding custom dialog flows for integrations](#).

## 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?
------	------	-------------	-----------

response_type	string	<b>image</b>	Y
source	string	The <b>https:</b> URL of the image. The specified image must be in .jpg, .gif, or .png format.	Y
title	string	The title to show before the image.	N
description	string	The text of the description that accompanies the image.	N
alt_text	string	Descriptive text that can be used for screen readers or other situations where the image cannot be seen.	N

## Example

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

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "image",
 "source": "https://example.com/image.jpg",
 "title": "Example image",
 "description": "An example image returned as part of a multimedia response."
 }
]
 }
}
```

## video

Displays a video specified by a URL.

## Fields

Name	Type	Description	Required?
response_type	string	<b>video</b>	Y
source	string	The <b>https:</b> URL of the video. The URL can specify either a video file or a streaming video on a supported hosting service. For more information, see <a href="#">Adding a Video response type</a> .	Y
title	string	The title to show before the video.	N
description	string	The text of the description that accompanies the video.	N
alt_text	string	Descriptive text that can be used for screen readers or other situations where the video cannot be seen.	N
channel_options.chat.dimensions.base_height	string	The base height (in pixels) to use to scale the video to a specific display size.	N

## Example

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

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "video",
 "source": "https://example.com/videos/example-video.mp4",
 "title": "Example video",
 }
]
 }
}
```

```

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

## audio

Plays an audio clip specified by a URL.

## Fields

Name	Type	Description	Required?
response_type	string	<b>audio</b>	Y
source	string	The <b>https:</b> URL of the audio clip. The URL can specify either an audio file or an audio clip on a supported hosting service. For more information, see <a href="#">Adding an Audio response type</a> .	Y
title	string	The title to show before the audio player.	N
description	string	The text of the description that accompanies the audio player.	N
alt_text	string	Descriptive text that can be used for screen readers or other situations where the audio player cannot be seen.	N
channel_options.voice_telephony.loop	string	Whether the audio clip should repeat indefinitely (phone integration only).	N

## Example

This example plays an audio clip with a title and descriptive text.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "audio",
 "source": "https://example.com/audio/example-file.mp3",
 "title": "Example audio file",
 "description": "An example audio clip returned as part of a multimedia response."
 }
]
 }
}
```

## iframe

Embeds content from an external website as an HTML **iframe** element.

## Fields

Name	Type	Description	Required?
response_type	string	<b>iframe</b>	Y
source	string	The URL of the external content. The URL must specify content that is embeddable in an HTML <b>iframe</b> element. For more information, see <a href="#">Adding an iframe response type</a> .	Y
title	string	The title to show before the embedded content.	N
description	string	The text of the description that accompanies the embedded content.	N

image_url	string	The URL of an image that shows a preview of the embedded content.	N
-----------	--------	-------------------------------------------------------------------	---

## Example

This example embeds an iframe with a title and description.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "iframe",
 "source": "https://example.com/embeddable/example",
 "title": "Example iframe",
 "description": "An example of embeddable content returned as an iframe 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	string	<b>option</b>	Y
title	string	The title to show before the options.	Y
description	string	The text of the description that accompanies the options.	N
preference	string	The preferred type of control to display, if supported by the channel ( <b>dropdown</b> or <b>button</b> ). The Watson Assistant connector currently supports only <b>button</b> .	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 object that includes the message input corresponding to the option, including input text and any other field that is a valid part of a Watson Assistant message. For more information about the structure of message input, see the <a href="#">API Reference</a> .	N

## Example

This example displays two options:

- Option 1 (labeled **Buy something**) sends a simple string message (**Place order**), which is sent to the workspace as the input text.
- Option 2 (labeled **Exit**) sends a complex message that includes both input text and an array of intents.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "option",
 "title": "Choose from the following options:",
 "preference": "button",
 "options": [
 {
 "label": "Buy something",
 "value": {
 "input": {
 "text": "Buy something"
 },
 "intents": [
 "buy"
]
 }
 }
]
 }
]
 }
}
```

```

 "value": {
 "input": {
 "text": "Place order"
 }
 },
 {
 "label": "Exit",
 "value": {
 "input": {
 "text": "Exit"
 },
 "intents": [
 {
 "intent": "exit_app",
 "confidence": 1.0
 }
]
 }
 }
]
}

```

## 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	string	<b>pause</b>	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	string	<b>text</b>	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 ( <code>\n</code> ) 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 <code>sequential</code> , <code>random</code> , and <code>multiline</code> .	N
delimiter	string	The delimiter to output as a separator between responses. Used only when <code>selection_policy=multiline</code> . The default delimiter is newline ( <code>\n</code> ).	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"
 }
]
 }
}
```

## search

Calls the search skill linked to the assistant to retrieve results that are relevant to the user's query.

## Fields

Name	Type	Description	Required?
response_type	string	<code>search_skill</code>	Y
query	string	The text to use for the search query. This string can be empty, in which case the user input is used as the query.	Y
filter	string	An optional filter that narrows the set of documents to be searched.	N
query_type	string	The type of search query to use ( <code>natural_language</code> or <code>discovery_query_language</code> ).	Y
discovery_version	string	The version of the Discovery service API to use. The default is <code>2018-12-03</code> .	N

You can author a `search_skill` response\_type that calls out to your search skill, and then return a `search` response\_type that you can render in a different format. Go to Message > Response > output > MessageOutput > generic > RuntimeResponseTypeSearch in [Watson Assistant v2 API](#) to view how response\_type `search` renders when `search_skill` response is processed..

## Example

This example uses the user input text to send a natural-language query to the search skill.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "search_skill",
 "query": "",
 "query_type": "natural_language"
 }
]
 }
}
```

```

 }
]
}
}
```

### connect\_to\_agent

Requests that the conversation be transferred to a service desk agent for help.

## Fields

Name	Type	Description	Required?
response_type	string	<code>connect_to_agent</code>	Y
message_to_human_agent	string	A message to display to the live agent to whom the conversation is being transferred.	Y
agent_available	string	A message to display to the user when agents are available.	Y
agent_unavailable	string	A message to display to the user when no agents are available.	Y
transfer_info	object	Information used by the web chat service desk integrations for routing the transfer.	N
transfer_info.target.zendesk.department	string	A valid department from your Zendesk account.	N
transfer_info.target.salesforce.button_id	string	A valid button ID from your Salesforce deployment.	N

## Example

This example requests a transfer to a live agent and specifies messages to be displayed both to the user and to the agent at the time of transfer.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "connect_to_agent",
 "message_to_human_agent": "User asked to speak to an agent.",
 "agent_available": {
 "message": "Please wait while I connect you to an agent."
 },
 "agent_unavailable": {
 "message": "I'm sorry, but no agents are online at the moment. Please try again later."
 }
 }
]
 }
}
```

### channel\_transfer

Requests that the conversation be transferred to a different integration.

## Fields

Name	Type	Description	Required?
response_type	string	<code>channel_transfer</code>	Y
message_to_user	string	A message to display to the user before the link for initiating the transfer.	Y
transfer_info	object	Information used by an integration to transfer the conversation to a different channel.	Y

transfer_info.chat	string	The URL for the website hosting the web chat to which the conversation is to be transferred.	Y
--------------------	--------	----------------------------------------------------------------------------------------------	---

## Example

This example requests a transfer from Slack to web chat. In addition to the `channel_transfer` response, the output also includes a `text` response to be displayed by the web chat integration after the transfer. The use of the `channels` array ensures that the `channel_transfer` response is handled only by the Slack integration (before the transfer), and the `connect_to_agent` response only by the web chat integration (after the transfer). For more information about using `channels` to target specific integrations, see [Targeting specific integrations](#).

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "channel_transfer",
 "channels": [
 {
 "channel": "whatsapp"
 }
],
 "message_to_user": "Click the link to connect with an agent using our website.",
 "transfer_info": {
 "target": {
 "chat": {
 "url": "https://example.com/webchat"
 }
 }
 }
 },
 {
 "response_type": "connect_to_agent",
 "channels": [
 {
 "channel": "chat"
 }
],
 "message_to_human_agent": "User asked to speak to an agent.",
 "agent_available": {
 "message": "Please wait while I connect you to an agent."
 },
 "agent_unavailable": {
 "message": "I'm sorry, but no agents are online at the moment. Please try again later."
 },
 "transfer_info": {
 "target": {
 "zendesk": {
 "department": "Payments department"
 }
 }
 }
 }
]
 }
}
```

## dtmf

Sends commands to the phone integration to control input or output using dual-tone multi-frequency (DTMF) signals. (DTMF is a protocol used to transmit the tones that are generated when a user presses keys on a push-button phone.)

## Fields

Name	Type	Description	Required?
response_type	string	<code>dtmf</code>	Y
command_info	object	Information specifying the DTMF command to send to the phone integration.	Y
command_info.type	string	The DTMF command to send ( <code>collect</code> , <code>disable_barge_in</code> , <code>enable_barge_in</code> , or <code>send</code> ).	Y

The `command_info.type` field can specify any of the following supported commands:

- `collect`: Collects DTMF keypad input.
- `disable_barge_in`: Disables DTMF barge-in so that playback from the phone integration is not interrupted when the customer presses a key.
- `enable_barge_in`: Enables DTMF barge-in so that the customer can interrupt playback from the phone integration by pressing a key.
- `send`: Sends DTMF signals.

For detailed information about how to use each of these commands, see [Handling phone interactions](#).

## Example

This example shows the `dtnf` response type with the `collect` command, used to collect DTMF input. For more information, including examples of other DTMF commands, see [Handling phone interactions](#).

```
{
 "output": {
 "generic": [
 {
 "response_type": "dtnf",
 "command_info": {
 "type": "collect",
 "parameters": {
 "termination_key": "#",
 "count": 16,
 "ignore_speech": true
 }
 },
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

## stop\_activities

Sends a command to a channel integration to stop one or more activities that are specific to that channel. The activities remain stopped until they are restarted using the `start_activities` response type.



**Note:** Currently, only the phone integration supports the `stop_activities` response type.

## Fields

Name	Type	Description	Required?
response_type	string	<code>stop_activities</code>	Y
activities	list	A list of objects identifying the activities to stop.	Y
activities[].type	string	The name of the activity to stop.	Y

Currently, the following activities for the phone integration can be stopped:

- `speech_to_text_recognition`: Stops recognizing speech. All streaming audio to the Speech to Text service is stopped.
- `dtnf_collection`: Stops processing of inbound DTMF signals.

## Example

This example uses the `stop_activities` response type to stop recognizing speech. Because this command is specific to the phone integration, the `channels` property specifies `voice_telephony` only.

```
{
 "output": {
 "generic": [
 {
 "response_type": "stop_activities",
 "activities": [
 {
 "type": "speech_to_text_recognition"
 }
],
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

### **start\_activities**

Sends a command to a channel integration to start one or more activities that are specific to that channel. You can use this response type to restart any activity you previously stopped using the **stop\_activities** response type.



**Note:** Currently, only the phone integration supports the **start\_activities** response type.

## Fields

Name	Type	Description	Required?
response_type	string	<b>start_activities</b>	Y
activities	list	A list of objects identifying the activities to start.	Y
activities[].type	string	The name of the activity to start.	Y

Currently, the following activities for the phone integration can be started:

- **speech\_to\_text\_recognition**: Starts recognizing speech. Streaming audio to the Speech to Text service is resumed.
- **dtmf\_collection**: Starts processing of inbound DTMF signals.

## Example

This example uses the **start\_activities** response type to restart recognizing speech. Because this command is specific to the phone integration, the **channels** property specifies **voice\_telephony** only.

```
{
 "output": {
 "generic": [
 {
 "response_type": "start_activities",
 "activities": [
 {
 "type": "speech_to_text_recognition"
 }
],
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

### **text\_to\_speech**

Sends a command to the Text to Speech service instance used by the phone integration. These commands can dynamically change the configuration or behavior of the service during a conversation.

## Fields

Name	Type	Description	Required?
response_type	string	<b>text_to_speech</b>	Y
command_info	object	Information specifying the command to send to the Text to Speech.	Y
command_info.type	string	The command to send ( <code>configure</code> , <code>disable_barge_in</code> , or <code>enable_barge_in</code> ).	Y
command_info.parameters	object	See <a href="#">Applying advanced settings to the Text to Speech service</a> .	N

The `command_info.type` field can specify any of the following supported commands:

- **configure**: Dynamically updates the Text to Speech configuration. Configuration changes can be applied only to the next conversation turn, or for the rest of the session.
- **disable\_barge\_in**: Disables speech barge-in so that playback from the phone integration is not interrupted when the customer speaks.
- **enable\_barge\_in**: Enables speech barge-in so that the customer can interrupt playback from the phone integration by speaking.

For detailed information about how to use each of these commands, see [Applying advanced settings to the Text to Speech service](#).

## Example

This example uses the `text_to_speech` response type with the `configure` command to change the voice used by the Text to Speech service.

```
{
 "output": {
 "generic": [
 {
 "response_type": "text_to_speech",
 "command_info": {
 "type": "configure",
 "parameters" : {
 "synthesize": {
 "voice": "en-US_LisaVoice"
 }
 }
 },
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

## speech\_to\_text

Sends a command to the Speech to Text service instance used by the phone integration. These commands can dynamically change the configuration or behavior of the service during a conversation.

## Fields

Name	Type	Description	Required?
response_type	string	<b>speech_to_text</b>	Y
command_info	object	Information specifying the command to send to the Speech to Text.	Y
command_info.type	string	The command to send (currently only the <code>configure</code> command is supported).	Y

The `command_info.type` field can specify any of the following supported commands:

- **configure**: Dynamically updates the Speech to Text configuration. Configuration changes can be applied only to the next conversation turn, or for the rest of the session.

For detailed information about how to this command, see [Applying advanced settings to the Speech to Text service](#).

## Example

This example uses the `speech_to_text` response type with the `configure` command to change the language model used by the Text to Speech service to Spanish, and to enable smart formatting.

```
{
 "output": {
 "generic": [
 {
 "response_type": "speech_to_text",
 "command_info": {
 "type": "configure",
 "parameters": {
 "narrowband_recognize": {
 "model": "es-ES_NarrowbandModel",
 "smart_formatting": true
 }
 }
 },
 "channels": [
 {
 "channel": "voice_telephony"
 }
]
 }
]
 }
}
```

## end\_session

Sends a command to the channel ending the session. This response type instructs the phone integration to hang up the call.

## Fields

Name	Type	Description	Required?
response_type	string	<code>end_session</code>	Y

For the phone integration, you can use the `channel_options` object to include custom headers with the SIP `BYE` request that is generated. For more information, see [End the call](#).

## Example

This example uses the `end_session` response type to end a conversation.

```
{
 "output": {
 "generic": [
 {
 "response_type": "end_session"
 }
]
 }
}
```

## user\_defined

A custom response type containing any JSON data the client or integration knows how to handle. For example, you might customize the web chat to display a special kind of card, or build a custom application to format responses using a table or chart.

 **Note:** The user-defined response type is not displayed unless you have implemented code specifically to handle it. For more information about customizing the web chat, see [Applying advanced customizations](#). For more information about handling responses in a custom client app, see [Implementing responses](#).

## Fields

Name	Type	Description	Required?
response_type	string	<code>user_defined</code>	Y
user_defined	object	An object containing any data the client or integration knows how to handle. This object can contain any valid JSON data, but it cannot exceed a total size of 5000 bytes.	Y

## Example

This examples shows a generic example of a user-defined response. The `user_defined` object can contain any valid JSON data.

```
$ {
 "output": {
 "generic": [
 {
 "response_type": "user_defined",
 "user_defined": {
 "field_1": "String value",
 "array_1": [
 1,
 2
],
 "object_1": {
 "property_1": "Another string value"
 }
 }
 }
]
 }
}
```

## Native JSON format

In addition to the generic JSON format, the dialog node JSON also supports channel-specific responses written using the native JSON format for a specific channel (such as Slack or Facebook Messenger). You might want to use the native JSON format to specify a response type that is not currently supported by the generic JSON format. By checking the integration-specific context (`context.integrations`), a dialog node can determine the originating integration and then send appropriate native JSON responses.

You can specify native JSON for Slack or Facebook using the `output.integrations` object in the dialog node response. Each child of `output.integrations` represents output intended for a specific integration:

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

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## 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 conversational skill in which it resides. See [Downloading a skill](#) 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 tool, you can define a frame node handler by clicking the **Manage handlers** link from a node with slots. (The tool 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 tool. 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 tool, 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`.

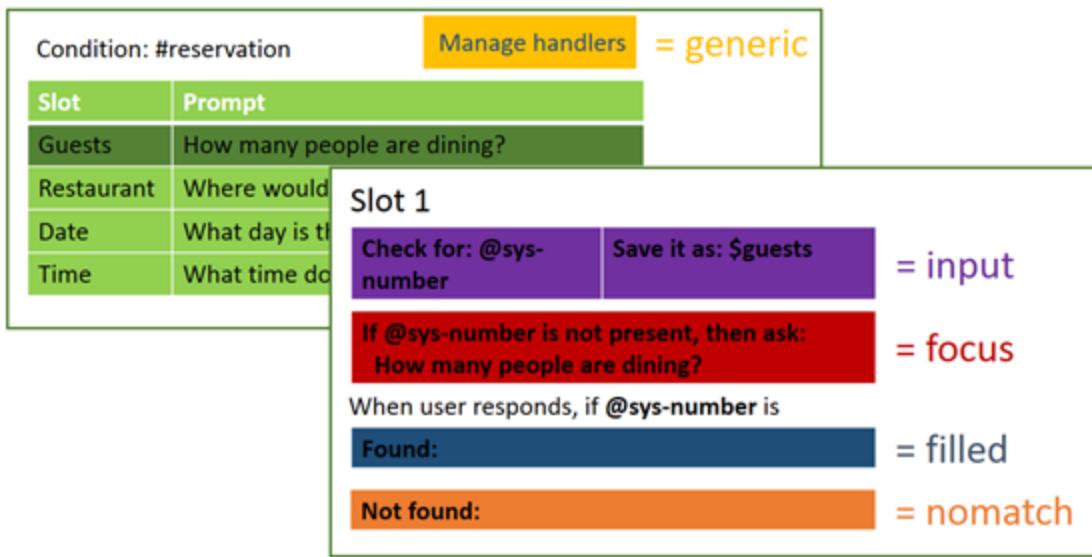
- `slot`: A child node of a node of type `frame`.

This node type is represented in the tool 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 tool user interface you define the code that is triggered for each named event.



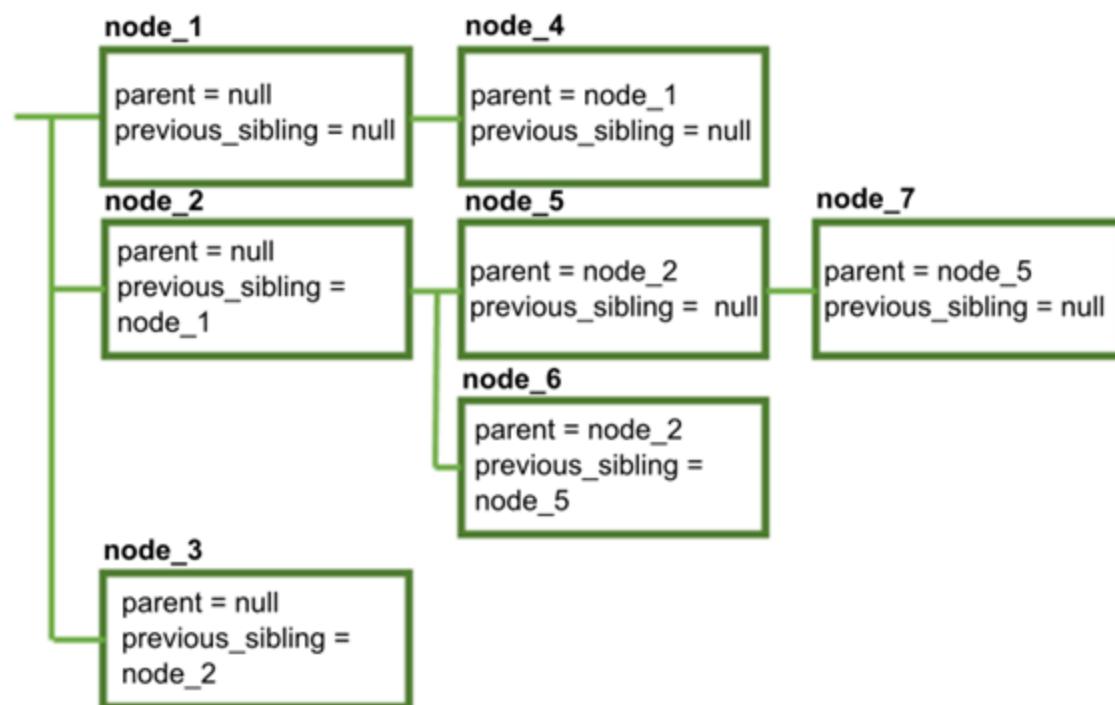
- A node of type `event_handler` with an `event_name` of `generic` can have a parent of type `slot` or `frame`.
- A node of type `event_handler` with an `event_name` of `focus`, `input`, `filled`, or `nomatch` must have a parent of type `slot`.
- If more than one `event_handler` with the same `event_name`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

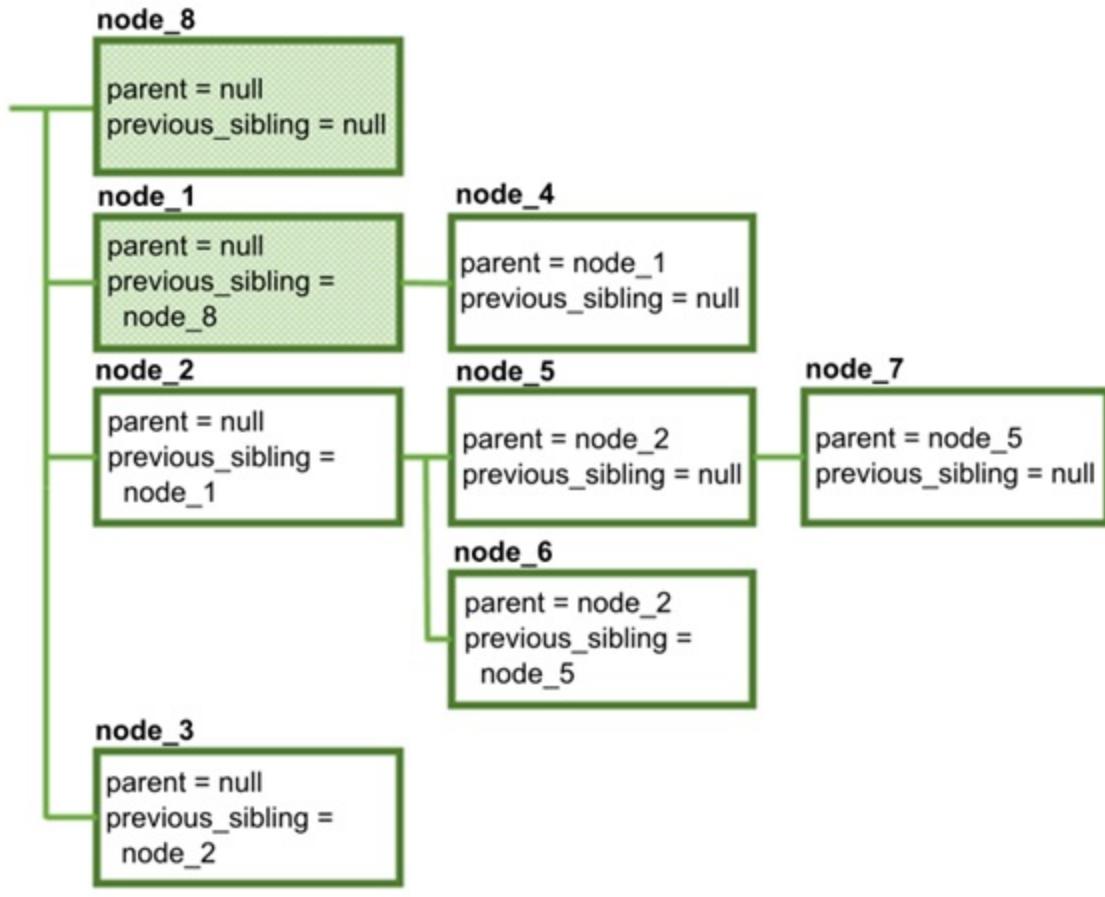
Consider the following simple dialog tree:



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.

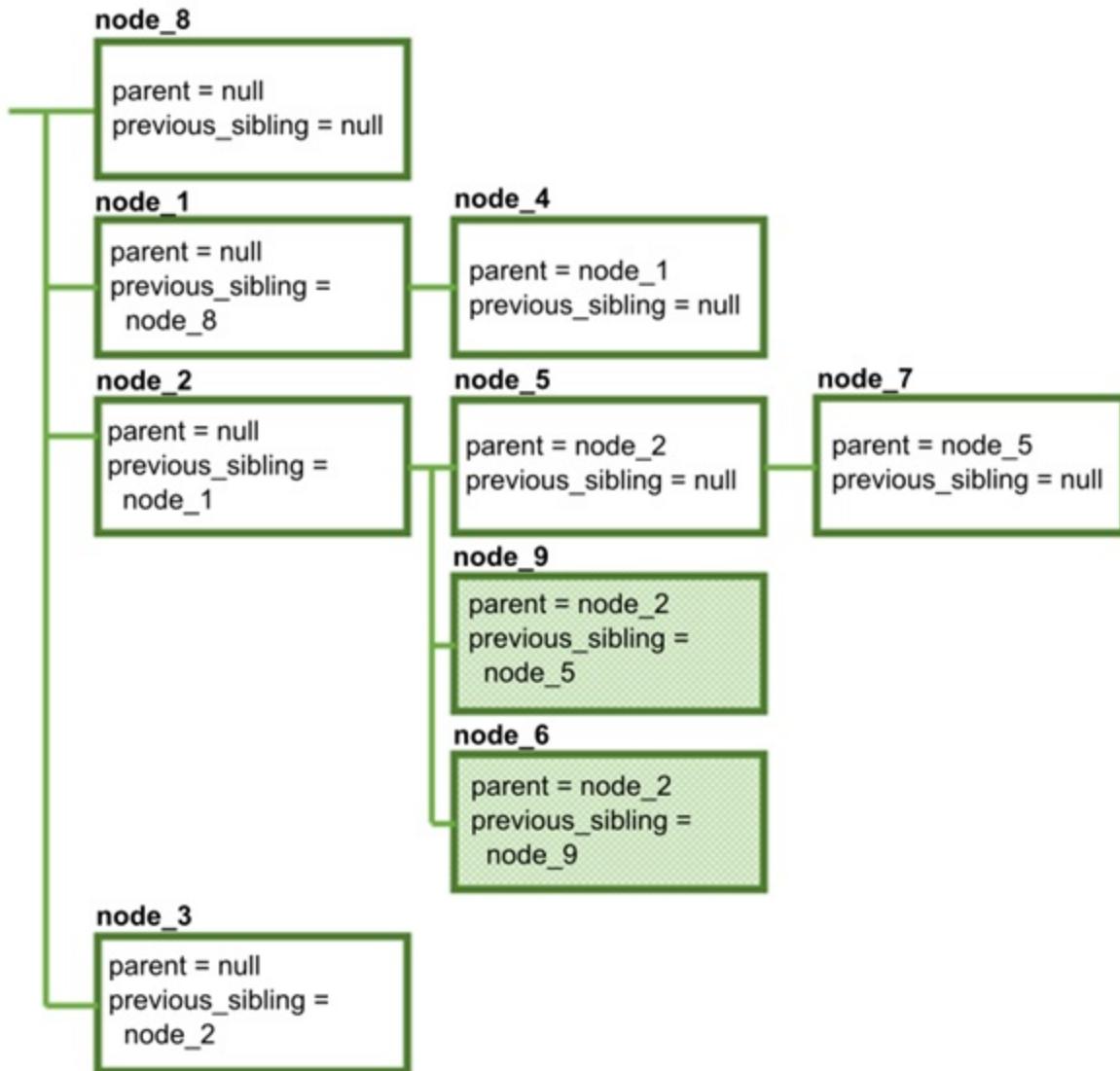
You can create a node somewhere else in the dialog by specifying the parent and previous sibling:

```
$ {
 "dialog_node": "node_9",
 "parent": "node_2",
 "previous_sibling": "node_5"
}
```

The values you specify for `parent` and `previous_sibling` 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

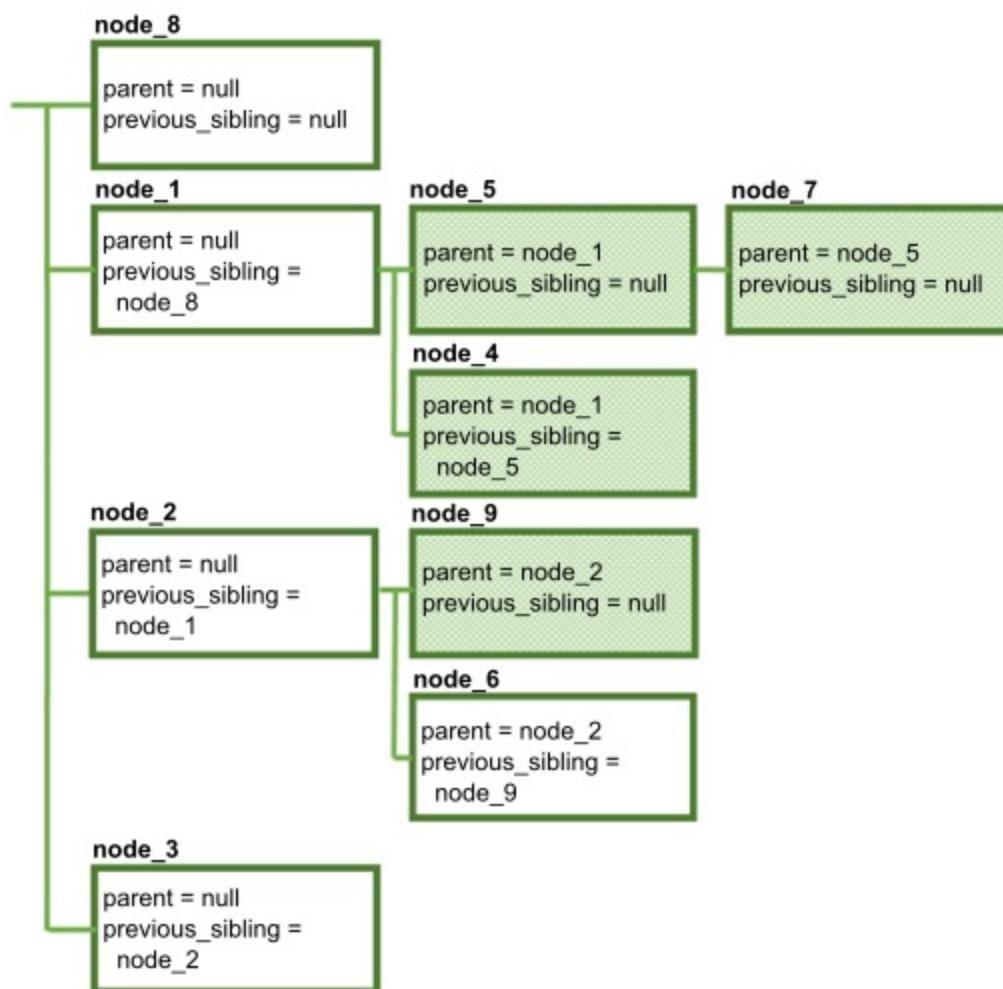
Let's move **node\_5** to a different parent by using the POST /dialog\_nodes/node\_5 method with the following body:

```
$ {
 "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

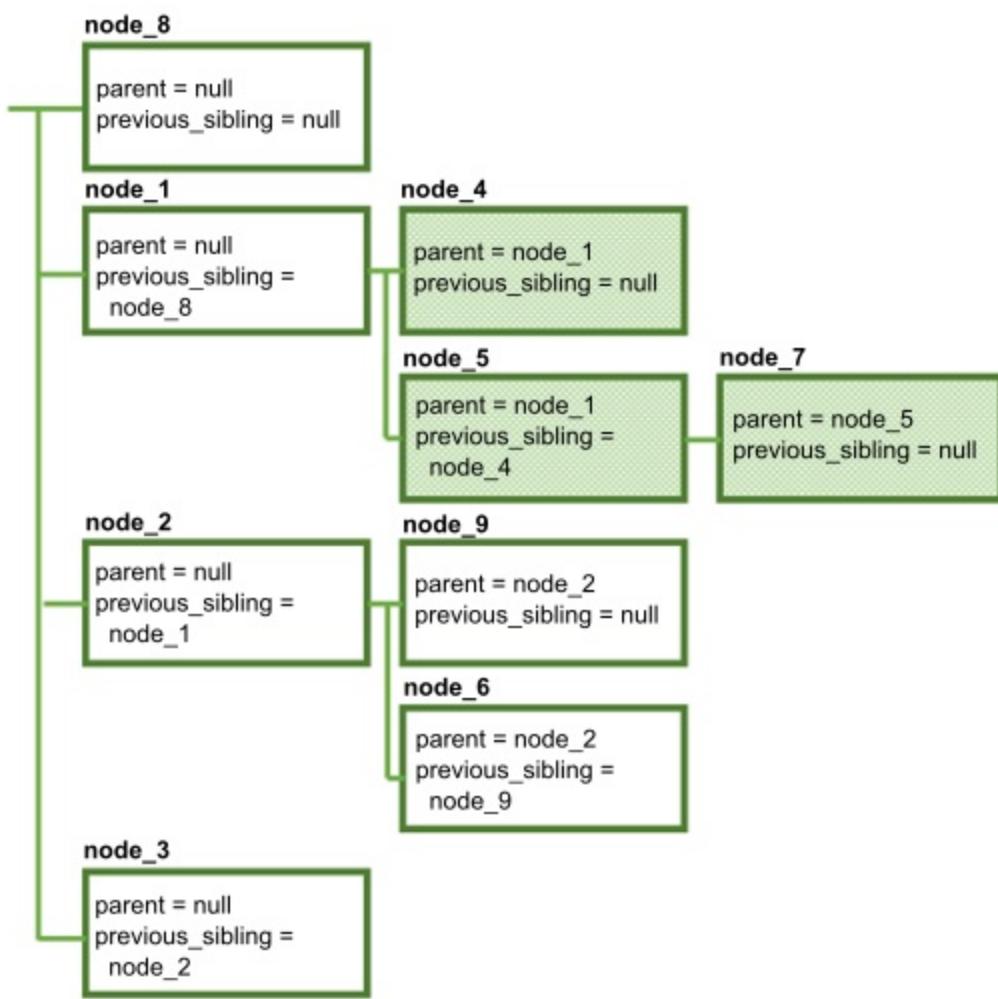
Now let's make **node\_5** the second sibling instead of the first. We can do this by using the POST /dialog\_nodes/node\_5 method with the following body:

```
$ {
 "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:

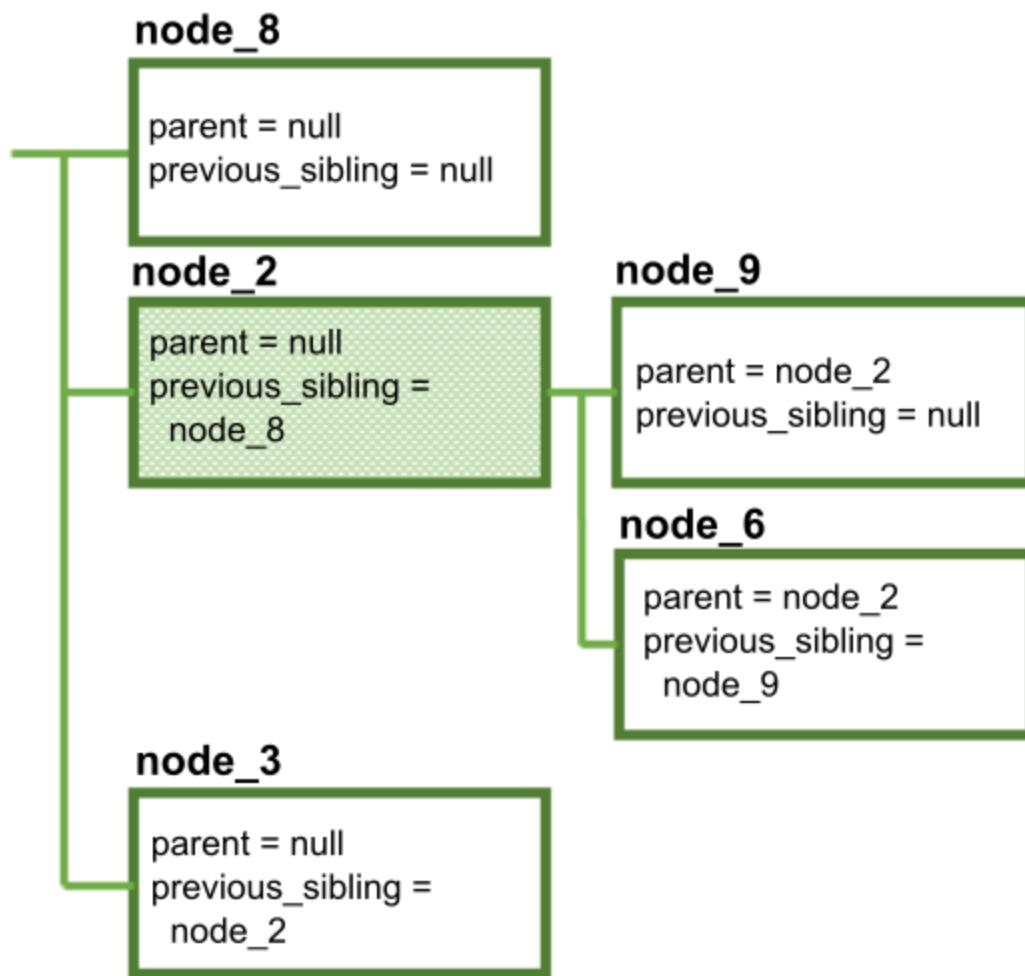


Note that once again, **node\_7** stays with its parent. In addition, **node\_4** is modified so that its `previous_sibling` is `null`, because it is now the first sibling.

## Deleting a node

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

The result is this:



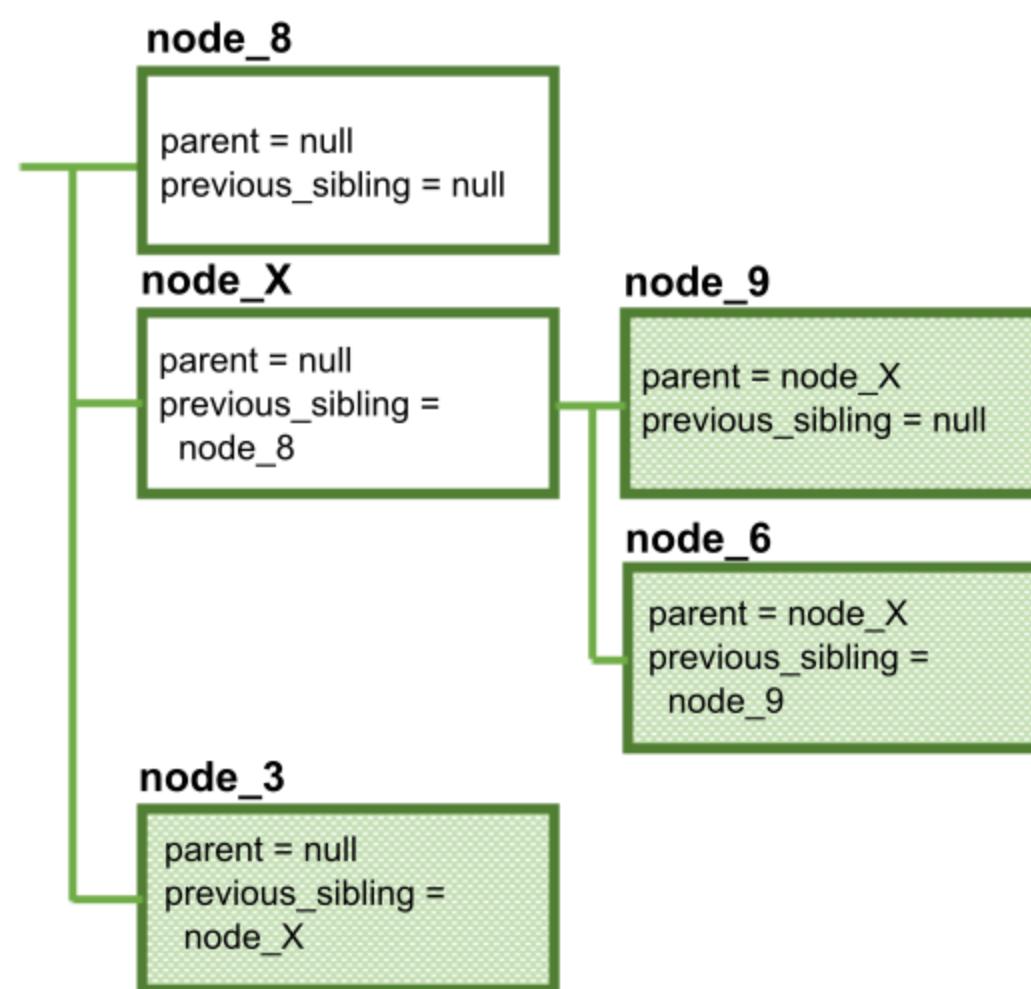
Note that **node\_1**, **node\_4**, **node\_5**, and **node\_7** were all deleted. When you delete a node, all descendants of that node are deleted as well. Therefore, if you delete a root node, you are actually deleting an entire branch of the dialog tree. Any other references to the deleted node (such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

## Web chat development

**⚠️ Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

### Web chat overview

Learn more about the web chat that you can add to your company website.

Web chat is a code snippet that you can immediately embed in your website.

When you build a custom user interface for your assistant, you spend a lot of time and effort writing code to solve typical UI problems. For example, you must keep up with browser support changes, manage scrolling behavior, validate input, and design the layout and styling. The time you spend designing and maintaining a UI can be better spent building a quality assistant instead. When you use the web chat integration, you can rely on us to manage the user interface, so you can focus on designing conversational exchanges that address the unique business needs of your customers. Cutting-edge functionality from IBM Design and Research is incorporated into the web chat to deliver an exceptional user experience.

For more information about how to deploy the web chat, see [Integrating the web chat with your website](#).

### Browser support

The web chat supports a variety of devices and platforms. As a general rule, if the last two versions of a browser account for more than 1% of all desktop or mobile traffic, the web chat supports that browser.

The following list specifies the minimum required browser software for the web chat (including the two most recent versions, except as noted):

- Google Chrome
- Apple Safari
- Mobile Safari
- Chrome for Android
- Microsoft Edge (Chromium and non-Chromium)
- Mozilla Firefox

- Firefox ESR (most recent ESR only)
- Opera
- Samsung Mobile Browser
- UC Browser for Android
- Mobile Firefox

For optimal results when rendering the web chat on mobile devices, the `<head>` element of your web page must include the following metadata element:

```
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

## Global audience support

The underlying skills understand customer messages that are written in any of the languages that are supported by the service. For more information, see [Supported languages](#). The responses from your assistant are defined by you in the underlying skill and can be written in any language you want.

Even if your skill includes responses in a language other than English, some of the phrases that are displayed in the web chat widget are added by the web chat itself and do not come from the underlying skill. These hardcoded phrases are specified in English unless you choose to apply a different language.

There are language files that contain translations of each English-language phrase that is used by the web chat. You can instruct the web chat to use one of these other languages files by using the `instance.updateLanguagePack()` method.

Likewise, the web chat applies an American English locale to content that is added by the web chat unless you specify something else. The locale setting affects how values such as dates and times are formatted.

To configure the web chat for customers outside the US, follow these steps:

1. To apply the appropriate syntax to dates and times *and* to use a translation of the English-language phrases, set the locale. Use the `instance.updateLocale()` method.

For example, if you apply the Spanish locale (`es`), the web chat uses Spanish-language phrases that are listed in the `es.json` file, and uses the `DD-MM-YYYY` format for dates instead of `MM-DD-YYYY`.

 **Note:** The locale you specify for the web chat does not impact the syntax of dates and times that are returned by the underlying skill.

2. To change only the language of the hardcoded English phrases, use the `instance.updateLanguagePack()` method.

For more information, see [Instance methods](#).

3. To change the text direction of the page from right to left, use the `direction` method. For more information, see [Configuration](#).

## Customizing phrases

You can edit the hardcoded phrases that are displayed in the web chat to customize them for branding or style. Whether your web chat responds in English or another language, you can customize the wording that is used. For example, you might want to replace the phrase that says, `The live agent is typing` with the phrase, `A customer support agent is typing`.

The language files contain [ICU Message Format](#) JSON representations of all of the languages that are supported by the web chat. You can edit all or a subset of the phrases in a JSON file, and then configure the web chat to use your version by specifying the `instance.updateLanguagePack()` method.

For more information, see [Languages](#).

## Versioning

Web chat follows semantic versioning practices. Starting with web chat version 2.0.0, you can set the version of web chat that you want to use as a configuration option.

If you don't specify a version, the latest version is used automatically (`clientVersion="latest"`). When you apply the latest version, you benefit from the continuous improvements, feature additions, and bug fixes that are made to the web chat regularly.

However, if you apply extensive customizations to your deployment, such as overriding the theme with your own custom theme, for example, you might want to lock your deployment to a specific version. Locking on a version enables you to test new versions before you apply them to your live web chat.

To use a specific version (`clientVersion="major.minor.patch"`), specify it as follows:

The following examples show what to specify when the current version is `2.3.1`.

- If you want to stay on a major version, but get the latest minor and patch releases, specify `clientVersion="2"`.
- If you want to stay on a minor version, but get the latest patch releases, specify `clientVersion="2.3"`.
- If you want to lock on to a specific minor version and patch release, specify `clientVersion="2.3.1"`.

To test the updates in a version release of web chat before you apply the version to your live web chat, follow these steps:

1. Lock the web chat that is running in your production environment to a specific version.
2. Embed your web chat deployment into a new page in a test environment, and then override the version lock setting. For example, specify `clientVersion="latest"`.
3. Test the web chat, and make adjustments to the configuration if necessary.
4. Update your production deployment to use the latest version and apply any other configuration changes that you determined to be necessary after testing.

For more information about features that were introduced in previous web chat versions, see the [Web chat release notes](#).

## Billing

Watson Assistant charges based on the number of unique monthly active users (MAU).

The web chat uses the following methods to track users:

- You can pass in a de-identified user ID either as part of a secure JWT or as a string passed to `instance.updateUserID` method. If you are using advanced security features, the user ID is derived from the sub claim in the JWT.
- If you do not pass an identifier for the user when the session begins, the web chat creates one for you. It creates a first-party cookie with a generated anonymous ID. The cookie remains active for 45 days. If the same user returns to your site later in the month and chats with your assistant again, the web chat integration recognizes the user. And you are charged only once when the same anonymous user interacts with your assistant multiple times in a single month.

## Apple devices

On Apple devices, the Intelligent Tracking Prevention feature automatically deletes any client-side cookie after 7 days. This means that if an anonymous customer accesses your website and then visits again two weeks later, the two visits are treated as two different MAUs.

To avoid this problem, use a server-side first-party cookie in your web application. For example, when an anonymous user visits your website for the first time, you can generate a unique user ID and store it in a server-side cookie with any expiration date you choose. Then, your code can use the [updateUserID\(\)](#) instance method to set the user ID. You can then use the same cookie to set the same user ID for this customer on any future visits until it expires.

## More information

For more information about billing, see [User-based plans explained](#).

For more information about MAU limits per plan, see [Web chat integration limits](#).

For more information about deleting a user's data, see [Labeling and deleting data](#).

## Human agent service integration

You can configure the web chat to transfer a customer to a human customer support agent if the customer asks for help from a person. The following service desk integrations are supported:

Built-in service desk integrations:

- [Intercom](#) 
- [Web chat with Salesforce support](#)
- [Web chat with Zendesk support](#)

Web chat service desk reference implementations:

- [Genesys Cloud](#)
- [NICE inContact](#)
- [Twilio Flex](#)
- [Bring your own \(starter kit\)](#)

## Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

The web chat integration is enabled for [Web Content Accessibility 2.1 Level AA](#) compliance. It is tested with both screen readers and automated tools on a continual basis.

## More information

To learn more about advanced tasks you can perform with the web chat, read the following topics:

- [Securing the web chat](#)
- [Applying advanced customizations](#)

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Applying advanced customizations

Tailor the web chat to match your website and brand, and to behave in ways that meet your business needs.

### API overview

The following APIs are available:

- **Configuration object:** When the embedded web chat widget starts, a configuration object named `watsonAssistantChatOptions` is used to define the widget. By editing the configuration object, you can customize the appearance and behavior of the web chat before it is rendered.
- **Runtime methods:** Use the instance methods to perform tasks before a conversation between the assistant and your customer begins or after it ends.
- **Events:** Your website can listen for these events, and then take custom actions.

A developer can use these APIs to customize the web chat in the following ways:

- [Change how the web chat opens and closes](#)
- [Change the conversation](#)
- [Change the look of the web chat](#)

### Change how the web chat opens and closes

You can change the color of the launcher icon from the `Style` tab of the web chat configuration page. If you want to make more advanced customizations, you can make the following types of changes:

- Change the launcher icon that is used to open the web chat widget. For a tutorial that shows you how, see [Using a custom launcher](#).
- Change how the web chat widget opens. For example, you might want to launch the web chat from some other button or process that exists on your website, or maybe open it in a different location, or at a different size. For a tutorial that shows you how, see [Render to a custom element](#).
- Hide the launcher icon entirely and automatically start the web widget in open state, at its full length. For more information, see the [openChatByDefault method](#).
- Hide the close button so users cannot close the web chat widget. For more information, see the [hideCloseButton method](#).

### Change the conversation

The core of the conversation is defined in your skill. If you use more than one integration type, focus on defining an engaging conversation in the underlying skill. The same skill is used for all integrations. To customize the conversation for the web chat only, you can take the following actions:

- Update the text of a message before it is sent or after it is received, such as to hide personally-identifiable information.
- Pass contextual information, such as the customer's name, from the web chat to the underlying skill. For examples of how to complete common tasks, see [Passing values](#).
- Change the language that is used by the web chat. For more information, see [Global audience support](#).
- Render your own custom response types inside the web chat widget, including responses that incorporate code from your website at run time. For a tutorial that shows you how, see [Creating a custom response](#).
- Use React portals to render your custom response type as part of your application. For a tutorial that shows you how, see [Custom responses with React](#).

### Change the look of the web chat

You can make simple changes to the color of things like the text font and web chat header from the `Style` tab of the web chat configuration page. To make more extensive style changes, you can set the CSS style color theme to a different theme or specify your own theme.

- You can choose to use a different base Carbon Design theme. The supported base themes are color themes that are defined by [IBM Carbon Design](#). For more information, see [Theming](#).
- Alternatively, you can set individual variables within the theme to customize specific UI elements. For example, the text that is displayed in the chat window uses the fonts `IBMPlexSans`, `Arial`, `Helvetica`, `sans-serif`. If you want to use a different font, you can specify it by using the `instance.updateCSSVariables()` method.
- Apply style settings to user-defined response types. If you enable the web chat to return custom response types, be sure to apply existing or custom CSS classes to them. For more information, see [Custom content](#).
- Change the style of the home screen that can be displayed when the web chat is opened. For more information about the home screen, see [Adding a home screen](#). For more information about how to customize it, see [HTML content](#)



**Note:** The web chat is embedded directly on your page, not inside an iframe. Therefore, the cascading style sheet (CSS) rules for your website can sometimes override the web chat CSS rules. The web chat applies aggressive CSS resets, but the resets can be affected if your website uses the `!important` property in elements where style is defined.

### Passing values

Here are some common tasks you might want to perform:

- [Setting and passing context variable values](#)
- [Adding user identity information \(if you don't enable security\)](#)

For a tutorial that describes how to set context values from the web chat, see [Setting context](#).

## Setting and passing context variable values

A context variable is a variable that you can use to pass information to your assistant before a conversation starts. It can also collect information during a conversation, and reference it later in the same conversation. For example, you might want to ask for the customer's name and then address the person by name later on.

The following script preserves the context of the conversation. In addition, it adds an `$ismember` context variable and sets it to `true`.

The name that is specified for the skill (`main skill`) is a hardcoded name that is used to refer to any skill that you create from the product user interface. You do not need to edit your skill name.

```
<script>
function preSendhandler(event) {
 event.data.context.skills['main skill'].user_defined.ismember = true;
}
window.watsonAssistantChatOptions = {
 integrationID: "YOUR_INTEGRATION_ID",
 region: "YOUR_REGION",
 serviceInstanceId: "YOUR_SERVICE_INSTANCE_ID",

onLoad: function(instance) {
 // Subscribe to the "pre:send" event.
 instance.on({ type: "pre:send", handler: preSendhandler });
 instance.render();
}
};

setTimeout(function(){
 const t=document.createElement('script');
 t.src='https://web-chat.global.assistant.dev.watson.appdomain.cloud/versions/' +
 (window.watsonAssistantChatOptions.clientVersion || 'latest') +
 '/WatsonAssistantChatEntry.js';
 document.head.appendChild(t);
};

</script>
```

You can reference the `$ismember` context variable from your dialog. For example, the following screen capture shows a dialog node that conditions on `#General_Greetings`. It has multiple conditioned responses. The first response checks whether the current user is a member of your rewards program by checking for the presence of the `$ismember` context variable. If the variable is present, the response addresses the user as a member. The next response has a more generic greeting.

If assistant recognizes:



Then respond with

IF ASSISTANT RECOGNIZES	RESPOND WITH	
1    \$ismember	Hello member!	
2    true	Hi there, non-member.	

If you enable security, you can encrypt the data that you pass to your dialog. For more information, see [Passing sensitive data](#).

**⚠️ Important:** If you're using a Lite plan, remember that a session ends if there's no interaction with the user after 5 minutes. When the session ends, any contextual information that you pass or collect is reset.

## Adding user identity information

If you do not enable security, and you want to perform tasks where you need to know the user who submitted the input, then you must pass the user ID to the web chat integration.

If you do enable security, you set the user ID in the JSON Web Token instead. For more information, see [Authenticating users](#).

Choose a non-human-identifiable ID. For example, do not use a person's email address as the `user_id`.

User information is used in the following ways:

- User-based service plans use the `user_id` associated with user input for billing purposes. See [User-based plans](#).
- The ability to delete any data created by someone who requests to be forgotten requires that a `customer_id` be associated with the user input. When a `user_id` is defined, the product can reuse it to pass a `customer_id` parameter. See [Labeling and deleting data](#).

**Note:** Because the `user_id` value that you submit is included in the `customer_id` value that is added to the `X-Watson-Metadata` header in each message request, the `user_id` syntax must meet the requirements for header fields as defined in [RFC 7230](#).

To support these user-based capabilities, add the [updateUserID\(\) method](#) in the code snippet before you paste it into your web page.

In the following example, the user ID `L12345` is added to the script.

```
<script>
window.watsonAssistantChatOptions = {
 integrationID: 'YOUR_INTEGRATION_ID',
 region: 'YOUR_REGION',
 serviceInstanceId: 'YOUR_SERVICE_INSTANCE',
 onLoad: function(instance) {
 instance.updateUserID('L12345');
 instance.render();
 }
};
setTimeout(function(){
 const t=document.createElement('script');
 t.src='https://web-chat.global.assistant.dev.watson.appdomain.cloud/versions/' +
 (window.watsonAssistantChatOptions.clientVersion || 'latest') +
 '/WatsonAssistantChatEntry.js';
 document.head.appendChild(t);
});
</script>
```

**⚠️ Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

## Securing the web chat

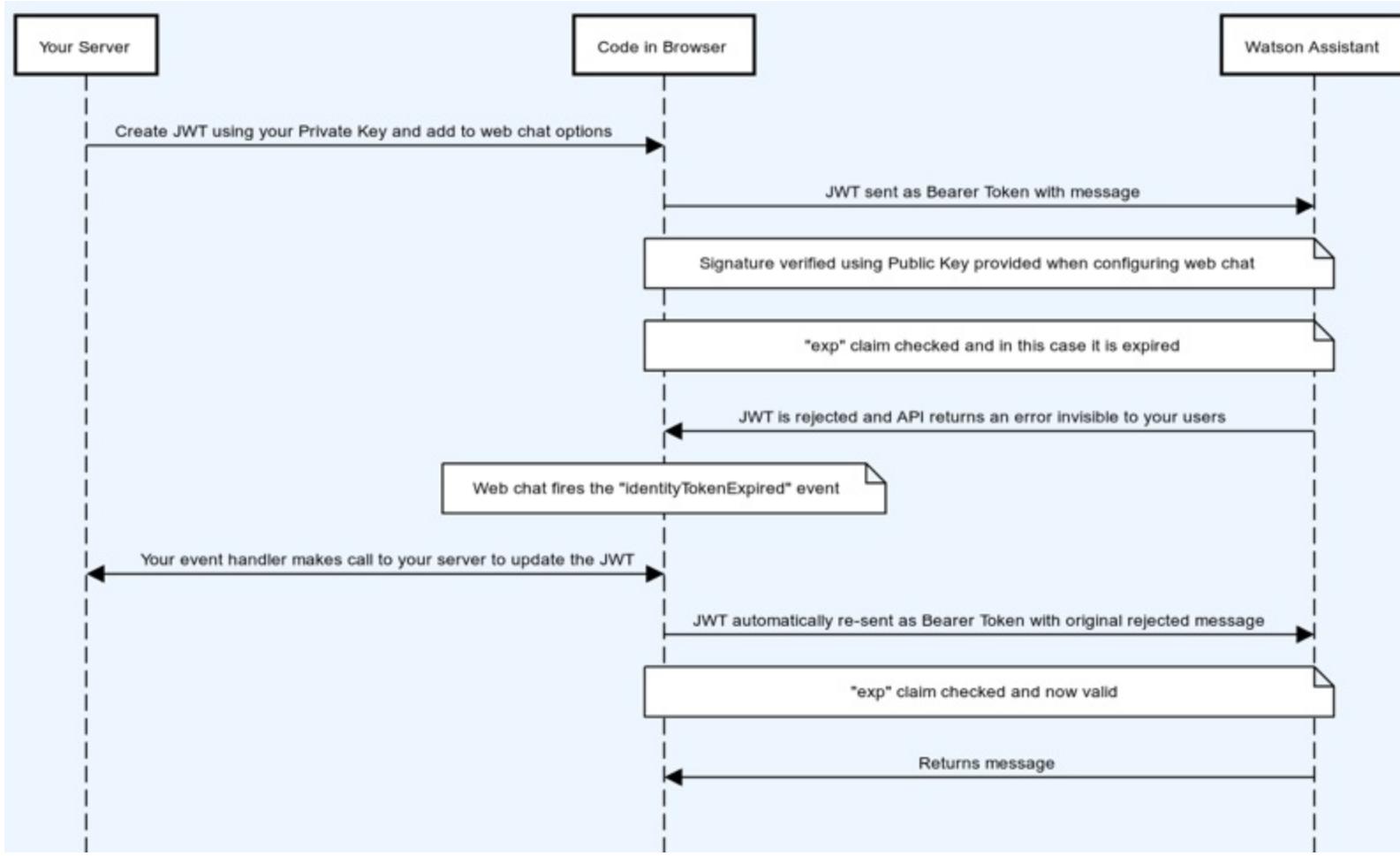
Understand what you need to do to secure your web chat integration.

Configure the web chat to authenticate users and send private data from your embedded web chat.

All messages that are sent from the web chat are encrypted. When you enable security, your assistant takes an additional step to verify that messages originate from the web chat that is embedded in your website only.

The web chat uses an RSA signature with SHA-256 (RS256) to encrypt communication. RS256 signatures use a sophisticated type of RSA encryption. An RSA key pair includes a private and a public key. The RSA private key is used to generate digital signatures, and the RSA public key is used to verify digital signatures. The complexity of the RSA algorithm that is used to scramble the message makes it nearly impossible to unscramble the message without the key.

The following diagram illustrates the requests that are sent back and forth to authenticate a request.



You can implement the following security measures:

- Ensure that messages sent from the web chat to your assistant come from your customers only
- Send private data from the web chat to your assistant

## Before you begin

The process you use to add the web chat to your website is simple. Its simplicity also means it can be misused. That's why it's important to verify that the messages sent to your assistant are coming from authorized users only.

Before you enable security, complete the following steps:

1. Create a RS256 private/public key pair.

You can use a tool such as the OpenSSL command line or PuTTYgen.

- For example, to create the key pair: `openssl genrsa -out key.pem 2048`

2. Use your private key to sign a JSON Web Token (JWT). You will pass the token with the messages that are sent from your website as proof of their origin.

The JWT payload must specify values for the following claims:

- `iss` : Represents the issuer of the JWT. This value is a case-sensitive string.
- `sub` : Represents the principal that is the subject of the JWT. This value must either be scoped to be locally unique in the context of the issuer or be globally unique. The value you specify for `sub` is used as the `user_id`.

The user ID that is specified in the `sub` claim is also sent in the `customer_id` section of the `X-Watson-Metadata` HTTP header. The `customer_id` can be used to make requests to delete user data. Because the ID is sent in a header field, the syntax must meet the requirements for header fields as defined in [RFC 7230](#) (all visible ASCII characters). For more information about deleting user data, see [Labeling and deleting data](#).

- `exp` : Represents the expiration time on or after which the JWT cannot be accepted for processing. Many libraries set this value for you automatically. Set a short-lived `exp` claim with whatever library you use.

For more information about JSON Web Tokens, see the [RFC7519](#) and [OpenID Connect 1.0](#) specifications.

Most programming languages offer JWT libraries that you can use to generate a token. The following NodeJS code sample illustrates how to generate a JWT token.

```

// Sample NodeJS code on your server.
const jwt = require('jsonwebtoken');

/**
 * Returns a signed JWT generated by RS256 algorithm.
 */
function mockLogin() {
 const payload = {
 /*
 * Even if this is an unauthenticated user, add a userID in the sub claim that can be used
 */

```

```

 * for billing purposes.
 * This ID will help us keep track "unique users". For unauthenticated users, drop a
 * cookie in the browser so you can make sure the user is counted uniquely across visits.
 */
 sub: 'some-user-id', // Required
 iss: 'yourdomain.com' // Required
};

// The "expiresIn" option adds an "exp" claim to the payload.
return jwt.sign(payload, process.env.YOUR_PRIVATE_RSA_KEY, { algorithm: 'RS256', expiresIn: '10000ms' });
}

```

## Enable security

To enable security, complete the following steps:

1. From the **Security** tab of the web chat integration setup page in Watson Assistant, set the **Secure your web chat** switch to **On**.
2. Add your public key to the **Your public key** field.

The public key that you add is used to verify that data that claims to come from your web chat instance *is* coming from your web chat instance.

3. To prove that a message is coming from your website, each message that is submitted from your web chat implementation must include the JSON Web Token (JWT) that you created earlier.

Add the token to the web chat code snippet that you embed in your website page. Specify the token in the **identityToken** property.

 **Note:** Starting with web chat version **3.2.0**, this step is optional. You do not have to add the **identityToken** property initially. You can skip this step as long as you perform the next step where you add the **identityTokenExpired** event. The event is fired when the web chat is first opened, and if a token wasn't provided, it obtains one from your handler at that time.

For example:

```

<script>
window.watsonAssistantChatOptions = {
 integrationID: 'YOUR_INTEGRATION_ID',
 region: 'YOUR_REGION',
 serviceInstanceId: 'YOUR_SERVICE_INSTANCE',
 identityToken: 'YOUR_JWT',
 onLoad: function(instance) {
 instance.render();
 }
};
setTimeout(function(){
 const t=document.createElement('script');
 t.src="https://web-chat.global.assistant.watson.appdomain.cloud/loadWatsonAssistantChat.js";
 document.head.appendChild(t);
});
</script>

```

The JSON Web Token is automatically included on each subsequent request that is sent from the web chat until it expires.

4. You can add an event that is triggered when your token expires, or (starting with web chat version 3.2.0) when no token is specified initially. The event has a callback you can use to update the token and to process any messages that were queued for processing during the time the token was expired.

For example:

```

<script>
window.watsonAssistantChatOptions = {
 integrationID: 'YOUR_INTEGRATION_ID',
 region: 'YOUR_REGION',
 serviceInstanceId: 'YOUR_SERVICE_INSTANCE',
 identityToken: 'YOUR_JWT',
 onLoad: function(instance) {
 instance.on({ type: 'identityTokenExpired', handler: function(event) {
 // Perform whatever actions you need to take on your system to get a new token.
 return new Promise(function(resolve, reject) {
 // And then pass the new JWT into the callback and the service will resume processing messages.
 event.identityToken = 'YOUR NEW JWT';
 resolve();
 });
 }});
 instance.render();
 }
};
setTimeout(function(){

```

```

const t=document.createElement('script');
t.src="https://web-chat.global.assistant.watson.appdomain.cloud/loadWatsonAssistantChat.js";
document.head.appendChild(t);
};

</script>

```

## Passing sensitive data

You can optionally copy the public key that is provided by IBM, and use it to add an additional level of encryption to support passing sensitive data from the web chat.

Use this method to send sensitive information in messages that come from your website, such as a information about a customer's loyalty level, a user ID, or security tokens to use in webhooks that you call from your dialog. Information that is passed to your assistant in this way is stored in a private variable in your assistant. Private variables cannot be seen by customers and are never sent back to the web chat.

For example, you might start a business process for a VIP customer that is different from the process you start for less important customers. You likely do not want non-VIPs to know that they are categorized as such. But you must pass this informataion to your dialog because it changes the route of the conversation. You can pass the customer MVP status as an encrypted variable. This private context variable will be available for use by the dialog, but not by anything else.

1. From the web chat configuration page, copy the public key from the **IBM provided public key** field.
2. From your website, write a function that signs a JSON Web Token.

For example, the following NodeJS code snippet shows a function that accepts a userID and payload content and sends it to the web chat. If a payload is provided, its content is encrypted and signed with the IBM public key.

```

// Sample NodeJS code on your server.
const jwt = require('jsonwebtoken');
const RSA = require('node-rsa');

const rsaKey = new RSA(process.env.PUBLIC_IBM_RSA_KEY);

/**
 * Returns a signed JWT. Optionally, also adds an encrypted user_payload.
 * The userPayload is stringified JSON.
 */
function mockLogin(userID, userPayload) {
const payload = {
 sub: userID, // Required
 // The exp claim is automatically added by the jsonwebtoken library.
 // We recommend you set a short lived exp claim
 // with whatever library you are using.
};
if (userPayload) {
 // If there is a user payload we encrypt it using the IBM public key.
 // Should be encrypted to base64 format.
 payload.user_payload = rsaKey.encrypt(userPayload, 'base64');
}
const token = jwt.sign(payload, process.env.YOUR_PRIVATE_RSA_KEY, { algorithm: 'RS256', expiresIn: '10000ms' });
return token;
}

```

3. The encrypted user payload is decrypted and then saved to the `context.integrations.chat.private.user_payload` object.

For information about how to access the payload data from the dialog, see [Web chat: Accessing sensitive data](#). You might want to access the payload, for example, to get the customer importance information or single sign-on credentials that you can subsequently use to authenticate a webhook.

## Authenticating users

To authenticate and specify a unique ID for each customer, add the user ID information to the token.

1. From the web chat configuration page, copy the public key from the **IBM provided public key** field. You will specify this value as the `PUBLIC_IBM_RSA_KEY` later.
2. From your website, write a function that signs a JSON Web Token.

The function must accept a UserID parameter and set the userID as the `sub` claim value.

For example, the following NodeJS code snippet set the user's ID to `L12345`.

```

// Sample NodeJS code on your server.
const jwt = require('jsonwebtoken');

```

```

const RSA = require('node-rsa');

const rsaKey = new RSA(process.env.PUBLIC_IBM_RSA_KEY);

/**
 * Returns a signed JWT. Optionally, adds an encrypted user payload in stringified JSON.
 */
function mockLogin() {
 const payload = {
 sub: 'L12345', // Required
 iss: 'www.example.com' // Required
 };
 const token = jwt.sign(payload, process.env.YOUR_PRIVATE_RSA_KEY, { algorithm: 'RS256', expiresIn: '10000ms' });
 return token;
}

```

After you set the value of the `sub` claim to be the user's userID, you cannot change the claim to another user. The userID that you specify with this method is used for billing purposes and can be used to delete customer data upon request.

The userID is stored in the context object as `context.global.system.user_id`.

The user ID that is specified in the `sub` claim is also sent in the `customer_id` section of the `X-Watson-Metadata` HTTP header. The `customer_id` can be used to make requests to delete user data. Because the ID is sent in a header field, the syntax must meet the requirements for header fields as defined in [RFC 7230](#) (all visible ASCII characters). For more information about deleting user data, see [Labeling and deleting data](#).

If you disable security, then you can use the `instance.updateUserID()` method to specify user IDs. For more information, see [Adding user identity information](#).

## Logging out

To log out a customer, you must destroy the web chat.

If you reload the page when a customer logs out, you must call the `instance.destroySession()` method to remove any reference to the current session from the browser's cookies and storage. If you forget to call this method, information that is protected by the `identityToken` is not at risk, but the web chat will try to connect to the previous session and fail.

If you do not perform a full page reload when a customer logs out, call the `instance.destroy()` method. The `destroy` method removes the current instance of the web chat that is configured for the current userID from the DOM and browser memory. Next, call the `instance.destroySession()` method.

## Updating site security policies

If your website uses a Content Security Policy (CSP), you must update it to grant permission to the web chat.

The following table lists the values to add to your CSP.

Property	Additional values
default-src	'self' *.watson.appdomain.cloud fonts.gstatic.com 'unsafe-inline'
connect-src	*.watsonplatform.net *.watson.appdomain.cloud
CSP properties	

The following example shows a complete CSP metadata tag:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self' *.watson.appdomain.cloud fonts.gstatic.com 'unsafe-inline';connect-src *.watsonplatform.net *.watson.appdomain.cloud" />
```

## Allowing elements

If your CSP uses a nonce to add elements, such as `<script>` and `<style>` tags, to an allowlist, do not use `unsafe-inline` to allow all such elements. Instead, provide a nonce value to the web chat widget as a configuration option. The web chat will then set the nonce on any of the `<script>` and `<style>` elements that it generates dynamically.

A CSP that passes a nonce to the web chat widget might look like this:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self' *.watson.appdomain.cloud fonts.gstatic.com 'nonce-';connect-src *.watsonplatform.net *.watson.appdomain.cloud"
```

You can pass the nonce to the web chat by editing the embed script as follows:

```
window.watsonAssistantChatOptions = {
 integrationID: "YOUR_INTEGRATION_ID",
 region: "YOUR_REGION",
 serviceInstanceID: "YOUR_SERVICE_INSTANCE",

 cspNonce: "<server generated value>",

 onLoad: function(instance) {
 instance.render();
 }
};
```

## Security measures

The web chat integration undergoes tests and scans on a regular basis to find and address potential security issues, such as cross-site scripting (XSS) vulnerabilities.

Be sure to run your own security reviews to see how the web chat fits in with your current website structure and policies. The web chat is hosted on your site and can inherit any vulnerabilities that your site has. Only serve content over HTTPS, use Content Security Policy (CSP), and implement other basic web security precautions.

## Custom UI development

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Watson Assistant API overview](#).

## Watson Assistant API overview

You can use the Watson Assistant REST APIs, and the corresponding SDKs, to develop applications that interact with the service.

## Client applications

To build a virtual assistant or other client application that communicates with an assistant at run time, use the new v2 API. By using this API, you can develop a user-facing client that can be deployed for production use, an application that brokers communication between an assistant and another service (such as a chat service or back-end system), or a testing application.

By using the v2 runtime API to communicate with your assistant, your application can take advantage of the following features:

- **Automatic state management.** The v2 runtime API manages each session with an end user, storing and maintaining all of the context data your assistant needs for a complete conversation.
- **Ease of deployment using assistants.** In addition to supporting custom clients, an assistant can be easily deployed to popular messaging channels such as Slack and Facebook Messenger.
- **Versioning.** With dialog skill versioning, you can save a snapshot of your skill and link your assistant to that specific version. You can then continue to update your development version without affecting the production assistant.
- **Search capabilities.** The v2 runtime API can be used to receive responses from both dialog skills and search skills. When a query is submitted that your dialog skill cannot answer, the assistant can use a search skill to find the best answer from the configured data sources. (Search skills are a beta feature available only to Plus or Premium plan users.)

For details about the v2 API, see the Watson Assistant [v2 API Reference](#).

**Note:** The Watson Assistant v1 API still supports the legacy `/message` method that sends user input directly to the workspace used by a dialog skill. The v1 runtime API is supported primarily for backward compatibility purposes. If you use the v1 `/message` method, you must implement your own state management, and you cannot take advantage of versioning or any of the other features of an assistant.

