

---

# Amazon Lex

## V2 Developer Guide



## Amazon Lex: V2 Developer Guide

Copyright © 2022 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

## Table of Contents

What is Amazon Lex V2? .....	1
Are You a First-time User of Amazon Lex V2? .....	2
How it works .....	3
Supported languages .....	4
Supported languages and locales .....	4
Languages and locales supported by Amazon Lex V2 features .....	5
Language guidance for Amazon Lex V2 .....	6
Getting started .....	7
Step 1: Set Up an Account .....	7
Sign Up for AWS .....	7
Create an IAM User .....	8
Next step .....	8
Step 2: Getting started (Console) .....	8
Exercise 1: Create a bot from an example .....	9
Exercise 2: Review the conversation flow .....	10
Building bots .....	17
Understanding conversation flow management .....	18
Creating a bot .....	19
Adding a language .....	19
Visual conversation builder .....	19
Adding intents .....	32
Initial response .....	33
Slots .....	35
Confirmation .....	39
Fulfillment .....	42
Closing response .....	44
Configuring prompts .....	44
Adding slot types .....	45
Testing a bot .....	45
Creating versions .....	48
The Draft version .....	48
Creating a version .....	49
Updating an Amazon Lex V2 bot .....	49
Deleting an Amazon Lex V2 bot or version .....	49
Creating conversation paths .....	49
Configure next steps in the conversation .....	53
Set values during the conversation .....	54
Add conditions to branch conversations .....	54
Invoke dialog code hook .....	58
Built-in intents and slot types .....	61
Built-in intents .....	62
Built-in slot types .....	75
Using a custom grammar slot type .....	83
Adding a grammar slot type .....	84
Grammar definition .....	84
Script format .....	93
Industry grammars .....	100
Creating custom slot types .....	181
Using composite slots .....	182
Creating a custom vocabulary to improve speech recognition .....	187
Custom vocabulary basics .....	187
Best practices for creating a custom vocabulary .....	188
Creating a custom vocabulary for eliciting intents and slots .....	188
Creating a custom vocabulary file .....	192

Using multiple values in a slot .....	193
Using a Lambda function .....	194
Attaching a Lambda function to a bot alias .....	195
Input event format .....	196
Response format .....	201
Lambda router function .....	203
Using the Automated Chatbot Designer .....	205
Importing conversation transcripts .....	205
Importing transcripts from Contact Lens for Amazon Connect .....	206
Prepare transcripts .....	206
Upload your transcripts to an S3 bucket .....	207
Analyze your transcripts using Amazon Lex V2 console .....	207
Creating intents and slot types .....	208
Input transcript format .....	209
Output transcript format .....	209
Deploying bots .....	211
Aliases .....	211
Integrating with a Java application .....	212
Integrating your bots .....	216
Messaging platforms .....	216
Integrating with Facebook .....	216
Integrating with Slack .....	219
Integrating with Twilio SMS .....	222
Contact centers .....	223
Amazon Chime SDK .....	224
Amazon Connect .....	225
Genesys Cloud .....	225
Using bots .....	227
Managing conversations .....	227
Managing conversation context .....	228
Setting intent context .....	228
Using default slot values .....	230
Setting session attributes .....	230
Setting request attributes .....	231
Setting the session timeout .....	232
Sharing information between intents .....	232
Setting complex attributes .....	233
Managing sessions .....	234
Starting a new session .....	235
Switching intents .....	235
Resuming a prior intent .....	236
Validating slot values .....	236
Analyzing sentiment .....	236
Using confidence scores .....	237
Using intent confidence scores .....	238
Using voice transcription confidence scores .....	240
Using runtime hints to improve recognition of slot values .....	246
Adding slot values in context .....	247
Adding hints to a slot .....	247
Using spelling styles to capture slot values .....	248
Enabling spelling .....	249
Example code .....	249
Streaming conversations .....	253
Starting a stream to a bot .....	254
Time sequence of events for an audio conversation .....	256
Starting a streaming conversation .....	258
Event stream encoding .....	269

Enabling your bot to be interrupted .....	270
Waiting for the user to provide additional information .....	271
Configuring fulfillment progress updates .....	272
Fulfillment updates .....	273
Post-fulfillment response .....	274
Timeouts for user input .....	274
Interrupt behavior .....	275
Timeouts for voice input .....	275
Timeouts for text input .....	276
Configuration for DTMF input .....	276
Monitoring .....	278
Monitoring with conversation logs .....	278
Configuring conversation logs .....	278
Viewing text logs in Amazon CloudWatch Logs .....	280
Accessing audio logs in Amazon S3 .....	286
Monitoring conversation log status with CloudWatch metrics .....	286
Obscuring slot values .....	286
Viewing utterance statistics .....	287
Logging with CloudTrail .....	288
Amazon Lex V2 information in CloudTrail .....	289
Understanding Amazon Lex V2 log file entries .....	289
Monitoring with CloudWatch .....	290
Importing and exporting .....	296
Exporting .....	296
IAM permissions required to export .....	297
Exporting a bot (console) .....	297
Importing .....	298
IAM permissions required to import .....	299
Importing a bot (console) .....	300
Using a password when importing or exporting .....	301
JSON format for importing and exporting .....	301
Manifest file structure .....	302
Bot file structure .....	302
Bot locale file structure .....	302
Intent file structure .....	303
Slot file structure .....	304
Slot type file structure .....	306
Custom vocabulary file structure .....	308
Tagging resources .....	309
Tagging your resources .....	309
Tag restrictions .....	309
Tagging resources (console) .....	310
Security .....	311
Data protection .....	311
Encryption at rest .....	312
Encryption in transit .....	312
Identity and access management .....	313
Audience .....	313
Authenticating with identities .....	313
Managing access using policies .....	316
How Amazon Lex V2 works with IAM .....	317
Identity-based policy examples .....	324
Resource-based policy examples .....	333
AWS managed policies .....	340
Troubleshooting .....	347
Using service-linked roles .....	348
Service-linked role permissions for Amazon Lex V2 .....	349

Creating a service-linked role for Amazon Lex V2 .....	350
Editing a service-linked role for Amazon Lex V2 .....	350
Deleting a service-linked role for Amazon Lex V2 .....	350
Supported regions for Amazon Lex V2 service-linked roles .....	351
Logging and monitoring .....	351
Compliance validation .....	351
Resilience .....	352
Infrastructure security .....	352
VPC endpoints (AWS PrivateLink) .....	352
Considerations for Amazon Lex V2 VPC endpoints .....	353
Creating an interface VPC endpoint for Amazon Lex V2 .....	353
Creating a VPC endpoint policy for Amazon Lex V2 .....	353
Migration guide .....	355
Amazon Lex V2 overview .....	355
Multiple languages in a bot .....	355
Simplified information architecture .....	355
Improved builder productivity .....	355
AWS CloudFormation resources .....	357
Amazon Lex V2 and AWS CloudFormation templates .....	357
Learn more about AWS CloudFormation .....	357
Guidelines and quotas .....	358
Regions .....	358
General guidelines .....	358
Quotas .....	359
Build-time quotas .....	359
Runtime quotas .....	360
Document history .....	362
API reference .....	368
AWS glossary .....	369

# What is Amazon Lex V2?

Amazon Lex V2 is an AWS service for building conversational interfaces for applications using voice and text. Amazon Lex V2 provides the deep functionality and flexibility of natural language understanding (NLU) and automatic speech recognition (ASR) so you can build highly engaging user experiences with lifelike, conversational interactions, and create new categories of products.

Amazon Lex V2 enables any developer to build conversational bots quickly. With Amazon Lex V2, no deep learning expertise is necessary—to create a bot, you specify the basic conversation flow in the Amazon Lex V2 console. Amazon Lex V2 manages the dialog and dynamically adjusts the responses in the conversation. Using the console, you can build, test, and publish your text or voice chatbot. You can then add the conversational interfaces to bots on mobile devices, web applications, and chat platforms (for example, Facebook Messenger).

Amazon Lex V2 provides integration with AWS Lambda, and you can integrate with many other services on the AWS platform, including Amazon Connect, Amazon Comprehend, and Amazon Kendra. Integration with Lambda provides bots access to pre-built serverless enterprise connectors to link to data in SaaS applications such as Salesforce.

For bots created after August 17, 2022, you can use conditional branching to control the conversation flow with your bot. With conditional branching you can create complex conversations without needing to write Lambda code.

Amazon Lex V2 provides the following benefits:

- **Simplicity** – Amazon Lex V2 guides you through using the console to create your own bot in minutes. You supply a few example phrases, and Amazon Lex V2 builds a complete natural language model through which the bot can interact using voice and text to ask questions, get answers, and complete sophisticated tasks.
- **Democratized deep learning technologies** – Amazon Lex V2 provides ASR and NLU technologies to create a Speech Language Understanding (SLU) system. Through SLU, Amazon Lex V2 takes natural language speech and text input, understands the intent behind the input, and fulfills the user intent by invoking the appropriate business function.

Speech recognition and natural language understanding are some of the most challenging problems to solve in computer science, requiring sophisticated deep learning algorithms to be trained on massive amounts of data and infrastructure. Amazon Lex V2 puts deep learning technologies within reach of all developers. Amazon Lex V2 bots convert incoming speech to text and understand the user intent to generate an intelligent response so you can focus on building your bots with added value for your customers and define entirely new categories of products made possible through conversational interfaces.

- **Seamless deployment and scaling** – With Amazon Lex V2, you can build, test, and deploy your bots directly from the Amazon Lex V2 console. Amazon Lex V2 enables you to publish your voice or text bots for use on mobile devices, web apps, and chat services (for example, Facebook Messenger). Amazon Lex V2 scales automatically. You don't need to worry about provisioning hardware and managing infrastructure to power your bot experience.

- **Built-in integration with the AWS platform** – Amazon Lex V2 operates natively with other AWS services, such as AWS Lambda and Amazon CloudWatch. You can take advantage of the power of the AWS platform for security, monitoring, user authentication, business logic, storage, and mobile app development.
- **Cost-effectiveness** – With Amazon Lex V2, there are no upfront costs or minimum fees. You are charged only for the text or speech requests that are made. The pay-as-you-go pricing and the low cost per request make the service a cost-effective way to build conversational interfaces. With the Amazon Lex V2 free tier, you can easily try Amazon Lex V2 without any initial investment.

## Are You a First-time User of Amazon Lex V2?

If you are a first-time user of Amazon Lex V2, we recommend that you read the following sections in order:

1. [How it works \(p. 3\)](#) – This section introduces Amazon Lex V2 and the features that you use to create a chatbot.
2. [Getting started with Amazon Lex V2 \(p. 7\)](#) – In this section, you set up your account and test Amazon Lex V2.
3. [API Reference](#) – This section contains details about API operations.

# How it works

Amazon Lex V2 enables you to build applications using a text or speech interface for a conversation with a user. Following are the typical steps for working with Amazon Lex V2:

1. Create a bot and add one or more languages. Configure the bot so that it understands the user's goal, engages in conversation with the user to elicit information, and fulfills the user's intent.
2. Test the bot. You can use the test window client provided by the Amazon Lex V2 console.
3. Publish a version and create an alias.
4. Deploy the bot. You can deploy the bot on your own applications or messaging platforms such as Facebook Messenger or Slack

Before you get started, familiarize yourself with the following Amazon Lex V2 core concepts and terminology:

- **Bot** – A bot performs automated tasks such as ordering a pizza, booking a hotel, ordering flowers, and so on. An Amazon Lex V2 bot is powered by automatic speech recognition (ASR) and natural language understanding (NLU) capabilities.

Amazon Lex V2 bots can understand user input provided with text or speech and converse natural language.

- **Language** – An Amazon Lex V2 bot can converse in one or more languages. Each language is independent of the others, you can configure Amazon Lex V2 to converse with a user using native words and phrases. For more information, see [Languages and locales supported by Amazon Lex V2 \(p. 4\)](#).
- **Intent** – An intent represents an action that the user wants to perform. You create a bot to support one or more related intents. For example, you might create an intent that orders pizzas and drinks. For each intent, you provide the following required information:
  - **Intent name** – A descriptive name for the intent. For example, **OrderPizza**.
  - **Sample utterances** – How a user might convey the intent. For example, a user might say "Can I order a pizza" or "I want to order a pizza."
  - **How to fulfill the intent** – How you want to fulfill the intent after the user provides the necessary information. We recommend that you create a Lambda function to fulfill the intent.

You can optionally configure the intent so Amazon Lex V2 returns the information back to the client application for the necessary fulfillment.

In addition to custom intents, Amazon Lex V2 provides built-in intents to quickly set up your bot. For more information, see [Built-in intents and slot types \(p. 61\)](#).

Amazon Lex always includes a fallback intent for each bot. The fallback intent is used whenever Amazon Lex can't deduce the user's intent. For more information, see [AMAZON.FallbackIntent \(p. 62\)](#).

- **Slot** – An intent can require zero or more slots, or parameters. You add slots as part of the intent configuration. At runtime, Amazon Lex V2 prompts the user for specific slot values. The user must provide values for all required slots before Amazon Lex V2 can fulfill the intent.

For example the OrderPizza intent requires slots such as size, crust type, and number of pizzas. For each slot, you provide the slot type and one or more prompts that Amazon Lex V2 sends to the client to elicit values from the user. A user can reply with a slot value that contains additional words, such as "large pizza please" or "let's stick with small." Amazon Lex V2 still understands the slot value.

- **Slot type** – Each slot has a type. You can create your own slot type, or you can use built-in slot types. For example, you might create and use the following slot types for the OrderPizza intent:

- **Size** – With enumeration values Small, Medium, and Large.
- **Crust** – With enumeration values Thick and Thin.

Amazon Lex V2 also provides built-in slot types. For example, AMAZON.Number is a built-in slot type that you can use for the number of pizzas ordered. For more information, see [Built-in intents and slot types \(p. 61\)](#).

- **Version** – A version is a numbered snapshot of your work that you can publish for use in different parts of your workflow, such as development, beta deployment, and production. Once you create a version, you can use a bot as it existed when the version was made. After you create a version, it stays the same while you continue to work on your application.
- **Alias** – An alias is a pointer to a specific version of a bot. With an alias, you can update the version the your client applications are using. For example, you can point an alias to version 1 of your bot. When you are ready to update the bot, you publish version 2 and change the alias to point to the new version. Because your applications use the alias instead of a specific version, all of your clients get the new functionality without needing to be updated.

For a list of the AWS Regions where Amazon Lex V2 is available, see [Amazon Lex V2 endpoints and quotas](#) in the *Amazon Web Services General Reference*.

## Languages and locales supported by Amazon Lex V2

Amazon Lex V2 supports a variety of languages and locales. The languages that are supported, the features that support these languages, and language-specific guidance to improve your bot's performance are provided in this topic.

### Supported languages and locales

Amazon Lex V2 supports the following languages and locales.

Code	Language and locale
ca_ES	Catalan (Spain)
de_AT	German (Austria)
de_DE	German (Germany)
en_AU	English (Australia)
en_GB	English (UK)
en_IN	English (India)
en_US	English (US)
en_ZA	English (South Africa)
es_419	Spanish (Latin America)
es_ES	Spanish (Spain)
es_US	Spanish (US)

<b>Code</b>	<b>Language and locale</b>
fr_CA	French (Canada)
fr_FR	French (France)
hi_IN	Hindi (India)
it_IT	Italian (Italy)
ja_JP	Japanese (Japan)
ko_KR	Korean (Korea)
nl_NL	Dutch (The Netherlands)
pt_BR	Portuguese (Brazil)
pt_PT	Portuguese (Portugal)
zh_CN	Mandarin (PRC)

## Languages and locales supported by Amazon Lex V2 features

The following table lists Amazon Lex V2 features that are limited to certain languages and locales. All other Amazon Lex V2 features are supported in all languages and locales.

<b>Feature</b>	<b>Supported languages and locales</b>
<a href="#">AMAZON.AlphaNumeric (p. 76)</a>	All languages and locales except Korean (ko_KR)
<a href="#">AMAZON.KendraSearchIntent (p. 64)</a>	English (US) (en_US)
<a href="#">Creating a custom vocabulary to improve speech recognition (p. 187)</a>	English (UK) (en_GB) English (US) (en_US)
<a href="#">Automated Chatbot Designer</a>	English (US) (en_US)
Region availability	All languages and locales are available in all Regions except for Catalan (Spain) (ca_ES), Hindi (India) (hi_IN), Dutch (The Netherlands) (nl_NL), Portuguese (Brazil) (pt_BR), Portuguese (Portugal) (pt_PT), and Mandarin (PRC) (zh_CN), which are not available in the following Regions: <ul style="list-style-type: none"> <li>• Asia Pacific (Singapore) (ap-southeast-1)</li> <li>• Africa (Cape Town) (ap-south-1)</li> </ul>
<a href="#">Setting intent context (p. 228)</a>	English (US) (en_US)
<a href="#">Using a custom grammar slot type (p. 83)</a>	English (Australia) (en_AU) English (UK) (en_GB) English (US) (en_US)

Feature	Supported languages and locales
<a href="#">Using multiple values in a slot (p. 193)</a>	English (US) (en_US)
<a href="#">Using runtime hints to improve recognition of slot values (p. 246)</a>	English (UK) (en_GB) English (US) (en_US)
<a href="#">Using spelling styles to capture slot values (p. 248)</a>	English (Australia) (en_AU) English (UK) (en_GB) English (US) (en_US)
<a href="#">Using voice transcription confidence scores (p. 240)</a>	English (UK) (en_GB) English (US) (en_US)

## Language guidance for Amazon Lex V2

To improve your bot's performance, you should adhere to these guidelines for the following languages.

### Hindi

Amazon Lex V2 is able to serve Hindi end-users who switch freely between Hindi and English. If you plan to build a bot that supports this language switching, we recommend the following best practices:

- In the bot definition, write English words in Latin script.
- At least 50% of your sample utterances should represent language switching within the same sentence. In these utterances, use Devanagari script for Hindi words and Latin script for English words (for example, "म ticket book करना चाहता हूँ").
- If you expect users to communicate with the bot using Hindi words in Latin script or English words in Devanagari script, then you should include examples of Hindi words in Latin script (for example, "mujhe ek ticket book karni hai") and English words in Devanagari script (for example, "मेज्ज टकिट की बुकिंग में मदद चाहौं") in your sample utterances.
- If you expect users to communicate with the bot using sentences that are completely in Hindi or completely in English, then you should include sample utterances that are fully in one language (for example, "I want to book a ticket").

# Getting started with Amazon Lex V2

Amazon Lex V2 provides API operations that you can integrate with your existing applications. For a list of supported operations, see the [API Reference](#). You can use any of the following options:

- AWS SDK — When using the SDKs your requests to Amazon Lex V2 are automatically signed and authenticated using the credentials that you provide. We recommend that you use an SDK to build your application.
- AWS CLI — You can use the AWS CLI to access any Amazon Lex V2 feature without having to write any code.
- AWS Console — The console is the easiest way to get started testing and using Amazon Lex V2

If you are new to Amazon Lex V2, we recommend that you read [How it works \(p. 3\)](#) first.

## Topics

- [Step 1: Set Up an AWS Account and Create an Administrator User \(p. 7\)](#)
- [Step 2: Getting started \(Console\) \(p. 8\)](#)

## Step 1: Set Up an AWS Account and Create an Administrator User

Before you use Amazon Lex V2 for the first time, complete the following tasks:

1. [Sign Up for AWS \(p. 7\)](#)
2. [Create an IAM User \(p. 8\)](#)

## Sign Up for AWS

If you already have an AWS account, skip this task.

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all services in AWS, including Amazon Lex V2. You are charged only for the services that you use.

With Amazon Lex V2, you pay only for the resources that you use. If you are a new AWS customer, you can get started with Amazon Lex V2 for free. For more information, see [AWS Free Usage Tier](#).

If you already have an AWS account, skip to the next task. If you don't have an AWS account, use the following procedure to create one.

### To create an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

Write down your AWS account ID because you'll need it for the next task.

## Create an IAM User

Services in AWS, such as Amazon Lex V2, require that you provide credentials when you access them so that the service can determine whether you have permissions to access the resources owned by that service. The console requires your password. You can create access keys for your AWS account to access the AWS CLI or API.

However, we don't recommend that you access AWS using the credentials for your AWS account. Instead, we recommend that you:

- Use AWS Identity and Access Management (IAM) to create an IAM user
- Add the user to an IAM group with administrative permissions
- Grant administrative permissions to the IAM user that you created.

You can then access AWS using a special URL and the IAM user's credentials.

The Getting Started exercises in this guide assume that you have a user (`adminuser`) with administrator privileges. Follow the procedure to create `adminuser` in your account.

### To create an administrator user and sign in to the console

1. Create an administrator user called `adminuser` in your AWS account. For instructions, see [Creating Your First IAM User and Administrators Group](#) in the *IAM User Guide*.
2. As a user, you can sign in to the AWS Management Console using a special URL. For more information, [How Users Sign In to Your Account](#) in the *IAM User Guide*.

For more information about IAM, see the following:

- [AWS Identity and Access Management \(IAM\)](#)
- [Getting started](#)
- [IAM User Guide](#)

## Next step

[Step 2: Getting started \(Console\) \(p. 8\)](#)

# Step 2: Getting started (Console)

The easiest way to learn how to use Amazon Lex V2 is by using the console. To get you started, we created the following exercises, all of which use the console:

- Exercise 1 — Create an Amazon Lex V2 bot using a blueprint, a predefined bot that provides all of the necessary bot configuration. You do only a minimum of work to test the end-to-end setup.

- Exercise 2 — Review the JSON structures sent between your client application and an Amazon Lex V2 bot.

#### Topics

- [Exercise 1: Create a bot from an example \(p. 9\)](#)
- [Exercise 2: Review the conversation flow \(p. 10\)](#)

## Exercise 1: Create a bot from an example

In this exercise, you create your first Amazon Lex V2 bot and test it in the Amazon Lex V2 console. For this exercise, you use the **OrderFlowers** example.

### Example overview

You use the **OrderFlowers** example to create an Amazon Lex V2 bot. For more information about the structure of a bot, see [How it works \(p. 3\)](#).

- **Intent** – OrderFlowers
- **Slot types** – One custom slot type called FlowerTypes with enumeration values: roses, lilies, and tulips.
- **Slots** – The intent requires the following information (that is, slots) before the bot can fulfill the intent.
  - PickupTime (AMAZON.TIME built-in type)
  - FlowerType (FlowerTypes custom type)
  - PickupDate (AMAZON.DATE built-in type)
- **Utterance** – The following sample utterances indicate the user's intent:
  - "I would like to pick up flowers."
  - "I would like to order some flowers."
- **Prompts** – After the bot identifies the intent, it uses the following prompts to fill the slots:
  - Prompt for the FlowerType slot – "What type of flowers would you like to order?"
  - Prompt for the PickupDate slot – "What day do you want the {FlowerType} to be picked up?"
  - Prompt for the PickupTime slot – "At what time do you want the {FlowerType} to be picked up?"
  - Confirmation statement – "Okay, your {FlowerType} will be ready for pickup by {PickupTime} on {PickupDate}. Does this sound okay?"

#### To create an Amazon Lex V2 bot (Console)

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose **Create bot**.
3. For the **Creation method**, choose **Start with an example**.
4. In the **Example bots** section, choose **OrderFlowers** from the list.
5. In the **Bot configuration** section give the bot a name and a description. The name must be unique in your account.
6. In the **Permissions** section, choose **Create a new role with basic Amazon Lex permissions**. This will create an AWS Identity and Access Management (IAM) role with the permissions that Amazon Lex V2 needs to run your bot.
7. In the **Children's Online Privacy Protection Act (COPPA)** section, make the appropriate choice.

8. In the **Session timeout** and **Advanced settings** sections, leave the defaults.
9. Choose **Next**. Amazon Lex V2 creates your bot.

After you create your bot, you must add one or more languages that the bot supports. A language contains the intents, slot types, and slots that the bot uses to converse with users.

#### To add a language to a bot

1. In the **Language** section, choose a supported language, and add a description.
2. Leave the **Voice interaction** and **Intent classification confidence score threshold** fields with their defaults.
3. Choose **Add language** to add the language to the bot.
4. After the language is added, choose **Done** to continue.

After you choose **Done**, the console opens the intent editor. You can use the intent editor to examine the intents used by the bot. When you are done examining the bot, you can test to bot.

#### To test the OrderFlowers bot

1. From the bottom menu, choose **Build**. Wait for the bot to build.
2. When the build is complete, choose **Test** to open the test window.
3. Test the bot. Start the conversation with one of the sample utterances, such as "I would like to pick up flowers."

## Next steps

Now that you've created your first bot using a template, you can use the console to create your own bot. For instruction on creating a custom bot, and for more information about creating bots, see [Building bots \(p. 17\)](#).

## Exercise 2: Review the conversation flow

In this exercise you review the JSON structures that are sent between your client application and the Amazon Lex V2 bot that you created in [Exercise 1: Create a bot from an example \(p. 9\)](#). The conversation uses the [RecognizeText](#) operation to generate the JSON structures. The [RecognizeUtterance](#) returns the same information as HTTP headers in the response.

The JSON structures are divided by each turn of the conversation. A *turn* is a request from the client application and a response from the bot.

### Turn 1

During the first turn of the conversation, the client application initiates the conversation with your bot. Both the URI and the body of the request provide information about the request.

```
POST /bots/botId/botAliases/botAliasId/botLocales/localeId/sessions/sessionId/text HTTP/1.1
Content-type: application/json

{
    "text": "I would like to order flowers"
}
```

- The URI identifies the bot that the client application is communicating with. It also includes a session identifier generated by the client application that identifies a specific conversation between a user and the bot.
- The body of the request contains the text that the user typed to the client application. In this case, only the text is sent, however you application can send additional information, such as request attributes or session state. For more information, see the [RecognizeText](#) operation.

From text, Amazon Lex V2 detects the user's intent, to order flowers. Amazon Lex V2 chooses one of the intent's slots (`FlowerType`) and one of the prompts for the slot, and then sends the following response to the client application. The client displays the response to the user.

```
{  
    "interpretations": [  
        {  
            "intent": {  
                "confirmationState": "None",  
                "name": "OrderFlowers",  
                "slots": {  
                    "FlowerType": null,  
                    "PickupDate": null,  
                    "PickupTime": null  
                },  
                "state": "InProgress"  
            },  
            "nluConfidence": {  
                "score": 0.95  
            }  
        },  
        {  
            "intent": {  
                "name": "FallbackIntent",  
                "slots": {}  
            }  
        }  
    ],  
    "messages": [  
        {  
            "content": "What type of flowers would you like to order?",  
            "contentType": "PlainText"  
        }  
    ],  
    "sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",  
    "sessionState": {  
        "dialogAction": {  
            "slotToElicit": "FlowerType",  
            "type": "ElicitSlot"  
        },  
        "intent": {  
            "confirmationState": "None",  
            "name": "OrderFlowers",  
            "slots": {  
                "FlowerType": null,  
                "PickupDate": null,  
                "PickupTime": null  
            },  
            "state": "InProgress"  
        },  
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"  
    }  
}
```

## Turns 2 and 3

In turn 2 and 3, the user responds to prompts from the Amazon Lex V2 bot with values that fill the FlowerType and PickupDate slots.

The URI for the second and third turns is the same as the first.

```
{  
    "text": "1 dozen roses"  
}
```

The response for turn 2 shows the FlowerType slot filled and provides a prompt to elicit the next slot value.

```
{  
    "interpretations": [  
        {  
            "intent": {  
                "confirmationState": "None",  
                "name": "OrderFlowers",  
                "slots": {  
                    "FlowerType": {  
                        "value": {  
                            "interpretedValue": "dozen roses",  
                            "originalValue": "dozen roses",  
                            "resolvedValues": []  
                        }  
                    },  
                    "PickupDate": null,  
                    "PickupTime": null  
                },  
                "state": "InProgress"  
            },  
            "nluConfidence": {  
                "score": 0.98  
            }  
        },  
        {  
            "intent": {  
                "name": "FallbackIntent",  
                "slots": {}  
            }  
        }  
    ],  
    "messages": [  
        {  
            "content": "What day do you want the dozen roses to be picked up?",  
            "contentType": "PlainText"  
        }  
    ],  
    "sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",  
    "sessionState": {  
        "dialogAction": {  
            "slotToElicit": "PickupDate",  
            "type": "ElicitSlot"  
        },  
        "intent": {  
            "confirmationState": "None",  
            "name": "OrderFlowers",  
            "slots": {  
                "FlowerType": {  
                    "value": {  
                        "interpretedValue": "dozen roses",  
                        "originalValue": "dozen roses",  
                        "resolvedValues": []  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        "value": {
            "interpretedValue": "dozen roses",
            "originalValue": "dozen roses",
            "resolvedValues": []
        }
    },
    "PickupDate": null,
    "PickupTime": null
},
"state": "InProgress"
},
"originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
}
```

## Turn 4

In turn 4, the user provides the final slot value for the intent, the time that the flowers are picked up.

```
{
    "text": "5 in the evening"
}
```

In the response, Amazon Lex V2 sends a confirmation prompt to the user to confirm that the order is correct. The dialogAction is set to ConfirmIntent and the confirmationState is None.

```
{
    "interpretations": [
        {
            "intent": {
                "confirmationState": "None",
                "name": "OrderFlowers",
                "slots": {
                    "FlowerType": {
                        "value": {
                            "interpretedValue": "dozen roses",
                            "originalValue": "dozen roses",
                            "resolvedValues": []
                        }
                    },
                    "PickupDate": {
                        "value": {
                            "interpretedValue": "2021-01-04",
                            "originalValue": "next monday",
                            "resolvedValues": [
                                "2021-01-04"
                            ]
                        }
                    },
                    "PickupTime": {
                        "value": {
                            "interpretedValue": "17:00",
                            "originalValue": "5 evening",
                            "resolvedValues": [
                                "17:00"
                            ]
                        }
                    }
                },
                "state": "InProgress"
            }
        }
    ]
}
```

```
        },
        "nluConfidence": {
            "score": 1.0
        }
    },
    {
        "intent": {
            "name": "FallbackIntent",
            "slots": {}
        }
    }
],
"messages": [
{
    "content": "Okay, your dozen roses will be ready for pickup by 17:00 on
2021-01-04. Does this sound okay?",
    "contentType": "PlainText"
}
],
"sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",
"sessionState": {
    "dialogAction": {
        "type": "ConfirmIntent"
    },
    "intent": {
        "confirmationState": "None",
        "name": "OrderFlowers",
        "slots": {
            "FlowerType": {
                "value": {
                    "interpretedValue": "dozen roses",
                    "originalValue": "dozen roses",
                    "resolvedValues": []
                }
            }
        },
        "PickupDate": {
            "value": {
                "interpretedValue": "2021-01-04",
                "originalValue": "next monday",
                "resolvedValues": [
                    "2021-01-04"
                ]
            }
        },
        "PickupTime": {
            "value": {
                "interpretedValue": "17:00",
                "originalValue": "5 evening",
                "resolvedValues": [
                    "17:00"
                ]
            }
        }
    },
    "state": "InProgress"
},
"originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
}
```

## Turn 5

In the final turn, the user responds with to the confirmation prompt.

```
{  
    "text": "yes"  
}
```

In the response, Amazon Lex V2 sends indicates that the intent has been fulfilled by setting the confirmationState to Confirmed and the dialogAction to close. All of the slot values are available to the client application.

```
{  
    "interpretations": [  
        {  
            "intent": {  
                "confirmationState": "Confirmed",  
                "name": "OrderFlowers",  
                "slots": {  
                    "FlowerType": {  
                        "value": {  
                            "interpretedValue": "dozen roses",  
                            "originalValue": "dozen roses",  
                            "resolvedValues": []  
                        }  
                    },  
                    "PickupDate": {  
                        "value": {  
                            "interpretedValue": "2021-01-04",  
                            "originalValue": "next monday",  
                            "resolvedValues": [  
                                "2021-01-04"  
                            ]  
                        }  
                    },  
                    "PickupTime": {  
                        "value": {  
                            "interpretedValue": "17:00",  
                            "originalValue": "5 evening",  
                            "resolvedValues": [  
                                "17:00"  
                            ]  
                        }  
                    },  
                    "state": "Fulfilled"  
                },  
                "nluConfidence": {  
                    "score": 1.0  
                }  
            },  
            {  
                "intent": {  
                    "name": "FallbackIntent",  
                    "slots": {}  
                }  
            }  
        ],  
        "messages": [  
            {  
                "content": "Thanks. ",  
                "contentType": "PlainText"  
            }  
        ],  
        "sessionId": "bf445a49-7165-4fcd-9a9c-a782493fba5c",  
        "sessionState": {  
            "dialogAction": {
```

```
        "type": "Close"
    },
    "intent": {
        "confirmationState": "Confirmed",
        "name": "OrderFlowers",
        "slots": {
            "FlowerType": {
                "value": {
                    "interpretedValue": "dozen roses",
                    "originalValue": "dozen roses",
                    "resolvedValues": []
                }
            },
            "PickupDate": {
                "value": {
                    "interpretedValue": "2021-01-04",
                    "originalValue": "next monday",
                    "resolvedValues": [
                        "2021-01-04"
                    ]
                }
            },
            "PickupTime": {
                "value": {
                    "interpretedValue": "17:00",
                    "originalValue": "5 evening",
                    "resolvedValues": [
                        "17:00"
                    ]
                }
            },
            "state": "Fulfilled"
        },
        "originatingRequestId": "9e8add70-4106-4a10-93f5-2ce2cb959e5f"
    }
}
```

# Building bots

You create an Amazon Lex V2 bot to interact with your users to elicit information to accomplish a task. For example, you can create a bot that gathers the information needed to order a bouquet of flowers or to book a hotel room.

To build a bot, you need the following information:

1. The language that the bot uses to interact with the customer. You can choose one or more languages, each language contains independent intents, slots, and slot types.
2. The intents, or goals, that the bot will help the user fulfill. A bot can contain one or more intents, such as ordering flowers, or booking a hotel and rental car. You need to decide which statements, or utterances, that the user makes to initiate the intent.
3. The information, or slots, that you need to gather from the user to fulfill an intent. For example, you might need to get the type of flowers from the user or the start date of a hotel reservation. You need to define one or more prompts that Amazon Lex V2 uses to elicit the slot value from the user.
4. The type of the slots that you need from the user. You may need to create a custom slot type, such as a list of flowers that a user can order, or you can use a built-in slot type, such as using the AMAZON.Date slot type for the start date of a reservation.
5. The user interaction flow within and between intents. You can configure the conversation flow to define the interaction between the user and the bot once the intent is invoked. You can create a Lambda function to validate and fulfill the intent.

## Topics

- [Understanding conversation flow management \(p. 18\)](#)
- [Creating a bot \(p. 19\)](#)
- [Adding a language \(p. 19\)](#)
- [Visual conversation builder \(p. 19\)](#)
- [Adding intents \(p. 32\)](#)
- [Adding slot types \(p. 45\)](#)
- [Testing a bot using the console \(p. 45\)](#)
- [Creating versions \(p. 48\)](#)
- [Creating conversation paths \(p. 49\)](#)
- [Built-in intents and slot types \(p. 61\)](#)
- [Using a custom grammar slot type \(p. 83\)](#)
- [Creating custom slot types \(p. 181\)](#)
- [Using composite slots \(p. 182\)](#)
- [Creating a custom vocabulary to improve speech recognition \(p. 187\)](#)
- [Using multiple values in a slot \(p. 193\)](#)
- [Using an AWS Lambda function \(p. 194\)](#)

## Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow](#)

[management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Understanding conversation flow management

On August 17, 2022 Amazon Lex V2 released a change to the way that conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation.

Before the change, Amazon Lex V2 managed the conversation by eliciting slots based on their priorities in intent. You could modify this behavior dynamically and change the conversation path based on user inputs by using DialogAction in Lambda function. This could be done by keeping track of the current state of the conversation and programmatically deciding what to do next based on the session state.

With this change, you can create conversational paths and conditional branches using the Amazon Lex V2 console or APIs without using a Lambda function. Amazon Lex V2 keeps track of the state of the conversation and makes decisions about what to do next based on conditions defined when the bot is created. This enables you to easily create complex conversations while designing your bot.

These changes give you complete control over the conversation with your customer. However, you are not required to define a path. If you do not specify an explicit conversation path, Amazon Lex V2 creates a default path for the conversation based on the priority of slots in your intent. You can continue to use Lambda functions to define conversation paths dynamically. In such scenario, the conversation will resume based on the session state configured in the Lambda function.

This update provides the following:

- A new console experience for creating bots with complex conversation flows.
- Updates to the existing APIs for creating bots to support the new conversation flows.
- An initial response to send a message on intent invocation.
- New responses for slot elicitation, Lambda invocation as dialog code hook and confirmation.
- Ability to specify next steps at each turn of the conversation.
- Evaluation of conditions to design multiple conversation paths.
- Setting of slot values and session attributes at any point during the conversation.

Note the following for older bots:

- Bots created before August 17, 2022 continue to use the old mechanism to manage conversation flows. Bots created after that date use the new way of conversation flow management.
- New bots created via imports after August 17, 2022 use the new conversation flow management. Imports on existing bots will continue to use the old way of conversation management.
- To enable the new conversation flow management on an existing bot created prior to August 17, 2022, export the bot and then import the bot using a new bot name. The newly created bot from the import will use the new conversation flow management

Note the following for new bots created after August 17, 2022:

- Amazon Lex V2 follows the defined conversation flow exactly as designed to deliver the desired experience. You should configure all flow branches in order to avoid default conversation paths during runtime.
- Conversation steps following a code hook should be fully configured, because incomplete steps can lead to bot execution failure. We recommend that you validate bots created before August 17, 2022, because for these bots, there is no automatic validation of conversation steps following a code hook.

# Creating a bot

Start creating your bot by defining the name, description and some basic information.

## To create a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose **Create bot**.
3. In the **Creation method** section, choose **Create**.
4. In the **Bot configuration** section, give the bot a name and an optional description.
5. In the **IAM permissions** section, choose an AWS Identity and Access Management (IAM) role that provides Amazon Lex V2 permission to access other AWS services, such as Amazon CloudWatch. You can have Amazon Lex V2 create the role, or you can choose an existing role with CloudWatch permissions.
6. In the **Children's Online Privacy Protection Act (COPPA)** section, choose the appropriate response.
7. In the **Idle session timeout** section, choose the duration that Amazon Lex V2 keeps a session with a user open. Amazon Lex V2 maintains session variables for the duration of the session so that your bot can resume a conversation with the same variables.
8. In the **Advanced settings** section add tags that help identify the bot, and can be used to control access and monitor resources.
9. Choose **Next** to create the bot and move to adding a language.

# Adding a language

You add one or more languages and locales to your bot to enable it to communicate with users in their languages. You define the intents, slots, and slot types separately for each language so that the utterances, prompts, and slot values are specific to the language.

Your bot must contain at least one language.

## To add a language to your bot

1. In the **New language** section, choose the language that you want to use. You can add a description to help identify the language in lists.
2. If your bot supports voice interaction, in the **Voice interaction** section, choose the Amazon Polly voice that Amazon Lex V2 uses to communicate with the user. If your bot doesn't support voice, choose **None**.
3. For the **Intent classification confidence score threshold**, set the value that Amazon Lex V2 uses to determine whether an intent is the correct intent. You can adjust this value after testing your bot.
4. Choose **Add**.

# Visual conversation builder

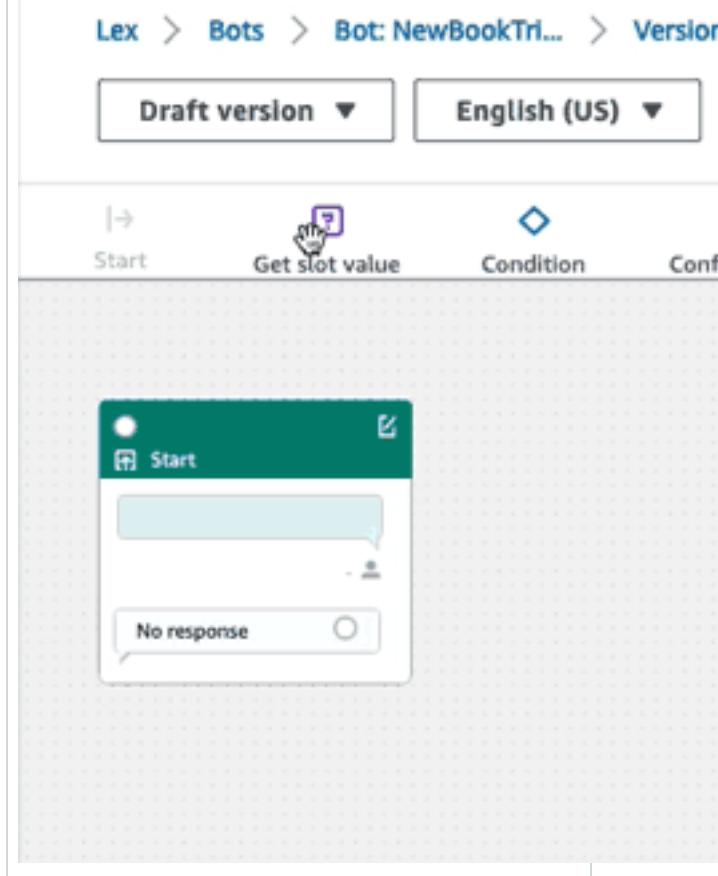
Visual conversation builder is a drag and drop conversation builder to easily design and visualize conversation paths by using intents within a rich visual environment.

The screenshot shows the Amazon Lex V2 developer console. At the top, there's a navigation bar with the AWS logo, a services menu, and a search bar. Below the search bar, the title "Amazon Lex" is displayed, along with a back button labeled "Back to intents list (3)". On the left side of the main area, there are three intent names listed: "BookCar", "BookHotel" (highlighted in orange), and "FallbackIntent". On the right side, there's a "Draft version" dropdown, a "Start" button with a help icon, and a "Get slot value" button with a question mark icon. A visual conversation builder interface is visible, showing a green "Start" node connected to a white "Book a hotel" message node, which then connects to a grey "No response" node.

Visual conversation builder offers a more intuitive user interface with the ability to visualize and modify the conversation flow. By dragging and dropping the blocks, you can extend an existing flow or reorder the conversation steps. You can develop conversation flow with complex branching without writing any Lambda code.

This change will help decouple the conversation flow design from other business logic in Lambda. Visual conversation builder can be used in conjunction with the existing intent editor and can be used to build conversation flows. However, it is recommended to use the visual editor view for more complex conversation flows.

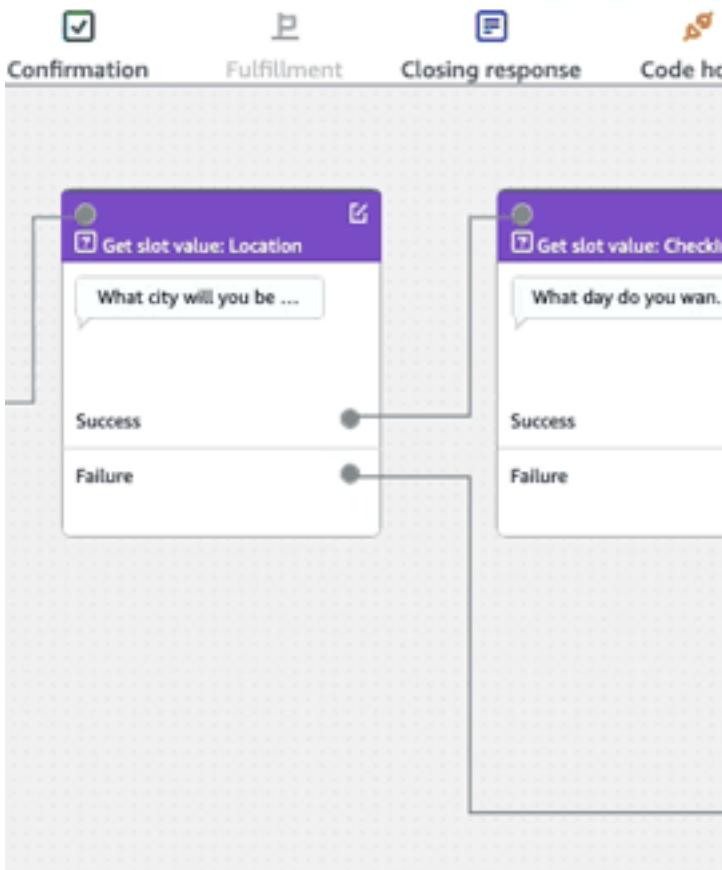
When you save an intent, Amazon Lex V2 can auto-connect intents when it determines that there are missed connections, Amazon Lex V2 will suggest a connection, or you can select your own connection for the block.

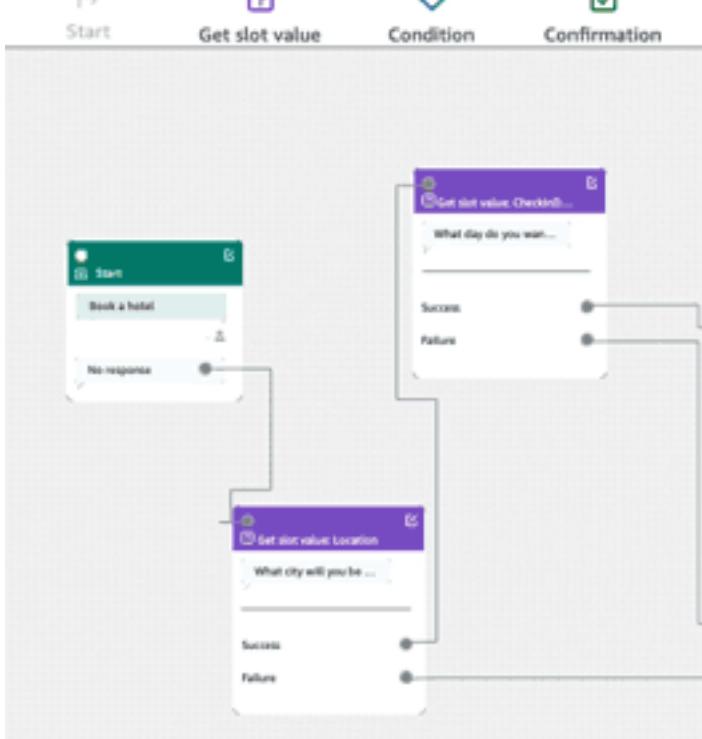
Action	Example
Adding a block to the workspace	 The screenshot shows the Amazon Lex V2 console's visual editor. At the top, there are navigation links: Lex > Bots > Bot: NewBookTri... > Versions. Below these are dropdown menus for Draft version (set to Draft version) and English (US). The workspace contains a sequence of blocks: a green Start block at the top, followed by a light blue Get slot value block with a purple hand icon pointing to its input slot, and a blue Condition block to its right. A grey Confirmation block is partially visible on the far right. A flow arrow connects the Start block to the Get slot value block. The workspace has a light gray background with a white central area for blocks.

Action	Example
Making a connection between blocks	<p>Lex &gt; Bots &gt; Bot: NewBookTri... &gt; Versions</p> <p>Draft version ▾ English (US) ▾</p> <p>Start Get slot value Condition Confirmation</p> <p>The screenshot shows the Amazon Lex V2 developer console. At the top, there's a navigation bar with 'Lex', 'Bots', 'Bot: NewBookTri...', and 'Versions'. Below that are dropdown menus for 'Draft version' and 'English (US)'. A toolbar at the top has four icons: 'Start', 'Get slot value', 'Condition', and 'Confirmation'. The main area contains two blocks: a green 'Start' block and a purple 'Get slot value' block. A connection line is being drawn from the 'Start' block towards the 'Get slot value' block, indicating a flow or condition being established.</p>

Action	Example
Opening the configuration panel on a block	<p>The screenshot shows the Amazon Lex V2 Visual Conversation Builder interface. At the top, there are four icons: 'Start' (green), 'Get slot value' (purple), 'Condition' (blue diamond), and 'Confirmation' (green checkmark). Below these are two main components. On the left is a green 'Start' block labeled 'Book a hotel'. It has a speech bubble below it with the text 'No response'. On the right is a purple 'Get slot value' block labeled 'Locate city'. A line connects the 'Get slot value' block to the 'No response' speech bubble of the 'Start' block. To the right of the 'Get slot value' block is a vertical stack of three boxes: 'Success' (light blue), 'Failure' (light red), and 'Error' (light green). At the bottom of the screen are three buttons: 'Editor' (white), 'Visual builder' (grayed out), and 'New' (blue).</p>

Action	Example
Zoom to fit	<p>The screenshot shows a visual conversation builder interface. At the top, there are icons for Start, Get slot value, Condition, Confirmation, and Fulfill. Below these, a purple action card is displayed with the title "Get slot value: Location". Inside the card, there is a speech bubble containing the text "What city will you be ...". Below the speech bubble is a horizontal line. Underneath the line, there are two options: "Success" and "Failure", each preceded by a small circular icon. The background of the interface has a grid pattern.</p>

Action	Example
Delete a block from the conversation flow	 <p>The screenshot shows a visual conversation builder interface. At the top, there are four tabs: 'Confirmation' (with a checkmark icon), 'Fulfillment' (with a document icon), 'Closing response' (with a mail icon), and 'Code hook' (with a gear icon). Below the tabs, a conversation flow is displayed. It starts with a purple block labeled 'Get slot value: Location' containing the text 'What city will you be ...'. This block has two outgoing arrows: one labeled 'Success' and one labeled 'Failure'. To the right of this is another purple block labeled 'Get slot value: Checkin' containing the text 'What day do you wan...'. This block also has two outgoing arrows: one labeled 'Success' and one labeled 'Failure'. The flow is represented by a light gray rectangular area.</p>

Action	Example
Auto clean the workspace	 <p>The screenshot shows the Amazon Lex V2 Visual Conversation Builder interface. At the top, there are four icons: 'Start' (green), 'Get slot value' (purple), 'Condition' (blue diamond), and 'Confirmation' (green checkmark). Below these are two main blocks: a 'Start' block labeled 'Book a hotel' and a 'Get slot value' block labeled 'What day do you want...'. Each block has an 'Edit' icon in the top right corner. Below each block is a configuration panel with fields for 'Success' and 'Failure' messages. Arrows indicate the flow from the 'Start' block to the 'Get slot value' block, and from there to the 'Condition' and 'Confirmation' blocks.</p>

### Terminology:

**Block** – The basic building unit of a conversation flow. Each block has a specific functionality to handle different use cases of a conversation.

**Port** – Each block contains ports, which can be used to connect one block to another. Blocks can contain input ports and output ports. Each output port represents a particular functional variation of a block (such as errors, timeouts, or success).

**Edge** – An edge is a connection between the output port of one block to the input port of another block. It is a part of a branch in a conversation flow.

**Conversation flow** – A set of blocks connected by edges that describes intent level interactions with a customer.

### Blocks

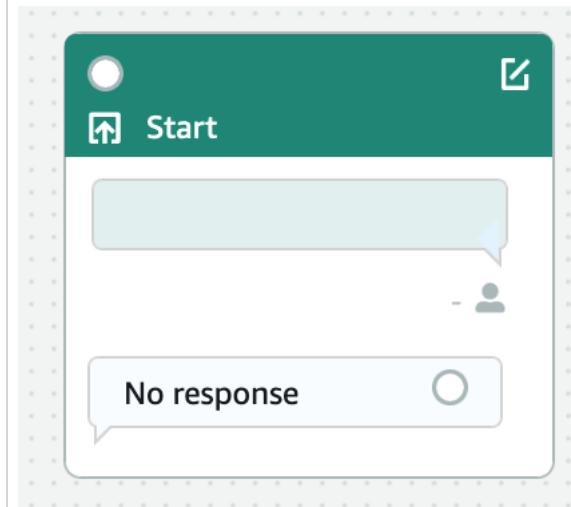
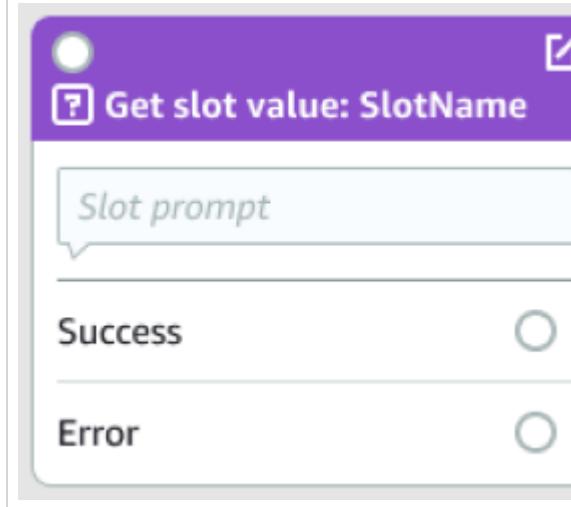
Blocks are the building blocks of a conversation flow design and represent different states within the intent that spans from the start of the intent to user input to the closing.

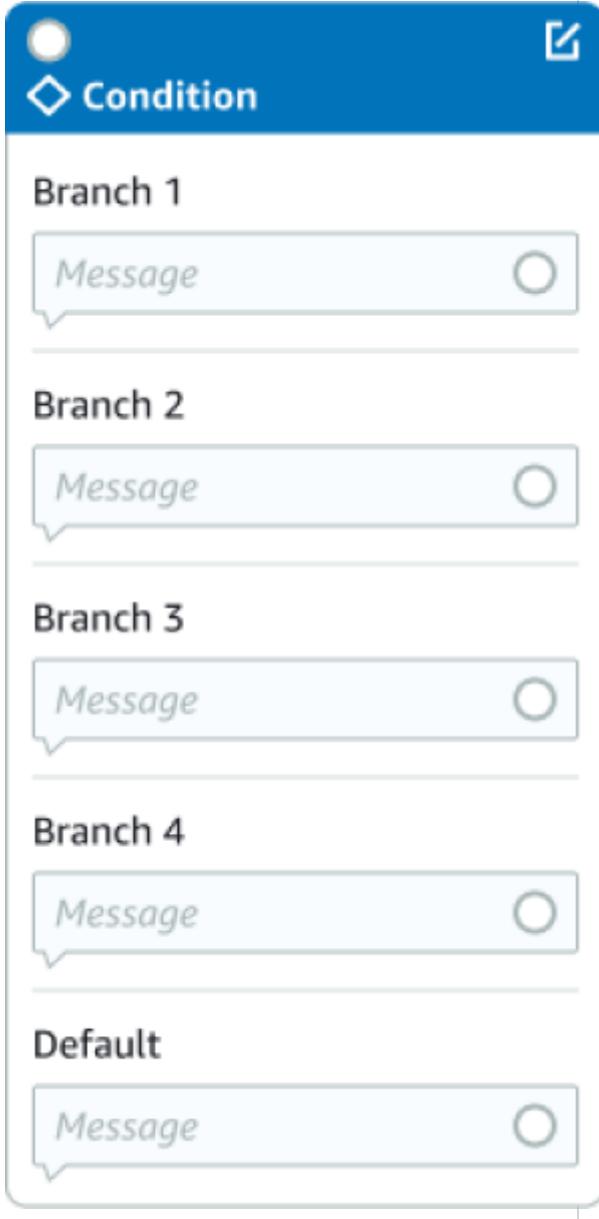
Each block has an entry point and one or many exit points based on the block type. Each exit point can be configured with a corresponding message as the conversation proceeds through the exit points. For blocks with multiple exit points, exit points relate to the status corresponding to the node. For a condition node, the exit points represent the different conditions.

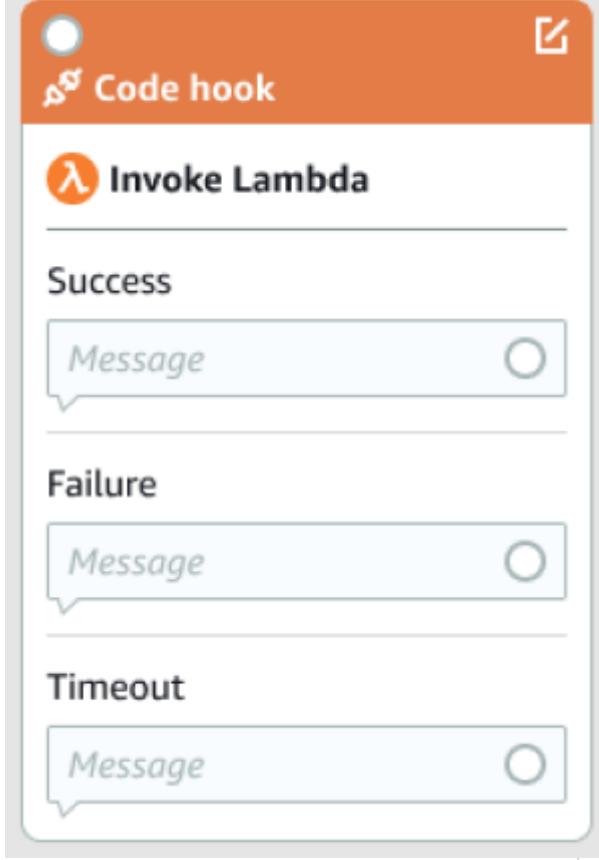
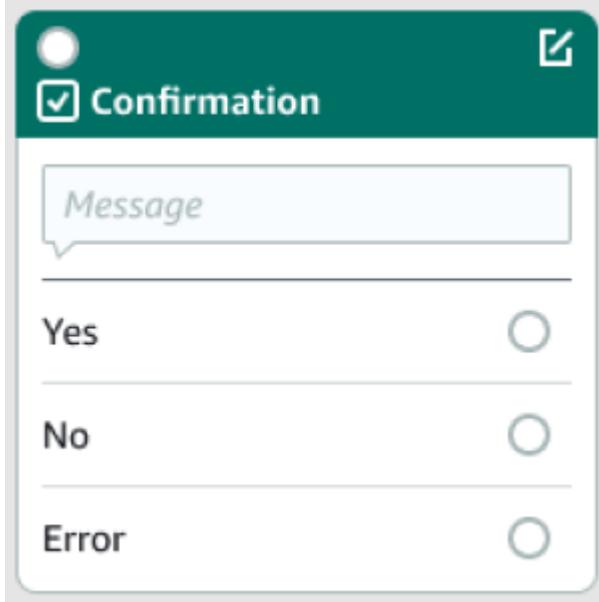
Each block has a configuration panel, which opens by clicking on the **Edit** icon on the top right corner of the block. The configuration panel contains detailed fields that can be configured to correspond with each block.

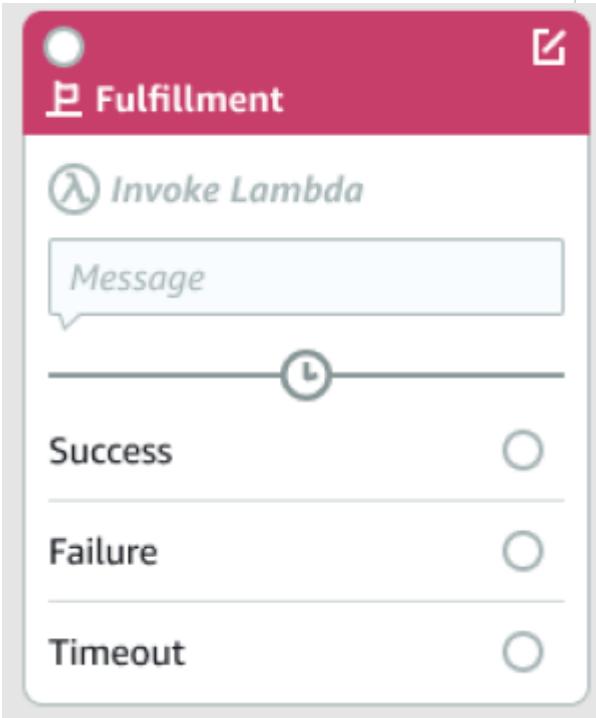
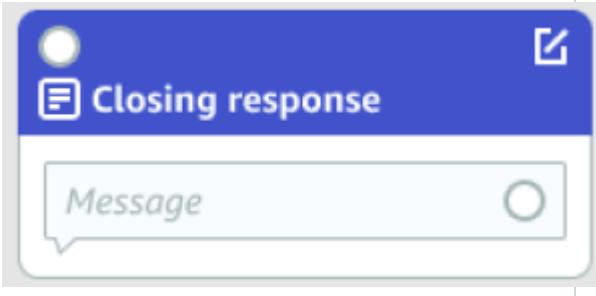
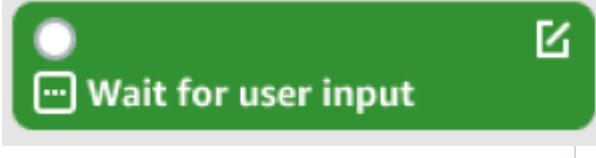
The bot prompts and messages can be configured directly on the node by dragging a new block, or they can be modified within the right panel, along with other attributes of the block.

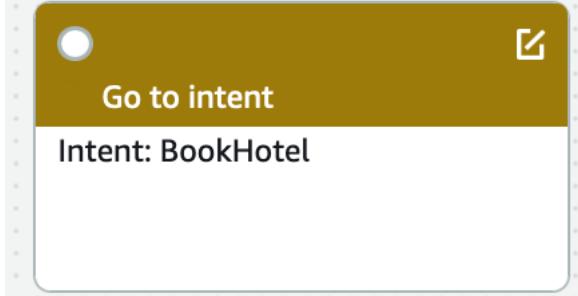
**Block types** – Here are the block types that you can use with visual conversation builder.

Block Type	Block
<b>Start</b> – The root or first block of the conversation flow. This block can also be configured such that the bot can send an initial response (message the intent has been recognized). For more information, see <a href="#">Initial response (p. 33)</a> .	
<b>Get slot value</b> – This block tries to elicit value for a single slot. This block has a setting to wait for customer response to the slot elicitation prompt. For more information, see <a href="#">Slots (p. 35)</a> .	

Block Type	Block
<p><b>Condition</b> – This block will contain conditionals. It contains up to 4 custom branches (with conditions) and one default branch. For more information, see <a href="#">Add conditions to branch conversations (p. 54)</a>.</p>	

Block Type	Block
<p><b>Dialog code hook</b> – This block handles invocation of the dialog Lambda function. This block contains bot responses based on dialog Lambda function succeeding, failing, or timing out. For more information, see <a href="#">Invoke dialog code hook (p. 58)</a>.</p>	 <p><b>Code hook</b></p> <p><b>Invoke Lambda</b></p> <p><b>Success</b></p> <p><b>Failure</b></p> <p><b>Timeout</b></p>
<p><b>Confirmation</b> – This block queries the customer before fulfillment of the intent. It contains bot responses based on customer saying yes or no to the confirmation prompt. For more information, see <a href="#">Confirmation (p. 39)</a>.</p>	 <p><b>Confirmation</b></p> <p><b>Message</b></p> <p><b>Yes</b></p> <p><b>No</b></p> <p><b>Error</b></p>

Block Type	Block
<b>Fulfillment</b> – This block handles fulfillment of intent, usually after slots elicitation. It can be configured to invoke Lambda functions, as well as respond with messages, if fulfillment succeeds or fails. For more information, see <a href="#">Fulfillment (p. 42)</a> .	 <p><b>Fulfillment</b></p> <p>Λ <i>Invoke Lambda</i></p> <p><i>Message</i></p> <hr/> <p>Success <input type="radio"/></p> <hr/> <p>Failure <input type="radio"/></p> <hr/> <p>Timeout <input type="radio"/></p>
<b>Closing response</b> – This block allows the bot to respond with a message before ending the conversation. For more information, see <a href="#">Closing response (p. 44)</a> .	 <p><b>Closing response</b></p> <p><i>Message</i></p>
<b>End conversation</b> – This block indicates the end of the conversation flow.	 <p>→ <b>End conversation</b></p>
<b>Wait for user input</b> – This block can be used to capture input from the customer and switch to another intent based on the utterance.	 <p><b>Wait for user input</b></p>

Block Type	Block
<b>Go to intent</b> – This block can be used to go to a new intent, or to directly elicit a specific slot of that intent.	

### Port types

All blocks contain one input port, which is used to connect its parent blocks. The conversation can only flow to a particular block's input port from its parent block's output port. However, blocks can contain zero, one, or many output ports. The blocks without any output ports signify the end of the conversation flow in the current intent (GoToIntent, EndConversation, WaitForUserInput).

### Rules of intent design:

- All flows in an intent begin with the start block.
- Messages corresponding to each exit point are optional.
- You can configure the blocks to set values corresponding to each exit point in the configuration panel.
- Only a single start, confirmation, fulfillment and closing blocks can exist in a single flow within an intent. Multiple conditions, dialog code hook, get slot values, end conversation, transfer, and wait for user input blocks may exist.
- A condition block cannot have a direct connection to a condition block. The same applies for dialog code hook.
- Circular flows are allowed three blocks, but an incoming connector to Start Intent is not allowed.
- An optional slot doesn't have an incoming connector or an outgoing connection and is primarily used to capture any data present during intent elicitation. Every other slot that is part of the conversation path must be a mandatory slot.

### Blocks:

- The start block must have an outgoing edge.
- Every get slot value block must have an outgoing edge from the success port, if the slot is required.
- Every condition block must have an outgoing edge from each branch if the block is active.
- A condition block cannot have more than one parent.
- An active condition block must have an incoming edge.
- Every active code hook block must have an outgoing edge from each port: success, failure, and timeout.
- An active code hook block must have an incoming edge.
- An active confirmation block must have an incoming edge.
- An active fulfillment block must have an incoming edge.
- An active closing block must have an incoming edge.
- A condition block must have at least one non-default branch.
- A go to intent block must have an intent specified.

Edges:

- A condition block cannot be connected to another condition block.
- A code hook block cannot be connected to another code hook block.
- A condition block can only be connected to zero or one code hook block.
- The connection (code hook -> condition -> code hook) is invalid.
- A fulfillment block cannot have a code hook block as a child.
- A condition block, which is a child of the fulfillment block, cannot have a code hook block child.
- A closing block cannot have a code hook block as a child.
- A condition block which is a child of the closing block cannot have a code hook block child.
- A start, confirmation, or get slot value block can have no more than one code hook block in its dependency chain.

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Adding intents

Intents are the goals that your users want to accomplish, such as ordering flowers or booking a hotel. Your bot needs to have at least one intent.

By default, all bots contain a single built-in intent, the fallback intent. This intent is used when Amazon Lex V2 does not recognize any other intent. For example, if a user says "I want to order flowers" to a hotel booking intent, the fallback intent is triggered.

### To add an intent

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the list of bots, choose the bot that you want to add the intent to, then from **Add languages** choose **View languages**.
3. Choose the language to add the intent to, then choose **Intents**.
4. Choose **Add intent**, give your intent a name, and then choose **Add**.
5. In the intent editor, add the details of your intent.
  - **Conversation flow** – Use the conversation flow diagram to see how a dialog with your bot might look. You can choose different sections of the conversation to jump to that section of the intent editor.
  - **Intent details** – Give the intent a name and description to help identify the purpose of the intent. You can also see the unique identifier that Amazon Lex V2 assigned to the intent.
  - **Contexts** – Set the input and output contexts for the intent. A context is a state variable associated with an intent. An output context is set when an intent is fulfilled. An intent with an input context can only be recognized if the context is active. An intent with no input contexts can always be recognized.
  - **Sample utterances** – You should provide 10 or more phrases that you expect your users to use to initiate an intent. Amazon Lex V2 generalizes from these phrases to recognize that the user wants to initiate the intent.

- **Initial response** – The initial message sent to the user after the intent is invoked. You can provide responses, initialize values, and define the next step that Amazon Lex V2 takes to respond to the user at the beginning of the intent.
  - **Slots** – Define the slots, or parameters, required to fulfill the intent. Each slot has a type that defines the values that can be entered in the slot. You can choose from your custom slot types, or you can choose a built-in slot type.
  - **Confirmation** – These prompts and responses are used to confirm or decline fulfillment of the intent. The confirmation prompt asks the user to review slot values. For example, "I've booked a hotel room for Friday. Is this correct?" The declination response is sent to the user when they decline the confirmation. You can provide responses, set values, and define the next step that Amazon Lex V2 takes corresponding to a confirmation or declination response from the user.
  - **Fulfillment** – Response sent to the user during the course of fulfillment. You can set fulfillment progress updates at the start of fulfillment and periodically while the fulfillment is in progress. For example, "I'm changing your password, this may take a few minutes" and "I'm still working on your request." Fulfillment updates are used only for streaming conversations. You can also set a post-fulfillment success message, a failure message, and a timeout message. You can send post-fulfillment messages for both streaming and regular conversations. For example, if the fulfillment succeeds, you can send "I've changed your password." If the fulfillment doesn't succeed, you can send a response with more information, such as "I couldn't change your password, contact the help desk for assistance." If the fulfillment takes too long to process and exceeds the configured timeout period, you can send a message informing the user, such as "Our servers are very busy right now. Try your request again later." You can provide responses, set values, and define the next step that Amazon Lex V2 takes to respond to the user.
  - **Closing responses** – Response sent to the user after the intent is fulfilled and all other messages are played. For example, a thank you for booking a hotel room. Or it can prompt the user to start a different intent, such as, "Thank you for booking a room, would you like to book a rental car?" You can provide responses and configure follow-up next actions after fulfilling the intent and responding with the closing response.
  - **Code hooks** – Indicate whether you are using an AWS Lambda function to initialize the intent and validate user input. You specify the Lambda function in the alias that you use to run the bot.
6. Choose **Save intent** to save the intent.

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Initial response

The initial response is sent to the user after Amazon Lex V2 determines the intent and before it starts to elicit slot values. You can use this response to inform the user of the intent that was recognized and to prepare them for the information that you collect to fulfill the intent.

For example, if the intent is to schedule a service appointment for a car, the initial response might be:

I can help you schedule an appointment. You'll need to provide the make, model, and year of your car.

An initial response message isn't required. If you don't provide one, Amazon Lex V2 will continue to follow the next step of the initial response.

You can configure the following options within the initial response:

- **Configure next step** – You can provide the next step in the conversation such as jumping to a specific dialog action, eliciting a particular slot, or jumping to a different intent. For more information, see [Configure next steps in the conversation \(p. 53\)](#).
- **Set values** – You can set values for slots and session attributes. For more information, see [Set values during the conversation \(p. 54\)](#).
- **Add conditional branching** – You can apply conditions after playing the initial response. When a condition evaluates to true, the actions that you define are taken. For more information, see [Add conditions to branch conversations \(p. 54\)](#).
- **Execute dialog code hook** – You can define a Lambda code hook to initialize data and execute business logic. For more information, see [Invoke dialog code hook \(p. 58\)](#). If the option to execute Lambda function is enabled for the intent, the dialog code hook will be executed by default. You can disable dialog code hook by toggling the **Active** button.

In the absence of a condition or an explicit next step, Amazon Lex V2 moves to the next slot in priority order.