## Authoring applications

The v1 API provides methods that enable an application to create or modify dialog skills, as an alternative to building a skill graphically using the Watson Assistant user interface. An authoring application uses the API to create and modify skills, intents, entities, dialog nodes, and other artifacts that make up a dialog skill. For more information, see the [v1 API Reference](#).

**Note:** The v1 authoring methods create and modify workspaces rather than skills. A workspace is a container for the dialog and training data (such as intents and entities) within a dialog skill. If you create a new workspace using the API, it will appear as a new dialog skill in the Watson Assistant user interface.

For a list of the available API methods, see [API methods summary](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [API methods summary](#).

## API methods summary

The following tables list the methods that are available using the Watson Assistant APIs.

## Runtime methods

Method	Description	API ver	Reference
<b>Create a session</b>	Create a new session. A session is used to send user input to a skill and receive responses. It also maintains the state of the conversation.	v2	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete session</b>	Deletes a session explicitly before it times out.	v2	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Send user input to assistant (stateful)</b>	Send user input to an assistant and receive a response, with conversation state (including context data) stored by Watson Assistant for the duration of the session.	v2	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Send user input to assistant (stateless)</b>	Send user input to an assistant and receive a response, with conversation state (including context data) managed by your application.	v2	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>

## Authoring methods

Method	Description	API ver	Reference
<b>List workspaces</b>	List the workspaces associated with a Watson Assistant service instance.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Create workspace</b>	Create a workspace based on component objects. You must provide workspace components defining the content of the new workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get information about a workspace</b>	Get information about a workspace, optionally including all workspace content.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update workspace</b>	Update an existing workspace with new or modified data. You must provide component objects defining the content of the updated workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete workspace</b>	Delete a workspace from the service instance.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List intents</b>	List the intents for a workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Create intent</b>	Create a new intent.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get intent</b>	Get information about an intent, optionally including all intent content.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update intent</b>	Update an existing intent with new or modified data. You must provide component objects defining the content of the updated intent.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete intent</b>	Delete an intent from a workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List user input examples</b>	List the user input examples for an intent, optionally including contextual entity mentions.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>

<b>Create user input example</b>	Add a new user input example to an intent.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get user input example</b>	Get information about a user input example.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update user input example</b>	Update the text of a user input example.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete user input example</b>	Delete a user input example from an intent.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List counterexamples</b>	List the counterexamples for a workspace. Counterexamples are examples that have been marked as irrelevant input.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Create counterexample</b>	Add a new counterexample to a workspace. Counterexamples are examples that have been marked as irrelevant input.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get counterexample</b>	Get information about a counterexample. Counterexamples are examples that have been marked as irrelevant input.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update counterexample</b>	Update the text of a counterexample. Counterexamples are examples that have been marked as irrelevant input.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete counterexample</b>	Delete a counterexample from a workspace. Counterexamples are examples that have been marked as irrelevant input.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List entities</b>	List the entities for a workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Create entity</b>	Create a new entity, or enable a system entity.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get entity</b>	Get information about an entity, optionally including all entity content.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update entity</b>	Update an existing entity with new or modified data. You must provide component objects defining the content of the updated entity.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete entity</b>	Delete an entity from a workspace, or disable a system entity.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List entity mentions</b>	List mentions for a contextual entity. An entity mention is an occurrence of a contextual entity in the context of an intent user input example.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List entity values</b>	List the values for an entity.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Add entity value</b>	Create a new value for an entity.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get entity value</b>	Get information about an entity value.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update entity value</b>	Update an existing entity value with new or modified data. You must provide component objects defining the content of the updated entity value.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete entity value</b>	Delete a value from an entity.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>

<b>List entity value synonyms</b>	List the synonyms for an entity value.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Add entity value synonym</b>	Add a new synonym to an entity value.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get entity value synonym</b>	Get information about a synonym of an entity value.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update entity value synonym</b>	Update an existing entity value synonym with new text.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete entity value synonym</b>	Delete a synonym from an entity value.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List dialog nodes</b>	List the dialog nodes for a workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Create dialog node</b>	Create a new dialog node.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Get dialog node</b>	Get information about a dialog node.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Update dialog node</b>	Update an existing dialog node with new or modified data.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete dialog node</b>	Delete a dialog node from a workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List log events in a workspace</b>	List the events from the log of a specific workspace.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>List log events in all workspaces</b>	List the events from the logs of all workspaces in the service instance.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>
<b>Delete labeled data</b>	Deletes all data associated with a specified customer ID. The method has no effect if no data is associated with the customer ID.	v1	<a href="#">cURL</a> <a href="#">node</a> <a href="#">java</a> <a href="#">python</a>

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Building a custom client by using the API](#).

## Building a custom client

If none of the built-in integrations meet your requirements, you can deploy your assistant by developing a custom client application that interacts with your users, communicates with the IBM Watson® Assistant service, and integrates with other services as needed.

## Setting up the Watson Assistant service

The example application we will create in this section implements several simple functions to illustrate how a client application interacts with the service. The application code will connect to an assistant, where the cognitive processing (such as the detection of user intents) takes place.

If you want to try this example yourself, first you need to set up the required assistant:

1. Download the dialog skill [JSON file](#).
2. [Import the skill](#) into an instance of the Watson Assistant service.
3. [Create an assistant](#) and add the skill you imported.

## 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 assistant. You'll need to copy the service credentials and assistant ID and paste them into your application code. You'll also need the URL for the location of your service instance (for example, <https://api.us-south.assistant.watson.cloud.ibm.com>).

To access the service credentials and the assistant ID from the Watson Assistant tool, go to the **Assistants** page and click the  menu for the assistant you

want to connect to. Select **Settings** and then navigate to the **API details** page to see the details for the assistant, including the URL, assistant ID, and API key.

## Communicating with the Watson Assistant service

Interacting with the Watson Assistant service from your client application is simple. We'll start with an example that connects to the service, sends a single message, and prints the output to the console:

### Node

```
// Example 1: Creates service object, sends initial message, and
// receives response.

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

// Create Assistant service object.
const assistant = new AssistantV2({
 version: '2020-09-24',
 authenticator: new IamAuthenticator({
 apikey: '{apikey}', // replace with API key
 }),
 url: '{url}', // replace with URL
});

const assistantId = '{assistant_id}'; // replace with assistant ID

// Start conversation with empty message
messageInput = {
 messageType: 'text',
 text: '',
};

sendMessage(messageInput);

// Send message to assistant.
function sendMessage(messageInput) {
 assistant
 .messageStateless({
 assistantId,
 input: messageInput,
 })
 .then(res => {
 processResponse(res.result);
 })
 .catch(err => {
 console.log(err); // something went wrong
 });
}

// Process the response.
function processResponse(response) {
 // Display the output from assistant, if any. Supports only a single
 // text response.
 if (response.output.generic) {
 if (response.output.generic.length > 0) {
 if (response.output.generic[0].response_type === 'text') {
 console.log(response.output.generic[0].text);
 }
 }
 }
}
```

### Python

```
Example 1: Creates service object, sends initial message, and
receives response.

from ibm_watson import AssistantV2
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator

Create Assistant service object.
authenticator = IAMAuthenticator('{apikey}') # replace with API key
assistant = AssistantV2(
 version = '2020-09-24',
 authenticator = authenticator
)
assistant.set_service_url('{url}'
```

```

assistant_id = '{assistant_id}' # replace with assistant ID

Start conversation with empty message.
response = assistant.message_stateless(
 assistant_id,
).get_result()

Print the output from dialog, if any. Supports only a single
text response.
if response['output']['generic']:
 if response['output']['generic'][0]['response_type'] == 'text':
 print(response['output']['generic'][0]['text'])

```

## Java

```

/*
 * Example 1: Creates service object, sends initial message, and
 * receives response.
 */

import com.ibm.cloud.sdk.core.security.Authenticator;
import com.ibm.cloud.sdk.core.security.IamAuthenticator;
import com.ibm.watson.assistant.v2.Assistant;
import com.ibm.watson.assistant.v2.model.MessageResponseStateless;
import com.ibm.watson.assistant.v2.model.MessageStatelessOptions;
import com.ibm.watson.assistant.v2.model.RuntimeResponseGeneric;
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();

 // Create Assistant service object.
 Authenticator authenticator = new IamAuthenticator("{apikey}"); // replace with API key
 Assistant assistant = new Assistant("2020-09-24", authenticator);
 assistant.setServiceUrl("{url}");
 String assistantId = "{assistant_id}"; // replace with assistant ID

 // Start conversation with empty message.
 MessageStatelessOptions messageOptions = new MessageStatelessOptions.Builder(assistantId)
 .build();
 MessageResponseStateless response = assistant.messageStateless(messageOptions)
 .execute()
 .getResult();

 // Print the output from dialog, if any. Assumes a single text response.
 List<RuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
 if(responseGeneric.size() > 0) {
 if(responseGeneric.get(0).responseType().equals("text")) {
 System.out.println(responseGeneric.get(0).text());
 }
 }
 }
}

```

The first step is to create a the service object, a sort of wrapper for the Watson Assistant service.

You use the service object for sending input to, and receiving output from, the service. When you create the service object, you 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 service object is an instance of `AssistantV2`, stored in the variable `assistant`. The Watson SDKs for other languages provide equivalent mechanisms for instantiating a service object.

In this Python example, the service object is an instance of `watson_developer_cloud.AssistantV2`, stored in the variable `assistant`. The Watson SDKs for other languages provide equivalent mechanisms for instantiating a service object.

In this Java example, the service object is an instance of `Assistant`, stored in the variable `assistant`. The Watson SDKs for other languages provide equivalent mechanisms for instantiating a service object.

After creating the service object, we use it to send a message to the assistant, using the stateless `message` method. 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. We then print the response text to the console.

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 ibm-watson`.

**Note:** Make sure you have installed the Watson SDK for Python using `pip install --upgrade ibm-watson` or `easy_install --upgrade ibm-watson`.

**Note:** Make sure you have installed the [Watson SDK for Java](#).

Assuming everything works as expected, the assistant returns the output from the dialog, which the app then prints 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 client 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.

### Node

```
// Example 2: Adds user input and detects intents.

const prompt = require('prompt-sync')();
const AssistantV2 = require('ibm-watson/assistant/v2');
const { IamAuthenticator } = require('ibm-watson/auth');

// Create Assistant service object.
const assistant = new AssistantV2({
 version: '2020-09-24',
 authenticator: new IamAuthenticator({
 apikey: '{apikey}', // replace with API key
 }),
 url: '{url}', // replace with URL
});

const assistantId = '{assistant_id}'; // replace with assistant ID

// Start conversation with empty message
messageInput = {
 messageType: 'text',
 text: '',
};
sendMessage(messageInput);

// Send message to assistant.
function sendMessage(messageInput) {
 assistant
 .messageStateless({
 assistantId,
 input: messageInput,
 })
 .then(res => {
 processResponse(res.result);
 })
 .catch(err => {
 console.log(err); // something went wrong
 });
}

// Process the response.
function processResponse(response) {

 // If an intent was detected, log it out to the console.
 if (response.output.intents.length > 0) {
 console.log('Detected intent: #' + response.output.intents[0].intent);
 }

 // Display the output from assistant, if any. Supports only a single
 // text response.
 if (response.output.generic) {
 if (response.output.generic.length > 0) {
```

```

 if (response.output.generic[0].response_type === 'text') {
 console.log(response.output.generic[0].text);
 }
 }

// If we're not done, prompt for the next round of input.
const newMessageFromUser = prompt('>>');
if (newMessageFromUser !== 'quit') {
 newMessageInput = {
 messageType: 'text',
 text: newMessageFromUser,
 }
 sendMessage(newMessageInput);
}
}

```

## Python

```

Example 2: Adds user input and detects intents.

from ibm_watson import AssistantV2
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator

Create Assistant service object.
authenticator = IAMAuthenticator('{apikey}') # replace with API key
assistant = AssistantV2(
 version = '2020-09-24',
 authenticator = authenticator
)
assistant.set_service_url('{url}')
assistant_id = '{assistant_id}' # replace with assistant ID

Initialize with empty message to start the conversation.
message_input = {
 'message_type': 'text',
 'text': ""
}

Main input/output loop
while message_input['text'] != 'quit':

 # Send message to assistant.
 response = assistant.message_stateless(
 assistant_id,
 input = message_input
).get_result()

 # If an intent was detected, print it to the console.
 if response['output']['intents']:
 print('Detected intent: #' + response['output']['intents'][0]['intent'])

 # Print the output from dialog, if any. Supports only a single
 # text response.
 if response['output']['generic']:
 if response['output']['generic'][0]['response_type'] == 'text':
 print(response['output']['generic'][0]['text'])

 # Prompt for next round of input.
 user_input = input('>> ')
 message_input = {
 'text': user_input
}

```

## Java

```

/*
 * Example 2: Adds user input and detects intents.
 */

import com.ibm.cloud.sdk.core.security.Authenticator;
import com.ibm.cloud.sdk.core.security.IamAuthenticator;
import com.ibm.watson.assistant.v2.Assistant;
import com.ibm.watson.assistant.v2.model.MessageInputStateless;
import com.ibm.watson.assistant.v2.model.MessageResponseStateless;
import com.ibm.watson.assistant.v2.model.MessageStatelessOptions;
import com.ibm.watson.assistant.v2.model.RuntimeIntent;
import com.ibm.watson.assistant.v2.model.RuntimeResponseGeneric;

```

```

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();

 // Create Assistant service object.
 Authenticator authenticator = new IamAuthenticator("{apikey}"); // replace with API key
 Assistant assistant = new Assistant("2020-09-24", authenticator);
 assistant.setServiceUrl("{url}");
 String assistantId = "{assistant_id}"; // replace with assistant ID

 // Initialize with empty message to start the conversation.
 MessageInputStateless input = new MessageInputStateless.Builder()
 .messageType("text")
 .text("")
 .build();

 // Main input/output loop
 do {
 // Send message to assistant.
 MessageStatelessOptions messageOptions = new MessageStatelessOptions.Builder(assistantId)
 .input(input)
 .build();
 MessageResponseStateless response = assistant.messageStateless(messageOptions)
 .execute()
 .getResult();

 // If an intent was detected, print it to the console.
 List<RuntimeIntent> responseIntents = response.getOutput().getIntents();
 if(responseIntents.size() > 0) {
 System.out.println("Detected intent: #" + responseIntents.get(0).intent());
 }

 // Print the output from dialog, if any. Assumes a single text response.
 List<RuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
 if(responseGeneric.size() > 0) {
 if(responseGeneric.get(0).responseType().equals("text")) {
 System.out.println(responseGeneric.get(0).text());
 }
 }

 // Prompt for next round of input.
 System.out.print(">> ");
 String inputText = System.console().readLine();
 input = new MessageInputStateless.Builder()
 .messageType("text")
 .text(inputText)
 .build();
 } while(!input.text().equals("quit"));
 }
}

```

This version of the application begins the same way as before: sending an empty message to the assistant to start the conversation.

The `processResponse()` function now displays any intent detected by the dialog skill, along with the output text. It then prompts for the next round of user input.

It then displays any intent detected by the dialog skill, along with the output text. It then 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 haven't yet implemented a natural-language way to end the conversation, so instead, the client app is just watching for the literal command `quit` to indicate that the program should exit.

But something still isn't right:

```

Welcome to the Watson Assistant example!
>> hello
Detected intent: #hello
Welcome to the Watson Assistant example!
>> goodbye
Detected intent: #goodbye
Welcome to the Watson Assistant example!
>> quit

```

The Watson Assistant service is detecting the correct `#hello` and `#goodbye` 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 we are using the stateless `message` method, which means that it is the responsibility of our client application to maintain state information for the conversation. Because we are not yet doing anything to maintain state, the Watson Assistant service sees every round of user input as the first turn of a new conversation, triggering the `conversation_start` node.

## Maintaining state

State information for your conversation is maintained using the `context`. The context is an object that can be passed back and forth between your application and the Watson Assistant service, storing information that can be preserved or updated as the conversation goes on. Because we are using the stateless `message` method, the assistant does not store the context, so it is the responsibility of our client application to maintain it from one turn of the conversation to the next.

The context includes a session ID for each conversation with a user, as well as a counter that is incremented with each turn of the conversation. The assistant updates the context and returns it with each response. But our previous version of the example did not preserve the context, so these updates were lost, and each round of input appeared to be the start of a new conversation. We can easily 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).

### Node

```
// Example 3: Preserves context to maintain state.

const prompt = require('prompt-sync')();
const AssistantV2 = require('ibm-watson/assistant/v2');
const { IamAuthenticator } = require('ibm-watson/auth');

// Create Assistant service object.
const assistant = new AssistantV2({
 version: '2020-09-24',
 authenticator: new IamAuthenticator({
 apikey: '{apikey}', // replace with API key
 }),
 url: '{url}', // replace with URL
});

const assistantId = '{assistant_id}'; // replace with assistant ID

// Start conversation with empty message
messageInput = {
 messageType: 'text',
 text: '',
};
context = {};
sendMessage(messageInput, context);

// Send message to assistant.
function sendMessage(messageInput, context) {
 assistant
 .messageStateless({
 assistantId,
 input: messageInput,
 context: context,
 })
 .then(res => {
 processResponse(res.result);
 })
 .catch(err => {
 console.log(err); // something went wrong
 });
}

// Process the response.
function processResponse(response) {

 let context = response.context;

 // If an intent was detected, log it out to the console.
 if (response.output.intents.length > 0) {
 console.log('Detected intent: #' + response.output.intents[0].intent);
 }

 // Display the output from assistant, if any. Supports only a single
```

```

// text response.
if (response.output.generic) {
 if (response.output.generic.length > 0) {
 if (response.output.generic[0].response_type === 'text') {
 console.log(response.output.generic[0].text);
 }
 }
}

// If we're not done, prompt for the next round of input.
const newMessageFromUser = prompt('>>');
if (newMessageFromUser !== 'quit') {
 newMessageInput = {
 messageType: 'text',
 text: newMessageFromUser,
 }
 sendMessage(newMessageInput, context);
}
}

```

## Python

```

Example 3: Preserves context to maintain state.

from ibm_watson import AssistantV2
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator

Create Assistant service object.
authenticator = IAMAuthenticator('{apikey}') # replace with API key
assistant = AssistantV2(
 version = '2020-09-24',
 authenticator = authenticator
)
assistant.set_service_url('{url}')
assistant_id = '{assistant_id}' # replace with assistant ID

Initialize with empty message to start the conversation.
message_input = {
 'message_type': 'text',
 'text': ''
}
context = {}

Main input/output loop
while message_input['text'] != 'quit':

 # Send message to assistant.
 response = assistant.message_stateless(
 assistant_id,
 input = message_input,
 context = context
).get_result()

 context = response['context']

 # If an intent was detected, print it to the console.
 if response['output']['intents']:
 print('Detected intent: #' + response['output']['intents'][0]['intent'])

 # Print the output from dialog, if any. Supports only a single
 # text response.
 if response['output']['generic']:
 if response['output']['generic'][0]['response_type'] == 'text':
 print(response['output']['generic'][0]['text'])

 # Prompt for next round of input.
 user_input = input('>> ')
 message_input = {
 'text': user_input
 }
}

```

## Java

```

/*
 * Example 3: Preserves context to maintain state.
 */
import com.ibm.cloud.sdk.core.security.Authenticator;

```

```

import com.ibm.cloud.sdk.core.security.IamAuthenticator;
import com.ibm.watson.assistant.v2.Assistant;
import com.ibm.watson.assistant.v2.model.MessageContextStateless;
import com.ibm.watson.assistant.v2.model.MessageInputStateless;
import com.ibm.watson.assistant.v2.model.MessageResponseStateless;
import com.ibm.watson.assistant.v2.model.MessageStatelessOptions;
import com.ibm.watson.assistant.v2.model.RuntimeIntent;
import com.ibm.watson.assistant.v2.model.RuntimeResponseGeneric;
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();

 // Create Assistant service object.
 Authenticator authenticator = new IamAuthenticator("{apikey}"); // replace with API key
 Assistant assistant = new Assistant("2020-09-24", authenticator);
 assistant.setServiceUrl("{url}");
 String assistantId = "{assistant_id}"; // replace with assistant ID

 // Initialize with empty message to start the conversation.
 MessageInputStateless input = new MessageInputStateless.Builder()
 .messageType("text")
 .text("")
 .build();
 MessageContextStateless context = new MessageContextStateless.Builder()
 .build();

 // Main input/output loop
 do {
 // Send message to assistant.
 MessageStatelessOptions messageOptions = new MessageStatelessOptions.Builder(assistantId)
 .input(input)
 .context(context)
 .build();
 MessageResponseStateless response = assistant.messageStateless(messageOptions)
 .execute()
 .getResult();

 context = response.getContext();

 // If an intent was detected, print it to the console.
 List<RuntimeIntent> responseIntents = response.getOutput().getIntents();
 if(responseIntents.size() > 0) {
 System.out.println("Detected intent: #" + responseIntents.get(0).intent());
 }

 // Print the output from dialog, if any. Assumes a single text response.
 List<RuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
 if(responseGeneric.size() > 0) {
 if(responseGeneric.get(0).responseType().equals("text")) {
 System.out.println(responseGeneric.get(0).text());
 }
 }

 // Prompt for next round of input.
 System.out.print(">>");
 String inputText = System.console().readLine();
 input = new MessageInputStateless.Builder()
 .messageType("text")
 .text(inputText)
 .build();
 } while(!input.text().equals("quit"));
 }
}

```

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:

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:

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 message options along with the next round of user input:

## Node

```
assistant
 .messageStateless({
 assistantId,
 input: messageInput,
 context: context,
 })
```

## Python

```
response = assistant.message_stateless(
 assistant_id,
 input = message_input,
 context = context
).get_result()
```

## Java

```
MessageStatelessOptions messageOptions = new MessageStatelessOptions.Builder(assistantId)
 .input(input)
 .context(context)
 .build();
```

This ensures that the context is maintained from one turn to the next, so the Watson Assistant service no longer thinks every turn is the first:

```
Welcome to the Watson Assistant example!
>> hello
Detected intent: #hello
Good day to you.
>> goodbye
Detected intent: #goodbye
OK! See you later.
>> quit
```

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, although the service is recognizing the #goodbye intent and even responding appropriately, the conversation doesn't actually end. That's because we have not yet implemented the client actions that need to be carried out when the assistant requests them.

## Implementing client actions

In addition to the output text to be displayed to the user, our Watson Assistant dialog skill uses the `actions` array in the response JSON to signal when the application needs to carry out an action, based on the detected intents. When the dialog determines that the client application needs to do something, it returns an action object with a `type` of `client`. The `name` of the action indicates the specific action the client should carry out, and additional properties of the action can specify parameters, credentials, or other information needed for completing the action.

A client action might be a function the app needs to perform locally (such as accessing a database), or it might be a function the app calls from another service. By using client actions, you can build a client app that not only handles user interaction, but also orchestrates the integration between your assistant and external services.



**Note:** For more information about how client actions are requested from a dialog node, see [Requesting client actions](#).

The skill we are using for our example recognizes can request two different actions that must be completed by the client app:

- If the user indicates that the conversation is over (the #goodbye intent), the dialog requests a client action called `end_conversation`, which signals that the app should exit.
- If the user asks for a weather forecast (the #weather intent), the dialog requests the `get_weather` action. This signals that the app should request the client retrieve the weather forecast and then send the retrieved information back to the assistant so it can be included in a response to the user.

This version of our example client app adds support for these two actions. Our dialog will never request more than one action at a time, so our client only needs to check for the existence of a single `client` action in the `actions` array. (The `actions` array supports up to 5 actions of all types.) If a `client` action is present, the app carries out the specified action. (This version also removes the display of detected intents, now that we're sure those are being correctly identified.)

To keep things simple, we're simulating the call to an external weather service with a simple `get_weather_forecast` `getWeatherForecast` `getWeatherForecast` stub function. This function only knows how to predict the weather in Boston, and it assumes that it's always sunny.

## Node

```
// Example 4: Implements app actions.
```

```

const prompt = require('prompt-sync')();
const AssistantV2 = require('ibm-watson/assistant/v2');
const { IamAuthenticator } = require('ibm-watson/auth');

// Create Assistant service object.
const assistant = new AssistantV2({
 version: '2020-09-24',
 authenticator: new IamAuthenticator({
 apikey: '{apikey}', // replace with API key
 }),
 url: '{url}', // replace with URL
});

const assistantId = '{assistant_id}'; // replace with assistant ID

// Start conversation with empty input text
messageInput = {
 messageType: 'text',
 text: '',
};

context = {
 'skills': {
 'main skill': {
 'user_defined': {
 'user_location': 'Boston'
 }
 }
 }
};
sendMessage(messageInput, context);

// Send message to assistant.
function sendMessage(messageInput, context) {
 assistant
 .messageStateless({
 assistantId,
 input: messageInput,
 context: context,
 })
 .then(res => {
 processResponse(res.result);
 })
 .catch(err => {
 console.log(err); // something went wrong
 });
}

// Process the response.
function processResponse(response) {

 let endConversation = false;
 let skipUserInput = false;
 let newMessageFromUser = "";
 let context = response.context;

 // Check for client actions requested by the assistant.
 if (response.output.actions) {
 if (response.output.actions[0].type === 'client') {
 if (response.output.actions[0].name === 'get_weather') {
 // User asked for the weather forecast.
 let forecastLocation = response.output.actions[0].parameters['forecast_location'];
 let resultVar = response.output.actions[0].result_variable;
 let weatherForecast = getWeatherForecast(forecastLocation);
 context['skills']['main skill']['user_defined'][resultVar] = weatherForecast;
 skipUserInput = true;
 } else if (response.output.actions[0].name === 'end_conversation') {
 // User said goodbye, so we're done.
 endConversation = true;
 }
 }
 }
 //else{
 // Display the output from assistant, if any. Assumes a single text response.
 if (response.output.generic) {
 if (response.output.generic.length > 0) {
 if (response.output.generic[0].response_type === 'text') {
 console.log(response.output.generic[0].text);
 }
 }
 }
 //}
}

```

```

 }

// If we're not done, send the next round of input.
if (!endConversation) {
 if (!skipUserInput) {
 newMessageFromUser = prompt('>>');
 }
 newMessageInput = {
 messageType: 'text',
 text: newMessageFromUser,
 },
 sendMessage(newMessageInput, response.context);
} else {
 return;
}
}

// Function to simulate a weather service that only knows about Boston
function getWeatherForecast(forecastLocation) {
 let weatherForecast = "";
 if (forecastLocation === 'Boston') {
 weatherForecast = 'sunny'
 }
 return weatherForecast;
}

```

## Python

```

Example 4: Implements app actions.

from ibm_watson import AssistantV2
from ibm_cloud_sdk_core.authenticators import IAMAuthenticator
import json

Create Assistant service object.
authenticator = IAMAuthenticator('{apikey}') # replace with API key
assistant = AssistantV2(
 version = '2020-09-24',
 authenticator = authenticator
)
assistant.set_service_url('{url}')
assistant_id = '{assistant_id}' # replace with assistant ID

Initialize with empty input text to start the conversation.
message_input = {
 'message_type': 'text',
 'text': ""
}
context = {
 "skills": {
 "main skill": {
 "user_defined": {
 "user_location": "Boston"
 }
 }
 }
}
current_action = ""

Function to simulate a weather service that only knows about Boston
def get_weather_forecast(forecast_location):
 weather_forecast = ""
 if forecast_location == 'Boston':
 weather_forecast = 'sunny'
 return weather_forecast

Main input/output loop
while current_action != 'end_conversation':
 # Clear any action flag set by the previous response.
 current_action = ""
 skip_user_input = False

 # Send message to assistant.

 response = assistant.message_stateless(
 assistant_id,
 input = message_input,
 context = context

```

```

).get_result()

context = response['context']

Print the output from dialog, if any. Supports only a single
text response.
if response['output']['generic']:
 if response['output']['generic'][0]['response_type'] == 'text':
 print(response['output']['generic'][0]['text'])

Check for client actions requested by the assistant.
if 'actions' in response['output']:
 if response['output']['actions'][0]['type'] == 'client':
 current_action = response['output']['actions'][0]['name']

User asked for the weather forecast.
if current_action == 'get_weather':
 forecast_location = response['output']['actions'][0]['parameters']['forecast_location']
 result_var = response['output']['actions'][0]['result_variable']
 weather_forecast = get_weather_forecast(forecast_location)
 context['skills']['main skill']['user_defined'][result_var] = weather_forecast
 skip_user_input = True

If we're not done, send the next round of input.
if current_action != 'end_conversation' and not skip_user_input:
 user_input = input('>> ')
 message_input['text'] = user_input

```

## Java

```

/*
 * Example 4: implements app actions.
 */

import com.ibm.cloud.sdk.core.security.Authenticator;
import com.ibm.cloud.sdk.core.security.IamAuthenticator;
import com.ibm.watson.assistant.v2.Assistant;
import com.ibm.watson.assistant.v2.model.DialogNodeAction;
import com.ibm.watson.assistant.v2.model.MessageContextSkill;
import com.ibm.watson.assistant.v2.model.MessageContextSkills;
import com.ibm.watson.assistant.v2.model.MessageContextStateless;
import com.ibm.watson.assistant.v2.model.MessageInputStateless;
import com.ibm.watson.assistant.v2.model.MessageResponseStateless;
import com.ibm.watson.assistant.v2.model.MessageStatelessOptions;
import com.ibm.watson.assistant.v2.model.RuntimeResponseGeneric;
import java.util.HashMap;
import java.util.List;
import java.util.logging.LogManager;
import java.util.Map;

public class AssistantSimpleExample {
 public static void main(String[] args) {

 // Suppress log messages in stdout.
 LogManager.getLogManager().reset();

 // Create Assistant service object.
 Authenticator authenticator = new IamAuthenticator("{apikey}"); // replace with API key
 Assistant assistant = new Assistant("2020-09-24", authenticator);
 assistant.setServiceUrl("{url}");
 String assistantId = "{assistant_id}"; // replace with assistant ID

 // Initialize with empty input text to start the conversation.
 MessageInputStateless input = new MessageInputStateless.Builder()
 .messageType("text")
 .text("")
 .build();
 String currentAction;
 Boolean skipUserInput;
 Map<String, Object> userDefinedContext = new HashMap<>();
 userDefinedContext.put("user_location", "Boston");
 Map<String, Object> systemContext = new HashMap<>();
 MessageContextSkill mainSkillContext = new MessageContextSkill.Builder()
 .userDefined(userDefinedContext)
 .system(systemContext)
 .build();
 MessageContextSkills skillsContext = new MessageContextSkills();
 skillsContext.put("main skill", mainSkillContext);
 MessageContextStateless context = new MessageContextStateless.Builder()

```

```

.skills(skillsContext)
.build();

// Main input/output loop
do {
 // Clear any action flag set by the previous response.
 currentAction = "";
 skipUserInput = false;

 // Send message to assistant.
 MessageStatelessOptions messageOptions = new MessageStatelessOptions.Builder(assistantId)
 .input(input)
 .context(context)
 .build();
 MessageResponseStateless response = assistant.messageStateless(messageOptions)
 .execute()
 .getResult();

 context = response.getContext();

 // Print the output from dialog, if any. Assumes a single text response.
 List<RuntimeResponseGeneric> responseGeneric = response.getOutput().getGeneric();
 if(responseGeneric.size() > 0) {
 if(responseGeneric.get(0).responseType().equals("text")) {
 System.out.println(responseGeneric.get(0).text());
 }
 }

 // Check for any actions requested by the assistant.
 List<DialogNodeAction> responseActions = response.getOutput().getActions();
 if(responseActions != null) {
 if(responseActions.get(0).getType().equals("client")) {
 currentAction = responseActions.get(0).getName();
 }
 }

 // User asked for the weather forecast.
 if(currentAction.equals("get_weather")) {
 String forecastLocation = responseActions.get(0).getParameters().get("forecast_location").toString();
 String resultVariable = responseActions.get(0).getResultVariable();
 String weatherForecast = getWeatherForecast(forecastLocation);
 context.skills()
 .get("main skill")
 .userDefined()
 .put(resultVariable, weatherForecast);
 skipUserInput = true;
 }

 // If we're not done, prompt for next round of input.
 if(!currentAction.equals("end_conversation") && !skipUserInput) {
 System.out.print(">>");
 String inputText = System.console().readLine();
 input = new MessageInputStateless.Builder()
 .messageType("text")
 .text(inputText)
 .build();
 }

} while(!currentAction.equals("end_conversation"));

// Function to simulate a weather service that only knows about Boston
public static String getWeatherForecast(String forecastLocation) {
 String weatherForecast = "";
 if(forecastLocation.equals("Boston")) {
 weatherForecast = "sunny";
 }
 return weatherForecast;
}

```

When it receives a response, the app now checks the `actions` array to see if an action with `type = client` is present. If so, it checks the `name` value of the action and carries out the appropriate action.

If the requested action is `end_conversation`, the app uses this as an indicator that the conversation is over. The main input/output loop can now check for this (instead of the hardcoded `quit` command we used before) and exit, instead of prompting for another round of input.

The `get_weather` action is more interesting. In this case, we need some more information from the client action request:

\$ - The location parameter to pass to the weather service, which is included in the `parameters` array. (For our example, the dialog is taking the location we specified in the `location` context variable, but of course a real application would use some more intelligent mechanism to determine the location to use.)

- The name of the context variable where the result should be stored, specified by the `result\_variable` property.

After we retrieve these values from the client action request, we can call the weather service and store the result in the context variable so it will be sent to the service with the next message.