## User request acknowledgement Info

You can provide messages to acknowledge a user's request. You can provide responses, set values, and next steps. You can also add conditions to the response based on conditions.

### ▼ Response for acknowledging the user's request

*Message:* -

#### Message - optional

*Okay, I can help you with that*

### ► Variations - optional

#### More response options

Add custom payloads, SSML, and card groups.

### ► Set values

-

Next step in conversation

*Execute dialog code hook*

### + Add conditional branching

## Dialog code hook Info

You can enable Lambda functions to manage initialize the conversation.

### ► Lambda dialog code hook

*Invoke Lambda for user request validation: Yes*

### Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Slots

Slots are values provided by the user to fulfill the intent. There are two types of slots:

- **Built-in slot type** – You can use built-in slot types to capture standard values such as number, name, and city. For a list of supported built-in slot types, see [Built-in slot types \(p. 75\)](#).
- **Custom slot type** – You can use custom slot types to capture custom values specific to the intent. For example, you can use a custom slot type to capture account type as "Checking" or "Savings". For more information, see [Creating custom slot types \(p. 181\)](#).

To define a slot in an intent, you have to configure the following:

- **Slot info** – This field contains a name and an optional description for the slot. For example, you can provide slot name as "AccountNumber" to capture account numbers. If the slot is required as part of the conversation flow for fulfilling the intent, it must be marked as required.
- **Slot type** – A slot type defines the list of values that a slot can accept. You can create a custom slot type or use a pre-defined slot type.
- **Slot prompt** – A slot prompt is a question posed to the user to gather information. You can configure the number of retries used to gather information and the variation of the prompt used for each retry. You can also enable a Lambda function invocation after each retry to process the input captured and attempt to resolve to a valid input.
- **Wait and Continue (optional)** – By enabling this behavior, users can say phrases such as "hold on a second" to make the bot wait for them to find the information and provide it. This is enabled only for streaming conversations. For more information, see [Enabling the bot to wait for the user to provide more information \(p. 271\)](#).
- **Slot capture responses** – You can configure a success response and a failure response depending on the success or failure of capturing the slot value from user input.
- **Conditional branching** – You can apply conditions after playing the initial response. When a condition evaluates to true, the actions that you define are taken. For more information, see [Add conditions to branch conversations \(p. 54\)](#).
- **Dialog code hook** – You can also use a Lambda code hook to validate the slot values and execute business logic. For more information, see [Invoke dialog code hook \(p. 58\)](#).
- **User input type** – You can configure input type so the bot can accept a specific modality. By default, both audio and DTMF modalities are accepted. You can selectively set it to audio only or DTMF only.
- **Audio input timeouts and lengths** – You can set audio timeouts including voice timeout and silence timeout. Also, you can set the max audio length.
- **DTMF input timeout, characters, and lengths** – You can set the DTMF timeout along with the deletion character and the end character. Also, you can set the max DTMF length.
- **Text length** – You can set the max length for text modality.

After the slot prompt is played, the user provides the slot value as an input. If Amazon Lex V2 does not understand a slot value provided by the user, it retries eliciting the slot until it understands a value or until it exceeds the maximum number of retries that you configured for the slot. Using the advanced retry settings you can configure the timeouts, restrict the type of input, and enable or disable interrupt for the initial prompt and retries. After each attempt at capturing the input, Amazon Lex V2 can call the Lambda function configured for the bot with an invocation label provided for retries. You can use the Lambda function, for example, to apply your business logic to attempt resolving it to a valid value. This Lambda function can be enabled within **Advanced options** for slot prompts.

## Slot prompts Info

Prompts to elicit the slot.

### ► Bot elicits information

*Message: What is your account number?*

You can define responses that the bot should send to the user once the slot value is entered or if the maximum number of retries is exceeded. For example, for a bot for scheduling service for a car, you can send a message to the user when the vehicle identification number (VIN) is entered:

Thank you for providing the VIN number of your car. I will now proceed to schedule an appointment.

You can create two responses:

- **Success response** – sent when Amazon Lex V2 understands a slot value.
- **Failure response** – sent when Amazon Lex V2 can't understand a slot value from the user after the maximum number of retries.

You can set values, configure the next steps, and apply conditions that correspond to each response to design the conversation flow.

In the absence of a condition or an explicit next step, Amazon Lex V2 moves to the next slot in priority order.

## Slot capture: success response Info

You can provide responses, set values, and next steps. You can also branch based on conditions.

- ▶ Response when user provides slot value

*Message:* -

- ▶ Set values

-

Next step in conversation

*Elicit a slot*

 Add conditional branching

## Slot capture: failure response Info

You can provide responses, set values, and next steps. You can also branch based on conditions.

- ▶ Response when slot value isn't understood

*Message:* -

- ▶ Set values

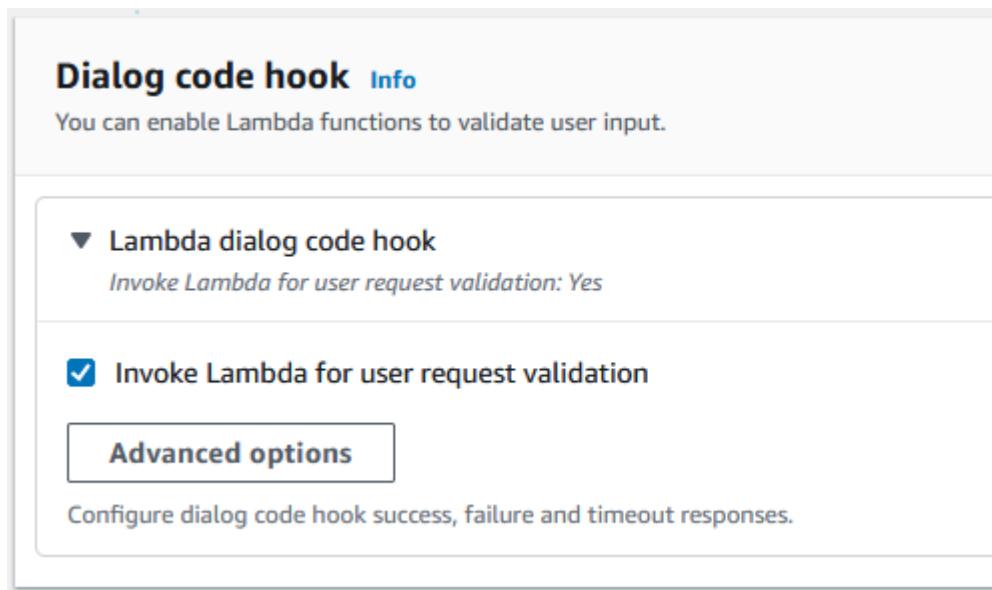
-

Next step in conversation

*Switch to intent: FallbackIntent*

 Add conditional branching

After a user has entered a slot value, you can validate the value and determine what the next action should be. The validation is performed in the dialog code hook using a Lambda function that you define for the language and alias of your bot. For example, use the validation function to make sure that the entered value is in the correct range, or to make sure that it is in the correct format. You can specify an invocation label for the dialog code hook. This invocation label can be used in Lambda function to write the business logic corresponding to the slot elicitation.



Slots that are not required for the intent are not part of the main conversation flow. These slots can be part of sample utterances for the intent and can be optionally populated with values if the user input contains them. For example, a business intelligence bot can process inputs such as "What is the sales for April?" and "What is the sales for April in San Diego?". In such cases, the city slot is not required for the intent and the business logic can be configured to use the slot value if present.

Slots not required for the intent cannot be elicited using next steps and can be populated only during intent elicitation (as in above example) or can be elicited by setting the dialog state within the Lambda function. If the slot is elicited using the Lambda function, you must use the Lambda function to decide the next step in the conversation after the slot elicitation is completed. To enable support for next step while building the bot, you must mark the slot as required for the intent.

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Confirmation

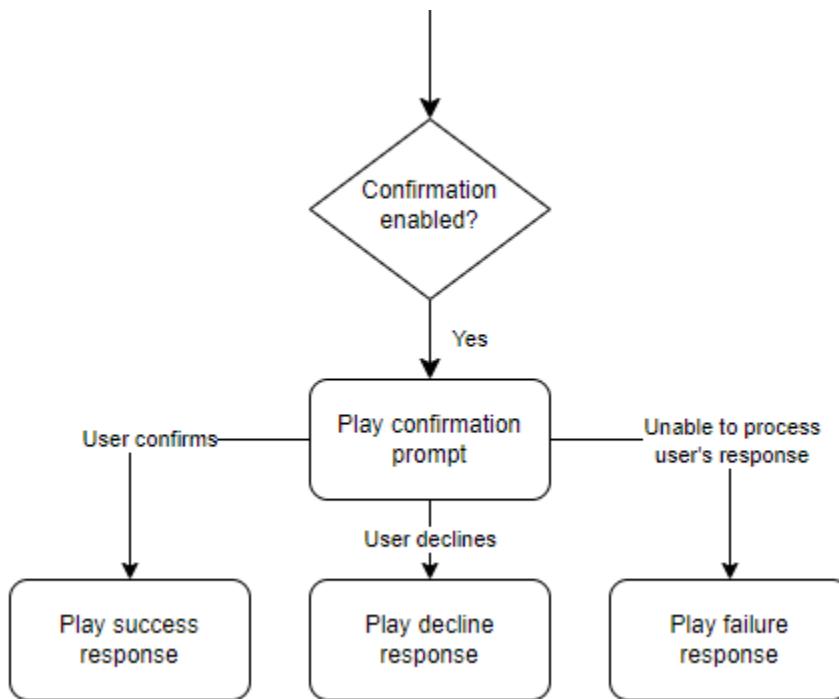
After the conversation with the user is complete and the slot values for the intent are filled, you can configure a confirmation prompt to ask the user if the slot values are correct. For example, a bot that schedules service appointments for cars might prompt the user with the following:

I've got service for your 2017 Honda Civic scheduled for March 25th at 3:00 PM. Is that all right?

You can define 3 types of responses to the confirmation prompt:

- **Confirmation response** – This response is sent to the user when the user confirms the intent. For example, after the user replies "yes" to the prompt "do you want to place the order?"
- **Decline response** – This response is sent to the user when the user declines the intent. For example, after the user replies "no" to the prompt "do you want to place the order?"

- **Failure response** – This response is sent to the user when the confirmation prompt can't be processed. For example, if the user's response couldn't be understood or couldn't be resolved to a yes or a no.



If you don't specify a confirmation prompt, Amazon Lex V2 moves to the fulfillment step or the closing response.

You can set values, configure the next steps and apply conditions corresponding to each response to design the conversation flow. In the absence of a condition or an explicit next step, Amazon Lex V2 moves to the fulfillment step.

You can also enable the dialog code hook to validate the information captured in the intent prior to sending it for fulfillment. To use a code hook, enable the dialog code hook in the confirmation prompt advanced options. In addition, configure the next step of the previous state to execute the dialog code hook. For more information, see [Invoke dialog code hook \(p. 58\)](#).

**Note**

If you use a code hook to trigger the confirmation step at runtime, you must mark the confirmation step as **Active** at build time.

**Confirmation and decline options** [Info](#) X

### Confirmation prompt

These messages are used to confirm an intent.

► Bot elicits information  
*Message: Can I go ahead with your request?*

### Confirmation response

When the user confirms a confirmation response, these are the responses that Amazon Lex uses.

► Bot replies to confirmation  
*Message: -*

► Set values Next step in conversation  
*End conversation*

[+ Add conditional branching](#)

### Decline response

When the user declines a confirmation prompt, these are the responses Amazon Lex uses.

► Bot confirms cancellation  
*Message: Okay. Your request will not be submitted.*

► Set values Next step in conversation  
*End conversation*

[+ Add conditional branching](#)

### Failure response

When there is a problem processing the user's response to the confirmation prompt, Amazon Lex responds with this message.

► Bot informs user of problem  
*Message: -*

► Set values Next step in conversation  
*Switch to intent: FallbackIntent*

[+ Add conditional branching](#)

### Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Using a Lambda function to validate an intent.

You can define a Lambda code hook to validate the intent before you send it for fulfillment. To use a code hook, enable the dialog code hook in the confirmation prompt advanced options.

When you use a code hook, you can define the actions that Amazon Lex V2 takes after the code hook runs. You can create three types of responses:

- **Success response** – Sent to the user when the code hook completes successfully.
- **Failure response** – Sent to the user when the code hook doesn't run successfully or when the code hook returns Failure in the response.
- **Timeout response** – Sent to the user when the code hook does not complete in its configured timeout period.

## Fulfillment

After all the slot values are provided by the user for the intent, Amazon Lex V2 fulfills the user's request. You can configure the following options for fulfillment.

- **Fulfillment code hook** – You can use this option to control the fulfillment Lambda invocation. If the option is disabled, the fulfillment will succeed without invoking the Lambda function.
- **Fulfillment updates** – You can enable fulfillment updates for Lambda functions that take more than a few seconds to complete, so that the user is aware of the fulfillment progress. For more information, see [Configuring fulfillment progress updates \(p. 272\)](#). This functionality is only available for streaming conversations.
- **Fulfillment responses** – You can configure a success response, a failure response, and a timeout response. The appropriate response is returned to the user based on the status of the fulfillment Lambda invocation.

There are three possible fulfillment responses:

- **Success response** – A message sent when the fulfillment Lambda completes successfully.
- **Failure response** – A message sent if the fulfillment failed or Lambda can't be completed for some reason.
- **Timeout response** – A message sent if the fulfillment Lambda function doesn't finish within the configured timeout.

You can set values, configure the next steps and apply conditions corresponding to each response to design the conversation flow. In the absence of a condition or an explicit next step, Amazon Lex V2 moves to closing response.

**Fulfillment advanced options** [Info](#) X

**Fulfillment updates** [Info](#)  Active

You can configure the Lambda function to execute in the background. You can set the messages sent at the start and during fulfillment.

▶ Tell the user fulfillment started  
*Message:* -

▶ Periodically update the user about fulfillment progress  
*Message:* -

**Success response** [Info](#)

The success response is sent to the user when the fulfillment function successfully completes its work.

▶ Tell the user that fulfillment completed successfully  
*Message:* -

▶ Set values Next step in conversation  
*Closing response*

+ Add conditional branching

**Failure response** [Info](#)

The failure response is sent to the user when there is a problem completing fulfillment.

▶ Inform the user that fulfillment didn't complete  
*Message:* -

▶ Set values Next step in conversation  
*End conversation*

+ Add conditional branching

**Timeout response** [Info](#)

The timeout response is sent to the user when the fulfillment function doesn't complete its work in the configured time.

▶ Inform the user that fulfillment reached its timeout before it was complete  
*Message:* -

▶ Set values Next step in conversation  
*End conversation*

+ Add conditional branching

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

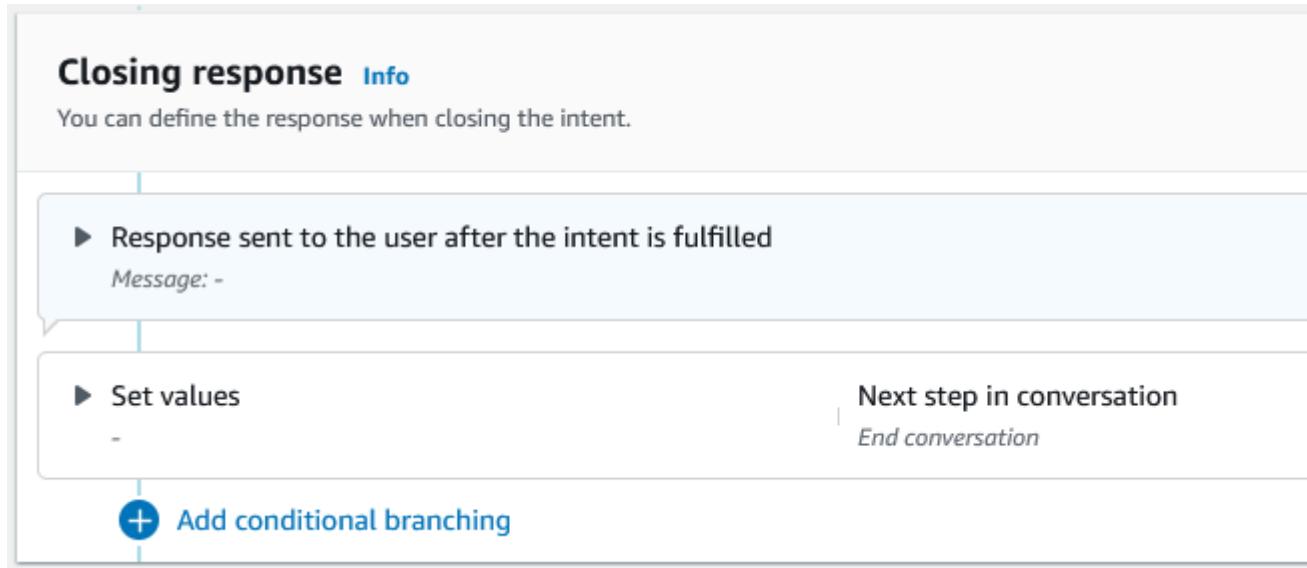
## Closing response

The closing response is sent to your user after their intent is fulfilled. You can use the closing response to end the conversation, or you can use it to let the user know that they can continue with another intent. For example, in a travel booking bot, you can set the closing response for the book hotel room intent to this:

All right, I've booked your hotel room. Is there anything else I can help you with?

You can set values, configure the next steps and apply conditions after the closing response to the design the conversation path. In the absence of a condition or an explicit next step, Amazon Lex V2 ends the conversation.

If you don't supply a closing response, or if none of the conditions evaluates to true, Amazon Lex V2 ends the conversation with your bot.



**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Configuring prompts

You can configure the bot to play messages in a predefined order by checking the box for **Play messages in order**. Otherwise, the bot plays the message and the variations in random order.

Ordered prompts allow the message and variations of a message group to play in order among retries. You can use alternate rephrasing of a message when an invalid response for the prompt is given by the user, or for intent confirmation. Up to two variations of the original message may be set in each slot. You can choose whether to play the messages in order or randomly.

Ordered prompt supports all four types of messages: text, custom payload response, SSML, and card group. Responses are ordered within the same message group. Different message groups are independent.

## Adding slot types

Slot types define the values that users can supply for your intent variables. You define slot types for each language so that the values are specific to that language. For example, for a slot type that lists paint colors, you could include the value "red" in English, "rouge" in French, and "rojo" in Spanish.

This topic describes how to create custom slot types that provide values for your intent's slots. You can also use built-in slot types for standard values. For example, you can use the built-in slot type AMAZON.Country for a list of countries in the world.

### To create a slot type

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the list of bots, choose the bot that you want to add the language to, then choose **Conversation structure** and then **All languages**.
3. Choose the language to add the slot type to, then choose **Slot types**.
4. Choose **Add slot type**, give your slot type a name, and then choose **Add**.
5. In the slot type editor, add the details of your slot type.
  - **Slot value resolution** – Determines how slot values are resolved. If you choose **Expand values**, Amazon Lex V2 uses the values as representative values for training. If you use **Restrict to slot values**, the allowed values for the slot are restricted to the ones that you provide.
  - **Slot type values** – The values for the slot. If you chose **Restrict to slot values**, you can add synonyms for the value. For example, for the value "football" you can add the synonym "soccer." If the user enters "soccer" in a conversation with your bot, the actual value of the slot is "football."
  - **Use slot values as custom vocabulary** – Enable this option to help improve recognition of slot values and synonyms in audio conversations. Don't enable this option when the slot values are common terms, such as "yes," "no," "one," "two," "three," etc.
6. Choose **Save slot type**.

## Testing a bot using the console

The Amazon Lex V2 console contains a test window that you can use to test the interaction with your bot. You use the test window to have a test conversation with your bot and to see the responses that your application receives from the bot.

There are two types of testing that you can perform with your bot. The first, express testing, enables you to test your bot with the exact phrases that you used for creating the bot. For example, if you added the utterance "I want to pick up flowers" to your intent, you can test the bot using that exact phrase.

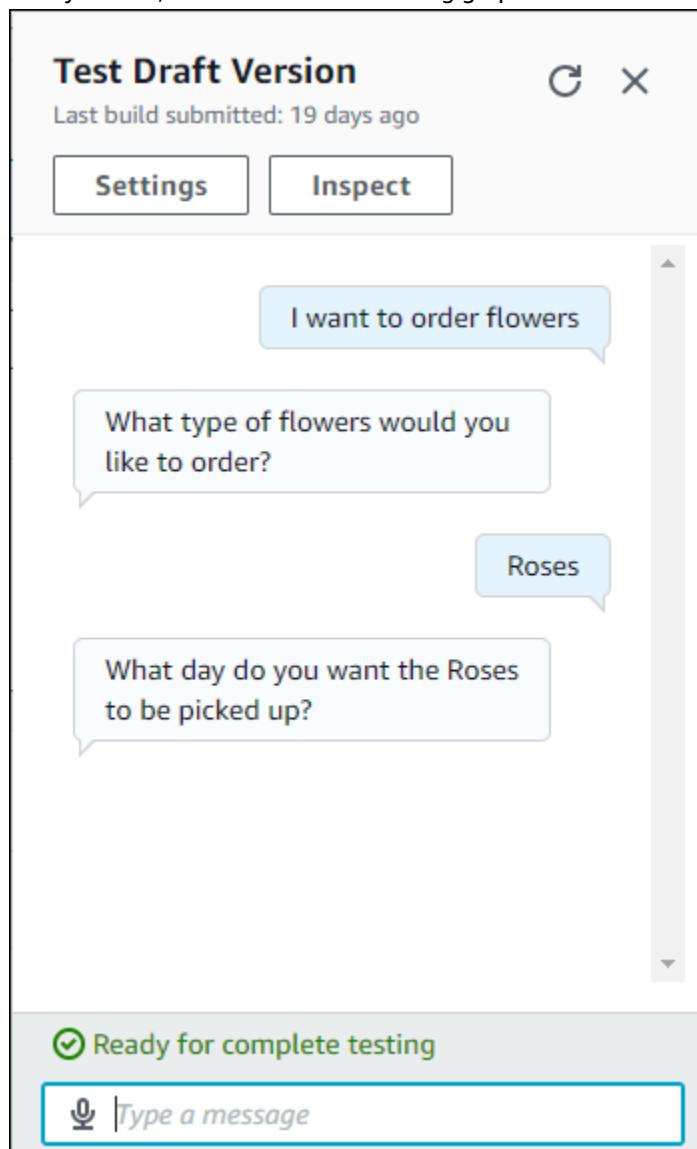
The second type, complete testing, enables you to test your bot using phrases related to the utterances that you configured. For example, you can use the phrase "Can I order flowers" to start a conversation with your bot.

You test a bot using a specific alias and language. If you are testing the development version of the bot, you use the TestBotAlias alias for testing.

### To open the test window

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose the bot to test from the list of bots.
3. From the left menu, choose **Aliases**.
4. From the list of aliases, choose the alias to test.
5. From **Languages**, choose the radio button of the language to test, and then choose **Test**.

After you choose **Test**, the test window opens in the console. You can use the test window to interact with your bot, as shown in the following graphic.



In addition to the conversation, you can also choose **Inspect** in the test window to see the responses returned from the bot. The first view shows you a summary of the information returned from your bot to the test window.

The screenshot shows the 'Test Draft Version' interface for Amazon Lex V2. At the top, there are 'Settings' and 'Inspect' buttons. The 'Inspect' tab is selected, showing a 'Summary' tab and a 'JSON input and output' tab. In the 'Summary' section, under 'Intent', it shows 'OrderFlowers'. Under 'Slots', 'FlowerType' is listed with 'Roses' and 'PickupDate' and 'PickupTime' both listed with a dash. Under 'Active contexts', 'Weather' is listed with '5 turns or 90s'. To the right, a conversation history is shown with messages: 'I want to c...', 'What type of flowers wo...', 'What day do you want th...', 'Ready for complete testin...', and 'Type a message'.

You can also use the test inspection window to see the JSON structures that are sent between the bot and the test window. You can see both the request from the test window and the response from Amazon Lex V2.

The screenshot shows the 'Inspect' interface for an Amazon Lex V2 bot. On the left, there's a 'Request' section containing a JSON object:

```
{  
  "botAliasId": "TSTALIASID",  
  "botId": "Q2NA3VH5E3",  
  "localeId": "en_US",  
  "text": "I want to order flowers"  
  "sessionId": "130772450386735"  
}
```

On the right, there's a 'Test Draft Version' panel with a message history:

- I want to order flowers
- What type of flowers would you like to order?
- What day do you want the flowers to be picked up?

At the bottom right of the test panel, there's a green checkmark icon followed by the text 'Ready for complete testing'.

## Creating versions

Amazon Lex V2 supports publishing versions of bots so that you can control the implementation that your client applications use. A *version* is a numbered snapshot of your work that you can publish for use in different parts of your workflow, such as development, beta deployment, and production.

### The Draft version

When you create an Amazon Lex V2 bot there is only one version, the Draft version.

Draft is the working copy of your bot. You can update only the Draft version and until you publish your first version, Draft is the only version of the bot that you have.

The Draft version of your bot is associated with the TestBotAlias. The TestBotAlias should only be used for manual testing. Amazon Lex V2 limits the number of runtime requests that you can make to the TestBotAlias alias of the bot.

## Creating a version

When you version an Amazon Lex V2 bot you create a numbered snapshot of the bot so that you can use the bot as it existed when the version was made. Once you've created a numeric version it will stay the same while you continue to work on the draft version of your application.

When you create a version, you can choose the locales to include in the version. You don't need to choose all of the locales in a bot. Also, when you create a version you can choose a locale from a previous version. For example, if you have three versions of a bot, you can choose one locale from the Draft version and one from version two when you create version four.

If you delete a locale from the Draft version, it is not deleted from a numbered version.

If a bot version is not used for six months, Amazon Lex V2 will mark the version inactive. When a version is inactive, you can't use runtime operations with the bot. To make the bot active, rebuild all the languages associated with the version.

## Updating an Amazon Lex V2 bot

You can update only the Draft version of an Amazon Lex V2 bot. Versions can't be changed. You can publish a new version any time after you update a resource in the console or with the [CreateBotVersion](#) operation.

## Deleting an Amazon Lex V2 bot or version

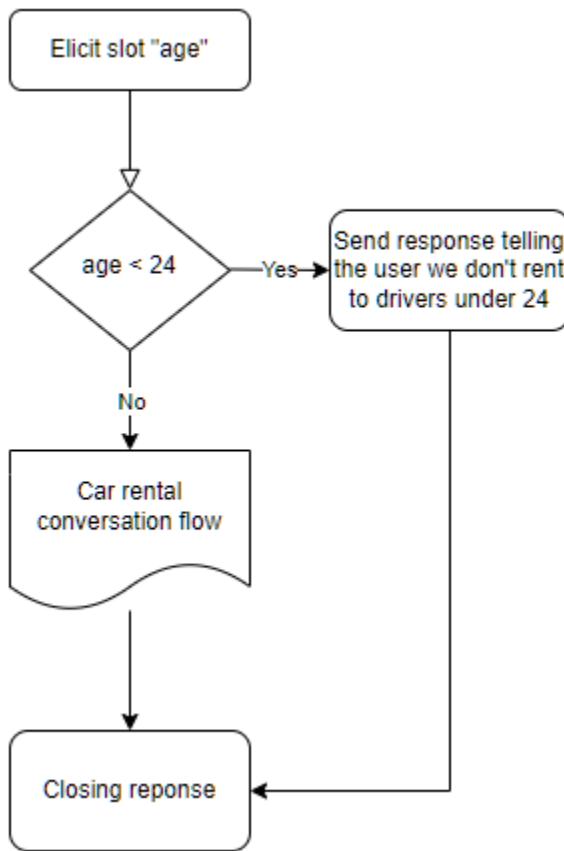
Amazon Lex V2 supports deleting a bot or version using the console or one of the API operations:

- [DeleteBot](#)
- [DeleteBotVersion](#)

## Creating conversation paths

Typically, Amazon Lex V2 manages the flow of conversations with your users. For simple bots, the default flow can be enough to create a good experience for your users. However, for more complex bots, you might want to take control of the conversation and direct the flow into more complex paths.

For example, in a bot that books car rentals, you might not rent to younger drivers. In this case, you can create a condition that checks to see if a driver is below a certain age, and if so, jump to the closing response.



To design such interactions, you can configure the next step at each point in the conversation, evaluate conditions, set values and invoke code hooks.

Conditional branching helps you create paths for your users through complex interactions. You can use a conditional branch at any point that you pass control of the conversation to your bot. For example, you can create a condition before the bot elicits the first slot value, you can create a condition between eliciting each slot value, or you can create a condition before the bot closes the conversation. For a list of the places that you can add conditions, see [Adding intents \(p. 32\)](#).

When you create a bot, Amazon Lex V2 creates a default path through the conversation based on the priority order of the slots. To customize the conversation path, you can modify the next step at any point in the conversation. For more information, see [Configure next steps in the conversation \(p. 53\)](#).

To create alternative paths based on conditions, you can use a conditional branch at any point in the conversation. For example, you can create a condition before the bot elicits the first slot value. You can create a condition between eliciting each slot value, or you can create a condition before the bot closes the conversation. For a list of the places allowing you to add conditions, see [Add conditions to branch conversations \(p. 54\)](#).

You can set conditions based on slot values, session attributes, the input mode and input transcript, or a response from Amazon Kendra.

You can set slot and session attribute values at each point in the conversation. For more information, see [Set values during the conversation \(p. 54\)](#).

You can also set the next action to dialog code hook to run a Lambda function. For more information, see [Invoke dialog code hook \(p. 58\)](#).

The following image shows the creation of a path for a slot in the console. In this example, Amazon Lex V2 will elicit the slot "age". If the value of the slot is less than 24, Amazon Lex V2 jumps to the closing response, otherwise Amazon Lex will follow the default path.

## Conditional branching Info

Jump to different parts of the conversation based on conditions you define. You can add up to 4 conditional branches.

Branch1 X + Add branch ... if no matches ... Default flow

Condition for Branch1

If {age} < 24

Condition

If {age} < 24

Response

Message: I'm sorry, we don't rent to drivers under the age of 24.

Message

I'm sorry, we don't rent to drivers under the age of 24.

Variations - optional

Advanced options

Add custom payloads, SSML, and card groups.

Set values

-

Slot values - optional

Add slot values as: {slot} = "value"

{intent.slot} = "value"

Separate values with a new line.

Session attributes - optional

Add session attributes as: [session attribute] = "value"

[session attribute] = "value"

Separate values with a new line.

Next step in conversation

Closing response

Next step in conversation

Closing response

52

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Configure next steps in the conversation

You can configure a next step at each stage of the conversation to design conversations. Typically, Amazon Lex V2 automatically configures the default next steps for each stage of the conversation as per the following order.

Initial Response → Slot Elicitation → Confirmation (if active) → Fulfillment (if active) → Closing Response (if active) → End conversation

You can modify the default next steps and design the conversation based on the expected user experience. The following next steps can be configured at each stage of the conversation:

**Jump to**

- **Initial response** – The conversation is restarted from the beginning of the intent. You can choose to skip the initial response while configuring this next step
- **Elicit a slot** – You can elicit any slot in the intent.
- **Evaluate conditions** – You can evaluate conditions and branch conversation at any step of the conversation.
- **Invoke dialog code hook** – You can invoke business logic at any step.
- **Confirm intent** – The user will be prompted to confirm the intent.
- **Fulfill intent** – The fulfillment of the intent will begin as a next step.
- **Closing response** – The closing response will be returned to the user.

**Switch to**

- **Intent** – You can transition to a different intent and continue the conversation for this intent. You can optionally skip the initial response of the intent while making the transition.
- **Intent: specific slot** – You can directly elicit a specific slot in a different intent if you have already captured some slot values in the current intent.

**Wait for user input** – The bot waits for the user to provide inputs for recognizing any new intent. You can configure prompts such as “Is there anything else I can help you with?” prior to setting this next step. The bot will be in `ElicitIntent` dialog state.

**End conversation** – The conversation with the bot is closed.

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Set values during the conversation

Amazon Lex V2 provides the ability to set slot values and session attribute values at every step of the conversation. You can then use these values during the conversation to evaluate conditions or use them during intent fulfillment.

You can set slot values for the current intent. If the next step in the conversation is to invoke another intent, you can set slot values of the new intent.

Use the following syntax when using slot values and session attributes:

- **Slot values** – surround the slot name with braces ("{}"). For slot values in the current intent, you only need to use the slot name. For example, {slot}. If you are setting a value in the next intent, you must use both the intent name and the slot name to identify the slot. For example, {intent.slot}.

```
{PhoneNumber} = "1234567890" {CheckBalance.AccountNumber} =  
"99999999" {BookingID} = "ABC123" {FirstName} = "John"
```

- **Session attributes** – surround the attribute name with square brackets ("[]"). For example, [sessionAttribute].

```
[username] = "john.doe"
```

### Note

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

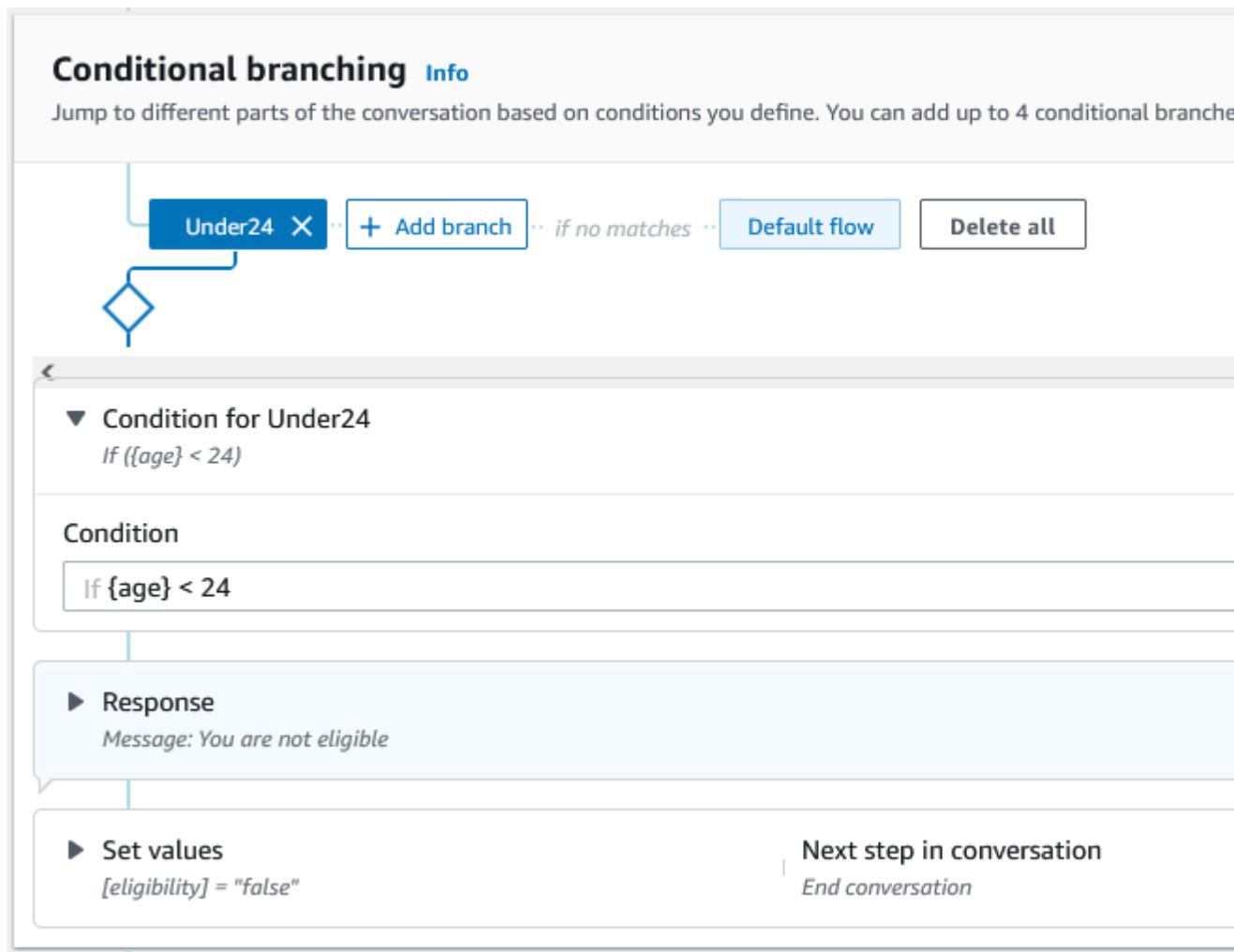
## Add conditions to branch conversations

You can use *conditional branching* to control the path that your customer takes through the conversation with your bot. You can branch the conversation based on slot values, session attributes, the contents of the input mode and input transcript fields, or a response from Amazon Kendra.

You can define up to four branches. Each branch has a condition that must be satisfied in order for Amazon Lex V2 to follow that branch. If none of the branches has its condition satisfied, a default branch is followed.

When you define a branch, you define the action that Amazon Lex V2 should take if the conditions corresponding to that branch evaluate to true. You can define any of the following actions:

- A response sent to the user.
- Slot values to apply to slots.
- Session attribute values for the current session.
- The next step in the conversation. For more information, see [Creating conversation paths \(p. 49\)](#).



Each conditional branch has a Boolean expression that must be satisfied for Amazon Lex V2 to follow the branch. There are comparison and Boolean operators, functions, and quantifier operators that you can use for your conditions. For example, the following condition returns true if the `{age}` slot is less than 24.

`{age} < 24`

The following condition returns true if the `{toppings}` multi-value slot contains the word "pineapple".

`{toppings} CONTAINS "pineapple"`

You can combine multiple comparison operators with a Boolean operator for more complex conditions. For example, the following condition returns true if the `{make}` slot value is "Honda" and the `{model}` slot value is "Civic". Use parentheses to set the evaluation order.

`({make} = "Honda") AND ({model} = "Civic")`

The following topics provide details on the conditional branch operators and functions.

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user

takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

#### Topics

- [Comparison operators \(p. 56\)](#)
- [Boolean operators \(p. 56\)](#)
- [Quantifier operators \(p. 57\)](#)
- [Functions \(p. 57\)](#)
- [Sample conditional expressions \(p. 57\)](#)

## Comparison operators

Amazon Lex V2 supports the following comparison operators for conditions:

- Equals (=)
- Not equals (!=)
- Less than (<)
- Less than or equals (<=)
- Greater than (>)
- Greater than or equals (>=)

When using a comparison operator, it uses the following rules.

- The left-hand side must be a reference. For example, to reference a slot value, you use `{slotName}`. To reference a session attribute value, you use `[attribute]`. For input mode and input transcript, you use `$.inputMode` and `$inputTranscript`.
- The right-hand side must be a constant and the same type as the left hand side.
- Any expression referencing an attribute which has not been set is treated as invalid, and is not evaluated.
- When you compare a multi-valued slot, the value used is a comma-separated list of all interpreted values.

Comparisons are based on the slot type of the reference. They are resolved as follows:

- **Strings** – strings are compared based on their ASCII representation. The comparison is case-insensitive.
- **Numbers** – number-based slots are converted from the string representation to a number and then compared.
- **Date/Time** – time-based slots are compared based on the time series. The earlier date or time is considered smaller. For durations, shorter periods are considered smaller.

## Boolean operators

Amazon Lex V2 supports Boolean operators to combine comparison operators. They let you create statements similar to the following:

```
({number} >= 5) AND ({number} <= 10)
```

You can use the following Boolean operators:

- AND (&&)
- OR (||)
- NOT (!)

## Quantifier operators

Quantifier operators evaluate the elements of a sequence and determine if one or more elements satisfy the condition.

- **CONTAINS** – determines if the specified value is contained in a multi-valued slot and returns true if it is. For example, {toppings} CONTAINS "pineapple" returns true if the user ordered pineapple on their pizza.

## Functions

Functions must be prefixed with the string fn.. The argument to the function is a reference to a slot. Amazon Lex V2 provides two functions for getting information from the values of slots.

- **fn.COUNT()** – counts the number of values in a multi-valued slot. For example, if the slot {toppings} contains the value "pepperoni, pineapple", fn.COUNT({toppings}) equals 2.
- **fn.IS\_SET()** – returns true if a slot is set in the current session. Continuing the last example, fn.IS\_SET({toppings}) is true.

## Sample conditional expressions

Here are some sample conditional expressions.

<b>Value type</b>	<b>Use case</b>	<b>Conditional expression</b>
Custom slot	pizzaSize slot value is equal to large	{pizzaSize} = "large"
Custom slot	pizzaSize is equal to large or medium	{pizzaSize} = "large" OR {pizzaSize} = "medium"
Custom slot	Expressions with ( ) and AND/OR	{pizzaType} = "pepperoni" OR {pizzaSize} = "medium" OR {pizzaSize} = "small"
Custom slot (Multi-Valued Slot)	Check if one of the topping is Onion	{toppings} CONTAINS "Onion"
Custom slot (Multi-Valued Slot)	Number of toppings are more than 3	fn.COUNT({topping}) > 2
AMAZON.AlphaNumeric	bookingID is ABC123	{bookingID} = "ABC123"
AMAZON.Number	age slot value is greater than 30	{age} > 30
AMAZON.Number	age slot value is equal to 10	{age} = 10
AMAZON.Date	dateOfBirth slot value before 1990	{dateOfBirth} < "1990-10-01"

Value type	Use case	Conditional expression
AMAZON.State	destinationState slot value is equal to Washington	{destinationState} = "washington"
AMAZON.Country	destinationCountry slot value is not United States	{destinationCountry} != "united states"
AMAZON.FirstName	firstName slot value is John	{firstName} = "John"
AMAZON.PhoneNumber	phoneNumber slot value is 716767891932	{phoneNumer} = 716767891932
AMAZON.Percentage	Check if percentage slot value is greater than or equals 78	{percentage} >= 78
AMAZON.EmailAddress	emailAddress slot value is userA@hmail.com	{emailAddress} = "userA@hmail.com"
AMAZON.LastName	lastName slot value is Doe	{lastName} = "Doe"
AMAZON.City	City slot value is equal to Seattle	{city} = "Seattle"
AMAZON.Time	Time is after 8 PM	{time} > "20:00"
AMAZON.StreetName	streetName slot value is Boren Avenue	{streetName} = "boren avenue"
AMAZON.Duration	travelDuration slot value is less than 2 hours	{travelDuration} < P2H
Input mode	Input mode is speech	\$.inputMode = "Speech"
Input transcript	Input transcript is equal to "I want a large pizza"	\$.inputTranscript == "I want a large pizza"
Session attribute	check retry_enabled flag	((retry_enabled)) == true
Kendra response	Kendra response contains FAQ	fn.IS_SET(x-amz-lex:kendra-search-response-question_answer-question-1)

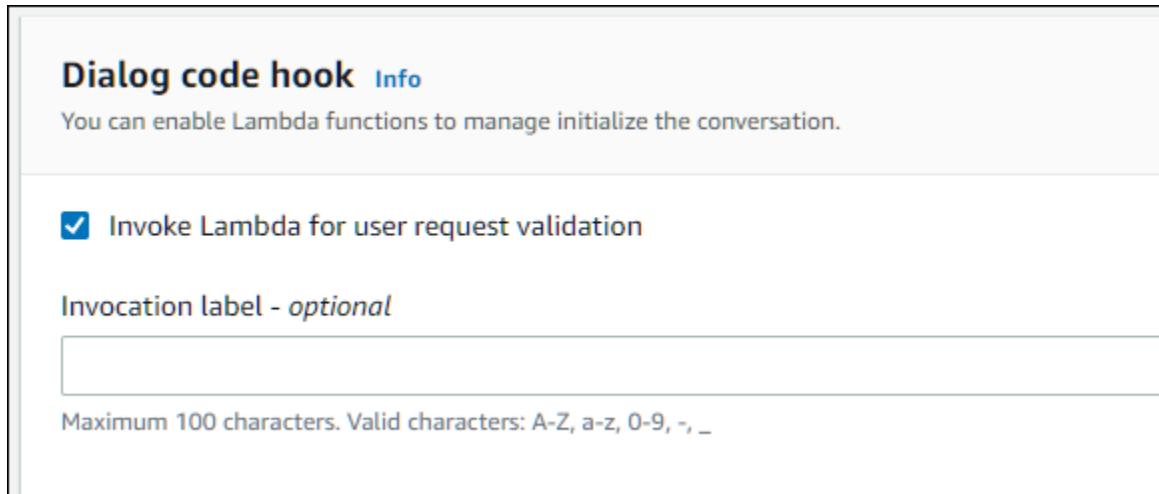
## Invoke dialog code hook

At each step in the conversation when Amazon Lex sends a message to the user, you can use a Lambda function as the next step in the conversation. You can use the function to implement business logic based on current state of the conversation.

The Lambda function that runs is associated with the bot alias that you are using. To invoke Lambda function across all dialog code hooks in your intent, you must select **Use a Lambda function for initializing and validation** for the intent. For more information on choosing a Lambda function, see [Attaching a Lambda function to a bot alias \(p. 195\)](#).

There are two steps to using a Lambda function. First, you must activate the dialog code hook at any point in the conversation. Second, you must set the next step in the conversation to use the dialog code hook.

The following image shows the dialog code hook activated.



Next, set the code hook as the next action for the conversation step. You can do this by configuring the next step in conversation to Invoke dialog code hook. The following image shows a conditional branch where invoking the dialog code hook is the next step for the default path of the conversation.

## Conditional branching Info

Jump to different parts of the conversation based on conditions you define. You can add multiple branches to your dialog.

```
graph LR; Start(( )) --> Decision{ }; Decision --> Branch1[Branch1 X]; Decision --> DefaultFlow[Default flow]; Branch1 --> Response[Response]; Branch1 --> SetValues[Set values]; SetValues --> Placeholder["{{slot}} = \"value\""]; DefaultFlow --> Decision;
```

**Response**  
*Message:* -

**Set values**  
-

**Slot values - optional**  
Add slot values as: {slot} = "value"

`{slot} = "value"`

Separate values with a new line.

**Next step in conversation**

**Invoke dialog code hook**

**Session attributes**  
Add session attributes

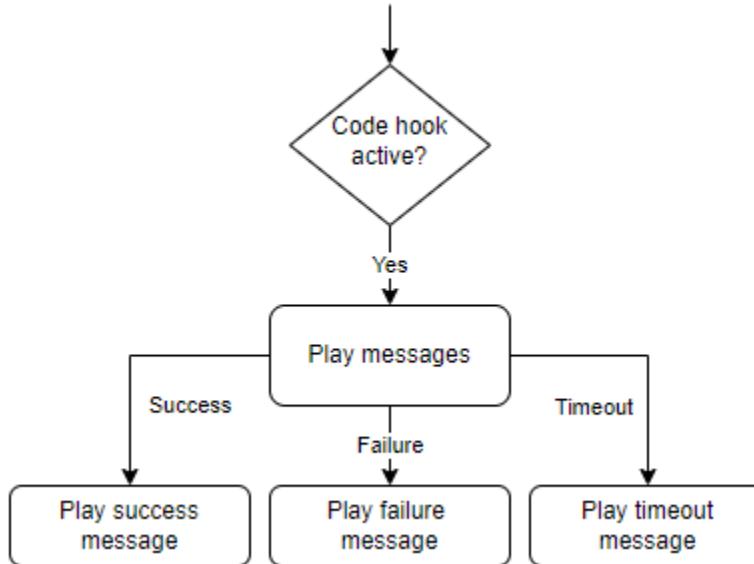
`[session attribute]`

Separate values with a new line.

When code hooks are active, you can set three responses to return to the user:

- **Success** – Sent when the Lambda function completed successfully.

- **Failure** – Sent if there was a problem with running the Lambda function, or the Lambda function returned an `intent.state` value of Failed.
- **Timeout** – Sent if the Lambda function did not complete in its configured timeout period.



Choose **Lambda dialog code hook** and then choose **Advanced options** to see the three options for responses that correspond to the Lambda function invocation. You can set values, configure the next steps and apply conditions corresponding to each response to design the conversation flow. In the absence of a condition or an explicit next step, Amazon Lex V2 decides the next step based on the current state of the conversation.

On the **Advanced options** page you can also choose to enable or disable your Lambda function invocation. When the function is enabled, the dialog code hook is invoked with Lambda invocation, followed by the success, failure or timeout message based on Lambda invocation results. When the function is disabled, Amazon Lex V2 doesn't run the Lambda function and proceeds as if the dialog code hook is successful.

You can also set an invocation label that is sent to the Lambda function when it is invoked by this message. You can use this to help identify the section of your Lambda function to run.

**Note**

On August 17, 2022, Amazon Lex V2 released a change to the way conversations are managed with the user. This change gives you more control over the path that the user takes through the conversation. For more information, see [Understanding conversation flow management \(p. 18\)](#). Bots created before August 17, 2022 do not support dialog code hook messages, setting values, configuring next steps, and adding conditions.

## Built-in intents and slot types

To make it easier to create bots, Amazon Lex V2 allows you to use standard built-in intents and slot types.

**Topics**

- [Built-in intents \(p. 62\)](#)
- [Built-in slot types \(p. 75\)](#)

## Built-in intents

For common actions, you can use the standard built-in intents library. To create an intent from a built-in intent, choose a built-intent in the console, and give it a new name. The new intent has the configuration of the base intent, such as the sample utterances.

In the current implementation, you can't do the following:

- Add or remove sample utterances from the base intent
- Configure slots for built-in intents

### To add a built-in intent to a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose the bot to add the built-in intent to.
3. In the left menu, choose the language and then choose **Intents**.
4. Choose **Add intent**, and then choose **Use built-in intent**.
5. In **Built-in intent**, choose the intent to use.
6. Give the intent a name, and then choose **Add**.
7. Use the intent editor to configure the intent as required for your bot.

#### Topics

- [AMAZON.CancelIntent \(p. 62\)](#)
- [AMAZON.FallbackIntent \(p. 62\)](#)
- [AMAZON.HelpIntent \(p. 64\)](#)
- [AMAZON.KendraSearchIntent \(p. 64\)](#)
- [AMAZON.PauseIntent \(p. 74\)](#)
- [AMAZON.RepeatIntent \(p. 74\)](#)
- [AMAZON.ResumeIntent \(p. 74\)](#)
- [AMAZON.StartOverIntent \(p. 74\)](#)
- [AMAZON.StopIntent \(p. 75\)](#)

## AMAZON.CancelIntent

Responds to words and phrases that indicate the user wants to cancel the current interaction. Your application can use this intent to remove slot type values and other attributes before ending the interaction with the user.

Common utterances:

- cancel
- never mind
- forget it

## AMAZON.FallbackIntent

When a user's input to an intent isn't what a bot expects, you can configure Amazon Lex V2 to invoke a *fallback intent*. For example, if the user input "I'd like to order candy" doesn't match an intent in your OrderFlowers bot, Amazon Lex V2 invokes the fallback intent to handle the response.

The built-in AMAZON.FallbackIntent intent type is added to your bot automatically when you create a bot using the console, when you use the API you can specify the intent using the [CreateBot](#) operation.

Invoking a fallback intent uses two steps. In the first step the fallback intent is matched based on the input from the user. When the fallback intent is matched, the way the bot behaves depends on the number of retries configured for a prompt.

Amazon Lex V2 matches the fallback intent in these situations:

- The user's input to an intent doesn't match the input that the bot expects
- Audio input is noise, or text input isn't recognized as words.
- The user's input is ambiguous and Amazon Lex V2 can't determine which intent to invoke.

The fallback intent is invoked when:

- An intent doesn't recognize the user input as a slot value after the configured number of tries.
- An intent doesn't recognize the user input as a response to a confirmation prompt after the configured number of tries.

You can't add the following to a fallback intent:

- Utterances
- Slots
- A confirmation prompt

## Using a Lambda Function with a Fallback Intent

When a fallback intent is invoked, the response depends on the setting of the fulfillmentCodeHook parameter to the [CreateIntent](#) operation. The bot does one of the following:

- Returns the intent information to the client application.
- Calls the aliases's validation and fulfillment Lambda function. It calls the function with the session variables that are set for the session.

For more information about setting the response when a fallback intent is invoked, see the fulfillmentCodeHook parameter of the [CreateIntent](#) operation.

If you use the Lambda function with your fallback intent, you can use this function to call another intent or to perform some form of communication with the user, such as collecting a callback number or opening a session with a customer service representative.

A fallback intent can be invoked multiple times in the same session. For example, suppose that your Lambda function uses the ElicitIntent dialog action to prompt the user for a different intent. If Amazon Lex V2 can't infer the user's intent after the configured number of tries, it invokes the fallback intent again. It also invokes the fallback intent when the user doesn't respond with a valid slot value after the configured number of tries.

You can configure your Lambda function to keep track of the number of times that the fallback intent is called using a session variable. Your Lambda function can take a different action if it is called more times than the threshold that you set in your Lambda function. For more information about session variables, see [Setting session attributes \(p. 230\)](#).

## AMAZON.HelpIntent

Responds to words or phrases that indicate the user needs help while interacting with your bot. When this intent is invoked, you can configure your Lambda function or application to provide information about the your bot's capabilities, ask follow up questions about areas of help, or hand the interaction over to a human agent.

Common utterances:

- help
- help me
- can you help me

## AMAZON.KendraSearchIntent

To search documents that you have indexed with Amazon Kendra, use the AMAZON.KendraSearchIntent intent. When Amazon Lex V2 can't determine the next action in a conversation with the user, it triggers the search intent.

The AMAZON.KendraSearchIntent is available only in the English (US) (en-US) locale and in the US East (N. Virginia), US West (Oregon) and Europe (Ireland) Regions.

Amazon Kendra is a machine-learning-based search service that indexes natural language documents such as PDF documents or Microsoft Word files. It can search indexed documents and return the following types of responses to a question:

- An answer
- An entry from a FAQ that might answer the question
- A document that is related to the question

For an example of using the AMAZON.KendraSearchIntent, see [Example: Creating a FAQ Bot for an Amazon Kendra Index \(p. 70\)](#).

If you configure an AMAZON.KendraSearchIntent intent for your bot, Amazon Lex V2 calls the intent whenever it can't determine the user utterance for an intent. If there is no response from Amazon Kendra, the conversation continues as configured in the bot.

### Note

Amazon Lex V2 currently does not support the AMAZON.KendraSearchIntent during slot elicitation. If Amazon Lex V2 can't determine the user utterance for a slot, it calls the AMAZON.FallbackIntent.

When you use the AMAZON.KendraSearchIntent with the AMAZON.FallbackIntent in the same bot, Amazon Lex V2 uses the intents as follows:

1. Amazon Lex V2 calls the AMAZON.KendraSearchIntent. The intent calls the Amazon Kendra Query operation.
2. If Amazon Kendra returns a response, Amazon Lex V2 displays the result to the user.
3. If there is no response from Amazon Kendra, Amazon Lex V2 re-prompts the user. The next action depends on response from the user.
  - If the response from the user contains an utterance that Amazon Lex V2 recognizes, such as filling a slot value or confirming an intent, the conversation with the user proceeds as configured for the bot.
  - If the response from the user does not contain an utterance that Amazon Lex V2 recognizes, Amazon Lex V2 makes another call to the Query operation.

4. If there is no response after the configured number of retries, Amazon Lex V2 calls the AMAZON.FallbackIntent and ends the conversation with the user.

There are three ways to use the AMAZON.KendraSearchIntent to make a request to Amazon Kendra:

- Let the search intent make the request for you. Amazon Lex V2 calls Amazon Kendra with the user's utterance as the search string. When you create the intent, you can define a query filter string that limits the number of responses that Amazon Kendra returns. Amazon Lex V2 uses the filter in the query request.
- Add additional query parameters to the request to narrow the search results using your Lambda function. You add a `kendraQueryFilterString` field that contains Amazon Kendra query parameters to the delegate dialog action. When you add query parameters to the request with the Lambda function, they take precedence over the query filter that you defined when you created the intent.
- Create a new query using the Lambda function. You can create a complete Amazon Kendra query request that Amazon Lex V2 sends. You specify the query in the `kendraQueryRequestPayload` field in the delegate dialog action. The `kendraQueryRequestPayload` field takes precedence over the `kendraQueryFilterString` field.

To specify the `queryFilterString` parameter when you create a bot, or to specify the `kendraQueryFilterString` field when you call the delegate action in a dialog Lambda function, you specify a string that is used as the attribute filter for the Amazon Kendra query. If the string isn't a valid attribute filter, you'll get an `InvalidBotConfigException` exception at runtime. For more information about attribute filters, see [Using document attributes to filter queries in the Amazon Kendra Developer Guide](#).

To have control over the query that Amazon Lex V2 sends to Amazon Kendra, you can specify a query in the `kendraQueryRequestPayload` field in your Lambda function. If the query isn't valid, Amazon Lex V2 returns an `InvalidLambdaResponseException` exception. For more information, see the [Query operation](#) in the *Amazon Kendra Developer Guide*.

For an example of how to use the AMAZON.KendraSearchIntent, see [Example: Creating a FAQ Bot for an Amazon Kendra Index \(p. 70\)](#).

## IAM Policy for Amazon Kendra Search

To use the AMAZON.KendraSearchIntent intent, you must use a role that provides AWS Identity and Access Management (IAM) policies that enable Amazon Lex V2 to assume a runtime role that has permission to call the Amazon Kendra Query intent. The IAM settings that you use depend on whether you create the AMAZON.KendraSearchIntent using the Amazon Lex V2 console, or using an AWS SDK or the AWS Command Line Interface (AWS CLI). When you use the console, you can choose between adding permission to call Amazon Kendra to the Amazon Lex V2 service-linked role or using a role specifically for calling the Amazon Kendra Query operation. When you use the AWS CLI or an SDK to create the intent, you must use a role specifically for calling the Query operation.

### Attaching Permissions

You can use the console to attach permissions to access the Amazon Kendra Query operation to the default Amazon Lex V2 service-linked role. When you attach permissions to the service-linked role, you don't have to create and manage a runtime role specifically to connect to the Amazon Kendra index.

The user, role, or group that you use to access the Amazon Lex V2 console must have permissions to manage role policies. Attach the following IAM policy to the console access role. When you grant these permissions, the role has permissions to change the existing service-linked role policy.

```

    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:AttachRolePolicy",
                "iam:PutRolePolicy",
                "iam:GetRolePolicy"
            ],
            "Resource": "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/AWSServiceRoleForLexBots*"
        },
        {
            "Effect": "Allow",
            "Action": "iam>ListRoles",
            "Resource": "*"
        }
    ]
}

```

## Specifying a Role

You can use the console, the AWS CLI, or the API to specify a runtime role to use when calling the Amazon Kendra Query operation.

The IAM user, role, or group that you use to specify the runtime role must have the `iam:PassRole` permission. The following policy defines the permission. You can use the `iam:AssociatedResourceArn` and `iam:PassedToService` condition context keys to further limit the scope of the permissions. For more information, see [IAM and AWS STS Condition Context Keys](#) in the [AWS Identity and Access Management User Guide](#).

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::account:role/role"
        }
    ]
}

```

The runtime role that Amazon Lex V2 needs to use to call Amazon Kendra must have the `kendra:Query` permissions. When you use an existing IAM role for permission to call the Amazon Kendra Query operation, the role must have the following policy attached.

You can use the IAM console, the IAM API, or the AWS CLI to create a policy and attach it to a role. These instructions use the AWS CLI to create the role and policies.

### Note

The following code is formatted for Linux and MacOS. For Windows, replace the Linux line continuation character (\) with a caret (^).

## To add Query operation permission to a role

1. Create a document called **KendraQueryPolicy.json** in the current directory, add the following code to it, and save it

```

{
    "Version": "2012-10-17",
    "Statement": [

```

```
{  
    "Effect": "Allow",  
    "Action": [  
        "kendra:Query"  
    ],  
    "Resource": [  
        "arn:aws:kendra:region:account:index/index ID"  
    ]  
}  
}
```

2. In the AWS CLI, run the following command to create the IAM policy for running the Amazon Kendra Query operation.

```
aws iam create-policy \  
    --policy-name query-policy-name \  
    --policy-document file://KendraQueryPolicy.json
```

3. Attach the policy to the IAM role that you are using to call the Query operation.

```
aws iam attach-role-policy \  
    --policy-arn arn:aws:iam::account-id:policy/query-policy-name \  
    --role-name role-name
```

You can choose to update the Amazon Lex V2 service-linked role or to use a role that you created when you create the AMAZON.KendraSearchIntent for your bot. The following procedure shows how to choose the IAM role to use.

#### To specify the runtime role for AMAZON.KendraSearchIntent

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose the bot that you want to add the AMAZON.KendraSearchIntent to.
3. Choose the plus (+) next to **Intents**.
4. In **Add intent**, choose **Search existing intents**.
5. In **Search intents**, enter **AMAZON.KendraSearchIntent** and then choose **Add**.
6. In **Copy built-in intent**, enter a name for the intent, such as **KendraSearchIntent**, and then choose **Add**.
7. Open the **Amazon Kendra query** section.
8. For **IAM role** choose one of the following options:
  - To update the Amazon Lex V2 service-linked role to enable your bot to query Amazon Kendra indexes, choose **Add Amazon Kendra permissions**.
  - To use a role that has permission to call the Amazon Kendra Query operation, choose **Use an existing role**.

## Using Request and Session Attributes as Filters

To filter the response from Amazon Kendra to items related to current conversation, use session and request attributes as filters by adding the `queryFilterString` parameter when you create your bot. You specify a placeholder for the attribute when you create the intent, and then Amazon Lex V2 substitutes a value before it calls Amazon Kendra. For more information about request attributes, see [Setting request attributes \(p. 231\)](#). For more information about session attributes, see [Setting session attributes \(p. 230\)](#).

The following is an example of a `queryFilterString` parameter that uses a string to filter the Amazon Kendra query.

```
{"equalsTo": {"key": "City", "value": {"stringValue": "Seattle"}}}
```

The following is an example of a `queryFilterString` parameter that uses a session attribute called "SourceURI" to filter the Amazon Kendra query.

```
{"equalsTo": {"key": "SourceURI", "value": {"stringValue": "[FileURL]}}}}
```

The following is an example of a `queryFilterString` parameter that uses a request attribute called "DepartmentName" to filter the Amazon Kendra query.

```
{"equalsTo": {"key": "Department", "value": {"stringValue": "((DepartmentName))}}}}
```

The `AMAZON.KendraSearchIntent` filters use the same format as the Amazon Kendra search filters. For more information, see [Using document attributes to filter search results](#) in the *Amazon Kendra developer guide*.

The query filter string used with the `AMAZON.KendraSearchIntent` must use lower-case letters for the first letter of each filter. For example, the following is a valid query filter for the `AMAZON.KendraSearchIntent`.

```
{
    "andAllFilters": [
        {
            "equalsTo": {
                "key": "City",
                "value": {
                    "stringValue": "Seattle"
                }
            }
        },
        {
            "equalsTo": {
                "key": "State",
                "value": {
                    "stringValue": "Washington"
                }
            }
        }
    ]
}
```

## Using the Search Response

Amazon Kendra returns the response to a search in a response from the intent's `IntentClosingSetting` statement. The intent must have a `closingResponse` statement unless a Lambda function produces a closing response message.

Amazon Kendra has five types of responses.

- The following two responses require an FAQ to be set up for your Amazon Kendra index. For more details, see [Adding questions and answers directly to a index](#).
  - `x-amz-lex:kendra-search-response-question_answer-question-<N>` – The question from a FAQ that matches the search.

- `x-amz-lex:kendra-search-response-question_answer-<N>` – The answer from a FAQ that matches the search.
- The following three responses require a data source to be set up for your Amazon Kendra index. For more details, see [Creating a data source](#).
  - `x-amz-lex:kendra-search-response-document-<N>` – An excerpt from a document in the index that is related to the text of the utterance.
  - `x-amz-lex:kendra-search-response-document-link-<N>` – The URL of a document in the index that is related to the text of the utterance.
  - `x-amz-lex:kendra-search-response-answer-<N>` – An excerpt from a document in the index that answers the question.

The responses are returned in `request` attributes. There can be up to five responses for each attribute, numbered 1 through 5. For more information about responses, see [Types of response](#) in the *Amazon Kendra Developer Guide*.

The `closingResponse` statement must have one or more message groups. Each message group contains one or more messages. Each message can contain one or more placeholder variables that are replaced by request attributes in the response from Amazon Kendra. There must be at least one message in the message group where all of the variables in the message are replaced by request attribute values in the runtime response, or there must be a message in the group with no placeholder variables. The request attributes are set off with double parentheses ("((" "))). The following message group messages match any response from Amazon Kendra:

- "I found a FAQ question for you: ((x-amz-lex:kendra-search-response-question\_answer-question-1)), and the answer is ((x-amz-lex:kendra-search-response-question\_answer-answer-1))"
- "I found an excerpt from a helpful document: ((x-amz-lex:kendra-search-response-document-1))"
- "I think the answer to your questions is ((x-amz-lex:kendra-search-response-answer-1))"

## Using a Lambda Function to Manage the Request and Response

The `AMAZON.KendraSearchIntent` intent can use your dialog code hook and fulfillment code hook to manage the request to Amazon Kendra and the response. Use the dialog code hook Lambda function when you want to modify the query that you send to Amazon Kendra, and the fulfillment code hook Lambda function when you want to modify the response.

### Creating a Query with the Dialog Code Hook

You can use the dialog code hook to create a query to send to Amazon Kendra. Using the dialog code hook is optional. If you don't specify a dialog code hook, Amazon Lex V2 constructs a query from the user utterance and uses the `queryFilterString` that you provided when you configured the intent, if you provided one.

You can use two fields in the dialog code hook response to modify the request to Amazon Kendra:

- `kendraQueryFilterString` – Use this string to specify attribute filters for the Amazon Kendra request. You can filter the query using any of the index fields defined in your index. For the structure of the filter string, see [Using document attributes to filter queries](#) in the *Amazon Kendra Developer Guide*. If the specified filter string isn't valid, you will get an `InvalidLambdaResponseException` exception. The `kendraQueryFilterString` string overrides any query string specified in the `queryFilterString` configured for the intent.
- `kendraQueryRequestPayload` – Use this string to specify an Amazon Kendra query. Your query can use any of the features of Amazon Kendra. If you don't specify a valid query, you get a `InvalidLambdaResponseException` exception. For more information, see [Query](#) in the *Amazon Kendra Developer Guide*.

After you have created the filter or query string, you send the response to Amazon Lex V2 with the dialogAction field of the response set to delegate. Amazon Lex V2 sends the query to Amazon Kendra and then returns the query response to the fulfillment code hook.

### Using the Fulfillment Code Hook for the Response

After Amazon Lex V2 sends a query to Amazon Kendra, the query response is returned to the AMAZON.KendraSearchIntent fulfillment Lambda function. The input event to the code hook contains the complete response from Amazon Kendra. The query data is in the same structure as the one returned by the Amazon Kendra Query operation. For more information, see [Query response syntax](#) in the *Amazon Kendra Developer Guide*.

The fulfillment code hook is optional. If one does not exist, or if the code hook doesn't return a message in the response, Amazon Lex V2 uses the closingResponse statement for responses.

### Example: Creating a FAQ Bot for an Amazon Kendra Index

This example creates an Amazon Lex V2 bot that uses an Amazon Kendra index to provide answers to users' questions. The FAQ bot manages the dialog for the user. It uses the AMAZON.KendraSearchIntent intent to query the index and to present the response to the user. Here is a summary of how you will create your FAQ bot using an Amazon Kendra index:

1. Create a bot that your customers will interact with to get answers from your bot.
2. Create a custom intent. Your bot requires at least one intent that is neither the AMAZON.KendraSearchIntent nor the AMAZON.FallbackIntent. This intent must contain least one utterance. This intent enables your bot to build, but is not used otherwise. Your FAQ bot will therefore contain at least three intents, as in the image below:

The screenshot shows the Amazon Lex console interface. At the top, it says "Amazon Lex" and "Lex > Bots > Bot: KendraTestBot". A dropdown menu shows "Draft version". On the left, a sidebar lists navigation options: "Bots", "KendraTestBot", "Bot versions", "Draft version", "All languages", "▼ English (US)" (which is expanded to show "Intents" and "Slot types"), "▼ Deployment" (expanded to show "Aliases" and "Channel integrations"), and "▼ Analytics" (expanded to show "CloudWatch metrics" and "Utterances statistics"). At the bottom of the sidebar is a "Related resources" section with a right-pointing arrow. The main content area on the right shows a table titled "Intents (3) Info". It includes a search bar with "Search intents" and a table with columns "Name" and "Intent". The table rows are: "KendraSearchIntent", "RequiredIntent", and "FallbackIntent".

3. Add the AMAZON.KendraSearchIntent intent to your bot and configure it to work with your [Amazon Kendra index](#).
4. Test the bot by making a query and verifying that the results from your Amazon Kendra index are documents that answer the query.

## Prerequisites

Before you can use this example, you need to create an Amazon Kendra index. For more information, see [Getting started with the Amazon Kendra console](#) in the *Amazon Kendra Developer Guide*. For this example, choose the sample dataset ([Sample AWS documentation](#)) as your data source.

### To create a FAQ bot:

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. In the navigation pane, choose **Bots**.
3. Choose **Create bot**.
  - a. For the **Creation method**, choose **Create a blank bot**.
  - b. In the **Bot configuration** section, give the bot a name that indicates its purpose, such as **KendraTestBot**, and an optional description. The name must be unique in your account.
  - c. In the **IAM Permissions** section, choose **Create a role with basic Amazon Lex permissions**. This will create an [AWS Identity and Access Management \(IAM\)](#) role with the permissions that Amazon Lex V2 needs to run your bot.
  - d. In the **Children's Online Privacy Protection Act (COPPA)** section, choose **No**.
  - e. In the **Idle session timeout** and **Advanced settings** sections, leave the default settings and choose **Next**.
  - f. Now you are in the **Add language to bot** section. In the menu under **Voice interaction**, select **None. This is only a text based application**. Leave the default settings for the remaining fields.
  - g. Choose **Done**. Amazon Lex V2 creates your bot and a default intent called **NewIntent**, and takes you to the page to configure this intent

To successfully build a bot, you must create at least one intent that is separate from the **AMAZON.FallbackIntent** and the **AMAZON.KendraSearchIntent**. This intent is required to build your Amazon Lex V2 bot, but isn't used for the FAQ response. This intent must contain at least one sample utterance and the utterance must not apply to any of the questions that your customer asks.

### To create the required intent:

1. In the **Intent details** section, give the intent a name, such as **RequiredIntent**.
2. For **Add intent**, choose **Create intent**.
3. In the **Sample utterances** section, type an utterance in the box next to **Add utterance**, such as **Required utterance**. Then choose **Add utterance**.
4. Choose **Save intent**.

Create the intent to search an Amazon Kendra index and the response message that it should return.

### To create an **AMAZON.KendraSearchIntent** intent and response message:

1. Select **Back to intents list** in the navigation pane to return to the **Intents** page for your bot. Choose **Add intent** and select **Use built-in intent** from the dropdown menu.
2. In the box that pops up, select the menu under **Built-in intent**. Enter **AMAZON.KendraSearchIntent** in the search bar and then choose it from the list.
3. Give the intent a name, such as **KendraSearchIntent**.
4. From the **Amazon Kendra index** dropdown menu, choose the index that you want the intent to search. The index that you created in the **Prerequisites** section should be available.
5. Select **Add**.

6. In the intent editor, scroll down to the **Fulfillment** section, select the right arrow to expand the section, and add the following message in the box under **On successful fulfillment**:

I found a link to a document that could help you: ((x-amz-lex:kendra-search-response-document-link-1)).

The screenshot shows the Amazon Lex V2 Intent Editor interface. At the top, there's a header with the intent name and a 'Create intent' button. Below it, the 'Fulfillment' section is expanded, showing a message box containing the placeholder text: "I found a link to a document that could help you: ((x-amz-lex:kendra-search-response-document-link-1)).". To the right of this, there's a partially visible 'In case of failure' section. Below the fulfillment section, the 'Closing response' section is expanded, showing another message box with a placeholder. At the bottom of the editor, there are buttons for 'Set values', 'Add conditional branching', and a plus sign icon.

For more information about the Amazon Kendra Search Response, see [Using the Search Response](#).

7. Choose **Save intent**, and then choose **Build** to build the bot. When the bot is ready, the banner at the top of the screen turns green and displays a success message.

Finally, use the console test window to test responses from your bot.

**To test your FAQ bot:**

1. After the bot is successfully built, choose **Test**.
2. Enter **What is Amazon Kendra?** in the console test window. Verify that the bot responds with a link.
3. For more information about configuring AMAZON.KendraSearchIntent, see [AMAZON.KendraSearchIntent](#) and [KendraConfiguration](#).

## AMAZON.PauseIntent

Responds to words and phrases that enable the user to pause an interaction with a bot so that they can return to it later. Your Lambda function or application needs to save intent data in session variables, or you need to use the [GetSession](#) operation to retrieve intent data when you resume the current intent.

Common utterances:

- pause
- pause that

## AMAZON.RepeatIntent

Responds to words and phrases that enable the user to repeat the previous message. Your application needs to use a Lambda function to save the previous intent information in session variables, or you need to use the [GetSession](#) operation to get the previous intent information.

Common utterances:

- repeat
- say that again
- repeat that

## AMAZON.ResumeIntent

Responds to words and phrases that enable the user to resume a previously paused intent. Your Lambda function or application must manage the information required to resume the previous intent.

Common utterances:

- resume
- continue
- keep going

## AMAZON.StartOverIntent

Responds to words and phrases that enable the user to stop processing the current intent and start over from the beginning. You can use your Lambda function or the [PutSession](#) operation to elicit the first slot value again.

Common utterances:

- start over
- restart
- start again

## AMAZON.StopIntent

Responds to words and phrases that indicate that the user wants to stop processing the current intent and end the interaction with a bot. Your Lambda function or application should clear any existing attributes and slot type values and then end the interaction.

Common utterances:

- stop
- off
- shut up

## Built-in slot types

Amazon Lex supports built-in slot types that define how data in the slot is recognized and handled. You can create slots of these types in your intents. This eliminates the need to create enumeration values for commonly used slot data such as date, time, and location. Built-in slot types do not have versions.

Slot Type	Short Description	Supported Locales	
AMAZON.AlphaNumeric ( <a href="#">p. 76</a> )	Recognizes words made up of letters and numbers.	All locales except Korean (ko-KR)	
AMAZON.City ( <a href="#">p. 77</a> )	Recognizes words that represent a city.	All locales	
AMAZON.Country ( <a href="#">p. 77</a> )	Recognizes words that represent a country.	All locales	
AMAZON.Date ( <a href="#">p. 78</a> )	Recognizes words that represent a date and converts them to a standard format.	All locales	
AMAZON.Duration ( <a href="#">p. 78</a> )	Recognizes words that represent duration and converts them to a standard format.	All locales	
AMAZON.EmailAddress ( <a href="#">p. 78</a> )	Recognizes words that represent an email address and converts them into a standard email address.	All locales	
AMAZON.FirstName ( <a href="#">p. 79</a> )	Recognizes words that represent a first name.	All locales	
AMAZON.LastName ( <a href="#">p. 79</a> )	Recognizes words that represent a last name.	All locales	
AMAZON.Number ( <a href="#">p. 79</a> )	Recognizes numeric words and converts them into digits.	All locales	

Slot Type	Short Description	Supported Locales	
<a href="#">AMAZON.Percentage (p. 80)</a>	Recognizes words that represent a percentage and converts them to a number and a percent sign (%).	All locales	
<a href="#">AMAZON.PhoneNumber (p. 80)</a>	Recognizes words that represent a phone number and converts them into a numeric string.	All locales	
<a href="#">AMAZON.State (p. 80)</a>	Recognizes words that represent a state.	All locales	
<a href="#">AMAZON.StreetName (p. 80)</a>	Recognizes words that represent a street name.	All locales	
<a href="#">AMAZON.Time (p. 81)</a>	Recognizes words that indicate times and converts them into a time format.	All locales	
<a href="#">AMAZON.UKPostalCode (p. 80)</a>	Recognizes words that represent a UK post code and converts them to a standard form.	English (British) (en-GB) only	
<a href="#">AMAZON.FreeFormInput (p. 80)</a>	Recognizes strings that consist of any words or characters.	All locales	

## AMAZON.AlphaNumeric

Recognizes strings made up of letters and numbers, such as **APQ123**.

This slot type is not available in the Korean (ko-KR) locale.

You can use the AMAZON.AlphaNumeric slot type for strings that contain:

- Alphabetical characters, such as **ABC**
- Numeric characters, such as **123**
- A combination of alphanumeric characters, such as **ABC123**

The AMAZON.AlphaNumeric slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter letters. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

You can add a regular expression to the AMAZON.AlphaNumeric slot type to validate values entered for the slot. For example, you can use a regular expression to validate:

- Canadian postal codes
- Driver's license numbers

- Vehicle identification numbers

Use a standard regular expression. Amazon Lex V2 supports the following characters in the regular expression:

- A-Z, a-z
- 0-9

Amazon Lex V2 also supports Unicode characters in regular expressions. The form is \u*Unicode*. Use four digits to represent Unicode characters. For example, [\u0041-\u005A] is equivalent to [A-Z].

The following regular expression operators are not supported:

- Infinite repeaters: \*, +, or {x,} with no upper bound.
- Wild card (.)

The maximum length of the regular expression is 300 characters. The maximum length of a string stored in an AMAZON.AlphaNumeric slot type that uses a regular expression is 30 characters.

The following are some example regular expressions.

- Alphanumeric strings, such as **APQ123** or **APQ1**: [A-Z]{3}[0-9]{1,3} or a more constrained [A-DP-T]{3} [1-5]{1,3}
- US Postal Service Priority Mail International format, such as **CP123456789US**: CP[0-9]{9}US
- Bank routing numbers, such as **123456789**: [0-9]{9}

To set the regular expression for a slot type, use the console or the [CreateSlotType](#) operation. The regular expression is validated when you save the slot type. If the expression isn't valid, Amazon Lex V2 returns an error message.

When you use a regular expression in a slot type, Amazon Lex V2 checks input to slots of that type against the regular expression. If the input matches the expression, the value is accepted for the slot. If the input does not match, Amazon Lex V2 prompts the user to repeat the input.

## AMAZON.City

Provides a list of local and world cities. The slot type recognizes common variations of city names. Amazon Lex V2 doesn't convert from a variation to an official name.

Examples:

- New York
- Reykjavik
- Tokyo
- Versailles

## AMAZON.Country

The names of countries around the world. Examples:

- Australia
- Germany

- Japan
- United States
- Uruguay

## AMAZON.Date

Converts words that represent dates into a date format.

The date is provided to your intent in ISO-8601 date format. The date that your intent receives in the slot can vary depending on the specific phrase uttered by the user.

- Utterances that map to a specific date, such as "today," "now," or "November twenty-fifth," convert to a complete date: 2020-11-25. This defaults to dates *on or after* the current date.
- Utterances that map to a future week, such as "next week," convert to the date of the last day of the current week. In ISO-8601 format, the week starts on Monday and ends on Sunday. For example, if today is 2020-11-25, "next week" converts to 2020-11-29. Dates that map to the current or previous week convert to the first day of the week. For example, if today is 2020-11-25, "last week" converts to 2020-11-16.
- Utterances that map to a future month, but not a specific day, such as "next month," convert to the last day of the month. For example, if today is 2020-11-25, "next month" converts to 2020-12-31. For dates that map to the current or previous month convert to the first day of the month. For example, if today is 2020-11-25, "this month" maps to 2020-11-01.
- Utterances that map to a future year, but not a specific month or day, such as "next year," convert to the last day of the following year. For example, if today is 2020-11-25, "next year" converts to 2021-12-31. For dates that map to the current or previous year convert to the first day of the year. For example, if today is 2020-11-25, "last year" converts to 2019-01-01.

## AMAZON.Duration

Converts words that indicate durations into a numeric duration.

The duration is resolved to a format based on the [ISO-8601 duration format](#), PnYnMnWnDTnHnMnS. The P indicates that this is a duration, the n is a numeric value, and the capital letter following the n is the specific date or time element. For example, P3D means 3 days. A T is used to indicate that the remaining values represent time elements rather than date elements.

Examples:

- "ten minutes": PT10M
- "five hours": PT5H
- "three days": P3D
- "forty five seconds": PT45S
- "eight weeks": P8W
- "seven years": P7Y
- "five hours ten minutes": PT5H10M
- "two years three hours ten minutes": P2YT3H10M

## AMAZON.EmailAddress

Recognizes words that represent an email address provided as username@domain. Addresses can include the following special characters in a user name: underscore (\_), hyphen (-), period (.), and the plus sign (+).

The AMAZON.EmailAddress slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter email addresses. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

## AMAZON.FirstName

Commonly used first names. This slot type recognizes both formal names and informal nicknames. The name sent to your intent is the value sent by the user. Amazon Lex V2 doesn't convert from the nickname to the formal name.

For first names that sound alike but are spelled differently, Amazon Lex V2 sends your intent a single common form.

The AMAZON.FirstName slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter names. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

Examples:

- Emily
- John
- Sophie

## AMAZON.LastName

Commonly used last names. For names that sound alike that are spelled differently, Amazon Lex V2 sends your intent a single common form.

The AMAZON.LastName slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter names. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

Examples:

- Brosky
- Dasher
- Evers
- Parres
- Welt

## AMAZON.Number

Converts words or numbers that express a number into digits, including decimal numbers. The following table shows how the AMAZON.Number slot type captures numeric words.

Input	Response
one hundred twenty three point four five	123.45
one hundred twenty three dot four five	123.45
point four two	0.42
point forty two	0.42

Input	Response
232.998	232.998
50	50
-15	-15
minus 15	-15

## AMAZON.Percentage

Converts words and symbols that represent a percentage into a numeric value with a percent sign (%).

If the user enters a number without a percent sign or the word "percent," the slot value is set to the number. The following table shows how the AMAZON.Percentage slot type captures percentages.

Input	Response
50 percent	50%
0.4 percent	0.4%
23.5%	23.5%
twenty five percent	25%

## AMAZON.PhoneNumber

Converts the numbers or words that represent a phone number into a string format without punctuation as follows.

Type	Description	Input	Result
International number with leading plus (+) sign	11-digit number with leading plus sign.	+61 7 4445 1061 +1 (509) 555-1212	+61744431061 +15095551212
International number without leading plus (+) sign	11-digit number without leading plus sign	1 (509) 555-1212 61 7 4445 1061	15095551212 61744451061
National number	10-digit number without international code	(03) 5115 4444 (509) 555-1212	0351154444 5095551212
Local number	7-digit phone number without an international code or an area code	555-1212	5551212

## AMAZON.State

The names of geographical and political regions within countries.

Examples:

- Bavaria
- Fukushima Prefecture
- Pacific Northwest
- Queensland
- Wales

## AMAZON.StreetName

The names of streets within a typical street address. This includes just the street name, not the house number.

Examples:

- Canberra Avenue
- Front Street
- Market Road

## AMAZON.Time

Converts words that represent times into time values. Includes resolutions for ambiguous times. When a user enters an ambiguous time, Amazon Lex V2 uses the `slots` attribute of a Lambda event to pass resolutions for the ambiguous times to your Lambda function. For example, if your bot prompts the user for a delivery time, the user can respond by saying "10 o'clock." This time is ambiguous. It means either 10:00 AM or 10:00 PM. In this case, the value in the `interpretedValue` field is `null`, and the `resolvedValues` field contains the two possible resolutions of the time. Amazon Lex V2 inputs the following into the Lambda function:

```
"slots": {  
    "deliveryTime": {  
        "value": {  
            "originalValue": "10 o'clock",  
            "interpretedValue": null,  
            "resolvedValues": [  
                "10:00", "22:00"  
            ]  
        }  
    }  
}
```

When the user responds with an unambiguous time, Amazon Lex V2 sends the time to your Lambda function in the `interpretedValue` field of the `slots` attribute of the Lambda event. For example, if your user responds to the prompt for a delivery time with "10:00 AM," Amazon Lex V2 inputs the following into the Lambda function:

```
"slots": {  
    "deliveryTime": {  
        "value": {  
            "originalValue": "10 AM",  
            "interpretedValue": 10:00,  
            "resolvedValues": [  
                "10:00"  
            ]  
        }  
    }  
}
```

}

For more information about the data sent from Amazon Lex V2 to a Lambda function, see [Input event format \(p. 196\)](#).

## AMAZON.UKPostalCode

Converts words that represent a UK postal code to a standard format. The AMAZON.UKPostalCode slot type validates and resolves the post code to a set of standardized formats, but it doesn't check to make sure that the post code is valid. Your application must validate the post code.

The AMAZON.UKPostalCode slot type is available only in the English (UK) (en-GB) locale.

The AMAZON.UKPostalCode slot type supports inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter letters. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

The slot type recognizes all valid post code formats, including British overseas territories. The valid formats are (A represents a letter, 9 represents a digit):

- AA9A 9AA
- A9A 9AA
- A9 9AA
- A99 9AA
- AA9 9AA
- AA99 9AA

For text input, the user can enter any mix of upper and lower case letters. The user can use or omit the space in the post code. The resolved value will always include the space in the proper location for the post code.

For spoken input, the user can speak the individual characters, or they can use double letter pronunciations, such as "double A" or "double 9". They can also use double-digit pronunciations, such as "ninety-nine" for "99".

## AMAZON.FreeFormInput

AMAZON.FreeFormInput can be used to capture free form input from the end user. It recognizes strings that consist of words or characters. The resolved value is the entire input utterance.

Example:

Bot: Please provide feedback from your call experience.

User: I got the answers to all of my questions, and I was able to complete the transaction.

Note:

- AMAZON.FreeFormInput can be used to capture free form input as-is from the end user.
- AMAZON.FreeFormInput cannot be used in intent sample utterances.
- AMAZON.FreeFormInput cannot have slot sample utterances.
- AMAZON.FreeFormInput is only recognized when elicited for.
- AMAZON.FreeFormInput does not support wait and continue.
- AMAZON.FreeFormInput is currently not supported in the Amazon Connect Chat channel.
- When a AMAZON.FreeFormInput slot is elicited, FallbackIntent will not be triggered.

- When a AMAZON.FreeFormInput slot is elicited, there will be no intent switch.

## Using a custom grammar slot type

With the grammar slot type, you can author your own grammar in the XML format per the SRGS specification to collect information in a conversation. Amazon Lex V2 recognizes utterances matched by the rules specified in the grammar. You can also provide semantic interpretation rules using ECMAScript tags within the grammar files. Amazon Lex then returns properties set in the tags as resolved values when a match occurs.

You can only create grammar slot types in the English (Australia), English (UK), and English (US) locales.

There are two parts to a grammar slot type. The first is the grammar itself written using the SRGS specification format. The grammar interprets the utterance from the user. If the utterance is accepted by the grammar it is matched, otherwise it is rejected. If an utterance is matched it is passed on to the script if there is one.

The second is part of a grammar slot type is an optional script written in ECMAScript that transforms the input to the resolved values returned by the slot type. For example, you can use a script to convert spoken numbers to digits. ECMAScript statements are enclosed in the <tag> element.

The following example is in the XML format per the SRGS specification that shows a valid grammar accepted by Amazon Lex V2. It defines a grammar slot type that accepts card numbers and determines if they are for regular or premium accounts. For more information about the acceptable syntax, see the [Grammar definition \(p. 84\)](#) and the [Script format \(p. 93\)](#) topics.

```
<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
    xml:lang="en-US" *tag-format="semantics/1.0" root="card_number">

    <rule id="card_number" scope="public">
        <item repeat="0-1">
            card number
        </item>
        <item>
            seven
            <tag>out.value = "7";</tag>
        </item>
        <item>
            <one-of>
                <item>
                    two four one
                    <tag> out.value = out.value + "241"; out.card_type = "premium"; </tag>
                </item>
                <item>
                    zero zero one
                    <tag> out.value = out.value + "001"; out.card_type = "regular";</tag>
                </item>
            </one-of>
        </item>
    </rule>
</grammar>
```

The above grammar only accepts two types of card numbers: 7241 or 7001. Both of these may be optionally prefixed with "card number". It also contains ECMAScript tags that can be used for semantic interpretation. With semantic interpretation, the utterance "card number seven two four one" would return following object:

```
{  
    "value": "7241",
```

```
    "card_type": "premium"  
}
```

This object is returned as a JSON-serialized string in the `resolvedValues` object returned by the [RecognizeText](#), [RecognizeUtterance](#), and [StartConversation](#) operations.

## Adding a grammar slot type

### To add a grammar slot type

1. Upload the XML definition of your slot type to an S3 bucket. Make a note of the bucket name and the path to the file.
2. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
3. From the left menu, choose **Bots** and then choose the bot to add the grammar slot type to.
4. Choose **View languages**, and then choose the language to add the grammar slot type to.
5. Choose **View slot types**.
6. Choose **Add slot type**, and then choose **Add grammar slot type**.
7. Give the slot type a name, and then choose **Add**.
8. Choose the S3 bucket that contains your definition file and enter the path to the file. Choose **Save slot type**.

## Grammar definition

This topic shows the parts of the SRGS specification that Amazon Lex V2 supports. All of the rules are defined in the SRGS specification. For more information, see the [Speech recognition grammar specification version 1.0](#) W3C recommendation.

### Topics

- [Header declarations \(p. 85\)](#)
- [Supported XML elements \(p. 87\)](#)
- [Tokens \(p. 87\)](#)
- [Rule reference \(p. 88\)](#)
- [Sequences and encapsulation \(p. 89\)](#)
- [Repeats \(p. 89\)](#)
- [Language \(p. 90\)](#)
- [Tags \(p. 91\)](#)
- [Weights \(p. 92\)](#)

This document includes material copied and derived from the W3C Speech Recognition Grammar Specification Version 1.0 (available at <https://www.w3.org/TR/speech-grammar/>). Citation information follows:

[Copyright](#) © 2004 W3C® (MIT, ERCIM, Keio, All Rights Reserved. W3C [liability](#), [trademark](#), [document use](#) and [software licensing](#) rules apply.

The SRGS specification document, a [W3C Recommendation](#), is available from the W3C under the following license.

### License text

License

By using and/or copying this document, or the W3C document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the W3C document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- A link or URL to the original W3C document.
- The pre-existing copyright notice of the original author, or if it doesn't exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [\$date-of-document] [World Wide Web Consortium, \(MIT, ERCIM, Keio, Beihang\)](#). <http://www.w3.org/Consortium/Legal/2015/doc-license>"
- *If it exists*, the STATUS of the W3C document.

When space permits, inclusion of the full text of this **NOTICE** should be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of W3C documents is granted pursuant to this license, except as follows: To facilitate implementation of the technical specifications set forth in this document, anyone may prepare and distribute derivative works and portions of this document in software, in supporting materials accompanying software, and in documentation of software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

In addition, "Code Components" —Web IDL in sections clearly marked as Web IDL; and W3C-defined markup (HTML, CSS, etc.) and computer programming language code clearly marked as code examples—are licensed under the [W3C Software License](#).

The notice is:

"Copyright © 2015 W3C® (MIT, ERCIM, Keio, Beihang). This software or document includes material copied from or derived from [title and URI of the W3C document]."

#### Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

## Header declarations

The following table shows the header declarations supported by the grammar slot type. For more information, see [Grammar header declarations](#) in the *Speech recognition grammar specification version 1* W3C recommendation.

Declaration	Specification requirement	XML form	Amazon Lex support	Specification
Grammar version	Required	<a href="#">4.3: version attribute on grammar element</a>	Required	SRGS
XML namespace	Required (XML only)	<a href="#">4.3: xmlns attribute on grammar element</a>	Required	SRGS
Document type	Required (XML only)	<a href="#">4.3: XML DOCTYPE</a>	Recommended	SRGS
Character encoding	Recommended	<a href="#">4.4: encoding attribute in XML declaration</a>	Recommended	SRGS
Language	Required in voice mode Ignored in DTMF mode	<a href="#">4.5: xml:lang attribute on grammar element</a>	Required in voice mode Ignored in DTMF mode	SRGS
Mode	Optional	<a href="#">4.6: mode attribute on grammar element</a>	Optional	SRGS
Root rule	Optional	<a href="#">4.7: root attribute on grammar element</a>	<b>Required</b>	SRGS
Tag format	Optional	<a href="#">4.8: tag-format attribute on grammar element</a>	<b>String literal and ECMAScript are supported</b>	SRGS, SISR
Base URI	Optional	<a href="#">4.9: xml:base attribute on grammar element</a>	Optional	SRGS
Pronunciation lexicon	Optional, multiple allowed	<a href="#">4.10: lexicon element</a>	<b>Not supported</b>	SRGS, PLS
Metadata	Optional, multiple allowed	<a href="#">4.11.1: meta element</a>	Required	SRGS
XML metadata	Optional, XML only	<a href="#">4.11.2: metadata element</a>	Optional	SRGS
Tag	Optional, multiple allowed	<a href="#">4.12: tag element</a>	<b>Global tags not supported</b>	SRGS

### Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
    "http://www.w3.org/TR/speech-grammar/grammar.dtd">
```

```
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xml:base="http://www.example.com/base-file-path"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US"
  version="1.0"
  mode="voice"
  root="city"
  tag-format="semantics/1.0">
```

## Supported XML elements

Amazon Lex V2 supports the following XML elements for custom grammars:

- <item>
- <token>
- <tag>
- <one-of>
- <rule-ref>

## Tokens

The following table shows the token specifications supported by the grammar slot type. For more information, see [Tokens](#) in the *Speech recognition grammar specification version 1 W3C recommendation*.

Token type	Example	Supported?
Single unquoted token	hello	Yes
Single unquoted token: non-alphabetic	2	Yes
Single quoted token, no white space	"hello"	Yes, drop double quotes when it only contains a single token
Two tokens delimited by white space	bon voyage	Yes
Four tokens delimited by white space	this is a test	Yes
Single quoted token, including white space	"San Francisco	No
Single XML token in <token> tag	<token>San Francisco</token>	No (same as single quoted token with white space)

### Notes

- *Single quoted token including white space* – The specification requires words enclosed in double quotes be treated as a single token. Amazon Lex V2 treats them as white space delimited tokens.
- *Single XML token in <token>* – The specification requires words delimited by <token> to represent one token. Amazon Lex V2 treats them as white space delimited tokens.

- Amazon Lex V2 throws a validation error when either usage is found in your grammar.

### Example

```
<rule id="state" scope="public">
  <one-of>
    <item>FL</item>
    <item>MA</item>
    <item>NY</item>
  </one-of>
</rule>
```

## Rule reference

The following table summarizes the various forms of rule reference that are possible within grammar documents. For more information, see [Rule reference](#) in the *Speech recognition grammar specification version 1 W3C recommendation*.

Reference type	XML form	Supported
<a href="#">2.2.1</a> Explicit local rule reference	<ruleref uri="#rulename"/>	Yes
<a href="#">2.2.2</a> Explicit reference to a named rule of a grammar identified by a <a href="#">URI</a>	<ruleref uri="grammarURI#rulename"/>	No
<a href="#">2.2.2</a> Implicit reference to the root rule of a grammar identified by a <a href="#">URI</a>	<ruleref uri="grammarURI"/>	No
<a href="#">2.2.2</a> Explicit reference to a named rule of a grammar identified by a <a href="#">URI</a> with a <a href="#">media type</a>	<ruleref uri="grammarURI#rulename" type="media-type"/>	No
<a href="#">2.2.2</a> Implicit reference to the root rule of a grammar identified by a <a href="#">URI</a> with a <a href="#">media type</a>	<ruleref uri="grammarURI" type="media-type"/>	No
<a href="#">2.2.3</a> Special rule definitions	<ruleref special="NULL"/> <ruleref special="VOID"/> <ruleref special="GARBAGE"/>	No

### Notes

1. Grammar URI is an external URI. For example, <http://grammar.example.com/world-cities.grxml>.
2. Media type can be:
  - `application/srgs+xml`

- text/plain

#### Example

```
<rule id="city" scope="public">
  <one-of>
    <item>Boston</item>
    <item>Philadelphia</item>
    <item>Fargo</item>
  </one-of>
</rule>

<rule id="state" scope="public">
  <one-of>
    <item>FL</item>
    <item>MA</item>
    <item>NY</item>
  </one-of>
</rule>

<!-- "Boston MA" -> city = Boston, state = MA -->
<rule id="city_state" scope="public">
  <ruleref uri="#city"/> <ruleref uri="#state"/>
</rule>
```

## Sequences and encapsulation

The following example shows the supported sequences. For more information, see [Sequences and encapsulation](#) in the *Speech recognition grammar specification version 1 W3C recommendation*.

#### Example

```
<!-- sequence of tokens -->
this is a test

<!--sequence of rule references-->
<ruleref uri="#action"/> <ruleref uri="#object"/>

<!--sequence of tokens and rule references-->
the <ruleref uri="#object"/> is <ruleref uri="#color"/>

<!-- sequence container -->
<item>fly to <ruleref uri="#city"/> </item>
```

## Repeats

The following table shows the supported repeated expansions for rules. For more information, see [Repeats](#) in the *Speech recognition grammar specification version 1 W3C recommendation*.

XML form	Behavior	Supported?
Example		
<code>repeat="n"</code>	The contained expression is repeated exactly "n" times. "n" must be "0" or a positive integer.	Yes
<code>repeat="6"</code>		

XML form	Behavior	Supported?
Example		
<code>repeat="m-n"</code> <code>repeat="4-6"</code>	The contained expansion is repeated between "m" and "n" times (inclusive). "m" and "n" must both be "0" or a positive integer, and "m" must be less than or equal to "n".	Yes
<code>repeat="m-"</code> <code>repeat="3-"</code>	The contained expansion is repeated "m" times or more (inclusive). "m" must be "0" or a positive integer. For example, "3-" declares that the contained expansion can occur three, four, five, or more times.	Yes
<code>repeat="0-1"</code>	The contained expansion is optional.	Yes
<code>&lt;item repeat="2-4" repeat-prob="0.8"&gt;</code>		No

## Language

The following discussion applies to language identifiers applied to grammars. For more information, see [Language](#) in the *Speech recognition grammar specification version 1* W3C recommendation.

By default a grammar is a single language document with a [language identifier](#) provided in the language declaration in the [grammar header](#). All tokens within that grammar, unless otherwise declared, **will be handled according to the grammar's language**. Grammar-level language declarations **are not supported**.

In the following example:

1. The grammar header declaration for the language "en-US" **is supported** by Amazon Lex V2.
2. Item-level language attachment (highlighted in *red*) **is not supported**. Amazon Lex V2 throws a validation error if a language attachment is different from the header declaration.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE grammar PUBLIC "-//W3C//DTD GRAMMAR 1.0//EN"
   "http://www.w3.org/TR/speech-grammar/grammar.dtd">

<!-- the default grammar language is US English -->
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0">

  <!--
      single language attachment to tokens
      "yes" inherits US English language
      "oui" is Canadian French language
  -->

```

```

-->
<rule id="yes">
  <one-of>
    <item>yes</item>
    <item xml:lang="fr-CA">oui</item>
  </one-of>
</rule>

<!-- Single language attachment to an expansion -->
<rule id="people1">
  <one-of xml:lang="fr-CA">
    <item>Michel Tremblay</item>
    <item>André Roy</item>
  </one-of>
</rule>
</grammar>

```

## Tags

The following discussion applies to defined for grammars. For more information, see [Tags in the Speech recognition grammar specification version 1 W3C recommendation](#).

Based on the SRGS specification, tags can be defined in the following ways:

1. As part of a header declaration as described in [Header declarations \(p. 85\)](#).
2. As part of a `<rule>` definition.

The following tag formats are supported:

- `semantics/1.0` (SISR, ECMAScript)
- `semantics/1.0-literals` (SISR string literals)

The following tag formats are not supported:

- `swi-semantics/1.0` (Nuance proprietary)

### Example

```

<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xml:base="http://www.example.com/base-file-path"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US"
  version="1.0"
  mode="voice"
  root="city"
  tag-format="semantics/1.0-literals">
  <rule id="no">
    <one-of>
      <item>no</item>
      <item>nope</item>
      <item>no way</item>
    </one-of>
    <tag>no</tag>
  </rule>
</grammar>

```

## Weights

You can add the *weight* attribute to an element. Weight is a positive floating point value that represents the degree to which the phrase in the item is boosted during speech recognition. For more information, see [Weights](#) in the Speech recognition grammar specification version 1 W3C recommendation.

Weights must be greater than 0 and less than or equal to 10, and can have only one decimal place. If the weight is greater than 0 and less than 1, the phrase is negatively boosted. If the weight is greater than 1 and less than or equal to 10, the phrase is positively boosted. A weight of 1 is equivalent to giving no weight at all, and there is no boosting for the phrase.

Assigning appropriate weights to items for improving speech recognition performance is a difficult task. Here are some tips you can follow for assigning weights:

- Start with a grammar without item weights assigned.
- Determine which patterns in the speech are frequently misidentified.
- Apply different values for weights until you notice an improvement in the speech recognition performance, and there are no regressions.

### Example 1

For example, if you have a grammar for airports, and you observe that *New York* is frequently misidentified as *Newark*, you can positively boost *New York* by assigning it a weight of 5.

```
<rule> id="airport">
  <one-of>
    <item>
      Boston
      <tag>out="Boston"</tag>
    </item>
    <item weight="5">
      New York
      <tag>out="New York"</tag>
    </item>
    <item>
      Newark
      <tag>out="Newark"</tag>
    </item>
  </one-of>
</rule>
```

### Example 2

For example, you have a grammar for the airline reservation code starting with an English alphabet followed by three digits. The reservation code most likely starts with B or D, but you observe that B is frequently misidentified as P, and D as T. You can positively boost B and D.

```
<rule> id="alphabet">
  <one-of>
    <item>A<tag>out.letters+= 'A';</tag></item>
    <item weight="3.5">B<tag>out.letters+= 'B';</tag></item>
    <item>C<tag>out.letters+= 'C';</tag></item>
    <item weight="2.9">D<tag>out.letters+= 'D';</tag></item>
    <item>E<tag>out.letters+= 'E';</tag></item>
    <item>F<tag>out.letters+= 'F';</tag></item>
    <item>G<tag>out.letters+= 'G';</tag></item>
    <item>H<tag>out.letters+= 'H';</tag></item>
    <item>I<tag>out.letters+= 'I';</tag></item>
    <item>J<tag>out.letters+= 'J';</tag></item>
```

```
<item>K<tag>out.letters+= 'K';</tag></item>
<item>L<tag>out.letters+= 'L';</tag></item>
<item>M<tag>out.letters+= 'M';</tag></item>
<item>N<tag>out.letters+= 'N';</tag></item>
<item>O<tag>out.letters+= 'O';</tag></item>
<item>P<tag>out.letters+= 'P';</tag></item>
<item>Q<tag>out.letters+= 'Q';</tag></item>
<item>R<tag>out.letters+= 'R';</tag></item>
<item>S<tag>out.letters+= 'S';</tag></item>
<item>T<tag>out.letters+= 'T';</tag></item>
<item>U<tag>out.letters+= 'U';</tag></item>
<item>V<tag>out.letters+= 'V';</tag></item>
<item>W<tag>out.letters+= 'W';</tag></item>
<item>X<tag>out.letters+= 'X';</tag></item>
<item>Y<tag>out.letters+= 'Y';</tag></item>
<item>Z<tag>out.letters+= 'Z';</tag></item>
</one-of>
</rule>
```

## Script format

Amazon Lex V2 supports the following ECMAScript features for defining grammars.

Amazon Lex V2 supports the following ECMAScript features when specifying tags in the grammar. tag-format must be sent to semantics/1.0 when ECMAScript tags are used in the grammar. For more information, see the [ECMA-262 ECMAScript 2021 language specification](#).

```
<grammar version="1.0"
xmlns="http://www.w3.org/2001/06/grammar"
xml:lang="en-US"
tag-format="semantics/1.0"
root="card_number">
```

### Topics

- [Variable statement \(p. 94\)](#)
- [Expressions \(p. 94\)](#)
- [If statement \(p. 97\)](#)
- [Switch statement \(p. 97\)](#)
- [Function declarations \(p. 97\)](#)
- [Iteration statement \(p. 97\)](#)
- [Block statement \(p. 98\)](#)
- [Comments \(p. 98\)](#)
- [Unsupported statements \(p. 98\)](#)

This document contains material from the ECMAScript standard (available at <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>). The ECMAScript language specification document is available from Ecma International under the following license.

### License text

© 2020 Ecma International

This document may be copied, published and distributed to others, and certain derivative works of it may be prepared, copied, published, and distributed, in whole or in part, provided that the above copyright notice and this Copyright License and Disclaimer are included on all such copies and derivative works. The only derivative works that are permissible under this Copyright License and Disclaimer are:

- (i) works which incorporate all or portion of this document for the purpose of providing commentary or explanation (such as an annotated version of the document),
- (ii) works which incorporate all or portion of this document for the purpose of incorporating features that provide accessibility,
- (iii) translations of this document into languages other than English and into different formats and
- (iv) works by making use of this specification in standard conformant products by implementing (e.g. by copy and paste wholly or partly) the functionality therein.

However, the content of this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as required to translate it into languages other than English or into a different format.

The official version of an Ecma International document is the English language version on the Ecma International website. In the event of discrepancies between a translated version and the official version, the official version shall govern.

The limited permissions granted above are perpetual and will not be revoked by Ecma International or its successors or assigns. This document and the information contained herein is provided on an "AS IS" basis and ECMA INTERNATIONAL DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE."

## Variable statement

A variable statement defines one or more variables.

```
var x = 10;
var x = 10, var y = <expression>;
```

## Expressions

Expression type	Syntax	Example	Supported?
Regular expression literal	String literal containing valid <a href="#">regex special characters</a>	"^\\d\\.\\\$"	No
Function	function functionName(parameters) { functionBody}	var x = function calc() {     return 10; }	No
Delete	delete expression	delete obj.property;	No
Void	void expression	void (2 == '2');	No
Typeof	typeof expression	typeof 42;	No
Member index	expression [ expressions ]	var fruits = ["apple"];	Yes

Expression type	Syntax	Example	Supported?
		fruits[0];	
Member dot	expression . identifier	out.value	yes
Arguments	expression (arguments)	new Date('1994-10-11')	Yes
Post increment	expression++	var x=10; x++;	Yes
Post decrement	expression--	var x=10; x--;	Yes
Pre increment	++expression	var x=10; ++x;	Yes
Pre decrement	--expression	var x=10; --x;	Yes
Unary plus / Unary minus	+expression / - expression	+x / -x;	Yes
Bit not	~ expression	const a = 5; console.log(~a);	Yes
Logical not	! expression	!(a > 0    b > 0)	Yes
Multiplicative	expression ('*'   '/'   '%') expression	(x + y) * (a / b)	Yes
Additive	expression ('+'   '-') expression	(a + b) - (a - (a + b))	Yes
Bit shift	expression ('<<'   '>>'   '>>>') expression	(a >> b) >>> c	Yes
Relative	expression ('<'   '>'   '<='   '>=' ) expression	if (a > b) { ... }	Yes
In	expression in expression	fruits[0] in otherFruits;	Yes
Equality	expression ('=='   '!=   '==='   '!==' ) expression	if (a == b) { ... }	Yes
Bit and / xor / or	expression ('&'   '^'   ' ') expression	a & b / a ^ b / a   b	Yes

Expression type	Syntax	Example	Supported?
Logical and / or	expression ('&&'   '  ') expression	if (a && (b   c)) { ...}	Yes
Ternary	expression ? expression : expression	a > b ? obj.prop : 0	Yes
Assignment	expression = expression	out.value = "string";	Yes
Assignment operator	expression ('*='   '/='   '+='   '-='   '%=') expression	a *= 10;	Yes
Assignment bitwise operator	expression ('<<='   '>>='   '>>>='   '&='   '^='   ' =') expression	a <<= 10;	Yes
Identifier	identifierSequence where <i>identifierSequence</i> is a sequence of <a href="#">valid characters</a>	fruits=[10, 20, 30];	Yes
Null literal	null	x = null;	Yes
Boolean literal	true   false	x = true;	Yes
String literal	'string' / "string"	a = 'hello', b = "world";	Yes
Decimal literal	integer [. digits [exponent]	111.11 e+12	Yes
Hex literal	0 (x   X)[0-9a-fA-F]	0x123ABC	Yes
Octal literal	0 [0-7]	"051"	Yes
Array literal	[ expression, ... ]	v = [a, b, c];	Yes
Object literal	{property: value, ...}	out = {value: 1, flag: false};	Yes
Parenthesized	( expressions )	x + (x + y)	Yes

## If statement

```
if (expressions) {  
    statements;  
} else {  
    statements;  
}
```

**Note:** In the preceding example, `expressions` and `statements` must be one of the supported ones from this document.

## Switch statement

```
switch (expression) {  
    case (expression):  
        statements  
        .  
        .  
        .  
    default:  
        statements  
}
```

**Note:** In the preceding example, `expressions` and `statements` must be one of the supported ones from this document.

## Function declarations

```
function functionIdentifier([parameterList, ...]) {  
    <function body>  
}
```

## Iteration statement

Iteration statements can be any one of the following:

```
// Do..While statement  
do {  
    statements  
} while (expressions)  
  
// While Loop  
while (expressions) {  
    statements  
}  
  
// For Loop  
for ([initialization]; [condition]; [final-expression])  
    statement  
  
// For..In  
for (variable in object) {  
    statement  
}
```

## Block statement

```
{  
    statements  
}  
  
// Example  
{  
    x = 10;  
    if (x > 10) {  
        console.log("greater than 10");  
    }  
}
```

**Note:** In the preceding example, statements provided in the block must be one of the supported ones from this document.

## Comments

```
// Single Line Comments  
/// <comment>"  
  
// Multiline comments  
/**  
<comment>  
**/
```

## Unsupported statements

Amazon Lex V2 doesn't support the following ECMAScript features.

### Topics

- [Empty statement \(p. 98\)](#)
- [Continue statement \(p. 98\)](#)
- [Break statement \(p. 99\)](#)
- [Return statement \(p. 99\)](#)
- [Throw statement \(p. 99\)](#)
- [Try statement \(p. 99\)](#)
- [Debugger statement \(p. 99\)](#)
- [Labeled statement \(p. 99\)](#)
- [Class declaration \(p. 100\)](#)

## Empty statement

The empty statement is used to provide no statement. The following is the syntax for an empty statement:

```
;
```

## Continue statement

The continue statement without a label is supported with the [Iteration statement \(p. 97\)](#). The continue statement with a label isn't supported.

```
// continue with label
// this allows the program to jump to a
// labelled statement (see labelled statement below)
continue <label>;
```

## Break statement

The break statement without a label is supported with the [Iteration statement \(p. 97\)](#). The break statement with a label isn't supported.

```
// break with label
// this allows the program to break out of a
// labelled statement (see labelled statement below)
break <label>;
```

## Return statement

```
return expression;
```

## Throw statement

The throw statement is used to throw a user-defined exception.

```
throw expression;
```

## Try statement

```
try {
    statements
}
catch (expression) {
    statements
}
finally {
    statements
}
```

## Debugger statement

The debugger statement is used to invoke debugging functionality provided by the environment.

```
debugger;
```

## Labeled statement

The labeled statement can be used with break or continue statements.

```
label:
    statements

// Example
let str = '';
loop1:
```

```
for (let i = 0; i < 5; i++) {
  if (i === 1) {
    continue loop1;
  }
  str = str + i;
}

console.log(str);
```

## Class declaration

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

## Industry grammars

*Industry grammars* are a set of XML files to use with the [grammar slot type](#). You can use these to quickly deliver a consistent end-user experience as you migrate interactive voice response work flows to Amazon Lex V2. You can select from a range of pre-built grammars across three domains: financial services, insurance, and telecom. There is also a generic set of grammars that you can use as a starting point for your own grammars.

The grammars contain the rules to collect the information and the [ECMAScript tags](#) for semantic interpretation.

## Grammars for financial services ([download](#))

The following grammars are supported for financial services: account and routing numbers, credit card and loan numbers, credit score, account opening and closing dates, and Social Security number.

### Account number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <!-- Test Cases

  Grammar will support the following inputs:

  Scenario 1:
    Input: My account number is A B C 1 2 3 4
    Output: ABC1234

  Scenario 2:
    Input: My account number is 1 2 3 4 A B C
    Output: 1234ABC

  Scenario 3:
```

```

Input: Hmm My account number is 1 2 3 4 A B C 1
Output: 123ABC1
-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item><ruleref uri="#alphanumeric"/><tag>out += rules.alphanumeric.alphanum;</tag></item>
    <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out += rules.alphabets.letters;</tag></item>
        <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.numbers;</tag></item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">account number is</item>
            <item repeat="0-1">Account Number</item>
            <item repeat="0-1">Here is my Account Number </item>
            <item repeat="0-1">Yes, It is</item>
            <item repeat="0-1">Yes It is</item>
            <item repeat="0-1">Yes It's</item>
            <item repeat="0-1">My account Id is</item>
            <item repeat="0-1">This is the account Id</item>
            <item repeat="0-1">account Id</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>

    <rule id="alphanumeric" scope="public">
        <tag>out.alphanum=""</tag>
        <item><ruleref uri="#alphabets"/><tag>out.alphanum += rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum += rules.digits.numbers;</tag></item>
        </rule>

        <rule id="alphabets">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.letters=""</tag>
            <tag>out.firstOccurrence=""</tag>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurrence += rules.digits.numbers; out.letters += out.firstOccurrence;</tag></item>
            <item repeat="1-1">
                <one-of>
                    <item>A<tag>out.letters+= 'A';</tag></item>
                    <item>B<tag>out.letters+= 'B';</tag></item>
                    <item>C<tag>out.letters+= 'C';</tag></item>
                    <item>D<tag>out.letters+= 'D';</tag></item>
                    <item>E<tag>out.letters+= 'E';</tag></item>
                    <item>F<tag>out.letters+= 'F';</tag></item>
                    <item>G<tag>out.letters+= 'G';</tag></item>
                    <item>H<tag>out.letters+= 'H';</tag></item>
                    <item>I<tag>out.letters+= 'I';</tag></item>
                    <item>J<tag>out.letters+= 'J';</tag></item>
                    <item>K<tag>out.letters+= 'K';</tag></item>
                    <item>L<tag>out.letters+= 'L';</tag></item>
                    <item>M<tag>out.letters+= 'M';</tag></item>
                </one-of>
            </item>
        </rule>
    
```

```

<item>N<tag>out.letters+='N';</tag></item>
<item>O<tag>out.letters+='O';</tag></item>
<item>P<tag>out.letters+='P';</tag></item>
<item>Q<tag>out.letters+='Q';</tag></item>
<item>R<tag>out.letters+='R';</tag></item>
<item>S<tag>out.letters+='S';</tag></item>
<item>T<tag>out.letters+='T';</tag></item>
<item>U<tag>out.letters+='U';</tag></item>
<item>V<tag>out.letters+='V';</tag></item>
<item>W<tag>out.letters+='W';</tag></item>
<item>X<tag>out.letters+='X';</tag></item>
<item>Y<tag>out.letters+='Y';</tag></item>
<item>Z<tag>out.letters+='Z';</tag></item>
</one-of>
</item>
</rule>

<rule id="digits">
<item repeat="0-1"><ruleref uri="#text"/></item>
<tag>out.numbers=""</tag>
<item repeat="1-10">
<one-of>
<item>0<tag>out.numbers+=0;</tag></item>
<item>1<tag>out.numbers+=1;</tag></item>
<item>2<tag>out.numbers+=2;</tag></item>
<item>3<tag>out.numbers+=3;</tag></item>
<item>4<tag>out.numbers+=4;</tag></item>
<item>5<tag>out.numbers+=5;</tag></item>
<item>6<tag>out.numbers+=6;</tag></item>
<item>7<tag>out.numbers+=7;</tag></item>
<item>8<tag>out.numbers+=8;</tag></item>
<item>9<tag>out.numbers+=9;</tag></item>
</one-of>
</item>
</rule>
</grammar>

```

## Routing number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="digits"
  mode="voice"
  tag-format="semantics/1.0">

  <!-- Test Cases

Grammar will support the following inputs:

  Scenario 1:
    Input: My routing number is 1 2 3 4 5 6 7 8 9
    Output: 123456789

  Scenario 2:
    Input: routing number 1 2 3 4 5 6 7 8 9
    Output: 123456789

  -->

<rule id="digits">

```

```

        <tag>out=""</tag>
        <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My routing number</item>
        <item repeat="0-1">Routing number of</item>
        <item repeat="0-1">The routing number is</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="singleDigit">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.digit=""</tag>
    <item repeat="16">
        <one-of>
            <item>0<tag>out.digit+=0;</tag></item>
            <item>zero<tag>out.digit+=0;</tag></item>
            <item>1<tag>out.digit+=1;</tag></item>
            <item>one<tag>out.digit+=1;</tag></item>
            <item>2<tag>out.digit+=2;</tag></item>
            <item>two<tag>out.digit+=2;</tag></item>
            <item>3<tag>out.digit+=3;</tag></item>
            <item>three<tag>out.digit+=3;</tag></item>
            <item>4<tag>out.digit+=4;</tag></item>
            <item>four<tag>out.digit+=4;</tag></item>
            <item>5<tag>out.digit+=5;</tag></item>
            <item>five<tag>out.digit+=5;</tag></item>
            <item>6<tag>out.digit+=6;</tag></item>
            <item>six<tag>out.digit+=5;</tag></item>
            <item>7<tag>out.digit+=7;</tag></item>
            <item>seven<tag>out.digit+=7;</tag></item>
            <item>8<tag>out.digit+=8;</tag></item>
            <item>eight<tag>out.digit+=8;</tag></item>
            <item>9<tag>out.digit+=9;</tag></item>
            <item>nine<tag>out.digit+=9;</tag></item>
        </one-of>
    </item>
</rule>
</grammar>
```

## Credit card number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="digits"
    mode="voice"
    tag-format="semantics/1.0">
```

```

<!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: My credit card number is 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7
Output: 1234567891234567

Scenario 2:
Input: card number 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7
Output: 1234567891234567

-->

<rule id="digits">
    <tag>out=""</tag>
    <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My credit card number is</item>
        <item repeat="0-1">card number</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="singleDigit">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.digit=""</tag>
    <item repeat="16">
        <one-of>
            <item>0<tag>out.digit+=0;</tag></item>
            <item>zero<tag>out.digit+=0;</tag></item>
            <item>1<tag>out.digit+=1;</tag></item>
            <item>one<tag>out.digit+=1;</tag></item>
            <item>2<tag>out.digit+=2;</tag></item>
            <item>two<tag>out.digit+=2;</tag></item>
            <item>3<tag>out.digit+=3;</tag></item>
            <item>three<tag>out.digit+=3;</tag></item>
            <item>4<tag>out.digit+=4;</tag></item>
            <item>four<tag>out.digit+=4;</tag></item>
            <item>5<tag>out.digit+=5;</tag></item>
            <item>five<tag>out.digit+=5;</tag></item>
            <item>6<tag>out.digit+=6;</tag></item>
            <item>six<tag>out.digit+=5;</tag></item>
            <item>7<tag>out.digit+=7;</tag></item>
            <item>seven<tag>out.digit+=7;</tag></item>
            <item>8<tag>out.digit+=8;</tag></item>
            <item>eight<tag>out.digit+=8;</tag></item>
            <item>9<tag>out.digit+=9;</tag></item>
            <item>nine<tag>out.digit+=9;</tag></item>
        </one-of>
    </item>
</rule>
</grammar>

```

## Loan ID

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

    Grammar will support the following inputs:

    Scenario 1:
        Input: My loan Id is A B C 1 2 3 4
        Output: ABC1234
    -->

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item><ruleref uri="#alphanumeric"/><tag>out += rules.alphanumeric.alphanum;</tag></item>
        <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out += rules.alphabets.letters;</tag></item>
        <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.numbers;</tag></item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">my loan number is</item>
            <item repeat="0-1">The loan number</item>
            <item repeat="0-1">The loan is </item>
            <item repeat="0-1">The number is</item>
            <item repeat="0-1">loan number</item>
            <item repeat="0-1">loan number of</item>
            <item repeat="0-1">loan Id is</item>
            <item repeat="0-1">My loan Id is</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>

    <rule id="alphanumeric" scope="public">
        <tag>out.alphanum=""</tag>
        <item><ruleref uri="#alphabets"/><tag>out.alphanum += rules.alphabets.letters;</tag></item>
        <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum += rules.digits.numbers;</tag></item>
    </rule>

    <rule id="alphabets">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.letters=""</tag>
    
```

```

<tag>out.firstOccurrence=""</tag>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurrence +=
rules.digits.numbers; out.letters += out.firstOccurrence;</tag></item>
<item repeat="1->
    <one-of>
        <item>A<tag>out.letters+='A';</tag></item>
        <item>B<tag>out.letters+='B';</tag></item>
        <item>C<tag>out.letters+='C';</tag></item>
        <item>D<tag>out.letters+='D';</tag></item>
        <item>E<tag>out.letters+='E';</tag></item>
        <item>F<tag>out.letters+='F';</tag></item>
        <item>G<tag>out.letters+='G';</tag></item>
        <item>H<tag>out.letters+='H';</tag></item>
        <item>I<tag>out.letters+='I';</tag></item>
        <item>J<tag>out.letters+='J';</tag></item>
        <item>K<tag>out.letters+='K';</tag></item>
        <item>L<tag>out.letters+='L';</tag></item>
        <item>M<tag>out.letters+='M';</tag></item>
        <item>N<tag>out.letters+='N';</tag></item>
        <item>O<tag>out.letters+='O';</tag></item>
        <item>P<tag>out.letters+='P';</tag></item>
        <item>Q<tag>out.letters+='Q';</tag></item>
        <item>R<tag>out.letters+='R';</tag></item>
        <item>S<tag>out.letters+='S';</tag></item>
        <item>T<tag>out.letters+='T';</tag></item>
        <item>U<tag>out.letters+='U';</tag></item>
        <item>V<tag>out.letters+='V';</tag></item>
        <item>W<tag>out.letters+='W';</tag></item>
        <item>X<tag>out.letters+='X';</tag></item>
        <item>Y<tag>out.letters+='Y';</tag></item>
        <item>Z<tag>out.letters+='Z';</tag></item>
    </one-of>
</item>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.numbers=""</tag>
    <item repeat="1-10">
        <one-of>
            <item>0<tag>out.numbers+=0;</tag></item>
            <item>1<tag>out.numbers+=1;</tag></item>
            <item>2<tag>out.numbers+=2;</tag></item>
            <item>3<tag>out.numbers+=3;</tag></item>
            <item>4<tag>out.numbers+=4;</tag></item>
            <item>5<tag>out.numbers+=5;</tag></item>
            <item>6<tag>out.numbers+=6;</tag></item>
            <item>7<tag>out.numbers+=7;</tag></item>
            <item>8<tag>out.numbers+=8;</tag></item>
            <item>9<tag>out.numbers+=9;</tag></item>
        </one-of>
    </item>
</rule>
</grammar>
```

## Credit score

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main">
```

```

mode="voice"
tag-format="semantics/1.0">

<!-- Test Cases

Grammar will support the following inputs:

    Scenario 1:
        Input: The number is fifteen
        Output: 15

    Scenario 2:
        Input: My credit score is fifteen
        Output: 15
    -->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <one-of>
        <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
    item>
        <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
        <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
    rules.above_twenty;</tag></item>
    </one-of>
</rule>

<rule id="text">
    <one-of>
        <item repeat="0-1">Credit score is</item>
        <item repeat="0-1">Last digits are</item>
        <item repeat="0-1">The number is</item>
        <item repeat="0-1">That's</item>
        <item repeat="0-1">It is</item>
        <item repeat="0-1">My credit score is</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>

```

```

<item>ten<tag>out=10;</tag></item>
<item>eleven<tag>out=11;</tag></item>
<item>twelve<tag>out=12;</tag></item>
<item>thirteen<tag>out=13;</tag></item>
<item>fourteen<tag>out=14;</tag></item>
<item>fifteen<tag>out=15;</tag></item>
<item>sixteen<tag>out=16;</tag></item>
<item>seventeen<tag>out=17;</tag></item>
<item>eighteen<tag>out=18;</tag></item>
<item>nineteen<tag>out=19;</tag></item>
<item>10<tag>out=10;</tag></item>
<item>11<tag>out=11;</tag></item>
<item>12<tag>out=12;</tag></item>
<item>13<tag>out=13;</tag></item>
<item>14<tag>out=14;</tag></item>
<item>15<tag>out=15;</tag></item>
<item>16<tag>out=16;</tag></item>
<item>17<tag>out=17;</tag></item>
<item>18<tag>out=18;</tag></item>
<item>19<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
        <item>20<tag>out=20;</tag></item>
        <item>30<tag>out=30;</tag></item>
        <item>40<tag>out=40;</tag></item>
        <item>50<tag>out=50;</tag></item>
        <item>60<tag>out=60;</tag></item>
        <item>70<tag>out=70;</tag></item>
        <item>80<tag>out=80;</tag></item>
        <item>90<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>

```

## Account opening date

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

```

```

Scenario 1:
Input: I opened account on July Two Thousand and Eleven
Output: 07/11

Scenario 2:
Input: I need account number opened on July Two Thousand and Eleven
Output: 07/11

-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item repeat="1-10">
        <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ "/";</tag></item>
        <one-of>
            <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</tag></item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty;</tag></item>
        </one-of>
    </item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">I opened account on </item>
        <item repeat="0-1">I need account number opened on </item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>
<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.mon=""</tag>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>
        <item>june<tag>out.mon+="06";</tag></item>
        <item>july<tag>out.mon+="07";</tag></item>
        <item>august<tag>out.mon+="08";</tag></item>
        <item>september<tag>out.mon+="09";</tag></item>
        <item>october<tag>out.mon+="10";</tag></item>
        <item>november<tag>out.mon+="11";</tag></item>
        <item>december<tag>out.mon+="12";</tag></item>
        <item>jan<tag>out.mon+="01";</tag></item>
        <item>feb<tag>out.mon+="02";</tag></item>
        <item>aug<tag>out.mon+="08";</tag></item>
        <item>sept<tag>out.mon+="09";</tag></item>
        <item>oct<tag>out.mon+="10";</tag></item>
        <item>nov<tag>out.mon+="11";</tag></item>
    </one-of>
</rule>

```

```

        <item>dec<tag>out.mon+="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <one-of>
        <item>zero<tag>out=0;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="thousands">
    <item>two thousand<!--<tag>out=2000;</tag>--></item>
    <item repeat="#0-1">and</item>
    <item repeat="#0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
    <item repeat="#0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></item>
    <item repeat="#0-1"><ruleref uri="#above_twenty"/><tag>out =
rules.above_twenty;</tag></item>
</rule>

<rule id="above_twenty">
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
    </one-of>
    <item repeat="#0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></
item>
    </rule>
</grammar>
```

## Automatic pay date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                    http://www.w3.org/TR/speech-grammar/grammar.xsd"
xml:lang="en-US" version="1.0"
root="main"
mode="voice"
tag-format="semantics/1.0">

<!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: I want to schedule auto pay for twenty five Dollar
Output: $25

Scenario 2:
Input: Setup automatic payments for twenty five dollars
Output: $25

-->

<rule id="main" scope="public">
    <tag>out="$"</tag>
    <one-of>
        <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</tag></item>
        <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</tag></item>
    </one-of>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">I want to schedule auto pay for</item>
        <item repeat="0-1">Setup automatic payments for twenty five dollars</item>
        <item repeat="0-1">Auto pay amount of</item>
        <item repeat="0-1">Set it up for</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
    </one-of>
</rule>

```

```

<item>three<tag>out.num+=3;</tag></item>
<item>four<tag>out.num+=4;</tag></item>
<item>five<tag>out.num+=5;</tag></item>
<item>six<tag>out.num+=6;</tag></item>
<item>seven<tag>out.num+=7;</tag></item>
<item>eight<tag>out.num+=8;</tag></item>
<item>nine<tag>out.num+=9;</tag></item>
</one-of>
<item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
        <item>fifteen<tag>out.teen+=15;</tag></item>
        <item>sixteen<tag>out.teen+=16;</tag></item>
        <item>seventeen<tag>out.teen+=17;</tag></item>
        <item>eighteen<tag>out.teen+=18;</tag></item>
        <item>nineteen<tag>out.teen+=19;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.tens = 0;</tag>
    <one-of>
        <item>twenty<tag>out.tens+=20;</tag></item>
        <item>thirty<tag>out.tens+=30;</tag></item>
        <item>forty<tag>out.tens+=40;</tag></item>
        <item>fifty<tag>out.tens+=50;</tag></item>
        <item>sixty<tag>out.tens+=60;</tag></item>
        <item>seventy<tag>out.tens+=70;</tag></item>
        <item>eighty<tag>out.tens+=80;</tag></item>
        <item>ninety<tag>out.tens+=90;</tag></item>
        <item>hundred<tag>out.tens+=100;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens += rules.digits.num;</tag></item>
</rule>

<rule id="currency">
    <one-of>
        <item repeat="0-1">dollars</item>
        <item repeat="0-1">Dollars</item>
        <item repeat="0-1">dollar</item>
        <item repeat="0-1">Dollar</item>
    </one-of>
</rule>

<rule id="sub_hundred">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.sh = 0;</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></item>
        <item><ruleref uri="#above_twenty"/><tag>out.sh += rules.above_twenty.tens;</tag></item>
    </one-of>
</rule>
```

```

        </item>
        <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></item>
    </one-of>
</rule>

<rule id="subThousands">
    <ruleref uri="#sub_hundreded"/><tag>out = (100 * rules.sub_hundreded.sh);</tag>
    hundred
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty.tens;</tag></item>
        <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</tag></item>
        <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
</grammar>

```

### Credit card expiration date

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="dateCardExpiration"
    mode="voice"
    tag-format="semantics/1.0">

    <rule id="dateCardExpiration" scope="public">
        <tag>out=""</tag>
        <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months;</tag></item>
        <item repeat="1"><ruleref uri="#year"/><tag>out += " " + rules.year.yr;</tag></item>
    </rule>

    <!-- Test Cases

    Grammar will support the following inputs:

    Scenario 1:
        Input: My card expiration date is july eleven
        Output: 07 2011

    Scenario 2:
        Input: My card expiration date is may twenty six
        Output: 05 2026

    -->

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">My card expiration date is </item>
            <item repeat="0-1">Expiration date is </item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>

```

```

        <item>My</item>
    </one-of>
</rule>

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>january<tag>out="01";</tag></item>
        <item>february<tag>out="02";</tag></item>
        <item>march<tag>out="03";</tag></item>
        <item>april<tag>out="04";</tag></item>
        <item>may<tag>out="05";</tag></item>
        <item>june<tag>out="06";</tag></item>
        <item>july<tag>out="07";</tag></item>
        <item>august<tag>out="08";</tag></item>
        <item>september<tag>out="09";</tag></item>
        <item>october<tag>out="10";</tag></item>
        <item>november<tag>out="11";</tag></item>
        <item>december<tag>out="12";</tag></item>
        <item>jan<tag>out="01";</tag></item>
        <item>feb<tag>out="02";</tag></item>
        <item>aug<tag>out="08";</tag></item>
        <item>sept<tag>out="09";</tag></item>
        <item>oct<tag>out="10";</tag></item>
        <item>nov<tag>out="11";</tag></item>
        <item>dec<tag>out="12";</tag></item>
        <item>1<tag>out="01";</tag></item>
        <item>2<tag>out="02";</tag></item>
        <item>3<tag>out="03";</tag></item>
        <item>4<tag>out="04";</tag></item>
        <item>5<tag>out="05";</tag></item>
        <item>6<tag>out="06";</tag></item>
        <item>7<tag>out="07";</tag></item>
        <item>8<tag>out="08";</tag></item>
        <item>9<tag>out="09";</tag></item>
        <item>ten<tag>out="10";</tag></item>
        <item>eleven<tag>out="11";</tag></item>
        <item>twelve<tag>out="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>
```

```

<rule id="year">
    <tag>out.yr="20"</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.yr += rules.teens;</tag></item>
        <item><ruleref uri="#above_twenty"/><tag>out.yr += rules.above_twenty;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
        <item>10<tag>out=10;</tag></item>
        <item>11<tag>out=11;</tag></item>
        <item>12<tag>out=12;</tag></item>
        <item>13<tag>out=13;</tag></item>
        <item>14<tag>out=14;</tag></item>
        <item>15<tag>out=15;</tag></item>
        <item>16<tag>out=16;</tag></item>
        <item>17<tag>out=17;</tag></item>
        <item>18<tag>out=18;</tag></item>
        <item>19<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
        <item>20<tag>out=20;</tag></item>
        <item>30<tag>out=30;</tag></item>
        <item>40<tag>out=40;</tag></item>
        <item>50<tag>out=50;</tag></item>
        <item>60<tag>out=60;</tag></item>
        <item>70<tag>out=70;</tag></item>
        <item>80<tag>out=80;</tag></item>
        <item>90<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>
```

## Statement date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                    http://www.w3.org/TR/speech-grammar/grammar.xsd"
xml:lang="en-US" version="1.0"
root="main"
mode="voice"
tag-format="semantics/1.0">

<!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: Show me statements from July Five Two Thousand and Eleven
Output: 07/5/11

Scenario 2:
Input: Show me statements from July Sixteen Two Thousand and Eleven
Output: 07/16/11

Scenario 3:
Input: Show me statements from July Thirty Two Thousand and Eleven
Output: 07/30/11
-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item>
        <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ "/";</tag></item>
        <one-of>
            <item><ruleref uri="#digits"/><tag>out += rules.digits + "/";</tag></
item>
            <item><ruleref uri="#teens"/><tag>out += rules.teens+ "/";</tag></item>
            <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
"/";</tag></item>
        </one-of>
        <one-of>
            <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></
item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</
tag></item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
        </one-of>
        </item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">I want to see bank statements from </item>
            <item repeat="0-1">Show me statements from</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>

```

```

<rule id="months">
    <tag>out.mon=""</tag>
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>
        <item>june<tag>out.mon+="06";</tag></item>
        <item>july<tag>out.mon+="07";</tag></item>
        <item>august<tag>out.mon+="08";</tag></item>
        <item>september<tag>out.mon+="09";</tag></item>
        <item>october<tag>out.mon+="10";</tag></item>
        <item>november<tag>out.mon+="11";</tag></item>
        <item>december<tag>out.mon+="12";</tag></item>
        <item>jan<tag>out.mon+="01";</tag></item>
        <item>feb<tag>out.mon+="02";</tag></item>
        <item>aug<tag>out.mon+="08";</tag></item>
        <item>sept<tag>out.mon+="09";</tag></item>
        <item>oct<tag>out.mon+="10";</tag></item>
        <item>nov<tag>out.mon+="11";</tag></item>
        <item>dec<tag>out.mon+="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <one-of>
        <item>zero<tag>out=0;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="thousands">
    <item>two thousand</item>
    <item repeat="0-1">and</item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></item>
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
rules.above_twenty;</tag></item>
</rule>

```

```

<rule id="above_twenty">
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>

```

## Transaction date

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
    Input: My last incorrect transaction date is july twenty three
    Output: 07/23

Scenario 2:
    Input: My last incorrect transaction date is july fifteen
    Output: 07/15

-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item repeat="1-10">
        <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></item>
    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
    <item><ruleref uri="#above_twenty"/><tag>out+= rules.above_twenty;</tag></item>
    </one-of>
    </item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My last incorrect transaction date is</item>
        <item repeat="0-1">It is</item>
    </one-of>
</rule>
<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

```

```

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.mon=""</tag>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>
        <item>june<tag>out.mon+="06";</tag></item>
        <item>july<tag>out.mon+="07";</tag></item>
        <item>august<tag>out.mon+="08";</tag></item>
        <item>september<tag>out.mon+="09";</tag></item>
        <item>october<tag>out.mon+="10";</tag></item>
        <item>november<tag>out.mon+="11";</tag></item>
        <item>december<tag>out.mon+="12";</tag></item>
        <item>jan<tag>out.mon+="01";</tag></item>
        <item>feb<tag>out.mon+="02";</tag></item>
        <item>aug<tag>out.mon+="08";</tag></item>
        <item>sept<tag>out.mon+="09";</tag></item>
        <item>oct<tag>out.mon+="10";</tag></item>
        <item>nov<tag>out.mon+="11";</tag></item>
        <item>dec<tag>out.mon+="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>first<tag>out=01;</tag></item>
        <item>second<tag>out=02;</tag></item>
        <item>third<tag>out=03;</tag></item>
        <item>fourth<tag>out=04;</tag></item>
        <item>fifth<tag>out=05;</tag></item>
        <item>sixth<tag>out=06;</tag></item>
        <item>seventh<tag>out=07;</tag></item>
        <item>eighth<tag>out=08;</tag></item>
        <item>ninth<tag>out=09;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>tenth<tag>out=10;</tag></item>

```

```

<item>eleven<tag>out=11;</tag></item>
<item>twelve<tag>out=12;</tag></item>
<item>thirteen<tag>out=13;</tag></item>
<item>fourteen<tag>out=14;</tag></item>
<item>fifteen<tag>out=15;</tag></item>
<item>sixteen<tag>out=16;</tag></item>
<item>seventeen<tag>out=17;</tag></item>
<item>eighteen<tag>out=18;</tag></item>
<item>nineteen<tag>out=19;</tag></item>
<item>tenth<tag>out=10;</tag></item>
<item>eleventh<tag>out=11;</tag></item>
<item>twelfth<tag>out=12;</tag></item>
<item>thirteenth<tag>out=13;</tag></item>
<item>fourteenth<tag>out=14;</tag></item>
<item>fifteenth<tag>out=15;</tag></item>
<item>sixteenth<tag>out=16;</tag></item>
<item>seventeenth<tag>out=17;</tag></item>
<item>eighteenth<tag>out=18;</tag></item>
<item>nineteenth<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>
```

## Transfer amount

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
    Input: I want to transfer twenty five Dollar
    Output: $25

Scenario 2:
    Input: transfer twenty five dollars
    Output: $25

-->

<rule id="main" scope="public">
    <tag>out="$"</tag>
    <one-of>
        <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</tag></item>

```

```

    <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</tag></
item>
    </one-of>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">I want to transfer</item>
        <item repeat="0-1">transfer</item>
        <item repeat="0-1">make a transfer for</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
        <item>three<tag>out.num+=3;</tag></item>
        <item>four<tag>out.num+=4;</tag></item>
        <item>five<tag>out.num+=5;</tag></item>
        <item>six<tag>out.num+=6;</tag></item>
        <item>seven<tag>out.num+=7;</tag></item>
        <item>eight<tag>out.num+=8;</tag></item>
        <item>nine<tag>out.num+=9;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
        <item>fifteen<tag>out.teen+=15;</tag></item>
        <item>sixteen<tag>out.teen+=16;</tag></item>
        <item>seventeen<tag>out.teen+=17;</tag></item>
        <item>eighteen<tag>out.teen+=18;</tag></item>
        <item>nineteen<tag>out.teen+=19;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>
```

```

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.tens = 0;</tag>
    <one-of>
        <item>twenty<tag>out.tens+=20;</tag></item>
        <item>thirty<tag>out.tens+=30;</tag></item>
        <item>forty<tag>out.tens+=40;</tag></item>
        <item>fifty<tag>out.tens+=50;</tag></item>
        <item>sixty<tag>out.tens+=60;</tag></item>
        <item>seventy<tag>out.tens+=70;</tag></item>
        <item>eighty<tag>out.tens+=80;</tag></item>
        <item>ninety<tag>out.tens+=90;</tag></item>
        <item>hundred<tag>out.tens+=100;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens += rules.digits.num;</tag></item>
</rule>

<rule id="currency">
    <one-of>
        <item repeat="0-1">dollars</item>
        <item repeat="0-1">Dollars</item>
        <item repeat="0-1">dollar</item>
        <item repeat="0-1">Dollar</item>
    </one-of>
</rule>

<rule id="sub_hundred">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.sh = 0;</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></item>
        <item><ruleref uri="#above_twenty"/><tag>out.sh += rules.above_twenty.tens;</tag>
            </item>
            <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></item>
    </one-of>
</rule>

<rule id="subThousands">
    <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
    hundred
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty.tens;</tag></item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</tag></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</tag></item>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>
</grammar>

```

## Social Security number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"

```

```

root="main"
mode="voice"
tag-format="semantics/1.0"

<rule id="main" scope="public">
    <tag>out=""</tag>
    <ruleref uri="#digits"/><tag>out += rules.digits.numbers;</tag>
</rule>