#### Node

```
let forecastLocation = response.output.actions[0].parameters['forecast_location'];
let resultVar = response.output.actions[0].result_variable;
let weatherForecast = getWeatherForecast(forecastLocation);
context['skills']['main skill']['user_defined'][resultVar] = weatherForecast;
skipUserInput = true;
```

#### Python

```
forecast_location = response['output']['actions'][0]['parameters']['forecast_location']
result_var = response['output']['actions'][0]['result_variable']
weather_forecast = get_weather_forecast(forecast_location)
context['skills']['main skill']['user_defined'][result_var] = weather_forecast
skip_user_input = True
```

#### Java

```
String forecastLocation = responseActions.get(0).getParameters().get("forecast_location").toString();
String resultVariable = responseActions.get(0).getResultVariable();
String weatherForecast = getWeatherForecast(forecastLocation);
context.skills()
 .get("main skill")
 .userDefined()
 .put(resultVariable, weatherForecast);
skipUserInput = true;
```

We also set a `skip_user_input` flag, which tells the app not to prompt the user for input before sending the next message. We do this because, from the dialog's point of view, the result of the client action effectively *is* the "user input" for this turn of the conversation; the child node that receives the next message only needs the data we stored in the context.

```
Welcome to the Watson Assistant example!
>> hello
Good day to you.
>> what's the weather today?
Checking the weather...
It will be sunny in Boston today.
>> 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.

The dialog in our assistant also includes another child node that handles the case where the weather service fails, and no forecast information is stored in the context. Since our simulated weather service only knows about Boston, we can easily test this case by changing our hardcoded `user_location` context variable to a different city:

#### Node

```
context = {
 'skills': {
 'main skill': {
 'user_defined': {
 'user_location': 'Tokyo'
 }
 }
 }
};
```

#### Python

```
context = {
```

```
"skills": {
 "main skill": {
 "user_defined": {
 "user_location": "Tokyo"
 }
 }
}
```

## Java

```
userDefinedContext.put("user_location", "Tokyo");
```

Now the weather service fails to return any forecast, so a different dialog node handles the response:

```
Welcome to the Watson Assistant example!
>> what's the weather today?
Checking the weather...
Oops! I couldn't get the weather forecast. Maybe try again later?
>> goodbye
OK! See you later.
```

This simple example illustrates how a client app serves both as the channel for users communicating with the assistant, as well as a point of integration with other services. Of course, a real-world application would use a more sophisticated user interface, such as a web interface; and it would implement more complex actions, possibly integrating with a customer database or other business systems. It would also need to send additional data to the assistant, such as a user ID to identify each unique user. 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](#).

## Using the v1 runtime API

Using the v2 API is the recommended way to build a runtime client application that communicates with the Watson Assistant service. However, some older applications might still be using the v1 runtime API, which includes a similar method for sending messages to the workspace within a dialog skill. Note that if your app uses the v1 runtime API, it communicates directly with the workspace, bypassing the skill orchestration and state-management capabilities of the assistant.

For more information about the v1 `/message` method and context, see the [v1 API Reference](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Migrating to the v2 API](#).

## Migrating to the v2 API

The Assistant v2 runtime API, which supports the use of assistants and skills, was introduced in November 2018. This API offers significant advantages over the v1 runtime API, including automatic state management, ease of deployment, skill versioning, and the availability of new features such as the search skill.

The v2 API is available for all users, regardless of service plan, at no additional cost.



**Note:** The v2 API currently supports only runtime interaction with an existing assistant. Authoring applications that create or modify workspaces should continue to use the v1 API.

## Overview

With the v2 API, your client app communicates with an assistant, rather than directly with a workspace. An assistant is a new orchestration layer that offers several new capabilities, including automatic state management, skill versioning, easier deployment, and (for Plus and Premium plans) search skills. Your existing workspace (now referred to as a *dialog skill*) continues to function as before, but the new capabilities are provided by the new assistant layer.

All communication with an assistant takes place within the context of a `session`, which maintains conversation state throughout the duration of the conversation. State data, including any context variables that are defined by your dialog or client application, are automatically stored by Watson Assistant, without any action required on the part of your application.

State data persists until you explicitly delete the session, or until the session times out because of inactivity.



**Note:** If you prefer to manage state yourself, the v2 API also provides a stateless `message` method that functions more like the v1 API. If you use the stateless `message` method, you do not need to explicitly create or delete sessions, and your app is responsible for maintaining context. For more information about the stateless `message` method, see the [API Reference](#).

If you have an existing application that uses the v1 API to send user input directly to a workspace, migrating your app to use the v2 API is a straightforward

process.

## Set up an assistant

The v2 runtime API sends messages to an assistant, which routes the messages to your dialog skill (formerly workspace). To set up an assistant, use the Watson Assistant user interface:

1. Click the **Skills** tab. Verify that your workspace is shown as an available skill. (All existing workspaces for your service instance are automatically converted to skills in the Watson Assistant user interface. This conversion does not make any change to the underlying workspace.)
  2. Click the **Assistants** tab. Click **Create assistant** to create a new assistant. When prompted to add skills, click **Add dialog skill** and select the dialog skill that corresponds to your workspace.
- For more information about creating assistants, see [Creating an assistant](#).
3. After your new assistant is created, click the  menu and then select **Settings**.
  4. On the **Assistant Settings** page, find the assistant ID. Your application will use this ID (instead of a workspace ID) to communicate with the assistant. The service credentials are the same for both the v1 and v2 APIs.



**Note:** Currently, there is no API support for retrieving an assistant ID. To find the assistant ID, you must use the Watson Assistant user interface.

## Call the v2 runtime API

After you have created an assistant, you can update your client application to use the v2 runtime API instead of the v1 runtime API.

1. Before sending the first message in a conversation, use the v2 [Create a session](#) method to create a session. Save the returned session ID:

### Node

```
service.createSession({ assistant_id: assistantId, }).then(res => { sessionId = res.session_id; })
```

```
$ ````python {: codeblock python }
session_id = service.create_session(
 assistant_id = assistant_id
).get_result()['session_id']
```

```
$ ````java {: codeblock java}
```

```
CreateSessionOptions createSessionOptions = new CreateSessionOptions.Builder(assistantId).build(); SessionResponse session =
service.createSession(createSessionOptions).execute().getResult(); String sessionId = session.getSessionId();
```

```
$ 1. Use the v2 [**Send user input to assistant**](https:///{DomainName}/apidocs/assistant/assistant-v2#message) method to send user input to the assistant. Instead of specifying the workspace ID as you did with the v1 API, you specify the assistant ID and the session ID:
```

```
```javascript {: codeblock javascript}
service
.message({
  assistant_id: assistantId,
  session_id: sessionId,
  input: messageInput
})
```

```
$ ````python {: codeblock python}
```

```
response = service.message(assistant_id, session_id, input = message_input).get_result()
```

```
$ ````java {: codeblock java}
MessageInput input = new MessageInput.Builder().text(inputText).build();
MessageOptions messageOptions = new MessageOptions.Builder(assistantId, sessionId)
.input(input)
.build();
MessageResponse response = service.message(messageOptions)
.execute()
.getResult();
```

```
$ The basic message structure has not changed; in particular, the user input is still sent as `input.text`.
```

1. After a conversation ends, use the v2 [Delete session](#) method to delete the session.

Node

```
service.deleteSession({ assistant_id: assistantId, session_id: sessionId, })
```

```
$ ````python {: codeblock python}
service.delete_session(
    assistant_id = assistant_id,
    session_id = session_id
)
```

```
$ ````java {: codeblock java}
```

```
DeleteSessionOptions deleteSessionOptions = new DeleteSessionOptions.Builder(assistantId, sessionId).build();
service.deleteSession(deleteSessionOptions).execute();
```

\$ If you do not explicitly delete the session, it will be automatically deleted after the configured timeout interval. (The timeout duration depends on your plan; for more information, see [Session limits](/docs/assistant?topic=assistant-assistant-settings#assistant-settings-session-limits).)

To see examples of the v2 APIs in the context of a simple client application, see [Building a client application](/docs/assistant?topic=assistant-api-client).

Handle the v2 response format

Your application might need to be updated to handle the v2 runtime response format, depending on which parts of the response your application needs to access:

- Output for all response types (such as `text` and `option`) are still returned in the `output.generic` object. Application code for handling these responses should work without modification.
- Detected intents and entities are now returned as part of the `output` object, rather than at the root of the response JSON.
- The conversation context is now organized into two objects:
 - The **global context** contains system-level context data shared by all skills used by the assistant.
 - The **skill context** contains any user-defined context variables used by your dialog skill.

However, keep in mind that state data, including conversation context, is now maintained by the assistant, so your application might not need to access the context at all (see [Let the assistant maintain state]({#api-migration-state})).

Refer to the v2 [API Reference]([{: external}](https://{{DomainName}}/apidocs/assistant/assistant-v2#message)) for complete documentation of the v2 response format.

Let the assistant maintain state {: #api-migration-state}

For most applications, you can now remove any code included for the purpose of maintaining state. It is no longer necessary to save the context and send it back to Watson Assistant with each turn of the conversation. The context is automatically maintained by {{site.data.keassistant_classic_shortnshort}} and can be accessed by your dialog as before.

Note that with the v2 API, the context is by default not included in responses to the client application. However, your code can still access context variables if necessary:

- You can still send a `context` object as part of the message input. Any context variables you include are stored as part of the context maintained by Watson Assistant. (If the context variable you send already exists in the context, the new value overwrites the previously stored value.)

Make sure the context object you send conforms to the v2 format. All user-defined context variables sent by your application should be part of the skill context; typically, the only global context variable you might need to set is `system.user_id`, which is used by Plus and Premium plans for billing purposes.

- You can still retrieve context variables from either the global or skill context. To have the `context` object included with message responses, use the **return_context** property in the message input options. For more information, see [Accessing context data](/docs/assistant?topic=assistant-api-client-get-context).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Accessing context data in dialog](#).

Accessing context data

The `context` is an object containing variables that persist throughout a conversation and can be shared by the dialog and the client application. Both the dialog and the client application can read and write context variables.

You can choose whether you want the context to be maintained by your application or by the Watson Assistant service:

- If you use the stateful v2 `message` API, the context is automatically maintained by the assistant on a per-session basis. Your application must explicitly create a session at the beginning of each conversation; the context is stored by the service as part of the session and is not returned in message responses unless you request it. For more information, see the [v2 API Reference](#).
- If you use the stateless v2 `message` API (or the legacy v1 `message` API) your application is responsible for storing the context after each conversation turn and sending it back to the service with the next message. For a complex application, or an application that needs to store personally identifiable information, you might choose to store the context in a database.

A session ID is automatically generated at the beginning of the conversation, but no session data is stored by the service. With the stateless `message` API, the context is always included with each message response. For more information, see the [v2 API Reference](#).

Important: One use of the context is to specify a unique user ID for each end user who interacts with the assistant. For user-based plans, this ID is used for billing purposes. (For more information, see [User-based plans](#).)

There are two types of context:

- **Global context:** context variables that are shared by all skills used by an assistant, including internal system variables used to manage conversation flow. The global context includes the user ID, as well as other global values such as the time zone and language of the assistant.
- **Skill-specific context:** context variables specific to a particular skill, including any user-defined variables needed by your application. Currently, only one skill (named `main skill`) is supported.

User-defined context variables that you specify in a dialog node are part of the `user_defined` object within the skill context when accessed using the API. Note that this structure differs from the `context` structure that appears in the JSON editor in the Watson Assistant user interface. For example, you might specify the following in the JSON editor:

```
$ "context": {  
    "my_context_var": "this is the value"  
}
```

In the v2 API, you would access this user-defined variable as follows:

```
$ "context": {  
    "skills": {  
        "main skill": {  
            "user_defined": {  
                "my_context_var": "this is the value"  
            }  
        }  
    }  
}
```

For detailed information about how to access context variables using the API, see the [v2 API Reference](#).

Example

The following example shows a stateful `/message` request that includes both global and skill-specific context variables; it also uses the `options.return_context` property to request that the context be returned with the response. Note that this option is applicable only if you are using the stateful `message` method, because the stateless `message` method always returns the context.

Node

```
service  
.message({  
    assistant_id: '{assistant_id}',  
    session_id: '{session_id}',  
    input: {  
        message_type: 'text',  
        text: 'Hello',  
        options: {  
            'return_context': true  
        }  
    },  
    context: {  
        'global': {  
            'system': {  
                'user_id': 'my_user_id'  
            }  
        }  
    }  
},
```

```

'skills': {
    'main skill': {
        'user_defined': {
            'account_number': '123456'
        }
    }
}
.then(res => {
    console.log(JSON.stringify(res, null, 2));
})
.catch(err => {
    console.log(err);
});

```

Python

```

response=service.message(
    assistant_id='{assistant_id}',
    session_id='{session_id}',
    input={
        'message_type': 'text',
        'text': 'Hello',
        'options': {
            'return_context': True
        }
    },
    context={
        'global': {
            'system': {
                'user_id': 'my_user_id'
            }
        },
        'skills': {
            'main skill': {
                'user_defined': {
                    'account_number': '123456'
                }
            }
        }
    }
).get_result()

print(json.dumps(response, indent=2))

```

Java

```

MessageInputOptions inputOptions = new MessageInputOptions.Builder()
    .returnContext(true)
    .build();

MessageInput input = new MessageInput.Builder()
    .messageType("text")
    .text("Hello")
    .options(inputOptions)
    .build();

// create global context with user ID
MessageContextGlobalSystem system = new MessageContextGlobalSystem.Builder()
    .userId("my_user_id")
    .build();
MessageContextGlobal globalContext = new MessageContextGlobal.Builder()
    .system(system)
    .build();

// build user-defined context variables, put in skill-specific context for main skill
Map<String, Object> userDefinedContext = new HashMap<>();
userDefinedContext.put("account_number", "123456");
MessageContextSkill mainSkillContext = new MessageContextSkill.Builder()
    .userDefined(userDefinedContext)
    .build();
Map<String, MessageContextSkill> skillsContext = new HashMap<>();
skillsContext.put("main skill", mainSkillContext);

MessageContext context = new MessageContext.Builder()
    .global(globalContext)
    .skills(skillsContext)

```

```

.build();

MessageOptions options = new MessageOptions.Builder()
  .assistantId("{assistant_id}")
  .sessionId("{session_id}")
  .input(input)
  .context(context)
  .build();

MessageResponse response = service.message(options).execute().getResult();

System.out.println(response);

```

In this example request, the application specifies a value for `user_id` as part of the global context. In addition, it sets one user-defined context variable (`account_number`) as part of the skill-specific context. This context variable can be accessed by dialog nodes as `$account_number`. (For more information about using the context in your dialog, see [How the dialog is processed](#).)

You can specify any variable name you want to use for a user-defined context variable. If the specified variable already exists, it is overwritten with the new value; if not, a new variable is added to the context.

The output from this request includes not only the usual output, but also the context, showing that the specified values have been added. If you are using the stateless `message` method, this context data must be stored locally and sent back to the Watson Assistant service as part of the next message. If you are using the stateful `message` method, this context is stored automatically and will persist for the life of the session.

```

$ {
  "output": {
    "generic": [
      {
        "response_type": "text",
        "text": "Welcome to the Watson Assistant example!"
      }
    ],
    "intents": [
      {
        "intent": "hello",
        "confidence": 1
      }
    ],
    "entities": []
  },
  "user_id": "my_user_id",
  "context": {
    "global": {
      "system": {
        "turn_count": 1,
        "user_id": "my_user_id"
      }
    },
    "skills": {
      "main skill": {
        "user_defined": {
          "account_number": "123456"
        }
      }
    }
  }
}

```

Restoring conversation state

In some situations, you might want the ability to restore a conversation to a previous state.

You can use the `export` option on stateful `message` requests to specify that you want the `context` object in the response to include complete session state data. If you specify `true` for this option, the returned skill context includes an encoded `state` property that represents the current conversation state.

If you are using the stateful `message` API, the service stores conversation state data only for the life of the session. However, if you save this context data (including `state`) and send it back to the service with a subsequent message request, you can restore the conversation to the same state, even if the original session has expired or has been deleted.

If you are using the stateless `message` API, the `state` property is always included in responses (along with the rest of `context`). Although stateless sessions do not expire, you can still use this state data to reset a conversation to a previous state.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Implementing dialog responses in a client application](#).

Implementing responses

A dialog node can respond to users with a response that includes text, images, or interactive elements such as clickable options. If you are building your own client application, you must implement the correct display of all response types returned by your dialog. (For more information about dialog responses, see [Responses](#)).

Response output format

By default, responses from a dialog node are specified in the `output.generic` object in the response JSON returned from the `/message` API. The `generic` object contains an array of up to 5 response elements that are intended for any channel. The following JSON example shows a response that includes text and an image:

```
$ {
  "output": {
    "generic": [
      {
        "response_type": "text",
        "text": "OK, here's a picture of a dog."
      },
      {
        "response_type": "image",
        "source": "http://example.com/dog.jpg"
      }
    ],
    "text" : ["OK, here's a picture of a dog."]
  },
  "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"
}
```



Note: As this example shows, the text response (`OK, here's a picture of a dog.`) is also returned in the `output.text` array. This is included for backward compatibility for older applications that do not support the `output.generic` format.

It is the responsibility of your client application to handle all response types appropriately. In this case, your application would need to display the specified text and image to the user.

Response types

Each element of a response is of one of the supported response types (currently `image`, `option`, `pause`, `text`, `suggestion`, and `user_defined`). Each response type is specified using a different set of JSON properties, so the properties included for each response will vary depending upon response type. For complete information about the response model of the `/message` API, see the [API Reference](#).)

This section describes the available response types and how they are represented in the `/message` API response JSON. (If you are using the Watson SDK, you can use the interfaces provided for your language to access the same objects.)



Note: The examples in this section show the format of the JSON data returned from the `/message API` at run time, and is different from the JSON format used to define responses within a dialog node. You can use the format in the examples to update `output.generic` with [webhooks](#). To update `output.generic` using the JSON editor, see [Defining responses using the JSON editor](#)).

Text

The `text` response type is used for ordinary text responses from the dialog:

```
$ {
  "output": {
    "generic": [
      {
        "response_type": "text",
        "text": "OK, you want to fly to Boston next Monday."
      }
    ],
    "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"
  }
}
```

Note that for backward compatibility, the same text is also included in the `output.text` array in the dialog response (as shown in the example at the beginning of this page).

Image

The `image` response type instructs the client application to display an image, optionally accompanied by a title and description:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "response_type": "image",  
        "source": "http://example.com/image.jpg",  
        "title": "Image example",  
        "description": "This is an example image"  
      }  
    ]  
  },  
  "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"  
}
```

Your application is responsible for retrieving the image specified by the `source` property and displaying it to the user. If the optional `title` and `description` are provided, your application can display them in whatever way is appropriate (for example, rendering the title after the image and the description as hover text).

Video

The `video` response type instructs the client application to display a video, optionally accompanied by a `title`, `description` and `alt_text` for accessibility:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "response_type": "video",  
        "source": "http://example.com/video.mp4",  
        "title": "Video example",  
        "description": "This is an example video",  
        "alt_text": "A video showing a great example",  
        "channel_options": {  
          "chat": {  
            "dimensions": {  
              "base_height": 180  
            }  
          }  
        }  
      }  
    ]  
  },  
  "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"  
}
```

Your application is responsible for retrieving the video specified by the `source` property and displaying it to the user. If the optional `title` and `description` are provided, your application can display them in whatever way is appropriate.

The optional `channel_options.chat.dimensions.base_height` property takes a number signifying the amount of pixels the video should be rendered at a width of 360 pixels. Your app should use this value to maintain the proper aspect ratio of the video if it is rendered in a nonstandard size.

Audio

The `audio` response type instructs the client application to play an audio file:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "response_type": "audio",  
        "source": "http://example.com/audio.mp3",  
        "channel_options": {  
          "voice_telephony": {  
            "loop": true  
          }  
        }  
      }  
    ]  
  },  
  "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"  
}
```

Your application is responsible for playing the audio file.

The optional `channel_options.voice_telephony.loop` property takes a boolean signifying if the audio file should be played as a continuous loop. (This option is typically used for hold music that might need to continue for an undefined time.)

iframe

The `iframe` response type instructs the client application to display content in an embedded `iframe` element, optionally accompanied by a title:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "response_type": "iframe",  
        "source": "http://example.com/iframe.html",  
        "title": "My IFrame"  
      }  
    ],  
    "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"  
  }  
}
```

Your application is responsible for displaying the `iframe` content. Content in an embedded `iframe` is useful for displaying third-party content, or for content from your own site that you do not want to reauthor using the [user_defined](#) response type.

Pause

The `pause` response type instructs the application to wait for a specified interval before displaying the next response:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "response_type": "pause",  
        "time": 500,  
        "typing": false  
      }  
    ],  
    "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"  
  }  
}
```

This pause might be requested by the dialog to allow time for a request to complete, or simply to mimic the appearance of a human agent who might pause between responses. The pause can be of any duration up to 10 seconds.

A `pause` response is typically sent in combination with other responses. Your application should pause for the interval specified by the `time` property (in milliseconds) before displaying the next response in the array. The optional `typing` property requests that the client application show a "user is typing" indicator, if supported, in order to simulate a human agent.

Option

The `option` response type instructs the client application to display a user interface control that enables the user to select from a list of options, and then send input back to the assistant based on the selected option:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "response_type": "option",  
        "title": "Available options",  
        "description": "Please select one of the following options:",  
        "preference": "button",  
        "options": [  
          {  
            "label": "Option 1",  
            "value": {  
              "input": {  
                "text": "option 1"  
              }  
            }  
          },  
          {  
            "label": "Option 2",  
            "value": {  
              "input": {  
                "text": "option 2"  
              }  
            }  
          }  
        ]  
      }  
    ]  
  }  
}
```

```

        "value": {
          "input": {
            "text": "option 2"
          }
        }
      ]
    ],
  },
  "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"
}

```

Your app can display the specified options using any suitable user-interface control (for example, a set of buttons or a drop-down list). The optional **preference** property indicates the preferred type of control your app should use (**button** or **dropdown**), if supported. For the best user experience, a good practice is to present three or fewer options as buttons, and more than three options as a drop-down list.

For each option, the **label** property specifies the label text that should appear for the option in the UI control. The **value** property specifies the input that should be sent back to the assistant (using the [/message](#) API) when the user selects the corresponding option.

For an example of implementing **option** responses in a simple client application, see [Example: Implementing option responses](#).

Suggestion

 **Tip:** This feature is available only to users with a paid plan.

The **suggestion** response type is used by the disambiguation feature to suggest possible matches when it isn't clear what the user wants to do. A **suggestion** response includes an array of **suggestions**, each one corresponding to a possible matching dialog node:

```

$ {
  "output": {
    "generic": [
      {
        "response_type": "suggestion",
        "title": "Did you mean:",
        "suggestions": [
          {
            "label": "I'd like to order a drink.",
            "value": {
              "intents": [
                {
                  "intent": "order_drink",
                  "confidence": 0.7330395221710206
                }
              ],
              "entities": [],
              "input": {
                "suggestion_id": "576aba3c-85b9-411a-8032-28af2ba95b13",
                "text": "I want to place an order"
              }
            },
            "output": {
              "text": [
                "I'll get you a drink."
              ],
              "generic": [
                {
                  "response_type": "text",
                  "text": "I'll get you a drink."
                }
              ],
              "nodes_visited_details": [
                {
                  "dialog_node": "node_1_1547675028546",
                  "title": "order drink",
                  "user_label": "I'd like to order a drink.",
                  "conditions": "#order_drink"
                }
              ]
            },
            "source_dialog_node": "root"
          },
          {
            "label": "I need a drink refill.",
            "value": {

```

```

"intents": [
    {
        "intent": "refill_drink",
        "confidence": 0.2529746770858765
    }
],
"entities": [],
"input": {
    "suggestion_id": "6583b547-53ff-4e7b-97c6-4d062270abcd",
    "text": "I need a drink refill"
},
"output": {
    "text": [
        "I'll get you a refill."
    ],
    "generic": [
        {
            "response_type": "text",
            "text": "I'll get you a refill."
        }
    ],
    "nodes_visited_details": [
        {
            "dialog_node": "node_2_1547675097178",
            "title": "refill drink",
            "user_label": "I need a drink refill.",
            "conditions": "#refill_drink"
        }
    ]
},
"source_dialog_node": "root"
}
],
"user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"
}
}

```

Note that the structure of a `suggestion` response is very similar to the structure of an `option` response. As with options, each suggestion includes a `label` that can be displayed to the user and a `value` specifying the input that should be sent back to the assistant if the user chooses the corresponding suggestion. To implement `suggestion` responses in your application, you can use the same approach that you would use for `option` responses.

For more information about the disambiguation feature, see [Disambiguation](#).

Search

 **Tip:** This feature is available only to users with a paid plan.

The `search` response type is used by a search skill to return the results from a Watson Discovery search. A `search` response includes an array of `results`, each of which provides information about a match returned from the Discovery search query:

```

$ {
  "output": {
    "generic": [
      {
        "response_type": "search",
        "header": "I found the following information that might be helpful.",
        "results": [
          {
            "title": "About",
            "body": "IBM Watson Assistant is a cognitive bot that you can customize for your business needs, and deploy across multiple channels to bring help to your customers where and when they need it.",
            "url": "https://cloud.ibm.com/docs/assistant?topic=assistant-index",
            "id": "6682eca3c5b3778ccb730b799a8063f3",
            "result_metadata": {
              "confidence": 0.08401551980328191,
              "score": 0.73975396
            },
            "highlight": {
              "Shortdesc": [
                "IBM <em>Watson</em> <em>Assistant</em> is a cognitive bot that you can customize for your business needs, and deploy across multiple channels to bring help to your customers where and when they need it."
              ]
            }
          }
        ]
      }
    ]
  }
}

```

```

        ],
        "url": [
            "https://cloud.ibm.com/docs/<em>assistant</em>?topic=<em>assistant</em>-index"
        ],
        "body": [
            "IBM <em>Watson</em> <em>Assistant</em> is a cognitive bot that you can customize for your business needs, and deploy across multiple channels to bring help to your customers where and when they need it."
        ]
    }
}
]
},
"user_id": "58e1b04e-f4bb-469a-9e4c-dffe1d4ebf23"
}

```

For each search result, the `title`, `body`, and `url` properties include content returned from the Discovery query. The search skill configuration determines which fields in the Discovery collection are mapped to these fields in the response. Your application can use these fields to display the results to the user (for example, you might use the `body` text to show an abstract or description of the matching document, and the `url` value to create a link the user can click to open the document).

In addition, the `header` property provides a message to display to the user about the results of the search. In the case of a successful search, `header` provides introductory text to be displayed before the search results (for example, `I found the following information that might be helpful.`). Different message text indicates that the search did not return any results, or that the connection to the Discovery service failed. You can customize these messages in the search skill configuration.

For more information about search skills, see [Creating a search skill](#).

User-defined

A user-defined response type can contain up to 5000 KB of data to support a type of a response you have implemented in your client. For example, you might define a user-defined response type to display a special color-coded card, or to format data in a table or graphic.

The `user_defined` property of the response is an object that can contain any valid JSON data:

```

$ {
  "output": {
    "generic": [
      {
        "response_type": "user_defined",
        "user_defined": {
          "field_1": "String value",
          "array_1": [
            1,
            2
          ],
          "object_1": {
            "property_1": "Another string value"
          }
        }
      }
    ],
    "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"
  }
}

```

Your application can parse and display the data in any way you choose.

Example: Implementing option responses

To show how a client application might handle option responses, which prompt the user to select from a list of choices, we can extend the client example described in [Building a client application](#). This is a simplified client app that uses standard input and output to handle three intents (sending a greeting, showing the current time, and exiting from the app):

```

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.

```

 **Note:** If you want to try the example code shown in this topic, you should first set up the required workspace and obtain the API details you will need. For more information, see [Building a client application](#).

Receiving an option response

The `option` response can be used when you want to present the user with a finite list of choices, rather than interpreting natural language input. This can be used in any situation where you want to enable the user to quickly select from a set of unambiguous options.

In our simplified client app, we will use this capability to select from a list of the actions the assistant supports (greetings, displaying the time, and exiting). In addition to the three intents previously shown (`#hello`, `#time`, and `#goodbye`), the example workspace supports a fourth intent: `#menu`, which is matched when the user asks to see a list of available actions.

When the workspace recognizes the `#menu` intent, the dialog responds with an `option` response:

```
$ {  
  "output": {  
    "generic": [  
      {  
        "title": "What do you want to do?",  
        "options": [  
          {  
            "label": "Send greeting",  
            "value": {  
              "input": {  
                "text": "hello"  
              }  
            }  
          },  
          {  
            "label": "Display the local time",  
            "value": {  
              "input": {  
                "text": "time"  
              }  
            }  
          },  
          {  
            "label": "Exit",  
            "value": {  
              "input": {  
                "text": "goodbye"  
              }  
            }  
          }  
        ],  
        "response_type": "option"  
      }  
    ],  
    "intents": [  
      {  
        "intent": "menu",  
        "confidence": 0.6178638458251953  
      }  
    ],  
    "entities": []  
  },  
  "user_id": "faf4a112-f09f-4a95-a0be-43c496e6ac9a"  
}
```

The `option` response contains multiple options to be presented to the user. Each option includes two objects, `label` and `value`. The `label` is a user-facing string identifying the option; the `value` specifies the corresponding message input that should be sent back to the assistant if the user chooses the option.

Our client app will need to use the data in this response to build the output we show to the user, and to send the appropriate message to the assistant.

Listing available options

The first step in handling an option response is to display the options to the user, using the text specified by the `label` property of each option. You can display the options using any technique that your application supports, typically a drop-down list or a set of clickable buttons. (The optional `preference` property of an option response, if specified, indicates which type of display the application should display if possible.)

Our simplified example uses standard input and output, so we don't have access to a real UI. Instead we will just present the options as a numbered list.

```

// Option example 1: lists options.

const prompt = require('prompt-sync')();
const AssistantV2 = require('ibm-watson/assistant/v2');
const { IamAuthenticator } = require('ibm-watson/auth');

// Set up Assistant service wrapper.
const service = new AssistantV2({
  version: '2019-02-28',
  authenticator: new IamAuthenticator({
    apikey: '{apikey}', // replace with API key
  })
});

const assistantId = '{assistant_id}'; // replace with assistant ID
let sessionId;

// Create session.
service
  .createSession({
    assistantId,
  })
  .then(res => {
    sessionId = res.result.session_id;
    sendMessage({
      messageType: 'text',
      text: '', // start conversation with empty message
    });
  })
  .catch(err => {
    console.log(err); // something went wrong
  });

// Send message to assistant.
function sendMessage(messageInput) {
  service
    .message({
      assistantId,
      sessionId,
      input: messageInput,
    })
    .then(res => {
      processResponse(res.result);
    })
    .catch(err => {
      console.log(err); // something went wrong
    });
}

// Process the response.
function processResponse(response) {

  let endConversation = false;

  // Check for client actions requested by the assistant.
  if (response.output.actions) {
    if (response.output.actions[0].type === 'client') {
      if (response.output.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.output.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 assistant, if any. Supports only a single
    // response.
    if (response.output.generic) {
      if (response.output.generic.length > 0) {
        switch (response.output.generic[0].response_type) {
          case 'text':
            // It's a text response, so we just display it.
            console.log(response.output.generic[0].text);
            break;
          case 'option':
            // It's an option response, so we'll need to show the user
            // a list of choices.
        }
      }
    }
  }
}

```

```

        console.log(response.output.generic[0].title);
        const options = response.output.generic[0].options;
        // List the options by label.
        for (let i = 0; i < options.length; i++) {
            console.log((i+1).toString() + '.' + options[i].label);
        }
        break;
    }
}
}

// If we're not done, prompt for the next round of input.
if (!endConversation) {
    const newMessageFromUser = prompt('>>');
    newMessageInput = {
        messageType: 'text',
        text: newMessageFromUser,
    }
    sendMessage(newMessageInput);
} else {
    // We're done, so we delete the session.
    service
        .deleteSession({
            assistantId,
            sessionId,
        })
        .then(res => {
            return;
        })
        .catch(err => {
            console.log(err); // something went wrong
        });
}
}
}

```

Let's take a closer look at the code that outputs the response from the assistant. Now, instead of assuming a `text` response, the application supports both the `text` and `option` response types:

```

// Display the output from assistant, if any. Supports only a single
// response.
if (response.output.generic) {
    if (response.output.generic.length > 0) {
        switch (response.output.generic[0].response_type) {
            case 'text':
                // It's a text response, so we just display it.
                console.log(response.output.generic[0].text);
                break;
            case 'option':
                // It's an option response, so we'll need to show the user
                // a list of choices.
                console.log(response.output.generic[0].title);
                const options = response.output.generic[0].options;
                // List the options by label.
                for (let i = 0; i < options.length; i++) {
                    console.log((i+1).toString() + '.' + options[i].label);
                }
                break;
        }
    }
}

```

If `response_type = text`, we just display the output, as before. But if `response_type = option`, we have to do a bit more work. First we display the value of the `title` property, which serves as lead-in text to introduce the list of options; then we list the options, using the value of the `label` property to identify each one. (A real-world application would show these labels in a drop-down list or as the labels on clickable buttons.)

You can see the result by triggering the `#menu` intent:

```

Welcome to the Watson Assistant example!
>> what are the available actions?
What do you want to do?
1. Send greeting
2. Display the local time
3. Exit
>> 2
Sorry, I have no idea what you're talking about.
>>

```

As you can see, the application is now correctly handling the `option` response by listing the available choices. However, we aren't yet translating the user's choice into meaningful input.

Selecting an option

In addition to the `label`, each option in the response also includes a `value` object, which contains the input data that should be sent back to the assistant if the user chooses the corresponding option. The `value.input` object is equivalent to the `input` property of the `/message` API, which means that we can just send this object back to the assistant as-is.

To do this in our application, we will set a new `promptOption` flag when the client receives an `option` response. When this flag is true, we know that we want to take the next round of input from `value.input` rather than accepting natural language text input from the user. (Again, we don't have a real user interface, so we will just prompt the user to select a valid option from the list by number.)

```
// Option example 2: sends back selected option value.

const prompt = require('prompt-sync')();
const AssistantV2 = require('ibm-watson/assistant/v2');
const { IamAuthenticator } = require('ibm-watson/auth');

// Set up Assistant service wrapper.
const service = new AssistantV2({
  version: '2019-02-28',
  authenticator: new IamAuthenticator({
    apikey: '{apikey}', // replace with API key
  })
});

const assistantId = '{assistant_id}'; // replace with assistant ID
let sessionId;

// Create session.
service
  .createSession({
    assistantId,
  })
  .then(res => {
    sessionId = res.result.session_id;
    sendMessage({
      messageType: 'text',
      text: '', // start conversation with empty message
    });
  })
  .catch(err => {
    console.log(err); // something went wrong
  });

// Send message to assistant.
function sendMessage(messageInput) {
  service
    .message({
      assistantId,
      sessionId,
      input: messageInput,
    })
    .then(res => {
      processResponse(res.result);
    })
    .catch(err => {
      console.log(err); // something went wrong
    });
}

// Process the response.
function processResponse(response) {

  let endConversation = false;
  let promptOption = false;

  // Check for client actions requested by the assistant.
  if (response.output.actions) {
    if (response.output.actions[0].type === 'client') {
      if (response.output.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.output.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 assistant, if any. Supports only a single
// response.
if (response.output.generic) {
  if (response.output.generic.length > 0) {
    switch (response.output.generic[0].response_type) {
      case 'text':
        // It's a text response, so we just display it.
        console.log(response.output.generic[0].text);
        break;
      case 'option':
        // It's an option response, so we'll need to show the user
        // a list of choices.
        console.log(response.output.generic[0].title);
        const options = response.output.generic[0].options;
        // List the options by label.
        for (let i = 0; i < options.length; i++) {
          console.log((i+1).toString() + ' ' + options[i].label);
        }
        promptOption = true;
        break;
    }
  }
}

// If we're not done, prompt for the next round of input.
if (!endConversation) {
  let messageInput;
  if (promptOption == true) {
    // Prompt for a valid selection from the list of options.
    let choice;
    do {
      choice = prompt('?');
      if (isNaN(choice)) {
        choice = 0;
      }
    } while (choice < 1 || choice > response.output.generic[0].options.length);
    const value = response.output.generic[0].options[choice-1].value;
    // Use message input from the selected option.
    messageInput = value.input;
  } else {
    // We're not showing options, so we just prompt for the next
    // round of input.
    const newText = prompt('>>');
    messageInput = {
      text: newText,
    }
  }
  sendMessage(messageInput);
} else {
// We're done, so we delete the session.
service
  .deleteSession({
    assistantId,
    sessionId,
  })
  .then(res => {
    return;
  })
  .catch(err => {
    console.log(err); // something went wrong
  });
}
}

```

All we have to do is use the `value.input` object from the selected response as the next round of message input, rather than building a new `input` object using text input. The assistant then responds exactly as it would if the user had typed the input text directly.

```

Welcome to the Watson Assistant example!
>> hi
Good day to you.
>> what are the choices?
What do you want to do?
1. Send greeting
2. Display the local time
3. Exit

```

```
? 2
The current time is 1:29:14 PM.
>> bye
OK! See you later.
```

We can now access all of the functions of the assistant either by making natural-language requests or by selecting from a menu of options.

Note that the same approach is used for **suggestion** responses as well. If your plan supports the disambiguation feature, you can use similar logic to prompt users to select from a list when it isn't clear which of several possible options is correct. For more information about the disambiguation feature, see [Disambiguation](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Processing input attachments](#).

Processing input attachments

If you are building a custom channel application using the REST API, you can add support for sending media files as input attachments.

The request body of the **message** method supports an **attachments** array, which can specify up to 5 media objects. Media objects sent in the **attachments** array can be intercepted and processed by a configured premessage webhook. (Watson Assistant itself does not process input attachments.)

For detailed information about how to access attachments using the API, see the [API reference](#).

Examples

The following example shows the request body sent to a premessage webhook with a message body that includes the text **Hello** and a JPEG image file as an attachment. The application receiving the webhook request can parse the **attachments** array and process the attachment, optionally modifying the message, before it is processed by the assistant.

```
{
  "event": {
    "name": "message_received"
  },
  "options": {
  },
  "payload": {
    "input": {
      "message_type": "text",
      "text": "Hello",
      "source": {
        "type": "user",
        "id": "0000000000000000000000000000000000000000000000000000000000000000"
      },
      "options": {
        "suggestion_only": false,
        "return_context": true
      },
      "attachments": [
        {
          "media_type": "image/jpeg",
          "url": "https://yourphoto.com/yourphoto.jpeg"
        }
      ]
    },
    "context": {
      "global": {
        "system": {
          "user_id": "0000000000000000000000000000000000000000000000000000000000000000"
        }
      },
      "integrations": {
        "text.messaging": {
          "assistant_phone_number": "+12223334444",
          "private": {
            "user_phone_number": "+14443332222",
            "request_id": "00000000-0000-0000-0000-000000000000",
            "ip_address": "172.10.10.10"
          }
        }
      }
    }
  }
}
```

This example shows a stateful **/message** request sent by a custom channel application and including a single media attachment.