<rule id="digits">
    <tag>out.numbers=""</tag>
    <item repeat="1-12">
        <one-of>
            <item>0<tag>out.numbers+=0;</tag></item>
            <item>1<tag>out.numbers+=1;</tag></item>
            <item>2<tag>out.numbers+=2;</tag></item>
            <item>3<tag>out.numbers+=3;</tag></item>
            <item>4<tag>out.numbers+=4;</tag></item>
            <item>5<tag>out.numbers+=5;</tag></item>
            <item>6<tag>out.numbers+=6;</tag></item>
            <item>7<tag>out.numbers+=7;</tag></item>
            <item>8<tag>out.numbers+=8;</tag></item>
            <item>9<tag>out.numbers+=9;</tag></item>
            <item>zero<tag>out.numbers+=0;</tag></item>
            <item>one<tag>out.numbers+=1;</tag></item>
            <item>two<tag>out.numbers+=2;</tag></item>
            <item>three<tag>out.numbers+=3;</tag></item>
            <item>four<tag>out.numbers+=4;</tag></item>
            <item>five<tag>out.numbers+=5;</tag></item>
            <item>six<tag>out.numbers+=6;</tag></item>
            <item>seven<tag>out.numbers+=7;</tag></item>
            <item>eight<tag>out.numbers+=8;</tag></item>
            <item>nine<tag>out.numbers+=9;</tag></item>
            <item>dash</item>
        </one-of>
    </item>
</rule>
</grammar>
```

## Grammars for insurance (download)

The following grammars are supported for insurance domain: claim and policy numbers, driver's license and license plate numbers, expiration dates, start dates and renewal dates, claim and policy amounts.

### Claim ID

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="digits"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

    Grammar will support the following inputs:

    Scenario 1:
        Input: My claim number is One Five Four Two
        Output: 1542
-->
```

```

Scenario 2:
Input: Claim number One Five Four Four
Output: 1544

-->

<rule id="digits">
    <tag>out=""</tag>
    <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My claim number is</item>
        <item repeat="0-1">Claim number</item>
        <item repeat="0-1">This is for claim</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="singleDigit">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.digit=""</tag>
    <item repeat="1-10">
        <one-of>
            <item>0<tag>out.digit+=0;</tag></item>
            <item>zero<tag>out.digit+=0;</tag></item>
            <item>1<tag>out.digit+=1;</tag></item>
            <item>one<tag>out.digit+=1;</tag></item>
            <item>2<tag>out.digit+=2;</tag></item>
            <item>two<tag>out.digit+=2;</tag></item>
            <item>3<tag>out.digit+=3;</tag></item>
            <item>three<tag>out.digit+=3;</tag></item>
            <item>4<tag>out.digit+=4;</tag></item>
            <item>four<tag>out.digit+=4;</tag></item>
            <item>5<tag>out.digit+=5;</tag></item>
            <item>five<tag>out.digit+=5;</tag></item>
            <item>6<tag>out.digit+=6;</tag></item>
            <item>six<tag>out.digit+=6;</tag></item>
            <item>7<tag>out.digit+=7;</tag></item>
            <item>seven<tag>out.digit+=7;</tag></item>
            <item>8<tag>out.digit+=8;</tag></item>
            <item>eight<tag>out.digit+=8;</tag></item>
            <item>9<tag>out.digit+=9;</tag></item>
            <item>nine<tag>out.digit+=9;</tag></item>
        </one-of>
    </item>
</rule>
</grammar>

```

## Policy ID

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                    http://www.w3.org/TR/speech-grammar/grammar.xsd"
xml:lang="en-US" version="1.0"
root="main"
mode="voice"
tag-format="semantics/1.0">

<!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: My policy number is A B C 1 2 3 4
Output: ABC1234

Scenario 2:
Input: This is the policy number 1 2 3 4 A B C
Output: 1234ABC

Scenario 3:
Input: Hmm My policy number is 1 2 3 4 A B C 1
Output: 123ABC1
-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item><ruleref uri="#alphanumeric"/><tag>out += rules.alphanumeric.alphanum;</tag></item>
    <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out += rules.alphabets.letters;</tag></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.numbers;</tag></item>
    <item repeat="0-1"><ruleref uri="#thanks"/></item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My policy number is</item>
        <item repeat="0-1">This is the policy number</item>
        <item repeat="0-1">Policy number</item>
        <item repeat="0-1">Yes, It is</item>
        <item repeat="0-1">Yes It is</item>
        <item repeat="0-1">Yes It's</item>
        <item repeat="0-1">My policy Id is</item>
        <item repeat="0-1">This is the policy Id</item>
        <item repeat="0-1">Policy Id</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="thanks">
    <one-of>
        <item>Thanks</item>
        <item>I think</item>
    </one-of>
</rule>

```

```

<rule id="alphanumeric" scope="public">
    <tag>out.alphanum=""</tag>
    <item><ruleref uri="#alphabets"/><tag>out.alphanum += rules.alphabets.letters;</tag></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum += rules.digits.numbers</tag></item>
</rule>

<rule id="alphabets">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.letters=""</tag>
    <tag>out.firstOccurrence=""</tag>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurrence += rules.digits.numbers; out.letters += out.firstOccurrence;</tag></item>
    <item repeat="1-"
        <one-of>
            <item>A<tag>out.letters+='A';</tag></item>
            <item>B<tag>out.letters+='B';</tag></item>
            <item>C<tag>out.letters+='C';</tag></item>
            <item>D<tag>out.letters+='D';</tag></item>
            <item>E<tag>out.letters+='E';</tag></item>
            <item>F<tag>out.letters+='F';</tag></item>
            <item>G<tag>out.letters+='G';</tag></item>
            <item>H<tag>out.letters+='H';</tag></item>
            <item>I<tag>out.letters+='I';</tag></item>
            <item>J<tag>out.letters+='J';</tag></item>
            <item>K<tag>out.letters+='K';</tag></item>
            <item>L<tag>out.letters+='L';</tag></item>
            <item>M<tag>out.letters+='M';</tag></item>
            <item>N<tag>out.letters+='N';</tag></item>
            <item>O<tag>out.letters+='O';</tag></item>
            <item>P<tag>out.letters+='P';</tag></item>
            <item>Q<tag>out.letters+='Q';</tag></item>
            <item>R<tag>out.letters+='R';</tag></item>
            <item>S<tag>out.letters+='S';</tag></item>
            <item>T<tag>out.letters+='T';</tag></item>
            <item>U<tag>out.letters+='U';</tag></item>
            <item>V<tag>out.letters+='V';</tag></item>
            <item>W<tag>out.letters+='W';</tag></item>
            <item>X<tag>out.letters+='X';</tag></item>
            <item>Y<tag>out.letters+='Y';</tag></item>
            <item>Z<tag>out.letters+='Z';</tag></item>
        </one-of>
    </item>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.numbers=""</tag>
    <item repeat="1-10"
        <one-of>
            <item>0<tag>out.numbers+=0;</tag></item>
            <item>1<tag>out.numbers+=1;</tag></item>
            <item>2<tag>out.numbers+=2;</tag></item>
            <item>3<tag>out.numbers+=3;</tag></item>
            <item>4<tag>out.numbers+=4;</tag></item>
            <item>5<tag>out.numbers+=5;</tag></item>
            <item>6<tag>out.numbers+=6;</tag></item>
            <item>7<tag>out.numbers+=7;</tag></item>
            <item>8<tag>out.numbers+=8;</tag></item>
            <item>9<tag>out.numbers+=9;</tag></item>
        </one-of>
    </item>
</rule>
</grammar>
```

## Driver's license number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="digits"
          mode="voice"
          tag-format="semantics/1.0">

    <!-- Test Cases

    Grammar will support the following inputs:

        Scenario 1:
            Input: My drivers license number is One Five Four Two
            Output: 1542

        Scenario 2:
            Input: driver license number One Five Four Four
            Output: 1544

    -->

    <rule id="digits">
        <tag>out=""</tag>
        <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">My drivers license number is</item>
            <item repeat="0-1">My drivers license id is</item>
            <item repeat="0-1">Driver license number</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>

    <rule id="singleDigit">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.digit=""</tag>
        <item repeat="1-10">
            <one-of>
                <item>0<tag>out.digit+=0;</tag></item>
                <item>zero<tag>out.digit+=0;</tag></item>
                <item>1<tag>out.digit+=1;</tag></item>
                <item>one<tag>out.digit+=1;</tag></item>
                <item>2<tag>out.digit+=2;</tag></item>
                <item>two<tag>out.digit+=2;</tag></item>
                <item>3<tag>out.digit+=3;</tag></item>
                <item>three<tag>out.digit+=3;</tag></item>
                <item>4<tag>out.digit+=4;</tag></item>
                <item>four<tag>out.digit+=4;</tag></item>
                <item>5<tag>out.digit+=5;</tag></item>
            </one-of>
        </item>
    </rule>

```

```

<item>five<tag>out.digit+=5;</tag></item>
<item>6<tag>out.digit+=6;</tag></item>
<item>six<tag>out.digit+=5;</tag></item>
<item>7<tag>out.digit+=7;</tag></item>
<item>seven<tag>out.digit+=7;</tag></item>
<item>8<tag>out.digit+=8;</tag></item>
<item>eight<tag>out.digit+=8;</tag></item>
<item>9<tag>out.digit+=9;</tag></item>
<item>nine<tag>out.digit+=9;</tag></item>
</one-of>
</item>
</rule>
</grammar>

```

### License plate number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="main"
         mode="voice"
         tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: my license plate is A B C D 1 2
Output: ABCD12

Scenario 2:
Input: license plate number A B C 1 2 3 4
Output: ABC1234

Scenario 3:
Input: my plates say A F G K 9 8 7 6 Thanks
Output: AFGK9876
-->

<rule id="main" scope="public">
    <tag>out.licenseNum=""</tag>
    <item><ruleref uri="#alphabets"/><tag>out.licenseNum +=
rules.alphabets.letters;</tag></item>
    <item repeat="0-1"><ruleref uri="#thanks"/></item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">my license plate is</item>
        <item repeat="0-1">license plate number</item>
        <item repeat="0-1">my plates say</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
    </one-of>
</rule>

```

```

        <item>My</item>
    </one-of>
</rule>

<rule id="thanks">
    <one-of>
        <item>Thanks</item>
        <item>I think</item>
    </one-of>
</rule>

<rule id="alphabets">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.letters=""</tag>
    <tag>out.firstOccurence=""</tag>
    <item repeat="3-4">
        <one-of>
            <item>A<tag>out.letters+= 'A';</tag></item>
            <item>B<tag>out.letters+= 'B';</tag></item>
            <item>C<tag>out.letters+= 'C';</tag></item>
            <item>D<tag>out.letters+= 'D';</tag></item>
            <item>E<tag>out.letters+= 'E';</tag></item>
            <item>F<tag>out.letters+= 'F';</tag></item>
            <item>G<tag>out.letters+= 'G';</tag></item>
            <item>H<tag>out.letters+= 'H';</tag></item>
            <item>I<tag>out.letters+= 'I';</tag></item>
            <item>J<tag>out.letters+= 'J';</tag></item>
            <item>K<tag>out.letters+= 'K';</tag></item>
            <item>L<tag>out.letters+= 'L';</tag></item>
            <item>M<tag>out.letters+= 'M';</tag></item>
            <item>N<tag>out.letters+= 'N';</tag></item>
            <item>O<tag>out.letters+= 'O';</tag></item>
            <item>P<tag>out.letters+= 'P';</tag></item>
            <item>Q<tag>out.letters+= 'Q';</tag></item>
            <item>R<tag>out.letters+= 'R';</tag></item>
            <item>S<tag>out.letters+= 'S';</tag></item>
            <item>T<tag>out.letters+= 'T';</tag></item>
            <item>U<tag>out.letters+= 'U';</tag></item>
            <item>V<tag>out.letters+= 'V';</tag></item>
            <item>W<tag>out.letters+= 'W';</tag></item>
            <item>X<tag>out.letters+= 'X';</tag></item>
            <item>Y<tag>out.letters+= 'Y';</tag></item>
            <item>Z<tag>out.letters+= 'Z';</tag></item>
        </one-of>
    </item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence += rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.numbers=""</tag>
    <item repeat="2-4">
        <one-of>
            <item>0<tag>out.numbers+=0;</tag></item>
            <item>1<tag>out.numbers+=1;</tag></item>
            <item>2<tag>out.numbers+=2;</tag></item>
            <item>3<tag>out.numbers+=3;</tag></item>
            <item>4<tag>out.numbers+=4;</tag></item>
            <item>5<tag>out.numbers+=5;</tag></item>
            <item>6<tag>out.numbers+=6;</tag></item>
            <item>7<tag>out.numbers+=7;</tag></item>
            <item>8<tag>out.numbers+=8;</tag></item>
            <item>9<tag>out.numbers+=9;</tag></item>
        </one-of>
    </item>

```

```
</rule>
</grammar>
```

### Credit card expiration date

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="dateCardExpiration"
  mode="voice"
  tag-format="semantics/1.0">

  <rule id="dateCardExpiration" scope="public">
    <tag>out=""</tag>
    <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months;</tag></
item>
    <item repeat="1"><ruleref uri="#year"/><tag>out += " " + rules.year.yr;</tag></
item>
    <item repeat="0-1"><ruleref uri="#thanks"/></item>
  </rule>

  <!-- Test Cases

Grammar will support the following inputs:

  Scenario 1:
    Input: My card expiration date is july eleven
    Output: 07 2011

  Scenario 2:
    Input: My card expiration date is may twenty six
    Output: 05 2026

  -->

  <rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
      <item repeat="0-1">My card expiration date is </item>
    </one-of>
  </rule>

  <rule id="hesitation">
    <one-of>
      <item>Hmm</item>
      <item>Mmm</item>
      <item>My</item>
    </one-of>
  </rule>

  <rule id="thanks">
    <one-of>
      <item>Thanks</item>
      <item>I think</item>
    </one-of>
  </rule>

  <rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
      <item>january</item>
      <tag>out="01";</tag></item>
    </one-of>
  </rule>
```

```

<item>february<tag>out="02";</tag></item>
<item>march<tag>out="03";</tag></item>
<item>april<tag>out="04";</tag></item>
<item>may<tag>out="05";</tag></item>
<item>june<tag>out="06";</tag></item>
<item>july<tag>out="07";</tag></item>
<item>august<tag>out="08";</tag></item>
<item>september<tag>out="09";</tag></item>
<item>october<tag>out="10";</tag></item>
<item>november<tag>out="11";</tag></item>
<item>december<tag>out="12";</tag></item>
<item>jan<tag>out="01";</tag></item>
<item>feb<tag>out="02";</tag></item>
<item>aug<tag>out="08";</tag></item>
<item>sept<tag>out="09";</tag></item>
<item>oct<tag>out="10";</tag></item>
<item>nov<tag>out="11";</tag></item>
<item>dec<tag>out="12";</tag></item>
<item>1<tag>out="01";</tag></item>
<item>2<tag>out="02";</tag></item>
<item>3<tag>out="03";</tag></item>
<item>4<tag>out="04";</tag></item>
<item>5<tag>out="05";</tag></item>
<item>6<tag>out="06";</tag></item>
<item>7<tag>out="07";</tag></item>
<item>8<tag>out="08";</tag></item>
<item>9<tag>out="09";</tag></item>
<item>ten<tag>out="10";</tag></item>
<item>eleven<tag>out="11";</tag></item>
<item>twelve<tag>out="12";</tag></item>
</one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="year">
    <tag>out.yr="20"</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.yr += rules.teens;</tag></item>
        <item><ruleref uri="#above_twenty"/><tag>out.yr += rules.above_twenty;</tag></item>
    </one-of>
</rule>

```

```

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
        <item>10<tag>out=10;</tag></item>
        <item>11<tag>out=11;</tag></item>
        <item>12<tag>out=12;</tag></item>
        <item>13<tag>out=13;</tag></item>
        <item>14<tag>out=14;</tag></item>
        <item>15<tag>out=15;</tag></item>
        <item>16<tag>out=16;</tag></item>
        <item>17<tag>out=17;</tag></item>
        <item>18<tag>out=18;</tag></item>
        <item>19<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
        <item>20<tag>out=20;</tag></item>
        <item>30<tag>out=30;</tag></item>
        <item>40<tag>out=40;</tag></item>
        <item>50<tag>out=50;</tag></item>
        <item>60<tag>out=60;</tag></item>
        <item>70<tag>out=70;</tag></item>
        <item>80<tag>out=80;</tag></item>
        <item>90<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>
```

### Policy expiration date, day/month/year

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">
```

```

<!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: My policy expired on July Five Two Thousand and Eleven
Output: 07/5/11

Scenario 2:
Input: My policy will expire on July Sixteen Two Thousand and Eleven
Output: 07/16/11

Scenario 3:
Input: My policy expired on July Thirty Two Thousand and Eleven
Output: 07/30/11

-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item>
        <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ "/";</tag></item>
        <one-of>
            <item><ruleref uri="#digits"/><tag>out += rules.digits + "/";</tag></
item>
            <item><ruleref uri="#teens"/><tag>out += rules.teens+ "/";</tag></item>
            <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
"/";</tag></item>
            </one-of>
            <one-of>
                <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></
item>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
                <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</
tag></item>
                <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
                </one-of>
            </item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>
                <item repeat="0-1">My policy expired on</item>
                <item repeat="0-1">My policy will expire on</item>
            </one-of>
        </rule>

        <rule id="hesitation">
            <one-of>
                <item>Hmm</item>
                <item>Mmm</item>
                <item>My</item>
            </one-of>
        </rule>

        <rule id="months">
            <tag>out.mon=""</tag>
<item repeat="0-1"><ruleref uri="#text"/></item>
            <one-of>
                <item>january<tag>out.mon+="01";</tag></item>
                <item>february<tag>out.mon+="02";</tag></item>
                <item>march<tag>out.mon+="03";</tag></item>
                <item>april<tag>out.mon+="04";</tag></item>
            </one-of>
        </rule>
    </item>
</rule>

```

```

<item>may<tag>out.mon+="05";</tag></item>
<item>june<tag>out.mon+="06";</tag></item>
<item>july<tag>out.mon+="07";</tag></item>
<item>august<tag>out.mon+="08";</tag></item>
<item>september<tag>out.mon+="09";</tag></item>
<item>october<tag>out.mon+="10";</tag></item>
<item>november<tag>out.mon+="11";</tag></item>
<item>december<tag>out.mon+="12";</tag></item>
<item>jan<tag>out.mon+="01";</tag></item>
<item>feb<tag>out.mon+="02";</tag></item>
<item>aug<tag>out.mon+="08";</tag></item>
<item>sept<tag>out.mon+="09";</tag></item>
<item>oct<tag>out.mon+="10";</tag></item>
<item>nov<tag>out.mon+="11";</tag></item>
<item>dec<tag>out.mon+="12";</tag></item>
</one-of>
</rule>

<rule id="digits">
<one-of>
    <item>zero<tag>out=0;</tag></item>
    <item>one<tag>out=1;</tag></item>
    <item>two<tag>out=2;</tag></item>
    <item>three<tag>out=3;</tag></item>
    <item>four<tag>out=4;</tag></item>
    <item>five<tag>out=5;</tag></item>
    <item>six<tag>out=6;</tag></item>
    <item>seven<tag>out=7;</tag></item>
    <item>eight<tag>out=8;</tag></item>
    <item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="teens">
<one-of>
    <item>ten<tag>out=10;</tag></item>
    <item>eleven<tag>out=11;</tag></item>
    <item>twelve<tag>out=12;</tag></item>
    <item>thirteen<tag>out=13;</tag></item>
    <item>fourteen<tag>out=14;</tag></item>
    <item>fifteen<tag>out=15;</tag></item>
    <item>sixteen<tag>out=16;</tag></item>
    <item>seventeen<tag>out=17;</tag></item>
    <item>eighteen<tag>out=18;</tag></item>
    <item>nineteen<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="thousands">
    <item>two thousand</item>
    <item repeat="0-1">and</item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></item>
item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></item>
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
rules.above_twenty;</tag></item>
</rule>

<rule id="above_twenty">
<one-of>
    <item>twenty<tag>out=20;</tag></item>
    <item>thirty<tag>out=30;</tag></item>
</one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
item>
</rule>

```

```
</grammar>
```

### Policy renewal date, month/year

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="main"
          mode="voice"
          tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

    Scenario 1:
        Input: I renewed my policy on July Two Thousand and Eleven
        Output: 07/11

    Scenario 2:
        Input: My policy will renew on July Two Thousand and Eleven
        Output: 07/11

    -->

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item repeat="1-10">
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ "/";</tag></item>
            <one-of>
                <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></
item>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
                <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</
tag></item>
                <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
            </one-of>
        </item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">My policy will renew on</item>
            <item repeat="0-1">My policy was renewed on</item>
            <item repeat="0-1">Renew policy on</item>
            <item repeat="0-1">I renewed my policy on</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>
```

```

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.mon=""</tag>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>
        <item>june<tag>out.mon+="06";</tag></item>
        <item>july<tag>out.mon+="07";</tag></item>
        <item>august<tag>out.mon+="08";</tag></item>
        <item>september<tag>out.mon+="09";</tag></item>
        <item>october<tag>out.mon+="10";</tag></item>
        <item>november<tag>out.mon+="11";</tag></item>
        <item>december<tag>out.mon+="12";</tag></item>
        <item>jan<tag>out.mon+="01";</tag></item>
        <item>feb<tag>out.mon+="02";</tag></item>
        <item>aug<tag>out.mon+="08";</tag></item>
        <item>sept<tag>out.mon+="09";</tag></item>
        <item>oct<tag>out.mon+="10";</tag></item>
        <item>nov<tag>out.mon+="11";</tag></item>
        <item>dec<tag>out.mon+="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <one-of>
        <item>zero<tag>out=0;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="thousands">
    <item>two thousand<!--<tag>out=2000;</tag>--></item>
    <item repeat="0-1">and</item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></item>
item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></item>
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out = rules.above_twenty;</tag></item>
</rule>

```

```

<rule id="above_twenty">
  <one-of>
    <item>twenty<tag>out=20;</tag></item>
    <item>thirty<tag>out=30;</tag></item>
    <item>forty<tag>out=40;</tag></item>
    <item>fifty<tag>out=50;</tag></item>
    <item>sixty<tag>out=60;</tag></item>
    <item>seventy<tag>out=70;</tag></item>
    <item>eighty<tag>out=80;</tag></item>
    <item>ninety<tag>out=90;</tag></item>
  </one-of>
  <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>

```

### Policy start date

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <!-- Test Cases

Grammar will support the following inputs:

  Scenario 1:
    Input: I bought my policy on july twenty three
    Output: 07/23

  Scenario 2:
    Input: My policy started on july fifteen
    Output: 07/15

  -->

<rule id="main" scope="public">
  <tag>out=""</tag>
  <item repeat="1-10">
    <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></item>
  </item>
  <one-of>
    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
    <item><ruleref uri="#above_twenty"/><tag>out+= rules.above_twenty;</tag></item>
  </one-of>
</item>
</rule>

<rule id="text">
  <item repeat="0-1"><ruleref uri="#hesitation"/></item>
  <one-of>
    <item repeat="0-1">I bought my policy on</item>
    <item repeat="0-1">I bought policy on</item>
    <item repeat="0-1">My policy started on</item>
  </one-of>
</rule>

```

```

<rule id="hesitation">
  <one-of>
    <item>Hmm</item>
    <item>Mmm</item>
    <item>My</item>
  </one-of>
</rule>

<rule id="months">
  <item repeat="0-1"><ruleref uri="#text"/></item>
  <tag>out.mon=""</tag>
  <one-of>
    <item>january<tag>out.mon+="01";</tag></item>
    <item>february<tag>out.mon+="02";</tag></item>
    <item>march<tag>out.mon+="03";</tag></item>
    <item>april<tag>out.mon+="04";</tag></item>
    <item>may<tag>out.mon+="05";</tag></item>
    <item>june<tag>out.mon+="06";</tag></item>
    <item>july<tag>out.mon+="07";</tag></item>
    <item>august<tag>out.mon+="08";</tag></item>
    <item>september<tag>out.mon+="09";</tag></item>
    <item>october<tag>out.mon+="10";</tag></item>
    <item>november<tag>out.mon+="11";</tag></item>
    <item>december<tag>out.mon+="12";</tag></item>
    <item>jan<tag>out.mon+="01";</tag></item>
    <item>feb<tag>out.mon+="02";</tag></item>
    <item>aug<tag>out.mon+="08";</tag></item>
    <item>sept<tag>out.mon+="09";</tag></item>
    <item>oct<tag>out.mon+="10";</tag></item>
    <item>nov<tag>out.mon+="11";</tag></item>
    <item>dec<tag>out.mon+="12";</tag></item>
  </one-of>
</rule>

<rule id="digits">
  <item repeat="0-1"><ruleref uri="#text"/></item>
  <one-of>
    <item>0<tag>out=0;</tag></item>
    <item>1<tag>out=1;</tag></item>
    <item>2<tag>out=2;</tag></item>
    <item>3<tag>out=3;</tag></item>
    <item>4<tag>out=4;</tag></item>
    <item>5<tag>out=5;</tag></item>
    <item>6<tag>out=6;</tag></item>
    <item>7<tag>out=7;</tag></item>
    <item>8<tag>out=8;</tag></item>
    <item>9<tag>out=9;</tag></item>
    <item>first<tag>out=01;</tag></item>
    <item>second<tag>out=02;</tag></item>
    <item>third<tag>out=03;</tag></item>
    <item>fourth<tag>out=04;</tag></item>
    <item>fifth<tag>out=05;</tag></item>
    <item>sixth<tag>out=06;</tag></item>
    <item>seventh<tag>out=07;</tag></item>
    <item>eighth<tag>out=08;</tag></item>
    <item>ninth<tag>out=09;</tag></item>
    <item>one<tag>out=1;</tag></item>
    <item>two<tag>out=2;</tag></item>
    <item>three<tag>out=3;</tag></item>
    <item>four<tag>out=4;</tag></item>
    <item>five<tag>out=5;</tag></item>
    <item>six<tag>out=6;</tag></item>
    <item>seven<tag>out=7;</tag></item>
    <item>eight<tag>out=8;</tag></item>
    <item>nine<tag>out=9;</tag></item>
  </one-of>
</rule>

```

```

        </one-of>
    </rule>

    <rule id="teens">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>ten<tag>out=10;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
            <item>eleven<tag>out=11;</tag></item>
            <item>twelve<tag>out=12;</tag></item>
            <item>thirteen<tag>out=13;</tag></item>
            <item>fourteen<tag>out=14;</tag></item>
            <item>fifteen<tag>out=15;</tag></item>
            <item>sixteen<tag>out=16;</tag></item>
            <item>seventeen<tag>out=17;</tag></item>
            <item>eighteen<tag>out=18;</tag></item>
            <item>nineteen<tag>out=19;</tag></item>
            <item>tenth<tag>out=10;</tag></item>
            <item>eleventh<tag>out=11;</tag></item>
            <item>twelveth<tag>out=12;</tag></item>
            <item>thirteenth<tag>out=13;</tag></item>
            <item>fourteenth<tag>out=14;</tag></item>
            <item>fifteenth<tag>out=15;</tag></item>
            <item>sixteenth<tag>out=16;</tag></item>
            <item>seventeenth<tag>out=17;</tag></item>
            <item>eighteenth<tag>out=18;</tag></item>
            <item>nineteenth<tag>out=19;</tag></item>
        </one-of>
    </rule>

    <rule id="above_twenty">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <one-of>
            <item>twenty<tag>out=20;</tag></item>
            <item>thirty<tag>out=30;</tag></item>
        </one-of>
        <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
    </rule>
</grammar>
```

## Claim amount

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: I want to make a claim of one hundred ten dollars
Output: $110

Scenario 2:
Input: Requesting claim of Two hundred dollars
Output: $200

```

```

-->

<rule id="main" scope="public">
    <tag>out="$"</tag>
    <one-of>
        <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</tag></item>
        <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#thanks"/></item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">I want to place a claim for</item>
        <item repeat="0-1">I want to make a claim of</item>
        <item repeat="0-1">I assess damage of</item>
        <item repeat="0-1">Requesting claim of</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="thanks">
    <one-of>
        <item>Thanks</item>
        <item>I think</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
        <item>three<tag>out.num+=3;</tag></item>
        <item>four<tag>out.num+=4;</tag></item>
        <item>five<tag>out.num+=5;</tag></item>
        <item>six<tag>out.num+=6;</tag></item>
        <item>seven<tag>out.num+=7;</tag></item>
        <item>eight<tag>out.num+=8;</tag></item>
        <item>nine<tag>out.num+=9;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

```

```

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
        <item>fifteen<tag>out.teen+=15;</tag></item>
        <item>sixteen<tag>out.teen+=16;</tag></item>
        <item>seventeen<tag>out.teen+=17;</tag></item>
        <item>eighteen<tag>out.teen+=18;</tag></item>
        <item>nineteen<tag>out.teen+=19;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.tens = 0;</tag>
    <one-of>
        <item>twenty<tag>out.tens+=20;</tag></item>
        <item>thirty<tag>out.tens+=30;</tag></item>
        <item>forty<tag>out.tens+=40;</tag></item>
        <item>fifty<tag>out.tens+=50;</tag></item>
        <item>sixty<tag>out.tens+=60;</tag></item>
        <item>seventy<tag>out.tens+=70;</tag></item>
        <item>eighty<tag>out.tens+=80;</tag></item>
        <item>ninety<tag>out.tens+=90;</tag></item>
        <item>hundred<tag>out.tens+=100;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens += rules.digits.num;</tag></item>
</rule>

<rule id="currency">
    <one-of>
        <item repeat="0-1">dollars</item>
        <item repeat="0-1">Dollars</item>
        <item repeat="0-1">dollar</item>
        <item repeat="0-1">Dollar</item>
    </one-of>
</rule>

<rule id="sub_hundred">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.sh = 0;</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></item>
        <item>
            <ruleref uri="#above_twenty"/><tag>out.sh += rules.above_twenty.tens;</tag>
        </item>
        <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></item>
    </one-of>
</rule>

<rule id="subThousands">
    <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
    hundred
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
    rules.above_twenty.tens;</tag></item>

```

```

        <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</tag></
item>
      <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
      <item repeat="0-1"><ruleref uri="#currency"/></item>
    </rule>
</grammar>

```

## Premium amount

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <!-- Test Cases

Grammar will support the following inputs:

  Premium amounts
  Scenario 1:
    Input: The premium for one hundred ten dollars
    Output: $110

  Scenario 2:
    Input: RPremium amount of Two hundred dollars
    Output: $200

-->

<rule id="main" scope="public">
  <tag>out="$"</tag>
  <one-of>
    <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</
tag></item>
    <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</tag></
item>
  </one-of>
  <item repeat="0-1"><ruleref uri="#thanks"/></item>
</rule>

<rule id="text">
  <item repeat="0-1"><ruleref uri="#hesitation"/></item>
  <one-of>
    <item repeat="0-1">A premium of</item>
    <item repeat="0-1">Premium amount of</item>
    <item repeat="0-1">The premium for</item>
    <item repeat="0-1">Insurance premium for</item>
  </one-of>
</rule>

<rule id="hesitation">
  <one-of>
    <item>Hmm</item>
    <item>Mmm</item>
    <item>My</item>
  </one-of>
</rule>

```

```

<rule id="thanks">
    <one-of>
        <item>Thanks</item>
        <item>I think</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
        <item>three<tag>out.num+=3;</tag></item>
        <item>four<tag>out.num+=4;</tag></item>
        <item>five<tag>out.num+=5;</tag></item>
        <item>six<tag>out.num+=6;</tag></item>
        <item>seven<tag>out.num+=7;</tag></item>
        <item>eight<tag>out.num+=8;</tag></item>
        <item>nine<tag>out.num+=9;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
        <item>fifteen<tag>out.teen+=15;</tag></item>
        <item>sixteen<tag>out.teen+=16;</tag></item>
        <item>seventeen<tag>out.teen+=17;</tag></item>
        <item>eighteen<tag>out.teen+=18;</tag></item>
        <item>nineteen<tag>out.teen+=19;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.tens = 0;</tag>
    <one-of>
        <item>twenty<tag>out.tens+=20;</tag></item>
        <item>thirty<tag>out.tens+=30;</tag></item>
        <item>forty<tag>out.tens+=40;</tag></item>
        <item>fifty<tag>out.tens+=50;</tag></item>
        <item>sixty<tag>out.tens+=60;</tag></item>
        <item>seventy<tag>out.tens+=70;</tag></item>
        <item>eighty<tag>out.tens+=80;</tag></item>
        <item>ninety<tag>out.tens+=90;</tag></item>
        <item>hundred<tag>out.tens+=100;</tag></item>
    </one-of>

```

```

        <item repeat="0-1"><ruleref uri="#currency"/></item>
        <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens += rules.digits.num;</
tag></item>
        </rule>

        <rule id="currency">
            <one-of>
                <item repeat="0-1">dollars</item>
                <item repeat="0-1">Dollars</item>
                <item repeat="0-1">dollar</item>
                <item repeat="0-1">Dollar</item>
            </one-of>
        </rule>

        <rule id="sub_hundred">
            <item repeat="0-1"><ruleref uri="#text"/></item>
            <tag>out.sh = 0;</tag>
            <one-of>
                <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></item>
                <item>
                    <ruleref uri="#above_twenty"/><tag>out.sh += rules.above_twenty.tens;</
tag>
                </item>
                <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></item>
            </one-of>
        </rule>

        <rule id="subThousands">
            <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
            hundred
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty.tens;</tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</tag></
item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</
tag></item>
            <item repeat="0-1"><ruleref uri="#currency"/></item>
        </rule>
    </grammar>

```

## Policy quantity

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

    Grammar will support the following inputs:

    Scenario 1:
        Input: The number is one
        Output: 1

    Scenario 2:
        Input: I want policy for ten
        Output: 10

```

```

-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <one-of>
        <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></
item>
        <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
        <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
rules.above_twenty;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#thanks"/></item>
</rule>

<rule id="text">
    <one-of>
        <item repeat="0-1">I want policy for</item>
        <item repeat="0-1">I want to order policy for</item>
        <item repeat="0-1">The number is</item>
    </one-of>
</rule>

<rule id="thanks">
    <one-of>
        <item>Thanks</item>
        <item>I think</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
    </one-of>
</rule>

```

```

<item>eighteen<tag>out=18;</tag></item>
<item>nineteen<tag>out=19;</tag></item>
<item>10<tag>out=10;</tag></item>
<item>11<tag>out=11;</tag></item>
<item>12<tag>out=12;</tag></item>
<item>13<tag>out=13;</tag></item>
<item>14<tag>out=14;</tag></item>
<item>15<tag>out=15;</tag></item>
<item>16<tag>out=16;</tag></item>
<item>17<tag>out=17;</tag></item>
<item>18<tag>out=18;</tag></item>
<item>19<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
        <item>20<tag>out=20;</tag></item>
        <item>30<tag>out=30;</tag></item>
        <item>40<tag>out=40;</tag></item>
        <item>50<tag>out=50;</tag></item>
        <item>60<tag>out=60;</tag></item>
        <item>70<tag>out=70;</tag></item>
        <item>80<tag>out=80;</tag></item>
        <item>90<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>
```

## Grammars for telecom (download)

The following grammars are supported for telecom: Phone number, serial number, SIM number, US Zip code, credit card expiration date, plan start, renewal and expiration dates, service start date, equipment quantity and bill amount.

### Phone number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="digits"
         mode="voice"
         tag-format="semantics/1.0">

    <!-- Test Cases

    Grammar will support 10-12 digits number and here are couple of examples of valid
    inputs: -->
```

```

Scenario 1:
    Input: Mmm My phone number is two zero one two five two six seven eight
five
    Output: 2012526785

Scenario 2:
    Input: My phone number is two zero one two five two six seven eight five
Output: 2012526785

-->

<rule id="digits">
    <tag>out=""</tag>
    <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My phone number is</item>
        <item repeat="0-1">Phone number is</item>
        <item repeat="0-1">It is</item>
        <item repeat="0-1">Yes, it's</item>
        <item repeat="0-1">Yes, it is</item>
        <item repeat="0-1">Yes it is</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="singleDigit">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.digit=""</tag>
    <item repeat="10-12">
        <one-of>
            <item>0<tag>out.digit+=0;</tag></item>
            <item>zero<tag>out.digit+=0;</tag></item>
            <item>1<tag>out.digit+=1;</tag></item>
            <item>one<tag>out.digit+=1;</tag></item>
            <item>2<tag>out.digit+=2;</tag></item>
            <item>two<tag>out.digit+=2;</tag></item>
            <item>3<tag>out.digit+=3;</tag></item>
            <item>three<tag>out.digit+=3;</tag></item>
            <item>4<tag>out.digit+=4;</tag></item>
            <item>four<tag>out.digit+=4;</tag></item>
            <item>5<tag>out.digit+=5;</tag></item>
            <item>five<tag>out.digit+=5;</tag></item>
            <item>6<tag>out.digit+=6;</tag></item>
            <item>six<tag>out.digit+=5;</tag></item>
            <item>7<tag>out.digit+=7;</tag></item>
            <item>seven<tag>out.digit+=7;</tag></item>
            <item>8<tag>out.digit+=8;</tag></item>
            <item>eight<tag>out.digit+=8;</tag></item>
            <item>9<tag>out.digit+=9;</tag></item>
            <item>nine<tag>out.digit+=9;</tag></item>
        </one-of>
    </item>
</rule>

```

```
</grammar>
```

## Serial number

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="digits"
          mode="voice"
          tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

    Scenario 1:
        Input: My serial number is 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6
        Output: 123456789123456

    Scenario 2:
        Input: Device Serial number 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6
        Output: 123456789123456

    -->

    <rule id="digits">
        <tag>out=""</tag>
        <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">My serial number is</item>
            <item repeat="0-1">Device Serial number</item>
            <item repeat="0-1">The number is</item>
            <item repeat="0-1">The IMEI number is</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>

    <rule id="singleDigit">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.digit=""</tag>
        <item repeat="15">
            <one-of>
                <item>0<tag>out.digit+=0;</tag></item>
                <item>zero<tag>out.digit+=0;</tag></item>
                <item>1<tag>out.digit+=1;</tag></item>
                <item>one<tag>out.digit+=1;</tag></item>
                <item>2<tag>out.digit+=2;</tag></item>
                <item>two<tag>out.digit+=2;</tag></item>
                <item>3<tag>out.digit+=3;</tag></item>
```

```

<item>three<tag>out.digit+=3;</tag></item>
<item>4<tag>out.digit+=4;</tag></item>
<item>four<tag>out.digit+=4;</tag></item>
<item>5<tag>out.digit+=5;</tag></item>
<item>five<tag>out.digit+=5;</tag></item>
<item>6<tag>out.digit+=6;</tag></item>
<item>six<tag>out.digit+=6;</tag></item>
<item>7<tag>out.digit+=7;</tag></item>
<item>seven<tag>out.digit+=7;</tag></item>
<item>8<tag>out.digit+=8;</tag></item>
<item>eight<tag>out.digit+=8;</tag></item>
<item>9<tag>out.digit+=9;</tag></item>
<item>nine<tag>out.digit+=9;</tag></item>
</one-of>
</item>
</rule>
</grammar>

```

## SIM number

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <!-- Test Cases

  Grammar will support the following inputs:

  Scenario 1:
    Input: My SIM number is A B C 1 2 3 4
    Output: ABC1234

  Scenario 2:
    Input: My SIM number is 1 2 3 4 A B C
    Output: 1234ABC

  Scenario 3:
    Input: My SIM number is 1 2 3 4 A B C 1
    Output: 123ABC1
  -->

  <rule id="main" scope="public">
    <tag>out=""</tag>
    <item><ruleref uri="#alphanumeric"/><tag>out += rules.alphanumeric.alphanum;</tag></item>
    <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out +=
      rules.alphabets.letters;</tag></item>
      <item repeat="0-1"><ruleref uri="#digits"/><tag>out +=
        rules.digits.numbers;</tag></item>
    </rule>

    <rule id="text">
      <item repeat="0-1"><ruleref uri="#hesitation"/></item>
      <one-of>
        <item repeat="0-1">My SIM number is</item>
        <item repeat="0-1">SIM number is</item>
      </one-of>
    </rule>
  
```

```

</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="alphanumeric" scope="public">
    <tag>out.alphanum=""</tag>
    <item><ruleref uri="#alphabets"/><tag>out.alphanum +=  
rules.alphabets.letters;</tag></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum +=  
rules.digits.numbers</tag></item>
</rule>

<rule id="alphabets">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.letters=""</tag>
    <tag>out.firstOccurence=""</tag>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurence +=  
rules.digits.numbers; out.letters += out.firstOccurence;</tag></item>
    <item repeat="1->
        <one-of>
            <item>A<tag>out.letters+='A';</tag></item>
            <item>B<tag>out.letters+='B';</tag></item>
            <item>C<tag>out.letters+='C';</tag></item>
            <item>D<tag>out.letters+='D';</tag></item>
            <item>E<tag>out.letters+='E';</tag></item>
            <item>F<tag>out.letters+='F';</tag></item>
            <item>G<tag>out.letters+='G';</tag></item>
            <item>H<tag>out.letters+='H';</tag></item>
            <item>I<tag>out.letters+='I';</tag></item>
            <item>J<tag>out.letters+='J';</tag></item>
            <item>K<tag>out.letters+='K';</tag></item>
            <item>L<tag>out.letters+='L';</tag></item>
            <item>M<tag>out.letters+='M';</tag></item>
            <item>N<tag>out.letters+='N';</tag></item>
            <item>O<tag>out.letters+='O';</tag></item>
            <item>P<tag>out.letters+='P';</tag></item>
            <item>Q<tag>out.letters+='Q';</tag></item>
            <item>R<tag>out.letters+='R';</tag></item>
            <item>S<tag>out.letters+='S';</tag></item>
            <item>T<tag>out.letters+='T';</tag></item>
            <item>U<tag>out.letters+='U';</tag></item>
            <item>V<tag>out.letters+='V';</tag></item>
            <item>W<tag>out.letters+='W';</tag></item>
            <item>X<tag>out.letters+='X';</tag></item>
            <item>Y<tag>out.letters+='Y';</tag></item>
            <item>Z<tag>out.letters+='Z';</tag></item>
        </one-of>
    </item>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.numbers=""</tag>
    <item repeat="1-10">
        <one-of>
            <item>0<tag>out.numbers+=0;</tag></item>
            <item>1<tag>out.numbers+=1;</tag></item>
            <item>2<tag>out.numbers+=2;</tag></item>
            <item>3<tag>out.numbers+=3;</tag></item>
            <item>4<tag>out.numbers+=4;</tag></item>

```

```

<item>5<tag>out.numbers+=5;</tag></item>
<item>6<tag>out.numbers+=6;</tag></item>
<item>7<tag>out.numbers+=7;</tag></item>
<item>8<tag>out.numbers+=8;</tag></item>
<item>9<tag>out.numbers+=9;</tag></item>
</one-of>
</item>
</rule>
</grammar>

```

### US Zip code

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="digits"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

    Grammar will support 5 digits code and here are couple of examples of valid
    inputs:

    Scenario 1:
        Input: Mmmm My zipcode is umm One Oh Nine Eight Seven
        Output: 10987

    Scenario 2:
        Input: My zipcode is One Oh Nine Eight Seven
        Output: 10987

    -->

    <rule id="digits">
        <tag>out=""</tag>
        <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">My zipcode is</item>
            <item repeat="0-1">Zipcode is</item>
            <item repeat="0-1">It is</item>
        </one-of>
    </rule>

    <rule id="hesitation">
        <one-of>
            <item>Hmm</item>
            <item>Mmm</item>
            <item>My</item>
        </one-of>
    </rule>

    <rule id="singleDigit">
        <item repeat="0-1"><ruleref uri="#text"/></item>
        <tag>out.digit=""</tag>
        <item repeat="5">

```

```

<one-of>
    <item>0<tag>out.digit+=0;</tag></item>
    <item>zero<tag>out.digit+=0;</tag></item>
    <item>Oh<tag>out.digit+=0;</tag></item>
    <item>1<tag>out.digit+=1;</tag></item>
    <item>one<tag>out.digit+=1;</tag></item>
    <item>2<tag>out.digit+=2;</tag></item>
    <item>two<tag>out.digit+=2;</tag></item>
    <item>3<tag>out.digit+=3;</tag></item>
    <item>three<tag>out.digit+=3;</tag></item>
    <item>4<tag>out.digit+=4;</tag></item>
    <item>four<tag>out.digit+=4;</tag></item>
    <item>5<tag>out.digit+=5;</tag></item>
    <item>five<tag>out.digit+=5;</tag></item>
    <item>6<tag>out.digit+=6;</tag></item>
    <item>six<tag>out.digit+=5;</tag></item>
    <item>7<tag>out.digit+=7;</tag></item>
    <item>seven<tag>out.digit+=7;</tag></item>
    <item>8<tag>out.digit+=8;</tag></item>
    <item>eight<tag>out.digit+=8;</tag></item>
    <item>9<tag>out.digit+=9;</tag></item>
    <item>nine<tag>out.digit+=9;</tag></item>
</one-of>
</item>
</rule>
</grammar>

```

## Credit card expiration date

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                           http://www.w3.org/TR/speech-grammar/grammar.xsd"
         xml:lang="en-US" version="1.0"
         root="dateCardExpiration"
         mode="voice"
         tag-format="semantics/1.0">

    <rule id="dateCardExpiration" scope="public">
        <tag>out=""</tag>
        <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months;</tag></item>
        <item repeat="1"><ruleref uri="#year"/><tag>out += " " + rules.year.yr;</tag></item>
    </rule>

    <!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
Input: My card expiration date is july eleven
Output: 07 2011

Scenario 2:
Input: My card expiration date is may twenty six
Output: 05 2026

-->

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
<one-of>

```

```

        <item repeat="0-1">My card expiration date is </item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>january<tag>out="01";</tag></item>
        <item>february<tag>out="02";</tag></item>
        <item>march<tag>out="03";</tag></item>
        <item>april<tag>out="04";</tag></item>
        <item>may<tag>out="05";</tag></item>
        <item>june<tag>out="06";</tag></item>
        <item>july<tag>out="07";</tag></item>
        <item>august<tag>out="08";</tag></item>
        <item>september<tag>out="09";</tag></item>
        <item>october<tag>out="10";</tag></item>
        <item>november<tag>out="11";</tag></item>
        <item>december<tag>out="12";</tag></item>
        <item>jan<tag>out="01";</tag></item>
        <item>feb<tag>out="02";</tag></item>
        <item>aug<tag>out="08";</tag></item>
        <item>sept<tag>out="09";</tag></item>
        <item>oct<tag>out="10";</tag></item>
        <item>nov<tag>out="11";</tag></item>
        <item>dec<tag>out="12";</tag></item>
        <item>1<tag>out="01";</tag></item>
        <item>2<tag>out="02";</tag></item>
        <item>3<tag>out="03";</tag></item>
        <item>4<tag>out="04";</tag></item>
        <item>5<tag>out="05";</tag></item>
        <item>6<tag>out="06";</tag></item>
        <item>7<tag>out="07";</tag></item>
        <item>8<tag>out="08";</tag></item>
        <item>9<tag>out="09";</tag></item>
        <item>ten<tag>out="10";</tag></item>
        <item>eleven<tag>out="11";</tag></item>
        <item>twelve<tag>out="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
    </one-of>
</rule>

```

```

<item>five<tag>out=5;</tag></item>
<item>six<tag>out=6;</tag></item>
<item>seven<tag>out=7;</tag></item>
<item>eight<tag>out=8;</tag></item>
<item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="year">
<tag>out.yr="20"</tag>
<one-of>
    <item><ruleref uri="#teens"/><tag>out.yr += rules.teens;</tag></item>
    <item><ruleref uri="#above_twenty"/><tag>out.yr += rules.above_twenty;</tag></item>
</one-of>
</rule>

<rule id="teens">
<item repeat="0-1"><ruleref uri="#text"/></item>
<one-of>
    <item>ten<tag>out=10;</tag></item>
    <item>eleven<tag>out=11;</tag></item>
    <item>twelve<tag>out=12;</tag></item>
    <item>thirteen<tag>out=13;</tag></item>
    <item>fourteen<tag>out=14;</tag></item>
    <item>fifteen<tag>out=15;</tag></item>
    <item>sixteen<tag>out=16;</tag></item>
    <item>seventeen<tag>out=17;</tag></item>
    <item>eighteen<tag>out=18;</tag></item>
    <item>nineteen<tag>out=19;</tag></item>
    <item>10<tag>out=10;</tag></item>
    <item>11<tag>out=11;</tag></item>
    <item>12<tag>out=12;</tag></item>
    <item>13<tag>out=13;</tag></item>
    <item>14<tag>out=14;</tag></item>
    <item>15<tag>out=15;</tag></item>
    <item>16<tag>out=16;</tag></item>
    <item>17<tag>out=17;</tag></item>
    <item>18<tag>out=18;</tag></item>
    <item>19<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
<item repeat="0-1"><ruleref uri="#text"/></item>
<one-of>
    <item>twenty<tag>out=20;</tag></item>
    <item>thirty<tag>out=30;</tag></item>
    <item>forty<tag>out=40;</tag></item>
    <item>fifty<tag>out=50;</tag></item>
    <item>sixty<tag>out=60;</tag></item>
    <item>seventy<tag>out=70;</tag></item>
    <item>eighty<tag>out=80;</tag></item>
    <item>ninety<tag>out=90;</tag></item>
    <item>20<tag>out=20;</tag></item>
    <item>30<tag>out=30;</tag></item>
    <item>40<tag>out=40;</tag></item>
    <item>50<tag>out=50;</tag></item>
    <item>60<tag>out=60;</tag></item>
    <item>70<tag>out=70;</tag></item>
    <item>80<tag>out=80;</tag></item>
    <item>90<tag>out=90;</tag></item>
</one-of>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>

```

```
</grammar>
```

### Plan expiration date, day/month/year

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="main"
          mode="voice"
          tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
    Input: My plan expires on July Five Two Thousand and Eleven
    Output: 07/5/11

Scenario 2:
    Input: My plan will expire on July Sixteen Two Thousand and Eleven
    Output: 07/16/11

Scenario 3:
    Input: My plan will expire on July Thirty Two Thousand and Eleven
    Output: 07/30/11
-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item>
        <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ "/";</tag></item>
        <one-of>
            <item><ruleref uri="#digits"/><tag>out += rules.digits + "/";</tag></
item>
            <item><ruleref uri="#teens"/><tag>out += rules.teens+ "/";</tag></item>
            <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+
"/";</tag></item>
        </one-of>
        <one-of>
            <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></
item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
            <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</
tag></item>
            <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
        </one-of>
        </item>
    </rule>

    <rule id="text">
        <item repeat="0-1"><ruleref uri="#hesitation"/></item>
        <one-of>
            <item repeat="0-1">My plan expires on</item>
            <item repeat="0-1">My plan expired on</item>
            <item repeat="0-1">My plan will expire on</item>
        </one-of>
    </rule>

```

```

<rule id="hesitation">
  <one-of>
    <item>Hmm</item>
    <item>Mmm</item>
    <item>My</item>
  </one-of>
</rule>

<rule id="months">
  <tag>out.mon=""</tag>
<item repeat="0-1"><ruleref uri="#text"/></item>

  <one-of>
    <item>january<tag>out.mon+="01";</tag></item>
    <item>february<tag>out.mon+="02";</tag></item>
    <item>march<tag>out.mon+="03";</tag></item>
    <item>april<tag>out.mon+="04";</tag></item>
    <item>may<tag>out.mon+="05";</tag></item>
    <item>june<tag>out.mon+="06";</tag></item>
    <item>july<tag>out.mon+="07";</tag></item>
    <item>august<tag>out.mon+="08";</tag></item>
    <item>september<tag>out.mon+="09";</tag></item>
    <item>october<tag>out.mon+="10";</tag></item>
    <item>november<tag>out.mon+="11";</tag></item>
    <item>december<tag>out.mon+="12";</tag></item>
    <item>jan<tag>out.mon+="01";</tag></item>
    <item>feb<tag>out.mon+="02";</tag></item>
    <item>aug<tag>out.mon+="08";</tag></item>
    <item>sept<tag>out.mon+="09";</tag></item>
    <item>oct<tag>out.mon+="10";</tag></item>
    <item>nov<tag>out.mon+="11";</tag></item>
    <item>dec<tag>out.mon+="12";</tag></item>
  </one-of>
</rule>

<rule id="digits">
  <one-of>
    <item>zero<tag>out=0;</tag></item>
    <item>one<tag>out=1;</tag></item>
    <item>two<tag>out=2;</tag></item>
    <item>three<tag>out=3;</tag></item>
    <item>four<tag>out=4;</tag></item>
    <item>five<tag>out=5;</tag></item>
    <item>six<tag>out=6;</tag></item>
    <item>seven<tag>out=7;</tag></item>
    <item>eight<tag>out=8;</tag></item>
    <item>nine<tag>out=9;</tag></item>
  </one-of>
</rule>

<rule id="teens">
  <one-of>
    <item>ten<tag>out=10;</tag></item>
    <item>eleven<tag>out=11;</tag></item>
    <item>twelve<tag>out=12;</tag></item>
    <item>thirteen<tag>out=13;</tag></item>
    <item>fourteen<tag>out=14;</tag></item>
    <item>fifteen<tag>out=15;</tag></item>
    <item>sixteen<tag>out=16;</tag></item>
    <item>seventeen<tag>out=17;</tag></item>
    <item>eighteen<tag>out=18;</tag></item>
    <item>nineteen<tag>out=19;</tag></item>
  </one-of>
</rule>

```

```

<rule id="thousands">
    <item>two thousand</item>
    <item repeat="0-1">and</item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></item>
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
rules.above_twenty;</tag></item>
</rule>

<rule id="above_twenty">
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></
item>
</rule>
</grammar>

```

### Plan renewal date, month/year

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

    Scenario 1:
        Input: My plan will renew on July Two Thousand and Eleven
        Output: 07/11

    Scenario 2:
        Input: Renew plan on July Two Thousand and Eleven
        Output: 07/11

    -->

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item repeat="1-10">
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ "/";</tag></item>
            <one-of>
                <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></
item>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
                <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</
tag></item>
                <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
            </one-of>
        </item>
    </rule>

```

```

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My plan will renew on</item>
        <item repeat="0-1">My plan was renewed on</item>
        <item repeat="0-1">Renew plan on</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.mon=""</tag>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>
        <item>june<tag>out.mon+="06";</tag></item>
        <item>july<tag>out.mon+="07";</tag></item>
        <item>august<tag>out.mon+="08";</tag></item>
        <item>september<tag>out.mon+="09";</tag></item>
        <item>october<tag>out.mon+="10";</tag></item>
        <item>november<tag>out.mon+="11";</tag></item>
        <item>december<tag>out.mon+="12";</tag></item>
        <item>jan<tag>out.mon+="01";</tag></item>
        <item>feb<tag>out.mon+="02";</tag></item>
        <item>aug<tag>out.mon+="08";</tag></item>
        <item>sept<tag>out.mon+="09";</tag></item>
        <item>oct<tag>out.mon+="10";</tag></item>
        <item>nov<tag>out.mon+="11";</tag></item>
        <item>dec<tag>out.mon+="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <one-of>
        <item>zero<tag>out=0;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>

```

```

<item>sixteen<tag>out=16;</tag></item>
<item>seventeen<tag>out=17;</tag></item>
<item>eighteen<tag>out=18;</tag></item>
<item>nineteen<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="thousands">
    <item>two thousand</item>
    <item repeat="0-1">and</item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out = rules.digits;</tag></
item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out = rules.teens;</tag></item>
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out =
rules.above_twenty;</tag></item>
</rule>

<rule id="above_twenty">
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></
item>
</rule>
</grammar>
```

### Plan start date, month/day

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

    Scenario 1:
        Input: My plan will start on july twenty three
        Output: 07/23

    Scenario 2:
        Input: My plan will start on july fifteen
        Output: 07/15

    -->

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item repeat="1-10">
            <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></
item>
```

```

<one-of>
    <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
    <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
    <item><ruleref uri="#above_twenty"/><tag>out+= rules.above_twenty;</tag></item>
</one-of>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My plan started on</item>
        <item repeat="0-1">My plan will start on</item>
        <item repeat="0-1">I paid it on</item>
        <item repeat="0-1">I paid bill for</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.mon=""</tag>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>
        <item>june<tag>out.mon+="06";</tag></item>
        <item>july<tag>out.mon+="07";</tag></item>
        <item>august<tag>out.mon+="08";</tag></item>
        <item>september<tag>out.mon+="09";</tag></item>
        <item>october<tag>out.mon+="10";</tag></item>
        <item>november<tag>out.mon+="11";</tag></item>
        <item>december<tag>out.mon+="12";</tag></item>
        <item>jan<tag>out.mon+="01";</tag></item>
        <item>feb<tag>out.mon+="02";</tag></item>
        <item>aug<tag>out.mon+="08";</tag></item>
        <item>sept<tag>out.mon+="09";</tag></item>
        <item>oct<tag>out.mon+="10";</tag></item>
        <item>nov<tag>out.mon+="11";</tag></item>
        <item>dec<tag>out.mon+="12";</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
    </one-of>
</rule>

```

```

<item>first<tag>out=01;</tag></item>
<item>second<tag>out=02;</tag></item>
<item>third<tag>out=03;</tag></item>
<item>fourth<tag>out=04;</tag></item>
<item>fifth<tag>out=05;</tag></item>
<item>sixth<tag>out=06;</tag></item>
<item>seventh<tag>out=07;</tag></item>
<item>eighth<tag>out=08;</tag></item>
<item>ninth<tag>out=09;</tag></item>
<item>one<tag>out=1;</tag></item>
<item>two<tag>out=2;</tag></item>
<item>three<tag>out=3;</tag></item>
<item>four<tag>out=4;</tag></item>
<item>five<tag>out=5;</tag></item>
<item>six<tag>out=6;</tag></item>
<item>seven<tag>out=7;</tag></item>
<item>eight<tag>out=8;</tag></item>
<item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>tenth<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
        <item>tenth<tag>out=10;</tag></item>
        <item>eleventh<tag>out=11;</tag></item>
        <item>twelfth<tag>out=12;</tag></item>
        <item>thirteenth<tag>out=13;</tag></item>
        <item>fourteenth<tag>out=14;</tag></item>
        <item>fifteenth<tag>out=15;</tag></item>
        <item>sixteenth<tag>out=16;</tag></item>
        <item>seventeenth<tag>out=17;</tag></item>
        <item>eighteenth<tag>out=18;</tag></item>
        <item>nineteenth<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></
item>
</rule>
</grammar>

```

### Service start date, month/day

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                    http://www.w3.org/TR/speech-grammar/grammar.xsd"
xml:lang="en-US" version="1.0"
root="main"
mode="voice"
tag-format="semantics/1.0">

<!-- Test Cases

Grammar will support the following inputs:

Scenario 1:
    Input: My plan starts on july twenty three
    Output: 07/23

Scenario 2:
    Input: I want to activate on july fifteen
    Output: 07/15

-->

<rule id="main" scope="public">
    <tag>out=""</tag>
    <item repeat="1-10">
        <item><ruleref uri="#months"/><tag>out= rules.months.mon + "/";</tag></
item>
        <one-of>
            <item><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
            <item><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
            <item><ruleref uri="#above_twenty"/><tag>out+= rules.above_twenty;</
tag></item>
        </one-of>
    </item>
</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">My plan starts on</item>
        <item repeat="0-1">I want to start my plan on</item>
        <item repeat="0-1">Activation date of</item>
        <item repeat="0-1">Start activation on</item>
        <item repeat="0-1">I want to activate on</item>
        <item repeat="0-1">Activate plan starting</item>
        <item repeat="0-1">Starting</item>
        <item repeat="0-1">Start on</item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="months">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.mon=""</tag>
    <one-of>
        <item>january<tag>out.mon+="01";</tag></item>
        <item>february<tag>out.mon+="02";</tag></item>
        <item>march<tag>out.mon+="03";</tag></item>
        <item>april<tag>out.mon+="04";</tag></item>
        <item>may<tag>out.mon+="05";</tag></item>

```

```

<item>june<tag>out.mon+="06";</tag></item>
<item>july<tag>out.mon+="07";</tag></item>
<item>august<tag>out.mon+="08";</tag></item>
<item>september<tag>out.mon+="09";</tag></item>
<item>october<tag>out.mon+="10";</tag></item>
<item>november<tag>out.mon+="11";</tag></item>
<item>december<tag>out.mon+="12";</tag></item>
<item>jan<tag>out.mon+="01";</tag></item>
<item>feb<tag>out.mon+="02";</tag></item>
<item>aug<tag>out.mon+="08";</tag></item>
<item>sept<tag>out.mon+="09";</tag></item>
<item>oct<tag>out.mon+="10";</tag></item>
<item>nov<tag>out.mon+="11";</tag></item>
<item>dec<tag>out.mon+="12";</tag></item>
</one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>first<tag>out=01;</tag></item>
        <item>second<tag>out=02;</tag></item>
        <item>third<tag>out=03;</tag></item>
        <item>fourth<tag>out=04;</tag></item>
        <item>fifth<tag>out=05;</tag></item>
        <item>sixth<tag>out=06;</tag></item>
        <item>seventh<tag>out=07;</tag></item>
        <item>eighth<tag>out=08;</tag></item>
        <item>ninth<tag>out=09;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>tenth<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
        <item>tenth<tag>out=10;</tag></item>

```

```

<item>eleventh<tag>out=11;</tag></item>
<item>twelfth<tag>out=12;</tag></item>
<item>thirteenth<tag>out=13;</tag></item>
<item>fourteenth<tag>out=14;</tag></item>
<item>fifteenth<tag>out=15;</tag></item>
<item>sixteenth<tag>out=16;</tag></item>
<item>seventeenth<tag>out=17;</tag></item>
<item>eighteenth<tag>out=18;</tag></item>
<item>nineteenth<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>

```

## Equipment quantity

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

    Scenario 1:
        Input: The number is one
        Output: 1

    Scenario 2:
        Input: It is ten
        Output: 10

    -->

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <one-of>
            <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></item>
        item>
            <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
            <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
rules.above_twenty;</tag></item>
        </one-of>
            <item repeat="0-1"><ruleref uri="#thanks"/></item>
        </rule>

        <rule id="text">
            <item repeat="0-1"><ruleref uri="#hesitation"/></item>
            <one-of>

```

```

<item repeat="0-1">It is</item>
<item repeat="0-1">The number is</item>
<item repeat="0-1">Order</item>
<item repeat="0-1">I want to order</item>
<item repeat="0-1">Total equipment</item>
</one-of>
</rule>

<rule id="hesitation">
<one-of>
<item>Hmm</item>
<item>Mmm</item>
<item>My</item>
</one-of>
</rule>

<rule id="thanks">
<one-of>
<item>Thanks</item>
<item>I think</item>
</one-of>
</rule>

<rule id="digits">
<item repeat="0-1"><ruleref uri="#text"/></item>
<one-of>
<item>0<tag>out=0;</tag></item>
<item>1<tag>out=1;</tag></item>
<item>2<tag>out=2;</tag></item>
<item>3<tag>out=3;</tag></item>
<item>4<tag>out=4;</tag></item>
<item>5<tag>out=5;</tag></item>
<item>6<tag>out=6;</tag></item>
<item>7<tag>out=7;</tag></item>
<item>8<tag>out=8;</tag></item>
<item>9<tag>out=9;</tag></item>
<item>one<tag>out=1;</tag></item>
<item>two<tag>out=2;</tag></item>
<item>three<tag>out=3;</tag></item>
<item>four<tag>out=4;</tag></item>
<item>five<tag>out=5;</tag></item>
<item>six<tag>out=6;</tag></item>
<item>seven<tag>out=7;</tag></item>
<item>eight<tag>out=8;</tag></item>
<item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="teens">
<item repeat="0-1"><ruleref uri="#text"/></item>
<one-of>
<item>ten<tag>out=10;</tag></item>
<item>eleven<tag>out=11;</tag></item>
<item>twelve<tag>out=12;</tag></item>
<item>thirteen<tag>out=13;</tag></item>
<item>fourteen<tag>out=14;</tag></item>
<item>fifteen<tag>out=15;</tag></item>
<item>sixteen<tag>out=16;</tag></item>
<item>seventeen<tag>out=17;</tag></item>
<item>eighteen<tag>out=18;</tag></item>
<item>nineteen<tag>out=19;</tag></item>
<item>10<tag>out=10;</tag></item>
<item>11<tag>out=11;</tag></item>
<item>12<tag>out=12;</tag></item>
<item>13<tag>out=13;</tag></item>
<item>14<tag>out=14;</tag></item>

```

```

<item>15<tag>out=15;</tag></item>
<item>16<tag>out=16;</tag></item>
<item>17<tag>out=17;</tag></item>
<item>18<tag>out=18;</tag></item>
<item>19<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
        <item>20<tag>out=20;</tag></item>
        <item>30<tag>out=30;</tag></item>
        <item>40<tag>out=40;</tag></item>
        <item>50<tag>out=50;</tag></item>
        <item>60<tag>out=60;</tag></item>
        <item>70<tag>out=70;</tag></item>
        <item>80<tag>out=80;</tag></item>
        <item>90<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>
```

### Bill amount

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <!-- Test Cases

Grammar will support the following inputs:

    Input: I want to make a payment of one hundred ten dollars
    Output: $110

    -->

    <rule id="main" scope="public">
        <tag>out="$"</tag>
        <one-of>
            <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</tag></item>
            <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</tag></item>
        </one-of>
        <item repeat="0-1"><ruleref uri="#thanks"/></item>
    </rule>
</grammar>
```

```

</rule>

<rule id="text">
    <item repeat="0-1"><ruleref uri="#hesitation"/></item>
    <one-of>
        <item repeat="0-1">I want to make a payment for</item>
        <item repeat="0-1">I want to make a payment of</item>
        <item repeat="0-1">Pay a total of</item>
        <item repeat="0-1">Paying</item>
        <item repeat="0-1">Pay bill for </item>
    </one-of>
</rule>

<rule id="hesitation">
    <one-of>
        <item>Hmm</item>
        <item>Mmm</item>
        <item>My</item>
    </one-of>
</rule>

<rule id="thanks">
    <one-of>
        <item>Thanks</item>
        <item>I think</item>
    </one-of>
</rule>

<rule id="digits">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
        <item>three<tag>out.num+=3;</tag></item>
        <item>four<tag>out.num+=4;</tag></item>
        <item>five<tag>out.num+=5;</tag></item>
        <item>six<tag>out.num+=6;</tag></item>
        <item>seven<tag>out.num+=7;</tag></item>
        <item>eight<tag>out.num+=8;</tag></item>
        <item>nine<tag>out.num+=9;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="teens">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
        <item>fifteen<tag>out.teen+=15;</tag></item>
        <item>sixteen<tag>out.teen+=16;</tag></item>
    </one-of>
</rule>

```

```

<item>seventeen<tag>out.teen+=17;</tag></item>
<item>eighteen<tag>out.teen+=18;</tag></item>
<item>nineteen<tag>out.teen+=19;</tag></item>
</one-of>
<item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="above_twenty">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.tens = 0;</tag>
    <one-of>
        <item>twenty<tag>out.tens+=20;</tag></item>
        <item>thirty<tag>out.tens+=30;</tag></item>
        <item>forty<tag>out.tens+=40;</tag></item>
        <item>fifty<tag>out.tens+=50;</tag></item>
        <item>sixty<tag>out.tens+=60;</tag></item>
        <item>seventy<tag>out.tens+=70;</tag></item>
        <item>eighty<tag>out.tens+=80;</tag></item>
        <item>ninety<tag>out.tens+=90;</tag></item>
        <item>hundred<tag>out.tens+=100;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens += rules.digits.num;</tag></item>
</rule>

<rule id="currency">
    <one-of>
        <item repeat="0-1">dollars</item>
        <item repeat="0-1">Dollars</item>
        <item repeat="0-1">dollar</item>
        <item repeat="0-1">Dollar</item>
    </one-of>
</rule>

<rule id="sub_hundred">
    <item repeat="0-1"><ruleref uri="#text"/></item>
    <tag>out.sh = 0;</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></item>
        <item><ruleref uri="#above_twenty"/><tag>out.sh += rules.above_twenty.tens;</tag></item>
    </one-of>
</rule>

<rule id="subThousands">
    <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
    hundred
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty.tens;</tag></item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</tag></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</tag></item>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>
</grammar>
```

## Generic grammars (download)

We provide the following generic grammars: alphanumeric, currency, date (mm/dd/yy), numbers, greeting, hesitation, and agent.

### Alphanumeric

```
<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="main"
          mode="voice"
          tag-format="semantics/1.0">

    <!-- Test Cases

        Scenario 1:
            Input: A B C 1 2 3 4
            Output: ABC1234

        Scenario 2:
            Input: 1 2 3 4 A B C
            Output: 1234ABC

        Scenario 3:
            Input: 1 2 3 4 A B C 1
            Output: 123ABC1
    -->

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item><ruleref uri="#alphanumeric"/><tag>out += rules.alphanumeric.alphanum;</tag></item>
        <item repeat="0-1"><ruleref uri="#alphabets"/><tag>out += rules.alphabets.letters;</tag></item>
            <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.numbers;</tag></item>
        </rule>

        <rule id="alphanumeric" scope="public">
            <tag>out.alphanum=""</tag>
            <item><ruleref uri="#alphabets"/><tag>out.alphanum += rules.alphabets.letters;</tag></item>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out.alphanum += rules.digits.numbers;</tag></item>
            </rule>

            <rule id="alphabets">
                <tag>out.letters=""</tag>
                <tag>out.firstOccurrence=""</tag>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out.firstOccurrence += rules.digits.numbers; out.letters += out.firstOccurrence;</tag></item>
                <item repeat="1->">
                    <one-of>
                        <item>A<tag>out.letters+='A';</tag></item>
                        <item>B<tag>out.letters+='B';</tag></item>
                        <item>C<tag>out.letters+='C';</tag></item>
                        <item>D<tag>out.letters+='D';</tag></item>
                        <item>E<tag>out.letters+='E';</tag></item>
                        <item>F<tag>out.letters+='F';</tag></item>
                    </one-of>
                </item>
            </rule>
        </rule>
    </rule>
</grammar>
```

```

<item>G<tag>out.letters+='G';</tag></item>
<item>H<tag>out.letters+='H';</tag></item>
<item>I<tag>out.letters+='I';</tag></item>
<item>J<tag>out.letters+='J';</tag></item>
<item>K<tag>out.letters+='K';</tag></item>
<item>L<tag>out.letters+='L';</tag></item>
<item>M<tag>out.letters+='M';</tag></item>
<item>N<tag>out.letters+='N';</tag></item>
<item>O<tag>out.letters+='O';</tag></item>
<item>P<tag>out.letters+='P';</tag></item>
<item>Q<tag>out.letters+='Q';</tag></item>
<item>R<tag>out.letters+='R';</tag></item>
<item>S<tag>out.letters+='S';</tag></item>
<item>T<tag>out.letters+='T';</tag></item>
<item>U<tag>out.letters+='U';</tag></item>
<item>V<tag>out.letters+='V';</tag></item>
<item>W<tag>out.letters+='W';</tag></item>
<item>X<tag>out.letters+='X';</tag></item>
<item>Y<tag>out.letters+='Y';</tag></item>
<item>Z<tag>out.letters+='Z';</tag></item>
    </one-of>
</item>
</rule>

<rule id="digits">
    <tag>out.numbers=""</tag>
    <item repeat="1-10">
        <one-of>
            <item>0<tag>out.numbers+=0;</tag></item>
            <item>1<tag>out.numbers+=1;</tag></item>
            <item>2<tag>out.numbers+=2;</tag></item>
            <item>3<tag>out.numbers+=3;</tag></item>
            <item>4<tag>out.numbers+=4;</tag></item>
            <item>5<tag>out.numbers+=5;</tag></item>
            <item>6<tag>out.numbers+=6;</tag></item>
            <item>7<tag>out.numbers+=7;</tag></item>
            <item>8<tag>out.numbers+=8;</tag></item>
            <item>9<tag>out.numbers+=9;</tag></item>
        </one-of>
    </item>
</rule>
</grammar>
```

## Currency

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <rule id="main" scope="public">
        <tag>out="$"</tag>
        <one-of>
            <item><ruleref uri="#sub_hundred"/><tag>out += rules.sub_hundred.sh;</tag></item>
            <item><ruleref uri="#subThousands"/><tag>out += rules.subThousands;</tag></item>
        </one-of>
    </rule>
```

```

<rule id="digits">
    <tag>out.num = 0;</tag>
    <one-of>
        <item>0<tag>out.num+=0;</tag></item>
        <item>1<tag>out.num+=1;</tag></item>
        <item>2<tag>out.num+=2;</tag></item>
        <item>3<tag>out.num+=3;</tag></item>
        <item>4<tag>out.num+=4;</tag></item>
        <item>5<tag>out.num+=5;</tag></item>
        <item>6<tag>out.num+=6;</tag></item>
        <item>7<tag>out.num+=7;</tag></item>
        <item>8<tag>out.num+=8;</tag></item>
        <item>9<tag>out.num+=9;</tag></item>
        <item>one<tag>out.num+=1;</tag></item>
        <item>two<tag>out.num+=2;</tag></item>
        <item>three<tag>out.num+=3;</tag></item>
        <item>four<tag>out.num+=4;</tag></item>
        <item>five<tag>out.num+=5;</tag></item>
        <item>six<tag>out.num+=6;</tag></item>
        <item>seven<tag>out.num+=7;</tag></item>
        <item>eight<tag>out.num+=8;</tag></item>
        <item>nine<tag>out.num+=9;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="teens">
    <tag>out.teen = 0;</tag>
    <one-of>
        <item>ten<tag>out.teen+=10;</tag></item>
        <item>eleven<tag>out.teen+=11;</tag></item>
        <item>twelve<tag>out.teen+=12;</tag></item>
        <item>thirteen<tag>out.teen+=13;</tag></item>
        <item>fourteen<tag>out.teen+=14;</tag></item>
        <item>fifteen<tag>out.teen+=15;</tag></item>
        <item>sixteen<tag>out.teen+=16;</tag></item>
        <item>seventeen<tag>out.teen+=17;</tag></item>
        <item>eighteen<tag>out.teen+=18;</tag></item>
        <item>nineteen<tag>out.teen+=19;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
</rule>

<rule id="above_twenty">
    <tag>out.tens = 0;</tag>
    <one-of>
        <item>twenty<tag>out.tens+=20;</tag></item>
        <item>thirty<tag>out.tens+=30;</tag></item>
        <item>forty<tag>out.tens+=40;</tag></item>
        <item>fifty<tag>out.tens+=50;</tag></item>
        <item>sixty<tag>out.tens+=60;</tag></item>
        <item>seventy<tag>out.tens+=70;</tag></item>
        <item>eighty<tag>out.tens+=80;</tag></item>
        <item>ninety<tag>out.tens+=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#currency"/></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out.tens += rules.digits.num;</tag></item>
</rule>

<rule id="currency">
    <one-of>
        <item repeat="0-1">dollars</item>
        <item repeat="0-1">Dollars</item>
        <item repeat="0-1">dollar</item>

```

```

        <item repeat="0-1">Dollar</item>
    </one-of>
</rule>

<rule id="sub_hundred">
    <tag>out.sh = 0;</tag>
    <one-of>
        <item><ruleref uri="#teens"/><tag>out.sh += rules.teens.teen;</tag></item>
        <item>
            <ruleref uri="#above_twenty"/><tag>out.sh += rules.above_twenty.tens;</tag>
        </item>
        <item><ruleref uri="#digits"/><tag>out.sh += rules.digits.num;</tag></item>
    </one-of>
</rule>

<rule id="subThousands">
    <ruleref uri="#sub_hundred"/><tag>out = (100 * rules.sub_hundred.sh);</tag>
    hundred
    <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty.tens;</tag></item>
    <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens.teen;</tag></item>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits.num;</tag></item>
</rule>
</grammar>
```

### Date, dd/mm

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item repeat="1-10">
            <one-of>
                <item><ruleref uri="#digits"/><tag>out += rules.digits + " ";</tag></item>
                <item><ruleref uri="#teens"/><tag>out += rules.teens+ " ";</tag></item>
                <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+ " ";</tag></item>
            </one-of>
            <item><ruleref uri="#months"/><tag>out = out + rules.months;</tag></item>
        </rule>

        <rule id="months">
            <one-of>
                <item>january<tag>out="january";</tag></item>
                <item>february<tag>out="february";</tag></item>
                <item>march<tag>out="march";</tag></item>
                <item>april<tag>out="april";</tag></item>
                <item>may<tag>out="may";</tag></item>
                <item>june<tag>out="june";</tag></item>
                <item>july<tag>out="july";</tag></item>
```

```

<item>august<tag>out="august";</tag></item>
<item>september<tag>out="september";</tag></item>
<item>october<tag>out="october";</tag></item>
<item>november<tag>out="november";</tag></item>
<item>december<tag>out="december";</tag></item>
<item>jan<tag>out="january";</tag></item>
<item>feb<tag>out="february";</tag></item>
<item>aug<tag>out="august";</tag></item>
<item>sept<tag>out="september";</tag></item>
<item>oct<tag>out="october";</tag></item>
<item>nov<tag>out="november";</tag></item>
<item>dec<tag>out="december";</tag></item>
</one-of>
</rule>

<rule id="digits">
<one-of>
    <item>0<tag>out=0;</tag></item>
    <item>1<tag>out=1;</tag></item>
    <item>2<tag>out=2;</tag></item>
    <item>3<tag>out=3;</tag></item>
    <item>4<tag>out=4;</tag></item>
    <item>5<tag>out=5;</tag></item>
    <item>6<tag>out=6;</tag></item>
    <item>7<tag>out=7;</tag></item>
    <item>8<tag>out=8;</tag></item>
    <item>9<tag>out=9;</tag></item>
    <item>first<tag>out=1;</tag></item>
    <item>second<tag>out=2;</tag></item>
    <item>third<tag>out=3;</tag></item>
    <item>fourth<tag>out=4;</tag></item>
    <item>fifth<tag>out=5;</tag></item>
    <item>sixth<tag>out=6;</tag></item>
    <item>seventh<tag>out=7;</tag></item>
    <item>eighth<tag>out=8;</tag></item>
    <item>ninth<tag>out=9;</tag></item>
    <item>one<tag>out=1;</tag></item>
    <item>two<tag>out=2;</tag></item>
    <item>three<tag>out=3;</tag></item>
    <item>four<tag>out=4;</tag></item>
    <item>five<tag>out=5;</tag></item>
    <item>six<tag>out=6;</tag></item>
    <item>seven<tag>out=7;</tag></item>
    <item>eight<tag>out=8;</tag></item>
    <item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="teens">
<one-of>
    <item>ten<tag>out=10;</tag></item>
    <item>tenth<tag>out=10;</tag></item>
    <item>eleven<tag>out=11;</tag></item>
    <item>twelve<tag>out=12;</tag></item>
    <item>thirteen<tag>out=13;</tag></item>
    <item>fourteen<tag>out=14;</tag></item>
    <item>fifteen<tag>out=15;</tag></item>
    <item>sixteen<tag>out=16;</tag></item>
    <item>seventeen<tag>out=17;</tag></item>
    <item>eighteen<tag>out=18;</tag></item>
    <item>nineteen<tag>out=19;</tag></item>
    <item>tenth<tag>out=10;</tag></item>
    <item>eleventh<tag>out=11;</tag></item>
    <item>twelfth<tag>out=12;</tag></item>
    <item>thirteenth<tag>out=13;</tag></item>
    <item>fourteenth<tag>out=14;</tag></item>

```

```

<item>fifteenth<tag>out=15;</tag></item>
<item>sixteenth<tag>out=16;</tag></item>
<item>seventeenth<tag>out=17;</tag></item>
<item>eighteenth<tag>out=18;</tag></item>
<item>nineteenth<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="above_twenty">
<one-of>
    <item>twenty<tag>out=20;</tag></item>
    <item>thirty<tag>out=30;</tag></item>
</one-of>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>

```

### Date, mm/yy

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item repeat="1-10">
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
            + " ";</tag></item>
            <one-of>
                <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></item>
            item>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
                <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</tag></item>
                <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
                rules.above_twenty;</tag></item>
            </one-of>
        </item>
    </rule>

    <rule id="months">
        <tag>out.mon=""</tag>
        <one-of>
            <item>january<tag>out.mon+="january";</tag></item>
            <item>february<tag>out.mon+="february";</tag></item>
            <item>march<tag>out.mon+="march";</tag></item>
            <item>april<tag>out.mon+="april";</tag></item>
            <item>may<tag>out.mon+="may";</tag></item>
            <item>june<tag>out.mon+="june";</tag></item>
            <item>july<tag>out.mon+="july";</tag></item>
            <item>august<tag>out.mon+="august";</tag></item>
            <item>september<tag>out.mon+="september";</tag></item>
            <item>october<tag>out.mon+="october";</tag></item>
            <item>november<tag>out.mon+="november";</tag></item>
            <item>december<tag>out.mon+="december";</tag></item>
            <item>jan<tag>out.mon+="january";</tag></item>
        </one-of>
    </rule>

```

```

<item>feb<tag>out.mon+="february";</tag></item>
<item>aug<tag>out.mon+="august";</tag></item>
<item>sept<tag>out.mon+="september";</tag></item>
<item>oct<tag>out.mon+="october";</tag></item>
<item>nov<tag>out.mon+="november";</tag></item>
<item>dec<tag>out.mon+="december";</tag></item>
</one-of>
</rule>

<rule id="digits">
<one-of>
    <item>zero<tag>out=0;</tag></item>
    <item>one<tag>out=1;</tag></item>
    <item>two<tag>out=2;</tag></item>
    <item>three<tag>out=3;</tag></item>
    <item>four<tag>out=4;</tag></item>
    <item>five<tag>out=5;</tag></item>
    <item>six<tag>out=6;</tag></item>
    <item>seven<tag>out=7;</tag></item>
    <item>eight<tag>out=8;</tag></item>
    <item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="teens">
<one-of>
    <item>ten<tag>out=10;</tag></item>
    <item>eleven<tag>out=11;</tag></item>
    <item>twelve<tag>out=12;</tag></item>
    <item>thirteen<tag>out=13;</tag></item>
    <item>fourteen<tag>out=14;</tag></item>
    <item>fifteen<tag>out=15;</tag></item>
    <item>sixteen<tag>out=16;</tag></item>
    <item>seventeen<tag>out=17;</tag></item>
    <item>eighteen<tag>out=18;</tag></item>
    <item>nineteen<tag>out=19;</tag></item>
</one-of>
</rule>

<!-- <rule id="singleDigit">
    <item><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule> -->

<rule id="thousands">
<!-- <item>
    <ruleref uri="#digits"/>
    <tag>out = (1000 * rules.digits);</tag>
    thousand
</item> -->
<item>two thousand<tag>out=2000;</tag></item>
<item repeat="0-1">&lt;/item>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
item>
<item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</tag></item>
<item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
</rule>

<rule id="above_twenty">
<one-of>
    <item>twenty<tag>out=20;</tag></item>
    <item>thirty<tag>out=30;</tag></item>
    <item>forty<tag>out=40;</tag></item>
    <item>fifty<tag>out=50;</tag></item>
    <item>sixty<tag>out=60;</tag></item>
    <item>seventy<tag>out=70;</tag></item>

```

```

        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
    </one-of>
    <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></
item>
</rule>

</grammar>

```

### Date, dd/mm/yyyy

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/06/grammar
        http://www.w3.org/TR/speech-grammar/grammar.xsd"
    xml:lang="en-US" version="1.0"
    root="main"
    mode="voice"
    tag-format="semantics/1.0">

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <item repeat="1-10">
            <one-of>
                <item><ruleref uri="#digits"/><tag>out += rules.digits + " ";</tag></
item>
                <item><ruleref uri="#teens"/><tag>out += rules.teens+ " ";</tag></item>
                <item><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty+ "
";</tag></item>
            </one-of>
            <item repeat="1"><ruleref uri="#months"/><tag>out = out + rules.months.mon
+ " "</tag></item>
            <one-of>
                <item><ruleref uri="#thousands"/><tag>out += rules.thousands;</tag></
item>
                <item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</
tag></item>
                <item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</
tag></item>
                <item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out +=
rules.above_twenty;</tag></item>
            </one-of>
        </item>
    </rule>

    <rule id="months">
        <tag>out.mon=""</tag>
        <one-of>
            <item>january<tag>out.mon+="january";</tag></item>
            <item>february<tag>out.mon+="february";</tag></item>
            <item>march<tag>out.mon+="march";</tag></item>
            <item>april<tag>out.mon+="april";</tag></item>
            <item>may<tag>out.mon+="may";</tag></item>
            <item>june<tag>out.mon+="june";</tag></item>
            <item>july<tag>out.mon+="july";</tag></item>
            <item>august<tag>out.mon+="august";</tag></item>
            <item>september<tag>out.mon+="september";</tag></item>
            <item>october<tag>out.mon+="october";</tag></item>
            <item>november<tag>out.mon+="november";</tag></item>
            <item>december<tag>out.mon+="december";</tag></item>
            <item>jan<tag>out.mon+="january";</tag></item>
            <item>feb<tag>out.mon+="february";</tag></item>
            <item>aug<tag>out.mon+="august";</tag></item>

```

```

<item>sept<tag>out.mon+="september";</tag></item>
<item>oct<tag>out.mon+="october";</tag></item>
<item>nov<tag>out.mon+="november";</tag></item>
<item>dec<tag>out.mon+="december";</tag></item>
</one-of>
</rule>

<rule id="digits">
<one-of>
<item>zero<tag>out=0;</tag></item>
<item>one<tag>out=1;</tag></item>
<item>two<tag>out=2;</tag></item>
<item>three<tag>out=3;</tag></item>
<item>four<tag>out=4;</tag></item>
<item>five<tag>out=5;</tag></item>
<item>six<tag>out=6;</tag></item>
<item>seven<tag>out=7;</tag></item>
<item>eight<tag>out=8;</tag></item>
<item>nine<tag>out=9;</tag></item>
</one-of>
</rule>

<rule id="teens">
<one-of>
<item>ten<tag>out=10;</tag></item>
<item>eleven<tag>out=11;</tag></item>
<item>twelve<tag>out=12;</tag></item>
<item>thirteen<tag>out=13;</tag></item>
<item>fourteen<tag>out=14;</tag></item>
<item>fifteen<tag>out=15;</tag></item>
<item>sixteen<tag>out=16;</tag></item>
<item>seventeen<tag>out=17;</tag></item>
<item>eighteen<tag>out=18;</tag></item>
<item>nineteen<tag>out=19;</tag></item>
</one-of>
</rule>

<rule id="thousands">
<item>two thousand<tag>out=2000;</tag></item>
<item repeat="0-1">and</item>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
<item repeat="0-1"><ruleref uri="#teens"/><tag>out += rules.teens;</tag></item>
<item repeat="0-1"><ruleref uri="#above_twenty"/><tag>out += rules.above_twenty;</tag></item>
</rule>

<rule id="above_twenty">
<one-of>
<item>twenty<tag>out=20;</tag></item>
<item>thirty<tag>out=30;</tag></item>
<item>forty<tag>out=40;</tag></item>
<item>fifty<tag>out=50;</tag></item>
<item>sixty<tag>out=60;</tag></item>
<item>seventy<tag>out=70;</tag></item>
<item>eighty<tag>out=80;</tag></item>
<item>ninety<tag>out=90;</tag></item>
</one-of>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>

</grammar>
```

## Numbers, digits

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="digits"
          mode="voice"
          tag-format="semantics/1.0">

    <rule id="digits">
        <tag>out=""</tag>
        <item><ruleref uri="#singleDigit"/><tag>out += rules.singleDigit.digit;</tag></
item>
    </rule>

    <rule id="singleDigit">
        <tag>out.digit=""</tag>
        <item repeat="1-10">
            <one-of>
                <item>0<tag>out.digit+=0;</tag></item>
                <item>zero<tag>out.digit+=0;</tag></item>
                <item>1<tag>out.digit+=1;</tag></item>
                <item>one<tag>out.digit+=1;</tag></item>
                <item>2<tag>out.digit+=2;</tag></item>
                <item>two<tag>out.digit+=2;</tag></item>
                <item>3<tag>out.digit+=3;</tag></item>
                <item>three<tag>out.digit+=3;</tag></item>
                <item>4<tag>out.digit+=4;</tag></item>
                <item>four<tag>out.digit+=4;</tag></item>
                <item>5<tag>out.digit+=5;</tag></item>
                <item>five<tag>out.digit+=5;</tag></item>
                <item>6<tag>out.digit+=6;</tag></item>
                <item>six<tag>out.digit+=6;</tag></item>
                <item>7<tag>out.digit+=7;</tag></item>
                <item>seven<tag>out.digit+=7;</tag></item>
                <item>8<tag>out.digit+=8;</tag></item>
                <item>eight<tag>out.digit+=8;</tag></item>
                <item>9<tag>out.digit+=9;</tag></item>
                <item>nine<tag>out.digit+=9;</tag></item>
            </one-of>
        </item>
    </rule>
</grammar>

```

## Numbers, ordinal

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                               http://www.w3.org/TR/speech-grammar/grammar.xsd"
          xml:lang="en-US" version="1.0"
          root="main"
          mode="voice"
          tag-format="semantics/1.0">

    <rule id="main" scope="public">
        <tag>out=""</tag>
        <one-of>
            <item repeat="1"><ruleref uri="#digits"/><tag>out+= rules.digits;</tag></
item>
    </rule>
</grammar>

```

```

        <item repeat="1"><ruleref uri="#teens"/><tag>out+= rules.teens;</tag></item>
        <item repeat="1"><ruleref uri="#above_twenty"/><tag>out+=
rules.above_twenty;</tag></item>
    </one-of>
</rule>

<rule id="digits">
    <one-of>
        <item>0<tag>out=0;</tag></item>
        <item>1<tag>out=1;</tag></item>
        <item>2<tag>out=2;</tag></item>
        <item>3<tag>out=3;</tag></item>
        <item>4<tag>out=4;</tag></item>
        <item>5<tag>out=5;</tag></item>
        <item>6<tag>out=6;</tag></item>
        <item>7<tag>out=7;</tag></item>
        <item>8<tag>out=8;</tag></item>
        <item>9<tag>out=9;</tag></item>
        <item>one<tag>out=1;</tag></item>
        <item>two<tag>out=2;</tag></item>
        <item>three<tag>out=3;</tag></item>
        <item>four<tag>out=4;</tag></item>
        <item>five<tag>out=5;</tag></item>
        <item>six<tag>out=6;</tag></item>
        <item>seven<tag>out=7;</tag></item>
        <item>eight<tag>out=8;</tag></item>
        <item>nine<tag>out=9;</tag></item>
    </one-of>
</rule>

<rule id="teens">
    <one-of>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
        <item>ten<tag>out=10;</tag></item>
        <item>eleven<tag>out=11;</tag></item>
        <item>twelve<tag>out=12;</tag></item>
        <item>thirteen<tag>out=13;</tag></item>
        <item>fourteen<tag>out=14;</tag></item>
        <item>fifteen<tag>out=15;</tag></item>
        <item>sixteen<tag>out=16;</tag></item>
        <item>seventeen<tag>out=17;</tag></item>
        <item>eighteen<tag>out=18;</tag></item>
        <item>nineteen<tag>out=19;</tag></item>
    </one-of>
</rule>

<rule id="above_twenty">
    <one-of>
        <item>twenty<tag>out=20;</tag></item>
        <item>thirty<tag>out=30;</tag></item>
        <item>forty<tag>out=40;</tag></item>
        <item>fifty<tag>out=50;</tag></item>
        <item>sixty<tag>out=60;</tag></item>
        <item>seventy<tag>out=70;</tag></item>
        <item>eighty<tag>out=80;</tag></item>
        <item>ninety<tag>out=90;</tag></item>
        <item>twenty<tag>out=20;</tag></item>
    </one-of>
</rule>
```

```

<item>30<tag>out=30;</tag></item>
<item>40<tag>out=40;</tag></item>
<item>50<tag>out=50;</tag></item>
<item>60<tag>out=60;</tag></item>
<item>70<tag>out=70;</tag></item>
<item>80<tag>out=80;</tag></item>
<item>90<tag>out=90;</tag></item>
</one-of>
<item repeat="0-1"><ruleref uri="#digits"/><tag>out += rules.digits;</tag></item>
</rule>
</grammar>

```

## Agent

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <rule id="main" scope="public">
    <tag>out=""</tag>
    <ruleref uri="#text"/><tag>out = rules.text</tag>
  </rule>

  <rule id="text">
    <one-of>
      <item>Can I talk to the agent<tag>out="You will be transferred to the agent in
a while"</tag></item>
      <item>talk to an agent<tag>out="You will be transferred to the agent in a
while"</tag></item>
    </one-of>
  </rule>
</grammar>

```

## Greeting

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <rule id="main" scope="public">
    <tag>out=""</tag>
    <ruleref uri="#text"/><tag>out = rules.text</tag>
  </rule>

  <rule id="text">
    <one-of>
      <item>hey<tag>out="Greeting"</tag></item>
      <item>hi<tag>out="Greeting"</tag></item>
      <item>Hi<tag>out="Greeting"</tag></item>
    </one-of>
  </rule>
</grammar>

```

```

<item>Hey<tag>out="Greeting"</tag></item>
<item>Hello<tag>out="Greeting"</tag></item>
<item>hello<tag>out="Greeting"</tag></item>
</one-of>
</rule>
</grammar>

```

## Hesitation

```

<?xml version="1.0" encoding="UTF-8" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/06/grammar
    http://www.w3.org/TR/speech-grammar/grammar.xsd"
  xml:lang="en-US" version="1.0"
  root="main"
  mode="voice"
  tag-format="semantics/1.0">

  <rule id="main" scope="public">
    <tag>out=""</tag>
    <ruleref uri="#text"/><tag>out = rules.text</tag>
  </rule>

  <rule id="text">
    <one-of>
      <item>Hmm<tag>out="Waiting for your input"</tag></item>
      <item>Mmm<tag>out="Waiting for your input"</tag></item>
      <item>Can you please wait<tag>out="Waiting for your input"</tag></item>
    </one-of>
  </rule>
</grammar>

```

# Creating custom slot types

For each intent, you can specify parameters that indicate the information that the intent needs to fulfill the user's request. These parameters, or slots, have a type. A *slot type* is a list of values that Amazon Lex V2 uses to train the machine learning model to recognize values for a slot. For example, you can define a slot type called *Genres* with values such as "comedy," "adventure," "documentary," etc. You can define synonyms for a slot type value. For example, you can define the synonyms "funny" and "humorous" for the value "comedy."

You can configure the slot type to expand the slot values. Slot values will be used as training data and the model will resolve the slot to the value provided by the user if it is similar to the slot values and synonyms of those values. This is the default behavior. Amazon Lex V2 maintains a list of possible resolutions for a slot. Each entry in the list provides a *resolved value* that Amazon Lex V2 recognized as additional possibilities for the slot. A resolved value is the best effort to match the slot value. The list contains up to five values.

Alternatively, you can configure the slot type to restrict resolution to the slot values. In this case, the model will resolve a slot value entered by the user to an existing slot value only if it is the same as that slot value or it is a synonym. For example, if the user enters "funny" it will resolve to the slot value "comedy."

When the value entered by the user is a synonym of a slot type value, the model returns that slot type value as the first entry in the list of `resolvedValues`. For example, if the user enters "funny," the model populates the `originalValue` field with the value "funny" and the first entry in the `resolvedValues`

field with "comedy." You can configure the `valueSelectionStrategy` when you create or update a slot type with the [CreateSlotType](#) operation so that the slot value is filled with the first value in the resolution list.

Custom slot types support inputs using spelling styles. You can use the spell-by-letter and spell-by-word styles to help your customers enter letters. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

If you are using a Lambda function, the input event to the function includes a resolution list called `resolvedValues`. The following example shows the slot section of the input to a Lambda function:

```
"slots": {  
    "MovieGenre": {  
        "value": {  
            "originalValue": "funny",  
            "interpretedValue": "comedy",  
            "resolvedValues": [  
                "comedy"  
            ]  
        }  
    }  
}
```

For each slot type, you can define a maximum of 10,000 values and synonyms. Each bot can have a total number of 50,000 slot type values and synonyms. For example, you can have 5 slot types, each with 5,000 values and 5,000 synonyms, or you can have 10 slot types, each with 2,500 values and 2,500 synonyms.

## Using composite slots

A composite slot is a combination of two or more slots that capture multiple pieces of information in a single user input. For example, you can configure the bot to elicit the location by requesting for the "city and state or zipcode". In contrast, when the conversation is configured to use separate slot types resulting in a rigid conversational experience ("What is the city?" followed by "What is the zipcode?"). With a composite slot, you can capture all the information through a single slot. A composite slot is a combination of slots called subslots, such as city, state, and zip code.

You can use a combination of available Amazon Lex slot types (built-ins) and your own slots (custom slots). You can design logical expressions to capture information within the required subslots. For example: city and state or zipcode.

The composite slot type is only available in en-US.

### Creating a composite slot type

To use subslots within a composite slot, you must first configure the composite slot type. To do so, use the adding a slot type console steps or the API operation. After you have chosen the name and a description for the composite slot type, you have to provide information for subslots. For more information on adding a slot type, see [Adding slot types \(p. 45\)](#)

### Subslots

A composite slot type requires configuration of the underlying slots, called subslots. If you would like to elicit multiple pieces of information from a customer in one request, configure a combination of subslots. For example: city, state, and zipcode. You can add up to 6 subslots for a composite slot.

Slots of singular slot types may be used to add subslots to the composite slot type. However, you cannot use a composite slot type as a slot type for a subslot.

The following images are an illustration of a composite slot “Car”, which is a combination of subslots: Color, FuelType, Manufacturer, Model, VIN, and Year.

**Slot type** [Info](#)

Slot type name: Cars

Subslots: Color, FuelType, Manufacturer, Model, VIN, Year

[View slot type details](#)

Slot expression - *optional* [Info](#)  
Define the combination of subslots that your bot prompts for. If you don't define an expression, Amazon Lex prompts for all subslots.

(Color AND FuelType AND Manufacturer) OR (VIN AND Year)

Use , to separate different subslots; Use (), AND, OR to complete the expression.

---

**Subslots** [Info](#)

Subslot name	Subslot type	Action
Color	Colors	<a href="#">Remove</a>
FuelType	FuelTypes	<a href="#">Remove</a>
Manufacturer	Manufacturers	<a href="#">Remove</a>
Model	Models	<a href="#">Remove</a>
VIN	AMAZON.AlphaNumeric	<a href="#">Remove</a>
Year	Years	<a href="#">Remove</a>

[Add new subslot](#)

You have reached the limit of 6 subslots.

## Expression builder

To drive fulfillment of a composite slot, you can optionally use the expression builder. With the expression builder, you can design a logical slot expression to capture the required subslot values in the desired order. As part of the boolean expression, you can use operators such as AND and OR. Based on the designed expression, when the required subslots are fulfilled, the composite slot is considered fulfilled.

### Using a composite slot type

For some intents, you might want to capture different slots as part of a single slot. For example, a car maintenance scheduling bot might have an intent with the following utterance:

My car is a {car}

The intent expects that the {car} composite slot contains a list of the slots, comprising details of the car. For example, "2021 White Toyota Camry".

The composite slot differs from a multi-valued slot. The composite slot is comprised of multiple slots, each with its own value. Whereas, a multi-valued slot is a singular slot that can contain a list of values. For more information on multi-values slots see, [Using multiple values in a slot \(p. 193\)](#)

For a composite slot, Amazon Lex returns a value for each subslot in the response to the `RecognizeText` or `RecognizeUtterance` operation. The following is the slot information returned for the utterance: "I want to schedule a service for my "2021 White Toyota Camry" from the CarService bot.

```
{JSON}
  "slots": {
  };
  "values": {
  };
```

A composite slot can be elicited for in the first turn or the n-th turn of a conversation. Based on the input values supplied, the composite slot can elicit for the remaining required subslots.

Composite slots always return a value for each subslot. When the utterance does not contain a recognizable value for a given subslot, there is no response returned for that particular subslot.

Composite slots work with both text and voice input.

When adding a slot to an intent, a composite slot is only available as a custom slot type.

You can use Composite slots in prompts. For example, you can set the confirmation prompt for an intent.

Would you like me to schedule service for your 2021 White Toyota Camry?

When Amazon Lex sends the prompt to the user, it sends "Would you like me to schedule service for your 2021 White Toyota Camry?"

Each subslot is configured as a slot. You can add slot prompts to elicit the subslot and sample utterances. You can enable wait and continue for a subslot as well as default values. For more information, see [Using default slot values \(p. 230\)](#)

The screenshot shows the 'Cars (Composite)' slot configuration in the Amazon Lex V2 developer console. At the top, there's a navigation bar with tabs for 'Cars (Composite)', 'Color', 'FuelType', 'Manufacturer', 'Model', 'VIN', and 'Year'. Below the navigation bar, the title 'Car (Composite)' is displayed. A section titled 'Slot prompts' contains a note about eliciting the slot and a message example: 'Bot elicits information' followed by 'Message: What car do you have?'. Another section titled 'Sample utterances (0) - optional' includes a search bar ('Find utterances'), a sorting dropdown ('Sort by added (ascending)'), and buttons for 'Preview' and 'Plain text'. Below this, a message states 'No sample utterances' and 'You haven't added any sample utterances yet.' At the bottom, there's a text input field containing 'I want to fix a car', a character limit note ('Maximum 250 characters. Valid characters: A-Z, a-z, 0-9, @, #, \$'), and a 'Add utterance' button.

You can use slot obfuscation to mask the whole composite slot in conversation logs. Please note that slot obfuscation is applied at the composite slot level and when enabled, the values for subslots belonging to a composite slot are obfuscated. When you obfuscate slot values, the value of each of the slot values is replaced with the name of the slot. For more information, see [Obscuring slot values in logs \(p. 286\)](#).

## Slot info

**Slot info** [Info](#)

Slot name:  Maximum 100 characters. Valid characters: A-Z, a-z, 0-9, -, \_.

Description - optional:   
Maximum 200 characters.

Required for this intent

Enable slot obfuscation for entire slot

- Color: Store as {Color}
- FuelType: Store as {FuelType}
- Manufacturer: Store as {Manufacturer}
- Model: Store as {Model}
- VIN: Store as {VIN}
- Year: Store as {Year}

### Editing a composite slot type

You can edit a subslot from within the composite slot configuration in order to modify subslot name and slot type. However, when a composite slot is in use by an intent, you will have to edit the intents before modifying the subslot.

i Existing intents use this slot type. To build the language successfully, you may need to configure those intents after editing sub slots.

### Deleting a composite slot type

You can delete a subslot from within the composite slot configuration. Please note that when a subslot is in use within an intent, the subslots are still removed from that intent.

### Delete slot type Address?

⚠ This slot type is used by slots in existing intents. To build the language successfully, you may need to configure intents after deleting it.

This slot type **Address** will be deleted and cannot be recovered later.

[Cancel](#) [Delete](#)

The slot expression in the expression builder provides an alert to inform about the deleted subslots.

Slot type [Info](#)

Slot type name

Cars [▼](#) [⟳](#) [Create slot type](#)

Subslots

Color, FuelType, Manufacturer, Model, VIN, Year

[View slot type details](#)

Slot expression - *optional* [Info](#)

Define the combination of subslots that your bot prompts for. If you don't define an expression, Amazon Lex prompts for all subslots.

(Color AND FuelType AND Manufacturer) OR (VIN AND Year)

Use , to separate different subslots; Use (), AND, OR to complete the expression.

## Creating a custom vocabulary to improve speech recognition

You can give Amazon Lex V2 more information about how to process audio conversations with a bot by creating a custom vocabulary in a specific language. A *custom vocabulary* is a list of specific phrases that you want Amazon Lex V2 to recognize in the audio input. These are generally proper nouns or domain-specific words that Amazon Lex V2 doesn't recognize.

For example, suppose that you have a tech support bot. You can add "backup" to a custom vocabulary to help the bot transcribe the audio correctly as "backup," even when the audio sounds like "pack up." A custom vocabulary can also help recognize rare words in the audio such as "solvency" for financial services or proper nouns such as "Cognito" or "Monitron."

## Custom vocabulary basics

- A custom vocabulary works on the transcription of audio input to a bot. You must provide sample utterances to recognize an intent or slot value.
- A custom vocabulary is unique to a specific language. You must configure custom vocabularies independently for each language. Custom vocabularies are supported only for the English (UK) and English (US) languages.
- Custom vocabularies support only 8 kHz audio input. It is available with [contact center integrations](#) supported by Amazon Lex V2. The [test window](#) in the Amazon Lex V2 console does not support custom vocabularies as it uses 16 kHz audio input.

Amazon Lex V2 uses custom vocabularies to elicit both intents and slots. The same custom vocabulary file is used for intents and slots. You can selectively turn off the custom vocabulary capability for a slot when you add a slot type.

**Eliciting an intent** – You can create a custom vocabulary for eliciting an intent. These phrases are used to transcription when your bot is determining the user's intent. For example, if you configured the phrase "backup" in your custom vocabulary, Amazon Lex V2 transcribes the user input to "can you please backup my photos?"—even when the audio sounds like "can you please pack up my photos." You can specify

the degree of boosting for each phrase by configuring a weight of 0, 1, 2, or 3. You can also specify an alternate representation for the phrase in the final speech to text output by adding a `displayAs` field.

The custom vocabulary phrases used for improving transcription during intent elicitation don't affect transcriptions while eliciting slots. For more information about creating a custom vocabulary for eliciting intents, see [Creating a custom vocabulary for eliciting intents and slots \(p. 188\)](#).

**Eliciting custom slots** – You can use a custom vocabulary to improve slot recognition for audio conversations. To improve your Amazon Lex V2 bot's ability to recognize slot values, create a custom slot and add the slot values to the custom slot, then choose **Use slot values as custom vocabulary**. Examples of slot values include product names, catalogs, or proper nouns. You shouldn't use common words or phrases such as "yes" and "no" in custom vocabularies.

After the slot values are added, these values are used for improving slot recognition when the bot is expecting input for the custom slot. These values aren't used for transcription when eliciting an intent. For more information, see [Adding slot types \(p. 45\)](#).

## Best practices for creating a custom vocabulary

### Eliciting an intent

- Custom vocabularies work best when used to target specific words or phrases. Only add words to a custom vocabulary if they are not readily recognized by Amazon Lex V2.
- Decide how much weight to give a word based on how often the word isn't recognized in the transcription and how rare the word is in the input. Difficult to pronounce words require a higher weight.
- Use a representative test set to determine if a weight is appropriate. You can collect an audio test set by turning on audio logging in conversation logs.
- Avoid using short words like "on," "it," "to," "yes," "no" in a custom vocabulary.

### Eliciting a custom slot

- Add the values to the custom slot type that you expect to be recognized. Add all the possible slot values for the custom slot type, no matter how common or rare the slot value is.
- Enable the option only when the custom slot type contains a list of catalog values or entities such as product names or mutual funds.
- Disable the option if the slot type is used to capture generic phrases such as "yes," "no," "I don't know," "maybe," or generic words such as "one," "two," "three."
- Limit the number of slot values and synonyms to 500 or less for best performance.

Enter acronyms or other words whose letters should be pronounced individually as single letters separated by a period and a space. Don't use individual letters unless they are part of a phrase, such as "J. P. Morgan" or "A. W. S." You can use upper- or lower-case letters to define an acronym.

## Creating a custom vocabulary for eliciting intents and slots

You can use the Amazon Lex V2 console to create and manage a custom vocabulary, or you can use Amazon Lex V2 API operations. There are 2 ways of creating a custom vocabulary through the console:

## Console

### Import custom vocabulary in the console:

1. Open the Amazon Lex V2 console at <https://console.aws.amazon.com/lexv2/home>
2. From the list of bots, choose the bot which you want to add the custom vocabulary.
3. On the bot detail page, from the **Add languages** section, choose **View languages**.
4. From the list of languages, choose the language to which you want to add the custom vocabulary.

### Create a new custom vocabulary directly through the console:

1. Click on **Create** in the **Custom Vocabulary** section of the language details page. This will open an editing window with no custom vocabulary present.
2. Add inputs for phrase, DisplayAs, and weight as required. You can further make inline edits to added items by updating their fields or deleting them from the list.
3. Click on **Save**. Please note: the new custom vocabulary is only saved in your bot after you click on **Save**.
4. You can continue making inline edits in this page and click **Save** when done.
5. This page also allows you import, export, and delete a custom vocabulary file from the drop-down menu on the top right.

## API

### Use the **ListCustomVocabularyItems** API to view the custom vocabulary entries:

1. Use the **ListCustomVocabularyItems** operation to view the custom vocabulary entries. The request body will look like this:

```
{  
    "maxResults": number,  
    "nextToken": "string"  
}
```

2. Please note that **maxResults** and **nextToken** are optional fields for the request body.
3. The response from the **ListCustomVocabularyItems** operation looks like this:

```
{  
    "botId": "string",  
    "botVersion": "string",  
    "localeId": "string",  
    "customVocabularyItems": [  
        {  
            "itemId": "string",  
            "phrase": "string",  
            "weight": number,  
            "displayAs": "string"  
        }  
    ]  
}
```

### Use the **BatchCreateCustomVocabularyItem** API to create new custom vocabulary entries:

1. If your bot's locale does not have a custom vocabulary created yet, please follow the steps to use the [StartImport](#) to create a custom vocabulary.

2. After the custom vocabulary has been created, use the `BatchCreateCustomVocabularyItem` operation to create new custom vocabulary entries. The request body will look like this:

```
{  
    "customVocabularyItemList": [  
        {  
            "phrase": "string",  
            "weight": number,  
            "displayAs": "string"  
        }  
    ]  
}
```

3. Please note that `weight` and `displayAs` are optional fields for the request body.
4. The response from the `BatchCreateCustomVocabularyItem` will look like this:

```
{  
    "botId": "string",  
    "botVersion": "string",  
    "localeId": "string",  
    "errors": [  
        {  
            "itemId": "string",  
            "errorMessage": "string",  
            "errorCode": "string"  
        }  
    ],  
    "resources": [  
        {  
            "itemId": "string",  
            "phrase": "string",  
            "weight": number,  
            "displayAs": "string"  
        }  
    ]  
}
```

5. As this is a batch operation, the request will not fail if one of the items fails to create. The errors list will contain information about why the operation failed for that specific entry. The resources list will contain all of the entries that were successfully created.
6. For `BatchCreateCustomVocabularyItem`, you can expect see these types of errors:
  - `RESOURCE_DOES_NOT_EXIST`: The custom vocabulary does not exist. Follow the steps for creating a custom vocabulary before calling this operation.
  - `DUPLICATE_INPUT`: The list of inputs contains duplicate phrases.
  - `RESOURCE_ALREADY_EXISTS`: The given phrase for the entry already exists in your custom vocabulary.
  - `INTERNAL_SERVER_FAILURE`: There was an error in the backend while processing your request. This may indicate a service outage or another issue.

**Use the `BatchDeleteCustomVocabularyItem` API to delete existing custom vocabulary entries:**

1. If your bot's locale does not have a custom vocabulary created yet, please follow the steps for Use the [StartImport](#) to create a custom vocabulary to create one.
2. After the custom vocabulary has been created, use the `BatchDeleteCustomVocabularyItem` operation to delete existing custom vocabulary entries. The request body will look like this:

```
{  
    "customVocabularyItemList": [  
        {  
            "itemId": "string"  
        }  
    ]  
}
```

3. The response from the `BatchDeleteCustomVocabularyItem` will look like this:

```
{  
    "botId": "string",  
    "botVersion": "string",  
    "localeId": "string",  
    "errors": [  
        {  
            "itemId": "string",  
            "errorMessage": "string",  
            "errorCode": "string"  
        }  
    ],  
    "resources": [  
        {  
            "itemId": "string",  
            "phrase": "string",  
            "weight": number,  
            "displayAs": "string"  
        }  
    ]  
}
```

4. As this is a batch operation, the request will not fail if one of the items fails to delete. The errors list will contain information about why the operation failed for that specific entry. The resources list will contain all of the entries that were successfully deleted.
5. For `BatchDeleteCustomVocabularyItem`, you can expect see these types of errors:
- `RESOURCE_DOES_NOT_EXIST`: The custom vocabulary entry you are trying to delete does not exist.
  - `INTERNAL_SERVER_FAILURE`: There was an error in the backend while processing your request. This may indicate a service outage or another issue.

**Use the `BatchUpdateCustomVocabularyItem` API to update existing custom vocabulary entries:**

1. If your bot's locale does not have a custom vocabulary created yet, please follow the steps for Use the [StartImport](#) to create a custom vocabulary to create a custom vocabulary.
2. After the custom vocabulary has been created, use the `BatchUpdateCustomVocabularyItem` operation to update existing custom vocabulary entries. The request body will look like this:

```
{  
    "customVocabularyItemList": [  
        {  
            "itemId": "string",  
            "phrase": "string",  
            "weight": number,  
            "displayAs": "string"  
        }  
    ]  
}
```

}

3. Please note that weight and displayAs are optional fields for the request body.
4. The response from the BatchUpdateCustomVocabularyItem will look like this:

```
{  
    "botId": "string",  
    "botVersion": "string",  
    "localeId": "string",  
    "errors": [  
        {  
            "itemId": "string",  
            "errorMessage": "string",  
            "errorCode": "string"  
        }  
    ],  
    "resources": [  
        {  
            "itemId": "string",  
            "phrase": "string",  
            "weight": number,  
            "displayAs": "string"  
        }  
    ]  
}
```

5. As this is a batch operation, the request will not fail if one of the items fails to delete. The errors list will contain information about why the operation failed for that specific entry. The resources list will contain all of the entries that were successfully updated.
6. For BatchUpdateCustomVocabularyItem, you can expect see these types of errors:
  - RESOURCE\_DOES\_NOT\_EXIST: The custom vocabulary entry you are trying to update does not exist.
  - DUPLICATE\_INPUT: The list of inputs contains duplicate itemIds.
  - RESOURCE\_ALREADY\_EXISTS: The given phrase for the entry already exists in your custom vocabulary.
  - INTERNAL\_SERVER\_FAILURE: There was an error in the backend while processing your request. This may indicate a service outage or another issue.

## Creating a custom vocabulary file

A custom vocabulary file is a tab-separated list of values that contain the phrase to recognize, a weight to give the boost, and a displayAs field which will replace the phrase in the speech transcript. Phrases with a higher boost value are more likely to be used when they appear in the audio input.

The custom vocabulary file must be named **CustomVocabulary.tsv**, and must be compressed in a zip file before it can be imported. The zip file must be less than 300 MB in size. The maximum number of phrases in a custom vocabulary is 500.

- **phrase** 1–4 words that should be recognized. Separate words in the phrase with spaces. You can't have duplicate phrases in the file. The phrase field is required.
- **weight** – The degree to which the phrase recognition is boosted. The value is an integer 0, 1, 2, or 3. If you don't specify a weight, the default value is 1. Decide on the weight based on how often the word isn't recognized in the transcription and on how rare the word is in the input. The weight 0 means that no boosting will be applied and the entry will only be used for performing replacements using the displayAs field.

- **displayAs** – Defines how you want your phrase to look in your transcription output. This is an optional field in the custom vocabulary.

The custom vocabulary file must contain a header row with the headers "phrase," "weight," and "displayAs". The headers can be in any order, but must follow the above nomenclature.

The following example is a custom vocabulary file. The required tab character to separate the phrase, the weight, and the displayAs is represented by the text "[TAB]". If you use this example, replace the text with a tab character.

```
phrase[TAB]weight[TAB]displayAs
Newcastle[TAB]2
Hobart[TAB]2[TAB]Hobart, Australia
U. Dub[TAB]1[TAB]University of Washington, Seattle
W. S. U.[TAB]3
Issaquah
Kennewick
```

## Using multiple values in a slot

### Note

Multiple value slots are only supported in the English (US) language.

For some intents, you might want to capture multiple values for a single slot. For example, a pizza ordering bot might have an intent with the following utterance:

```
I want a pizza with {toppings}
```

The intent expects that the {toppings} slot contains a list of the toppings that the customer wants on their pizza, for example "pepperoni and pineapple".

To configure a slot to capture multiple values, you set the `allowMultipleValues` field on the slot to true. You can set the field using the console or with the [CreateSlot](#) or [UpdateSlot](#) operation.

You can only mark slots with custom slot types as multi-value slots.

For a multi-value slot, Amazon Lex V2 returns a list of slot values in the response to the [RecognizeText](#) or [RecognizeUtterance](#) operation. The following is the slot information returned for the utterance "I want a pizza with pepperoni and pineapple" from the OrderPizza bot.

```
"slots": {
    "toppings": {
        "shape": "List",
        "value": {
            "interpretedValue": "pepperoni and pineapple",
            "originalValue": "pepperoni and pineapple",
            "resolvedValues": [
                "pepperoni and pineapple"
            ]
        }
    },
    "values": [
        {
            "shape": "Scalar",
            "value": {
                "interpretedValue": "pepperoni",
                "originalValue": "pepperoni"
            }
        }
    ]
},
```

```
        "originalValue": "pepperoni",
        "resolvedValues": [
            "pepperoni"
        ]
    }
},
{
    "shape": "Scalar",
    "value": {
        "interpretedValue": "pineapple",
        "originalValue": "pineapple",
        "resolvedValues": [
            "pineapple"
        ]
    }
}
]
```

Multi-valued slots always return a list of values. When the utterance only contains one value, the list of values returned only contains one response.

Amazon Lex V2 recognizes multiple values separated by spaces, commas (,), and the conjunction "and". Multi-value slots work with both text and voice input.

You can use multi-valued slots in prompts. For example, you can set the confirmation prompt for an intent to

```
Would you like me to order your {toppings} pizza?
```

When Amazon Lex V2 sends the prompt to the user, it sends "Would you like me to order your pepperoni and pineapple pizza?"

Multi-valued slots support single default values. If multiple default values are provided, Amazon Lex V2 populates the slot with only the first available value. For more information, see [Using default slot values \(p. 230\)](#).

You can use slot obfuscation to mask the values of a multi-value slot in conversation logs. When you obfuscate slot values, the value of each of the slot values is replaced with the name of the slot. For more information, see [Obscuring slot values in logs \(p. 286\)](#).

## Using an AWS Lambda function

This section describes how to attach a Lambda function to a bot alias and the structure of the event data that Amazon Lex V2 provides to a Lambda function. Use this information to parse the input to your Lambda code. It also explains the format of the response that Amazon Lex V2 expects your Lambda function to return.

Amazon Lex V2 uses one Lambda function per bot alias per language instead of one Lambda function for each intent. To use an individual function for each intent, the Lambda function router section provides a function that you can use.

You can use Lambda functions at the following points in a conversation with a user:

- Before the conversation starts. For example, after telling the user the recognized intent.

- After eliciting a slot value from the user. For example, after the user tells the bot the size of pizza they want to order.
- Between each retry for eliciting a slot. For example, if the customer doesn't use a recognized pizza size.
- When confirming an intent. For example, when confirming a pizza order.
- To fulfill an intent. For example, to place an order for a pizza.
- After the intent has been fulfilled. For example, to switch to an intent to order a drink.

#### Topics

- [Attaching a Lambda function to a bot alias \(p. 195\)](#)
- [Input event format \(p. 196\)](#)
- [Response format \(p. 201\)](#)
- [Lambda router function \(p. 203\)](#)

## Attaching a Lambda function to a bot alias

With Amazon Lex V2 you indicate that an intent should use a Lambda function for each intent. But you assign the Lambda function that your intents use for each language supported by a bot alias. That way you can create a Lambda function tailored to each language that your bot alias supports.

You indicate that an intent should use a Lambda function using the console or the [CreateIntent](#) or [UpdateIntent](#) operation. In the intent editor, you turn on Lambda functions in the advanced options in the **Dialog code hook** section for each response in the editor.

There are two ways to control how Amazon Lex V2 calls the code hook for a response. You can mark the code hook active or inactive, and you can mark it enabled or disabled.

When a code hook is active, Amazon Lex V2 will call the code hook. When the code hook is inactive, Amazon Lex V2 does not run the code hook.

You can only enable or disable a code hook when it is marked active. When it is marked enabled, the code hook is run normally. When it is disabled, the code hook is not called and Amazon Lex V2 acts as if the code hook returned successfully.

The screenshot shows the 'Dialog code hook' section of the Intent Editor. It includes an 'Info' link, a note about enabling Lambda functions, and a 'Lambda dialog code hook' configuration section. This section contains a toggle for 'Invoke Lambda for user request validation' (set to 'No') and an 'Advanced options' button. Below the configuration is a note about configuring success, failure, and timeout responses.

**Dialog code hook** [Info](#)

You can enable Lambda functions to manage initialize the conversation.

▼ Lambda dialog code hook

*Invoke Lambda for user request validation: No*

[Invoke Lambda for user request validation](#)

[Advanced options](#)

Configure success, failure, and timeout responses for dialog code hook.

You define the Lambda function to use in each bot alias. The same Lambda function is used for all intents in a language supported by the bot.

### To choose a Lambda function to use with a bot alias

1. Open the Amazon Lex console at <https://console.aws.amazon.com/lexv2/>.
2. From the list of bots, choose the name of the bot that you want to use.
3. From **Create versions and aliases for deployment**, choose **View aliases**.
4. From the list of aliases, choose the name of the alias that you want to use.
5. From the list of supported languages, choose the language that the Lambda function is used for.
6. Choose the name of the Lambda function to use, then choose the version or alias of the function.
7. Choose **Save** to save your changes.

## Input event format

The following is the general format of an Amazon Lex V2 event that is passed to a Lambda function. Use this information when you're writing your Lambda function.

### Note

The input format may change without a corresponding change to the messageVersion. Your code shouldn't throw an error if new fields are present.

```
{  
    "messageVersion": "1.0",  
    "invocationSource": "DialogCodeHook | FulfillmentCodeHook",  
    "inputMode": "DTMF | Speech | Text",  
    "responseContentType": "CustomPayload | ImageResponseCard | PlainText | SSML",  
    "sessionId": "string",  
    "inputTranscript": "string",  
    "invocationLabel": "string",  
    "bot": {  
        "id": "string",  
        "name": "string",  
        "aliasId": "string",  
        "localeId": "string",  
        "version": "string"  
    },  
    "interpretations": [  
        {  
            "intent": {  
                "confirmationState": "Confirmed | Denied | None",  
                "name": "string",  
                "slots": {  
                    "string": {  
                        "value": {  
                            "interpretedValue": "string",  
                            "originalValue": "string",  
                            "resolvedValues": [  
                                "string"  
                            ]  
                        }  
                    }  
                }  
            },  
            "string": {  
                "shape": "List",  
                "value": {  
                    "interpretedValue": "string",  
                    "originalValue": "string",  
                    "resolvedValues": [  
                        "string"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```

                "string"
            ]
        },
        "values": [
            {
                "shape": "Scalar",
                "value": {
                    "originalValue": "string",
                    "interpretedValue": "string",
                    "resolvedValues": [
                        "string"
                    ]
                }
            },
            {
                "shape": "Scalar",
                "value": {
                    "originalValue": "string",
                    "interpretedValue": "string",
                    "resolvedValues": [
                        "string"
                    ]
                }
            }
        ]
    },
    "state": "Failed | Fulfilled | FulfillmentInProgress | InProgress |
ReadyForFulfillment | Waiting",
    "kendraResponse": {
        // Only present when intent is KendraSearchIntent. For details, see
        // https://docs.aws.amazon.com/kendra/latest/dg/
API_Query.html#API_Query_ResponseSyntax
    }
},
    "nluConfidence": number,
    "sentimentResponse": {
        "sentiment": "string",
        "sentimentScore": {
            "mixed": number,
            "negative": number,
            "neutral": number,
            "positive": number
        }
    }
}
],
"proposedNextState": {
    "dialogAction": {
        "slotToElicit": "string",
        "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"
    },
    "intent": {
        "name": "string",
        "confirmationState": "Confirmed | Denied | None",
        "slots": {},
        "state": "Failed | Fulfilled | InProgress | ReadyForFulfillment | Waiting"
    },
    "prompt": {
        "attempt": "string"
    }
},
"requestAttributes": {
    "string": "string"
},
"sessionState": {

```

```

"activeContexts": [
    {
        "name": "string",
        "contextAttributes": {
            "string": "string"
        },
        "timeToLive": {
            "timeToLiveInSeconds": number,
            "turnsToLive": number
        }
    }
],
"sessionAttributes": {
    "string": "string"
},
"runtimeHints": {
    "slotHints": {
        "string": {
            "string": {
                "runtimeHintValues": [
                    {
                        "phrase": "string"
                    },
                    {
                        "phrase": "string"
                    }
                ]
            }
        }
    }
},
"dialogAction": {
    "slotToElicit": "string",
    "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"
},
"intent": {
    "confirmationState": "Confirmed | Denied | None",
    "name": "string",
    "slots": {
        "string": {
            "value": {
                "interpretedValue": "string",
                "originalValue": "string",
                "resolvedValues": [
                    "string"
                ]
            }
        }
    },
    "string": {
        "shape": "List",
        "value": {
            "interpretedValue": "string",
            "originalValue": "string",
            "resolvedValues": [
                "string"
            ]
        }
    },
    "values": [
        {
            "shape": "Scalar",
            "value": {
                "originalValue": "string",
                "interpretedValue": "string",
                "resolvedValues": [
                    "string"
                ]
            }
        }
    ]
}
]

```



Note the following additional information about the event fields:

- **messageVersion** – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function.

**Note**

You configure this value when you define an intent. In the current implementation, only message version 1.0 is supported. Therefore, the console assumes the default value of 1.0 and doesn't show the message version.

- **invocationSource** – Indicates the action that called the Lambda function. When the source is DialogCodeHook, the Lambda function was called after input from the user. When the source is FulfillmentCodeHook the Lambda function was called after all required slots have been filled and the intent is ready for fulfillment.
- **inputMode** – The type of user utterance. Valid values are Speech, DTMF, or Text.
- **responseContentType** – Determines whether the bot responds to user input with text or speech.
- **sessionId** – Session identifier used for the conversation.
- **inputTranscript** – The text that was used to process the input from the user. For text or DTMF input, this is the text that the user typed. For speech input, this is the text that was recognized from the speech.
- **invocationLabel** – A value that indicates the response that invoked the Lambda function. You can set invocation labels for the initial response, slots, and confirmation response.
- **bot** – Information about the bot that processed the request.
  - **id** – The identifier assigned to the bot when you created it. You can see the bot ID in the Amazon Lex V2 console on the bot **Settings** page.
  - **name** – The name that you gave the bot when you created it.
  - **aliasId** – The identifier assigned to the bot alias when you created it. You can see the bot alias ID in the Amazon Lex V2 console on the **Aliases** page. If you can't see the alias ID in the list, choose the gear icon on the upper right and turn on **Alias ID**.
  - **localeId** – The identifier of the locale that you used for your bot. For a list of locales, see [Languages and locales supported by Amazon Lex V2 \(p. 4\)](#).
  - **version** – The version of the bot that processed the request.
- **interpretations** – One or more intents that Amazon Lex V2 considers possible matches to the user's utterance. For more information, see [Interpretation](#).
- **proposedNextState** – The predicted next state of the dialog between the user and the bot if the Lambda function sets the dialogAction of the sessionState to Delegate. If you override the dialog behavior in sessionState, the next state depends on the settings that you return from your Lambda function. You can use the information to modify your Lambda function's behavior based on what Amazon Lex V2 proposes as the next action. Note that this structure is only present when the invocationSource field is DialogCodeHook and when the predicted dialog action is ElicitSlot.
  - **dialogAction** – Contains the slotToElicit and the type of action that Amazon Lex V2 proposes in the next turn.
  - **intent** – The intent that the bot has determined that the user is trying to fulfill. For more information, see [Intent](#).
  - **prompt** – The prompt that the bot uses to elicit the slot.
  - **requestAttributes** – Request-specific attributes that the client sends in the request. Use request attributes to pass information that doesn't need to persist for the entire session.
  - **sessionState** – The current state of the conversation between the user and your Amazon Lex V2 bot. For more information about the session state, see [SessionState](#).
  - **dialogAction** – Determines the type of action that Amazon Lex V2 should take in response to the Lambda function. This field is always blank for now.

- **transcriptions** – one or more transcriptions that Amazon Lex V2 considers possible matches to the user's audio utterance. For more information, see [Using voice transcription confidence scores \(p. 240\)](#).

## Response format

Amazon Lex V2 expects a response from your Lambda function in the following format:

```
{  
    "sessionState": {  
        "activeContexts": [  
            {  
                "name": "string",  
                "contextAttributes": {  
                    "key": "value"  
                },  
                "timeToLive": {  
                    "timeToLiveInSeconds": number,  
                    "turnsToLive": number  
                }  
            }  
        ],  
        "sessionAttributes": {  
            "string": "string"  
        },  
        "runtimeHints": {  
            "slotHints": {  
                "string": {  
                    "string": {  
                        "runtimeHintValues": [  
                            {  
                                "phrase": "string"  
                            },  
                            {  
                                "phrase": "string"  
                            }  
                        ]  
                    }  
                }  
            }  
        }  
    },  
    "dialogAction": {  
        "slotElicitationStyle": "Default | SpellByLetter | SpellByWord",  
        "slotToElicit": "string",  
        "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot"  
    },  
    "intent": {  
        "confirmationState": "Confirmed | Denied | None",  
        "name": "string",  
        "slots": {  
            "string": {  
                "value": {  
                    "interpretedValue": "string",  
                    "originalValue": "string",  
                    "resolvedValues": [  
                        "string"  
                    ]  
                }  
            },  
            "string": {  
                "shape": "List",  
                "value": {  
                    "list": [  
                        "string"  
                    ]  
                }  
            }  
        }  
    }  
}
```

```

        "originalValue": "string",
        "interpretedValue": "string",
        "resolvedValues": [
            "string"
        ]
    },
    "values": [
        {
            "shape": "Scalar",
            "value": {
                "originalValue": "string",
                "interpretedValue": "string",
                "resolvedValues": [
                    "string"
                ]
            }
        },
        {
            "shape": "Scalar",
            "value": {
                "originalValue": "string",
                "interpretedValue": "string",
                "resolvedValues": [
                    "string"
                ]
            }
        }
    ]
},
{
    "state": "Failed | Fulfilled | FulfillmentInProgress | InProgress |
ReadyForFulfillment | Waiting"
}
],
"messages": [
{
    "contentType": "CustomPayload | ImageResponseCard | PlainText | SSML",
    "content": "string",
    "imageResponseCard": {
        "title": "string",
        "subtitle": "string",
        "imageUrl": "string",
        "buttons": [
            {
                "text": "string",
                "value": "string"
            }
        ]
    }
},
{
    "requestAttributes": {
        "string": "string"
    }
}
]

```

Note the following additional information about the response fields:

- **sessionState** – Required. The current state of the conversation with the user. The actual contents of the structure depends on the type of dialog action. For more information, see [SessionState](#).
- **activeContexts** – Contains information about the contexts that a user is using in a session. For more information, see [ActiveContext](#) and [Setting intent context](#).

- **sessionAttributes** – A map of key/value pairs representing session-specific context information. For more information, see [Setting session attributes](#).
- **runtimeHints** – Provides hints to the phrases that a customer is likely to use for a slot. For more information, see [RuntimeHints](#).
- **dialogAction** – Required. Determines the type of action that Amazon Lex V2 should take in response to the Lambda function.
  - **slotElicitationStyle** – Determines whether the slot uses spell-by-letter or spell-by-word style. For more information, see [Using spelling styles to capture slot values](#).
  - **slotToElicit** – Required only when dialogAction.type is ElicitSlot. Defines the slot to elicit from the user.
  - **type** – Required. Defines the action that the bot should execute.
- **intent** – The name of the intent that Amazon Lex V2 should use. Not required when dialogAction.type is Delegate or ElicitIntent.
- **state** – Required. The state can only be ReadyForFulfillment if dialogAction.type is Delegate.
- **messages** – Required if dialogAction.type is ElicitIntent. One or more messages that Amazon Lex V2 shows to the customer to perform the next turn of the conversation. If you don't supply messages, Amazon Lex V2 uses the appropriate message defined when the bot was created. For more information, see the [Message](#) data type.
  - **contentType** – The type of message to use.
  - **content** – If the message type is PlainText, CustomPayload, or SSML, the content field contains the message to send to the user.
  - **imageResponseCard** – If the message type is ImageResponseCard, contains the definition of the response card to show to the user. For more information, see the [ImageResponseCard](#) data type.
- **requestAttributes** – Request-specific attributes that the client sent in the request.

## Lambda router function

When you build a bot with the Amazon Lex V2 APIs, there is only one Lambda function per bot alias per language instead of a Lambda function for each intent. To use separate functions, you can create a router function that activates a separate function for each intent. The following is a router function that you can use or modify for your application.

```
import os
import json
import boto3

# reuse client connection as global
client = boto3.client('lambda')

def router(event):
    intent_name = event['sessionState']['intent']['name']
    fn_name = os.environ.get(intent_name)
    print(f"Intent: {intent_name} -> Lambda: {fn_name}")
    if (fn_name):
        # invoke lambda and return result
        invoke_response = client.invoke(FunctionName=fn_name, Payload = json.dumps(event))
        print(invoke_response)
        payload = json.load(invoke_response['Payload'])
        return payload
    raise Exception('No environment variable for intent: ' + intent_name)

def lambda_handler(event, context):
    print(event)
    response = router(event)
```

```
    return response
```

# Using the Automated Chatbot Designer

The Automated Chatbot Designer helps you design bots from existing conversation transcripts. It analyzes the transcripts and suggests an initial design with intents and slot types. You can iterate on the bot design, add prompts, build, test, and deploy the bot.

After you create a new bot or add a language to your bot using the Amazon Lex V2 console or API, you can upload transcripts of conversations between two parties. The automated chatbot designer analyzes the transcripts and determines the intents and slot types for the bot. It also labels the conversations that influenced the creation of a particular intent or slot type for your review.

You use the Amazon Lex V2 console or the API to analyze conversation transcripts and suggest intents and slot types for a bot.

You can review the suggested intents and slot types after the chatbot designer finishes the analysis. After you've added a suggested intent or slot type, you can modify it or delete it from the bot design using the console or the API.

The automated chatbot designer supports conversation transcript files using the Contact Lens for Amazon Connect schema. If you are using a different contact center application, you must transform the conversation transcripts to the format used by the chatbot designer. For information, see [Input transcript format \(p. 209\)](#).

To use the automated chatbot designer, you must allow the IAM role that is running the designer access. For the specific IAM policy, see [Allow users to use the Automated Chatbot Designer \(p. 327\)](#). To enable Amazon Lex V2 to encrypt output data with an optional AWS KMS key, you need to update the key with the policy shown in [Allow users to use a AWS KMS key to encrypt and decrypt files \(p. 328\)](#).

**Note**

If you use a KMS key, you must provide a **KMS key policy**, regardless of the IAM role used.

You can only analyze conversations in the English (US) language.

**Topics**

- [Importing conversation transcripts \(p. 205\)](#)
- [Creating intents and slot types \(p. 208\)](#)
- [Input transcript format \(p. 209\)](#)
- [Output transcript format \(p. 209\)](#)

## Importing conversation transcripts

Importing conversation transcripts is a three-step process:

1. Prepare the transcripts for importing by converting them to the correct format. If you are using Contact Lens for Amazon Connect the transcripts are already in the correct format.
2. Upload the transcripts to an Amazon S3 bucket. If you are using Contact Lens, your transcripts are already in an S3 bucket.
3. Analyze the transcripts using the Amazon Lex V2 console or API operations. The time that it takes to complete training depends on the volume of transcripts and the complexity of the conversation. Typically, 500 lines of transcripts are analyzed every minute.

Each of these steps is described in the following sections.

## Importing transcripts from Contact Lens for Amazon Connect

The Amazon Lex V2 automated chatbot designer is compatible with Contact Lens transcript files. To use Contact Lens transcript files, you must turn on Contact Lens and note the location of its output files.

### To export transcripts from Contact Lens

1. Turn on Contact Lens in your Amazon Connect instance. For instructions, see [Enable Contact Lens for Amazon Connect](#) in the *Amazon Connect administrator guide*.
2. Note the location of the S3 bucket that Amazon Connect is using for your instance. To see the location, open the **Data storage** page in the Amazon Connect console. For instructions, see [Update instance settings](#) in the *Amazon Connect administrator guide*.

After you have turned on Contact Lens and noted the location of your transcript files, go to [Analyze your transcripts using Amazon Lex V2 console \(p. 207\)](#) for instructions to import and analyze your transcripts.

## Prepare transcripts

Prepare your transcripts by creating transcript files.

- Create one transcript file per conversation listing the interaction between the parties. Each interaction in the conversation can span multiple lines. You can provide both redacted and non-redacted versions of the conversation. Each interaction in the conversation can span multiple lines.
- The file must be in the JSON format specified in [Input transcript format \(p. 209\)](#).
- You must provide at least 1,000 conversational turns. To improve the discovery of your intents and slot types, you should provide around 10,000 or more conversational turns. The automated chatbot designer will only process the first 700,000 turns.
- There is no limit to the number of transcript files that you can upload, nor is there a file size restriction.

If you plan to filter the transcripts that you import by date, the files must be in the following directory structure:

```
<path or bucket root>
  --> yyyy
    --> mm
      --> dd
        --> transcript files
```

The transcript file must contain the date in the format "yyyy-mm-dd" somewhere in the file name.

### To export transcripts from other contact center applications

1. Use your contact center application's tools to export conversations. The conversation must contain at least the information specified in [Input transcript format \(p. 209\)](#).
2. Transform the transcripts produced by your contact center application to the format described in [Input transcript format \(p. 209\)](#). You are responsible for performing the transformation.

We provide three scripts for preparing transcripts. They are:

- A script to combine Contact Lens transcripts with Amazon Lex V2 conversation logs. Contact Lens transcripts don't include parts of Amazon Connect conversations that interact with Amazon Lex V2 bots. The script requires conversation logs to be turned on for Amazon Lex V2, and appropriate permissions to query conversation log CloudWatch Logs and Contact Lens S3 buckets.
- A script to transform Amazon Transcribe call analytics to the Amazon Lex V2 input format.
- A script to transform Amazon Connect chat transcripts to the Amazon Lex V2 input format.

You can download the scripts from this GitHub repository: <https://github.com/aws-samples/amazon-lex-bot-recommendation-integration>.

## Upload your transcripts to an S3 bucket

If you are using Contact Lens, your transcript files are already contained in an S3 bucket. For the location and file names of your transcript files, see [Example Contact Lens output files](#) in the *Amazon Connect administrator guide*.

If you are using another contact center application and you have not set up an S3 bucket for your transcript files, follow this procedure. Otherwise, if you have an existing S3 bucket, after logging in to the Amazon S3 console, follow this procedure starting with step 5.

### To upload files to an S3 bucket

1. Sign in to the AWS Management Console and open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
2. Choose **Create bucket**.
3. Give the bucket a name and choose a Region. The Region must be the same one that you use for Amazon Lex V2. Set the other options as required for your use case.
4. Choose **Create bucket**.
5. From the list of buckets, choose an existing bucket or the bucket that you just created
6. Choose **Upload**.
7. Add the transcript files that you want to upload.
8. Choose **Upload**.

## Analyze your transcripts using Amazon Lex V2 console

You can only use automated bot design in an empty language. You can add a new language to an existing bot, or create a new bot.

### To create a new language in a new bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose **Create bot**
3. Choose **Start with Automated Chatbot Designer**. Fill out the information to create your new bot.
4. Choose **Next**
5. In **Add language to bot** fill out the information for the language.
6. In the **Transcript file location on S3** section, choose the S3 bucket that contains your transcript files and the local path to the files if necessary.
7. You can optionally choose the following:

- A AWS KMS key to encrypt the transcript data during processing. If you don't select a key, a service AWS KMS key is used.
  - To filter the transcripts to a specific date range. If you choose to filter the transcripts, they must be in the correct folder structure. For more information, see [Prepare transcripts \(p. 206\)](#).
8. Choose **Done**.

Wait for Amazon Lex V2 to process the transcript. You see a completion message when the analysis is complete.

#### How to stop analyzing your transcript

In case you need to stop the analysis of the transcripts you have uploaded, you can stop a running BotRecommendation job, which has a BotRecommendationStatus status as processing. You can click on the **Stop processing** button present on the banner after submitting a job from the console or by using CLI SDK for the StopBotRecommendation API. For more information, see [StopBotRecommendation](#)

After calling the StopBotRecommendation, the internal BotRecommendationStatus is set to Stopping and you are not charged. To make sure the job has stopped, you can call the DescribeBotRecommendation API and verify that the BotRecommendationStatus is Stopped. This usually takes 3-4 minutes.

You are not charged for the processing after the StopBotRecommendation API is called.

## Creating intents and slot types

After the chatbot designer creates intents and slot types, you select the intents and slot types to add to your bot. You can review the details of each intent and slot type to help you decide which recommendations are the most relevant to your use case.

You can click on a recommended intent's name to view the sample utterances and slots that the chatbot designer has suggested. If you select **Show associated transcripts**, you can also scroll through the conversations that you provided. These transcripts influence the chatbot designer's recommendation of this intent. If you click on a sample utterance, you can review the primary conversation and the relevant turn of dialog, which influenced that specific utterance.

You can click on a specific slot type's name to view the slot values that have been recommended. If you select **Show associated transcripts**, you can review the conversations that influenced this slot type, with the agent prompt that elicits for the slot type highlighted. If you click on a specific slot type value, you can review the primary conversation and the relevant turn of dialog that influenced this value.

#### To review and add intents and slot type

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the list of bots, choose the bot you want to work with.
3. Choose **View languages**.
4. From the list of languages, choose the language to work with.
5. In **Conversation structure**, choose **Review**.
6. In the list of intents and slot types, choose the ones to add to the bot. You can choose an intent or slot type to see details and the associated transcripts.

Intents are sorted by the confidence that Amazon Lex V2 has that the intent is associated with the processed transcripts.

## Input transcript format

The following is the input file format for generating intents and slot types for your bot. The input file must contain these fields. Other fields are ignored.

The input format is compatible with the output format from Contact Lens for Amazon Connect. If you are using Contact Lens, you don't need to modify your transcript files. For more information, see [Example Contact Lens output files](#). If you are using another contact center application, you must transform your transcript file to this format.