Node

```
service
  .message({
    assistant_id: '{assistant_id}',
    session_id: '{session_id}',
    input: {
      message_type: 'text',
      text: 'Hello',
      attachments: [
        {
          'media_type': 'image/jpeg',
          'url': 'https://yourphoto.com/yourphoto.jpeg'
        }
      ]
    }
  })
  .then(res => {
    console.log(JSON.stringify(res, null, 2));
  })
  .catch(err => {
    console.log(err);
  });
});
```

Python

```
response=service.message(
  assistant_id='{assistant_id}',
  session_id='{session_id}',
  input={
    'message_type': 'text',
    'text': 'Hello',
    'attachments': [
      {
        'media_type': 'image/jpeg',
        'url': 'https://yourphoto.com/yourphoto.jpeg'
      }
    ]
  }
).get_result()

print(json.dumps(response, indent=2))
```

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:

- [Java SDK](#)
- [Node.js SDK](#)
- [Python SDK](#)
- [.NET SDK](#)



Tip: The [API reference](#) for each service includes information and examples for these SDKs.

Community SDKs

The following SDKs are available from the Watson community of developers:

- [ABAP SDK for IBM Watson](#), using SAP NetWeaver
- [Android SDK](#)
- [Go SDK](#)
- [Ruby SDK](#)
- [Salesforce SDK](#)
- [Swift SDK](#)
- [Unity SDK](#)

SDK updates and deprecation

The supported Watson SDKs are updated according to the following guidelines.

Semantic versioning

Supported Watson SDKs adhere to semantic versioning with releases labeled as `{major}.{minor}.{patch}`.

Release frequency

SDKs are released independently and might not update on the same schedule.

- The current releases of the Watson SDKs are updated on a 2- to 6-week schedule. These releases are either minor updates or patches that do not include breaking changes. You can update to any version of the SDK with the same major version number.
- Major updates that might include breaking changes are released approximately every 6 months.

Deprecated release

When a major version is released, support continues on the previous major release for 12 months in a deprecation period. The deprecated release might be updated with bug fixes, but no new features will be added and documentation might not be available.

Obsolete release

After the 12-month deprecation period, a release is obsolete. The release might be functional but is unsupported and not updated. Update to the current release.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Using IBM Cloud Activity Tracker to audit user activity](#).

Activity Tracker events

As a security officer, auditor, or manager, you can use the Activity Tracker service to track how users and applications interact with IBM Watson® Assistant in IBM Cloud®.

IBM Cloud Activity Tracker records user-initiated activities that change the state of a service in IBM Cloud. You can use this service to investigate abnormal activity and critical actions and to comply with regulatory audit requirements. In addition, you can be alerted about actions as they happen. The events that are collected comply with the Cloud Auditing Data Federation (CADF) standard. For more information, see [Getting started with IBM Cloud Activity Tracker](#).

Enterprise This feature is available to Enterprise plan users only.

List of events

The following table lists the Watson Assistant actions that generate events.

Action	Triggered when someone...
<code>conversation.assistant.create</code>	creates an assistant.
<code>conversation.assistant.delete</code>	deletes an assistant.
<code>conversation.assistant.update</code>	updates an assistant. For example, renames the skill, changes the session timeout, or changes its associated skills.
<code>conversation.counterexample.create</code>	marks test user input in the "Try it out" pane as being irrelevant or corrects the categorization of a user input that was incorrectly assigned to an intent by marking it as irrelevant.
<code>conversation.counterexample.delete</code>	deletes a counterexample.
<code>conversation.counterexample.update</code>	edits a counterexample.
<code>conversation.data.update</code>	does a bulk action, such as importing a CSV file of intents or entities to the skill, or deleting multiple training data items, such as multiple entities or intents.
<code>conversation.entity.create</code>	creates an entity.
<code>conversation.entity.delete</code>	deletes an entity.
<code>conversation.entity.update</code>	edits an entity.
<code>conversation.example.create</code>	adds a user input example to an intent.
<code>conversation.example.delete</code>	deletes a user example from an intent.
<code>conversation.example.update</code>	edits a user example that is associated with an intent.
<code>conversation.intent.create</code>	creates an intent.
<code>conversation.intent.delete</code>	deletes an intent.
<code>conversation.intent.update</code>	edits an intent.
<code>conversation.log.create</code>	corrects an intent that was inaccurately categorized by the skill from the Analytics>User conversations page.
<code>conversation.node.create</code>	creates a dialog node.
<code>conversation.node.delete</code>	deletes a dialog node.

<code>conversation.node.update</code>	edits a dialog node.
<code>conversation.recommendationfile.create</code>	uploads a CSV file of utterances to a skill from which Watson can derive intent recommendations.
<code>conversation.recommendationfile.delete</code>	deletes a CSV file of utterances that is used to derive intent recommendations from a skill.
<code>conversation.recommendationsources.update</code>	updates a CSV file or assistant log that is being used as the source for intent recommendations.
<code>conversation.seedlog.create</code>	uploads a CSV file of utterances to a service instance so that skills in the instance can derive intent recommendations from it.
<code>conversation.seedlog.delete</code>	deletes all of the CSV files that are used to derive intent recommendations from a service instance.
<code>conversation.skill.create</code>	creates a skill, either dialog or search.
<code>conversation.skill.delete</code>	deletes a skill.
<code>conversation.skill.update</code>	updates a skill.
<code>conversation.skill_reference.create</code>	adds a specific skill to an assistant.
<code>conversation.skill_reference.delete</code>	removes a specific skill from an assistant.
<code>conversation.skill_reference.update</code>	updates a specific skill that is associated with an assistant.
<code>conversation.snapshot.create</code>	creates a version of a dialog skill.
<code>conversation.snapshot.delete</code>	deletes a version of a dialog skill.
<code>conversation.synonym.create</code>	creates a synonym for an entity value.
<code>conversation.synonym.delete</code>	deletes a synonym that is associated with an entity value.
<code>conversation.synonym.update</code>	edits a synonym that is associated with an entity value.
<code>conversation.userdata.delete</code>	deletes data that was created by a specified customer.
<code>conversation.value.create</code>	creates an entity value.
<code>conversation.value.delete</code>	deletes an entity value.
<code>conversation.value.update</code>	edits an entity value.
<code>conversation.workspace.create</code>	creates a workspace.
<code>conversation.workspace.delete</code>	deletes a workspace.
<code>conversation.workspace.update</code>	makes changes to a workspace.

Table 1. Actions that generate events

Viewing events

Events that are generated by an instance of the Watson Assistant service are automatically forwarded to the IBM Cloud Activity Tracker service instance that is available in the same location. However, if your service instance is hosted in the **Washington DC** location, create the IBM Cloud Activity Tracker

service instance in the **Dallas** region.

IBM Cloud Activity Tracker can have only one instance per location. To view events, you must access the web UI of the IBM Cloud Activity Tracker service in the same location where your service instance is available. For more information, see [Navigating to the UI](#).

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Automatic retraining](#).

Automatic retrain of old skills and workspaces

Watson Assistant was released as a service in July 2016. Since then, users have been creating and updating skills to meet their virtual assistant needs. Behind the scenes, Watson Assistant creates machine learning (ML) models to perform a variety of tasks on the user's behalf.

The primary ML models deal with action recognition, intent classification, and entity detection. For example, the model might detect what a user intends when saying **I want to open a checking account**, and what type of account the user is talking about.

These ML models rely on a sophisticated infrastructure. There are many intricate components that are responsible for analyzing what the user has said, breaking down the user's input, and processing it so the ML model can more easily predict what the user is asking.

Since Watson Assistant was first released, the product team has been making continuous updates to the algorithms that generate these sophisticated ML models. Older models have continued to function while running in the context of newer algorithms. Historically, the behavior of these existing ML model did not change unless the skill was updated, at which point the skill was retrained and a new model generated to replace the older one. This meant that many older models never benefited from improvements in our ML algorithms.

Watson Assistant uses continuous retraining. The {{site.data.keassistant_classic_shortnshort}} service continually monitors all ML models, and automatically retrains those models that have not been retrained in the previous 6 months. {{site.dassistant_classic_shortrsationshort}} retrains using the selected algorithm version. If the version you had selected is no longer supported, {{assistant_classic_short.assistant_classic_short}} retrains using the version labeled as **Previous**. This means that your assistant will automatically have the supported technologies applied. Assistants that have been modified during the previous 6 months will not be affected. For more information, see [Algorithm version](#).



Note: In most cases, this retraining will be seamless from an end-user point of view. The same inputs will result in the same actions, intents, and entities being detected. In some cases, the retraining might cause changes in accuracy.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

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.

Where to use the expression syntax

To expand variable values inside other variables, or apply methods to output text or context variables, use the `<? expression ?>` expression syntax. For example:

- Referencing a user's input from a dialog node text response

```
You said <? input.text ?>.
```

- Incrementing a numeric property from the JSON editor

```
"output": {"number": "<? output.number + 1 ?>"}
```

- Checking for a specific entity value from a dialog node condition

```
@city.toLowerCase() == 'paris'
```

- Checking for a specific date range from a dialog node response condition

```
@sys-date.after(today())
```

- Adding an element to a context variable array from the context editor

Context variable name	Context variable value
toppings	<code><? context.toppings.append('onions') ?></code>

You can use SpEL expressions in dialog node conditions and dialog node response conditions also.



Important: When a SpEL expression is used in a node condition, the surrounding `<? ?>` syntax is not required.

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.addAll(JSONArray)

This method appends one array to another.

For this dialog runtime context:

```
{
  "context": {
    "toppings_array": ["onion", "olives"],
    "more_toppings": ["mushroom", "pepperoni"]
  }
}
```

Make this update:

```
{
  "context": {
```

```
        "toppings_array": "<? $toppings_array.addAll($more_toppings) ?>"  
    }  
}
```

Result: The method itself returns `null`. However, the first array is updated to include the values from the second array.

```
{  
    "context": {  
        "toppings_array": ["onion", "olives", "mushroom", "pepperoni"]  
    }  
}
```

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.containsIgnoreCase(Object value)

This method returns `true` if the input JSONArray contains the input value, regardless of whether the value is specified in uppercase or lowercase letters.

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.containsIgnoreCase('HAM')
```

Result: `true` because the array contains the element `ham` and the case is ignored.

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. The `top_n` parameter is ignored if you specify a negative number.

Returns `false` if the specified intent is not in the array, does not have a confidence score that is equal to or greater than the minimum confidence score, or the array index of the intent is lower 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:

Operator	Description
<code>==</code>	Is equal to
<code>></code>	Is greater than
<code><</code>	Is less than

>=

Is greater than or equal to

<=

Is less than or equal to

!=

Is not equal to

Supported filter operators

- **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  
  }  
]
```

 **Tip:** 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 earlier expression 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 a user input mentions multiple toppings, and you defined an entity named `@toppings` that can recognize topping mentions, you could use the following expression in the response to list the toppings that were mentioned:

```
So, you'd like <? @toppings.values.join(',') ?>.
```

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, retainDataType)

This method extracts information from each item in the array and builds a new array that is formatted according to the template you specify. The template can be a string, a JSON object, or an array. The method returns an array of strings, an array of objects, or an array of arrays, depending on the type of template.

This method is useful for formatting information as a string you can return as part of the output of a dialog node, or for transforming data into a different structure so you can use it with an external API.

In the template, you can reference values from the source array using the following syntax:

```
$ %e.{property}%
```

where `{property}` represents the name of the property in the source array.

For example, suppose your assistant has stored an array containing flight details in a context variable. The stored data might look like this:

```
"flights": [
  {
    "flight": "AZ1040",
    "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": "VS4379",
    "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"
  }
]
```

]

To build an array of strings that describe these flights in a user-readable form, you might use the following expression:

```
$ ${Flight_data}.joinToArray("Flight %e.flight% to %e.destination%", true)
```

This expression would return the following array of strings: ["Flight AZ1040 to FC0", "Flight DL1710 to LAX", "Flight VS4379 to LHR"].

The optional `retainDataType` parameter specifies whether the method should preserve the data type of all input values in the returned array. If `retainDataType` is set to `false` or omitted, in some situations, strings in the input array might be converted to numbers in the returned array. For example, if the selected values from the input array are "1", "2", and "3", the returned array might be [1, 2, 3]. To avoid unexpected type conversions, specify `true` for this parameter.

Complex templates

A more complex template might contain formatting that displays the information in a legible layout. For a complex template, you might want to store the template in a context variable, which you can then pass to the `joinToArray` method instead of a string.

For example, this complex template contains a subset of the array elements, adding labels and formatting:

```
<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/>
```



Note: Make sure any formatting you use in your template is supported by the channel integration that will be displaying the assistant output.

If you create a context variable called `Template`, and assign this template as its value, you can then use that variable in your expressions:

```
$ ${Flight_data}.joinToArray(${Template})
```

At run time, the response would look like this:

```
Flight number: AZ1040  
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: VS4379  
Airline: Virgin Atlantic  
Departure date: 2019-02-02  
Departure time: 21:40  
Arrival time: 09:05
```

JSON Object templates

Instead of a string, you can define a template as a JSON object. This provides a way to standardize the formatting of information from different systems, or to transform data into the format required for an external service.

In this example, a template is defined as a JSON object that extracts flight details from the elements specified in the array stored in the `Flight data` context variable:

```
{  
  "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%."  
}
```

Using this template, the `joinToArray()` method returns a new array of objects with the specified structure.

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](#).

The following string formats are supported for date-time literals on which the methods below may be invoked.

- For time only: `HH:mm:ss` or `HH:mm`
- For date only: `yyyy-MM-dd`
- For date and time: `yyyy-MM-dd HH:mm:ss`
- For date and time with time zone: `yyyy-MM-dd HH:mm:ss VV`. The V symbol is from the [DateTimeFormatter](#) and represents time zone in IANA Time Zone Database (TZDB) format, for example, Europe/London.

.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(String time zone)

Returns a string with the current date and time in the format `yyyy-MM-dd HH:mm:ss`. Optionally specify a `timezone` value to get the current date and time for a specific time zone, with a returned string in the format `yyyy-MM-dd HH:mm:ss 'GMT'XXX`.

- 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.
- The user interface creates a `$timezone` context variable for you automatically so the correct time is returned when you test from the "Try it out" pane. If you don't pass a time zone, the time zone that is set automatically by the UI is used. Outside of the UI, `GMT` is used as the time zone. To learn about the syntax to use to specify the time zone, see [Time zones supported by system entities](#).

Example of `now()` being used to first check whether it's morning before responding with a morning-specific greeting.

```
{  
  "conditions": "now().before('12:00:00')",  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "Good morning!"  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

Example of using now() with a timezone to return the current time (in England):

```
{  
  "output": {  
    "generic": [  
      {  
        "values": [  
          {  
            "text": "The current date and time is: <? now('Europe/London') ?>"  
          }  
        ],  
        "response_type": "text",  
        "selection_policy": "sequential"  
      }  
    ]  
  }  
}
```

You can substitute the hard-coded time zone value with a context variable to dynamically change the time based on a time zone that is passed to the expression. For example: <? now('{\$myzone}') ?>. The `[$myzone]` context variable might be set to `'Australia/Sydney'` in one conversation and to `'Mexico/BajaNorte'` in another.

.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:

- `EEEE` for Tuesday
- `E` for Tue
- `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": "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
<code><time>.minusHours(n)</code>	Returns the time n hours before the specified time.
<code><time>.minusMinutes(n)</code>	Returns the time n minutes before the specified time.
<code><time>.minusSeconds(n)</code>	Returns the time n seconds before the specified time.
<code><time>.plusHours(n)</code>	Returns the time n hours after the specified time.
<code><time>.plusMinutes(n)</code>	Returns the time n minutes after the specified time.
<code><time>.plusSeconds(n)</code>	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, `'50000000000L'`. 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 $2^{63}-1$ (or 9,223,372,036,854,775,807).

If you need to find out if a number is too long to be recognized properly in the dialog, you can check whether there are more than 18 integers in the number by using an expression like this:

```
``` {: codeblock}
```

18 ?>

`$ If you need to work with numbers that are longer than 18 integers, consider using a pattern entity (with a regular expression such as '\d{20}') to work with them instead of using '@sys-number'.`

```
Standard math {:#dialog-methods-numbers-standard-math}
```

Use SpEL expressions to define standard math equations, where the operators are represented by using these symbols:

Arithmetic operation	Symbol
addition	+
division	/
multiplication	*
subtraction	-

For example, in a dialog node response, you might add a context variable that captures a number specified in the user input (`@sys-number`), and saves it as `'\$your\_number'`. You can then add the following text as a text response:

```
``` {:#codeblock}
I'm doing math. Given the value you specified ($your_number), when I add 5, I get: <? $your_number + 5 ?>.
When I subtract 5, I get: <? $your_number - 5 ?>.
When I multiply it by 5, I get: <? $your_number * 5 ?>.
When I divide it by 5, I get: <? $your_number/5 ?>.
```

If the user specifies **10**, then the resulting text response looks like this:

```
I'm doing math. Given the value you specified (10), when I add 5, I get: 15.
When I subtract 5, I get: 5.
When I multiply it by 5, I get: 50.
When I divide it by 5, I get: 2.
```

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 earlier, 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, some of the standard methods of the `com.google.gson.JsonObject` class are also supported.

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.equals(String)

This method returns **true** if the specified string equals the input string exactly.

Input: "Yes"

This syntax:

```
{  
  "conditions": "input.text.equals('Yes')"  
}
```

Results: The condition is **true**.

If the input is **Yes.**, then the result is **false** because the user included a period and the expression expects only the exact text, **Yes** without any punctuation.

String.equalsIgnoreCase(String)

This method returns **true** if the specified string equals the input string, regardless of whether the case of the letters match.

Input: "yes"

This syntax:

```
{  
  "conditions": "input.text.equalsIgnoreCase('Yes')"  
}
```

Results: The condition is **true**.

If the input is **Yes.**, then the result is **false** because the user included a period and the expression expects only the text, **Yes**, in uppercase or lowercase letters without any punctuation.

String.extract(String regexp, Integer groupIndex)

This method returns a string from the input that matches the regular expression group pattern that you specify. It returns an empty string if no match is found.



Note: This method is designed to extract matches for different regex pattern groups, not different matches for a single regex pattern. To find different matches, see the [getMatch](#) method.

In this example, the context variable is saving a string that matches the regex pattern group that you specify. In the expression, two regex patterns groups are defined, each one enclosed in parentheses. There is an inherent third group that is comprised of the two groups together. This is the first (groupIndex 0) regex group; it matches with a string that contains the full number group and text group together. The second regex group (groupIndex 1) matches with the first occurrence of a number group. The third group (groupIndex 2) matches with the first occurrence of a text group after a number group.

```
{  
  "context": {  
    "number_extract": "<? input.text.extract('([\d]+)(\b [A-Za-z]+)',n) ?>"  
  }  
}
```

When you specify the regex in JSON, you must provide two backslashes (\). If you specify this expression in a node response, you need one backslash only. For example:

```
<? input.text.extract('([\d]+)(\b [A-Za-z]+)',n) ?>
```

Input:

```
"Hello 123 this is 456".
```

Result:

- When n=0, the value is `123 this`.
- When n=1, the value is `123`.
- When n=2, the value is `this`.

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.getMatch(String regexp, Integer matchIndex)

This method returns a string from the input that matches the occurrence of the regular expression pattern that you specify. This method returns an empty string if no match is found.

As matches are found, they are added to what you can think of as a *matches array*. If you want to return the third match, because the array element count starts at 0, specify 2 as the `matchIndex` value. For example, if you enter a text string with three words that match the specified pattern, you can return the first, second, or third match only by specifying its index value.

In the following expression, you are looking for a group of numbers in the input. This expression saves the second pattern-matching string into the `$second_number` context variable because the index value 1 is specified.

```
{
  "context": {
    "second_number": "<? input.text.getMatch('([\\d]+)',1) ?>"
  }
}
```

If you specify the expression in JSON syntax, you must provide two backslashes (\). If you specify the expression in a node response, you need one backslash only.

For example:

```
<? input.text.getMatch('([\\d]+)',1) ?>
```

- User input:

```
"hello 123 i said 456 and 8910".
```

- Result: `456`

In this example the expression looks for the third block of text in the input.

```
<? input.text.getMatch('(^\\b [A-Za-z]+)',2) ?>
```

For the same user input, this expression returns `and`.

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.toJson()

This method parses a string that contains JSON data and returns a JSON object or array, as in this example:

```
$ ${json_var}.toJson()
```

If the context variable `${json_var}` contains the following string:

```
$ "{ \"firstname\": \"John\", \"lastname\": \"Doe\" }"
```

the `toJson()` method returns the following object:

```
$ {  
  "firstname": "John",  
  "lastname": "Doe"  
}
```

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_lower_case": "this is a dog!"  
  }  
}
```

String.toUpperCase()

This method returns the original String converted to uppercase 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.

Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

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": {"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 "

When you refer to a context variable in a text response or a dialog node condition, you can use the short syntax.

For example, `Hello, $name`. If the `$name` context variable contains `Sam`, then the response is shown as `Hello, Sam`.

If you want to reference a context variable by using the full syntax in a text response, be sure to surround the context variable in `<? ?>`. For example, `Hello, <? context['name'] ?>`.

If you want to reference a context variable that has multiple fields, such as `$context.integrations.chat.browser_info.page_url`. To use the full syntax, specify `<? context['integrations']['chat']['browser_info']['page_url'] ?>`.

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

<code>@year == 2016</code>	<code>entities['year']?.value == 2016</code>
<code>@year != 2016</code>	<code>entities['year']?.value != 2016</code>
<code>@city == 'Boston'</code>	<code>entities['city']?.value == 'Boston'</code>
<code>@city:Boston</code>	<code>entities['city']?.contains('Boston')</code>
<code>@city:(New York)</code>	<code>entities['city']?.contains('New York')</code>

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) 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
<code>'#help'</code>	<code>'intent == 'help''</code>
<code>'! #help'</code>	<code>'intent != 'help''</code>
<code>'NOT #help'</code>	<code>'intent != 'help''</code>
<code>'#help' or '#i_am_lost'</code>	<code>'(intent == 'help' intent == 'I_am_lost')'</code>
Intents shorthand syntax	

Built-in global variables

You can use the expression language to extract property information for the following global variables:

Global variable	Definition
<code>context</code>	JSON object part of the processed conversation message.
<code>entities[]</code>	List of entities that supports default access to 1st element.
<code>input</code>	JSON object part of the processed conversation message.
<code>intents[]</code>	List of intents that supports default access to first element.
<code>output</code>	JSON object part of the processed conversation 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, `today`, your assistant recognizes the @sys-date system entity, so the response contains this entity object:

```
[
  {
    "entity": "sys-date",
    "location": [0,5],
    "value": "2020-12-30",
```

```

"confidence":1.0,
"metadata":
{
  "calendar_type":"GREGORIAN",
  "timezone":"America/New_York"
},
"interpretation":
{
  "timezone":"America/New_York",
  "relative_day":0,
  "granularity":"day",
  "calendar_type":"GREGORIAN"
}
]

```

If you want to include text in the response, use the `toJson()` method in the expression to cast the returned entities list into a JSON object. For example:

Recognized entities are: <? entities.toJson() ?>

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 your assistant'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 <code>location</code> property.
<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 skill 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 your assistant 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.`

- To capture the literal values for multiple entity mentions, use the following syntax:

```
$ entities['myEntityName'].![literal]
```

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 your assistant'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*, your assistant 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}]
```

If you want to include text in the response, use the `toJson()` method in the expression to cast the returned intents list into a JSON object. For example:

```
Recognized intents are: <? intents.toJson() ?>
```

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`.

Filter query reference

The Watson Assistant service REST API offers powerful log search capabilities through filter queries. You can use the v2 /logs API `filter` parameter to search your skill log for events that match a specified query.

The `filter` parameter is a cacheable query that limits the results to those matching the specified filter. You can filter on various objects that are part of the JSON response model (for example, the user input text, the detected intents and entities, or the confidence score).

To see examples of filter queries, see [Examples](#).

For more information about the /logs `GET` method and its response model, refer to the [API Reference](#).

Filter query syntax

The following example shows the general form of a filter query:

Location	Query operator	Term
<code>request.input.text</code>	<code>::</code>	<code>"IBM Watson"</code>

- The *location* identifies the field that you want to filter on (in this example, `request.input.text`).
- The *query operator*, which specifies the type of matching you want to use (fuzzy matching or exact matching).
- The *term* specifies the expression or value you want to use to evaluate the field for matching. The term can contain literal text and operators, as described in the [next section](#).

Filtering by intent or entity requires slightly different syntax from filtering on other fields. For more information, see [Filtering by intent or entity](#).

Note: The filter query syntax uses some characters that are not allowed in HTTP queries. Make sure that all special characters, including spaces and quotation marks, are URL encoded when sent as part of an HTTP query. For example, the filter `response_timestamp<2020-01-01` would be specified as `response_timestamp%3C2020-01-01`.

Operators

You can use the following operators in your filter query.

Operator	Description
<code>:</code>	Fuzzy match query operator. Prefix the query term with <code>:</code> if you want to match any value that contains the query term, or a grammatical variant of the query term. Fuzzy matching is available for user input text and entity values.
<code>::</code>	Exact match query operator. Prefix the query term with <code>::</code> if you want to match only values that exactly equal the query term.
<code>:!</code>	Negative fuzzy match query operator. Prefix the query term with <code>:!</code> if you want to match only values that do <i>not</i> contain the query term or a grammatical variant of the query term.
<code>::!</code>	Negative exact match query operator. Prefix the query term with <code>::!</code> if you want to match only values that do <i>not</i> exactly match the query term.
<code><=, >=,</code> <code>>, <</code>	Comparison operators. Prefix the query term with these operators to match based on arithmetic comparison.
<code>\`</code>	Escape operator. Use in queries that include control characters that would otherwise be parsed as operators. For example, <code>!hello</code> would match the text <code>!hello</code> .
<code>" "</code>	Literal phrase. Use to encloses a query term that contains spaces or other special characters. No special characters within the quotation marks are parsed by the API.
<code>~</code>	Approximate match. Append this operator followed by a <code>1</code> or <code>2</code> to the end of the query term to specify the allowed number of single-character differences between the query string and a match in the response object. For example, <code>car~1</code> would match <code>car</code> , <code>cat</code> , or <code>cars</code> , but it would not match <code>cats</code> . This operator is not valid when filtering on <code>log_id</code> or any date or time field, or with fuzzy matching.
<code>*</code>	Wildcard operator. Matches any sequence of zero or more characters. This operator is not valid when filtering on <code>log_id</code> , <code>language</code> , <code>request.context.system.assistant_id</code> , <code>workspace_id</code> , <code>request.context.metadata.deployment</code> , or any date or time field.

(), [] Grouping operators. Use to enclose a logical grouping of multiple expressions using Boolean operators.

| Boolean *or* operator.

, Boolean *and* operator.

Filtering by intent or entity

Because of differences in how intents and entities are stored internally, the syntax for filtering on a specific intent or entity is different from the syntax used for other fields in the returned JSON. To specify an `intent` or `entity` field within an `intents` or `entities` collection, you must use the `:` match operator instead of a dot.

For example, this query matches any logged event where the response includes a detected intent named `hello`:

```
response.output.intents:intent::hello
```

Note the `:` operator in place of a dot (`intents:intent`)

Use the same pattern to match on any field of a detected intent or entity in the response. For example, this query matches any logged event where the response includes a detected entity with the value `soda`:

```
response.output.entities:value::soda
```

Similarly, you can filter on intents or entities sent as part of the request, as in this example:

```
request.input.intents:intent::hello
```

Filtering by other fields

To filter on another field in the log data, specify the location as a path identifying the levels of nested objects in the JSON response from the /logs API. Use dots (`.`) to specify successive levels of nesting in the JSON data. For example, the location `request.input.text` identifies the user input text field as shown in the following JSON fragment:

```
$ "request": {  
  "input": {  
    "text": "Good morning"  
  }  
}
```

Filtering is not available for all fields. You can filter on the following fields:

- `assistant_id`
- `customer_id`
- `language`
- `request.context.global.system.user_id`
- `request.input.text`
- `request_timestamp`
- `response.context.global.system.user_id`
- `response.output.entities`
- `response.output.intents`
- `response_timestamp`
- `session_id`
- `skill_id`
- `snapshot`

Filtering on other fields is not currently supported.

Examples

The following examples illustrate various types of queries using this syntax.

Description	Query
The date of the response is in the month of July 2020.	<code>response_timestamp>=2020-07-01, response_timestamp<2020-08-01</code>

The timestamp of the response is earlier than 2019-11-01T04:00:00.000Z .	<code>response_timestamp<2019-11-01T04:00:00.000Z</code>
The message is labeled with the customer ID my_id .	<code>customer_id::my_id</code>
The message was sent to a specific assistant.	<code>assistant_id::dcd5c5ad-f3a1-4345-89c5-708b0b5ff4f7</code>
The user input text contains the word "order" or a grammatical variant (for example, orders or ordering).	<code>request.input.text:order</code>
An intent name in the response exactly matches place_order .	<code>response.output.intents:intent::place_order</code>
An entity name in the response exactly matches beverage .	<code>response.output.entities:entity::beverage</code>
No intent name in the response exactly matches order .	<code>response.intents:intent::!order</code>
The user input text does not contain the word "order" or a grammatical variant.	<code>request.input.text:!order</code>
The user input text contains the string !hello .	<code>request.input.text:!hello</code>
The user input text contains the string IBM Watson .	<code>request.input.text:"IBM Watson"</code>
The user input text contains a string that has no more than 2 single-character differences from Watson .	<code>request.input.text:Watson~2</code>
The user input text contains a string consisting of comm , followed by zero or more additional characters, followed by s .	<code>request.input.text:comm*s</code>
An intent name in the response exactly matches either hello or goodbye .	<code>response.output.intents:intent::(hello goodbye)</code>
An intent name in the response exactly matches order , and an entity name in the response exactly matches beverage .	<code>[response.output.intents:intent::order, response.output.entities:entity::beverage]</code>

Filtering v1 logs

If your application is still using the v1 API, you can query and filter logs using the v1 /logs method. The filtering syntax is the same, but the structure of v1 logs and message requests is different. For more information, see [API Reference](#).

With the v1 /logs API, you can filter on the following fields:

- `language`
- `meta.message.entities_count`
- `request.context.metadata.deployment`

- `request.context.system.assistant_id`
- `request.input.text`
- `response.context.conversation_id`
- `response.entities`
- `response.input.text`
- `response.intents`
- `response.top_intent`
- `workspace_id`

Filtering on other fields is not currently supported.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Failover options](#)

Failover options

Learn about options that you can use to increase the availability of Watson Assistant for your organization.

Introduction

IBM Watson® Assistant instances are deployed across multizone regions (MZRs) in all data centers (except Seoul, Korea) and can withstand the loss of any individual zone within an MZR. However, regional outages can occur and while IBM strives to keep outages to a minimum, you might want to consider moving traffic to a different region for your business-critical applications.

You should have a copy of your Watson Assistant instance in a different MZR (for example, deploy your assistant in US East and US South). You must purchase a second service instance of Watson Assistant, and second instances of the necessary assistants and skills need to be instantiated. You must implement change management controls to synchronize the changes between the two regions. Watson Assistant and Speech services have APIs available to export and import the definitions.

With two instances, consider two topologies:

- **Active/active:** Both instances always serving traffic with the load sprayed between the two.
- **Active/passive:** One instance is active and the passive site receives traffic only if a failover happens.

Each approach has pros and cons. Considerations specific to Watson Assistant are detailed in the sections that follow.

Considerations

Watson Assistant instances in one region are unaware of instances in a second region, which can affect some features and capabilities in Watson Assistant.

Analytics

Watson Assistant analytics provide overview statistics on the number of interactions with users and containment rates. Analytics doesn't cumulate statistics across regions. With an active/passive topology, this approach to analytics is sufficient. However, using an active/active topology likely requires you to use [webhooks](#) to gather interaction data, and build custom data warehouses and reports to understand total usage.

Session history for web chat and the v2 api

Session history allows your web chats to maintain conversation history and context when users refresh a page or change to a different page on the same website. This feature doesn't work across instances, so in-progress conversations need to be restarted.

Billing

IBM calculates your bill based on the IBM Cloud Account. Watson Assistant calculates monthly average user (MAU) metrics by aggregating within a specific service instance as follows:

- The same MAU used in 2 different assistant resources in the **same** service instance counts as 1 MAU
- The same MAU used in 2 different assistant resources in **different** service instances counts as 2 MAUs

For an **active/active** topology, under the worst case scenario, the MAU count might end up being doubled for a billing period.

Phone integration

A Watson Assistant phone integration in one region is unaware of a phone integration in a different region. You need to ensure that your assistants are identically configured in both regions. You also need to rely on the upstream SIP trunking provider to detect and manage failing over between regions.

Monitoring

SIP trunking providers can be configured to actively health-check the Watson Assistant session border controllers (SBCs) by sending periodic SIP OPTIONS messages to each zone within a region. A failure to receive a response can be used to either provide notification of a failure to trigger a manual failover, or to automate removal of the failed zone from the route list.

Failover

The SIP trunking provider plays an important role in detecting and managing a failover, especially if an automatic failover is expected between regions. In most cases, SIP trunking providers should be configured to treat each zone within a region as active/active and two regions where an assistant is configured as active/passive. SIP trunking providers should always be configured to balance the load and fail over between zones within a single region.

Full outage

Phone integration failures have two types. The first type is a full outage where the session border controllers in all 3 regional zones become unreachable.

This type of outage is easier to detect and handle because the SIP trunking provider is immediately notified by SIP timeouts that the call fails and can be configured to either automatically fail over or the call routing can be manually reconfigured at the SIP trunking provider to direct traffic away from the failed region toward the passive backup region. If a failover is automated and a regional backup is enabled, it is always best to try a different zone first and redirect traffic to the passive backup region only if a preconfigured number of failures occur within a short period. This prevents an unnecessary failover between regions if only a short outage occurs.

Watson Assistant provides a round-robin fully qualified domain name (FQDN) that includes the IP addresses for each zone in the region. Many SIP trunking providers automatically retry each IP in the FQDN when failures occur. To support disaster recovery, the service provider might need to configure two separate SIP trunks, one for each region, and only when all the zones in a single region fail should the call be switched to the backup region. It's important to set the SIP INVITE failure timeouts at the SIP trunking provider low enough to avoid long call setup latencies when a failover is occurring.

Partial outage

The second type of failure is a partial service outage within the region. A partial outage is much harder to detect and manage because of the large number of variations in service failures that can occur within a region. In some cases, small issues affect the performance characteristics of the call but not cause the call to fail.

For issues that ultimately cause a call to fail, two ways Watson Assistant can handle the call. The first is to accept the call and then transfer it to a configured default SIP URI. You can configure this setting in the Watson Assistant phone integration and is also used for mid-call failures. The default transfer target SIP URI is defined in the **SIP target when a call fails** field that is on the Advanced tab of the phone integration configuration.

The phone integration can also be configured to respond to a SIP INVITE with a SIP 500 (service unavailable) message if an outage is detected during call setup instead of transferring a call to a live agent. A SIP 500 can then be used to redirect the call to another zone, or if many SIP 500s are received, to another region. Using a SIP 500 INVITE error is a better way to signal a failure to an upstream SIP trunking provider because it gives the provider a way to reroute the call. Using only the default transfer target to handle call failures is acceptable for low call volume scenarios but can result in large numbers of calls that are directed to a customer contact center when bigger call volumes are handled by Watson Assistant. To enable this 500 error response capability for a specific Watson Assistant instance, you need to make a request to IBM.

You should plan for both full and partial service outages. A good first step is to plan for a manual failover between regions before you enable automation. You need a complete replica of Watson Assistant in both regions, including all custom speech model training. When automation is enabled, it is best to start with a strategy for detecting and failing over to the passive backup region when a complete regional outage is detected. After implementation, develop a strategy to deal with partial outages, which should cover most of the failure conditions that can occur with a phone integration deployment.

Web chat

Monitoring

Web chat provides an `onError` listening feature that allows the host page to detect specific types of outage errors, in particular INITIAL_CONFIG, OPEN_CONFIG, and MESSAGE_COMMUNICATION errors.

For more information, see [Listening for errors](#).

Failover

Handling a failover for web chat is simple, assuming you set up an extra web chat integration in another region. When the failover needs to be manually triggered, make the following changes:

- The embed script that contains your integration ID, region, service instance ID, and subscription ID (if applicable) needs to be changed or updated to use the IDs for the new integration and region.
- If you are using Salesforce or ZenDesk integrations for connecting to human agents, update the configuration within those systems to make sure they can communicate with the correct integration. Follow the instructions on the **Live agent** tab in the web chat configuration for setting up those systems. This capability is only needed for obtaining the conversation history for the agent.
- If you enable web chat security and you are using encrypted payloads, the IBM-provided public key that is used for the encryption might be different depending on the region. If so, you need to update the system that generates the JSON Web Token (JWT) to use the correct key.

You can set up an active/active configuration of web chat by using different integration IDs in your embed scripts, and by ensuring the web chat integration is "sticky" by user. Otherwise, if the user fails over to a different integration, the conversation history might be lost.

API

Monitoring

Capture and monitor response codes of `/message` calls. Response codes of live `/message` traffic should be captured and aggregated.

Ideally, save the response code statistics in an external persistence store. This shared store can aggregate response code results from all deployed instances of your application and provide the greatest fidelity in determining whether a failover should be triggered.

Alternatively, response codes can be aggregated and evaluated in memory for an instance of your application. As only a fraction of traffic is contributing to the failover decision, this might affect how often the failover decision is acted upon.

With either aggregation approach, exceeding a defined threshold might trigger a failover. Common approaches for determining a failover include:

- Percentage-based: greater than X% of requests return a non-200 response code
- Consecutive-based: X calls in a row return a non-200 response code
- Limit-based: X calls return a non-200 response in a time frame

To avoid an unnecessary failover between regions, make sure that a robust retry mechanism is present when calling the `/message` endpoint.

Failover for v1 and v2 stateless API

For a successful failover:

- Training data changes should be synchronized across regions. Avoid pushing changes delayed over a large window of time (such as days) to mitigate the risk of algorithm changes that are deployed by Watson Assistant in between regions being updated.
- The same IBM Cloud account should be used for the service instances across regions to maintain a single overall bill for services.
- The client applications should support:
 - Watson Assistant API hostname
 - Service instance credentials
 - v1: workspace_id
 - v2: assistant_id

Although it does not affect the runtime flow of calling `/message`, if you are using fine-grained access control that uses IBM Identity and Access Management (IAM), make sure that the IAM policies are synchronized across the regions.

IAM is a global service, but the custom resources (assistants and skills) used by Watson Assistant access control means each region, which has specific resources, requires specific policies.

For an **active/passive** topology, some form of a [circuit break pattern](#) can be used. A single service instance in a region is used exclusively unless errors are detected. At that point, the system can respond by updating the relevant failover metadata to route traffic to the service instance in the other region. When a failover happens, you can decide to continue using the new region as the active instance, or if you want to resume using the initial region when it stabilizes.

For an **active/active** topology, some form of a load balancing can be used, where two or more service instances in different regions always receive a percentage of traffic. Extra logic would need to be established to determine when to pull a region out of rotation. This monitoring logic might use a [circuit break pattern](#) similar to the active/passive configuration or rely on a separate dedicated monitoring framework that determines region health. Similar to active/passive, determining when to insert a region back in rotation would need to be considered as well.

Failover for v2 stateful API

Failover for the v2 stateful API is similar to stateless, with these details to consider:

- The state of a conversation is persisted by Watson Assistant in a database that is tied to a particular region. As such, a failover for the stateful v2 `/message` may be more disruptive.
- For an **active/passive** topology, you should assume that all in-progress conversations are ended.
- For an **active/active** topology, given the region-locked persistence constraints of the v2 stateful `/message` architecture, all turns (`/message` API calls) of a conversation (session) should occur within the same region.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [High availability and disaster recovery](#).

High availability and disaster recovery

IBM Watson® Assistant is highly available within multiple IBM Cloud® locations (for example, Dallas and Washington, DC). However, recovering from potential disasters that affect an entire location requires planning and preparation.

You are responsible for understanding your configuration, customization, and usage of Watson Assistant. You are also responsible for being ready to re-create an instance of the service in a new location and to restore your data in any location. See [How do I ensure zero downtime?](#) for more information.

High availability

Watson Assistant supports high availability with no single point of failure. The service achieves high availability automatically and transparently by using the multi-zone region feature provided by IBM Cloud.

IBM Cloud enables multiple zones that do not share a single point of failure within a single location. It also provides automatic load balancing across the zones within a region.

Disaster recovery

Disaster recovery can become an issue if an IBM Cloud location experiences a significant failure that includes the potential loss of data. Because the multi-zone region feature is not available across locations, you must wait for IBM to bring a location back online if it becomes unavailable. If underlying data services are compromised by the failure, you must also wait for IBM to restore those data services.

If a catastrophic failure occurs, IBM might not be able to recover data from database backups. In this case, you need to restore your data to return your service instance to its most recent state. You can restore the data to the same or to a different location.

Your disaster recovery plan includes knowing, preserving, and being prepared to restore all data that is maintained on IBM Cloud. See [Backing up and restoring data](#) for information about how to back up your service instances.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Phone integration context variables

You can use context variables to manage the flow of conversations with customers who interact with your assistant over the telephone.

The following tables describe the context variables that have special meaning in the context of the phone integration. They should not be used for any purpose other than the documented use.

Context variables that are set by your dialog or actions

Name	Type	Description	Default
<code>final_utterance_timeout_count</code>	Number	The time (in milliseconds) that the phone integration waits to receive a final utterance from the Speech to Text service. The timeout occurs if the phone integration does not receive a final utterance within the specified time limit, even if hypotheses continue to be generated. When the timeout occurs, the phone integration sends Watson Assistant a text update that includes the word <code>vgwFinalUtteranceTimeout</code> to indicate that no final utterance was received.	N/A
<code>post_response_timeout_count</code>	Number	The time (in milliseconds) to wait for a new utterance after the response is played back to the caller. When this timeout occurs, the phone integration channel sends a text message to the assistant that includes the word <code>vgwPostResponseTimeout</code> and sets the context variable <code>input.integrations.voice_telephony.post_response_timeout_occurred</code> to <code>true</code> .	7000
<code>turn_settings.timeout_count</code>	Number	The time (in milliseconds) to wait for a response from Watson Assistant. If this time is exceeded, the phone integration tries again to contact {{site.data.keassistant_classic_shortnshort}}. If the service still can't be reached, the call fails.	N/A
<code>cdr_custom_data</code>	object	Any JSON key/value pairs to collect and store with the CDR record at the end of the phone call. Each time this object is received, it is merged with any previously received <code>cdr_custom_data</code> context.	N/A

Voice context variables set by the dialog or actions

Example