```
{  
    "Participants": [  
        {  
            "ParticipantId": "string",  
            "ParticipantRole": "AGENT | CUSTOMER"  
        }  
    ],  
    "Version": "1.1.0",  
    "ContentMetadata": {  
        "RedactionTypes": [  
            "PII"  
        ],  
        "Output": "Raw | Redacted"  
    },  
    "CustomerMetadata": {  
        "ContactId": "string"  
    },  
    "Transcript": [  
        {  
            "ParticipantId": "string",  
            "Id": "string",  
            "Content": "string"  
        }  
    ]  
}
```

The following fields must be present in the input file:

- **Participants** Identifies the participants in the conversation and the role that they play.
- **Version** The version of the input file format. Always "1.1.0".
- **ContentMetadata** Indicates whether you removed sensitive information from the transcript. Set the Output field to "Raw" if the transcript contains sensitive information.
- **CustomerMetadata** A unique identifier for the conversation.
- **Transcript** The text of the conversation between parties in the conversation. Each turn of the conversation is identified with a unique identifier.

## Output transcript format

The output transcript format is nearly the same as the input transcript format. However it also includes some customer metadata and a field listing segments that influenced the suggestion of intents and slot types. You can download the output transcript from the **Review** page in the console or using the Amazon Lex V2 API. For more information, see [Input transcript format \(p. 209\)](#).

```
{  
    "Participants": [  
        {  
            "ParticipantId": "string",  
            "ParticipantRole": "AGENT | CUSTOMER"  
        }  
    ],  
    "Version": "1.1.0",  
    "ContentMetadata": {  
        "RedactionTypes": [  
            "PII"  
        ],  
        "Output": "Raw | Redacted"  
    },  
    "CustomerMetadata": {  
        "ContactId": "string"  
    },  
    "Transcript": [  
        {  
            "ParticipantId": "string",  
            "Id": "string",  
            "Content": "string"  
        }  
    ]  
}
```

```

        "ParticipantId": "string",
        "ParticipantRole": "AGENT" | CUSTOMER"
    }
],
"Version": "1.1.0",
"ContentMetadata": {

    "RedactionTypes": [
        "PII"
    ],
    "Output": "Raw" | Redacted"
},
"CustomerMetadata": {
    "ContactId": "string",
    "FileName": "string",
    "InputFormat": "Lex"
},
"InfluencingSegments": [
{
    "Id": "string",
    "StartTurnIndex": number,
    "EndTurnIndex": number,
    "Intents": [
        {
            "Id": "string",
            "Name": "string",
            "SampleUtteranceIndex": [
                {
                    "Index": number,
                    "Content": "String"
                }
            ]
        }
    ],
    "SlotTypes": [
        {
            "Id": "string",
            "Name": "string",
            "SlotValueIndex": [
                {
                    "Index": number,
                    "Content": "String"
                }
            ]
        }
    ]
}
],
"Transcript": [
{
    "ParticipantId": "string",
    "Id": "string",
    "Content": "string"
}
]
}

```

- **CustomerMetadata** – There are two fields added to the CustomerMetadata field, the name of the input file that contains the conversation and the input format, which is always "Lex".
- **InfluencingSegments** – Identifies the segments of the conversation that influenced the suggestion of an intent or slot type. The ID of the intent or slot type identifies the specific one influenced by the conversation.

# Deploying bots

This section provides examples of deploying Amazon Lex V2 bots on messaging platforms, mobile applications, and Web sites.

## Topics

- [Aliases \(p. 211\)](#)
- [Using a Java application to interact with an Amazon Lex V2 bot \(p. 212\)](#)

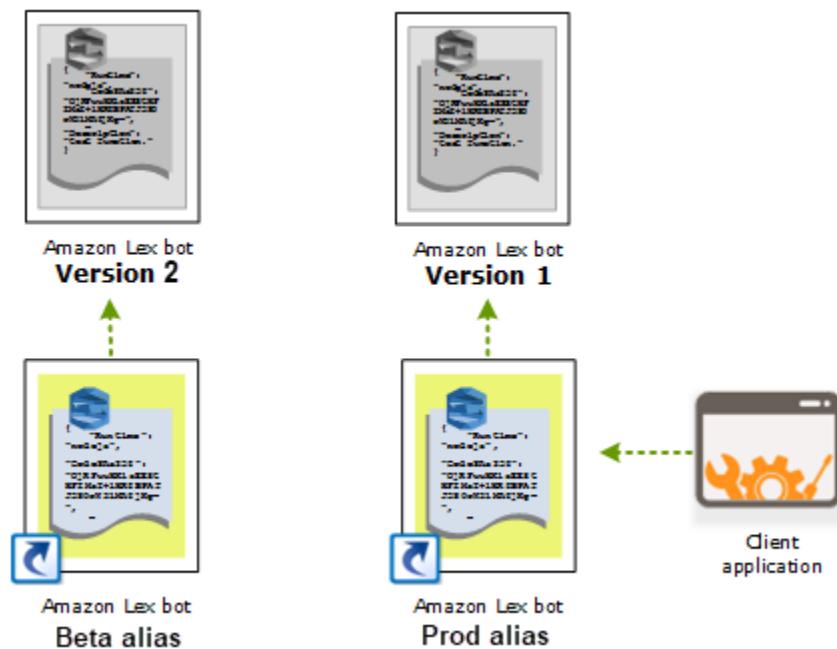
## Aliases

Amazon Lex V2 bots support aliases. An *alias* is a pointer to a specific version of a bot. With an alias, you can easily update the version that your client applications are using. For example, you can point an alias to version 1 of your bot. When you are ready to update the bot, you publish version 2 and change the alias to point to the new version. Because your applications use the alias instead of a specific version, all of your clients get the new functionality without needing to be updated.

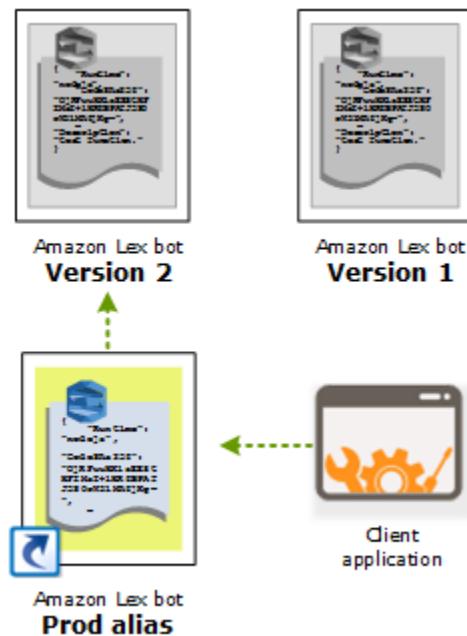
An alias is a pointer to a specific version of an Amazon Lex V2 bot. Use an alias to allow client applications to use a specific version of the bot without requiring the application to track which version that is.

When you create a bot, Amazon Lex V2 creates an alias called `TestBotAlias` that you can use for testing your bot. The `TestBotAlias` alias is always associated with the `Draft` version of your bot. You should only use the `TestBotAlias` alias for testing. Amazon Lex V2 limits the number of runtime requests that you can make to the alias.

The following example shows two versions of an Amazon Lex V2 bot, version `version 1` and `version 2`. Each of these bot versions has an associated alias, `BETA` and `PROD`, respectively. Client applications use the `PROD` alias to access the bot.



When you create a second version of the bot, you can update the alias to point to the new version of the bot using the console or the [UpdateBotAlias](#) operation. When you change the alias, all of your client applications use the new version. If there is a problem with the new version, you can roll back to the previous version by simply changing the alias to point to that version.



#### Note

Although you can test the Draft version of a bot in the console, we recommend that when you integrate a bot with your client application, you first publish a version and create an alias that points to that version. Use the alias in your client application for the reasons explained in this section. When you update an alias, Amazon Lex V2 will use the current version for all in-progress sessions. New sessions use the new version.

## Using a Java application to interact with an Amazon Lex V2 bot

The AWS SDK for Java provides an interface that you can use from your Java applications to interact with your bots. Use the SDK for Java to build client applications for users.

The following application interacts with the OrderFlowers bot that you created in [Exercise 1: Create a bot from an example \(p. 9\)](#). It uses the `LexRuntimeV2Client` from the SDK for Java to call the `RecognizeText` operation to conduct a conversation with the bot.

The output from the conversation looks like this:

```
User : I would like to order flowers
Bot : What type of flowers would you like to order?
User : 1 dozen roses
Bot : What day do you want the dozen roses to be picked up?
User : Next Monday
Bot : At what time do you want the dozen roses to be picked up?
User : 5 in the evening
Bot : Okay, your dozen roses will be ready for pickup by 17:00 on 2021-01-04. Does this
sound okay?
```

```
User : Yes
Bot   : Thanks.
```

For the JSON structures that are sent between the client application and the Amazon Lex V2 bot, see [Exercise 2: Review the conversation flow \(p. 10\)](#).

To run the application, you must provide the following information:

- **botId** – The identifier assigned to the bot when you created it. You can see the bot ID in the Amazon Lex V2 console on the bot **Settings** page.
- **botAliasId** – The identifier assigned to the bot alias when you created it. You can see the bot alias ID in the Amazon Lex V2 console on the **Aliases** page. If you can't see the alias ID in the list, choose the gear icon on the upper right and turn on **Alias ID**.
- **localeId** – The identifier of the locale that you used for your bot. For a list of locales, see [Languages and locales supported by Amazon Lex V2 \(p. 4\)](#).
- **accessKey** and **secretKey** – The authentication keys for your account. If you don't have a set of keys, create them using the AWS Identity and Access Management console.
- **sessionId** – An identifier for the session with the Amazon Lex V2 bot. In this case, the code uses a random UUID.
- **region** – If your bot is not in the US East (N. Virginia) Region, make sure that you change the Region.

The applications uses a function called `getRecognizeTextRequest` to create individual requests to the bot. The function builds a request with the required parameters to send to Amazon Lex V2.

```
package com.lex.recognizetext.sample;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lexruntimev2.LexRuntimeV2Client;
import software.amazon.awssdk.services.lexruntimev2.model.RecognizeTextRequest;
import software.amazon.awssdk.services.lexruntimev2.model.RecognizeTextResponse;

import java.net.URISyntaxException;
import java.util.UUID;

/**
 * This is a sample application to interact with a bot using RecognizeText API.
 */
public class OrderFlowersSampleApplication {

    public static void main(String[] args) throws URISyntaxException, InterruptedException {
        String botId = "";
        String botAliasId = "";
        String localeId = "en_US";
        String accessKey = "";
        String secretKey = "";
        String sessionId = UUID.randomUUID().toString();
        Region region = Region.US_EAST_1; // pick an appropriate region

        AwsBasicCredentials awsCreds = AwsBasicCredentials.create(accessKey, secretKey);
        AwsCredentialsProvider awsCredentialsProvider =
            StaticCredentialsProvider.create(awsCreds);

        LexRuntimeV2Client lexV2Client = LexRuntimeV2Client
            .builder()
```

```

        .credentialsProvider(awsCredentialsProvider)
        .region(region)
        .build();

    // utterance 1
    String userInput = "I would like to order flowers";
    RecognizeTextRequest recognizeTextRequest = getRecognizeTextRequest(botId,
botAliasId, localeId, sessionId, userInput);
    RecognizeTextResponse recognizeTextResponse =
lexV2Client.recognizeText(recognizeTextRequest);

    System.out.println("User : " + userInput);
    recognizeTextResponse.messages().forEach(message -> {
        System.out.println("Bot : " + message.content());
    });

    // utterance 2
    userInput = "1 dozen roses";
    recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
    recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

    System.out.println("User : " + userInput);
    recognizeTextResponse.messages().forEach(message -> {
        System.out.println("Bot : " + message.content());
    });

    // utterance 3
    userInput = "next monday";
    recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
    recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

    System.out.println("User : " + userInput);
    recognizeTextResponse.messages().forEach(message -> {
        System.out.println("Bot : " + message.content());
    });

    // utterance 4
    userInput = "5 in evening";
    recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
    recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

    System.out.println("User : " + userInput);
    recognizeTextResponse.messages().forEach(message -> {
        System.out.println("Bot : " + message.content());
    });

    // utterance 5
    userInput = "Yes";
    recognizeTextRequest = getRecognizeTextRequest(botId, botAliasId, localeId,
sessionId, userInput);
    recognizeTextResponse = lexV2Client.recognizeText(recognizeTextRequest);

    System.out.println("User : " + userInput);
    recognizeTextResponse.messages().forEach(message -> {
        System.out.println("Bot : " + message.content());
    });
}

private static RecognizeTextRequest getRecognizeTextRequest(String botId, String
botAliasId, String localeId, String sessionId, String userInput) {
    RecognizeTextRequest recognizeTextRequest = RecognizeTextRequest.builder()
        .botAliasId(botAliasId)
        .botId(botId)
}

```

```
        .localeId(localeId)
        .sessionId(sessionId)
        .text(userInput)
        .build();
    return recognizeTextRequest;
}
```

# Integrating your bots

This section provides information about integrating your Amazon Lex V2 bots with messaging platforms and contact center applications.

## Topics

- [Integrating an Amazon Lex V2 bot with a messaging platform \(p. 216\)](#)
- [Integrating an Amazon Lex V2 bot with a contact center \(p. 223\)](#)

## Integrating an Amazon Lex V2 bot with a messaging platform

This section explains how to integrate Amazon Lex V2 bots with the Facebook, Slack, and Twilio messaging platforms.

### Note

When storing your Facebook, Slack, or Twilio configurations, Amazon Lex V2 uses an AWS KMS key to encrypt information. The first time that you create a channel to one of these messaging platforms, Amazon Lex V2 creates a default customer managed key (aws/lex) in your AWS account or you can select your own customer managed key. Amazon Lex V2 supports only symmetric keys. For more information, see the [AWS Key Management Service Developer Guide](#).

When a messaging platform sends a request to Amazon Lex V2 it includes platform-specific information as a request attribute to your Lambda function. Use this attribute to customize the way that your bot behaves. For more information, see [Setting request attributes \(p. 231\)](#).

### Common request attribute

Attribute	Description
x-amz-lex:channels:platform	One of the following values: <ul style="list-style-type: none"><li>• Facebook</li><li>• Slack</li><li>• Twilio</li></ul>

## Integrating an Amazon Lex V2 bot with Facebook Messenger

You can host your Amazon Lex V2 bot in Facebook Messenger. When you do, Facebook users can interact with your bot to fulfill intents.

Before you start, you need to sign up for a Facebook developer account at <https://developers.facebook.com>.

You need to perform the following steps:

## Topics

- Step 1: Create an Amazon Lex V2 bot (p. 217)
- Step 2: Create a Facebook application (p. 217)
- Step 3: Integrate Facebook Messenger with the Amazon Lex V2 bot (p. 217)
- Step 4: Complete Facebook integration (p. 218)
- Step 5: Test the integration (p. 218)

## Step 1: Create an Amazon Lex V2 bot

If you don't already have an Amazon Lex V2 bot, create one. In this topic, we assume that you are using the bot that you created in [Exercise 1: Create a bot from an example \(p. 9\)](#). However, you can use any bot.

### Next step

[Step 2: Create a Facebook application \(p. 217\)](#)

## Step 2: Create a Facebook application

On the Facebook developer portal, create a Facebook application and a Facebook page.

### To create a Facebook application

1. Open <https://developers.facebook.com/apps>
2. Choose **Create App**.
3. In the **Create an App** page, choose **Business**, then choose **Next**.
4. For the **Add on app name**, **App contact email**, and **Business Account** fields, make the appropriate choices for your app. Choose **Create App** to continue.
5. From **Add Products to Your App**, choose **Set Up** from the **Messenger** tile.
6. In the **Access Tokens** section, choose **Add or Remove pages**.
7. Choose a page to use with your app, then choose **Next**.
8. For **What is app allowed to do**, leave the defaults then choose **Done**.
9. On the confirmation page, choose **OK**.
10. In the **Access Tokens** section, choose **Generate Token**, then copy the token. You enter this token in the Amazon Lex V2 console.
11. From the left menu, choose **Settings** and then choose **Basic**.
12. For **App Secret**, choose **Show** and then copy the secret. You enter this token in the Amazon Lex V2 console.

### Next step

[Step 3: Integrate Facebook Messenger with the Amazon Lex V2 bot \(p. 217\)](#)

## Step 3: Integrate Facebook Messenger with the Amazon Lex V2 bot

In this step you link your Amazon Lex V2 bot with Facebook.

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the list of bots, choose the Amazon Lex V2 bot that you created in step 1.
3. In the left menu, choose **Channel integrations** and then choose **Add channel**.

4. In **Create channel**, do the following:
  - a. For **Platform**, choose **Facebook**.
  - b. For **Identity policies**, choose the AWS KMS key to protect channel information. The default key is provided by Amazon Lex V2.
  - c. For **Integration configuration**, give the channel a name and an optional description. Choose the alias that points to the version of the bot to use, and choose the language that the channel supports.
  - d. For **Additional configuration**, enter the following:
    - **Alias** – A string that identifies the app that is calling Amazon Lex V2. You can use any string. Record this string, you enter it in the Facebook developer console.
    - **Page access token** – The page access token that you copied from the Facebook developer console.
    - **App secret key** – The secret key that you copied from the Facebook developer console.
  - e. Choose **Create**
  - f. Amazon Lex V2 shows the list of channels for your bot. From the list, choose the channel that you just created.
  - g. From **Callback URL**, record the callback URL. You enter this URL in the Facebook developer console.

## Next step

[Step 4: Complete Facebook integration \(p. 218\)](#)

## Step 4: Complete Facebook integration

In this step, use the Facebook developer console to complete integration with Amazon Lex V2.

### To complete Facebook Messenger integration

1. Open <https://developers.facebook.com/apps>
2. From the list of apps, choose the app that you are integrating with Facebook Messenger.
3. In the left menu, choose **Messenger**, then choose **Settings**.
4. In the **Webhooks** section:
  - a. Choose **Add Callback URL**.
  - b. In **Edit Callback URL**, enter the following:
    - **Callback URL** – Enter the callback URL that you recorded from the Amazon Lex V2 console.
    - **Verify Token** – Enter the alias that you entered in the Amazon Lex V2 console.
  - c. Choose **Verify and Save**.
  - d. Choose **Add subsrciptions** under **Webhooks** next to your page.
  - e. In the window that pops up, choose messages and then click **Save**.

## Next step

[Step 5: Test the integration \(p. 218\)](#)

## Step 5: Test the integration

You can now start a conversation from Facebook Messenger with your Amazon Lex V2 bot.

### To test the integration between Facebook Messenger and an Amazon Lex V2 bot

1. Open the Facebook page that you associated with your bot in step 2.
2. In the Messenger window, use the test utterances provided in [Exercise 1: Create a bot from an example \(p. 9\)](#).

## Integrating an Amazon Lex V2 bot with Slack

This topic provides instructions for integrating an Amazon Lex V2 bot with the Slack messaging application. You perform the following steps:

### Topics

- [Step 1: Create an Amazon Lex V2 bot \(p. 219\)](#)
- [Step 2: Sign up for Slack and create a Slack team \(p. 219\)](#)
- [Step 3: Create a Slack application \(p. 219\)](#)
- [Step 4: Integrate the Slack application with the Amazon Lex V2 bot \(p. 220\)](#)
- [Step 5: Complete Slack integration \(p. 221\)](#)
- [Step 6: Test the integration \(p. 222\)](#)

### Step 1: Create an Amazon Lex V2 bot

If you don't already have an Amazon Lex V2 bot, create one. In this topic, we assume that you are using the bot that you created in [Exercise 1: Create a bot from an example \(p. 9\)](#). However, you can use any bot.

#### Next step

[Step 2: Sign up for Slack and create a Slack team \(p. 219\)](#)

### Step 2: Sign up for Slack and create a Slack team

Sign up for a Slack account and create a Slack team. For instructions, see [Using Slack](#). In the next section you create a Slack application, which any Slack team can install.

#### Next step

[Step 3: Create a Slack application \(p. 219\)](#)

### Step 3: Create a Slack application

In this section, you do the following:

1. Create a Slack application in the Slack API Console.
2. Configure the application to add interactive messaging to your bot.

At the end of this section, you get application credentials (Client ID, Client Secret, and Verification Token). In the next step, you use this information to integrate the bot in the Amazon Lex V2 console.

#### To create a Slack application

1. Sign in to the Slack API Console at <https://api.slack.com> .

2. Create an application.

After you have successfully created the application, Slack displays the **Basic Information** page for the application.

3. Configure the application features as follows:

- In the left menu, choose **Interactivity & Shortcuts**.
  - Choose the toggle to turn interactive components on.
  - In the **Request URL** box, specify any valid URL. For example, you can use **https://slack.com**.

**Note**

For now, enter any valid URL to get the verification token that you need in the next step. You will update this URL after you add the bot channel association in the Amazon Lex console.

- Choose **Save Changes**.

4. In the left menu, in **Settings**, choose **Basic Information**. Record the following application credentials:

- Client ID
- Client Secret
- Verification Token

## Next step

[Step 4: Integrate the Slack application with the Amazon Lex V2 bot \(p. 220\)](#)

## Step 4: Integrate the Slack application with the Amazon Lex V2 bot

### To integrate the Slack application with your Amazon Lex V2 bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the list of bots, choose the Amazon Lex V2 bot that you created in step 1.
3. In the left menu, choose **Channel integrations** and then choose **Add channel**.
4. In **Create channel**, do the following:
  - a. For **Platform**, choose **Slack**.
  - b. For **Identity policies**, choose the AWS KMS key to protect channel information. The default key is provided by Amazon Lex V2.
  - c. For **Integration configuration**, give the channel a name and an optional description. Choose the alias that points to the version of the bot to use, and choose the language that the channel supports.
  - d. For **Additional configuration**, enter the following:
    - **Client ID** – enter the client ID from Slack.
    - **Client secret** – enter the client secret from Slack.
    - **Verification token** – enter the verification token from Slack.
    - **Success page URL** – The URL of the page that Slack should open when the user is authenticated. Typically you leave this blank.
5. Choose **Create** to create the channel.

6. Amazon Lex V2 shows the list of channels for your bot. From the list, choose the channel that you just created.
7. From **Callback URL**, record the endpoint and the OAuth endpoint.

## Next step

[Step 5: Complete Slack integration \(p. 221\)](#)

## Step 5: Complete Slack integration

In this section, use the Slack API console to complete integration with the Slack application.

### To complete Slack application integration

1. Sign in to the Slack API console at <https://api.slack.com>. Choose the app that you created in [Step 3: Create a Slack application \(p. 219\)](#).
2. Update the **OAuth & Permissions** feature as follows:
  - a. In the left menu, choose **OAuth & Permissions**.
  - b. In the **Redirect URLs** section, add the OAuth endpoint that Amazon Lex provided in the preceding step. Choose **Add**, and then choose **Save URLs**.
  - c. In the **Bot Token Scopes** section, add two permissions with the **Add an OAuth Scope** button. Filter the list with the following text:
    - **chat:write**
    - **team:read**
3. Update the **Interactivity & Shortcuts** feature by updating the **Request URL** value to the endpoint that Amazon Lex provided in the preceding step. Enter the endpoint that you saved in step 4, and then choose **Save Changes**.
4. Subscribe to the **Event Subscriptions** feature as follows:
  - Enable events by choosing the **On** option.
  - Set the **Request URL** value to the endpoint that Amazon Lex provided in the preceding step.
  - In the **Subscribe to Bot Events** section, select **Add Bot User Event** and add the **message.im** bot event to enable direct messaging between the end user and the Slack bot.
  - Save the changes.
5. Enable sending messages from the messages tab as follows:
  - From the left menu, choose **App Home**.
  - In the **Show Tabs** section, choose **Allow users to send Slash commands and messages from the messages tab**.
6. Choose **Manage Distribution** under **Settings**. Choose **Add to Slack** to install the application. If you are authenticated to multiple workspaces, first choose the correct workspace in the upper right-hand corner from the drop-down list. Next, select **Allow** to authorize the bot to respond to messages.

### Note

If you make any changes to your Slack application settings later, you must redo this substep.

## Next step

[Step 6: Test the integration \(p. 222\)](#)

## Step 6: Test the integration

Now use a browser window to test the integration of Slack with your Amazon Lex V2 bot.

### To test your Slack application

1. Launch Slack. From the left menu, in the **Direct Messages** section, choose your bot. If you don't see your bot, choose the plus icon (+) next to **Direct Messages** to search for it.
2. Engage in a chat with your Slack application. Your bot responds to messages.

If you created the bot using Getting started exercise 1, you can use the example conversations from that exercise.

## Integrating an Amazon Lex V2 bot with Twilio SMS

This topic provides instructions for integrating an Amazon Lex V2 bot with the Twilio simple message service (SMS). You perform the following steps:

### Topics

- [Step 1: Create an Amazon Lex V2 bot \(p. 222\)](#)
- [Step 2: Create a Twilio SMS account \(p. 222\)](#)
- [Step 3: Integrate the Twilio message service endpoint with the Amazon Lex V2 bot \(p. 222\)](#)
- [Step 4: Complete Twilio integration \(p. 223\)](#)
- [Step 5: Test the integration \(p. 223\)](#)

## Step 1: Create an Amazon Lex V2 bot

If you don't already have an Amazon Lex V2 bot, create one. In this topic, we assume that you are using the bot that you created in [Exercise 1: Create a bot from an example \(p. 9\)](#). However, you can use any bot.

### Next step

[Step 2: Create a Twilio SMS account \(p. 222\)](#)

## Step 2: Create a Twilio SMS account

Sign up for a Twilio account and record the following account information:

- **ACCOUNT SID**
- **AUTH TOKEN**

For sign-up instructions, see <https://www.twilio.com/console>.

### Next step

[Step 3: Integrate the Twilio message service endpoint with the Amazon Lex V2 bot \(p. 222\)](#)

## Step 3: Integrate the Twilio message service endpoint with the Amazon Lex V2 bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.

2. From the list of bots, choose the Amazon Lex V2 bot that you created in step 1.
3. In the left menu, choose **Channel integrations** and then choose **Add channel**.
4. In **Create channel**, do the following:
  - a. For **Platform**, choose **Twilio**.
  - b. For **Identity policies**, choose the AWS KMS key to protect channel information. The default key is provided by Amazon Lex V2.
  - c. For **Integration configuration**, give the channel a name and an optional description. Choose the alias that points to the version of the bot to use, and choose the language that the channel supports.
  - d. For **Additional configuration**, enter the account SID and authentication token from the Twilio dashboard.
5. Choose **Create**.
6. From the list of channels, choose the channel that you just created.
7. Copy the **Callback URL**.

### Next step

[Step 4: Complete Twilio integration \(p. 223\)](#)

## Step 4: Complete Twilio integration

Use the Twilio console to complete the integration of your Amazon Lex V2 bot with Twilio SMS.

1. Open the Twilio console at <https://www.twilio.com/console> .
2. From the left menu, choose **All Products & Services**, then choose **Phone Number**.
3. If you have a phone number, choose it. If you don't have a phone number, choose **Buy a Number** to get one.
4. In the **Messaging** section, in **A MESSAGE COMES IN**, enter the callback URL from the Amazon Lex V2 console.
5. Choose **Save**.

### Next step

[Step 5: Test the integration \(p. 223\)](#)

## Step 5: Test the integration

Use your mobile phone to test the integration between Twilio SMS and your bot. Using your mobile phone, send messages to the Twilio number.

If you created the bot using Getting started exercise 1, you can use the example conversations from that exercise.

# Integrating an Amazon Lex V2 bot with a contact center

You can integrate Amazon Lex V2 bots with your contact centers to enable self-service use-cases using the Amazon Lex V2 streaming API. Use these bots as interactive voice response (IVR) agents on

telephony or as a text-based chatbot integrated into your contact center. For more information about the streaming APIs, see [Streaming to an Amazon Lex V2 bot \(p. 253\)](#).

With streaming APIs, you can enable the following features:

- **Interruptions** ("barge-in") – Callers can interrupt the bot and answer a question before the prompt is complete. For more information, see [Enabling your bot to be interrupted by your user \(p. 270\)](#).
- **Wait and continue** – Callers can instruct the bot to wait if they need time for retrieving additional information during a call, such as a credit card number or a booking ID. For more information, see [Enabling the bot to wait for the user to provide more information \(p. 271\)](#).
- **DTMF support** – Callers can provide information via speech or DTMF interchangeably.
- **SSML support** – You can configure Amazon Lex V2 bot prompts using SSML tags for greater control over speech generation from text. For more information, see [Generating speech from SSML documents](#) in the *Amazon Polly developer guide*.
- **Configurable timeouts** – You can configure how long to wait for customers to finish speaking before Amazon Lex V2 collects their speech input, such as answering a yes or no question, or providing a date or credit card number. For more information, see [Configuring timeouts for capturing user input \(p. 274\)](#).

## Amazon Chime SDK

Use the Amazon Chime SDK to add real-time audio, video, screen sharing, and messaging capabilities to your web or mobile applications. The Amazon Chime SDK provides public switched telephone network (PSTN) audio service so that you can build custom telephony applications with an AWS Lambda function.

Amazon Chime PSTN audio is integrated with Amazon Lex V2. You can use this integration to access Amazon Lex V2 bots as interactive voice response (IVR) systems in contact centers for audio interactions. Use this to integrate Amazon Lex V2 using PSTN audio services in the following scenarios.

**Contact center integrations**—You can use the Amazon Chime Voice Connector and Amazon Chime SDK PSTN audio service to access Amazon Lex V2 bots. Use them in any contact center application that uses the session initiation protocol (SIP) for voice communications. This integration adds natural language voice conversation experiences to your existing on-premises or cloud-based contact center with SIP support. For a list of supported contact center platforms, see [Amazon Chime voice connector resources](#).

The following diagram shows the integration between a contact center using SIP and Amazon Lex V2.



**Direct telephony support**—You can build customized IVR solutions to directly access Amazon Lex V2 bots using a phone number provisioned in the Amazon Chime SDK.

For more information, see the following topics in the *Amazon Chime SDK guide*.

- [SIP integration using an Amazon Chime voice connector](#)
- [Using the Amazon Chime SDK PSTN audio service](#)
- [Integrating Amazon Chime PSTN audio with Amazon Lex V2](#)

When the Amazon Chime SDK sends a request to Amazon Lex V2, it includes platform-specific information to your Lambda function and conversation logs. Use this information to determine the contact center application that is sending traffic to your bot.

Common request attribute	Value
x-amz-lex-channels-platform	Amazon Chime SDK PSTN Audio

## Amazon Connect

Amazon Connect is an omnichannel cloud contact center. You can set up a contact center in a few steps, add agents located anywhere, and start engaging with your customers. For more information, see [Get started with Amazon Connect](#) in the *Amazon Connect administrator guide*.

You can create personalized experiences for your customers using omnichannel communications. For example, you can offer chat and voice contact based on customer preference and estimated wait times. Meanwhile agents can handle all customers from just one interface. For example, they can chat with customers, and create or respond to tasks as they are routed to them.

You can use Amazon Connect for audio interactions with your customers, or Amazon Connect Chat for text-only interactions.

For more information, see the following topics in the *Amazon Connect administrator guide*.

- [What is Amazon Connect](#)
- [Add an Amazon Lex V2 bot](#)
- [Amazon Connect get customer input contact block](#)

When a contact center sends a request to Amazon Lex V2, it includes platform-specific information as a request attribute to your Lambda function and conversation logs. Use this information to determine which contact center application is sending traffic to your bot.

### Common request attribute

Attribute	Value
x-amz-lex:channels:platform	One of the following values: <ul style="list-style-type: none"><li>• Connect</li><li>• Connect Chat</li></ul>

## Genesys Cloud

Genesys Cloud is a suite of cloud services for enterprise communication, collaboration, and contact center management. Genesys Cloud is built on top of AWS and uses a distributed cloud environment that provides secure access to organizations around the world.

For more information, see the following pages on the Genesys Cloud website.

- [About Genesys Cloud contact center](#)
- [About the Amazon Lex V2 integration](#)

When a contact center sends a request to Amazon Lex V2 it includes platform-specific information as a request attribute to your Lambda function and conversation logs. Use this information to determine which contact center application is sending traffic to your bot.

### Common request attribute

Attribute	Value
x-amz-lex:channels:platform	<ul style="list-style-type: none"><li>• Genesys Cloud</li></ul>

*Learn more*

- [Power your contact center with Amazon Lex and Genesys Cloud](#)

# Using bots

## Topics

- [Managing conversations \(p. 227\)](#)
- [Managing conversation context \(p. 228\)](#)
- [Managing sessions with the Amazon Lex V2 API \(p. 234\)](#)
- [Analyzing the sentiment of user utterances \(p. 236\)](#)
- [Using confidence scores \(p. 237\)](#)
- [Using runtime hints to improve recognition of slot values \(p. 246\)](#)
- [Using spelling styles to capture slot values \(p. 248\)](#)

## Managing conversations

After you build a bot, you integrate your client application with the Amazon Lex V2 runtime operations to hold conversations with your bot.

When a user starts a conversation with your bot, Amazon Lex V2 creates a *session*. A session encapsulates the information exchanged between your application and the bot. For more information, see [Managing sessions with the Amazon Lex V2 API \(p. 234\)](#).

A typical conversation involves a back and forth flow between the user and a bot. For example:

```
User : I'd like to make an appointment
Bot : What type of appointment would you like to schedule?
User : dental
Bot : When should I schedule your dental appointment?
User : Tomorrow
Bot : At what time do you want to schedule the dental appointment on 2021-01-01?
User : 9 am
Bot : 09:00 is available, should I go ahead and book your appointment?
User : Yes
Bot : Thank you. Your appointment has been set successfully.
```

When you use the [RecognizeText](#) or [RecognizeUtterance](#) operation, you must manage the conversation in your client application. When you use the [StartConversation](#) operation, Amazon Lex V2 manages the conversation for you.

To manage the conversation, you must send user utterances to the bot until the conversation reaches a logical end. The current conversation is captured in session state. The session state is updated after each user utterance. The session state contains the current state of the conversation and is returned by the bot in a response to each user utterance.

A conversation can be in any of the following states:

- **ElicitIntent** – Indicates that the bot has not yet determined the user's intent.
- **ElicitSlot** – Indicates that the bot has detected the user's intent and is gathering the required information to fulfill the intent.
- **ConfirmIntent** – Indicates that the bot is waiting for the user to confirm that the information collected is correct.

- **Closed** – Indicates that the user's intent is complete and that the conversation with the bot reached a logical end.

A user can specify a new intent after the first intent is completed. For more information, see [Managing conversation context \(p. 228\)](#).

An intent can have the one of the following states:

- **InProgress** – Indicates that the bot is gathering information necessary to complete the intent. This is in conjunction with the ElicitSlot conversation state.
- **Waiting** – Indicates that the user requested the bot to wait when the bot asked for information for a specific slot.
- **Fulfilled** – Indicates that the business logic in a Lambda function associated with the intent ran successfully.
- **ReadyForFulfillment** – Indicates that the bot gathered all of the information required to fulfill the intent and that the client application can run fulfillment business logic.
- **Failed** – Indicates that an intent has failed.

## Managing conversation context

*Conversation context* is information that the user, your client application, or a Lambda function provides to a Amazon Lex bot to fulfill and intent. Conversation context includes slot data that the user provides, request attributes set by the client application, and session attributes that the client application and Lambda functions create.

### Topics

- [Setting intent context \(p. 228\)](#)
- [Using default slot values \(p. 230\)](#)
- [Setting session attributes \(p. 230\)](#)
- [Setting request attributes \(p. 231\)](#)
- [Setting the session timeout \(p. 232\)](#)
- [Sharing information between intents \(p. 232\)](#)
- [Setting complex attributes \(p. 233\)](#)

## Setting intent context

You can have Amazon Lex trigger intents based on *context*. A *context* is a state variable that can be associated with an intent when you define a bot. You configure the contexts for an intent when you create the intent using the console or using the [CreateIntent](#) operation. You can only use context in the English (US) (en-US) locale.

There are two types of relationships for contexts, output contexts and input contexts. An *output context* becomes active when an associated intent is fulfilled. An output context is returned to your application in the response from the [RecognizeText](#) or [RecognizeUtterance](#) operation, and it is set for the current session. After a context is activated, it stays active for the number of turns or time limit configured when the context was defined.

An *input context* specifies conditions under which an intent can be recognized. An intent can only be recognized during a conversation when all of its input contexts are active. An intent with no input contexts is always eligible for recognition.

Amazon Lex automatically manages the lifecycle of contexts that are activated by fulfilling intents with output contexts. You can also set active contexts in a call to the `RecognizeText` or `RecognizeUtterance` operation.

You can also set the context of a conversation using the Lambda function for the intent. The output context from Amazon Lex is sent to the Lambda function input event. The Lambda function can send contexts in its response. For more information, see [Using an AWS Lambda function \(p. 194\)](#).

For example, suppose you have an intent to book a rental car that is configured to return an output context called "book\_car\_fulfilled". When the intent is fulfilled, Amazon Lex sets the output context variable "book\_car\_fulfilled". Since "book\_car\_fulfilled" is an active context, an intent with the "book\_car\_fulfilled" context set as an input context is now considered for recognition, as long as a user utterance is recognized as an attempt to elicit that intent. You can use this for intents that only make sense after booking a car, such as emailing a receipt or modifying a reservation.

## Output context

Amazon Lex makes an intent's output contexts active when the intent is fulfilled. You can use the output context to control the intents eligible to follow up the current intent.

Each context has a list of parameters that are maintained in the session. The parameters are the slot values for the fulfilled intent. You can use these parameters to pre-populate slot values for other intents. For more information, see [Using default slot values \(p. 230\)](#).

You configure the output context when you create an intent with the console or with the `CreateIntent` operation. You can configure an intent with more than one output context. When the intent is fulfilled, all of the output contexts are activated and returned in the `RecognizeText` or `RecognizeUtterance` response.

When you define an output context you also define its *time to live*, the length of time or number of turns that the context is included in responses from Amazon Lex. A *turn* is one request from your application to Amazon Lex. Once the number of turns or the time has expired, the context is no longer active.

Your application can use the output context as needed. For example, your application can use the output context to:

- Change the behavior of the application based on the context. For example, a travel application could have a different action for the context "book\_car\_fulfilled" than "rental\_hotel\_fulfilled."
- Return the output context to Amazon Lex as the input context for the next utterance. If Amazon Lex recognizes the utterance as an attempt to elicit an intent, it uses the context to limit the intents that can be returned to ones with the specified context.

## Input context

You set an input context to limit the points in the conversation where the intent is recognized. Intents without an input context are always eligible to be recognized.

You set the input contexts that an intent responds to using the console or the `CreateIntent` operation. An intent can have more than one input context.

For an intent with more than one input context, all contexts must be active to trigger the intent. You can set an input context when you call the `RecognizeText`, `RecognizeUtterance`, or `PutSession` operation.

You can configure the slots in an intent to take default values from the current active context. Default values are used when Amazon Lex recognizes a new intent but doesn't receive a slot value. You specify the context name and slot name in the form `#context-name.parameter-name` when you define the slot. For more information, see [Using default slot values \(p. 230\)](#).

## Using default slot values

When you use a default value, you specify a source for a slot value to be filled for new intents when no slot is provided by the user's input. This source can be previous dialog, request or session attributes, or a fixed value that you set at build-time.

You can use the following as the source for your default values.

- Previous dialog (contexts) – #context-name.parameter-name
- Session attributes – [attribute-name]
- Request attributes – <attribute-name>
- Fixed value – Any value that doesn't match the previous

When you use the [CreateIntent](#) operation to add slots to an intent, you can add a list of default values. Default values are used in the order that they are listed. For example, suppose you have an intent with a slot with the following definition:

```
"slots": [
  {
    "botId": "string",
    "defaultValueSpec": {
      "defaultValueList": [
        {
          "defaultValue": "#book-car-fulfilled.startDate"
        },
        {
          "defaultValue": "[reservationStartDate]"
        }
      ]
    },
    Other slot configuration settings
  }
]
```

When the intent is recognized, the slot named "reservation-start-date" has its value set to one of the following.

1. If the "book-car-fulfilled" context is active, the value of the "startDate" parameter is used as the default value.
2. If the "book-car-fulfilled" context is not active, or if the "startDate" parameter is not set, the value of the "reservationStartDate" session attribute is used as the default value.
3. If neither of the first two default values are used, then the slot doesn't have a default value and Amazon Lex will elicit a value as usual.

If a default value is used for the slot, the slot is not elicited even if it is required.

## Setting session attributes

*Session attributes* contain application-specific information that is passed between a bot and a client application during a session. Amazon Lex passes session attributes to all Lambda functions configured for a bot. If a Lambda function adds or updates session attributes, Amazon Lex passes the new information back to the client application.

Use session attributes in your Lambda functions to initialize a bot and to customize prompts and response cards. For example:

- Initialization — In a pizza ordering bot, the client application passes the user's location as a session attribute in the first call to the [RecognizeText](#) or [RecognizeUtterance](#) operation. For example, "Location": "111 Maple Street". The Lambda function uses this information to find the closest pizzeria to place the order.
- Personalize prompts — Configure prompts and response cards to refer to session attributes. For example, "Hey [FirstName], what toppings would you like?" If you pass the user's first name as a session attribute ({"FirstName": "Vivian"}), Amazon Lex substitutes the name for the placeholder. It then sends a personalized prompt to the user, "Hey Vivian, which toppings would you like?"

Session attributes persist for the duration of the session. Amazon Lex stores them in an encrypted data store until the session ends. The client can create session attributes in a request by calling either the [RecognizeText](#) or [RecognizeUtterance](#) operation with the `sessionAttributes` field set to a value. A Lambda function can create a session attribute in a response. After the client or a Lambda function creates a session attribute, the stored attribute value is used any time that the client application doesn't include `sessionAttribute` field in a request to Amazon Lex.

For example, suppose you have two session attributes, {"x": "1", "y": "2"}. If the client calls the [RecognizeText](#) or [RecognizeUtterance](#) operation without specifying the `sessionAttributes` field, Amazon Lex calls the Lambda function with the stored session attributes ({"x": 1, "y": 2}). If the Lambda function doesn't return session attributes, Amazon Lex returns the stored session attributes to the client application.

If either the client application or a Lambda function passes session attributes, Amazon Lex updates the stored session attributes. Passing an existing value, such as {"x": 2}, updates the stored value. If you pass a new set of session attributes, such as {"z": 3}, the existing values are removed and only the new value is kept. When an empty map, {}, is passed, stored values are erased.

To send session attributes to Amazon Lex, you create a string-to-string map of the attributes. The following shows how to map session attributes:

```
{  
    "attributeName": "attributeValue",  
    "attributeName": "attributeValue"  
}
```

For the [RecognizeText](#) operation, you insert the map into the body of the request using the `sessionAttributes` field of the `sessionState` structure, as follows:

```
"sessionState": {  
    "sessionAttributes": {  
        "attributeName": "attributeValue",  
        "attributeName": "attributeValue"  
    }  
}
```

For the [PostContent](#) operation, you base64 encode the map, and then send it as part of the `x-amz-lex-session-state` header.

If you are sending binary or structured data in a session attribute, you must first transform the data to a simple string. For more information, see [Setting complex attributes \(p. 233\)](#).

## Setting request attributes

*Request attributes* contain request-specific information and apply only to the current request. A client application sends this information to Amazon Lex. Use request attributes to pass information that doesn't need to persist for the entire session. You can create your own request attributes or you can

use predefined attributes. To send request attributes, use the `x-amz-lex-request-attributes` header in a `RecognizeUtterance` or the `requestAttributes` field in a `RecognizeText` request. Because request attributes don't persist across requests like session attributes do, they are not returned in `RecognizeUtterance` or `RecognizeText` responses.

**Note**

To send information that persists across requests, use session attributes.

## Setting user-defined request attributes

A *user-defined request attribute* is data that you send to your bot in each request. You send the information in the `amz-lex-request-attributes` header of a `RecognizeUtterance` request or in the `requestAttributes` field of a `RecognizeText` request.

To send request attributes to Amazon Lex, you create a string-to-string map of the attributes. The following shows how to map request attributes:

```
{  
    "attributeName": "attributeValue",  
    "attributeName": "attributeValue"  
}
```

For the `PostText` operation, you insert the map into the body of the request using the `requestAttributes` field, as follows:

```
"requestAttributes": {  
    "attributeName": "attributeValue",  
    "attributeName": "attributeValue"  
}
```

For the `PostContent` operation, you base64 encode the map, and then send it as the `x-amz-lex-request-attributes` header.

If you are sending binary or structured data in a request attribute, you must first transform the data to a simple string. For more information, see [Setting complex attributes \(p. 233\)](#).

## Setting the session timeout

Amazon Lex retains context information—slot data and session attributes—until a conversation session ends. To control how long a session lasts for a bot, set the session timeout. By default, session duration is 5 minutes, but you can specify any duration between 0 and 1,440 minutes (24 hours).

For example, suppose that you create a `ShoeOrdering` bot that supports intents such as `OrderShoes` and `GetOrderStatus`. When Amazon Lex detects that the user's intent is to order shoes, it asks for slot data. For example, it asks for shoe size, color, brand, etc. If the user provides some of the slot data but doesn't complete the shoe purchase, Amazon Lex remembers all of the slot data and session attributes for the entire session. If the user returns to the session before it expires, they can provide the remaining slot data, and complete the purchase.

In the Amazon Lex console, you set the session timeout when you create a bot. With the AWS command line interface (AWS CLI) or API, you set the timeout when you create a bot with the `CreateBot` operation by setting the `idleSessionTTLInSeconds` field.

## Sharing information between intents

Amazon Lex supports sharing information between intents. To share between intents, use output contexts or session attributes.

To use output contexts, you define an output context when you create or update an intent. When the intent is fulfilled, responses from Amazon Lex V2 contain the context and slot values from the intent as context parameters. You can use these parameters as default values in subsequent intents or in your application code or Lambda functions.

To use session attributes, you set the attributes in your Lambda or application code. For example, a user of the ShoeOrdering bot starts by ordering shoes. The bot engages in a conversation with the user, gathering slot data, such as shoe size, color, and brand. When the user places an order, the Lambda function that fulfills the order sets the `orderNumber` session attribute, which contains the order number. To get the status of the order, the user uses the `GetOrderStatus` intent. The bot can ask the user for slot data, such as order number and order date. When the bot has the required information, it returns the status of the order.

If you think that your users might switch intents during the same session, you can design your bot to return the status of the latest order. Instead of asking the user for order information again, you use the `orderNumber` session attribute to share information across intents and fulfill the `GetOrderStatus` intent. The bot does this by returning the status of the last order that the user placed.

## Setting complex attributes

Session and request attributes are string-to-string maps of attributes and values. In many cases, you can use the string map to transfer attribute values between your client application and a bot. In some cases, however, you might need to transfer binary data or a complex structure that can't be easily converted to a string map. For example, the following JSON object represents an array of the three most populous cities in the United States:

```
{  
  "cities": [  
    {  
      "city": {  
        "name": "New York",  
        "state": "New York",  
        "pop": "8537673"  
      }  
    },  
    {  
      "city": {  
        "name": "Los Angeles",  
        "state": "California",  
        "pop": "3976322"  
      }  
    },  
    {  
      "city": {  
        "name": "Chicago",  
        "state": "Illinois",  
        "pop": "2704958"  
      }  
    }  
  ]  
}
```

This array of data doesn't translate well to a string-to-string map. In such a case, you can transform an object to a simple string so that you can send it to your bot with the [RecognizeText](#) and [RecognizeUtterance](#) operations.

For example, if you are using JavaScript, you can use the `JSON.stringify` operation to convert an object to JSON, and the `JSON.parse` operation to convert JSON text to a JavaScript object:

```
// To convert an object to a string.  
var jsonString = JSON.stringify(object, null, 2);  
// To convert a string to an object.  
var obj = JSON.parse(JSON string);
```

To send attributes with the `RecognizeUtterance` operation, you must base64 encode the attributes before you add them to the request header, as shown in the following JavaScript code:

```
var encodedAttributes = new Buffer(attributeString).toString("base64");
```

You can send binary data to the `RecognizeText` and `RecognizeUtterance` operations by first converting the data to a base64-encoded string, and then sending the string as the value in the session attributes:

```
"sessionAttributes" : {  
    "binaryData": "base64 encoded data"  
}
```

## Managing sessions with the Amazon Lex V2 API

When a user starts a conversation with your bot, Amazon Lex V2 creates a *session*. The information exchanged between your application and Amazon Lex V2 makes up the session state for the conversation. When you make a request, the session is identified by an identifier that you specify. For more information about the session identifier, see the `sessionId` field in the [RecognizeText](#) or [RecognizeUtterance](#) operation.

You can modify the session state sent between your application and your bot. For example, you can create and modify session attributes that contain custom information about the session, and you can change the flow of the conversation by setting the dialog context to interpret the next utterance.

There are three ways that you can update session state.

- Pass the session information inline as part of a call to the `RecognizeText` or `RecognizeUtterance` operation.
- Use a Lambda function with the `RecognizeText` or `RecognizeUtterance` operation that is called after each turn of the conversation. For more information, see [Using an AWS Lambda function \(p. 194\)](#). The other is to use the Amazon Lex V2 runtime API in your application to make changes to the session state.
- Use operations that enable you to manage session information for a conversation with your bot. The operations are the `PutSession` operation, the `GetSession` operation, and the `DeleteSession` operation. You use these operations to get information about the state of your user's session with your bot, and to have fine-grained control over the state.

Use the `GetSession` operation when you want to get the current state of the session. The operation returns the current state of the session, including the state of the dialog with your user, any session attributes that have been set and slot values for the current intent and any other intents that Amazon Lex V2 has identified as possible intents that match the user's utterance.

The `PutSession` operation enables you to directly manipulate the current session state. You can set the session, including the type of dialog action that the bot will perform next and the messages that Amazon Lex V2 sends to the user. This gives you control over the flow of the conversation with the bot.

Set the dialog action type field to `Delegate` to have Amazon Lex V2 determine the next action for the bot.

You can use the `PutSession` operation to create a new session with a bot and set the intent that the bot should start with. You can also use the `PutSession` operation to change from one intent to another. When you create a session or change the intent you also can set session state, such as slot values and session attributes. When the new intent is finished, you have the option of restarting the prior intent.

The response from the `PutSession` operation contains the same information as the `RecognizeUtterance` operation. You can use this information to prompt the user for the next piece of information, just as you would with the response from the `RecognizeUtterance` operation.

Use the `DeleteSession` operation to remove an existing session and start over with a new session. For example, when you are testing your bot you can use the `DeleteSession` operation to remove test sessions from your bot.

The session operations work with your fulfillment Lambda functions. For example, if your Lambda function returns `Failed` as the fulfillment state you can use the `PutSession` operation to set the dialog action type to `close` and `fulfillmentState` to `ReadyForFulfillment` to retry the fulfillment step.

Here are some things that you can do with the session operations:

- Have the bot start a conversation instead of waiting for the user.
- Switch intents during a conversation.
- Return to a previous intent.
- Start or restart a conversation in the middle of the interaction.
- Validate slot values and have the bot re-prompt for values that are not valid.

Each of these are described further below.

## Starting a new session

If you want to have the bot start the conversation with your user, you can use the `PutSession` operation.

- Create a welcome intent with no slots and a conclusion message that prompts the user to state an intent. For example, "What would you like to order? You can say 'Order a drink' or 'Order a pizza.'"
- Call the `PutSession` operation. Set the intent name to the name of your welcome intent and set the dialog action to `Delegate`.
- Amazon Lex will respond with the prompt from your welcome intent to start the conversation with your user.

## Switching intents

You can use the `PutSession` operation to switch from one intent to another. You can also use it to switch back to a previous intent. You can use the `PutSession` operation to set session attributes or slot values for the new intent.

- Call the `PutSession` operation. Set the intent name to the name of the new intent and set the dialog action to `Delegate`. You can also set any slot values or session attributes required for the new intent.
- Amazon Lex will start a conversation with the user using the new intent.

## Resuming a prior intent

To resume a prior intent you use the `GetSession` operation to get the state of the intent, perform the needed interaction, and then use the `PutSession` operation to set the intent to its previous dialog state.

- Call the `GetSession` operation. Store the state of the intent.
- Perform another interaction, such as fulfilling a different intent.
- Using the information saved information for the previous intent, call the `PutSession` operation. This will return the user to the previous intent in the same place in the conversation.

In some cases it may be necessary to resume your user's conversation with your bot. For example, say that you have created a customer service bot. Your application determines that the user needs to talk to a customer service representative. After talking to the user, the representative can direct the conversation back to the bot with the information that they collected.

To resume a session, use steps similar to these:

- Your application determines that the user needs to speak to a customer service representative.
- Use the `GetSession` operation to get the current dialog state of the intent.
- The customer service representative talks to the user and resolves the issue.
- Use the `PutSession` operation to set the dialog state of the intent. This may include setting slot values, setting session attributes, or changing the intent.
- The bot resumes the conversation with the user.

## Validating slot values

You can validate responses to your bot using your client application. If the response isn't valid, you can use the `PutSession` operation to get a new response from your user. For example, suppose that your flower ordering bot can only sell tulips, roses, and lilies. If the user orders carnations, your application can do the following:

- Examine the slot value returned from the `PostText` or `PostContent` response.
- If the slot value is not valid, call the `PutSession` operation. Your application should clear the slot value, set the `slotToElicit` field, and set the `dialogAction.type` value to `elicitSlot`. Optionally, you can set the `message` and `messageFormat` fields if you want to change the message that Amazon Lex uses to elicit the slot value.

## Analyzing the sentiment of user utterances

You can use sentiment analysis to determine the sentiments expressed in a user utterance. With the sentiment information you can manage conversation flow or perform post-call analysis. For example, if the user sentiment is negative you can create a flow to hand over a conversation to a human agent.

Amazon Lex integrates with Amazon Comprehend to detect user sentiment. The response from Amazon Comprehend indicates whether the overall sentiment of the text is positive, neutral, negative, or mixed. The response contains the most likely sentiment for the user utterance and the scores for each of the sentiment categories. The score represents the likelihood that the sentiment was correctly detected.

You enable sentiment analysis for a bot using the console or by using the Amazon Lex API. You enable sentiment analysis on an alias for the bot. On the Amazon Lex console:

1. Choose an alias.
2. In **Details**, choose **Edit**.
3. Choose **Enable sentiment analysis** to sentiment analysis on or off.
4. Choose **Confirm** to save your changes.

If you are using the API, call the [CreateBotAlias](#) operation with the detectSentiment field set to true.

When sentiment analysis is enabled, the response from the [RecognizeText](#) and [RecognizeUtterance](#) operations return a field called `sentimentResponse` in the `interpretations` structure with other metadata. The `sentimentResponse` field has two fields, `sentiment` and `sentimentScore`, that contain the result of the sentiment analysis. If you are using a Lambda function, the `sentimentResponse` field is included in the event data sent to your function.

The following is an example of the `sentimentResponse` field returned as part of the `RecognizeText` or `RecognizeUtterance` response.

```
sentimentResponse {  
    "sentimentScore": {  
        "mixed": 0.030585512690246105,  
        "positive": 0.94992071056365967,  
        "neutral": 0.0141543131828308,  
        "negative": 0.00893945890665054  
    },  
    "sentiment": "POSITIVE"  
}
```

Amazon Lex calls Amazon Comprehend on your behalf to determine the sentiment in every utterance processed by the bot. By enabling sentiment analysis, you agree to the service terms and agreements for Amazon Comprehend. For more information about pricing for Amazon Comprehend, see [Amazon Comprehend Pricing](#).

For more information about how Amazon Comprehend sentiment analysis works, see [Determine the sentiment](#) in the [Amazon Comprehend Developer Guide](#).

## Using confidence scores

There are two steps that Amazon Lex V2 uses to determine what a user says. The first, automatic speech recognition (ASR), creates a transcript of the user's audio utterance. The second, natural language understanding (NLU), determines the meaning of the user's utterance to recognize the user's intent or the value of slots.

By default, Amazon Lex V2 returns the most likely result from ASR and NLU. At times it may be difficult for Amazon Lex V2 to determine the most likely result. In that case, it returns several possible results along with a *confidence score* that indicates how likely the result is correct. A confidence score is a rating that Amazon Lex V2 provides that shows the relative confidence that it has in the result. Confidence scores range from 0.0 to 1.0.

You can use your domain knowledge with the confidence score to help determine the correct interpretation of the ASR or NLU result.

The ASR, or transcription, confidence score is a rating on how confident Amazon Lex V2 is that a particular transcription is correct. The NLU, or intent, confidence score is a rating on how confident Amazon Lex V2 is that the intent specified by the top transcription is correct. Use the confidence score that best fits your application.

## Topics

- [Using intent confidence scores \(p. 238\)](#)
- [Using voice transcription confidence scores \(p. 240\)](#)

# Using intent confidence scores

When a user makes an utterance, Amazon Lex V2 uses natural language understanding (NLU) to understand the user's request and return the proper intent. By default Amazon Lex V2 returns the most likely intent defined by your bot.

In some cases it may be difficult for Amazon Lex V2 to determine the most likely intent. For example, the user might make an ambiguous utterance, or there might be two intents that are similar. To help determine the proper intent, you can combine your domain knowledge with the *NLU confidence scores* in a list of interpretations. A confidence score is a rating that Amazon Lex V2 provides that shows how confident it is that an intent is the correct intent.

To determine the difference between two intents within an interpretation, you can compare their confidence scores. For example, if one intent has a confidence score of 0.95 and another has a score of 0.65, the first intent is probably correct. However, if one intent has a score of 0.75 and another has a score of 0.72, there is ambiguity between the two intents that you may be able to discriminate using domain knowledge in your application.

You can also use confidence scores to create test applications that determine if changes to an intent's utterances make a difference in the behavior of the bot. For example, you can get the confidence scores for a bot's intents using a set of utterances, then update the intents with new utterances. You can then check the confidence scores to see if there was an improvement.

The confidence scores that Amazon Lex V2 returns are comparative values. You should not rely on them as an absolute score. The values may change based on improvements to Amazon Lex V2.

Amazon Lex V2 returns the most likely intent and up to 4 alternative intents with their associated scores in the `interpretations` structure in each response. The following JSON code shows the `interpretations` structure in the response from the [RecognizeText](#) operation:

```
"interpretations": [  
    {  
        "intent": {  
            "confirmationState": "string",  
            "name": "string",  
            "slots": {  
                "string" : {  
                    "value": {  
                        "interpretedValue": "string",  
                        "originalValue": "string",  
                        "resolvedValues": [ "string" ]  
                    }  
                }  
            },  
            "state": "string"  
        },  
        "nluConfidence": number  
    }  
]
```

## AMAZON.FallbackIntent

Amazon Lex V2 returns AMAZON.FallbackIntent as the top intent in two situations:

1. If the confidence scores of all possible intents are less than the confidence threshold. You can use the default threshold or you can set your own threshold. If you have the AMAZON.KendraSearchIntent configured, Amazon Lex V2 returns it as well in this situation.
2. If the interpretation confidence for AMAZON.FallbackIntent is higher than the interpretation confidence of all other intents.

Note that Amazon Lex V2 does not display a confidence score for AMAZON.FallbackIntent.

## Setting and changing the confidence threshold

The confidence threshold must be a number between 0.00 and 1.00. You can set the threshold for each language in your bot in the following ways:

### Using the Amazon Lex V2 console

- To set the threshold when you add a language to your bot with **Add language**, you can insert your desired value in the **Confidence score threshold** panel.
- To update the threshold, you can select **Edit** in the **Language details** panel in a language for your bot. Then insert your desired value in the **Confidence score threshold** panel.

### Using API operations

- To set the threshold, set the `nluIntentConfidenceThreshold` parameter of the [CreateBotLocale](#) operation.
- To update the confidence threshold, set the `nluIntentConfidenceThreshold` parameter of the [UpdateBotLocale](#) operation.

## Session Management

To change the intent that Amazon Lex V2 uses in a conversation with the user, you can use the response from your dialog code hook Lambda function, or you can use the session management APIs in your custom application.

### Using a Lambda function

When you use a Lambda function, Amazon Lex V2 calls it with a JSON structure that contains the input to the function. The JSON structure contains a field called `currentIntent` that contains the intent that Amazon Lex V2 has identified as the most likely intent for the user's utterance. The JSON structure also includes an `alternativeIntents` field that contains up to four additional intents that may satisfy the user's intent. Each intent includes a field called `nluIntentConfidenceScore` that contains the confidence score that Amazon Lex V2 assigned to the intent.

To use an alternative intent, you specify it in the `ConfirmIntent` or the `ElicitSlot` dialog action in your Lambda function.

For more information, see [Using an AWS Lambda function \(p. 194\)](#).

### Using the Session Management API

To use a different intent from the current intent, use the `PutSession` operation. For example, if you decide that the first alternative is preferable to the intent that Amazon Lex V2 chose, you can use the `PutSession` operation to change intents so that the next intent that the user interacts with is the one that you selected.

For more information, see [Managing sessions with the Amazon Lex V2 API \(p. 234\)](#).

## Using voice transcription confidence scores

When a user makes a voice utterance, Amazon Lex V2 uses automatic speech recognition (ASR) to transcribe the user's request before it is interpreted. By default, Amazon Lex V2 uses the most likely transcription of the audio for interpretation.

In some cases there might be more than one possible transcription of the audio. For example, a user might make an utterance with an ambiguous sound, such as "My name is John" that might be understood as "My name is Juan." In this case, you can use disambiguation techniques or combine your domain knowledge with the *transcription confidence score* to help determine which transcription in a list of transcriptions is the correct one.

Amazon Lex V2 includes the top transcription and up to two alternate transcriptions for user input in the request to your Lambda code hook function. Each transcription contains a confidence score that it is the correct transcription. Each transcription also includes any slot values inferred from the user input.

You can compare the confidence scores of two transcriptions to determine if there is ambiguity between them. For example, if one transcription has a confidence score of 0.95 and the other has a confidence score of 0.65, the first transcription is probably correct and the ambiguity between them is low. If the two transcriptions have confidence scores of 0.75 and 0.72, the ambiguity between them is high. You may be able to discriminate between them using your domain knowledge.

For example, if the inferred slot values in two transcripts with a confidence score of 0.75 and 0.72 are "John" and "Juan", you can query the users in your database for the existence of these names and eliminate one of the transcriptions. If "John" isn't a user in your database and "Juan" is, you can use the dialog code hook to change the inferred slot value for the first name to "Juan."

The confidence scores that Amazon Lex V2 returns are comparative values. Don't rely on them as an absolute score. The values may change based on improvements to Amazon Lex V2.

Audio transcription confidence scores are available only in the English (GB) (en\_GB) and English (US) (en\_US) languages. Confidence scores are supported only for 8 kHz audio input. Transcription confidence scores aren't provided for audio input from the [test window](#) on the Amazon Lex V2 console because it uses 16 kHz audio input.

### Note

Before you can use audio transcription confidence scores with an existing bot, you must first rebuild the bot. Existing versions of a bot don't support transcription confidence scores. You must create a new version of the bot to use them.

You can use confidence scores for multiple conversation design patterns:

- If the highest confidence score falls below a threshold due to a noisy environment or poor signal quality, you can prompt the user with the same question to capture better quality audio.
- If multiple transcriptions have similar confidence scores for slot values, such as "John" and "Juan," you can compare the values with a pre-existing database to eliminate inputs, or you can prompt the user to select one of the two values. For example, "say 1 for John or say 2 for Juan."
- If your business logic requires intent switching based on specific keywords in an alternative transcript with a confidence score close to the top transcript, you can change the intent using your dialog code hook Lambda function or using session management operations. For more information, see [Session management \(p. 242\)](#).

Amazon Lex V2 sends the following JSON structure with up to three transcriptions for the user's input to your Lambda code hook function:

```
"transcriptions": [  
    {  
        "transcription": "string",
```

```

    "transcriptionConfidence": {
        "score": "number"
    },
    "resolvedContext": {
        "intent": "string"
    },
    "resolvedSlots": {
        "string": {
            "shape": "List",
            "value": {
                "originalValue": "string",
                "resolvedValues": [
                    "string"
                ]
            }
        },
        "values": [
            {
                "shape": "Scalar",
                "value": {
                    "originalValue": "string",
                    "resolvedValues": [
                        "string"
                    ]
                }
            },
            {
                "shape": "Scalar",
                "value": {
                    "originalValue": "string",
                    "resolvedValues": [
                        "string"
                    ]
                }
            }
        ]
    }
}
]

```

The JSON structure contains transcription text, the intent that was resolved for the utterance, and values for any slots detected in the utterance. For text user input, the transcriptions contain a single transcript with a confidence score of 1.0.

The contents of the transcripts depend on the turn of the conversation and the recognized intent.

For the first turn, intent elicitation, Amazon Lex V2 determines the top three transcriptions. For the top transcription, it returns the intent and any inferred slot values in the transcription.

On subsequent turns, slot elicitation, the results depend on the inferred intent for each of the transcriptions, as follows.

- If the inferred intent for the top transcript is the same as the previous turn and all other transcripts have the same intent, then
  - All transcripts contain inferred slot values.
  
- If the inferred intent for the top transcript is different from the previous turn and all other transcripts have the previous intent, then
  - The top transcript contains the inferred slot values for the new intent.
  - Other transcripts have the previous intent and inferred slot values for the previous intent.

- If the inferred intent for the top transcript is different from the previous turn, one transcript is the same as the previous intent, and one transcript is a different intent, then
  - The top transcript contains the new inferred intent and any inferred slot values in the utterance.
  - The transcript that has the previous inferred intent contains inferred slot values for that intent.
  - The transcript with the different intent has no inferred intent name and no inferred slot values.
- If the inferred intent for the top transcript is the different from the previous turn and all other transcripts have different intents, then
  - The top transcript contains the new inferred intent and any inferred slot values in the utterance.
  - Other transcripts contain no inferred intents and no inferred slot values.
- If the inferred intent for the top two transcripts is the same and different from the previous turn, and the third transcript is a different intent, then
  - The top two transcripts contain the new inferred intent and any inferred slot values in the utterance.
  - The third transcript has no intent name and no resolved slot values.

## Session management

To change the intent that Amazon Lex V2 uses in a conversation with the user, use the response from your dialog code hook Lambda function. Or you can use the session management APIs in your custom application.

### Using a Lambda function

When you use a Lambda function, Amazon Lex V2 calls it with a JSON structure that contains the input to the function. The JSON structure contains a field called `transcriptions` that contains the possible transcriptions that Amazon Lex V2 has determined for the utterance. The `transcriptions` field contains one to three possible transcriptions, each with a confidence score.

To use the intent from an alternative transcription, you specify it in the `ConfirmIntent` or the `ElicitSlot` dialog action in your Lambda function. To use a slot value from an alternative transcription, set the value in the `intent` field in your Lambda function response. For more information, see [Using an AWS Lambda function \(p. 194\)](#).

#### Example code

The following code example is a Python Lambda function that uses audio transcriptions to improve the conversation experience for the user.

To use the example code, you must have:

- A bot with one language, either English (GB) (`en_GB`) or English (US) (`en_US`).
- One intent, `OrderBirthStone`. Make sure that the **Use a Lambda function for initialization and validation** is selected in the **Code hooks** section of the intent definition.
- The intent should have two slots, "BirthMonth" and "Name," both of type `AMAZON.AlphaNumeric`.
- An alias with the Lambda function defined. For more information, see [Attaching a Lambda function to a bot alias \(p. 195\)](#).

```
import time
```

```
import os
import logging

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

# --- Helpers that build all of the responses ---

def elicit_slot(session_attributes, intent_request, slots, slot_to_elicit, message):
    return {
        'sessionState': {
            'dialogAction': {
                'type': 'ElicitSlot',
                'slotToElicit': slot_to_elicit
            },
            'intent': {
                'name': intent_request['sessionState']['intent']['name'],
                'slots': slots,
                'state': 'InProgress'
            },
            'sessionAttributes': session_attributes,
            'originatingRequestId': 'e3ab4d42-fb5f-4cc3-bb78-caaf6fc7cccd'
        },
        'sessionId': intent_request['sessionId'],
        'messages': [message],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes' in
intent_request else None
    }

def close(intent_request, session_attributes, fulfillment_state, message):
    intent_request['sessionState']['intent']['state'] = fulfillment_state
    return {
        'sessionState': {
            'sessionAttributes': session_attributes,
            'dialogAction': {
                'type': 'Close'
            },
            'intent': intent_request['sessionState']['intent'],
            'originatingRequestId': '3ab4d42-fb5f-4cc3-bb78-caaf6fc7cccd'
        },
        'messages': [message],
        'sessionId': intent_request['sessionId'],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes' in
intent_request else None
    }

def delegate(intent_request, session_attributes):
    return {
        'sessionState': {
            'dialogAction': {
                'type': 'Delegate'
            },
            'intent': intent_request['sessionState']['intent'],
            'sessionAttributes': session_attributes,
            'originatingRequestId': 'abc'
        },
        'sessionId': intent_request['sessionId'],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes' in
intent_request else None
    }

def get_session_attributes(intent_request):
```

```

sessionState = intent_request['sessionState']
if 'sessionAttributes' in sessionState:
    return sessionState['sessionAttributes']

return {}

def get_slots(intent_request):
    return intent_request['sessionState']['intent']['slots']

""" --- Functions that control the behavior of the bot --- """

def order_birth_stone(intent_request):
    """
    Performs dialog management and fulfillment for ordering a birth stone.
    Beyond fulfillment, the implementation for this intent demonstrates the following:
        1) Use of N best transcriptions to re prompt user when confidence for top transcript is
           below a threshold
        2) Overrides resolved slot for birth month from a known fixed list if the top
           transcript
           is not accurate.
    """

    transcriptions = intent_request['transcriptions']

    if intent_request['invocationSource'] == 'DialogCodeHook':
        # Disambiguate if there are multiple transcriptions and the top transcription
        # confidence is below a threshold (0.8 here)
        if len(transcriptions) > 1 and transcriptions[0]['transcriptionConfidence'] < 0.8:
            if transcriptions[0]['resolvedSlots'] is not {} and 'Name' in transcriptions[0]
['resolvedSlots'] and \
                transcriptions[0]['resolvedSlots']['Name'] is not None:
                return prompt_for_name(intent_request)
            elif transcriptions[0]['resolvedSlots'] is not {} and 'BirthMonth' in
transcriptions[0]['resolvedSlots'] and \
                transcriptions[0]['resolvedSlots']['BirthMonth'] is not None:
                return validate_month(intent_request)

    return continue_conversation(intent_request)

def prompt_for_name(intent_request):
    """
    If the confidence for the name is not high enough, re prompt the user with the
    recognized names
    so it can be confirmed.
    """

    resolved_names = []
    for transcription in intent_request['transcriptions']:
        if transcription['resolvedSlots'] is not {} and 'Name' in
transcription['resolvedSlots'] and \
            transcription['resolvedSlots']['Name'] is not None:
            resolved_names.append(transcription['resolvedSlots']['Name']['value']
['originalValue'])
    if len(resolved_names) > 1:
        session_attributes = get_session_attributes(intent_request)
        slots = get_slots(intent_request)
        return elicit_slot(session_attributes, intent_request, slots, 'Name',
                           {'contentType': 'PlainText',
                            'content': 'Sorry, did you say your name is {} ?'.format(" or
".join(resolved_names))})
    else:
        return continue_conversation(intent_request)

```

```

def validate_month(intent_request):
    """
    Validate month from an expected list, if not valid looks for other transcriptions and
    to see if the month
    recognized there has an expected value. If there is, replace with that and if not
    continue conversation.
    """

    expected_months = ['january', 'february', 'march']
    resolved_months = []
    for transcription in intent_request['transcriptions']:
        if transcription['resolvedSlots'] is not {} and 'BirthMonth' in
            transcription['resolvedSlots'] and \
                transcription['resolvedSlots']['BirthMonth'] is not None:
            resolved_months.append(transcription['resolvedSlots']['BirthMonth']['value']
            ['originalValue'])

    for resolved_month in resolved_months:
        if resolved_month in expected_months:
            intent_request['sessionState']['intent']['slots']['BirthMonth']
            ['resolvedValues'] = [resolved_month]
            break

    return continue_conversation(intent_request)

def continue_conversation(event):
    session_attributes = get_session_attributes(event)

    if event["invocationSource"] == "DialogCodeHook":
        return delegate(event, session_attributes)

# --- Intents ---

def dispatch(intent_request):
    """
    Called when the user specifies an intent for this bot.
    """

    logger.debug('dispatch sessionId={},',
    intentName='}.format(intent_request['sessionId'],
    intent_request['sessionState']['intent']['name']))

    intent_name = intent_request['sessionState']['intent']['name']

    # Dispatch to your bot's intent handlers
    if intent_name == 'OrderBirthStone':
        return order_birth_stone(intent_request)

    raise Exception('Intent with name ' + intent_name + ' not supported')

# --- Main handler ---

def lambda_handler(event, context):
    """
    Route the incoming request based on intent.
    The JSON body of the request is provided in the event slot.
    """

    # By default, treat the user request as coming from the America/New_York time zone.

```

```
os.environ['TZ'] = 'America/New_York'  
time.tzset()  
logger.debug('event={}{}'.format(event))  
  
return dispatch(event)
```

## Using the session management API

To use a different intent from the current intent, use the [PutSession](#) operation. For example, if you decide that the first alternative is preferable to the intent that Amazon Lex V2 chose, you can use the PutSession operation to change intents. That way the next intent that the user interacts with will be the one that you selected.

You can also use the PutSession operation to change the slot value in the intent structure to use a value from an alternative transcription.

For more information, see [Managing sessions with the Amazon Lex V2 API \(p. 234\)](#).

# Using runtime hints to improve recognition of slot values

With *runtime hints* you can give Amazon Lex V2 a set of slot values based on context to get better recognition in audio conversations and improved slot resolutions. You can use runtime hints to provide a list of phrases at runtime that become candidates for the resolution of a slot value.

For example, if a user interacting with a flight reservation bot frequently travels to San Francisco, Jakarta, Seoul, and Moscow you can configure runtime hints with a list of these four cities when eliciting for the destination to improve recognition for frequently travelled cities.

Runtime hints are only available in the English (US) and English (UK) languages. They can be used with the following slot types:

- Custom slot types
- AMAZON.City
- AMAZON.Country
- AMAZON.FirstName
- AMAZON.LastName
- AMAZON.State
- AMAZON.StreetName

### Runtime hints basics

- Runtime hints are used only when eliciting a slot value from a user.
- When you use runtime hints, the values of the hints are preferred over similar values. For example, for a food ordering bot, you can set a list of menu items as runtime hints while eliciting for food items in a custom slot to prefer “fillet” over similar sounding “fella”.
- If the user input is different from the values provided in runtime hints, the original user input will be used for the slot.
- For custom slot types, values provided as runtime hints will be used for resolution of the slot even if they are not part of the custom slot during bot creation.
- Runtime hints are supported only for 8 kHz audio input. They are available with [contact center integrations](#) supported by Amazon Lex V2. Runtime hints aren't provided for audio input from the [test window](#) on the Amazon Lex V2 console because it uses 16 kHz audio input.

**Note**

Before you can use runtime hints with an existing bot, you must first rebuild the bot. Existing versions of a bot don't support runtime hints. You must create a new version of the bot to use them.

You can send runtime hints to Amazon Lex V2 using the [PutSession](#), [RecognizeText](#), [RecognizeUtterance](#), or [StartConversation](#) operation. You can also add runtime hints using a Lambda function.

You can send runtime hints at the beginning of a conversation to configure the hints for each slot used in the bot, or you can send hints as part of the session state during a conversation. The `runtimeHints` attribute maps a slot to the hints for that slot.

Once you send a runtime hint to Amazon Lex V2, they persist for every turn of the conversation until the session ends. If you send a null `runtimeHints` structure, the existing hints are used. You can modify the hints by:

- Sending a new `runtimeHints` structure to the bot. The contents of the new structure replace the existing ones.
- Sending an empty `runtimeHints` structure to the bot. This clears the runtime hints for the bot.

## Adding slot values in context

Add context for your bot by providing expected slot values as runtime hints when your application has information about the user's next likely utterance. This enables your bot to use information that your application has to improve the conversation with the user.

For example, in a banking app you can generate a list of account nicknames for a specific user, and then use the list when eliciting the account that the user wants to access.

Send runtime hints at the start of the conversation when you have context to help your bot interpret user input. For example, if you have the user's phone number, you can use this information to look up the user so that you can use the `PutSession` or `StartConversation` operation to pass first and last name hints to the bot if you are eliciting for the user's name to validate their credentials.

During a conversation, you might gather information from one slot value that can help with another slot value. For example, in a car care app when you have the user's account number you can do a look up to find the cars that the customer owns and pass them along as hints to another slot.

Enter acronyms, or other words whose letters should be pronounced individually, as single letters separated by a period and a space. Don't use individual letters unless they are part of a phrase, such as "J. P. Morgan" or "A.W.S". You can use upper- or lower-case letters to define an acronym.

## Adding hints to a slot

To add runtime hints to a slot, you use the `runtimeHints` structure that is part of the `sessionState` structure. The following is an example of the `runtimeHints` structure. It provides hints for two slots, "FirstName" and "LastName" for the "MakeAppointment" intent.

```
{  
    "sessionState": {  
        "intent": {},  
        "activeContexts": [],  
        "dialogAction": {},  
        "originatingRequestId": {},  
        "sessionAttributes": {},  
        "runtimeHints": {  
            "slotHints": {  
                "FirstName": "John",  
                "LastName": "Doe"  
            }  
        }  
    }  
}
```

```
"MakeAppointment": {
    "FirstName": {
        "runtimeHintValues": [
            {
                "phrase": "John"
            },
            {
                "phrase": "Mary"
            }
        ]
    },
    "LastName": {
        "runtimeHintValues": [
            {
                "phrase": "Stiles"
            },
            {
                "phrase": "Major"
            }
        ]
    }
}
```

You can also use a Lambda function to add runtime hints during a conversation. To add runtime hints, you add the `runtimeHints` structure to the session state of the response that your Lambda function sends to Amazon Lex V2. For more information, see [Response format \(p. 201\)](#).

You must specify a valid `intentName` and `slotName` in the request, otherwise Amazon Lex V2 returns a runtime error.

## Using spelling styles to capture slot values

Amazon Lex V2 provides built-in slots to capture user-specific information such as first name, last name, email address or alphanumeric identifiers. For example, you can use the `AMAZON.LastName` slot to capture surnames such as "Jackson" or "Garcia." However, Amazon Lex V2 may get confused with surnames that are difficult to pronounce or that are not common in a locale, such as "Xiulan." To capture such names, you can ask the user to provide input in *spell by letter* or *spell by word* style.

Amazon Lex V2 provides three *slot elicitation styles* for you to use. When you set a slot elicitation style, it changes the way Amazon Lex V2 interprets the input from the user.

**Spell by letter** – With this style, you can instruct the bot to listen for spellings instead of the whole phrase. For example, to capture a last name such as "Xiulan," you can tell the user to spell out their last name one letter at a time. The bot will capture the spelling and resolve the letters to a word. For example, if the user says "x i u l a n," the bot captures the last name as "xiulan."

**Spell by word** – In voice conversations, especially using the telephone, there are a few letters, such as "t," "b," "p", that sound similar. When capturing alphanumeric values or spelling names results in an incorrect value, you can prompt the user to provide an identifying word along with the letter. For example, if the voice response to a request for a booking ID is "abp123," your bot might instead recognize the phrase "abb123" instead. If this is an incorrect value, you can ask the user to provide the input as "a as in alpha b as in boy p as in peter one two three." The bot will resolve the input to "abp123."

When using spell by word, you can use the following formats:

- "as in" (a as in apple)

- "for" (a for apple)
- "like" (a like apple)

**Default** – This is the natural style of slot capture using word pronunciation. For example, it can capture names such as "John Stiles" naturally. If a slot elicitation style isn't specified, the bot uses the default style. For the AMAZON.AlphaNumeric and AMAZON.UKPostal code slot types, the default style supports spell by letter input.

If the name "Xiulan" is spoken using a mix of letters and words , such as "x as in x-ray i u l as in lion a n" the slot elicitation style must be set to spell-by-word style. The spell-by-letter style won't recognize it.

You should create a voice interface that captures slot values with natural conversational style for a better experience. For inputs that are not correctly captured using the natural style, you can re-prompt the user and set the slot elicitation style to spell-by-letter or spell-by-word.

You can use spell-by-word and spell-by-letter styles for the following slot types in the English (US), English (UK), and English (Australia) languages:

- [AMAZON.AlphaNumeric \(p. 76\)](#)
- [AMAZON.EmailAddress \(p. 78\)](#)
- [AMAZON.FirstName \(p. 79\)](#)
- [AMAZON.LastName \(p. 79\)](#)
- [AMAZON.UKPostalCode \(p. 82\)](#)
- [Custom Slot Types \(p. 181\)](#)

## Enabling spelling

You enable spell-by-letter and spell-by-word at runtime when you are eliciting slots from the user. You can set the spelling style with the [PutSession](#), [RecognizeText](#), [RecognizeUtterance](#), or [StartConversation](#) operation. You can also enable spell-by-letter and spell-by-word using a Lambda function.

You set the spelling style using the dialogAction field of the request's session state. You can only set the style when the dialog action type is ElicitSlot and when the slot to elicit is one of the supported slot types.

The following JSON code shows the dialogAction field set to use the spell-by-word style:

```
"dialogAction": {  
    "slotElicitationStyle": "SpellByWord",  
    "slotToElicit": "BookingId",  
    "type": "ElicitSlot"  
}
```

The slotElicitationStyle field can be set to SpellByLetter, SpellByWord, or Default. If you don't specify a value, then the value is set to Default.

## Example code

Changing the spelling style is usually performed if the first attempt to resolve a slot value that didn't work. The following code example is a Python Lambda function that uses the spell-by-word style on the second attempt to resolve a slot.

To use the example code, you must have:

- A bot with one language, English (GB) (en\_GB).

- One intent, "CheckAccount" with one sample utterance, "I would like to check my account". Make sure that **Use a Lambda function for initialization and validation** is selected in the **Code hooks** section of the intent definition.
- The intent should have one slot, "PostalCode", of the AMAZON.UKPostalCode built-in type.
- An alias with the Lambda function defined. For more information, see [Attaching a Lambda function to a bot alias \(p. 195\)](#).

```

import json
import time
import os
import logging

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

# --- Helpers that build all of the responses ---

def get_slots(intent_request):
    return intent_request['sessionState']['intent']['slots']

def get_session_attributes(intent_request):
    sessionState = intent_request['sessionState']
    if 'sessionAttributes' in sessionState:
        return sessionState['sessionAttributes']
    return {}

def get_slot(intent_request, slotName):
    slots = get_slots(intent_request)
    if slots is not None and slotName in slots and slots[slotName] is not None:
        logger.debug('resolvedValue={}'.format(slots[slotName]['value']))
    return slots[slotName]['value']['resolvedValues']
    else:
        return None

def elicit_slot(session_attributes, intent_request, slots, slot_to_elicit,
slot_elicitation_style, message):
    return {'sessionState': {'dialogAction': {'type': 'ElicitSlot',
                                             'slotToElicit': slot_to_elicit,
                                             'slotElicitationStyle':
slot_elicitation_style
                                         },
                           'intent': {'name': intent_request['sessionState']['intent']
['name'],
                                      'slots': slots,
                                      'state': 'InProgress'
                                         },
                           'sessionAttributes': session_attributes,
                           'originatingRequestId': 'REQUESTID'
                                         },
            'sessionId': intent_request['sessionId'],
            'messages': [ message ],
            'requestAttributes': intent_request['requestAttributes']
            if 'requestAttributes' in intent_request else None
        }

def build_validation_result(isvalid, violated_slot, slot_elicitation_style,
message_content):
    return {'isValid': isvalid,
            'violatedSlot': violated_slot,
            'slotElicitationStyle': slot_elicitation_style,
            'message': {'contentType': 'PlainText',

```

```

        'content': message_content}
    }

def GetItemInDatabase(postal_code):
    """
    Perform database check for transcribed postal code. This is a no-op
    check that shows that postal_code can't be found in the database.
    """
    return None

def validate_postal_code(intent_request):
    postal_code = get_slot(intent_request, 'PostalCode')

    if GetItemInDatabase(postal_code) is None:
        return build_validation_result(
            False,
            'PostalCode',
            'SpellByWord',
            "Sorry, I can't find your information. " +
            "To try again, spell out your postal " +
            "code using words, like a as in apple."
        )
    return {'isValid': True}

def check_account(intent_request):
    """
    Performs dialog management and fulfillment for checking an account
    with a postal code. Besides fulfillment, the implementation for this
    intent demonstrates the following:
    1) Use of elicitSlot in slot validation and re-prompting.
    2) Use of sessionAttributes to pass information that can be used to
       guide a conversation.
    """
    slots = get_slots(intent_request)
    postal_code = get_slot(intent_request, 'PostalCode')
    session_attributes = get_session_attributes(intent_request)

    if intent_request['invocationSource'] == 'DialogCodeHook':
        # Validate the PostalCode slot. If any aren't valid,
        # re-elicit for the value.
        validation_result = validate_postal_code(intent_request)
        if not validation_result['isValid']:
            slots[validation_result['violatedSlot']] = None
            return elicit_slot(
                session_attributes,
                intent_request,
                slots,
                validation_result['violatedSlot'],
                validation_result['slotElicitationStyle'],
                validation_result['message']
            )

    return close(
        intent_request,
        session_attributes,
        'Fulfilled',
        {'contentType': 'PlainText',
         'content': 'Thanks'
        }
    )

def close(intent_request, session_attributes, fulfillment_state, message):
    intent_request['sessionState']['intent']['state'] = fulfillment_state
    return {
        'sessionState': {

```

```
'sessionAttributes': session_attributes,
'dialogAction': {
    'type': 'Close'
},
'intent': intent_request['sessionState']['intent'],
'originatingRequestId': 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx',
},
'messages': [ message ],
'sessionId': intent_request['sessionId'],
'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes' in
intent_request else None
}

# --- Intents ---

def dispatch(intent_request):
    """
    Called when the user specifies an intent for this bot.
    """
    intent_name = intent_request['sessionState']['intent']['name']
    response = None

    # Dispatch to your bot's intent handlers
    if intent_name == 'CheckAccount':
        response = check_account(intent_request)

    return response

# --- Main handler ---

def lambda_handler(event, context):
    """
    Route the incoming request based on the intent.
    The JSON body of the request is provided in the event slot.
    """

    # By default, treat the user request as coming from
    # Eastern Standard Time.
    os.environ['TZ'] = 'America/New_York'
    time.tzset()

    logger.debug('event={}'.format(json.dumps(event)))
    response = dispatch(event)
    logger.debug("response={}".format(json.dumps(response)))

    return response
```

# Streaming to an Amazon Lex V2 bot

You can use the Amazon Lex V2 streaming API to start a bidirectional stream between an Amazon Lex V2 bot and your application. Starting a stream enables the bot to manage the conversation between the bot and the user. The bot responds to user input without you writing code to handle responses from the user. The bot can:

- Handle interruptions from the user while it's playing a prompt. For more information, see [Enabling your bot to be interrupted by your user \(p. 270\)](#).
- Wait for the user to provide input. For example, the bot can wait for the user to gather credit card information. For more information, see [Enabling the bot to wait for the user to provide more information \(p. 271\)](#).
- Take both dual-tone multiple-frequency (DTMF) and audio input in the same stream.
- Handle pauses in user input better than if you were managing the conversation from your application.

Not only does the Amazon Lex V2 bot respond to data sent from your application, but it also sends information about the state of the conversation to your application. You can use this information to change how your application responds to customers.

The Amazon Lex V2 bot also monitors the connection between the bot and your application. It can determine if the connection has timed out.

To use the API to start a stream to an Amazon Lex V2 bot, see [Starting a stream to a bot \(p. 254\)](#).

When you start streaming to an Amazon Lex V2 bot from your application, you can configure the bot to accept audio input or text input from the user. You can also choose whether the user receives audio or text in response to their input.

If you've configured the Amazon Lex V2 bot to accept audio input from the user, it can't take text input. If you've configured the bot to accept text input, the user can only use written text to communicate with it.

When an Amazon Lex V2 bot takes a streaming audio input, the bot determines when a user begins speaking and when they stop speaking. It handles any pauses or any interruptions from the user. It can also take DTMF (dual-tone multi-frequency) input and speech input in the same stream. This helps the user interact with the bot more naturally. You can present users with welcome messages and prompts. You can also enable users to interrupt those messages and prompts.

When you start a bidirectional stream, Amazon Lex V2 uses the [HTTP/2 protocol](#). Your application and the bot exchange data in a single stream as a series of *events*. An event can be one of the following:

- Text, audio, or DTMF input from the user.
- Signals from the application to the Amazon Lex V2 bot. These include an indication that audio playback of a message has been completed, or that the user has disconnected from the session.

For more information about events, see [Starting a stream to a bot \(p. 254\)](#). For information about how to encode events, see [Event stream encoding \(p. 269\)](#).

## Topics

- [Starting a stream to a bot \(p. 254\)](#)
- [Event stream encoding \(p. 269\)](#)
- [Enabling your bot to be interrupted by your user \(p. 270\)](#)
- [Enabling the bot to wait for the user to provide more information \(p. 271\)](#)

- [Configuring fulfillment progress updates \(p. 272\)](#)
- [Configuring timeouts for capturing user input \(p. 274\)](#)

## Starting a stream to a bot

You use the [StartConversation](#) operation to start a stream between the user and the Amazon Lex V2 bot in your application. The POST request from the application establishes a connection between your application and the Amazon Lex V2 bot. This enables your application and the bot to start exchanging information with each other through events.

The StartConversation operation is supported only in the following SDKs:

- [AWS SDK for C++](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

The first event your application must send to the Amazon Lex V2 bot is a [ConfigurationEvent](#). This event includes information such as the response type format. The following are the parameters that you can use in a configuration event:

- **responseContentType** – Determines whether the bot responds to user input with text or speech.
- **sessionState** – Information relating to the streaming session with the bot such as predetermined intent or dialog state.
- **welcomeMessages** – Specifies the welcome messages that play for the user at the beginning of their conversation with a bot. These messages play before the user provides any input. To activate a welcome message, you must also specify values for the sessionState and dialogAction parameters.
- **disablePlayback** – Determines whether the bot should wait for a cue from the client before it starts listening for caller input. By default, playback is activated, so the value of this field is false.
- **requestAttributes** – Provides additional information for the request.

For information about how to specify values for the preceding parameters, see the [ConfigurationEvent](#) data type of the [StartConversation](#) operation.

Each stream between a bot and your application can only have one configuration event. After your application has sent a configuration event, the bot can take additional communication from your application.

If you've specified that your user is using audio to communicate with the Amazon Lex V2 bot, your application can send the following events to the bot during that conversation:

- **AudioInputEvent** – Contains an audio chunk that has maximum size of 320 bytes. Your application must use multiple audio input events to send a message from the server to the bot. Every audio input event in the stream must have the same audio format.
- **DTMFInputEvent** – Sends a DTMF input to the bot. Each DTMF key press corresponds to a single event.
- **PlaybackCompletionEvent** – Informs the server that a response from the user's input has been played back to them. You must use a playback completion event if you're sending an audio response to the user. If disablePlayback of your configuration event is true, you can't use this feature.
- **DisconnectionEvent** – Informs the bot that the user has disconnected from the conversation.

If you've specified that the user is using text to communicate with the bot, your application can send the following events to the bot during that conversation:

- [TextInputEvent](#) – Text that is sent from your application to the bot. You can have up to 512 characters in a text input event.
- [PlaybackCompletionEvent](#) – Informs the server that a response from the user's input has been played back to them. You must use this event if you're playing audio back to the user. If `disablePlayback` of your configuration event is `true`, you can't use this feature.
- [DisconnectionEvent](#) – Informs the bot that the user has disconnected from the conversation.

You must encode every event that you send to an Amazon Lex V2 bot in the correct format. For more information, see [Event stream encoding \(p. 269\)](#).

Every event has an event ID. To help troubleshoot any issues that might occur in the stream, assign a unique event ID to each input event. You can then troubleshoot any processing failures with the bot.

Amazon Lex V2 also uses timestamps for each event. You can use these timestamps in addition to the event ID to help troubleshoot any network transmission issues.

During the conversation between the user and the Amazon Lex V2 bot, the bot can send the following outbound events in response to the user:

- [IntentResultEvent](#) – Contains the intent that Amazon Lex V2 determined from the user utterance. Each internal result event includes:
  - **inputMode** – The type of user utterance. Valid values are Speech, DTMF, or Text.
  - **interpretations** – Interpretations that Amazon Lex V2 determines from the user utterance.
  - **requestAttributes** – If you haven't modified the request attributes by using a lambda function, these are the same attributes that were passed at the start of the conversation.
  - **sessionId** – Session identifier used for the conversation.
  - **sessionState** – The state of the user's session with Amazon Lex V2.
- [TranscriptEvent](#) – If the user provides an input to your application, this event contains the transcript of the user's utterance to the bot. Your application does not receive a TranscriptEvent if there's no user input.

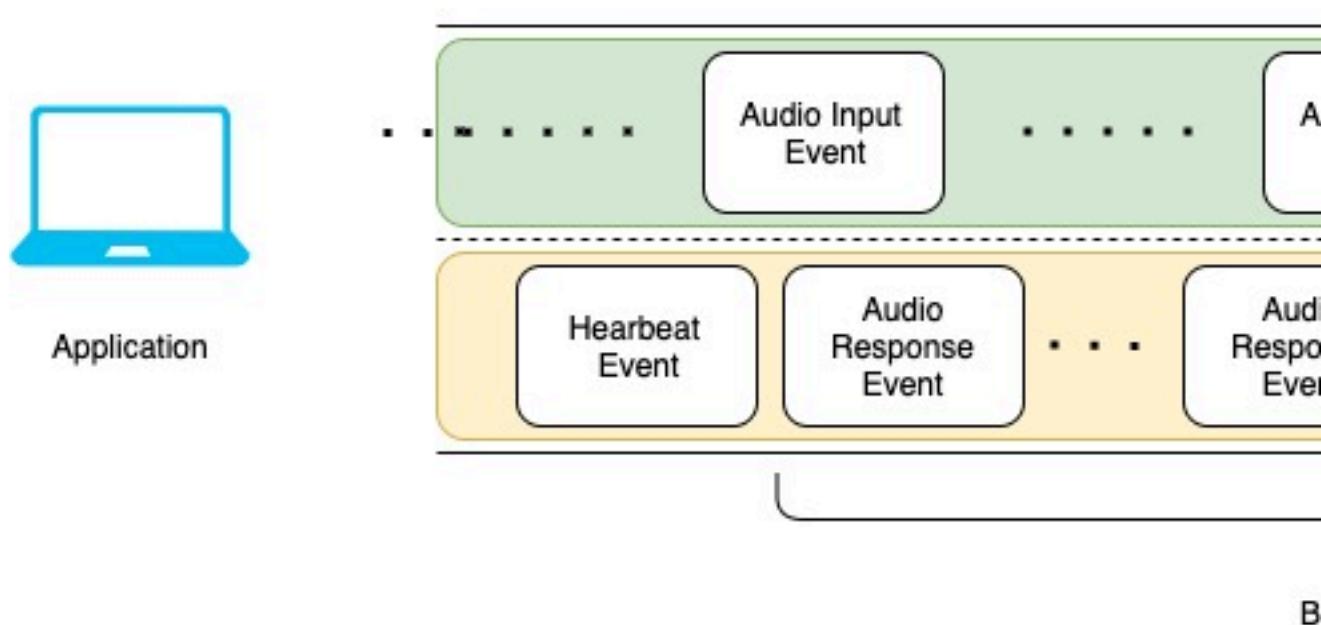
The value of the transcript event sent to your application depends on whether you've specified audio (speech and DMTF) or text as a conversation mode:

- Transcript of speech input – If the user is speaking with the bot, the transcript event is the transcription of the user's audio. It's a transcript of all the speech from the time the user begins speaking to the time they end speaking.
- Transcript of DTMF input – If the user is typing on a keypad, the transcript event contains all the digits the user pressed in their input.
- Transcript of text input – If the user is providing text input, the transcript event contains all of the text in the user's input.
- [TextResponseEvent](#) – Contains the bot response in text format. A text response is returned by default. If you've configured Amazon Lex V2 to return an audio response, this text is used to generate an audio response. Each text response event contains an array of message objects that the bot returns to the user.
- [AudioResponseEvent](#) – Contains the audio response synthesized from the text generated in the TextResponseEvent. To receive audio response events, you must configure Amazon Lex V2 to provide an audio response. All audio response events have the same audio format. Each event contains audio chunks of no more than 100 bytes. Amazon Lex V2 sends an empty audio chunk with the `bytes` field set to `null` to indicate that the end of the audio response event to your application.
- [PlaybackInterruptionEvent](#) – When a user interrupts a response that the bot has sent to your application, Amazon Lex V2 triggers this event to stop the playback of the response.
- [HeartbeatEvent](#) – Amazon Lex V2 sends this event back periodically to keep the connection between your application and the bot from timing out.

## Time sequence of events for an audio conversation

The following diagrams show a streaming audio conversation between a user and an Amazon Lex V2 bot. The application continuously streams audio to the bot, and the bot looks for user input from the audio. In this example, both the user and the bot are using speech to communicate. Each diagram corresponds to a user utterance and the response of the bot to that utterance.

The following diagram shows the beginning of a conversation between the application and the bot. The stream begins at time zero (t0).

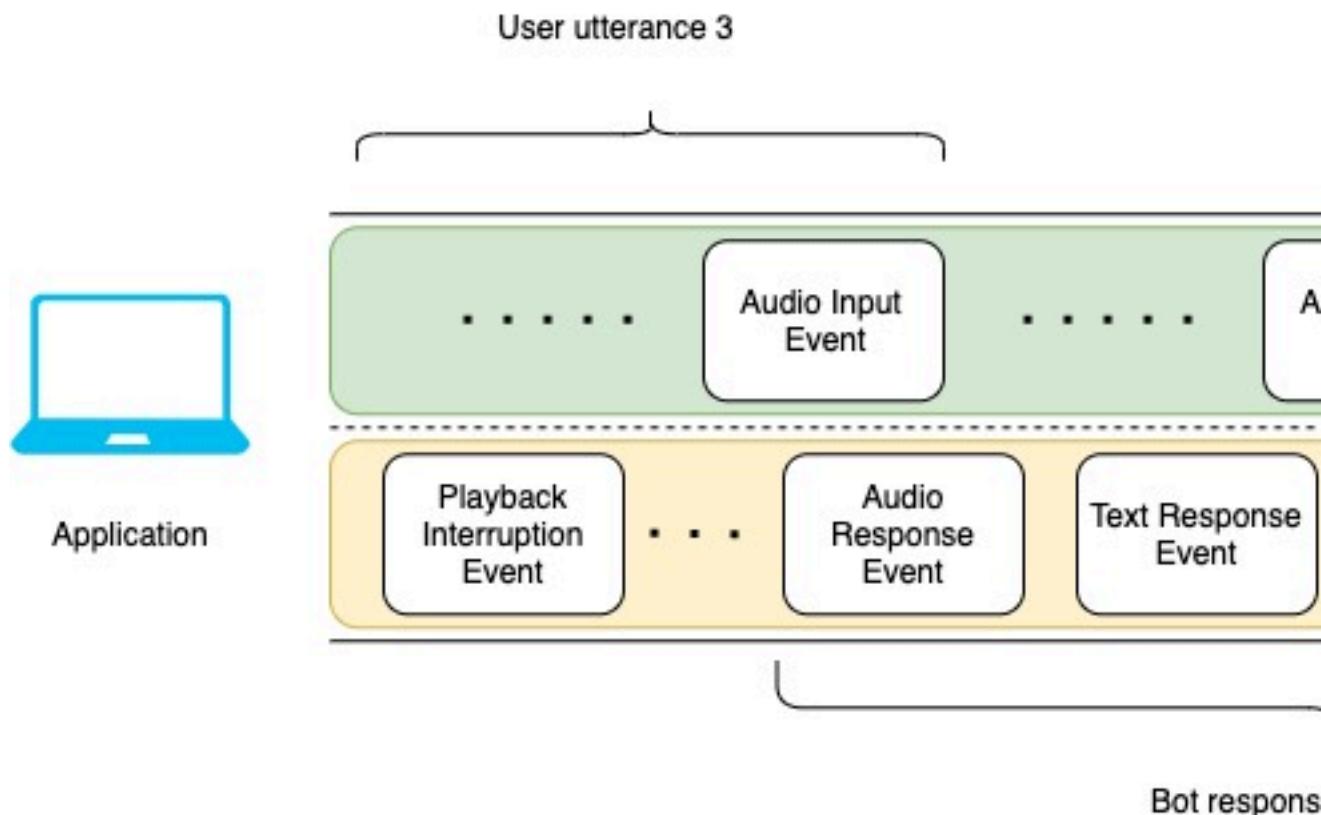


The following list describes the events of the preceding diagram.

- t0: The application sends a configuration event to the bot to start the stream.
- t1: The application streams audio data. This data is broken into a series of input events from the application.
- t2: For *user utterance 1*, the bot detects an audio input event when the user begins speaking.
- t2: While the user is speaking, the bot sends a heartbeat event to maintain the connection. It sends these events intermittently to make sure the connection doesn't time out.
- t3: The bot detects the end of the user's utterance.
- t4: The bot sends back a transcript event that contains a transcript of the user's speech to the application. This is the beginning of *Bot response to user utterance 1*.
- t5: The bot sends an intent result event to indicate the action that the user wants to perform.

- t6: The bot begins providing its response as text in a text response event.
- t7: The bot sends a series of audio response events to the application to play for the user.
- t8: The bot sends another heartbeat event to intermittently maintain the connection.

The following diagram is a continuation of the previous diagram. It shows the application sending a playback completion event to the bot to indicate that it has stopped playing the audio response for the user. The application plays back *Bot response to user utterance 1* to the user. The user responds to *Bot response to user utterance 1* with *User utterance 2*.

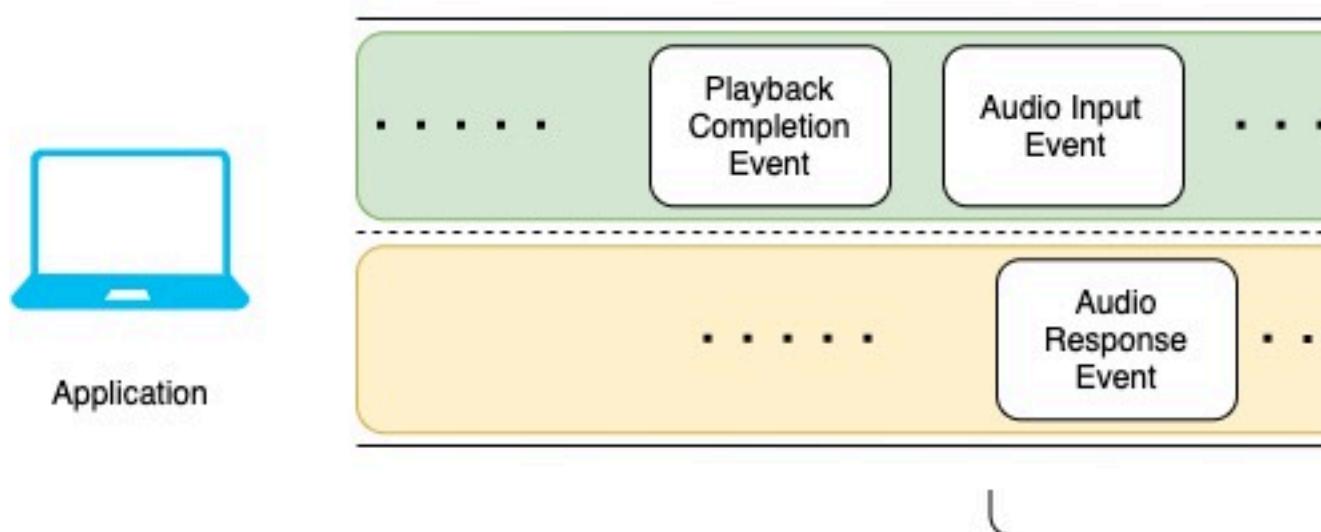


The following list describes the events of the preceding diagram:

- t10: The application sends a playback completion event to indicate that it has finished playing the bot's message to the user.
- t11: The application sends the user response back to the bot as *User utterance 2*.
- t12: For *Bot response to user utterance 2*, the bot waits for the user to stop speaking and then begins to provide an audio response.
- t13: While the bot sends *Bot response to user utterance 2* to the application, the bot detects the start of *User utterance 3*. The bot stops *Bot response to user utterance 2* and sends a playback interruption event.

- t14: The bot sends a playback interruption event to the application to signal that the user has interrupted the prompt.

The following diagram shows the *Bot response to user utterance 3*, and that the conversation continues after the bot responds to the user utterance.



## Using the API to start a streaming conversation

When you start a stream to an Amazon Lex V2 bot, you accomplish the following tasks:

1. Create an initial connection to the server.
2. Configure the security credentials and bot details. Bot details include whether the bot takes DTMF and audio input, or text input.
3. Send events to the server. These events are text data or audio data from the user.
4. Process events sent from the server. In this step, you determine whether the bot output is presented to the user as text or speech.

The following code examples initialize a streaming conversation with an Amazon Lex V2 bot and your local machine. You can modify the code to meet your needs.

The following code is an example request using the AWS SDK for Java to start the connection to a bot and configure the bot details and credentials.

```

package com.lex.streaming.sample;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lexruntimev2.LexRuntimeV2AsyncClient;
import software.amazon.awssdk.services.lexruntimev2.model.ConversationMode;
import software.amazon.awssdk.services.lexruntimev2.model.StartConversationRequest;

import java.net.URISyntaxException;
import java.util.UUID;
import java.util.concurrent.CompletableFuture;

/**
 * The following code creates a connection with the Amazon Lex bot and configures the bot
 * details and credentials.
 * Prerequisite: To use this example, you must be familiar with the Reactive streams
 * programming model.
 * For more information, see
 * https://github.com/reactive-streams/reactive-streams-jvm.
 * This example uses AWS SDK for Java for Amazon Lex V2.
 * <p>
 * The following sample application interacts with an Amazon Lex bot with the streaming
 * API. It uses the Audio
 * conversation mode to return audio responses to the user's input.
 * <p>
 * The code in this example accomplishes the following:
 * <p>
 * 1. Configure details about the conversation between the user and the Amazon Lex bot.
 * These details include the conversation mode and the specific bot the user is speaking
 * with.
 * 2. Create an events publisher that passes the audio events to the Amazon Lex bot after
 * you establish the connection. The code we provide in this example tells your computer to
 * pick up the audio from
 * your microphone and send that audio data to Amazon Lex.
 * 3. Create a response handler that handles the audio responses from the Amazon Lex bot
 * and plays back the audio to you.
 */
public class LexBidirectionalStreamingExample {

    public static void main(String[] args) throws URISyntaxException, InterruptedException
    {
        String botId = "";
        String botAliasId = "";
        String localeId = "";
        String accessKey = "";
        String secretKey = "";
        String sessionId = UUID.randomUUID().toString();
        Region region = Region.region_name; // Choose an AWS Region where the Amazon Lex
        Streaming API is available.

        AwsCredentialsProvider awsCredentialsProvider = StaticCredentialsProvider
            .create(AwsBasicCredentials.create(accessKey, secretKey));

        // Create a new SDK client. You need to use an asynchronous client.
        System.out.println("step 1: creating a new Lex SDK client");
        LexRuntimeV2AsyncClient lexRuntimeServiceClient = LexRuntimeV2AsyncClient.builder()
            .region(region)
            .credentialsProvider(awsCredentialsProvider)
            .build();
    }
}

```

```

// Configure the bot, alias and locale that you'll use to have a conversation.
System.out.println("step 2: configuring bot details");
StartConversationRequest.Builder startConversationRequestBuilder =
StartConversationRequest.builder()
    .botId(botId)
    .botAliasId(botAliasId)
    .localeId(localeId);

// Configure the conversation mode of the bot. By default, the
// conversation mode is audio.
System.out.println("step 3: choosing conversation mode");
startConversationRequestBuilder =
startConversationRequestBuilder.conversationMode(ConversationMode.AUDIO);

// Assign a unique identifier for the conversation.
System.out.println("step 4: choosing a unique conversation identifier");
startConversationRequestBuilder =
startConversationRequestBuilder.sessionId(sessionId);

// Start the initial request.
StartConversationRequest startConversationRequest =
startConversationRequestBuilder.build();

// Create a stream of audio data to the Amazon Lex bot. The stream will start after
the connection is established with the bot.
EventsPublisher eventsPublisher = new EventsPublisher();

// Create a class to handle responses from bot. After the server processes the user
data you've streamed, the server responds
// on another stream.
BotResponseHandler botResponseHandler = new BotResponseHandler(eventsPublisher);

// Start a connection and pass in the publisher that streams the audio and process
the responses from the bot.
System.out.println("step 5: starting the conversation ...");
CompletableFuture<Void> conversation = lexRuntimeServiceClient.startConversation(
    startConversationRequest,
    eventsPublisher,
    botResponseHandler);

// Wait until the conversation finishes. The conversation finishes if the dialog
state reaches the "Closed" state.
// The client stops the connection. If an exception occurs during the conversation,
the
// client sends a disconnection event.
conversation.whenComplete((result, exception) -> {
    if (exception != null) {
        eventsPublisher.disconnect();
    }
});

// The conversation finishes when the dialog state is closed and last prompt has
been played.
while (!botResponseHandler.isConversationComplete()) {
    Thread.sleep(100);
}

// Randomly sleep for 100 milliseconds to prevent JVM from exiting.
// You won't need this in your production code because your JVM is
// likely to always run.
// When the conversation finishes, the following code block stops publishing more
data and informs the Amazon Lex bot that there is no more data to send.
if (botResponseHandler.isConversationComplete()) {
    System.out.println("conversation is complete.");
}

```

```
        eventsPublisher.stop();
    }
}
```

The following code is an example request using the AWS SDK for Java to send events to the bot. The code in this example uses the microphone on your computer to send audio events.

```
package com.lex.streaming.sample;

import org.reactivestreams.Publisher;
import org.reactivestreams.Subscriber;
import
software.amazon.awssdk.services.lexruntimev2.model.StartConversationRequestEventStream;

/**
 * You use the Events publisher to send events to the Amazon Lex bot. When you establish a
connection, the bot uses the
 * subscribe() method and enables the events publisher starts sending events to
 * your computer. The bot uses the "request" method of the subscription to make more
requests. For more information on the request method, see https://github.com/reactive-streams/reactive-streams-jvm.
 */
public class EventsPublisher implements Publisher<StartConversationRequestEventStream> {

    private AudioEventsSubscription audioEventsSubscription;

    @Override
    public void subscribe(Subscriber<? super StartConversationRequestEventStream>
subscriber) {
        if (audioEventsSubscription == null) {

            audioEventsSubscription = new AudioEventsSubscription(subscriber);
            subscriber.onSubscribe(audioEventsSubscription);

        } else {
            throw new IllegalStateException("received unexpected subscription request");
        }
    }

    public void disconnect() {
        if (audioEventsSubscription != null) {
            audioEventsSubscription.disconnect();
        }
    }

    public void stop() {
        if (audioEventsSubscription != null) {
            audioEventsSubscription.stop();
        }
    }

    public void playbackFinished() {
        if (audioEventsSubscription != null) {
            audioEventsSubscription.playbackFinished();
        }
    }
}
```

The following code is an example request using the AWS SDK for Java to handle responses from the bot. The code in this example configures Amazon Lex V2 to play an audio response back to you.

```
package com.lex.streaming.sample;

import javazoom.jl.decoder.JavaLayerException;
import javazoom.jl.player.advanced.AdvancedPlayer;
import javazoom.jl.player.advanced.PlaybackEvent;
import javazoom.jl.player.advanced.PlaybackListener;
import software.amazon.awssdk.core.SdkPublisher;
import software.amazon.awssdk.services.lexruntimev2.model.AudioResponseEvent;
import software.amazon.awssdk.services.lexruntimev2.model.DialogActionType;
import software.amazon.awssdk.services.lexruntimev2.model.IntentResultEvent;
import software.amazon.awssdk.services.lexruntimev2.model.PlaybackInterruptedException;
import software.amazon.awssdk.services.lexruntimev2.model.StartConversationResponse;
import software.amazon.awssdk.services.lexruntimev2.model.StartConversationResponseStream;
import software.amazon.awssdk.services.lexruntimev2.model.StartConversationResponseHandler;
import software.amazon.awssdk.services.lexruntimev2.model.TextResponseEvent;
import software.amazon.awssdk.services.lexruntimev2.model.TranscriptEvent;

import java.io.IOException;
import java.io.UncheckedIOException;
import java.util.concurrent.CompletableFuture;

/**
 * The following class is responsible for processing events sent from the Amazon Lex bot.
 * The bot sends multiple audio events,
 * so the following code concatenates those audio events and uses a publicly available Java
 * audio player to play out the message to
 * the user.
 */
public class BotResponseHandler implements StartConversationResponseHandler {

    private final EventsPublisher eventsPublisher;

    private boolean lastBotResponsePlayedBack;
    private boolean isDialogStateClosed;
    private AudioResponse audioResponse;

    public BotResponseHandler(EventsPublisher eventsPublisher) {
        this.eventsPublisher = eventsPublisher;
        this.lastBotResponsePlayedBack = false; // At the start, we have not played back
        last response from bot.
        this.isDialogStateClosed = false; // At the start, the dialog state is open.
    }

    @Override
    public void responseReceived(StartConversationResponse startConversationResponse) {
        System.out.println("successfully established the connection with server. request
id:" + startConversationResponse.responseMetadata().requestId()); // would have 2XX,
request id.
    }

    @Override
    public void onEventStream(SdkPublisher<StartConversationResponseEventStream>
sdkPublisher) {

        sdkPublisher.subscribe(event -> {
            if (event instanceof PlaybackInterruptedException) {
                handle((PlaybackInterruptedException) event);
            } else if (event instanceof TranscriptEvent) {
                handle((TranscriptEvent) event);
            }
        });
    }
}
```

```

        } else if (event instanceof IntentResultEvent) {
            handle((IntentResultEvent) event);
        } else if (event instanceof TextResponseEvent) {
            handle((TextResponseEvent) event);
        } else if (event instanceof AudioResponseEvent) {
            handle((AudioResponseEvent) event);
        }
    });
}

@Override
public void exceptionOccurred(Throwable throwable) {
    System.err.println("got an exception:" + throwable);
}

@Override
public void complete() {
    System.out.println("on complete");
}

private void handle(PlaybackInterruptionEvent event) {
    System.out.println("Got a PlaybackInterruptionEvent: " + event);
}

private void handle(TranscriptEvent event) {
    System.out.println("Got a TranscriptEvent: " + event);
}

private void handle(IntentResultEvent event) {
    System.out.println("Got an IntentResultEvent: " + event);
    isDialogStateClosed =
DialogActionType.CLOSE.equals(event.sessionState().dialogAction().type()));
}

private void handle(TextResponseEvent event) {
    System.out.println("Got an TextResponseEvent: " + event);
    event.messages().forEach(message -> {
        System.out.println("Message content type:" + message.contentType());
        System.out.println("Message content:" + message.content());
    });
}

private void handle(AudioResponseEvent event) {//Synthesize speech
// System.out.println("Got a AudioResponseEvent: " + event);
if (audioResponse == null) {
    audioResponse = new AudioResponse();
    //Start an audio player in a different thread.
    CompletableFuture.runAsync(() -> {
        try {
            AdvancedPlayer audioPlayer = new AdvancedPlayer(audioResponse);

            audioPlayer.setPlayBackListener(new PlaybackListener() {
                @Override
                public void playbackFinished(PlaybackEvent evt) {
                    super.playbackFinished(evt);

                    // Inform the Amazon Lex bot that the playback has finished.
                    eventsPublisher.playbackFinished();
                    if (isDialogStateClosed) {
                        lastBotResponsePlayedBack = true;
                    }
                }
            });
            audioPlayer.play();
        } catch (JavaLayerException e) {

```

```

        throw new RuntimeException("got an exception when using audio player",
e);
    }
}

if (event.audioChunk() != null) {
    audioResponse.write(event.audioChunk().asByteArray());
} else {
    // The audio audio prompt has ended when the audio response has no
    // audio bytes.
    try {
        audioResponse.close();
        audioResponse = null; // Prepare for the next audio prompt.
    } catch (IOException e) {
        throw new UncheckedIOException("got an exception when closing the audio
response", e);
    }
}

// The conversation with the Amazon Lex bot is complete when the bot marks the Dialog
as DialogActionType.CLOSE
// and any prompt playback is finished. For more information, see
// https://docs.aws.amazon.com/lexv2/latest/dg/API\_runtime\_DialogAction.html.
public boolean isConversationComplete() {
    return isDialogStateClosed && lastBotResponsePlayedBack;
}

}

```

To configure a bot to respond to input events with audio, you must first subscribe to audio events from Amazon Lex V2 and then configure the bot to provide an audio response to the input events from the user.

The following code is an AWS SDK for Java example for subscribing to audio events from Amazon Lex V2.

```

package com.lex.streaming.sample;

import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.services.lexruntimev2.model.AudioInputEvent;
import software.amazon.awssdk.services.lexruntimev2.model.ConfigurationEvent;
import software.amazon.awssdk.services.lexruntimev2.model.DisconnectionEvent;
import software.amazon.awssdk.services.lexruntimev2.model.PlaybackCompletionEvent;
import
software.amazon.awssdk.services.lexruntimev2.model.StartConversationRequestEventStream;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.DataLine;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.TargetDataLine;
import java.io.IOException;
import java.io.UncheckedIOException;
import java.nio.ByteBuffer;
import java.util.Arrays;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.CompletableFuture;

```

```

import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicLong;

public class AudioEventsSubscription implements Subscription {
    private static final AudioFormat MIC_FORMAT = new AudioFormat(8000, 16, 1, true,
        false);
    private static final String AUDIO_CONTENT_TYPE = "audio/lpcm; sample-rate=8000; sample-
size-bits=16; channel-count=1; is-big-endian=false";
    //private static final String RESPONSE_TYPE = "audio/pcm; sample-rate=8000";
    private static final String RESPONSE_TYPE = "audio/mpeg";
    private static final int BYTES_IN_AUDIO_CHUNK = 320;
    private static final AtomicLong eventIdGenerator = new AtomicLong(0);

    private final AudioInputStream audioInputStream;
    private final Subscriber<? super StartConversationRequestEventStream> subscriber;
    private final EventWriter eventWriter;
    private CompletableFuture eventWriterFuture;

    public AudioEventsSubscription(Subscriber<? super StartConversationRequestEventStream>
        subscriber) {
        this.audioInputStream = getMicStream();
        this.subscriber = subscriber;
        this.eventWriter = new EventWriter(subscriber, audioInputStream);
        configureConversation();
    }

    private AudioInputStream getMicStream() {
        try {
            DataLine.Info dataLineInfo = new DataLine.Info(TargetDataLine.class,
                MIC_FORMAT);
            TargetDataLine targetDataLine = (TargetDataLine)
                AudioSystem.getLine(dataLineInfo);

            targetDataLine.open(MIC_FORMAT);
            targetDataLine.start();

            return new AudioInputStream(targetDataLine);
        } catch (LineUnavailableException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void request(long demand) {
        // If a thread to write events has not been started, start it.
        if (eventWriterFuture == null) {
            eventWriterFuture = CompletableFuture.runAsync(eventWriter);
        }
        eventWriter.addDemand(demand);
    }

    @Override
    public void cancel() {
        subscriber.onError(new RuntimeException("stream was cancelled"));
        try {
            audioInputStream.close();
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }

    public void configureConversation() {
        String eventId = "ConfigurationEvent-" +
            String.valueOf(eventIdGenerator.incrementAndGet());
    }
}

```

```
ConfigurationEvent configurationEvent = StartConversationRequestEventStream
    .configurationEventBuilder()
    .eventId(eventId)
    .clientTimestampMillis(System.currentTimeMillis())
    .responseContentType(RESPONSE_TYPE)
    .build();

System.out.println("writing config event");
eventWriter.writeConfigurationEvent(configurationEvent);
}

public void disconnect() {

    String eventId = "DisconnectionEvent-" +
String.valueOf(eventIdGenerator.incrementAndGet());

    DisconnectionEvent disconnectionEvent = StartConversationRequestEventStream
        .disconnectionEventBuilder()
        .eventId(eventId)
        .clientTimestampMillis(System.currentTimeMillis())
        .build();

    eventWriter.writeDisconnectEvent(disconnectionEvent);

    try {
        audioInputStream.close();
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}

//Notify the subscriber that we've finished.
public void stop() {
    subscriber.onComplete();
}

public void playbackFinished() {
    String eventId = "PlaybackCompletion-" +
String.valueOf(eventIdGenerator.incrementAndGet());

    PlaybackCompletionEvent playbackCompletionEvent =
StartConversationRequestEventStream
        .playbackCompletionEventBuilder()
        .eventId(eventId)
        .clientTimestampMillis(System.currentTimeMillis())
        .build();

    eventWriter.writePlaybackFinishedEvent(playbackCompletionEvent);
}

private static class EventWriter implements Runnable {
    private final BlockingQueue<StartConversationRequestEventStream> eventQueue;
    private final AudioInputStream audioInputStream;
    private final AtomicLong demand;
    private final Subscriber subscriber;

    private boolean conversationConfigured;

    public EventWriter(Subscriber subscriber, AudioInputStream audioInputStream) {
        this.eventQueue = new LinkedBlockingQueue<>();

        this.demand = new AtomicLong(0);
        this.subscriber = subscriber;
        this.audioInputStream = audioInputStream;
    }
}
```

```

public void writeConfigurationEvent(ConfigurationEvent configurationEvent) {
    eventQueue.add(configurationEvent);
}

public void writeDisconnectEvent(DisconnectionEvent disconnectionEvent) {
    eventQueue.add(disconnectionEvent);
}

public void writePlaybackFinishedEvent(PlaybackCompletionEvent playbackCompletionEvent) {
    eventQueue.add(playbackCompletionEvent);
}

void addDemand(long l) {
    this.demand.addAndGet(l);
}

@Override
public void run() {
    try {

        while (true) {
            long currentDemand = demand.get();

            if (currentDemand > 0) {
                // Try to read from queue of events.
                // If nothing is in queue at this point, read the audio events
                // directly from audio stream.
                for (long i = 0; i < currentDemand; i++) {

                    if (eventQueue.peek() != null) {
                        subscriber.onNext(eventQueue.take());
                        demand.decrementAndGet();
                    } else {
                        writeAudioEvent();
                    }
                }
            }
        } catch (InterruptedException e) {
            throw new RuntimeException("interrupted when reading data to be sent to
server");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

private void writeAudioEvent() {
    byte[] bytes = new byte[BYTES_IN_AUDIO_CHUNK];

    int numBytesRead = 0;
    try {
        numBytesRead = audioInputStream.read(bytes);
        if (numBytesRead != -1) {
            byte[] byteArrayCopy = Arrays.copyOf(bytes, numBytesRead);

            String eventId = "AudioEvent-" +
String.valueOf(eventIdGenerator.incrementAndGet());

            AudioInputEvent audioInputEvent = StartConversationRequestEventStream
                .audioInputEventBuilder()

                .audioChunk(SdkBytes.fromByteBuffer(ByteBuffer.wrap(byteArrayCopy)))
                .contentType(AUDIO_CONTENT_TYPE)
                .clientTimestampMillis(System.currentTimeMillis())
                .eventId(eventId).build();
        }
    }
}

```

```
//System.out.println("sending audio event:" + audioInputEvent);
subscriber.onNext(audioInputEvent);
demand.decrementAndGet();
//System.out.println("sent audio event:" + audioInputEvent);
} else {
    subscriber.onComplete();
    System.out.println("audio stream has ended");
}
}

} catch (IOException e) {
    System.out.println("got an exception when reading from audio stream");
    System.err.println(e);
    subscriber.onError(e);
}
}
}
```

The following AWS SDK for Java example configures the Amazon Lex V2 bot to provide an audio response to the input events.

```
package com.lex.streaming.sample;

import java.io.IOException;
import java.io.InputStream;
import java.io.UncheckedIOException;
import java.util.Optional;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class AudioResponse extends InputStream{

    // Used to convert byte, which is signed in Java, to positive integer (unsigned)
    private static final int UNSIGNED_BYTE_MASK = 0xFF;
    private static final long POLL_INTERVAL_MS = 10;

    private final LinkedBlockingQueue<Integer> byteQueue = new LinkedBlockingQueue<>();

    private volatile boolean closed;

    @Override
    public int read() throws IOException {
        try {
            Optional<Integer> maybeInt;
            while (true) {
                maybeInt = Optional.ofNullable(this.byteQueue.poll(POLL_INTERVAL_MS,
TimeUnit.MILLISECONDS));

                // If we get an integer from the queue, return it.
                if (maybeInt.isPresent()) {
                    return maybeInt.get();
                }

                // If the stream is closed and there is nothing queued up, return -1.
                if (this.closed) {
                    return -1;
                }
            }
        } catch (InterruptedException e) {
            throw new IOException(e);
        }
    }
}
```

```
        }

    /**
     * Writes data into the stream to be offered on future read() calls.
     */
    public void write(byte[] byteArray) {
        // Don't write into the stream if it is already closed.
        if (this.closed) {
            throw new UncheckedIOException(new IOException("Stream already closed when
attempting to write into it."));
        }

        for (byte b : byteArray) {
            this.byteQueue.add(b & UNSIGNED_BYTE_MASK);
        }
    }

    @Override
    public void close() throws IOException {
        this.closed = true;
        super.close();
    }
}
```

## Event stream encoding

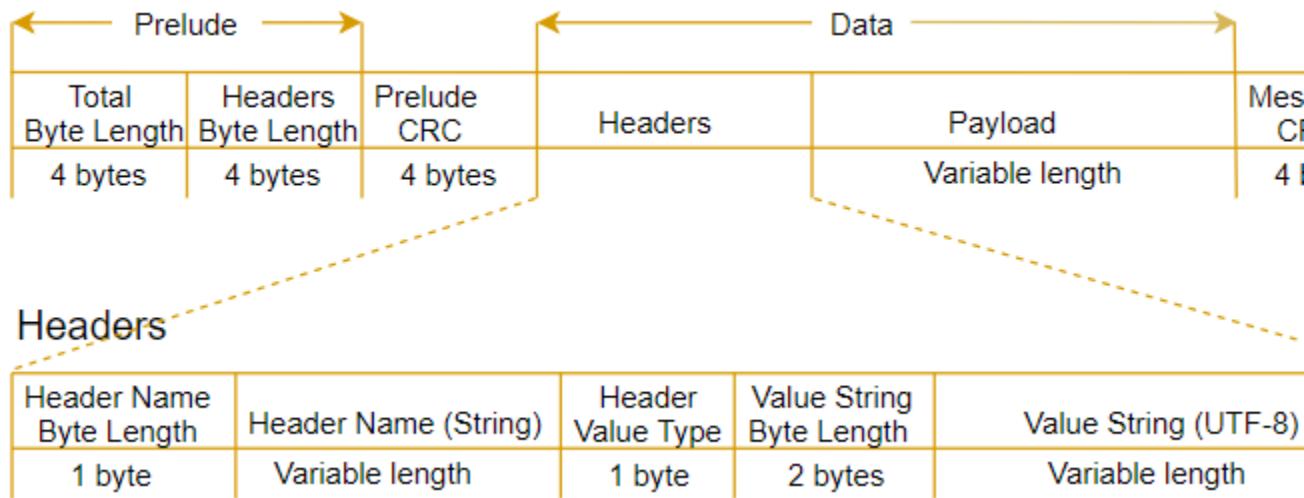
Event stream encoding provides bidirectional communication using messages between a client and a server. Data frames sent to the Amazon Lex V2 streaming service are encoded in this format. The response from Amazon Lex V2 also uses this encoding.

Each message consists of two sections: the prelude and the data. The prelude section contains the total byte length of the message and the combined byte length of all of the headers. The data section contains the headers and a payload.

Each section ends with a 4-byte big-endian integer CRC checksum. The message CRC checksum includes the prelude section and the data section. Amazon Lex V2 uses CRC32 (often referred to as GZIP CRC32) to calculate both CRCs. For more information about CRC32, see [GZIP file format specification version 4.3](#).

Total message overhead, including the prelude and both checksums, is 16 bytes.

The following diagram shows the components that make up a message and a header. There are multiple headers per message.



Each message contains the following components:

- **Prelude:** Always a fixed size of 8 bytes, two fields of 4 bytes each.
  - *First 4 bytes:* The total byte-length. This is the big-endian integer byte-length of the entire message, including the 4-byte length field itself.
  - *Second 4 bytes:* The headers byte-length. This is the big-endian integer byte-length of the headers portion of the message, excluding the headers length field itself.
- **Prelude CRC:** The 4-byte CRC checksum for the prelude portion of the message, excluding the CRC itself. The prelude has a separate CRC from the message CRC to ensure that Amazon Lex V2 can detect corrupted byte-length information immediately without causing errors such as buffer overruns.
- **Headers:** Metadata annotating the message, such as the message type, content type, and so on. Messages have multiple headers. Headers are key-value pairs where the key is a UTF-8 string. Headers can appear in any order in the headers portion of the message and any given header can appear only once. For the required header types, see the following sections.
- **Payload:** The audio or text content being sent to Amazon Lex.
- **Message CRC:** The 4-byte CRC checksum from the start of the message to the start of the checksum. That includes everything in the message except the CRC itself.

Each header contains the following components. There are multiple headers per frame.

- **Header name byte-length:** The byte-length of the header name.
- **Header name:** The name of the header indicating the header type. For valid values, see the following frame descriptions.
- **Header value type:** An enumeration indicating the header value type.
- **Value string byte length:** The byte-length of the header value string.
- **Header value:** The value of the header string. Valid values for this field depend on the type of header. For valid values, see the following frame descriptions.

## Enabling your bot to be interrupted by your user

When you start a bidirectional audio stream between an Amazon Lex V2 bot and your application, you can configure the bot to listen for user input while it is sending back a prompt. With this the user can interrupt the prompt before the bot has finished playing it back. You can use this configuration for

situations where the user might already know the answer to a question, such as when they're being prompted to provide a CVV code.

A bot knows when the user interrupts a prompt when it detects user input before your application can send back a `PlaybackCompletion` event. When the user interrupts a bot, the bot sends a `PlaybackInterruptionEvent`.

By default, the user can interrupt any prompt that the bot is streaming to your application. You can change this setting in the Amazon Lex V2 console.

You can change how a user can respond to a prompt by editing a slot. A slot is part of an intent, and it is the means by which the user provides you the information you want. Each slot has a prompt for the user to provide you with that information. To learn more about slots, see [How it works \(p. 3\)](#).

#### To change whether the user can interrupt a prompt (console)

1. Sign in to AWS Management Console and open the Amazon Lex V2 console at [Amazon Lex V2 console](#).
2. Under **Bots**, select a bot.
3. Under **Language**, select the language of the bot.
4. Choose **View intents**.
5. Choose the intent.
6. For **Slots**, choose a slot.
7. Under **Advanced options**, choose **Slot prompts**.
8. Choose **More prompt options**.
9. Select or deselect **Users can interrupt the prompt when it is being read**.

You can test this functionality by creating a bot with two slots and specifying that users can't interrupt a prompt for one slot. If you interrupt an interruptible prompt, the bot sends a playback interruption event. If you interrupt an uninterruptible, the prompt continues to play.

## Enabling the bot to wait for the user to provide more information

When you start a bidirectional stream from an Amazon Lex V2 bot to your application, you can configure the bot to wait for the user to provide additional information. There are circumstances when a user might not be ready to respond to a prompt. For example, a user might not be ready to provide their credit card information because their wallet is in another room.

By using the *Wait and continue* behavior of the Amazon Lex V2 bot, users can say phrases such as "hold on a second" to make the bot wait for them to find the information and provide it. When you enable this behavior, the bot sends periodic reminders to the user to provide the information. It does not send back transcript events because there are no user utterances for it to transcribe.

The Amazon Lex V2 bot automatically manages a streaming conversation. You don't have to write any additional code to enable this functionality. When a bot is prompted to wait by the user, the state of the Intent is `Waiting` and the type of the DialogAction is `ElicitSlot`. You can use this information to help customize your application for your needs. For example, you can configure your application to play music when the user is looking for their credit card.

You enable the wait and continue behavior for an individual slot. To learn more about slots, see [How it works \(p. 3\)](#).

### To enable wait and continue

1. Sign in to AWS Management Console and open the Amazon Lex V2 console at [Amazon Lex V2 console](#).
2. Under **Bots**, select a bot.
3. Under **Language**, select the language of the bot.
4. Choose **View intents**.
5. Choose the intent.
6. Under **Slots**, choose a slot.
7. Under **Advanced options**, choose **Wait and continue**.
8. Under **Wait and continue** specify the following fields:
  - **Response when user wants the bot to wait** – This is how the bot responds when the user asks it to wait for the additional information.
  - **Response if the user needs the bot to continue waiting** – This is the response the bot sends to remind the user that it's still waiting for the information. You can change how frequently the bot reminds the user.
  - **Response when the user wants to continue** – This is the bot's response when the user has the requested information.

For every bot response, you can give multiple variations of the response, and one is presented to the user at random. You can also choose whether these responses can be interrupted by the user.

To test the wait and continue functionality, configure your bot to wait for user input and start a stream to an Amazon Lex V2 bot. For information on streaming to a bot, see [Using the API to start a streaming conversation \(p. 258\)](#).

You may need to turn off the wait and continue responses. Use the **Active** toggle to set whether or not the wait and continue responses are used.

#### Wait and continue

You can use the responses below to manage a conversation if the user needs time to provide information requested functionality is available only in streaming conversations.

## Configuring fulfillment progress updates

When the fulfillment Lambda function for an intent is called, the bot doesn't send a response until the function completes. If the Lambda function takes more than a few seconds to complete, the user may think that the bot is unresponsive. To address this, you can configure your bot to send updates to the user while the fulfillment Lambda function is running so that the user knows that the bot is still working on their request.

When you add fulfillment updates to an intent, the bot responds at the start of fulfillment and periodically while fulfillment is in progress. When you configure the start response, you can specify a delay before the bot sends the response. With this, you can support cases where the fulfillment doesn't finish relatively quickly. When you configure an update response, you specify the frequency that you want the updates sent. You also configure a timeout to limit the time that the fulfillment function has to run.

You can also add post-fulfillment responses to a bot. This enables the bot to send a different response depending on whether fulfillment succeeds, fails, or times out.

Fulfillment updates are used only when interacting with a bot using the [StartConversation](#) operation. You can use the post-fulfillment update when interacting with the bot using the [StartConversation](#), [RecognizeText](#), and [RecognizeUtterance](#) operations.

## Fulfillment updates

Fulfillment updates are sent while your Lambda function is fulfilling an intent. When you turn on fulfillment updates, you provide a start response that is sent at the beginning of fulfillment and an update response that is sent periodically while fulfillment is in progress.

When you specify an update response, you also specify a timeout that determines how long the fulfillment function can run. You can specify a timeout length of up to 15 minutes (900 seconds).

If you turn off fulfillment updates by setting `active` to `false` in the console or using the [CreateIntent](#) or [UpdateIntent](#) operation, the timeout specified for the fulfillment updates isn't used and the default timeout of 30 seconds is used instead.

If the fulfillment function times out, Amazon Lex V2 does one of three things:

- Post-fulfillment response is configured and active – returns the timeout response.
- Post-fulfillment response is configured and not active – returns an exception.
- Post-fulfillment response isn't configured – returns an exception.

### Start response

Amazon Lex V2 returns the start response when the Lambda fulfillment function is called during a streaming conversation. It typically tells the user that fulfilling the intent takes some time and that they should wait. The start response isn't returned when you use the [RecognizeText](#) or [RecognizeUtterance](#) operations.

You can specify up to five response messages. Amazon Lex V2 chooses one of the messages to play to the user.

You can configure a delay between when the Lambda function is called and when the start response is returned. The start response isn't returned if the Lambda function completes its work before the delay is complete.

You can use the `active` toggle in the console or the [FulfillmentUpdatesSpecification](#) structure to turn the start response on and off. When `active` is `false`, the start response isn't played.

### Update response

Amazon Lex returns the update response periodically during a streaming conversation while the Lambda fulfillment function is running. The update response isn't played when you use the [RecognizeText](#) or [RecognizeUtterance](#) operations. You can configure how often the update response plays. For example, you can play an update response every 30 seconds while the fulfillment function runs to let the user know that the process is running and that they should continue to wait.

You can specify up to five update messages. Amazon Lex V2 chooses a message to play to the user. Using multiple messages keeps the updates from being repetitive.

If the user provides input via voice, DTMF, or text while the fulfillment Lambda function is running, Amazon Lex V2 returns the update response to the user.

If the Lambda function completes its work before the first update period ends, the update response isn't returned.

You can use the active toggle in the console or the [FulfillmentUpdatesSpecification](#) structure to turn the update response on and off. When active is false, the update response isn't returned.

## Post-fulfillment response

Amazon Lex V2 returns a post-fulfillment response when the fulfillment function ends. A post-fulfillment response can be used when fulfilling any intent, not just streaming conversations. The post-fulfillment response lets the user know that the function is complete and the result.

You can use the active toggle in the console or the [PostFulfillmentStatusSpecification](#) structure to turn the post-fulfillment response on and off. When active is false, the response is not played.

There are three types of post-fulfillment responses:

- Success – returned when the fulfillment Lambda function completes its work successfully. If post-fulfillment responses aren't active, Amazon Lex V2 takes the next configured action.
- Timeout – returned if the Lambda function doesn't complete its work before the configured timeout period elapses. If post-fulfillment responses aren't active, Amazon Lex V2 returns an exception.
- Failure – returned when the Lambda function returns the status Failed in the response or when Amazon Lex V2 encounters an error while fulfilling the intent. If post-fulfillment responses aren't active, Amazon Lex V2 returns an exception.

You can specify up to five messages for each type. Amazon Lex V2 chooses one of the messages to play to the user. If the Lambda function returns a post-fulfillment message, that message is used instead of the configured messages.

**Note**

If the intent has a closing response, the response is returned after the post-fulfillment response.

## Configuring timeouts for capturing user input

The Amazon Lex V2 streaming API enables a bot to automatically detect utterances in user input. When you create an intent or a slot, you can configure aspects of an utterance, such as maximum duration of an utterance, timeout while waiting for user input, or the end character for DTMF input. You can customize a bot's behavior for your use case. For example, you can limit the number of digits for a credit card number to 16.

You can also configure timeouts through session attributes when starting a conversation with a bot, and overwrite them in your Lambda function if necessary.

The configuration keys for an attribute use the following syntax:

```
x-amz-lex:<InputType>:<BehaviorName>:<IntentName>:<SlotName>
```

InputType can be **audio**, **dtmf**, or **text**.

You can configure default settings for all intents or slots in a bot by specifying \* as the intent or slot name. Any intent- or slot-specific settings take precedence over default settings.

Amazon Lex V2 provides predefined session attributes for managing the way the [StartConversation](#) operations works with text, voice, or DTMF input to your bot. All predefined attributes are in the x-amz-lex namespace.

You can configure default settings for all intents, slots or subslots in a bot by specifying \* as the intent or slot name. Any intent or slot-specific settings take precedence over default settings. Use these patterns for all the timeouts below.

For a composite slot's subslot you can separate by .. For example:

```
<slotName>.<subSlotName>
```

```
x-amz-lex:allow-interrupt:<intentName>:<slotName>.<subSlotName>
```

Expression	Scenario
Intent:Slot.SubSlot	Applicable to only sub slot named 'SubSlot' inside composite slot named 'Slot'
Intent:Slot.*	Applicable to any sub slot inside composite slot named 'Slot'
Intent:*.SubSlot	Applicable to only sub slot named 'SubSlot' inside any composite slot
Intent:*.*	Applicable to any sub slot inside any composite slot

## Interrupt behavior

You can set up the interrupt behavior for the bot. The attribute is defined by Amazon Lex V2.

Allow interrupt

```
x-amz-lex:allow-interrupt:<intentName>:<slotName>
```

Defines whether user can interrupt the prompt played by Amazon Lex V2 bot. You can selectively turn it off.

**Default:** True

## Timeouts for voice input

You can set time-out values for voice interaction with your bot using session attributes. The attributes are defined by Amazon Lex V2. These attributes enable you to specify how long Amazon Lex V2 waits for a customer to finish speaking before collecting input speech.

All of these attributes are in the x-amz-lex:audio namespace.

### Maximum utterance length

```
x-amz-lex:audio:max-length-ms:<intentName>:<slotName>
```

Defines how long Amazon Lex V2 waits before speech input is truncated and the speech is returned to your application. You can increase the length of the input when you expect long responses, or if you want to give customers more time to provide information.

**Default:** 13,000 milliseconds (13 seconds). The maximum value is 15,000 milliseconds (15 seconds)

If you set the max-length-ms attribute to more than 15,000 milliseconds, the value will default to 15,000 milliseconds.

## Voice timeout

```
x-amz-lex:audio:start-timeout-ms:<intentName>:<slotName>
```

How long a bot waits before assuming that the customer isn't going to speak. You can increase the time in situations where the customer may need more time to find or recall information before speaking. For example, you might want to give customers time to get out their credit card so they can enter the number.

**Default:** 4,000 milliseconds (4 seconds)

## Silence timeout

```
x-amz-lex:audio:end-timeout-ms:<intentName>:<slotName>
```

How long a bot waits after the customer stops speaking to assume the utterance is finished. You can increase the time in situations where periods of silence are expected while providing input.

**Default:** 600 milliseconds (0.6 seconds)

## Allow audio input

```
x-amz-lex:allow-audio-input:<intentName>:<slotName>
```

You can enable this attribute so that the bot accepts user input only via audio modality. The bot will not accept audio input if this flag is set to false. The value is set to true by default.

**Default:** True

## Timeouts for text input

Use the following session attribute to specify how your bot behaves with the text conversation mode.

This attribute is in the `x-amz-lex:text` namespace.

### Start timeout threshold

```
x-amz-lex:text:start-timeout-ms:<intentName>:<slotName>
```

How long the bot waits before re-prompting a customer for text input. You can increase the time in situations where you'd like to allow the customer more time to find or recall information before providing text input. For example, you might want to give customers more time to find details on their order. Alternatively, you may reduce the threshold to prompt customers earlier.

**Default:** 30,000 milliseconds (30 seconds)

## Configuration for DTMF input

Use the following session attributes to specify how your Amazon Lex V2 bot responds to DTMF input when using an audio conversation.

All of these attributes are in the `x-amz-lex:dtmf` namespace.

## Deletion character

```
x-amz-lex:dtmf:deletion-character:<intentName>:<slotName>
```

The DTMF character that clears the accumulated DTMF digits and immediately ends the input.

**Default:** \*

## End character

```
x-amz-lex:dtmf:end-character:<intentName>:<slotName>
```

The DTMF character that immediately ends input. If the user does not press this character, the input ends after the end timeout.

**Default:** #

## End timeout

```
x-amz-lex:dtmf:end-timeout-ms:<intentName>:<slotName>
```

How long the bot should wait from the last DTMF character input before assuming that the input has concluded.

**Default:** 5000 milliseconds (5 seconds)

## Maximum number of DTMF digits per utterance

```
x-amz-lex:dtmf:max-length:<intentName>:<slotName>
```

The maximum number of DTMF digits allowed in an utterance. For example, you could set this value to 16 to limit the number of characters that can be input for a credit card number. This value can't be increased.

**Default:** 1024 characters

## Allow DTMF input

You can set the type of input that the bot can accept using session attributes. The attributes are defined by Amazon Lex V2.