```
{  
  "output": {  
    "generic": [  
      {  
        "response_type": "text",  
        "text": "Hello"  
      }  
    ]  
  },  
  "context": {  
    "integrations": {  
      "voice_telephony": {  
        "post_response_timeout_count": 10000,  
        "turn_settings": {  
          "timeout_count": 5000  
        },  
        "cdr_custom_data": {  
          "key1": "value1",  
          "key2": "value2"  
        }  
      }  
    }  
  }  
}
```

Context variables that are set by the phone channel

Name	Type	Description
<code>sip_call_id</code>	string	The SIP call ID associated with the Watson Assistant session.

<code>sip_custom_invite_headers</code>	object	A JSON object containing key/value pairs defining SIP headers that are pulled from the initial SIP INVITE request and passed to the Watson Assistant service (for example, <code>{"Custom-Header1": "123"}</code>).
<code>private.sip_from_uri</code>	string	The SIP From URI associated with the Watson Assistant service.
<code>private.sip_request_uri</code>	string	The SIP request URI that started the conversation session.
<code>private.sip_to_uri</code>	string	The SIP To URI associated with the conversation session.
<code>private.user_phone_number</code>	string	The phone number that the call was received from.
<code>assistant_phone_number</code>	string	The phone number associated with the Watson Assistant side that received the phone call.

Context variables set by the phone channel

Input parameters that are set by the phone channel

The following input parameters are only valid for the current conversation turn.

Name	Type	Description
<code>post_response_timeout_occurred</code>	boolean	Whether the post response timeout expired.
<code>barge_in_occurred</code>	boolean	Whether barge-in occurred.
<code>final_utterance_timeout_occurred</code>	<code>true</code> or <code>false</code>	Whether the final utterance timeout expired.
<code>dtmf_collection_succeeded</code>	boolean	Whether the DTMF collection succeeded or failed. When <code>true</code> , a DTMF collection succeeded, and returns the expected number of digits. When <code>false</code> , a DTMF collection failed to collect the specified number of digits. Even when <code>dtmf_collection_succeeded</code> is <code>false</code> , all collected digits are passed to the dialog in the input string of the turn request.
<code>is_dtmf</code>	boolean	Whether the input to Watson Assistant is dual-tone multi-frequency signaling (DTMF).
<code>speech_to_text_result</code>	object	The final response from the Speech to Text service in JSON format, including the transcript and confidence score for the top hypothesis and any alternatives. The format matches exactly the format that is received from the Speech to Text service. (For more information, see the Speech to Text API documentation .)

Input parameters set by the phone channel

Example

```
{
  "input": {
    "text": "agent",
    "integrations": {
      "voice_telephony": {
        "speech_to_text_result": {
          "result_index": 0,
          "stopTimestamp": "2021-09-29T17:43:31.036Z",
          "transaction_ids": {
            "x-global-transaction-id": "43dd6ce0-139a-4d76-95aa-86e03fcfc434",
            "x-dp-watson-tran-id": "6e60695e-fed7-4efe-a376-0888b027d30f"
          },
          "results": [
            {
              "final": true,
              "alternatives": [
                {
                  "transcript": "agent",
                  "confidence": 0.78
                }
              ]
            }
          ]
        }
      }
    }
  }
}
```

```
        }
    ],
},
"transactionID": "43dd6ce0-139a-4d76-95aa-86e03fcfc434",
"startTimestamp": "2021-09-29T17:43:29.436Z"
},
"is_dtmf": false,
"barge_in_occurred": false
}
}
},
"context": {
"skills": {
"main skill": {
"user_defined": {},
"system": {}
}
},
"integrations": {
"voice_telephony": {
"private": {
"sip_to_uri": "sip:watson-conversation@10.10.10.10",
"sip_from_uri": "sip:10.10.10.11",
"sip_request_uri": "sip:test@10.10.10.10:5064;transport=tcp"
},
"sip_call_id": "QjryZsuAS4",
"assistant_phone_number": "18882346789"
}
}
}
}
```



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

SMS with Twilio integration reference

Add action commands to the message `context` object to manage the flow of conversations with customers who interact with your assistant by submitting SMS messages over the telephone.

Learn about the supported commands and reserved context variables that are used by the `SMS with Twilio` integration.

Supported commands

Each action consists of a `command` property, followed by an optional `parameter` property to define parameters for commands that require them. The commands that are described in the following table are supported by the `SMS with Twilio` integration.

Action command	Description	Parameters
<code>smsActForceNoInputTurn</code>	Forces a new turn in the conversation without waiting for input from the user. The <code>SMS with Twilio</code> integration sends a message request with <code>smsNoInputTurn</code> in the text field so that you can map this request to an intent in your dialog.	None
<code>terminateSession</code>	Ends the current SMS session. Use this command to ensure that the subsequent text message starts a new assistant-level session which does not retain any context values from the current session.	None
<code>smsActSendMedia</code>	Enables MMS messaging.	<code>mediaURL</code> : Specifies a JSON array of publicly accessible media URLs that are sent to the user.

Table 1. Actions that you can initiate from the dialog

Reserved context variables

The following table describes the context variables that have special meaning in the context of the `SMS with Twilio` integration. They should not be used for any purpose other than the documented use.

Table 2 describes the context variables that you can set by the `SMS with Twilio` integration.

Context variables that are set by the integration

Context variable name	Description
<code>assistant_phone_number</code>	The phone number associated with the assistant that received the text message.
<code>private.user_phone_number</code>	The phone number that the text message was received from.

SMS context variables set by the integration

Example:

```
{  
  "context": {  
    "global": {...},  
    "skills": {...},  
    "integrations": {  
      "text_messaging": {  
        "private":{  
          "user_phone_number":"+12223456789",  
        }  
        "assistant_phone_number":"+18883456789"  
      }  
    }  
  }  
}
```

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Managing your plan](#).

Managing your plan

This topic provides:

- A [plan information](#) reference
- Steps on [upgrading your plan](#)

Plan information

Billing for the use of Watson Assistant is managed through your IBM Cloud® account.

The metrics that are used for billing purposes differ based on your plan type. You can be billed based on the number of API calls made to a service instance or on the number of active users who interact with the instance.

For answers to common questions about subscriptions, see [How you're charged](#).

Explore the Watson Assistant [service plan options](#).

Paid plan features

The following features are available only to users of a Plus or Enterprise plan.

Plus

- [Phone integration](#)
- [Private endpoints](#)
- [Search](#)
- [v2 Logs API](#)
- [Log webhook](#)
- [Autolearning](#)
- [Intent conflict resolution](#)

The following features are available only to users of Enterprise plans.

Enterprise

- [Activity tracker](#)

The plan type of the service instance you are currently using is displayed in the page header. You can upgrade from one plan type to another. For more information, see [Upgrading](#).

User-based plans explained

Unlike API-based plans, which measure usage by the number of API calls made during a month, the Plus and Enterprise plans measure usage by the number of monthly active users.

A *monthly active user (MAU)* is any unique user who has at least one meaningful interaction with your assistant or custom application over the calendar billing month. A meaningful interaction is an exchange in which a user sends a request to your service and your service responds. Welcome messages that are displayed at the start of a conversation are not charged.

A unique user is recognized by the user ID that is associated with the person that interacts with your assistant. The web chat and other built-in integrations set this property for you automatically.

Specifying the user ID with the REST API

If you are using a custom client with the Watson Assistant API, you must set the `user_id` property in the message payload your client sends to the `message` method. The `user_id` property is specified at the root of the request body, as in this example:

```
$ {  
  "input": {  
    "message_type": "text",  
    "text": "I want to cancel my order"  
  },  
  "user_id": "my_user_id"  
}
```



Important: In some older SDK versions, the `user_id` property is not supported as a top-level method parameter. As an alternative, you can specify `user_id` within the nested `context.global.system` object.

For more information about the `user_id` property, see the API reference documentation:

- [v2 stateless /message](#)
- [v2 stateful /message](#)

If the user ID is not specified

If you are using a custom client application and do not set a `user_id` value, the service automatically sets it to one of the following values:

- **session_id (v2 API only)**: A property defined in the v2 API that identifies a single conversation between a user and the assistant. A session ID is provided in /message API calls that are generated by the built-in integrations. The session ends when a user closes the chat window or after the inactivity time limit is reached.



Note: If you use the stateless v2 message API, you must specify the `session_id` with each message in an ongoing conversation (in `context.global.session_id`).

- **conversation_id (v1 API only)**: A property defined in the v1 API that is stored in the context object of a /message API call. This property can be used to identify multiple /message API calls that are associated with a single conversational exchange with one user. However, the same ID is only used if you explicitly retain the ID and pass it back with each request that is made as part of the same conversation. Otherwise, a new ID is generated for each new /message API call.

For example, if the same person chats with your assistant on three separate occasions over the same billing period, how you represent that user in the API call impacts how the interactions are billed. If you identify the user interaction with a `user_id`, it counts as one use. If you identify the user interaction with a `session_id`, then it counts as three uses (because there is a separate session that is created for each interaction).

Design any custom applications to capture a unique `user_id` or `session_id` and pass the information to Watson Assistant. Choose a non-human-identifiable ID that doesn't change throughout the customer lifecycle. For example, don't use a person's email address as the user ID. In fact, the `user_id` syntax must meet the requirements for header fields as defined in [RFC 7230](#).

The built-in integrations derive the user ID in the following ways:

- For Facebook integrations, the `user_id` property is set to the sender ID that Facebook provides in its payload.
- For Slack integrations, the `user_id` property is a concatenation of the team ID, such as `T09LVDR7Y`, and the member ID of the user, such has `W4F8K9JNF`. For example: `T09LVDR7YW4F8K9JNF`.
- For web chat, you can set the value of the `user_id` property.

Billing is managed per monthly active user per service instance. If a single user interacts with assistants that are hosted by different service instances that belong to the same plan, each interaction is treated as a separate use. You are billed for the user's interaction with each service instance separately.

Test activity charges

Test messages that you send from the *Preview* button are charged. For the preview, a random `user_id` is generated and stored in a cookie. The multiple interactions that a single tester has with the assistant embedded in the preview are recognized as coming from a single user and are charged accordingly. If you are doing your own test, running a scripted regression test for example, use a single `user_id` for all of the calls within your regression test. Other uses are flagged as abuse.

Handling anonymous users

If your custom application or assistant interacts with users who are anonymous, you can generate a randomized universally unique ID to represent each anonymous user. For more information about UUIDs, see [RFC 4122](#).

- For web chat, if you do not pass an identifier for the user when the session begins, the web chat creates one for you. It creates a first-party cookie with a generated anonymous ID. The cookie remains active for 45 days. If the same user returns to your site later in the month and chats with your assistant again, the web chat integration recognizes the user. And you are charged only once when the same anonymous user interacts with your assistant multiple times in a single month.

If an anonymous user logs in and later is identified as being the same person who submitted a request with a known ID, you are charged twice. Each message with a unique user ID is charged as an independent active user. To avoid this situation, you can prompt users to log in before you initiate a chat or you can use the anonymous user ID to represent the user consistently.

Data centers

IBM Cloud has a network of global data centers that provide performance benefits to its cloud services. See [IBM Cloud global data centers](#) for more details.

You can create Watson Assistant service instances that are hosted in the following data center locations:

Location	Location code	API location
Dallas	us-south	N/A
Frankfurt	eu-de	fra
Seoul	kr-seo	seo

Sydney	au-syd	syd
Tokyo	jp-tok	tok
London	eu-gb	lon
Washington DC	us-east	wdc

Data center locations

Upgrading your plan

You can explore the Watson Assistant [service plan options](#) to decide which plan is best for you.

The page header shows the plan you are using today. To upgrade your plan, complete these steps:

1. Do one of the following things:

- **Trial plan only:** The number of days that are left in your trial is displayed in the page header. To upgrade your plan, click **Upgrade** from the page header before your trial period ends.
- For all other plan types, click **Manage** , and then choose **Upgrade** from the menu.

2. From here, you can see other available plan options. For most plan types, you can step through the upgrade process yourself.

- If you upgrade to an Enterprise with Data Isolation plan, you cannot do an in-place upgrade of your service instance. An Enterprise with Data Isolation plan instance must be provisioned for you first.
- When you upgrade from a legacy Standard plan, you change the metrics that are used for billing purposes. Instead of basing billing on API usage, the Plus plan bases billing on the number of monthly active users. If you built a custom app to deploy your assistant, you might need to update the app. Ensure that the API calls from the app include user ID information. For more information, see [User-based plans explained](#).
- You cannot change from a Trial plan to a Lite plan.

For answers to common questions about subscriptions, see the [How you're charged](#).



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

System entities

Learn about system entities that are provided by IBM for you to use out of the box. These built-in utility entities help your assistant recognize terms and references that are commonly used by customers in conversation, such as numbers and dates.

For information about how to add system entities to your dialog skill, see [Creating entities](#).



Note: In January 2020, a new version of the system entities was introduced. As of April 2021, only the new version of the system entities is supported for all languages. The option to switch to using the legacy version is no longer available.

For information about how to use contextual entities to identify people and location names, see the [Detecting Names And Locations With Watson Assistant](#) blog post on Medium.

@sys-currency entity

The `@sys-currency` system entity detects mentions of monetary currency values in user input. The currency can be expressed with a currency symbol or currency-specific terms. In either case, a number is returned.

Recognized formats

- 20 cents
- Five dollars
- \$10

Attributes

- `.literal`: Exact phrase in input that is interpreted to be a currency mention.
- `.numeric_value`: Canonical numeric value as an integer or a double, in base units.

- **.location**: Lists the index element values of the first and last letters in the text string for the phrase that is interpreted to be a currency mention.
- **.unit**: Base unit of currency specified as a 3-letter ISO currency code. For example, **USD** or **EUR**.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example (input = twenty dollars)	Example (input = \$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 examples

For the input **veinte euro** (**twenty euro** in Spanish) or **€1.234,56**, @sys-currency returns these values:

Attribute	Type	Input is veinte euro	Input is €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
@sys-currency.location	array	[0,11]	[0,9]
@sys-currency.unit	string	EUR	EUR

More @sys-currency examples

Equivalent results are returned for other supported languages and national currencies.

@sys-currency usage tips

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, the **null** in the condition is evaluated to false and the node is not processed. To check for currency values in a way that handles zeros properly, use the expression **@sys-currency OR @sys-currency == 0** in the node condition instead.

For more information about currencies that are recognized per language, see [Currency support](#).

@sys-date

The **@sys-date** system entity detects mentions of dates in user input. The date value is stored as a string in the format **yyyy-MM-dd**. For example, the mention **May 8** is stored as **"2020-05-08"**. The system augments missing elements of a date (such as the year for **"May 8"**) with the current date values.

If you select English as the skill language, the system uses US English (**en-us**) as the locale. For the US English locale only, the format of the date is **MM/DD/YYYY**. The format of the date changes to **DD/MM/YYYY** only if the first two numbers are greater than 12. The value that is stored has the format **yyyy-MM-dd**.

Recognized formats

- Friday
- today
- May 8

Attributes

- **.alternatives**: Saves alternative date values when the input is open to interpretation. Typically the future date is returned as the **@sys-date** value. If there is ambiguity, the past date is saved as an alternative date. For example, for the input **in January**, a future date is returned. However,

if it's February (January passed for this year), then the past January is returned also as an alternative date. An alternative date is also provided for weekdays when the weekday is specified without a modifier. For example, for the input `Are you open Wednesday?`, the future date is stored, but an alternative date with the date of the previous Wednesday is created also. Alternative dates are created for phrases that contain modifiers such as `next`, `this`, or `last`, which can have different meanings in different locales. The Watson Assistant service treats `last` and `next` dates as references to the most immediate last or next day that is referenced, which might be in either the same or a previous week. For some modifiers, an alternative date is created or not depending on the current day of the week. For example, maybe the input contains `this Wednesday`. If it's Monday, the future date is stored and no alternative date is created. However, if it's Wednesday, then the date is assumed to be the Wednesday of the coming week, but today's date is stored as an alternative date. Each alternative date is saved as an object that contains a value and confidence for each alternative date. The alternative date objects are stored in a JSON Array. To get an alternative value, use the syntax: `@sys-date.alternative`.

- `.calendar_type`: Specifies the calendar type for the timezone in use. `GREGORIAN` is the only supported type.
- `.datetime_link`: See [Date and time mentions](#).
- `.day`: Helper method that returns the day in the date.
- `.day_of_week`: Returns the day of the week in the date as a lowercase string.
- `.festival`: Recognizes locale-specific holiday names and can return the date of the upcoming holiday, such as `Thanksgiving`, `Christmas`, and `Halloween`. Use all lowercase letters if you need to specify a holiday name, in a condition for example `(@sys-date.festival == 'thanksgiving')`. For more information about recognized holidays, see [Festivals](#).
- `.granularity`: Recognizes time frame mentions. Options are `day`, `weekend`, `week`, `fortnight`, `month`, `quarter`, and `year`.
- `.interpretation`: The object that is returned by your assistant which contains new fields that increase the precision of the system entity classifier. You can omit `interpretation` from the expression that you use to refer to its fields. For example, you can access the value of the `festival` field in the interpretation object by using the shorthand syntax `<? @sys-date.festival ?>`.
- `.literal`: Exact phrase in input that is interpreted to be the date.
- `.location`: Index element values of the first and last letters in the text string for the phrase that is interpreted to be a date.
- `.month`: Helper method that returns the month in the date.
- `.range_link`: If present, indicates that the user's input contains a date range. The start and end index values of the string that includes the date range are saved as part of the link name. Additional information is provided, including the role that each `@sys-date` plays in the range relationship. For example, the start date has a role type of `date_from` and the end date has a role type of `date_to`. To check a role value, you can use the syntax `@sys-date.role?.type == 'date_from'`. If the user input implies a range, but only one date is specified, then the `range_link` property is not returned, but one role type is returned. For example, if the user asks, `Have you been open since yesterday`. Yesterday's date is recognized as the `@sys-date` mention, and a role of type `date_from` is returned for it. A `range_modifier` that identifies the word in the input that triggers the identification of a range is also returned. In this example, the modifier is `since`.
- `.relative_{timeframe}`: Recognizes and captures mentions of relative date values in the user's input. Returns a number that shows the number of units between now and the specified date. The {timeframe} units can be `year`, `month`, `week`, `weekend`, or `day`.
- `.specific_{timeframe}`: Recognizes and captures mentions of specific date units in the user's input. The {timeframe} units can be `year`, `quarter`, `month`, `day`, or `day_of_week`.
- `.year`: Helper method that returns the year in the date.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-date</code>	string	If the input is <code>May 13, 2020</code> , returns <code>2020-05-13</code> .
<code>@sys-date.alternatives</code>	array	For example, if today's date is <code>10 March 2019</code> , and the user enters <code>on March 1</code> , then the March 1st date for next year is saved as the <code>@sys-date (2020-03-01)</code> . However, the user might have meant March 1st of 2019. The system also saves this year's date (<code>2019-03-01</code>) as an alternative date value. As another example, if today is Tuesday and a user enters, <code>this Tuesday</code> or <code>next Tuesday</code> , then <code>@sys-date</code> is set to the date of Tuesday of next week and no alternative dates are created.
<code>@sys-date.calendar_type</code>	string	For any date mention, returns <code>GREGORIAN</code> .
<code>@sys-date.datetime_link</code>	string	N/A
<code>@sys-date.day</code>	string	If the input is <code>13 May 2020</code> , returns <code>13</code> .
<code>@sys-date.day_of_week</code>	string	If the input is <code>13 May 2020</code> and it's a Wednesday, returns <code>wednesday</code> .
<code>@sys-date.festival</code>	string	If the input contains <code>Thanksgiving Day</code> , then <code>2020-11-26</code> is returned.
<code>@sys-date.granularity</code>	string	If the input mentions <code>next year</code> , returns <code>year</code> .
<code>@sys-date.literal</code>	string	If the input is, <code>I plan to leave on Saturday</code> ., then <code>on Saturday</code> is returned.
<code>@sys-date.location</code>	array	If the input is <code>My vacation starts tomorrow</code> ., returns <code>[19, 27]</code> .

@sys-date.month	string	If the input is <code>13 May 2020</code> , returns <code>5</code> .
@sys-date.range_link	string	For example, the input might be <code>I will travel from March 15 to March 22</code> . Each @sys-date entity that is detected has a <code>range_link</code> property in its output that is named <code>date_range_14_39</code> where the location of the date range text is <code>[14, 39]</code> . The start date (<code>2020-03-15</code>) has a role type of <code>date_from</code> . The end date (<code>2020-03-22</code>) has a role type of <code>date_to</code> .
@sys-date.relative_{timeframe}	number	If the input is <code>two weeks ago</code> , returns <code>relative_week = -2</code> . If the input is <code>this weekend</code> , returns <code>relative_weekend = 0</code> . If the input is <code>today</code> , returns <code>relative_day = 0</code> .
@sys-date.specific_{timeframe}	string	For <code>2020</code> , returns <code>specific_year = 2020</code> . For <code>Q2 2020</code> or <code>second quarter 2020</code> , return <code>specific_quarter = 2</code> . For <code>March</code> , returns <code>specific_month = 3</code> . For <code>Monday</code> , returns <code>specific_day_of_week = monday</code> . For <code>May 3rd</code> , returns <code>specific_day = 3</code> .
@sys-date.year	string	If the input is <code>13 May 2020</code> , returns <code>2020</code> .

@sys-date attributes

@sys-time

The `@sys-time` system entity detects mentions of times in user input. Returns the time that is specified in user input in the format `HH:mm:ss`. For example, `"13:00:00"` for `1pm`.

Recognized formats

- `2pm`
- `at 4`
- `15:30`

Attributes

- `.alternatives`: Captures times other than the one saved as the `@sys-time` value that the user might have meant when the time is not clearly indicated. The alternative values are saved as an object that contains a value and confidence for each alternative time and is stored in a JSON Array. To get an alternative value, use the syntax: `@sys-time.alternative`.
- `.calendar_type`: Specifies the calendar type for the timezone in use. `GREGORIAN` is the only supported type.
- `.granularity`: Recognizes mentions of time frames. Options are `hour`, `minute`, `second` and `instant`.
- `.hour`: Helper method that returns the hour that is specified in the time as a numeric value.
- `.interpretation`: The object that is returned by your assistant which contains new fields that increase the precision of the system entity classifier. You can omit `interpretation` from the expression that you use to refer to its fields. For example, you can access the value of the `granularity` field in the interpretation object by using the shorthand syntax `<? @sys-time.granularity ?>`.
- `.literal`: Exact phrase in input that is interpreted to be the time.
- `.location`: Lists the index element values of the first and last letters in the text string for the phrase that is interpreted to be a time mention.
- `.minute`: Helper method that returns the minute value that is specified in the time.
- `.part_of_day`: Recognizes terms that represent the time of day, such as `morning`, `afternoon`, `evening`, `night`, or `now`. Also sets a time range for each part of the day. The response contains two `@sys-time` values, one for the start and one for the end of the time range. The entities array contains a `range_link` object with `time-from` and `time-to` role types. The time ranges for different parts of the day can differ by locale. For US English, morning is `6:00:00` to `12:00:00`, afternoon is `12:00:00` to `18:00:00`, evening is `18:00:00` to `22:00:00`, and night is `22:00:00` to `23:59:59`. Night ends just before midnight because otherwise it would overlap with a new day, which is measured by `@sys-date`.
- `.range_link`: If present, indicates that the user's input contains a time range. The start and end index values of the string that includes the time range are saved as part of the link name. Additional information is provided, including the role that each `@sys-time` plays in the range relationship. For example, the start time has a role type of `time_from` and the end time has a role type of `time_to`. To check a role value, you can use the syntax `@sys-time.role?.type == 'time_from'`. If the user input implies a range, but only one time is specified, then the `range_link` property is not returned, but one role type is returned. A `range_modifier` that identifies the word in the input that triggers the identification of a range is also returned.
- `.relative_{timeframe}`: Recognizes relative mentions of time and returns a number that shows the number of units between now and the specified time. The `{timeframe}` units can be `hour`, `minute`, or `second`.
- `.second`: Helper method that returns the second value that is specified in the time.
- `.specific_{timeframe}`: Recognizes and captures mentions of specific time units in the user input. The `{timeframe}` units can be `hour`, `minute`, or `second`.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-time</code>	string	For the input <code>6:30 PM</code> , returns <code>18:30:00</code> .

@sys-time.alternatives	array	If a user says The meeting is at 5 , the system calculates the time as 5:00:00 or 5 AM and saves that as the @sys-time. However, the user might have meant 17:00:00 or 5 PM . The system also saves the later time as an alternative time value. If the user enters The meeting is at 5pm , then @sys-time is set to 17:00:00 , and no alternative value is created because there is no AM and PM confusion.
@sys-time.calendar_type	string	For any time mention, returns GREGORIAN .
@sys-time.granularity	string	If the input is now , returns instant . For 3 o'clock or noon , returns hour . For 17:00:00 , returns second .
@sys-time.hour	number	If the input contains 5:30:10 PM , it returns 17 to represent 5 PM.
@sys-time.literal	string	For the input, The store closes at 8PM , returns at 8PM .
@sys-time.location	array	For the input, The store closes at 8PM , returns [17, 23] .
@sys-time.minute	number	If the input mentions the time 5:30:10 PM , returns 30 . If no minutes are specified, then this helper returns 0 .
@sys-time.part_of_day	string	If the input is this morning , returns two @sys-time values, 6:00:00 and 12:00:00 . Also returns a range_link object with time-from and time-to role types.
@sys-time.range_link	string	For the input Are you open from 9AM to 11AM , two @sys-time entities are detected. Each @sys-time entity that is detected has a range_link property in its output. If a range is implied but only one time value is provided, then only one @sys-time value is returned. For the input Are you open until 9PM , 9PM is recognized as the @sys-time mention, and a role of type time_to is returned for it. The range_modifier in this example is until .
@sys-time.relative_{timeframe}	number	For 5 hours ago , returns relative_hour = -5 . For in two minutes , returns relative_minute = 2 . For in a second , returns relative_second = 1 .
@sys-time.second	number	If the input mentions the time 5:30:10 PM , returns 10 . If no second value is specified, then this helper returns 0 .
@sys-time.specific_{timeframe}	string	For at 5 o'clock , returns specific_hour = 5 . For at 2:30 , returns specific_minute = 30 . For 23:30:22 , returns specific_second = 22 .

@sys-time attributes

Date and time mentions

Some user input contains information that includes both date and time information. Your assistant captures both values and saves them as two separate entity mentions, one @sys-date and one @sys-time.

The relationship between the date and time values is established in two ways:

- Each entity that participates in the relationship contains the same **datetime_link** attribute.
- The literal string that spans the complete date and time mention, when they are mentioned together in input, is the same for both entities. The location information is appended to the **datetime_link** name.

Recognized formats

- now
- two hours from now
- on Monday at 4pm

Attributes

- **.datetime_link**: If present, indicates that the user's input mentions a date and time together, which implies that the date and time are related to one another. The start and end index values of the string that includes the date and time are saved as part of the link name.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
-----------	------	---------

<code>@sys-date.datetime_link</code> and <code>@sys-time.datetime_link</code>	string	If the input is, <code>Are you open today at 5?</code> , then <code>today</code> is recognized as a date mentionn(<code>2020-05-21</code>) at location <code>[13,18]</code> and <code>at 5</code> is recognized as a time mention (<code>05:00:00</code>) at location <code>[19,23]</code> . A location value of <code>[13,23]</code> , which spans both the date and time mentions, is appended to the resulting <code>datetime_link</code> . It is named <code>datetime_link_13_23</code> and it is included in the output of the <code>@sys-date</code> and <code>@sys-time</code> entities that participate in the relationship.
--	--------	--

Date and time attributes

Time zones

Mentions of a date or time that are relative to the current time are resolved for a chosen time zone. By default, the time zone is Greenwich mean time. Therefore, REST API clients that are located in different time zones get the current Coordinated Universal Time when `now` is mentioned in input.

Optional, the REST API client can add the local time zone as the context variable `$timezone`. This context variable must be sent with every client request. For example, the `$timezone` value can 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.



Tip: For information about processing date and time values, see the [Date and time method reference](#).

@sys-number entity

The `@sys-number` system entity detects mentions of numbers in user input. The number can be written with either numerals or words. In either case, a number is returned.

Recognized formats

- 21
- twenty one
- 3.13

Attributes

- `.literal`: Exact phrase in input that is interpreted to be the number.
- `.location`: Index element values of the first and last letters in the text string that is interpreted to be a number.
- `.numeric_value`: Canonical numeric value as an integer or a double.
- `.range_link`: If present, indicates that the user's input contains a number range. The location value of the text that specifies the range is appended to the link name. Additional information is provided, including the role that each `@sys-number` plays in the range relationship. For example, the start number has a role type of `number_from` and the end number has a role type of `number_to`. To check a role value, you can use the syntax `@sys-number.role?.type == 'number_from'`.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-number</code>	string	If the input is <code>twenty</code> , returns <code>20</code> . If the input is <code>1,234.56</code> , returns <code>1234.56</code> .
<code>@sys-number.literal</code>	string	If the input is <code>twenty</code> , returns <code>twenty</code> . If the input is <code>1,234.56</code> returns <code>1,234.56</code> .
<code>@sys-number.location</code>	array	If the input is <code>twenty</code> , returns <code>[0,6]</code> . If the input is <code>1,234.56</code> , returns <code>[0,8]</code>
<code>@sys-number.numeric_value</code>	number	If the input is <code>twenty</code> , returns <code>20</code> . If the input is <code>1,234.56</code> , returns <code>1234.56</code>
<code>@sys-number.range_link</code>	string	If the input is <code>I'm interested in 5 to 7..</code> Each of the two <code>@sys-number</code> system entities that are detected have a <code>range_link</code> of <code>number_range_18_24</code> in their output. The first <code>@sys-number</code> entity (5) has a <code>number_from</code> role type. The last <code>@sys-number</code> entity (7) has a <code>number_to</code> role type.

@sys-number examples

@sys-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. However, the 0 is interpreted as a `null` value for the condition, which results in the node not being processed. To check for numbers in a way that

handles zeros properly, use the expression `@sys-number == 0` in the node condition also. The full expression to use is `@sys-number OR @sys-number == 0`.

- 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. Your comparison might evaluate 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 OR @sys-number == 0) AND @sys-number < 4` instead. If no number is present, the first condition evaluates to false. As a result, the whole condition evaluates to false.



Tip: For more information about processing number values, see the [Numbers method reference](#).

@sys-percentage entity

The `@sys-percentage` system entity detects mentions of percentages in user input. The percentage can be 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

Attributes

- `.literal`: Exact phrase in input that is interpreted to be a percentage.
- `.location`: Index element values of the first and last letters in the text string that is interpreted to be a percentage mention.
- `.numeric_value`: Canonical numeric value as an integer or a double
- `.range_link`: If present, indicates that the user's input contains a range, such as `from 2 to 3 percent`. The location value of the text that specifies the range is appended to the link name. Additional information is provided, including the role that each `@sys-percentage` plays in the range relationship. For example, the start number has a role type of `number_from` and the end number has a role type of `number_to`. To check a role value, you can use the syntax `@sys-percentage.role?.type == 'number_from'`.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-percentage</code>	string	If input contains <code>1,234.56%</code> , returns <code>1234.56</code> .
<code>@sys-percentage.literal</code>	string	If input contains <code>50 percent</code> returns <code>50 percent</code> .
<code>@sys-percentage.location</code>	array	If input contains <code>from 2 to 3 percent</code> , returns <code>[0,19]</code> .
<code>@sys-percentage.numeric_value</code>	number	If input contains <code>50 percent</code> returns <code>50</code> .
<code>@sys-percentage.range_link</code>	string	If input contains <code>from 2 to 3 percent</code> , returns <code>number_range_0_19</code> .

@sys-percentage examples

@sys-percentage usage tips

- 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%. As a result, the `null` in the condition is evaluated to false and the node is not processed. To check for percentages in a way that handles zero percentages properly, use the expression `@sys-percentage OR @sys-percentage == 0` in the node condition instead.
- If you input a value like `1-2%`, the values `1%` and `2%` are returned as system entities.

Using system entities in conditions

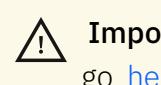
In a node that conditions on the `#Customer_Care_Store_Hours` intent, you can add conditional responses that use new system entity properties to provide slightly different answers about store hours depending on what the user asks.



Tip: You probably do not want to use all of these conditional responses in a real dialog; they are described here merely to illustrate what's possible.

Conditional response condition	Description	Example response text
syntax		
<code>@sys-date.festival == 'thanksgiving'</code>	Checks whether the customer is asking about hours on Thanksgiving day in particular.	We give out free meals to those in need on Thanksgiving Day.
<code>@sys-date.festival</code>	Checks whether the customer is asking about hours on another specific holiday.	We are closed on Christmas Day, July 4th, and President's Day. On other holidays, we are open from 10AM to 5PM.
<code>@sys-time.part_of_day == 'night'</code>	Checks whether the user includes any terms that mention night as the time of day in the input.	We are not open late; we close at 9PM most days.
<code>@sys-date.datetime_link && input.text.contains('today') && now().sameOrAfter(@sys-time)</code>	Checks whether the input contains a phrase such as <code>today at 8</code> . It also checks whether the <code>@sys-time</code> that is detected is earlier than the current time of day. You can add a context variable to the conditional response that creates a <code>\$new_time</code> variable with the value <code><? @sys-time.plusHours(12) ?></code> . You can use this approach to create a date variable that captures the more likely time meant by a user who, at noontime asks, <code>Are you open today at 8</code> . The system assumes the user means 8AM instead of 8PM.	You mean today at \$new_time, correct?
<code>@sys-time.range_link && entities[0].role?.type == 'time_from' && entities[1].role?.type == 'time_to'</code>	Checks whether the input contains a time range. The condition also makes sure that the first <code>@sys-time</code> system entity listed in the entities array is the start time and that the second one is the end time before including them in the response.	You want to know if we're open between <code><? entities[0] ?></code> and <code><? entities[1] ?></code> , is that correct?
<code>@sys-time && entities.size() < 2 && entities[0].role?.type == 'time_to'</code>	Checks whether the input contains one <code>@sys-time</code> mention that has a <code>time_to</code> role type. If the user input matches this condition, it suggests that the user specified the end time of an open-ended time range. For example, the response would be triggered by the input, <code>Are you open until 9pm?</code> . This input matches because it identifies only one time in a time span, and the mention has a <code>time_to</code> role.	Do you mean from now (<code><? now().reformatDateTime('h:mm a') ?></code>) until <code><? @sys-time.reformatDateTime('h:mm a') ?></code> ?
<code>@sys-date.day_of_week == 'sunday'</code>	Checks whether a specific date that the user is asking about falls on a Sunday.	We are closed on Sundays.
<code>@sys-date.specific_day_of_week == 'monday' && @sys-date.alternative</code>	Checks whether the user mentioned the weekday <code>Monday</code> in their query. For example, <code>Are you open Monday</code> . The condition also checks whether any alternative dates were stored. Alternative dates are created when your assistant isn't entirely sure which Monday the user means, so it stores the dates of alternative Mondays also. When a user specifies a weekday, your assistant assumes that the user means the future occurrence of the day (the coming Monday). You can add a response that double checks the intended date, by using the detected alternative value. In this case, the alternative date is the previous Monday's date.	Do you mean <code>@sys-date</code> or <code>@sys-date.alternative</code> ?
<code>true</code>	Responds to any other requests for store hour information.	We're open from 9AM to 9PM Monday through Saturday.

System entities in conditional responses



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

System entities

Learn about system entities that are provided by IBM for you to use out of the box. These built-in utility entities help your assistant recognize terms and

references that are commonly used by customers in conversation, such as numbers and dates.

For information about how to add system entities to your dialog skill, see [Creating entities](#).



Note: In January 2020, a new version of the system entities was introduced. As of April 2021, only the new version of the system entities is supported for all languages. The option to switch to using the legacy version is no longer available.

For information about how to use contextual entities to identify people and location names, see the [Detecting Names And Locations With Watson Assistant](#) blog post on Medium.

@sys-currency entity

The `@sys-currency` system entity detects mentions of monetary currency values in user input. The currency can be expressed with a currency symbol or currency-specific terms. In either case, a number is returned.

Recognized formats

- 20 cents
- Five dollars
- \$10

Attributes

- `.literal`: Exact phrase in input that is interpreted to be a currency mention.
- `.numeric_value`: Canonical numeric value as an integer or a double, in base units.
- `.location`: Lists the index element values of the first and last letters in the text string for the phrase that is interpreted to be a currency mention.
- `.unit`: Base unit of currency specified as a 3-letter ISO currency code. For example, `USD` or `EUR`.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example (input = twenty dollars)	Example (input = \$1,234.56)
<code>@sys-currency</code>	string	<code>20</code>	<code>1234.56</code>
<code>@sys-currency.literal</code>	string	<code>twenty dollars</code>	<code>\$1,234.56</code>
<code>@sys-currency.numeric_value</code>	number	<code>20</code>	<code>1234.56</code>
<code>@sys-currency.location</code>	array	<code>[0,14]</code>	<code>[0,9]</code>
<code>@sys-currency.unit</code>	string	<code>USD</code>	<code>USD</code>

@sys-currency examples

For the input `veinte euro` (`twenty euro` in Spanish) or `€1.234,56`, `@sys-currency` returns these values:

Attribute	Type	Input is veinte euro	Input is €1.234,56
<code>@sys-currency</code>	string	<code>20</code>	<code>1234.56</code>
<code>@sys-currency.literal</code>	string	<code>veinte euro</code>	<code>€1.234,56</code>
<code>@sys-currency.numeric_value</code>	number	<code>20</code>	<code>1234.56</code>
<code>@sys-currency.location</code>	array	<code>[0,11]</code>	<code>[0,9]</code>
<code>@sys-currency.unit</code>	string	<code>EUR</code>	<code>EUR</code>

More @sys-currency examples

Equivalent results are returned for other supported languages and national currencies.

@sys-currency usage tips

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, the `null` in the condition is evaluated to false and the node is not processed. To check for currency values in a way that handles zeros properly, use the expression `@sys-currency OR @sys-currency == 0` in the node condition instead.

For more information about currencies that are recognized per language, see [Currency support](#).

@sys-date

The `@sys-date` system entity detects mentions of dates in user input. The date value is stored as a string in the format `yyyy-MM-dd`. For example, the mention `May 8` is stored as `"2020-05-08"`. The system augments missing elements of a date (such as the year for `"May 8"`) with the current date values.

If you select English as the skill language, the system uses US English (`en-us`) as the locale. For the US English locale only, the format of the date is `MM/DD/YYYY`. The format of the date changes to `DD/MM/YYYY` only if the first two numbers are greater than 12. The value that is stored has the format `yyyy-MM-dd`.

Recognized formats

- Friday
- today
- May 8

Attributes

- `.alternatives`: Saves alternative date values when the input is open to interpretation. Typically the future date is returned as the `@sys-date` value. If there is ambiguity, the past date is saved as an alternative date. For example, for the input `in January`, a future date is returned. However, if it's February (January passed for this year), then the past January is returned also as an alternative date. An alternative date is also provided for weekdays when the weekday is specified without a modifier. For example, for the input `Are you open Wednesday?`, the future date is stored, but an alternative date with the date of the previous Wednesday is created also. Alternative dates are created for phrases that contain modifiers such as `next`, `this`, or `last`, which can have different meanings in different locales. The Watson Assistant service treats `last` and `next` dates as references to the most immediate last or next day that is referenced, which might be in either the same or a previous week. For some modifiers, an alternative date is created or not depending on the current day of the week. For example, maybe the input contains `this Wednesday`. If it's Monday, the future date is stored and no alternative date is created. However, if it's Wednesday, then the date is assumed to be the Wednesday of the coming week, but today's date is stored as an alternative date. Each alternative date is saved as an object that contains a value and confidence for each alternative date. The alternative date objects are stored in a JSON Array. To get an alternative value, use the syntax: `@sys-date.alternative`.
- `.calendar_type`: Specifies the calendar type for the timezone in use. `GREGORIAN` is the only supported type.
- `.datetime_link`: See [Date and time mentions](#).
- `.day`: Helper method that returns the day in the date.
- `.day_of_week`: Returns the day of the week in the date as a lowercase string.
- `.festival`: Recognizes locale-specific holiday names and can return the date of the upcoming holiday, such as `Thanksgiving`, `Christmas`, and `Halloween`. Use all lowercase letters if you need to specify a holiday name, in a condition for example `(@sys-date.festival == 'thanksgiving')`. For more information about recognized holidays, see [Festivals](#).
- `.granularity`: Recognizes time frame mentions. Options are `day`, `weekend`, `week`, `fortnight`, `month`, `quarter`, and `year`.
- `.interpretation`: The object that is returned by your assistant which contains new fields that increase the precision of the system entity classifier. You can omit `interpretation` from the expression that you use to refer to its fields. For example, you can access the value of the `festival` field in the interpretation object by using the shorthand syntax `<? @sys-date.festival ?>`.
- `.literal`: Exact phrase in input that is interpreted to be the date.
- `.location`: Index element values of the first and last letters in the text string for the phrase that is interpreted to be a date.
- `.month`: Helper method that returns the month in the date.
- `.range_link`: If present, indicates that the user's input contains a date range. The start and end index values of the string that includes the date range are saved as part of the link name. Additional information is provided, including the role that each `@sys-date` plays in the range relationship. For example, the start date has a role type of `date_from` and the end date has a role type of `date_to`. To check a role value, you can use the syntax `@sys-date.role?.type == 'date_from'`. If the user input implies a range, but only one date is specified, then the `range_link` property is not returned, but one role type is returned. For example, if the user asks, `Have you been open since yesterday`. Yesterday's date is recognized as the `@sys-date` mention, and a role of type `date_from` is returned for it. A `range_modifier` that identifies the word in the input that triggers the identification of a range is also returned. In this example, the modifier is `since`.
- `.relative_{timeframe}`: Recognizes and captures mentions of relative date values in the user's input. Returns a number that shows the number of units between now and the specified date. The {timeframe} units can be `year`, `month`, `week`, `weekend`, or `day`.
- `.specific_{timeframe}`: Recognizes and captures mentions of specific date units in the user's input. The {timeframe} units can be `year`, `quarter`, `month`, `day`, or `day_of_week`.
- `.year`: Helper method that returns the year in the date.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-date</code>	string	If the input is <code>May 13, 2020</code> , returns <code>2020-05-13</code> .

@sys-date.alternatives	array	For example, if today's date is 10 March 2019 , and the user enters on March 1 , then the March 1st date for next year is saved as the @sys-date (2020-03-01). However, the user might have meant March 1st of 2019. The system also saves this year's date (2019-03-01) as an alternative date value. As another example, if today is Tuesday and a user enters, this Tuesday or next Tuesday , then @sys-date is set to the date of Tuesday of next week and no alternative dates are created.
@sys-date.calendar_type	string	For any date mention, returns GREGORIAN .
@sys-date.datetime_link	string	N/A
@sys-date.day	string	If the input is 13 May 2020 , returns 13 .
@sys-date.day_of_week	string	If the input is 13 May 2020 and it's a Wednesday, returns wednesday .
@sys-date.festival	string	If the input contains Thanksgiving Day , then 2020-11-26 is returned.
@sys-date.granularity	string	If the input mentions next year , returns year .
@sys-date.literal	string	If the input is, I plan to leave on Saturday. , then on Saturday is returned.
@sys-date.location	array	If the input is My vacation starts tomorrow. , returns [19, 27] .
@sys-date.month	string	If the input is 13 May 2020 , returns 5 .
@sys-date.range_link	string	For example, the input might be I will travel from March 15 to March 22 . Each @sys-date entity that is detected has a range_link property in its output that is named date_range_14_39 where the location of the date range text is [14, 39] . The start date (2020-05-15) has a role type of date_from . The end date (2020-05-22) has a role type of date_to .
@sys-date.relative_{timeframe}	number	If the input is two weeks ago , returns relative_week = -2 . If the input is this weekend , returns relative_weekend = 0 . If the input is today , returns relative_day = 0 .
@sys-date.specific_{timeframe}	string	For 2020 , returns specific_year = 2020 . For Q2 2020 or second quarter 2020 , return specific_quarter = 2 . For March , returns specific_month = 3 . For Monday , returns specific_day_of_week = monday . For May 3rd , returns specific_day = 3 .
@sys-date.year	string	If the input is 13 May 2020 , returns 2020 .

@sys-date attributes

@sys-time

The **@sys-time** system entity detects mentions of times in user input. Returns the time that is specified in user input in the format **HH:mm:ss**. For example, **"13:00:00"** for **1pm**.

Recognized formats

- 2pm
- at 4
- 15:30

Attributes

- **.alternatives**: Captures times other than the one saved as the @sys-time value that the user might have meant when the time is not clearly indicated. The alternative values are saved as an object that contains a value and confidence for each alternative time and is stored in a JSON Array. To get an alternative value, use the syntax: **@sys-time.alternative**.
- **.calendar_type**: Specifies the calendar type for the timezone in use. **GREGORIAN** is the only supported type.
- **.granularity**: Recognizes mentions of time frames. Options are **hour**, **minute**, **second** and **instant**.
- **.hour**: Helper method that returns the hour that is specified in the time as a numeric value.
- **.interpretation**: The object that is returned by your assistant which contains new fields that increase the precision of the system entity classifier. You can omit **interpretation** from the expression that you use to refer to its fields. For example, you can access the value of the **granularity** field in the interpretation object by using the shorthand syntax **<? @sys-time.granularity ?>**.
- **.literal**: Exact phrase in input that is interpreted to be the time.

- **.location**: Lists the index element values of the first and last letters in the text string for the phrase that is interpreted to be a time mention.
- **.minute**: Helper method that returns the minute value that is specified in the time.
- **.part_of_day**: Recognizes terms that represent the time of day, such as **morning**, **afternoon**, **evening**, **night**, or **now**. Also sets a time range for each part of the day. The response contains two **@sys-time** values, one for the start and one for the end of the time range. The entities array contains a **range_link** object with **time-from** and **time-to** role types. The time ranges for different parts of the day can differ by locale. For US English, morning is **6:00:00** to **12:00:00**, afternoon is **12:00:00** to **18:00:00**, evening is **18:00:00** to **22:00:00**, and night is **22:00:00** to **23:59:59**. Night ends just before midnight because otherwise it would overlap with a new day, which is measured by **@sys-date**.
- **.range_link**: If present, indicates that the user's input contains a time range. The start and end index values of the string that includes the time range are saved as part of the link name. Additional information is provided, including the role that each **@sys-time** plays in the range relationship. For example, the start time has a role type of **time_from** and the end time has a role type of **time_to**. To check a role value, you can use the syntax **@sys-time.role?.type == 'time_from'**. If the user input implies a range, but only one time is specified, then the **range_link** property is not returned, but one role type is returned. A **range_modifier** that identifies the word in the input that triggers the identification of a range is also returned.
- **.relative_{timeframe}**: Recognizes relative mentions of time and returns a number that shows the number of units between now and the specified time. The {timeframe} units can be **hour**, **minute**, or **second**.
- **.second**: Helper method that returns the second value that is specified in the time.
- **.specific_{timeframe}**: Recognizes and captures mentions of specific time units in the user input. The {timeframe} units can be **hour**, **minute**, or **second**.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-time</code>	string	For the input 6:30 PM , returns 18:30:00 .
<code>@sys-time.alternatives</code>	array	If a user says The meeting is at 5 , the system calculates the time as 5:00:00 or 5 AM and saves that as the <code>@sys-time</code> . However, the user might have meant 17:00:00 or 5 PM . The system also saves the later time as an alternative time value. If the user enters The meeting is at 5pm , then <code>@sys-time</code> is set to 17:00:00 , and no alternative value is created because there is no AM and PM confusion.
<code>@sys-time.calendar_type</code>	string	For any time mention, returns GREGORIAN .
<code>@sys-time.granularity</code>	string	If the input is now , returns instant . For 3 o'clock or noon , returns hour . For 17:00:00 , returns second .
<code>@sys-time.hour</code>	number	If the input contains 5:30:10 PM , it returns 17 to represent 5 PM.
<code>@sys-time.literal</code>	string	For the input, The store closes at 8PM , returns at 8PM .
<code>@sys-time.location</code>	array	For the input, The store closes at 8PM , returns [17, 23] .
<code>@sys-time.minute</code>	number	If the input mentions the time 5:30:10 PM , returns 30 . If no minutes are specified, then this helper returns 0 .
<code>@sys-time.part_of_day</code>	string	If the input is this morning , returns two <code>@sys-time</code> values, 6:00:00 and 12:00:00 . Also returns a range_link object with time-from and time-to role types.
<code>@sys-time.range_link</code>	string	For the input Are you open from 9AM to 11AM , two <code>@sys-time</code> entities are detected. Each <code>@sys-time</code> entity that is detected has a range_link property in its output. If a range is implied but only one time value is provided, then only one <code>@sys-time</code> value is returned. For the input Are you open until 9PM , 9PM is recognized as the <code>@sys-time</code> mention, and a role of type time_to is returned for it. The range_modifier in this example is until .
<code>@sys-time.relative_{timeframe}</code>	number	For 5 hours ago , returns relative_hour = -5 . For in two minutes , returns relative_minute = 2 . For in a second , returns relative_second = 1 .
<code>@sys-time.second</code>	number	If the input mentions the time 5:30:10 PM , returns 10 . If no second value is specified, then this helper returns 0 .
<code>@sys-time.specific_{timeframe}</code>	string	For at 5 o'clock , returns specific_hour = 5 . For at 2:30 , returns specific_minute = 30 . For 23:30:22 , returns specific_second = 22 .

`@sys-time` attributes

Date and time mentions

Some user input contains information that includes both date and time information. Your assistant captures both values and saves them as two separate entity mentions, one `@sys-date` and one `@sys-time`.

The relationship between the date and time values is established in two ways:

- Each entity that participates in the relationship contains the same `datetime_link` attribute.
- The literal string that spans the complete date and time mention, when they are mentioned together in input, is the same for both entities. The location information is appended to the `datetime_link` name.

Recognized formats

- now
- two hours from now
- on Monday at 4pm

Attributes

- `.datetime_link`: If present, indicates that the user's input mentions a date and time together, which implies that the date and time are related to one another. The start and end index values of the string that includes the date and time are saved as part of the link name.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
<code>@sys-date.datetime_link</code> and <code>@sys-time.datetime_link</code>	string	If the input is, <code>Are you open today at 5?</code> , then <code>today</code> is recognized as a date mention (<code>2020-05-21</code>) at location <code>[13, 18]</code> and <code>at 5</code> is recognized as a time mention (<code>05:00:00</code>) at location <code>[19, 23]</code> . A location value of <code>[13, 23]</code> , which spans both the date and time mentions, is appended to the resulting <code>datetime_link</code> . It is named <code>datetime_link_13_23</code> and it is included in the output of the <code>@sys-date</code> and <code>@sys-time</code> entities that participate in the relationship.

Date and time attributes

Time zones

Mentions of a date or time that are relative to the current time are resolved for a chosen time zone. By default, the time zone is Greenwich mean time. Therefore, REST API clients that are located in different time zones get the current Coordinated Universal Time when `now` is mentioned in input.

Optionally, the REST API client can add the local time zone as the context variable `$timezone`. This context variable must be sent with every client request. For example, the `$timezone` value can 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.



Tip: For information about processing date and time values, see the [Date and time method reference](#).

@sys-number entity

The `@sys-number` system entity detects mentions of numbers in user input. The number can be written with either numerals or words. In either case, a number is returned.

Recognized formats

- 21
- twenty one
- 3.13

Attributes

- `.literal`: Exact phrase in input that is interpreted to be the number.
- `.location`: Index element values of the first and last letters in the text string that is interpreted to be a number.
- `.numeric_value`: Canonical numeric value as an integer or a double.
- `.range_link`: If present, indicates that the user's input contains a number range. The location value of the text that specifies the range is appended to the link name. Additional information is provided, including the role that each `@sys-number` plays in the range relationship. For example, the start number has a role type of `number_from` and the end number has a role type of `number_to`. To check a role value, you can use the syntax `@sys-number.role?.type == 'number_from'`.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
@sys-number	string	If the input is <code>twenty</code> , returns <code>20</code> . If the input is <code>1,234.56</code> , returns <code>1234.56</code> .
@sys-number.literal	string	If the input is <code>twenty</code> , returns <code>twenty</code> . If the input is <code>1,234.56</code> returns <code>1,234.56</code> .
@sys-number.location	array	If the input is <code>twenty</code> , returns <code>[0,6]</code> . If the input is <code>1,234.56</code> , returns <code>[0,8]</code>
@sys-number.numeric_value	number	If the input is <code>twenty</code> , returns <code>20</code> . If the input is <code>1,234.56</code> , returns <code>1234.56</code>
@sys-number.range_link	string	If the input is <code>I'm interested in 5 to 7..</code> Each of the two <code>@sys-number</code> system entities that are detected have a <code>range_link</code> of <code>number_range_18_24</code> in their output. The first <code>@sys-number</code> entity (5) has a <code>number_from</code> role type. The last <code>@sys-number</code> entity (7) has a <code>number_to</code> role type.

@sys-number examples

@sys-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. However, the 0 is interpreted as a `null` value for the condition, which results in the node not being processed. To check for numbers in a way that handles zeros properly, use the expression `@sys-number == 0` in the node condition also. The full expression to use is `@sys-number OR @sys-number == 0`.
- 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. Your comparison might evaluate 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 OR @sys-number == 0) AND @sys-number < 4` instead. If no number is present, the first condition evaluates to false. As a result, the whole condition evaluates to false.

 **Tip:** For more information about processing number values, see the [Numbers method reference](#).

@sys-percentage entity

The `@sys-percentage` system entity detects mentions of percentages in user input. The percentage can be 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`

Attributes

- `.literal`: Exact phrase in input that is interpreted to be a percentage.
- `.location`: Index element values of the first and last letters in the text string that is interpreted to be a percentage mention.
- `.numeric_value`: Canonical numeric value as an integer or a double
- `.range_link`: If present, indicates that the user's input contains a range, such as `from 2 to 3 percent`. The location value of the text that specifies the range is appended to the link name. Additional information is provided, including the role that each `@sys-percentage` plays in the range relationship. For example, the start number has a role type of `number_from` and the end number has a role type of `number_to`. To check a role value, you can use the syntax `@sys-percentage.role?.type == 'number_from'`.

The following table illustrates the information that each attribute captures from the user input.

Attribute	Type	Example
@sys-percentage	string	If input contains <code>1,234.56%</code> , returns <code>1234.56</code> .
@sys-percentage.literal	string	If input contains <code>50 percent</code> returns <code>50 percent</code> .
@sys-percentage.location	array	If input contains <code>from 2 to 3 percent</code> , returns <code>[0,19]</code> .

@sys-percentage.numeric_value	number	If input contains 50 percent returns 50.
@sys-percentage.range_link	string	If input contains from 2 to 3 percent, returns number_range_0_19.

@sys-percentage examples

@sys-percentage usage tips

- 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%. As a result, the null in the condition is evaluated to false and the node is not processed. To check for percentages in a way that handles zero percentages properly, use the expression @sys-percentage OR @sys-percentage == 0 in the node condition instead.
- If you input a value like 1-2%, the values 1% and 2% are returned as system entities.

Using system entities in conditions

In a node that conditions on the #Customer_Care_Store_Hours intent, you can add conditional responses that use new system entity properties to provide slightly different answers about store hours depending on what the user asks.

 **Tip:** You probably do not want to use all of these conditional responses in a real dialog; they are described here merely to illustrate what's possible.

Conditional response condition	Description	Example response text
syntax		
<code>@sys-date.festival == 'thanksgiving'</code>	Checks whether the customer is asking about hours on Thanksgiving day in particular.	We give out free meals to those in need on Thanksgiving Day.
<code>@sys-date.festival</code>	Checks whether the customer is asking about hours on another specific holiday.	We are closed on Christmas Day, July 4th, and President's Day. On other holidays, we are open from 10AM to 5PM.
<code>@sys-time.part_of_day == 'night'</code>	Checks whether the user includes any terms that mention night as the time of day in the input.	We are not open late; we close at 9PM most days.
<code>@sys-date.datetime_link && input.text.contains('today') && now().sameOrAfter(@sys-time)</code>	Checks whether the input contains a phrase such as today at 8. It also checks whether the @sys-time that is detected is earlier than the current time of day. You can add a context variable to the conditional response that creates a \$new_time variable with the value <? @sys-time.plusHours(12) ?>. You can use this approach to create a date variable that captures the more likely time meant by a user who, at noontime asks, Are you open today at 8. The system assumes the user means 8AM instead of 8PM.	You mean today at \$new_time, correct?
<code>@sys-time.range_link && entities[0].role?.type == 'time_from' && entities[1].role?.type == 'time_to'</code>	Checks whether the input contains a time range. The condition also makes sure that the first @sys-time system entity listed in the entities array is the start time and that the second one is the end time before including them in the response.	You want to know if we're open between <? entities[0] ?> and <? entities[1] ?>, is that correct?
<code>@sys-time && entities.size() < 2 && entities[0].role?.type == 'time_to'</code>	Checks whether the input contains one @sys-time mention that has a time_to role type. If the user input matches this condition, it suggests that the user specified the end time of an open-ended time range. For example, the response would be triggered by the input, Are you open until 9pm?. This input matches because it identifies only one time in a time span, and the mention has a time_to role.	Do you mean from now (<? now().reformatDateTime('h:mm a') ?>) until <? @sys-time.reformatDateTime('h:mm a') ?>?
<code>@sys-date.day_of_week == 'sunday'</code>	Checks whether a specific date that the user is asking about falls on a Sunday.	We are closed on Sundays.

```
@sys-date.specific_day_of_week ==  
'monday' && @sys-date.alternative
```

Checks whether the user mentioned the weekday **Monday** in their query. For example, **Are you open Monday**. The condition also checks whether any alternative dates were stored. Alternative dates are created when your assistant isn't entirely sure which Monday the user means, so it stores the dates of alternative Mondays also. When a user specifies a weekday, your assistant assumes that the user means the future occurrence of the day (the coming Monday). You can add a response that double checks the intended date, by using the detected alternative value. In this case, the alternative date is the previous Monday's date.

Do you mean **@sys-date** or **@sys-date.alternative**?

true

Responds to any other requests for store hour information.

We're open from 9AM to 9PM Monday through Saturday.

System entities in conditional responses



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Time zones supported by system entities

The following list of supported time zones can be used with the time zone functions for the [@sys-date and @sys-time entities](#).

Time zone	Time zone
Africa/Abidjan	Africa/Accra
Africa/Addis_Ababa	Africa/Algiers
Africa/Asmara	Africa/Asmera
Africa/Bamako	Africa/Bangui
Africa/Banjul	Africa/Bissau
Africa/Blantyre	Africa/Brazzaville
Africa/Bujumbura	Africa/Cairo
Africa/Casablanca	Africa/Ceuta
Africa/Conakry	Africa/Dakar
Africa/Dar_es_Salaam	Africa/Djibouti
Africa/Douala	Africa/El_Aaiun
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
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
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
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
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
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
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
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
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
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
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

Supported time zones

 **Important:** This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Festivals

The `@sys-date` system entity recognizes dates that fall on national holidays in different geographic regions.

The following sections list the holidays that `@sys-date` recognizes for different locales.

Holidays (en-us)

Holiday name	String to use to check for a festival match
New Year's Day	<code>newyear</code>
Inauguration Day	<code>inauguration</code>
Martin Luther King, Jr Day	<code>luther</code>
Groundhog Day	<code>groundhog</code>
Washington's Birthday	<code>washington</code>
President's Day	<code>president</code>
Valentine's Day	<code>valentine</code>
International Women's Day	<code>women</code>
St Patrick's Day	<code>patrick</code>
April Fool's Day	<code>fool</code>

Good Friday	goodfriday
Easter Sunday	easter
Easter Monday	eastermonday
Earth Day	earth
Memorial Day	memorial
Mother's Day	mother
Father's Day	father
Independence Day	independence
Labor Day	labor
Columbus Day	columbus
Veteran's Day	veterans
Thanksgiving	thanksgiving
Halloween	halloween
Christmas Eve	christmaseve
Christmas	christmas
Boxing Day	boxing
New Year's Eve	newyearseve
US English holidays	

Holidays (en-ca)

Holiday name	String to use to check for a festival match
New Year's Day	newyear
Epiphany	epiphany
Groundhog Day	groundhog
Valentines Day	valentine
International Women's Day	women
Commonwealth Day	commonwealth
St Patrick's Day	patrick
April Fool's Day	fool
Good Friday	goodfriday

Easter Sunday	easter
Easter Monday	eastermonday
National Tartan Day	tartan
Vimy Ridge Day	vimyridge
Earth Day	earth
Mother's Day	mother
Victoria Day	victoria
Father's Day	father
National Indigenous Day	indigenous
Canada Day	canada
Labor Day	labor
Thanksgiving	thanksgiving
Remembrance Day	remembrance
Halloween	halloween
Anniversary of the Statue of Westminster	westminster
Christmas Eve	christmaseve
Christmas Day	christmas
Boxing Day	boxing
New Year's Eve	newyearseve
Canadian holidays	

Holidays (pt-br)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
Carnival Friday	carnivalday
Carnival Friday	carnivalfriday
Carnival Saturday	carnivalsaturday
Carnival Sunday	carnivalsunday
Carnival Monday	carnivalmonday
Carnival Tuesday	carnivaltuesday

Carnival End (Until 2pm)	carnivalend
Good Friday	goodfriday
Tiradentes Day	tiradentesday
Easter Sunday	eastersunday
Labor Day / May Day	laborday
Mother's Day	mothersday
Brazilian Valentine's Day	valentinesday
Corpus Christi	corpuschristi
Father's Day	fatherday
Independence Day	independenceday
Our Lady of Aparecida / Children's Day	ourladyofaparecida
Teacher's Day	teachersday
Public Service Holiday	publicserviceholiday
All Souls' Day	allsoulsday
Republic Proclamation Day	republicproclamationday
Black Consciousness Day	blackconsciousnessday
Christmas Eve (from 2pm)	christmaseve
Christmas Day	christmasday
New Year's Eve (from 2pm)	newyearseve
Brazilian holidays	

Holidays (en-gb)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
2nd January (Scotland)	2ndjanuary
St. David's Day (Wales)	st.davidsday
Valentines Day	valentine
St Patrick's Day (Northern Ireland)	stpatricksday
St Patrick's Day Off (Northern Ireland)	stpatricksdayoff
Good Friday	goodfriday

Easter Sunday	easter
Easter Monday (England, Wales, Northern Ireland, Scotland)	eastermonday
St. George's Day	st.georgesday
Early May Bank Holiday	earlymaybankholiday
Spring Bank Holiday	springbankholiday
Queen's Birthday	queensbirthday
Battle of the Boyne (Northern Ireland)	battleoftheboyne
Summer Bank Holiday (Scotland)	summerbankholiday
Summer Bank Holiday (England, Wales, Northern Ireland, Scotland)	summerbankholiday
Halloween	halloween
Guy Fawkes Day	guyfawkesday
Remberance Sunday	rememberancesunday
St Andrew's Day (Scotland)	standrewsday
St Andrew's Day Observed (Scotland)	standrewsdayobserved
Christmas Day	christmas
Boxing Day	boxingday
Battle of the Boyne Observed (Northern Ireland)	battleoftheboyneobserved
Bank Holiday (Day 1)	bankholidayfirst
Bank Holiday (Day 2)	bankholidaysecond
2nd January (substitute day) (Scotland)	2ndjanuary
New Year's Eve	newyearseve
'New Year's Day' Observed	newyearsdayobserved
British English holidays	

Holidays (cs-cz)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
Restoration of the Czech Independence Day	restorationofczechindependenceday
St. Valentine's Day	valentinesday
International Women's Day	internationalwomensday

Good Friday	goodfriday
Easter Monday	eastermonday
April Fools Day	aprilfools
Labor Day / May Day	laborday
Victory in Europe Day	victoryineuropeday
Mother's Day	mothersday
Childeren's Day	childrensday
Father's Day	fathersday
Saints Cyril and Methodius	saintcyrilandmethodius
Jan Hus Day	janhusday
St. Wenceslas Day	wenceslasday
Independent Czechoslovak State Day	independenczechoslovakstateday
Struggle for Freedom and Democracy Day	struggleforfreedomanddemocracyday
Christmas Eve	christmaseve
Christmas Day	christmasday
St. Stephen's Day	st.stephensday
New Year's Eve	newyearseve
Czech holidays	

Holidays (nl-nl)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
Valentine's Day	Valentinesday
Good Friday	goodfriday
Easter Sunday	eastersunday
Easter Monday	eastermonday
King's Birthday	kingsbirthday
Liberation Day	liberationday
Ascension Day	ascensionday
Whit Sunday	whitsunday

Whit Monday	<code>whitmonday</code>
St Nicholas' Eve/Sinterklaas	<code>stnicholaseve</code>
Christmas Eve	<code>christmaseve</code>
Christmas Day	<code>christmasday</code>
Second Day of Christmas	<code>seconddayofchristmas</code>
New Year's Eve	<code>newyearseve</code>
	Dutch holidays

Holidays (fr-fr)

Holiday name	String to use to check for a festival match
New Year's Day	<code>newyearsday</code>
Valentine's Day	<code>valentinesday</code>
Easter Sunday	<code>eastersunday</code>
Easter Monday	<code>eastermonday</code>
Labor Day / May Day	<code>laborday</code>
WWII Victory Day	<code>wwiivictoryday</code>
Mother's Day	<code>mothersday</code>
Ascension Day	<code>ascensionday</code>
Whit Sunday	<code>whitsunday</code>
Whit Monday	<code>whitmonday</code>
Father's Day	<code>fathersday</code>
Bastille Day	<code>bastilleday</code>
Assumption of Mary	<code>assumptionofmary</code>
All Saints' Day	<code>allsaintsday</code>
Armistice Day	<code>armisticeday</code>
Christmas Eve	<code>christmaseve</code>
Christmas Day	<code>christmasday</code>
New Year's Eve	<code>newyearseve</code>
	French holidays

Holidays (de-de)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
Epiphany (BW, BY, ST)	epiphany
Valentine's Day	valentinesday
Shrove Monday	shrovetuesday
Carnival / Shrove Tuesday	carnival
Carnival / Ash Wednesday	carnival
International Women's Day (Most regions)	internationalwomensday
April Fool's Day	aprilfools
Palm Sunday	palmsunday
Maundy Thursday	maundythursday
Good Friday	goodfriday
Easter Sunday (Most regions)	eastersunday
Easter Monday	eastermonday
May Day	mayday
Mother's Day	mothersday
Father's Day	fathersday
Ascension Day	ascensionday
Whit Sunday (Most regions)	whitsunday
Whit Sunday (Brandenburg)	whitsunday
Whit Monday	whitmonday
Corpus Christi (Many regions)	corpuschristi
Assumption of Mary (Bavaria, Saarland)	assumptioonmary
Day of German Unity	dayofgermanunity
Reformation Day (Most regions)	reformationday
Halloween	halloween
All Saints' Day (Many regions)	allsaintsday
St Martin's Day	st.martinsday

National Day of Mourning	<code>nationaldayofmourning</code>
Repentance Day (Saxony)	<code>repentanceday</code>
Sunday of the Dead	<code>sundayofthedead</code>
First Advent Sunday	<code>firstadventsunday</code>
Saint Nicholas Day	<code>saintnicholasday</code>
Second Advent Sunday	<code>secondadventsunday</code>
Third Advent Sunday	<code>thirdadventsunday</code>
Fourth Advent Sunday	<code>fourthadventsunday</code>
Christmas Day	<code>christmasday</code>
Boxing Day	<code>boxingday</code>
Whit Sunday (Most regions)	<code>whitsunday</code>
New Year's Eve	<code>newyearseve</code>
German holidays	

Holidays (it-it)

Holiday name	String to use to check for a festival match
New Year's Day	<code>newyearsday</code>
Valentine's Day	<code>valentineday</code>
Epiphany	<code>epiphany</code>
Good Friday	<code>goodfriday</code>
Easter Sunday	<code>eastersunday</code>
Easter Monday	<code>eastermonday</code>
Liberation Day	<code>liberationday</code>
Labor Day / May Day	<code>laborday</code>
Republic Day	<code>republicday</code>
Assumption of Mary / Ferragosto	<code>assumptionofmary</code>
All Saints' Day	<code>allsaintsday</code>
Feast of the Immaculate Conception	<code>feastoftheimmaculateconception</code>
Christmas Day	<code>christmasday</code>
St Stephen's Day	<code>st.stephensday</code>

Labor Day / May Day	<code>laborday</code>
New Year's Eve	<code>newyearseve</code>
Italian holidays	

Holidays (pt-pt)

Holiday name	String to use to check for a festival match
New Year's Day	<code>newyearsday</code>
Valentine's Day	<code>valentinesday</code>
Carnival / Shrove Tuesday	<code>carnivalday</code>
Good Friday	<code>goodfriday</code>
Easter Sunday	<code>eastersunday</code>
Liberty Day	<code>libertyday</code>
Labor Day / May Day	<code>laborday</code>
Mother's Day	<code>mothersday</code>
Father's Day	<code>fathersday</code>
Portugal Day	<code>portugalday</code>
Corpus Christi	<code>corpuschristi</code>
Assumption of Mary	<code>assumptionofmary</code>
Republic Implantation	<code>republicimplantation</code>
All Saints' Day	<code>allsaintsday</code>
Restoration of Independence	<code>restorationofindependence</code>
Feast of the Immaculate Conception	<code>immaculateconception</code>
Christmas Eve	<code>christmaseve</code>
Christmas Day	<code>christmasday</code>
New Year's Eve	<code>newyearseve</code>
Portuguese holidays	

Holidays (sk-sk)

Holiday name	String to use to check for a festival match
Republic Day	<code>republicday</code>
Epiphany	<code>epiphany</code>

Good Friday	goodfriday
Easter Sunday	eastersunday
Easter Monday	eastermonday
Labor Day	laborday
End of World War II	endofworldwarii
St. Cyril & St. Methodius Day	st.cyril&st.methodiusday
National Uprising Day	nationaluprisingday
Constitution Day	constitutionday
Day of Our Lady of Sorrows	dayofourladyofsorrows
All Saints' Day	allsaintsday
Fight for Freedom and Democracy Day	fightforfreedomanddemocracyday
Christmas Eve	christmaseve
Christmas Day	christmasday
St. Stephen's Day	st.stephensday
St. Cyril & St. Methodius Day	cyrilmethodiusday
new year's eve	newyearseve
	Slovak holidays

Holidays (es-es)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
Epiphany	epiphany
Epiphany observed (Many regions)	epiphanyobserved
Valentine's Day	valentinesday
Andalusia Day	andalusiaday
Baleares Day	balearesday
Ash Wednesday	ashwednesday
San Jose (Many regions)	sanjose
Palm Sunday	palmsunday
Maundy Thursday (Many regions)	maundythursday

Good Friday	goodfriday
Easter Sunday	eastersunday
Easter Monday (Many regions)	eastermonday
Aragon Day	aragonday
Saint George Day	stgeorgeday
Labor Day / May Day	laborday
Community Day	communityday
San Isidro	saintisidro
Mothers' Day	mothersday
Whit Sunday/Pentecost	whitsunday
Feast of Saint James the Apostle (GA, PV, S)	feastofsaintjamestheapostle
Assumption of Mary	assumptionofmary
Hispanic Day	hispanicday
All Saints' Day (All)	allsaintsday
Constitution Day	constitutionday
Immaculate Conception	immaculateconception
Immaculate Conception observed (Many regions)	immaculateconceptionobserved
Christmas Eve	christmaseve
Christmas Day	christmasday
Feast of the Holy Family	feastoftheholystfamily
Corpus Christi	corpuschristi
All Saints' Day' observed (Most regions)	allsaintsdayobserved
Assumption observed	assumptionobserved
Cantabria Day	cantabriaday
Christmas Day observed (Balearic Islands)	christmasdayobserved
Labor Day observed (Many regions)	labordayobserved
Christmas Day observed (Many regions)	christmasdayobserved
Immaculate Conception observed	immaculateconceptionobserved

Hispanic Day observed (Many regions)	<code>hispanicdayobserved</code>
Almuneda Day	<code>almunedaday</code>
New Year's Eve	<code>newyearseve</code>
Spanish holidays	

Holidays (ar-ar)

Holiday name	String to use to check for a festival match
new year	<code>newyear</code>
Valentine	<code>valentine</code>
easter sunday	<code>easter</code>
christmas eve	<code>christmaseve</code>
christmas	<code>christmas</code>
Eid al-Fitr	<code>eidalfitr</code>
Eid al-Adha	<code>eidaladha</code>
Ramadan	<code>ramadan</code>
Islamic New Year	<code>islamicnewyear</code>
Ashura	<code>ashura</code>
Mawlid an-Nabi	<code>mawlidannabi</code>
Day of Arafat	<code>dayofarafat</code>
Laylat al-Miraj	<code>laylatalmiraj</code>
Arabic holidays	

Holidays (iw-il)

Holiday name	String to use to check for a festival match
Erev Purim	<code>erevpurim</code>
Yom HaAliyah	<code>yomhaaliyah</code>
Erev Pesach	<code>erevpesach</code>
Pesach I (First day of Passover)	<code>pesachi</code>
Pesach II (Passover)	<code>pesachii</code>
Pesach III (Passover)	<code>pesachiii</code>
Pesach IV (Passover)	<code>pesachiv</code>

Pesach V (Passover)	pesachv
Pesach VI (Passover)	pesachvi
Pesach VII (Last day of Passover)	pesachvii
Yom HaShoah/Holocaust Memorial Day	yomhashoah
Yom HaZikaron (Memorial Day)	yomhazikaron
Yom HaAtzmaut (Independence Day)	yomhaatzmaut
Yom Yerushalayim (Jerusalem Day)	yomyerushalayim
Erev Shavuot	erevshavuot
Shavuot (Pentecost)	shavuot
Erev Tisha B'Av	erevtishabav
Tisha B'Av	tishabav
Erev Rosh Hashana	erevroshhashana
Rosh Hashana (New Year)	roshhashana
Rosh Hashana II (New Year day 2)	roshhashanaii
Erev Yom Kippur	erevyomkippur
Yom Kippur	yomkippur
Erev Sukkot	erevsukkot
Sukkot I	sukkoti
Sukkot II	sukkotii
Sukkot III	sukkotiii
Sukkot IV	sukkotiv
Sukkot V	sukkotv
Sukkot VI	sukkotvi
Sukkot VII/Hoshanah Rabah	sukkotvii
Shmini Atzeret/Simchat Torah	shminiatzeret
Yom HaAliyah School Observance	yomhaaliyahschoolobservance
Hanukkah I (Holiday of lights)	hanukkahI
Hanukkah II	hanukkahII

Hanukkah III	hanukkahiii
Hanukkah IV	hanukkahiv
Hanukkah V	hanukkahv
Hanukkah VI/Rosh Chodesh Tevet	hanukkahvi
Hanukkah VII	hanukkahvii
Hanukkah VIII	hanukkahviii
Hebrew holidays	

Holidays (zh-cn)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
New Year's Weekend	newyearsweekend
Double Seventh Festival	doubleseventhfestival
Double Ninth Festival	doubleninthfestival
Laba Festival	labafestival
Ching Ming Festival	chingmingfestival
Longtaitou Festival	longtaitoufestival
Spring Festival Eve	springfestivaleve
Chinese New Year Eve	chinesenewyeareve
Chinese New Year	chinesenewyear
Lantern Festival	lanternfestival
Zhonghe Festival	zhonghefestival
christmas	christmas
Labor Day	laborday
Valentine	valentine
easter sunday	easter
Dragon Boat Festival	dragonboatfestival
Mid-Autumn Festival	midautumnfestival
National Day	nationalday
Simplified Chinese holidays	

Holidays (zh-tw)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
New Year's weekend	newyearsweekend
Double Seventh Festival	doubleseventhfestival
Double Ninth Festival	doubleninthfestival
Laba Festival	labafestival
Ching Ming Festival	chingmingfestival
Longtaitou Festival	longtaitoufestival
Spring Festival Eve	springfestivaleve
Chinese New Year Eve	chinesenewyeareve
Chinese New Year	chinesenewyear
Lantern Festival	lanternfestival
Zhonghe Festival	zhonghefestival
christmas	christmas
Labor Day	laborday
Valentine	valentine
easter sunday	easter
Dragon Boat Festival	dragonboatfestival
Mid-Autumn Festival	midautumnfestival
National Day	nationalday

Traditional Chinese holidays

Holidays (ja-jp)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
January 2 Bank Holiday	january2bankholiday
January 3 Bank Holiday	january3bankholiday
Coming of Age Day	comingofageday
National Foundation Day	nationalfoundationday

easter sunday	easter
Saint Valentines Day	valentinesday
Dolls' Festival/Girls' Festival	dollsestival
Spring Equinox	springequinox
Shōwa Day	showaday
Coronation Day holiday	coronationdayholiday
Coronation Day	coronationday
Constitution Memorial Day	constitutionmemorialday
Greenery Day	greenerday
Children's Day	childrensday
'Children's Day' observed	childrensdayobserved
Star Festival	starfestival
Sea Day	seaday
Mountain Day	mountainday
'Mountain Day' observed	mountaindayobserved
Respect for the Aged Day	respectfortheagedday
Autumn Equinox	autumnequinox
Health and Sports Day	healthandsportsday
Enthronement Ceremony Day	enthronementceremonyday
Culture Day	cultureday
'Culture Day' Observed	culturedayobserved
7-5-3 Day	753day
Labor Thanksgiving Day	laborthanksgivingday
Christmas	christmas
December 31 Bank Holiday	december31bankholiday
Emperor's Birthday	emperorsbirthday
'Emperor's Birthday' observed	emperorsbirthdayobserved
'Constitution Memorial Day' observed	constitutionmemorialdayobserved

Sports Day	sportsday
'New Year's Day' observed	newyearsdayobserved
'National Foundation Day' observed	nationalfoundationdayobserved
'Autumn Equinox' observed	autumnequinoxobserved
'Greenery Day' observed	greenergydayobserved
'Labor Thanksgiving Day' observed	laborthanksgivingdayobserved
Bridge Public holiday	bridgepublicholiday
'Spring Equinox' observed	springequinoxobserved
December 31 Bank Holiday	december31bankholiday
Japanese holidays	

Holidays (ko-kr)

Holiday name	String to use to check for a festival match
New Year's Day	newyearsday
Independence Movement Day	independencemovementday
Labor Day	laborday
Children's Day	childrensday
Parents' Day	parentsday
Teacher's Day	teachersday
Buddha's Birthday	buddhasbirthday
Memorial Day	memorialday
Constitution Day	constitutionmemorialday
Liberation Day	liberationday
Chuseok	chuseok
Armed Forces Day	armedforcesday
National Foundation Day	nationalfoundationday
Hangeul Proclamation Day	hangeulproclamationday
Christmas Eve	christmaseve
Christmas Day	christmas
New Year's Eve	newyearseve

⚠️ Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Currency support

The `@sys-currency` system entity recognizes different currencies in different geographic regions.

The following sections list the currencies that `@sys-currency` recognizes for different locales.

Currencies (en-us)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
CAD	CA\$
AUS	A\$
GBP	£,gbp,quid
INR	₹,inr,rs,rs.
EUR	€,eur
JPY	¥
MXN	mxn
CHF	chf

English currencies

Currencies (cs-cz)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥,jpy
CZK	kč,kč,czk
MXN	mxn

Czech currencies

Currencies (da-dk)

Currency code	Example of recognized values (case insensitive)
---------------	---

USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn
DKK	dkk
SEK	sek
NOK	nok

Czech currencies

Currencies (nl-nl)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn

Dutch currencies

Currencies (fr-fr)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn
CHF	fr.,chf

French currencies

Currencies (de-de)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn
CHF	chf

German currencies

Currencies (it-it)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd,us\$
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn

Italian currencies

Currencies (pt-pt)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd,us\$
BRL	r\$,brl
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn

Portuguese currencies

Currencies (sk-sk)

Currency code	Example of recognized values (case insensitive)
---------------	---

USD	\$
GBP	£
INR	₹
EUR	€
JPY	¥
CZK	kc,kč

Slovak currencies

Currencies (es-es)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd,us\$
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn

Spanish currencies

Currencies (sv-se)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn
DKK	dkk
SEK	sek
NOK	nok

Swedish currencies

Currencies (tr-TR)

Currency code	Example of recognized values (case insensitive)
---------------	---

TRL	₺,tl,try
USD	\$,usd
GBP	£,gbp,quid
INR	₹,inr,rs
EUR	€,eur
JPY	¥
MXN	mxn

Turkish currencies

Currencies (ar-ar)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥
MXN	mxn
EGP	£,egp
IQD	iqd
SYP	syp

Arabic currencies

Currencies (iw-il)

Currency code	Example of recognized values (case insensitive)
USD	\$
GBP	£
INR	₹
EUR	€
JPY	¥
MXN	mxn
ILS	₪

Hebrew currencies

Currencies (zh-cn)

Currency code	Example of recognized values (case insensitive)
USD	\$
GBP	£
INR	₹
EUR	€
CNY	¥

Simplified Chinese currencies

Currencies (zh-tw)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd,\\\\
GBP	£
INR	₹
EUR	€,\\\\
CNY	¥,\\\\\\
TWD	nt\$,twd,\\\\
HKD	\\\\\\

Traditional Chinese currencies

Currencies (ja-jp)

Currency code	Example of recognized values (case insensitive)
USD	\$,usd,\\\\\\,us\\\\
GBP	£,gbp
INR	₹,inr
EUR	€,eur
JPY	¥,jpy
MXN	mxn

Japanese currencies

Currencies (ko-kr)

Currency code	Example of recognized values (case insensitive)
USD	USD

GBP	£
INR	₹
EUR	€
JPY	¥
KRW	₩

Korean currencies

Currencies (universal model)

Currency code	Example of recognized values (case insensitive)
USD	'\$', 'usd'
CAD	'\$', 'c\$', 'ca\$', 'can\$', 'cad'
AUD	'a\$', 'au\$', 'aud'
CHF	'chf'
GBP	'£', 'gbp', 'quid'
INR	'₹', 'inr', 'rs', 'rs.'
EUR	'€', 'eur'
JPY	'¥', 'jpy'
MXN	'mxn'
RUB	'₽', 'rub'

Universal model currencies

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [Supported languages](#).

Supported languages

Watson Assistant supports individual features to varying degrees per language.

Watson Assistant has classifier models that are designed specifically to support conversational skills in the following languages:

Language	Language code
Arabic	ar
Chinese (Simplified)	zh-cn
Chinese (Traditional)	zh-tw
Czech	cs
Dutch	nl
English	en-us
French	fr
German	de
Italian	it
Japanese	ja
Korean	ko
Portuguese (Brazilian)	pt-br
Spanish	es
Universal*	xx

Table 1. Supported languages

* If you want to support conversations in a language for which Watson Assistant doesn't have a dedicated model, such as Russian, use the *Universal* language model. For more information, see [Adding support for global audiences](#).

Feature support details

The following tables illustrate the level of language support available for product features.

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. Features might 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.

Skill support details

Language	Actions skill	Dialog skill	Search skill
English (en)	GA	GA	GA
Arabic (ar)	GA	GA	GA
Chinese (Simplified) (zh-cn)	GA	GA	GA

Chinese (Traditional) (zh-tw)	GA	GA	GA
Czech (cs)	GA	GA	GA
Dutch (nl)	GA	GA	GA
French (fr)	GA	GA	GA
German (de)	GA	GA	GA
Italian (it)	GA	GA	GA
Japanese (ja)	GA	GA	GA
Korean (ko)	GA	GA	GA
Portuguese (Brazilian) (pt-br)	GA	GA	GA
Spanish (es)	GA	GA	GA
Universal (xx)	GA	GA	GA

Table 2. Skill support details

Intent feature support details

Language	Content Catalog	Algorithm version
English (en)	GA	GA
Arabic (ar)	GA (except Covid-19)	GA
Chinese (Simplified) (zh-cn)	NA	GA
Chinese (Traditional) (zh-tw)	NA	GA
Czech (cs)	NA	GA
Dutch (nl)	NA	GA
French (fr)	GA	GA
German (de)	GA (except Covid-19)	GA
Italian (it)	GA (except Covid-19)	GA
Japanese (ja)	GA (except Covid-19)	GA
Korean (ko)	NA	GA
Portuguese (Brazilian) (pt-br)	GA	GA
Spanish (es)	GA	GA
Universal (xx)	NA	NA

Table 3. Intent feature support details

User input processing support details

Language	Dictionary-based entity support	Fuzzy matching (Misspelling)	Fuzzy matching (Stemming)	Fuzzy matching (Partial match)	Autocorrection
English (en)	GA	GA	GA	GA	GA
Arabic (ar)	GA	GA	NA	NA	NA
Chinese (Simplified) (zh-cn)	GA	NA	NA	NA	NA
Chinese (Traditional) (zh-tw)	GA	NA	NA	NA	NA
Czech (cs)	GA	GA	GA	NA	NA
Dutch (nl)	GA	GA	NA	NA	NA
French (fr)	GA	GA	GA	NA	Beta
German (de)	GA	GA	GA	NA	NA
Italian (it)	GA	GA	NA	NA	NA
Japanese (ja)	GA	GA	NA	NA	NA
Korean (ko)	GA	GA	NA	NA	NA
Portuguese (Brazilian) (pt-br)	GA	GA	NA	NA	NA
Spanish (es)	GA	GA	NA	NA	NA
Universal (xx)	GA	GA	NA	NA	NA

Table 4. User input processing support details

Entity feature support details

Language	Contextual entities
English (en)	GA
Arabic (ar)	NA
Chinese (Simplified) (zh-cn)	NA
Chinese (Traditional) (zh-tw)	NA
Czech (cs)	NA
Dutch (nl)	NA
French (fr)	Beta
German (de)	NA
Italian (it)	NA

Japanese (ja)	NA
Korean (ko)	NA
Portuguese (Brazilian) (pt-br)	NA
Spanish (es)	NA
Universal (xx)	NA

Table 5. Entity feature support details

System entity feature support details

Language	System entities
English (en)	GA
Arabic (ar)	GA
Chinese (Simplified) (zh-cn)	GA
Chinese (Traditional) (zh-tw)	GA
Czech (cs)	GA
Dutch (nl)	GA
French (fr)	GA
German (de)	GA
Italian (it)	GA
Japanese (ja)	GA
Korean (ko)	GA
Portuguese (Brazilian) (pt-br)	GA
Spanish (es)	GA
Universal (xx)	GA

Table 6. System entity feature support details



Note: The Watson Assistant service supports multiple languages as noted, but the tool 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 skill language

Once a skill has been created, its language cannot be modified. If it is necessary to change the supported language of a skill, you can do so by editing the skill's underlying JSON.

To change the skill language, take the following steps:

1. Download the skill that you want to edit.
2. Open the downloaded skill JSON file in a text editor.
3. Search for the property named `language`.

The `language` property is set to the original language of the skill. For example, the language property is `en` for an English skill.

4. Change the value of this property to the language you want to use instead. For example, change it to `fr` for French or `de` for German.
5. Save the changes to the JSON file, and then upload the edited file, overwriting the existing skill.

Configuring bidirectional languages

For bidirectional languages, such as Arabic, you can change your skill preferences.

1. From your skill tile, click the *Actions* drop-down menu, and then select **Language Preferences**.



Note: This option is only available for skills that are configured to use a bidirectional language.

2. Select from the following options for your skill:

- o **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.
- o **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.
- o **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.
- o **Calendar Type:** Specifies how you choose filtering dates in the skill UI. Choose `Islamic-Civil`, `Islamic-Tabular`, `Islamic-Umm al-Qura`, or `Gregorian`.



Note: This setting is not reflected in the "Try it out" panel.

Language Preferences

GUI Direction

LTR

Text Direction

Auto

Numeric Shaping

Arabic Indic

Calendar Type

Islamic-Civil

3. Click the **X** to close the page. Your changes are saved automatically.

Working with accented characters

In a conversational setting, users might or might not use accents while interacting with the Watson Assistant service. As such, both accented and non-accented versions of words might 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 might implicitly have an accent, your assistant 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), your assistant 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.

Documentation for the **classic Watson Assistant** experience has moved. For the most up-to-date version, see [FAQs for IBM Watson® Assistant](#).

FAQ

Find answers to frequently-asked questions and quick fixes for common problems.

What's a...

Term	Definition
Action	An action that you add to an actions skill represents a discrete task or question that your assistant is designed to help customers with. Learn more .
Assistant	Container for your skills. You add skills to an assistant, and then deploy the assistant when you are ready to start helping your customers. Learn more .
Condition	Logic that is defined in the <i>If assistant recognizes</i> section of a dialog node that determines whether the node is processed. The dialog node conditions is equivalent to an If statement in If-Then-Else programming logic.
Content catalog	A set of prebuilt intents that are categorized by subject, such as customer care. You can add these intents to your skill and start using them immediately. Or you can edit them to complement other intents that you create. Learn more .
Context variable	A variable that you can use to collect information during a conversation, and reference it later in the same conversation. For example, you might want to ask for the customer's name and then address the person by name later on. A context variable is used by the dialog skill. Learn more .
Dialog	The component where you build the conversation that your assistant has with your customers. For each defined intent, you can author the response your assistant should return. Learn more .
Digression	A feature that gives the user the power to direct the conversation. It prevents customers from getting stuck in a dialog thread; they can switch topics whenever they choose. Learn more .
Disambiguation	A feature that enables the assistant to ask customers to clarify their meaning when the assistant isn't sure what a user wants to do next. Learn more .
Entity	Information in the user input that is related to the user's purpose. An intent represents the action a user wants to do. An entity represents the object of that action. Learn more .
Integrations	Ways you can deploy your assistant to existing platforms or social media channels. Learn more .
Intent	The goal that is expressed in the user input, such as answering a question or processing a bill payment. Learn more .
Message	A single turn within a conversation that includes a single call to the <code>/message</code> API endpoint and its corresponding response.
Monthly active user (MAU)	A single unique user who interacts with an assistant one or many times in a given month.
Preview	Embeds your assistant in a chat window that is displayed on an IBM-branded web page. From the preview, you can test how a conversation flows through any and all skills that are attached to your assistant, from end to end. Learn more .
Response	Logic that is defined in the <i>Assistant responds</i> section of a dialog node that determines how the assistant responds to the user. When the node's condition evaluates to true, the response is processed. The response can consist of an answer, a follow-up question, a webhook that sends a programmatic request to an external service, or slots which represent pieces of information that you need the user to provide before the assistant can help. The dialog node response is equivalent to a Then statement in If-Then-Else programming logic.
Skill	Does the work of the assistant. A dialog skill has the training data and dialog that your assistant uses to chat with customers. An actions skill is a new way to build a conversation. Actions offer step-by-step flows for a conversations and are made so that anybody can build them. A search skill is configured to search the appropriate external data sources for answers to customer questions. Learn more .
Skill version	Versions are snapshots of a skill that you can create at key points during the development lifecycle. You can deploy one version to production, while you continue to make and test improvements that you make to another version of the skill. Learn more .

Slots	A special set of fields that you can add to a dialog node that enable the assistant to collect necessary pieces of information from the customer. For example, the assistant can require a customer to provide valid date and location details before it gets weather forecast information on the customer's behalf. Learn more .
Step	A step that you add to an action represents a single interaction or exchange of information with a customer, a turn in the conversation. Learn more .
System entity	Prebuilt entities that recognize references to common things like dates and numbers. You can add these to your skill and start using them immediately. Learn more .
Try it out	A chat window that you use to test as you build. For example, from the dialog skill's "Try it out" pane, you can mimic the behavior of a customer and enter a query to see how the assistant responds. You can test only the current skill; you cannot test your assistant and all attached skills from end to end. Learn more .
Variable	A variable is data that a customer shares with the assistant, which is collected and saved so it can be referenced later. In an actions skill, you can collect <i>action</i> and <i>session</i> variables. Learn more .
Web chat	An integration that you can use to embed your assistant in your company website. Learn more .
Webhook	A mechanism for calling out to an external program during a conversation. For example, your assistant can call an external service to translate a string from English to French and back again in the course of the conversation. Learn more .

I can't log in

If you are having trouble logging in to a service instance or see messages about tokens, such as `unable to fetch access token` or `400 bad request - header or cookie too large`, it might mean that you need to clear your browser cache. Open a private browser window, and then try again.

- If accessing the page by using a private browsing window fixes the issue, then consider always using a private window or clear the cache of your browser. You can typically find an option for clearing the cache or deleting cookies in the browser's privacy and security settings.
- If accessing the page by using a private browsing window doesn't fix the issue, then try deleting the API key for the instance and creating a new one.

I'm being asked to log in repeatedly

If you keep getting messages, such as `you are getting redirected to login`, it might be due to one of the following things:

- The Lite plan you were using has expired. Lite plans expire if they are not used within a 30-day span. To begin again, log in to IBM Cloud and create a new service instance of Watson Assistant.
- An instance is locked when you exceed the plan limits for the month. To log in successfully, wait until the start of the next month when the plan limit totals are reset.

I'm getting a 401 response

The 401 response code is returned for many reasons, including:

- You exceeded the API call limit for your plan for this month. For example, for Lite plans, the monthly limit for API calls 10,000 messages per month. If you reach your limit for the month, but the logs show that you have made fewer calls than the limit, remember that the Lite plan stores log information for 7 days only. The 401 response will go away as soon as the next billing cycle begins, which is the first day of the calendar month.
- You are trying to use an instance-level API key that you created in one data center location to connect to a service instance that is hosted from another location. You must create an API key in the same data center location where your service instance is hosted.

Getting `Unable to fetch access token for account` message

The full message is, `Assistants could not be loaded at this time. Unable to fetch access token for account`.

This message is displayed for a few reasons:

- Your IBM Cloud® refresh tokens might have expired. Log out and then log back in to IBM Cloud to generate fresh tokens.
- Make sure that the IBM ID that you logged in with has access to this service instance. To confirm, you can go to the [IBM Cloud resources list](#), and then log in with the same IBM ID. You should see this service instance listed as one of your available services.

Getting `Authentication Required` or `Sign in` message

You are being asked for credentials to access a Watson Assistant service instance that you have been able to access without trouble in the past. You might see, `Authentication Required: {service-url} is requesting your username and password.` or just a `Sign in` dialog box with

fields for a username and password.

This message can be displayed for service instances that were migrated from Cloud Foundry, but for which access roles were not subsequently updated. After the migration, the service instance owner must update the user permissions to ensure that anyone who needs access to the instance is assigned to the appropriate Platform and Service access roles.

To regain access to the service instance, ask the service instance owner to review your access permissions. Ask to be given at least a service access role of Writer.

After your access roles are fixed, be sure to use the correct web address, the URL of the migrated service instance, to open it.

I don't see the Analytics page

To view the Analytics page, you must have a service role of Manager and a platform role of at least Viewer. For more information about access roles and how to request an access role change, see [Managing access to resources](#).

I am unable to view the API details, API key, or service credentials

If you cannot view the API details or service credentials, it is likely that you do not have Manager access to the service instance in which the resource was created. Only people with Manager service access to the instance can use the service credentials. For more information, see [Getting API information](#).

I can't edit intents, entities, or dialog nodes

To edit skills, you must have Writer service access to the service instance and a platform role of at least Viewer. For more information about access roles and how to request an access role change, see [Managing access to resources](#).

Where can I find an example for creating my first assistant?

Follow the steps in the [Getting started with Watson Assistant](#) tutorial for a product introduction and to get help creating your first assistant.

Can I export the user conversations from the Analytics page?

You cannot directly export conversations from the User conversation page. You can, however, use the `/logs` API to list events from the transcripts of conversations that occurred between your users and your assistant. For more information, see the [API reference](#) and the [Filter query reference](#). Or, you can use a Python script to export logs. For more information, see [export_logs.py](#).

Can I export and import dialog nodes?

No, you cannot export and import dialog nodes from the product user interface.

If you want to copy dialog nodes from one skill into another skill, follow these steps:

1. Download as JSON files both the dialog skill that you want to copy the dialog nodes from and the dialog skill that you want to copy the nodes to.
2. In a text editor, open the JSON file for the dialog skill that you want to copy the dialog nodes from.
3. Find the `dialog_nodes` array, and copy it.
4. In a text editor, open the JSON file for the dialog skill that you want to copy the dialog nodes to, and then paste the `dialog_nodes` array into it.
5. Import the JSON file that you edited in the previous step to create a new dialog skill with the dialog nodes you wanted.

Is it possible to recover a deleted skill?

Regularly [back up data](#) to prevent problems that might arise from inadvertent deletions. If you do not have a backup, there is a short window of time during which a deleted skill might be recoverable. Immediately following the deletion, [open a case](#) with Support to determine if the data can be recovered. Include the following information in your case:

- skill ID
- instance ID or name
- region where the service instance is hosted from which the skill was deleted

Can I change my plan to a Lite plan?

No, you cannot change from a Trial, Plus, or Standard plan to a Lite plan. And you cannot upgrade from a Trial to a Standard plan. For more information, see [Upgrading](#).

How many Lite plan instances of Watson Assistant can I create?

You can have only one Lite plan instance of Watson Assistant per resource group.

How long are log files kept for a workspace?

The length of time for which messages are retained depends on your service plan. For more information, see [Log limits](#).

How do I create a webhook?

To define a webhook and add its details, open the skill where you want to add the webhook. Open the **Options** page, and then click **Webhooks** to add details about your webhook. To invoke the webhook, call it from one or more of your dialog nodes. For more information, see [Making a programmatic call from dialog](#).

Can I have more than one entry in the URL field for a webhook?

No, you can define only one webhook URL for a dialog skill. For more information, see [Defining the webhook](#).

Can I extend the webhook time limit?

No. The service that you call from the webhook must return a response in 8 seconds or less, or the call is canceled. You cannot increase this time limit.

I received the message “Query cancelled” when importing a skill

This message is displayed when the skill import stops because artifacts in the skill, such as dialog nodes or synonyms, exceed the plan limits. For information about how to address this problem, see [Troubleshooting skill import issues](#).

If a timeout occurs due to the size of the skill but no plan limits are exceeded, you can reduce the number of elements that are imported at a time by completing the following steps:

1. Make a copy of the JSON file that you are trying to import.
2. Open the copy of the JSON file in an editor, and delete the **entities** array.
3. Import the edited JSON file as a new skill.
4. If this step is successful, edit the original copy of the JSON file.
5. Remove the **dialog_nodes**, **intents**, and **counterexamples** arrays.
6. Update the skill by using the API. Be sure to include the workspace ID and the **append=true** flag, as in this example:

```
curl -X POST -H "content-type: application/json" -H "accept: application/json" -u "apikey:{apikey}" -d@./skill.json "url/api/v1/workspaces/{workspace_id}?version=2019-02-28&append=true"
```

The training process takes a long time and appears stuck

If the training process gets stuck, first check whether there is an outage for the service by going to the [Cloud status page](#). You can start a new training process to stop the current process and start over. To do so, add a new intent or entity, and then delete it. This action starts a new training process.

Is there a range of IP addresses that are being used by a webhook?

Unfortunately, the IP address ranges from which Watson Assistant may call a webhook URL are subject to change, which in turn prevent using them in any static firewall configuration. Please use the https transport and specify an authorization header to control access to the webhook.

How do I see my monthly active users in Watson Assistant?

To see your monthly active users (MAU) do the following:

1. Sign in to <https://cloud.ibm.com>
2. Click on the **Manage** menu, then choose **Billing and usage**.
3. Click on **Usage**.
4. For Watson Assistant, select **View Plans**.
5. Under Time Frame, select the month you need.
6. Select your Plus plans or Plus Trial plans to see monthly active users and the API calls.

Error: New Off Topic not supported

You see the error **New Off Topic not supported** after editing the JSON file for a dialog skill and changing the skill language from English to another language.

To resolve this issue, modify the JSON file by setting `off_topic` to `false`. For more information about this feature, see [Defining what's irrelevant](#).

Is it possible to increase the number of intents per skill

No, it is not possible to increase the number of intents per skill.



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Getting help

Get help with solving issues that you encounter while using the product.

Use these resources to get answers to your questions:

- For answers to frequently asked questions, see the [FAQ](#).
- Click the **Learning center** link that is displayed in the header of the skill pages to find helpful product tours. The tours guide you through the steps to follow to complete a range of tasks, from adding your first intent to a dialog skill to enhancing the conversation in an actions skill. The *Additional resources* page has links to relevant documentation topics and how-to videos. You can search the resource link titles to find what you're looking for quickly.
- Find answers to common questions or ask questions where experts and other community members can answer by visiting the [Watson Assistant Community forum](#).

If your service plan covers it, you can get help by creating a case from [IBM Cloud Support](#).



Important: This documentation is for the **classic Watson Assistant** experience. To see the documentation for the new Watson Assistant, please go [here](#).

Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

IBM Watson® Assistant uses standard Windows navigation keys.

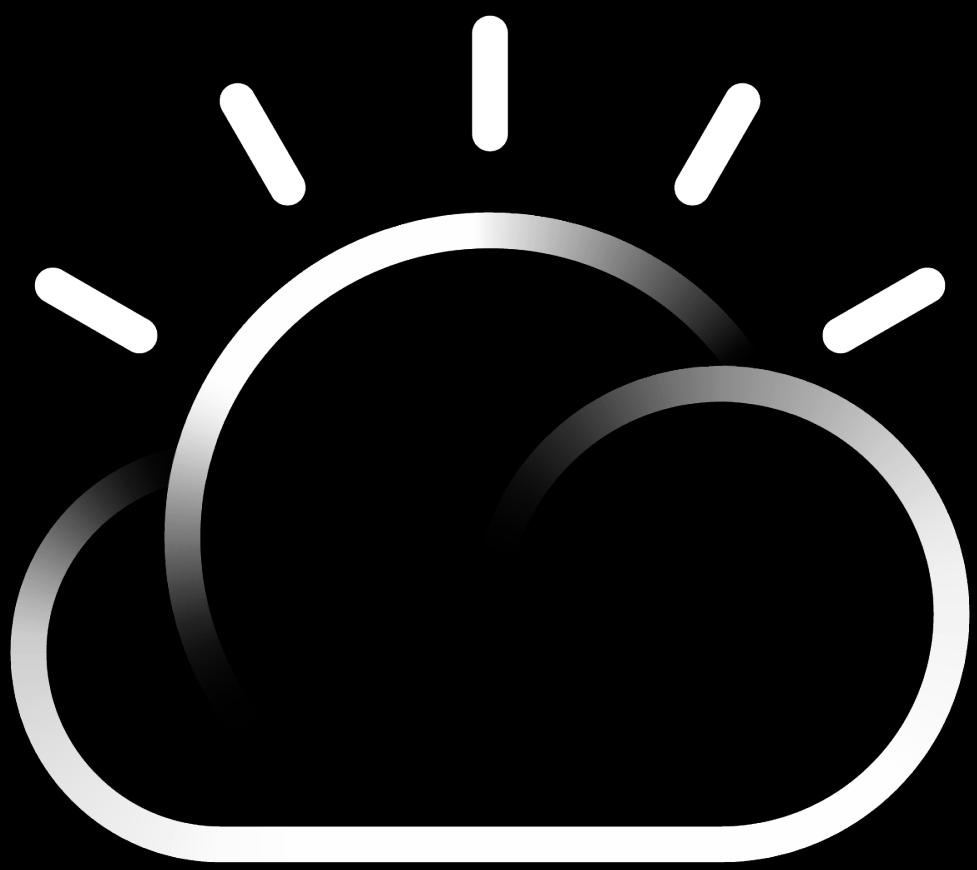
Accessibility features in the product documentation

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