```
x-amz-lex:allow-dtmf-input:<intentName>:<slotName>
```

You can enable this attribute so that the bot accepts user input via DTMF modality. The bot will not accept DTMF input if this flag is set to false. The value is set to true by default.

**Default:** True

# Monitoring Amazon Lex V2

Monitoring is important for maintaining the reliability, availability, and performance of your Amazon Lex V2 chatbots. This topic describes using conversation logs to monitor conversations between your users and your chatbots, using utterance statistics to determine the utterances that your bots detect and miss, and how to use Amazon CloudWatch Logs and AWS CloudTrail to monitor Amazon Lex V2 and describes the Amazon Lex V2 runtime and channel association metrics.

## Topics

- [Monitoring with conversation logs \(p. 278\)](#)
- [Obscuring slot values in logs \(p. 286\)](#)
- [Viewing utterance statistics \(p. 287\)](#)
- [Logging Amazon Lex V2 API calls with AWS CloudTrail \(p. 288\)](#)
- [Monitoring Amazon Lex V2 with Amazon CloudWatch \(p. 290\)](#)

## Monitoring with conversation logs

You enable *conversation logs* to store bot interactions. You can use these logs to review the performance of your bot and to troubleshoot issues with conversations. You can log text for the [RecognizeText](#) operation. You can log both text and audio for the [RecognizeUtterance](#) operation. By enabling conversation logs, you get a detailed view of conversations that users have with your bot.

For example, a session with your bot has a session ID. You can use this ID to get the transcript of the conversation including user utterances and the corresponding bot responses. You also get metadata such as intent name and slot values for an utterance.

### Note

You can't use conversation logs with a bot subject to the Children's Online Privacy Protection Act (COPPA).

Conversation logs are configured for an alias. Each alias can have different settings for their text and audio logs. You can enable text logs, audio logs, or both for each alias. Text logs store text input, transcripts of audio input, and associated metadata in CloudWatch Logs. Audio logs store audio input in Amazon S3. You can enable encryption of text and audio logs using AWS KMS customer managed CMKs.

To configure logging, use the console or the [CreateBotAlias](#) or [UpdateBotAlias](#) operation. After enabling conversation logs for an alias, using the [RecognizeText](#) or [RecognizeUtterance](#) operation for that alias logs the text or audio utterances in the configured CloudWatch Logs log group or S3 bucket.

## Topics

- [Configuring conversation logs \(p. 278\)](#)
- [Viewing text logs in Amazon CloudWatch Logs \(p. 280\)](#)
- [Accessing audio logs in Amazon S3 \(p. 286\)](#)
- [Monitoring conversation log status with CloudWatch metrics \(p. 286\)](#)

## Configuring conversation logs

You enable and disable conversation logs using the console or the `conversationLogSettings` field of the [CreateBotAlias](#) or [UpdateBotAlias](#) operation. You can turn on or turn off audio logs, text logs, or both. Logging starts on new bot sessions. Changes to log settings aren't reflected for active sessions.

To store text logs, use an Amazon CloudWatch Logs log group in your AWS account. You can use any valid log group. The log group must be in the same region as the Amazon Lex V2 bot. For more information about creating a CloudWatch Logs log group, see [Working with Log Groups and Log Streams](#) in the *Amazon CloudWatch Logs User Guide*.

To store audio logs, use an Amazon S3 bucket in your AWS account. You can use any valid S3 bucket. The bucket must be in the same region as the Amazon Lex V2 bot. For more information about creating an S3 bucket, see [Creating a bucket](#) in the *Amazon Simple Storage Service Getting Started Guide*.

When you manage conversation logs using the console, the console updates your service role so that it has access to the log group and S3 bucket. If you are not using the console, you must provide an IAM role with policies that enable Amazon Lex V2 to write to the configured log group or bucket.

The IAM role that you use to enable conversation logs must have the `iam:PassRole` permission. The following policy should be attached to the role.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iam:PassRole",  
            "Resource": "arn:aws:iam::account:role/role"  
        }  
    ]  
}
```

## Enabling conversation logs

### To turn on logs using the console

1. Open the Amazon Lex V2 console <https://console.aws.amazon.com/lexv2>.
2. From the list, choose a bot.
3. From the left menu, choose **Aliases**.
4. In the list of aliases, choose the alias for which you want to configure conversation logs.
5. In the **Conversation logs** section, choose **Manage conversation logs**.
6. For text logs, choose **Enable** then enter the Amazon CloudWatch Logs log group name.
7. For audio logs, choose **Enable** then enter the S3 bucket information.
8. Optional. To encrypt audio logs, choose the AWS KMS key to use for encryption.
9. Choose **Save** to start logging conversations. If necessary, Amazon Lex V2 will update your service role with permissions to access the CloudWatch Logs log group and selected S3 bucket.

## Disabling conversation logs

### To turn off logs using the console

1. Open the Amazon Lex V2 console <https://console.aws.amazon.com/lexv2>.
2. From the list, choose a bot.
3. From the left menu, choose **Aliases**.
4. In the list of aliases, choose the alias for which you want to configure conversation logs.
5. In the **Conversation logs** section, choose **Manage conversation logs**.
6. Disable text logging, audio logging, or both to turn off logging.
7. Choose **Save** to stop logging conversations.

## Viewing text logs in Amazon CloudWatch Logs

Amazon Lex V2 stores text logs for your conversations in Amazon CloudWatch Logs. To view the logs, use the CloudWatch Logs console or API. For more information, see [Search Log Data Using Filter Patterns](#) and [CloudWatch Logs Insights Query Syntax](#) in the *Amazon CloudWatch Logs User Guide*.

### To view logs using the Amazon Lex V2 console

1. Open the Amazon Lex V2 console <https://console.aws.amazon.com/lexv2>.
2. From the list, choose a bot.
3. From the left menu, choose **Analytics** and then choose **CloudWatch metrics**.
4. View metrics for your bot on the **CloudWatch metrics** page.

You can also use the CloudWatch console or API to view your log entries. To find the log entries, navigate to the log group that you configured for the alias. You can find the log stream prefix for your logs in the Amazon Lex V2 console or by using the [DescribeBotAlias](#) operation.

Log entries for a user utterance are found in multiple log streams. An utterance in the conversation has an entry in one of the log streams with the specified prefix. An entry in the log stream contains the following information.

message-version

The message schema version.

bot

Details about the bot that the customer is interacting with.

messages

The response that the bot sent back to the user.

utteranceContext

Information about processing this utterance.

- runtimeHints—runtime context used to transcribe and interpret the user's input. For more information, see [Using runtime hints to improve recognition of slot values \(p. 246\)](#).
- slotElicitationStyle—Slot elicitation style used to interpret user input. For more information, see [Using spelling styles to capture slot values \(p. 248\)](#).

sessionState

The current state of the conversation between the user and the bot. For more information, see [Managing conversations \(p. 227\)](#).

interpretations

A list of intents that Amazon Lex V2 determined could satisfy the user's utterance. [Using confidence scores \(p. 237\)](#).

sessionId

The identifier of the user session that is having the conversation.

inputTranscript

The transcript of the user input used to recognize intent and slot values.

transcriptions

A list of potential transcriptions of the user's input. For more information, see [Using voice transcription confidence scores \(p. 240\)](#).

**missedUtterance**

Indicates whether Amazon Lex V2 was able to recognize the user's utterance.

**requestId**

Amazon Lex V2 generated request ID for the user input.

**timestamp**

The timestamp of the user's input.

**developerOverride**

Indicates whether the conversation flow was updated using a dialog code hook. For more information on using a dialog code hook, see [Using an AWS Lambda function \(p. 194\)](#).

**inputMode**

Indicates the type of input. Can be audio, DTMF, or text.

**requestAttributes**

The request attributes used when processing the user's input.

**audioProperties**

If audio conversation logs are enabled and the user input was in audio format, includes the total duration of the audio input, the duration of voice and the duration of silence in the audio. It also includes a link to the audio file.

**bargain**

Indicates whether the user input interrupted the previous bot response.

**responseReason**

The reason a response was generated. Can be one of:

- UtteranceResponse – response to user input
- StartTimeout – server generated response when the user didn't provide input
- StillWaitingResponse – server generated response when the user requests the bot wait
- FulfillmentInitiated – server generated response that fulfillment is about to be initiated
- FulfillmentStartedResponse – server generated response that fulfillment has begun
- FulfillmentUpdateResponse – periodic server generated response while fulfillment is in progress
- FulfillmentCompletedResponse – server generated response when fulfillment is complete.

**operationName**

The API used to interact with the bot. Can be one of PutSession, RecognizeText, RecognizeUtterance, or StartConversation.

```
{  
    "message-version": "2.0",  
    "bot": {  
        "id": "string",  
        "name": "string",  
        "aliasId": "string",  
        "aliasName": "string",  
        "localeId": "string",  
        "version": "string"  
    },  
    "messages": [  
        {  
            "contentType": "PlainText | SSML | CustomPayload | ImageResponseCard",  
            "content": "string"  
        }  
    ]  
}
```

```
"content": "string",
"imageResponseCard": {
    "title": "string",
    "subtitle": "string",
    "imageUrl": "string",
    "buttonsList": [
        {
            "text": "string",
            "value": "string"
        }
    ]
},
"utteranceContext": {
    "activeRuntimeHints": {
        "slotHints": {
            "string": {
                "string": {
                    "runtimeHintValues": [
                        {
                            "phrase": "string"
                        },
                        {
                            "phrase": "string"
                        }
                    ]
                }
            }
        }
    }
},
"slotElicitationStyle": "string"
},
"sessionState": {
    "dialogAction": {
        "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot",
        "slotToElicit": "string"
    },
    "intent": {
        "name": "string",
        "slots": {
            "string" : {
                "value": {
                    "interpretedValue": "string",
                    "originalValue": "string",
                    "resolvedValues": [ "string" ]
                }
            }
        },
        "string": {
            "shape": "List",
            "value": {
                "originalValue": "string",
                "interpretedValue": "string",
                "resolvedValues": [ "string" ]
            }
        },
        "values": [
            {
                "shape": "Scalar",
                "value": {
                    "originalValue": "string",
                    "interpretedValue": "string",
                    "resolvedValues": [ "string" ]
                }
            }
        ],
        "shape": "Scalar",
    }
}
```

```

        "value": {
            "originalValue": "string",
            "interpretedValue": "string",
            "resolvedValues": [ "string" ]
        }
    }
},
"kendraResponse": {
    // Only present when intent is KendraSearchIntent. For details, see
    // https://docs.aws.amazon.com/kendra/latest/dg/
API_Query.html#API_Query_ResponseSyntax
},
"state": "InProgress | ReadyForFulfillment | Fulfilled | Failed",
"confirmationState": "Confirmed | Denied | None"
},
"originatingRequestId": "string",
"sessionAttributes": {
    "string": "string"
},
"runtimeHints": {
    "slotHints": {
        "string": {
            "string": {
                "runtimeHintValues": [
                    {
                        "phrase": "string"
                    },
                    {
                        "phrase": "string"
                    }
                ]
            }
        }
    }
},
"dialogEventLogs": [
    {
        // only for conditional
        "conditionalEvaluationResult": [
            // all the branches until true

            {
                "conditionalBranchName": "String",
                "expressionString": "String",
                "evaluatedExpression": " String",
                "evaluationResult": true/false
            }
        ],
        "dialogCodeHookInvocationLabel": String,
        "response": String,
        "nextStep": {
            "dialogAction": {
                "type": "Close | ConfirmIntent | Delegate | ElicitIntent | ElicitSlot",
                "slotToElicit": "string"
            },
            "intent": {
                "name": "string",
                "slots": {
                    ...
                }
            }
        }
    ]
},
"interpretations": [

```

```
{
    "nluConfidence": "string",
    "intent": {
        "name": "string",
        "slots": {
            "string": {
                "value": {
                    "originalValue": "string",
                    "interpretedValue": "string",
                    "resolvedValues": [ "string" ]
                }
            },
            "string": {
                "shape": "List",
                "value": {
                    "interpretedValue": "string",
                    "originalValue": "string",
                    "resolvedValues": [ "string" ]
                }
            },
            "values": [
                {
                    "shape": "Scalar",
                    "value": {
                        "interpretedValue": "string",
                        "originalValue": "string",
                        "resolvedValues": [ "string" ]
                    }
                },
                {
                    "shape": "Scalar",
                    "value": {
                        "interpretedValue": "string",
                        "originalValue": "string",
                        "resolvedValues": [ "string" ]
                    }
                }
            ]
        }
    },
    "kendraResponse": {
        // Only present when intent is KendraSearchIntent. For details, see
        // https://docs.aws.amazon.com/kendra/latest/dg/
        API_Query.html#API_Query_ResponseSyntax
    },
    "state": "InProgress | ReadyForFulfillment | Fulfilled | Failed",
    "confirmationState": "Confirmed | Denied | None"
},
    "sentimentResponse": {
        "sentiment": "string",
        "sentimentScore": {
            "positive": "string",
            "negative": "string",
            "neutral": "string",
            "mixed": "string"
        }
    }
},
    "sessionId": "string",
    "inputTranscript": "string",
    "transcriptions": [
        {
            "transcription": "string",
            "transcriptionConfidence": {
                "score": "number"
            }
        }
    ]
}
```

```
        },
        "resolvedContext": {
            "intent": "string"
        },
        "resolvedSlots": {
            "string": {
                "name": "slotName",
                "shape": "List",
                "value": {
                    "originalValue": "string",
                    "resolvedValues": [
                        "string"
                    ]
                }
            }
        }
    ],
    "missedUtterance": "bool",
    "requestId": "string",
    "timestamp": "string",
    "developerOverride": "bool",
    "inputMode": "DTMF | Speech | Text",
    "requestAttributes": {
        "string": "string"
    },
    "audioProperties": {
        "contentType": "string",
        "s3Path": "string",
        "duration": {
            "total": "integer",
            "voice": "integer",
            "silence": "integer"
        }
    },
    "bargeIn": "string",
    "responseReason": "string",
    "operationName": "string"
}
```

The contents of the log entry depend on the result of a transaction and the configuration of the bot and request.

- The intent, slots, and slotToElicit fields don't appear in an entry if the missedUtterance field is true.
- The s3PathForAudio field doesn't appear if audio logs are disabled or if the inputDialogMode field is Text.
- The responseCard field only appears when you have defined a response card for the bot.
- The requestAttributes map only appears if you have specified request attributes in the request.
- The kendraResponse field is only present when the AMAZON.KendraSearchIntent makes a request to search an Amazon Kendra index.
- The developerOverride field is true when an alternative intent was specified in the bot's Lambda function.
- The sessionAttributes map only appears if you have specified session attributes in the request.
- The sentimentResponse map only appears if you configure the bot to return sentiment values.

**Note**

The input format may change without a corresponding change in the messageVersion. Your code should not throw an error if new fields are present.

## Accessing audio logs in Amazon S3

Amazon Lex V2 stores audio logs for your conversations in an S3 bucket.

You can use the Amazon S3 console or API to access audio logs. You can see the S3 object key prefix of the audio files in the Amazon Lex V2 console, or in the conversationLogSettings field in the `DescribeBotAlias` operation response.

## Monitoring conversation log status with CloudWatch metrics

Use Amazon CloudWatch to monitor delivery metrics of your conversation logs. You can set alarms on metrics so that you are aware of issues with logging if they should occur.

Amazon Lex V2 provides four metrics in the AWS/Lex namespace for conversation logs:

- `ConversationLogsAudioDeliverySuccess`
- `ConversationLogsAudioDeliveryFailure`
- `ConversationLogsTextDeliverySuccess`
- `ConversationLogsTextDeliveryFailure`

The success metrics show that Amazon Lex V2 has successfully written your audio or text logs to their destinations.

The failure metrics show that Amazon Lex V2 couldn't deliver audio or text logs to the specified destination. Typically, this is a configuration error. When your failure metrics are above zero, check the following:

- Make sure that Amazon Lex V2 is a trusted entity for the IAM role.
- For text logging, make sure that the CloudWatch Logs log group exists. For audio logging, make sure that the S3 bucket exists.
- Make sure that the IAM role that Amazon Lex V2 uses to access the CloudWatch Logs log group or S3 bucket has write permission for the log group or bucket.
- Make sure that the S3 bucket exists in the same region as the Amazon Lex V2 bot and belongs to your account.

## Obscuring slot values in logs

Amazon Lex V2 enables you to obfuscate, or hide, the contents of slots so that the content is not visible. To protect sensitive data captured as slot values, you can enable slot obfuscation to mask those values for logging.

When you choose to obfuscate slot values, Amazon Lex V2 replaces the value of the slot with the name of the slot in conversation logs. For a slot called `full_name`, the value of the slot would be obfuscated as follows:

```
Before:  
  My name is John Stiles  
After:  
  My name is {full_name}
```

If an utterance contains bracket characters ({}), Amazon Lex V2 escapes the bracket characters with two back slashes (\\\). For example, the text `{John Stiles}` is obfuscated as follows:

Before:

My name is {John Stiles}

After:

My name is \\{{full\_name}}\\

Slot values are obfuscated in conversation logs. The slot values are still available in the response from the RecognizeText and RecognizeUtterance operations, and the slot values are available to your validation and fulfillment Lambda functions. If you are using slot values in your prompts or responses, those slot values are not obfuscated in conversation logs.

In the first turn of a conversation, Amazon Lex V2 obfuscates slot values if it recognizes a slot and slot value in the utterance. If no slot value is recognized, Amazon Lex V2 does not obfuscate the utterance.

On the second and later turns, Amazon Lex V2 knows the slot to elicit and if the slot value should be obfuscated. If Amazon Lex V2 recognizes the slot value, the value is obfuscated. If Amazon Lex V2 does not recognize a value, the entire utterance is obfuscated. Any slot values in missed utterances won't be obfuscated.

Amazon Lex V2 also doesn't obfuscate slot values that you store in request or session attributes. If you are storing slot values that should be obfuscated as an attribute, you must encrypt or otherwise obfuscate the value.

Amazon Lex V2 doesn't obfuscate the slot value in audio. It does obfuscate the slot value in the audio transcription.

All slots are obfuscated by default. However, you don't need to obfuscate all of the slots in a bot. You can choose which slots obfuscate using the console or by using the Amazon Lex V2 API. In the console, choose **Slot obfuscation** in the settings for a slot. If you are using the API, set the obfuscationSetting field of the slot to DEFAULT\_OBFUSCATION when you call the [CreateSlot](#) or [UpdateSlot](#) operation.

## Viewing utterance statistics

You can use utterance statistics to determine the utterances that your users are sending to your bot. You can see both the utterances that Amazon Lex V2 successfully detects as well as the utterances that it does not. You can use this information to help tune your bot.

For example, if you find that your users are sending an utterance that Amazon Lex V2 is missing, you can add the utterance to an intent. The Draft version of the intent is updated with the new utterance and you can test it before deploying it to your bot.

An utterance is detected when Amazon Lex V2 recognizes the utterance as an attempt to invoke an intent configured for a bot. An utterance is missed when Amazon Lex V2 doesn't recognize the utterance and invokes the AMAZON.FallbackIntent instead.

Utterance statistics are not generated under the following conditions:

- The Child Online Privacy Protection Act setting was set to **Yes** when the bot was created with the console, or the childDirected field was set to true when the bot was created with the CreateBot operation.
- You are using slot obfuscation with one or more slots.
- You opted out of participating in improving Amazon Lex.

Using utterance statistics, you can see whether a specific utterance was detected or missed, as well as the last time that the utterance was used in a bot interaction.

Amazon Lex V2 stores utterances continuously while users interact with your bot. You can query the statistics using the console or the [ListAggregatedUtterances](#) operation. Utterances are stored for 15 days for use with utterance statistics, and then indefinitely for use in improving the ability of your bot to respond to user input. You can delete utterances using the [DeleteUtterances](#) operation or by opting out of data storage. All utterances are deleted if you close your AWS account. Stored utterances are encrypted with a server-managed key.

Utterance statistics are generated for two combinations:

- Bot, language, version
- Bot, language, alias

When you delete a bot version, utterance statistics are available for the version for up to 15 days. You can't see statistics for deleted version in the Amazon Lex V2 console. To see the statistics for deleted versions, use the [ListAggregatedUtterances](#) operation.

If an alias points to a deleted version, you still get statistics from the deleted version when you get statistics for the alias. If the statistics from the [ListAggregatedUtterances](#) operation contains information from a deleted version, the `containsDataFromDeletedResources` field is set to true.

You can choose the time period that you want to see statistics for. You can choose to see hours, days, or weeks of information.

- **Hours** – You can request utterance statistics for 1, 3, 6, 12, or 24 hour time windows. Statistics are refreshed every half hour for 1 hour time windows, and hourly for the other time windows.
- **Days** – You can request utterance statistics for 3 days. Statistics are refreshed every 6 hours.
- **Weeks** – You can see statistics for one or two weeks. Statistics are refreshed every 12 hours for one week time windows, and once per day for two week time windows.

Utterances are aggregated by the text of the utterance. For example, all instances where the customer used the phrase "I want to order a pizza" are aggregated into the same line in a response. When you use the [RecognizeUtterance](#) operation, the text used is the input transcript.

To list aggregated utterances, apply the following policy to a role.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ListAggregatedUtterancesPolicy",  
            "Effect": "Allow",  
            "Action": "lex>ListAggregatedUtterances",  
            "Resource": "*"  
        }  
    ]  
}
```

## Logging Amazon Lex V2 API calls with AWS CloudTrail

Amazon Lex V2 is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Amazon Lex V2. CloudTrail captures API calls for Amazon Lex V2 as events. The calls captured include calls from the Amazon Lex V2 console and code calls to the Amazon

Lex V2 API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Amazon Lex V2. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Amazon Lex V2, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the [AWS CloudTrail User Guide](#).

## Amazon Lex V2 information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon Lex V2, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing events with CloudTrail Event history](#).

For an ongoing record of events in your AWS account, including events for Amazon Lex V2, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
  - CloudTrail supported services and integrations
  - Configuring Amazon SNS notifications for CloudTrail
  - Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

Amazon Lex V2 supports logging for all of the actions listed in [Model Building API V2](#).

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
  - Whether the request was made with temporary security credentials for a role or federated user.
  - Whether the request was made by another AWS service.

For more information, see the [CloudTrail userIdentity element](#).

## Understanding Amazon Lex V2 log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the [CreateBotAlias](#) action.

```
{  
    "eventVersion": "1.05",  
    "userIdentity": {  
        "type": "AssumedRole",
```

```

"principalId": "ID of caller:temporary credentials",
"arn": "arn:aws:sts::111122223333:assumed-role/role name/role ARN",
"accountId": "111122223333",
"accessKeyId": "AKIAIOSFODNN7EXAMPLE",
"sessionContext": {
    "sessionIssuer": {
        "type": "Role",
        "principalId": "ID of caller",
        "arn": "arn:aws:iam::111122223333:role/role name",
        "accountId": "111122223333",
        "userName": "role name"
    },
    "webIdFederationData": {},
    "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "creation date"
    }
},
"eventTime": "event timestamp",
"eventSource": "lex.amazonaws.com",
"eventName": "CreateBotAlias",
"awsRegion": "Region",
"sourceIPAddress": "192.0.2.0",
"userAgent": "user agent",
"requestParameters": {
    "botAliasLocaleSettingsMap": {
        "en_US": {
            "enabled": true
        }
    },
    "botId": "bot ID",
    "botAliasName": "bot alias name",
    "botVersion": "1"
},
"responseElements": {
    "botAliasLocaleSettingsMap": {
        "en_US": {
            "enabled": true
        }
    },
    "botAliasId": "bot alias ID",
    "botAliasName": "bot alias name",
    "botId": "bot ID",
    "botVersion": "1",
    "creationDateTime": creation timestamp
},
"requestID": "unique request ID",
"eventID": "unique event ID",
"readOnly": false,
"eventType": "AwsApiCall",
"recipientAccountId": "111122223333"
}

```

## Monitoring Amazon Lex V2 with Amazon CloudWatch

You can monitor Amazon Lex V2 using CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your web application or service is performing. You

can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the [Amazon CloudWatch User Guide](#).

The Amazon Lex V2 service reports the following metrics in the AWS/Lex namespace.

Metric	Description
KendraIndexAccessErrors	<p>The number of times that Amazon Lex V2 could not access your Amazon Kendra index.</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>
KendraLatency	<p>The amount of time that it takes Amazon Kendra to respond to a request from the AMAZON.KendraSearchIntent.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Milliseconds</p>
KendraSuccess	<p>The number of times that Amazon Lex V2 couldn't access your Amazon Kendra index.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>
KendraSystemErrors	<p>The number of times that Amazon Lex V2 couldn't query the Amazon Kendra index.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Unit: Count</p>
KendraThrottledEvents	<p>The number of times Amazon Kendra throttled requests from the AMAZON.KendraSearchIntent.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Unit: Count</p>
MissedUtteranceCount	<p>The number of utterances that were not recognized in the specified time period.</p> <p>Valid dimensions:</p>

Metric	Description
	<ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimeConcurrency	<p>The number of concurrent connections in the specified time period. <code>RuntimeConcurrency</code> is reported as a <code>StatisticSet</code>.</p> <p>Valid dimensions for the <code>RecognizeUtterance</code> or <code>StartConversation</code> operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Valid dimensions for other operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimeInvalidLambdaResponses	<p>The number of invalid AWS Lambda responses in the specified period.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimeLambdaErrors	<p>The number of Lambda runtime errors in the specified time period.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimePollyErrors	<p>The number of invalid Amazon Polly responses in the specified time period.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Unit: Count</p>

Metric	Description
RuntimeRequestCount	<p>The number of runtime requests in the specified time period.</p> <p>Valid dimensions for the <code>RecognizeUtterance</code> and <code>StartConversation</code> operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Valid dimensions for other operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimeRequestLength	<p>Total length of a conversation with a Amazon Lex V2 bot. Only applicable to the <code>StartConversation</code> operation.</p> <p>Valid dimensions:</p> <ul style="list-style-type: none"> <li>• BotAliasId, BotId, LocaleId, Operation</li> <li>• BotId, BotAliasId, LocaleId, Operation</li> </ul> <p>Unit: milliseconds</p>
RuntimeSuccessfulRequestLatency	<p>The latency for successful requests between the time the request was made and the response was passed back.</p> <p>Valid dimensions for the <code>RecognizeUtterance</code> and <code>StartConversation</code> operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Valid dimensions for other operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: milliseconds</p>

Metric	Description
RuntimeSystemErrors	<p>The number of system errors in the specified period. The response code range for a system error is 500 to 599.</p> <p>Valid dimensions for the RecognizeUtterance and StartConversation operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Valid dimensions for other operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimeThrottledEvents	<p>The number of throttled events. Amazon Lex V2 throttles an event when it receives more requests than the limit of transactions per second set for your account. If the limit set for your account is frequently exceeded, you can request a limit increase. To request an increase, see <a href="#">AWS service limits</a>.</p> <p>Valid dimensions for the RecognizeUtterance and StartConversation operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Valid dimensions for other operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>
RuntimeUserErrors	<p>The number of user errors in the specified period. The response code range for a user error is 400 to 499.</p> <p>Valid dimensions for the RecognizeUtterance and StartConversation operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, InputMode, LocaleId</li> <li>• Operation, BotId, BotAliasId, InputMode, LocaleId</li> </ul> <p>Valid dimensions for other operations:</p> <ul style="list-style-type: none"> <li>• Operation, BotId, BotVersion, LocaleId</li> <li>• Operation, BotId, BotAliasId, LocaleId</li> </ul> <p>Unit: Count</p>

The following dimensions are supported for the Amazon Lex V2 metrics.

Dimension	Description
Operation	The name of the Amazon Lex V2 operation – RecognizeText, RecognizeUtterance, StartConversation, GetSession, PutSession, DeleteSession – that generated the entry.
BotId	The alphanumeric unique identifier for the bot.
BotAliasId	The alphanumeric unique identifier for the bot alias.
BotVersion	The numeric version of the bot.
InputMode	The type of input to the bot – speech, text, or DTMF.
LocaleId	The identifier of the bot's locale, such as en-US or fr-CA.

# Importing and exporting

You can export a bot definition, a bot locale or a custom vocabulary and then import it back to create a new resource or to overwrite an existing resource in an AWS account. For example, you can export a bot from a test account and then create a copy of the bot in your production account. You can also copy a bot from one AWS Region to another Region.

You can change the resources of the exported resource before importing it. For example, you can export a bot and then edit the JSON file for a slot to add or remove slot value elicitation utterances from a specific slot. After you finish editing the definition, you can import the modified file.

## Topics

- [Exporting \(p. 296\)](#)
- [Importing \(p. 298\)](#)
- [Using a password when importing or exporting \(p. 301\)](#)
- [JSON format for importing and exporting \(p. 301\)](#)

## Exporting

You export a bot, bot locale, or custom vocabulary using the console or the `StartExport` operation. You specify the resource to export, and you can provide an optional password to help protect the .zip file when you start an export. After you download the .zip file, you must use the password to access the file before you can use it. For more information, see [Using a password when importing or exporting \(p. 301\)](#).

Exporting is an asynchronous operation. Once you have started the export, you can use the console or the `DescribeExport` operation to monitor the progress of the export. Once the export is complete, the console or the `DescribeExport` operation shows a status of COMPLETED, and the console downloads the export .zip file to your browser. If you use the `DescribeExport` operation, Amazon Lex V2 provides a pre-signed Amazon S3 URL where you can download the results of the export. The download URL is available for only five minutes, but you can get a new URL by calling the `DescribeExport` operation again.

You can see the history of exports for a resource with the console or with the `ListExports` operation. The results show the exports along with their current status. An export is available in the history for seven days.

When you export the Draft version of a bot or a bot locale, it is possible for the definition in the JSON file to be in an inconsistent state because the Draft version of a bot or bot locale can be changed while an export is in progress. If the Draft version is changed while it is being exported, the changes may not be included in the export file.

When you export a bot locale, Amazon Lex exports all of the information that defines the locale, including the locale, custom vocabulary, intents, slot types, and slots.

When you export a bot, Amazon Lex exports all of the locales defined for the bot, including the intents, slot types, and slots. The following items are not exported with a bot:

- Bot aliases
- Role ARN associated with a bot
- Tags associated with bots and bot aliases
- Lambda code hooks associated with a bot alias

The role ARN and tags are entered as request parameters when you import a bot. You need to create bot aliases and assign Lambda code hooks after importing, if necessary.

You can remove an export and the associated .zip file using the console or the `DeleteExport` operation.

For an example of exporting a bot using the console, see [Exporting a bot \(console\) \(p. 297\)](#).

## IAM permissions required to export

To export bots, bot locales, and custom vocabularies, the user running the export must have the following IAM permissions.

API	• Required IAM actions	Resource
<a href="#">CreateExport</a>	• <code>CreateExport</code>	Bot
<a href="#">UpdateExport</a>	• <code>UpdateExport</code>	Bot
<a href="#">DescribeExport</a>	<ul style="list-style-type: none"> <li>• <code>DescribeExport</code></li> <li>• <code>DescribeBot</code></li> <li>• <code>DescribeCustomVocabulary</code></li> <li>• <code>DescribeLocale</code></li> <li>• <code>DescribeIntent</code></li> <li>• <code>DescribeSlot</code></li> <li>• <code>DescribeSlotType</code></li> <li>• <code>ListLocale</code></li> <li>• <code>ListIntent</code></li> <li>• <code>ListSlot</code></li> <li>• <code>ListSlotType</code></li> </ul>	Bot
<a href="#">DescribeExport</a> for custom vocabularies	<ul style="list-style-type: none"> <li>• <code>DescribeExport</code></li> <li>• <code>DescribeCustomVocabulary</code></li> </ul>	bot
<a href="#">DeleteExport</a>	• <code>DeleteExport</code>	Bot
<a href="#">ListExports</a>	• <code>ListExports</code>	*

For an example IAM policy, see [Allow a user to export bots and bot locales \(p. 329\)](#).

## Exporting a bot (console)

You can export a bot from the bot list, from the list of versions, or from the version details page. When you choose a version, Amazon Lex V2 exports that version. The following instructions assume that you start exporting the bot from the list of bots, but when you start with a version the steps are the same.

### To export a bot using the console

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at <https://console.aws.amazon.com/lexv2/home>.
2. From the list of bots, choose the bot to export.
3. From **Action**, choose **Export**.
4. Choose the bot version, platform, and export format.
5. (Optional) Enter a password for the .zip file. Providing a password helps protect the output archive.

## 6. Choose Export.

After you start the export, you return to the list of bots. To monitor the progress of the export, use the **Import/export history** list. When the status of the export is **Complete**, the console automatically downloads the .zip file to your computer.

To download the export again, on the import/export list, choose the export, and then choose **Download**. You can provide a password for the downloaded .zip file.

### To export a bot language

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at <https://console.aws.amazon.com/lexv2/home>.
2. From the list of bots, choose the bot whose language you want to export.
3. From **Add languages**, choose **View languages**.
4. From the **All languages** list, choose the language to export.
5. From **Action**, choose **Export**.
6. Choose the bot version, platform, and format.
7. (Optional) Enter a password for the .zip file. Providing a password helps protect the output archive.
8. Choose **Export**.

After you start the export, you return to the list of languages. To monitor the progress of the export, use the **Import/export history** list. When the status of the export is **Complete**, the console automatically downloads the .zip file to your computer.

To download the export again, on the import/export list, choose the export, and then choose **Download**. You can provide a password for the downloaded .zip file.

# Importing

To use the console to import a previously exported bot, bot locale or custom vocabulary, you provide the file location on your local computer and the optional password to unlock the file. For an example, see [Importing a bot \(console\) \(p. 300\)](#).

When you use the API, importing a resource is a three step process:

1. Create an upload URL using the `CreateUploadUrl` operation. You don't need to create an upload URL when you are using the console.
2. Upload the .zip file that contains the resource definition.
3. Start the import with the `StartImport` operation.

The upload URL is a pre-signed Amazon S3 URL with write permission. The URL is available for five minutes after it is generated. If you password protect the .zip file, you must provide the password when you start the import. For more information, see [Using a password when importing or exporting \(p. 301\)](#).

An import is an asynchronous process. You can monitor the progress of an import using the console or the `DescribeImport` operation.

When you import a bot or bot locale, there may be conflicts between resource names in the import file and existing resource names in Amazon Lex V2. Amazon Lex V2 can handle the conflict in three ways:

- **Fail on conflict** – The import stops and no resources are imported from the import .zip file.

- **Overwrite** – Amazon Lex V2 imports all of the resources from the import .zip file and replaces any existing resource with the definition from the import file.
- **Append** – Amazon Lex V2 imports all of the resources from the import .zip file and adds to any existing resource with the definition from the import file. This is available only for the bot locale.

You can see a list of the imports to a resource using the console or the `ListImports` operation. Imports remain in the list for seven days. You can use the console or the `DescribeImport` operation to see details about a specific import.

You can also remove an import and the associated .zip file using the console or the `DeleteImport` operation.

For an example of importing a bot using the console, see [Importing a bot \(console\) \(p. 300\)](#).

## IAM permissions required to import

To import bots, bot locales, and custom vocabularies, the user running the import must have the following IAM permissions.

API	Required IAM actions	Resource
<a href="#">CreateUploadUrl</a>	<ul style="list-style-type: none"> <li>• <code>CreateUploadUrl</code></li> </ul>	*
<a href="#">StartImport</a> for bot and bot locale	<ul style="list-style-type: none"> <li>• <code>StartImport</code></li> <li>• <code>iam:PassRole</code></li> <li>• <code>CreateBot</code></li> <li>• <code>CreateCustomVocabulary</code></li> <li>• <code>CreateLocale</code></li> <li>• <code>CreateIntent</code></li> <li>• <code>CreateSlot</code></li> <li>• <code>CreateSlotType</code></li> <li>• <code>UpdateBot</code></li> <li>• <code>UpdateCustomVocabulary</code></li> <li>• <code>UpdateLocale</code></li> <li>• <code>UpdateIntent</code></li> <li>• <code>UpdateSlot</code></li> <li>• <code>UpdateSlotType</code></li> <li>• <code>DeleteBot</code></li> <li>• <code>DeleteCustomVocabulary</code></li> <li>• <code>DeleteLocale</code></li> <li>• <code>DeleteIntent</code></li> <li>• <code>DeleteSlot</code></li> <li>• <code>DeleteSlotType</code></li> </ul>	1. To import a new bot: bot, bot alias. 2. To overwrite an existing bot: bot. 3. To import a new locale: bot.
<a href="#">StartImport</a> for custom vocabularies	<ul style="list-style-type: none"> <li>• <code>StartImport</code></li> <li>• <code>CreateCustomVocabulary</code></li> <li>• <code>DeleteCustomVocabulary</code></li> <li>• <code>UpdateCustomVocabulary</code></li> </ul>	bot
<a href="#">DescribeImport</a>	<ul style="list-style-type: none"> <li>• <code>DescribeImport</code></li> </ul>	Bot

API	Required IAM actions	Resource
<a href="#">DeleteImport</a>	<ul style="list-style-type: none"> <li>DeleteImport</li> </ul>	Bot
<a href="#">ListImports</a>	<ul style="list-style-type: none"> <li>ListImports</li> </ul>	*

For an example IAM policy, see [Allow a user to import bots and bot locales \(p. 330\)](#).

## Importing a bot (console)

### To import a bot using the console

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at <https://console.aws.amazon.com/lexv2/home>.
2. From **Action**, choose **Import**.
3. In **Input file**, give the bot a name and then choose the .zip file that contains the JSON files that define the bot.
4. If the .zip file is password protected, enter the password for the .zip file. Password protecting the archive is optional, but it helps protect the contents.
5. Create or enter the IAM role that defines permissions for your bot.
6. Indicate whether your bot is subject to the Children's Online Privacy Protection Act (COPPA).
7. Provide an idle timeout setting for your bot. If you don't provide a value, the value from the zip file is used. If the .zip file does not contain a timeout setting, Amazon Lex V2 uses the default of 300 seconds (five minutes).
8. (Optional) Add tags for your bot.
9. Choose whether to warn about overwriting existing bots with the same name. If you enable warnings, if the bot you are importing would overwrite an existing bot, you receive a warning and the bot is not imported. If you disable warnings, the imported bot replaces the existing bot with the same name.
10. Choose **Import**.

After you start the import, you return to the list of bots. To monitor the progress of the import, use the **Import/export history** list. When the status of the import is **Complete**, you can choose the bot from the list of bots to modify or build the bot.

### To import a bot language

1. Sign in to the AWS Management Console and open the Amazon Lex V2 console at <https://console.aws.amazon.com/lexv2/home>.
2. From the list of bots, choose the bot to which you want to import a language.
3. From **Add languages**, choose **View languages**.
4. From **Action**, choose **import**.
5. In **Input file**, choose the file that contains the language to import. If you protected the .zip file, provide the password in **Password**.
6. In **Language**, choose the language to import as. The language doesn't have to match the language in the import file. You can copy the intents from one language to another.
7. In **Voice**, choose the Amazon Polly voice to use for voice interaction, or choose **None** for a text-only bot.
8. In **Confidence score threshold**, enter the threshold where Amazon Lex V2 inserts the AMAZON.FallbackIntent, the AMAZON.KendraSearchIntent, or both when returning alternative intents.

9. Choose whether to warn about overwriting an existing language. If you enable warnings, if the language you are importing would overwrite an existing language, you receive a warning and the language is not imported. If you disable warnings, the imported language replaces the existing language.
10. Choose **Import** to start importing the language.

After you start the import, you return to the list of languages. To monitor the progress of the import, use the **Import/export history** list. When the status of the import is **Complete**, you can choose the language from the list of bots to modify or build the bot.

## Using a password when importing or exporting

Amazon Lex V2 can password protect your export archives or read your protected import archives using standard .zip file compression. You should always password protect your import and export archives.

Amazon Lex V2 sends your export archive to an S3 bucket, and it is available to you with a pre-signed S3 URL. The URL is only available for five minutes. The archive is available to anyone with access to the download URL. To help protect the data in the archive, provide a password when you export the resource. If you need to get the archive after the URL expires, you can use the console or the `DescribeExport` operation to get a new URL.

If you lose the password for an export archive, you can create a new password for an existing file by choosing **Download** from the import/export history table or by using the `UpdateExport` operation. If you choose **Download** in the history table for an export and you don't provide a password, Amazon Lex V2 downloads an unprotected zip file.

## JSON format for importing and exporting

You import and export bots, bot locales, or custom vocabularies from Amazon Lex V2 using a .zip file that contains JSON structures that describe the parts of the resource. When you export a resource, Amazon Lex V2 creates the .zip file and makes it available to you using an Amazon S3 pre-signed URL. When you import a resource, you must create a .zip file that contains the JSON structures and upload it to an S3 pre-signed URL.

Amazon Lex creates the following directory structure in the .zip file when you export a bot. When you export a bot locale, only the structure under the locale is exported. When you export a custom vocabulary, only the structure under the custom vocabulary is exported.

```
BotName_BotVersion_ExportID_LexJson.zip
  -OR-
BotName_BotVersion_LocaleId_ExportId_LEX_JSON.zip
    --> manifest.json
    --> BotName
      ----> Bot.json
      ----> BotLocales
        -----> Locale_A
          -----> BotLocale.json
          -----> Intents
            -----> Intent_A
              -----> Intent.json
            -----> Slots
              -----> Slot_A
                -----> Slot.json
              -----> Slot_B
```

```
-----> Slot.json
-----> Intent_B
...
-----> SlotTypes
-----> SlotType_A
-----> SlotType.json
-----> SlotType_B
...
-----> CustomVocabulary
-----> CustomVocabulary.json
...
-----> Locale_B
...
```

## Manifest file structure

The manifest file contains metadata for the export file.

```
{
  "metadata": {
    "schemaVersion": "1.0",
    "fileFormat": "LexJson",
    "resourceType": "Bot | BotLocale | CustomVocabulary"
  }
}
```

## Bot file structure

The bot file contains the configuration information for the bot.

```
{
  "name": "BotName",
  "identifier": "identifier",
  "version": "number",
  "description": "description",
  "dataPrivacy": {
    "childDirected": true | false
  },
  "idleSessionTTLInSeconds": seconds
}
```

## Bot locale file structure

The bot locale file contains a description of the locale or language of a bot. When you export a bot, there can be more than one bot locale file in the .zip file. When you export a bot locale, there is only one locale in the zip file.

```
{
  "name": "locale name",
  "identifier": "locale ID",
  "version": "number",
  "description": "description",
  "voiceSettings": {
    "voiceId": "voice",
    "engine": "standard | neural"
  },
  "nluConfidenceThreshold": number
}
```

}

## Intent file structure

The intent file contains the configuration information for an intent. There is one intent file in the .zip file for each intent in a specific locale.

The following is an example of a JSON structure for the BookCar intent in the sample BookTrip bot. For a complete example of the JSON structure for an intent, see the [CreateIntent](#) operation.

```
{  
    "name": "BookCar",  
    "identifier": "891RWHHICO",  
    "description": "Intent to book a car.",  
    "parentIntentSignature": null,  
    "sampleUtterances": [  
        {  
            "utterance": "Book a car"  
        },  
        {  
            "utterance": "Reserve a car"  
        },  
        {  
            "utterance": "Make a car reservation"  
        }  
    ],  
    "intentConfirmationSetting": {  
        "confirmationPrompt": {  
            "messageGroupList": [  
                {  
                    "message": {  
                        "plainTextMessage": {  
                            "value": "OK, I have you down for a {CarType} hire in  
{PickUpCity} from {PickUpDate} to {ReturnDate}. Should I book the reservation?"  
                        },  
                        "ssmlMessage": null,  
                        "customPayload": null,  
                        "imageResponseCard": null  
                    },  
                    "variations": null  
                }  
            ],  
            "maxRetries": 2  
        },  
        "declinationResponse": {  
            "messageGroupList": [  
                {  
                    "message": {  
                        "plainTextMessage": {  
                            "value": "OK, I have cancelled your reservation in progress."  
                        },  
                        "ssmlMessage": null,  
                        "customPayload": null,  
                        "imageResponseCard": null  
                    },  
                    "variations": null  
                }  
            ]  
        },  
        "intentClosingSetting": null,  
        "inputContexts": null,  
        "outputContexts": null,  
    }  
}
```

```

    "kendraConfiguration": null,
    "dialogCodeHook": null,
    "fulfillmentCodeHook": null,
    "slotPriorities": [
        {
            "slotName": "DriverAge",
            "priority": 4
        },
        {
            "slotName": "PickUpDate",
            "priority": 2
        },
        {
            "slotName": "ReturnDate",
            "priority": 3
        },
        {
            "slotName": "PickUpCity",
            "priority": 1
        },
        {
            "slotName": "CarType",
            "priority": 5
        }
    ]
}

```

## Slot file structure

The slot file contains the configuration information for a slot in an intent. There is one slot file in the .zip file for each slot defined for an intent in a specific locale.

The following example is the JSON structure of a slot that enables the customer to choose the type of car they wish to rent in the BookCar intent in the BookTrip example bot. For a complete example of the JSON structure for an slot, see the [CreateSlot](#) operation.

```

{
    "name": "CarType",
    "identifier": "KDHJWNGZGC",
    "description": "Type of car being reserved.",
    "multipleValuesSetting": {
        "allowMutlipleValues": false
    },
    "slotTypeName": "CarTypeValues",
    "obfuscationSetting": null,
    "slotConstraint": "Required",
    "defaultValueSpec": null,
    "slotValueElicitationSetting": {
        "promptSpecification": {
            "messageGroupList": [
                {
                    "message": {
                        "plainTextMessage": {
                            "value": "What type of car would you like to rent? Our most
popular options are economy, midsize, and luxury"
                        },
                        "ssmlMessage": null,
                        "customPayload": null,
                        "imageResponseCard": null
                    },
                    "variations": null
                }
            ],
            "variations": null
        }
    }
}

```

```

        "maxRetries": 2
    },
    "sampleValueElicitingUtterances": null,
    "waitAndContinueSpecification": null,
}
}

```

The following example shows the JSON structure of a composite slot.

```
{
    "name": "CarType",
    "identifier": "KDHJWNGZGC",
    "description": "Type of car being reserved.",
    "multipleValuesSetting": {
        "allowMutlipleValues": false
    },
    "slotTypeName": "CarTypeValues",
    "obfuscationSetting": null,
    "slotConstraint": "Required",
    "defaultValueSpec": null,
    "slotValueElicitationSetting": {
        "promptSpecification": {
            "messageGroupList": [
                {
                    "message": {
                        "plainTextMessage": {
                            "value": "What type of car would you like to rent? Our most
popular options are economy, midsize, and luxury"
                        },
                        "ssmlMessage": null,
                        "customPayload": null,
                        "imageResponseCard": null
                    },
                    "variations": null
                }
            ],
            "maxRetries": 2
        },
        "sampleValueElicitingUtterances": null,
        "waitAndContinueSpecification": null,
    },
    "subSlotSetting": {
        "slotSpecifications": {
            "firstname": {
                "valueElicitationSetting": {
                    "promptSpecification": {
                        "allowInterrupt": false,
                        "messageGroupsList": [
                            {
                                "message": {
                                    "imageResponseCard": null,
                                    "ssmlMessage": null,
                                    "customPayload": null,
                                    "plainTextMessage": {
                                        "value": "please provide firstname"
                                    }
                                },
                                "variations": null
                            }
            ],
            "maxRetries": 2,
            "messageSelectionStrategy": "Random"
        },
        "defaultValueSpecification": null,
        "sampleUtterances": [

```

```
{
    "utterance": "my name is {firstName}"
}
],
"waitAndContinueSpecification": null
},
"slotTypeId": "AMAZON.FirstName"
},
"eyeColor": {
    "valueElicitationSetting": {
        "promptSpecification": {
            "allowInterrupt": false,
            "messageGroupsList": [
                {
                    "message": {
                        "imageResponseCard": null,
                        "ssmlMessage": null,
                        "customPayload": null,
                        "plainTextMessage": {
                            "value": "please provide eye color"
                        }
                    },
                    "variations": null
                }
            ],
            "maxRetries": 2,
            "messageSelectionStrategy": "Random"
        },
        "defaultValueSpecification": null,
        "sampleUtterances": [
            {
                "utterance": "eye color is {eyeColor}"
            },
            {
                "utterance": "I have eyeColor eyes"
            }
        ],
        "waitAndContinueSpecification": null
    },
    "slotTypeId": "7FEVCB2PQE"
},
"expression": "(firstname OR eyeColor)"
}
}
```

## Slot type file structure

The slot type file contains the configuration information for a custom slot type used in a language or locale. There is one slot type file in the .zip file for each custom slot type in a specific locale.

The following is the JSON structure for the slot type that lists the types of cars available in the BookTrip example bot. For a complete example of the JSON structure for a slot type, see the [CreateSlotType](#) operation.

```
{
    "name": "CarTypeValues",
    "identifier": "T1YUHGD9ZR",
    "description": "Enumeration representing possible types of cars available for hire",
    "slotTypeValues": [
        {
            "synonyms": null,
            "sampleValue": {
                "value": "economy"
            }
        }
    ]
}
```

```

        },
        {
            "synonyms": null,
            "sampleValue": {
                "value": "standard"
            }
        },
        {
            "synonyms": null,
            "sampleValue": {
                "value": "midsize"
            }
        },
        {
            "synonyms": null,
            "sampleValue": {
                "value": "full size"
            }
        },
        {
            "synonyms": null,
            "sampleValue": {
                "value": "luxury"
            }
        },
        {
            "synonyms": null,
            "sampleValue": {
                "value": "minivan"
            }
        }
    ],
    "parentSlotTypeSignature": null,
    "valueSelectionSetting": {
        "resolutionStrategy": "TOP_RESOLUTION",
        "advancedRecognitionSetting": {
            "audioRecognitionStrategy": "UseSlotValuesAsCustomVocabulary"
        },
        "regexFilter": null
    }
}

```

The following example shows the JSON structure for a composite slot type.

```

{
    "name": "CarCompositeType",
    "identifier": "TPA3CC9V",
    "description": null,
    "slotTypeValues": null,
    "parentSlotTypeSignature": null,
    "valueSelectionSetting": {
        "regexFilter": null,
        "resolutionStrategy": "CONCATENATION"
    },
    "compositeSlotTypeSetting": {
        "subSlots": [
            {
                "name": "model",
                "slotTypeId": "MODELTYPID" # custom slot type Id for model
            },
            {
                "name": "city",
                "slotTypeId": "AMAZON.City"
            },
            {
                "name": "country",
                "slotTypeId": "AMAZON.Country"
            },
            {
                "name": "make",
            }
        ]
    }
}

```

```

        "slotTypeId": "MAKETYPEID" # custom slot type Id for make
    }
}
}
```

The following is a slot type that uses a custom grammar to understand the customer's utterances. For more information, see [Using a custom grammar slot type \(p. 83\)](#).

```
{
  "name": "custom_grammar",
  "identifier": "7KEAQI0KPX",
  "description": "Slot type using a custom grammar",
  "slotTypeValues": null,
  "parentSlotTypeSignature": null,
  "valueSelectionSetting": null,
  "externalSourceSetting": {
    "grammarSlotTypeSetting": {
      "source": {
        "kmsKeyArn": "arn:aws:kms:Region:123456789012:alias/customer-grxml-key",
        "s3BucketName": "grxml-test",
        "s3ObjectKey": "grxml_files/grammar.grxml"
      }
    }
  }
}
```

## Custom vocabulary file structure.

The custom vocabulary file contains the entries in a custom vocabulary for single language or locale. There is one custom vocabulary file in the .zip file for each locale that has a custom vocabulary.

The following is a custom vocabulary file for a bot that takes restaurant orders. There is one file per locale in the bot.

```
{
  "customVocabularyItems": [
    {
      "weight": 3,
      "phrase": "wafers"
    },
    {
      "weight": null,
      "phrase": "extra large"
    },
    {
      "weight": null,
      "phrase": "cremini mushroom soup"
    },
    {
      "weight": null,
      "phrase": "ramen"
    },
    {
      "weight": null,
      "phrase": "orzo"
    }
  ]
}
```

# Tagging resources

To help you manage your Amazon Lex V2 bots and bot aliases, you can assign metadata to each resource as *tags*. A tag is a label that you assign to an AWS resource. Each tag consists of a key and a value.

Tags enable you to categorize your AWS resources in different ways, for example, by purpose, owner, or application. Tags help you to:

- Identify and organize your AWS resources. Many AWS resources support tagging, so you can assign the same tag to resources in different services to indicate that the resources are the same. For example, you can tag a bot and the Lambda functions that it uses with the same tag.
- Allocate costs. You activate tags on the AWS Billing and Cost Management dashboard. AWS uses the tags to categorize your costs and deliver a monthly cost allocation report to you. For Amazon Lex V2, you can allocate costs for each alias using tags specific to the alias. For more information, see [Use cost allocation tags](#) in the *AWS Billing and Cost Management User Guide*.
- Control access to your resources. You can use tags with Amazon Lex V2 to create policies to control access to Amazon Lex V2 resources. These policies can be attached to an IAM role or user to enable tag-based access control.

You can work with tags using the AWS Management Console, the AWS Command Line Interface, or the Amazon Lex V2 API.

## Tagging your resources

If you are using the Amazon Lex V2 console, you can tag resources when you create them, or you can add the tags later. You can also use the console to update or remove existing tags.

If you are using the AWS CLI or Amazon Lex V2 API, you use the following operations to manage tags for your resource:

- [CreateBot](#) and [CreateBotAlias](#) – apply tags when you create a bot or a bot alias.
- [ListTagsForResource](#) – view the tags associated with a resource.
- [TagResource](#) – add and modify tags on an existing resource.
- [UntagResource](#) – remove tags from a resource.

The following resources in Amazon Lex V2 support tagging:

- Bots – use an Amazon Resource Name (ARN) like the following:
  - `arn:aws:lex:${Region}:${account}:bot/${bot-id}`
- Bot aliases – use an ARN like the following:
  - `arn:aws:lex:${Region}:${account}:bot-alias/${bot-id}/${bot-alias-id}`

The bot-id and bot-alias-id values are capitalized alphanumeric strings 10 characters long.

## Tag restrictions

The following basic restrictions apply to tags on Amazon Lex V2 resources:

- Maximum number of keys – 50 using the console, 200 using the API
- Maximum key length – 128 characters
- Maximum value length – 256 characters
- Valid characters for key and value – a-z, A-Z, 0-9, space, and the following characters: \_.:/=+- and @
- Keys and values are case-sensitive
- Don't use aws : as a prefix for keys, it's reserved for AWS use

## Tagging resources (console)

You can use the console to manage tags on a bot or bot alias. You can add tags when you create the resource, or you can add, modify, or remove tags from existing resources.

### To add a tag when you create a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose **Create bot**.
3. In the **Advanced settings** section of **Configure bot settings**, choose **Add new tag**. You can add tags to the bot and to the TestBotAlias alias.
4. Choose **Next** to continue creating your bot.

### To add a tag when you create a bot alias

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose the bot that you want to add the bot alias to.
3. From the left menu, choose **Aliases** and then choose **Create alias**.
4. In **General info**, choose **Add new tag** from **Tags**.
5. Choose **Create**.

### To add, remove, or modify a tag on an existing bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose the bot that you want to modify.
3. From the left menu, choose **Settings**, and then choose **Edit**.
4. In **Tags**, make your changes.
5. Choose **Save** to save your changes to the bot.

### To add, remove, or modify a tag on an existing alias

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose the bot that you want to modify.
3. From the left menu, choose **Aliases** and then from the list of aliases, choose the alias to modify.
4. From **Alias details**, in **Tags**, choose **Modify tags**.
5. In **Manage tags**, make your changes.
6. Choose **Save** to save your changes to the alias.

# Security in Amazon Lex V2

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security of the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS Compliance Programs](#). To learn about the compliance programs that apply to Amazon Lex V2, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon Lex V2. The following topics show you how to configure Amazon Lex V2 to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Amazon Lex V2 resources.

## Topics

- [Data protection in Amazon Lex V2 \(p. 311\)](#)
- [Identity and access management for Amazon Lex V2 \(p. 313\)](#)
- [Using service-linked roles for Amazon Lex V2 \(p. 348\)](#)
- [Logging and monitoring in Amazon Lex V2 \(p. 351\)](#)
- [Compliance validation for Amazon Lex V2 \(p. 351\)](#)
- [Resilience in Amazon Lex V2 \(p. 352\)](#)
- [Infrastructure security in Amazon Lex V2 \(p. 352\)](#)
- [Amazon Lex V2 and interface VPC endpoints \(AWS PrivateLink\) \(p. 352\)](#)

## Data protection in Amazon Lex V2

Amazon Lex V2 conforms to the AWS [shared responsibility model](#), which includes regulations and guidelines for data protection. AWS is responsible for protecting the global infrastructure that runs all the AWS services. AWS maintains control over data hosted on this infrastructure, including the security configuration controls for handling customer content and personal data. AWS customers and APN partners, acting either as data controllers or data processors, are responsible for any personal data that they put in the AWS Cloud.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM), so that each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources.

- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This includes when you work with Amazon Lex V2 or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Amazon Lex V2 or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

For more information about data protection, see the [AWS Shared Responsibility Model and GDPR blog post](#) on the [AWS Security Blog](#).

## Encryption at rest

Amazon Lex V2 encrypts user utterances and other information that it stores.

### Topics

- [Sample utterances \(p. 312\)](#)
- [Session attributes \(p. 312\)](#)
- [Request attributes \(p. 312\)](#)

## Sample utterances

When you develop a bot, you can provide sample utterances for each intent and slot. You can also provide custom values and synonyms for slots. This information is encrypted at rest, and it is used only to build the bot and create the customer experience.

## Session attributes

Session attributes contain application-specific information that is passed between Amazon Lex V2 and client applications. Amazon Lex V2 passes session attributes to all AWS Lambda functions configured for a bot. If a Lambda function adds or updates session attributes, Amazon Lex V2 passes the new information back to the client application.

Session attributes persist in an encrypted store for the duration of the session. You can configure the session to remain active for a minimum of 1 minute and up to 24 hours after the last user utterance. The default session duration is 5 minutes.

## Request attributes

Request attributes contain request-specific information and apply only to the current request. A client application uses request attributes to send information to Amazon Lex V2 at runtime.

You use request attributes to pass information that doesn't need to persist for the entire session. Because request attributes don't persist across requests, they aren't stored.

## Encryption in transit

Amazon Lex V2 uses the HTTPS protocol to communicate with your client application. It uses HTTPS and AWS signatures to communicate with other services, such as Amazon Polly and AWS Lambda on your application's behalf.

# Identity and access management for Amazon Lex V2

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon Lex V2 resources. IAM is an AWS service that you can use with no additional charge.

## Topics

- [Audience \(p. 313\)](#)
- [Authenticating with identities \(p. 313\)](#)
- [Managing access using policies \(p. 316\)](#)
- [How Amazon Lex V2 works with IAM \(p. 317\)](#)
- [Identity-based policy examples for Amazon Lex V2 \(p. 324\)](#)
- [Resource-based policy examples for Amazon Lex V2 \(p. 333\)](#)
- [AWS managed policies for Amazon Lex V2 \(p. 340\)](#)
- [Troubleshooting Amazon Lex V2 identity and access \(p. 347\)](#)

## Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Amazon Lex V2.

**Service user** – If you use the Amazon Lex V2 service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Amazon Lex V2 features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Amazon Lex V2, see [Troubleshooting Amazon Lex V2 identity and access \(p. 347\)](#).

**Service administrator** – If you're in charge of Amazon Lex V2 resources at your company, you probably have full access to Amazon Lex V2. It's your job to determine which Amazon Lex V2 features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Amazon Lex V2, see [How Amazon Lex V2 works with IAM \(p. 317\)](#).

**IAM administrator** – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Amazon Lex V2. To view example Amazon Lex V2 identity-based policies that you can use in IAM, see [Identity-based policy examples for Amazon Lex V2 \(p. 324\)](#).

## Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (successor to AWS Single Sign-On) (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the [AWS Sign-In User Guide](#).

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signature Version 4 signing process](#) in the [AWS General Reference](#).

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the [AWS IAM Identity Center \(successor to AWS Single Sign-On\) User Guide](#) and [Using multi-factor authentication \(MFA\) in AWS](#) in the [IAM User Guide](#).

## AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the *AWS account root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the [AWS General Reference](#).

## Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center (successor to AWS Single Sign-On). You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see [What is IAM Identity Center?](#) in the [AWS IAM Identity Center \(successor to AWS Single Sign-On\) User Guide](#).

## IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the [IAM User Guide](#).

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but

roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

## IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Creating a role for a third-party Identity Provider](#) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see [Permission sets](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, resources, and condition keys for Amazon Lex V2](#) in the *Service Authorization Reference*.
  - **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

## Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies in the IAM User Guide](#).

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

Every IAM entity (user or role) starts with no permissions. By default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

### Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies in the IAM User Guide](#).

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

### Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

### Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

## Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

## Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

## How Amazon Lex V2 works with IAM

Before you use IAM to manage access to Amazon Lex V2, learn what IAM features are available to use with Amazon Lex V2.

### IAM features you can use with Amazon Lex V2

IAM feature	Amazon Lex V2 support
<a href="#">Identity-based policies (p. 318)</a>	Yes
<a href="#">Resource-based policies (p. 318)</a>	Yes
<a href="#">Policy actions (p. 321)</a>	Yes
<a href="#">Policy resources (p. 322)</a>	Yes
<a href="#">Policy condition keys (p. 322)</a>	No
<a href="#">ACLs (p. 323)</a>	No
<a href="#">ABAC (tags in policies) (p. 323)</a>	Yes
<a href="#">Temporary credentials (p. 323)</a>	No

IAM feature	Amazon Lex V2 support
Principal permissions ( <a href="#">p. 324</a> )	Yes
Service roles ( <a href="#">p. 324</a> )	Yes
Service-linked roles ( <a href="#">p. 324</a> )	Partial

To get a high-level view of how Amazon Lex V2 and other AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

## Identity-based policies for Amazon Lex V2

Supports identity-based policies	Yes
----------------------------------	-----

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see [IAM JSON policy elements reference](#) in the *IAM User Guide*.

### Identity-based policy examples for Amazon Lex V2

To view examples of Amazon Lex V2 identity-based policies, see [Identity-based policy examples for Amazon Lex V2 \(p. 324\)](#).

## Resource-based policies within Amazon Lex V2

Supports resource-based policies	Yes
----------------------------------	-----

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM role trust policies and Amazon S3 bucket policies. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include users, roles, federated users, or AWS services.

You can't use cross-account or cross-region policies with Amazon Lex. If you create a policy for a resource with a cross-account or cross-region ARN, Amazon Lex returns an error.

The Amazon Lex service supports resource-based policies called a *bot policy* and a *bot alias policy*, which are attached to a bot or a bot alias. These policies define which principals can perform actions on the bot or bot alias.

Actions can only be used on specific resources. For example, the `UpdateBot` action can only be used on bot resources, the `UpdateBotAlias` action can only be used on bot alias resources. If you specify an

action in a policy that can't be used on the resource specified in the policy, Amazon Lex returns an error. For the list of actions and the resources that they can be used with, see the following table.

Action	Supports resource-based policy	Resource
BuildBotLocale	Supported	BotId
CreateBot	No	
CreateBotAlias	No	
CreateBotChannel [permission only]	Supported	BotId
CreateBotLocale	Supported	BotId
CreateBotVersion	Supported	BotId
CreateExport	Supported	BotId
CreateIntent	Supported	BotId
CreateResourcePolicy	Supported	BotId, BotAliasId
CreateSlot	Supported	BotId
CreateSlotType	Supported	BotId
CreateUploadUrl	No	
DeleteBot	Supported	BotId, BotAliasId
DeleteBotAlias	Supported	BotAliasId
DeleteBotChannel [permission only]	Supported	BotId
DeleteBotLocale	Supported	BotId
DeleteBotVersion	Supported	BotId
DeleteExport	Supported	BotId
DeleteImport	Supported	BotId
DeleteIntent	Supported	BotId
DeleteResourcePolicy	Supported	BotId, BotAliasId
DeleteSession	Supported	BotAliasId
DeleteSlot	Supported	BotId
DeleteSlotType	Supported	BotId
DescribeBot	Supported	BotId
DescribeBotAlias	Supported	BotAliasId
DescribeBotChannel [permission only]	Supported	BotId

Action	Supports resource-based policy	Resource
DescribeBotLocale	Supported	BotId
DescribeBotVersion	Supported	BotId
DescribeExport	Supported	BotId
DescribeImport	Supported	BotId
DescribeIntent	Supported	BotId
DescribeResourcePolicy	Supported	BotId, BotAliasId
DescribeSlot	Supported	BotId
DescribeSlotType	Supported	BotId
GetSession	Supported	BotAliasId
ListBotAliases	Supported	BotAliasId
ListBotChannels [permission only]	Supported	BotId
ListBotLocales	Supported	BotId
ListBots	No	
ListBotVersions	Supported	BotId
ListBuiltInIntents	No	
ListBuiltInSlotTypes	No	
ListExports	No	
ListImports	No	
ListIntents	Supported	BotId
ListSlots	Supported	BotId
ListSlotTypes	Supported	BotId
PutSession	Supported	BotAliasId
RecognizeText	Supported	BotAliasId
RecognizeUtterance	Supported	BotAliasId
StartConversation	Supported	BotAliasId
StartImport	Supported	BotId, BotAliasId
TagResource	No	
UpdateBot	Supported	BotId
UpdateBotAlias	Supported	BotAliasId
UpdateBotLocale	Supported	BotId

Action	Supports resource-based policy	Resource
UpdateBotVersion	Supported	BotId
UpdateExport	Supported	BotId
UpdateIntent	Supported	BotId
UpdateResourcePolicy	Supported	BotId, BotAliasId
UpdateSlot	Supported	BotId
UpdateSlotType	Supported	BotId
UntagResource	No	

To learn how to attach a resource-based policy to a bot or bot alias, see [Resource-based policy examples for Amazon Lex V2 \(p. 333\)](#).

## Resource-based policy examples within Amazon Lex V2

To view examples of Amazon Lex V2 resource-based policies, see [Resource-based policy examples for Amazon Lex V2 \(p. 333\)](#).

## Policy actions for Amazon Lex V2

Supports policy actions	Yes
-------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Action element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Amazon Lex V2 actions, see [Actions defined by Amazon Lex V2 in the Service Authorization Reference](#).

Policy actions in Amazon Lex V2 use the following prefix before the action:

lex

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
    "lex:action1",
    "lex:action2"
]
```

To view examples of Amazon Lex V2 identity-based policies, see [Identity-based policy examples for Amazon Lex V2 \(p. 324\)](#).

## Policy resources for Amazon Lex V2

Supports policy resources	Yes
---------------------------	-----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (\*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Amazon Lex V2 resource types and their ARNs, see [Resources defined by Amazon Lex V2](#) in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Actions defined by Amazon Lex V2](#).

To view examples of Amazon Lex V2 identity-based policies, see [Identity-based policy examples for Amazon Lex V2 \(p. 324\)](#).

## Policy condition keys for Amazon Lex V2

Supports service-specific policy condition keys	No
---	----

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Condition element (or Condition *block*) lets you specify conditions in which a statement is in effect. The Condition element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple Condition elements in a statement, or multiple keys in a single Condition element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

To see a list of Amazon Lex V2 condition keys, see [Condition keys for Amazon Lex V2](#) in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Actions defined by Amazon Lex V2](#).

To view examples of Amazon Lex V2 identity-based policies, see [Identity-based policy examples for Amazon Lex V2 \(p. 324\)](#).

## Access control lists (ACLs) in Amazon Lex V2

Supports ACLs	No
---------------	----

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

## Attribute-based access control (ABAC) with Amazon Lex V2

Supports ABAC (tags in policies)	Yes
----------------------------------	-----

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the [condition element](#) of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see [What is ABAC?](#) in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see [Use attribute-based access control \(ABAC\)](#) in the *IAM User Guide*.

## Using Temporary credentials with Amazon Lex V2

Supports temporary credentials	No
--------------------------------	----

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see [AWS services that work with IAM](#) in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see [Switching to a role \(console\)](#) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary

credentials instead of using long-term access keys. For more information, see [Temporary security credentials in IAM](#).

## Cross-service principal permissions for Amazon Lex V2

Supports principal permissions	Yes
--------------------------------	-----

When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, resources, and condition keys for Amazon Lex V2](#) in the *Service Authorization Reference*.

## Service roles for Amazon Lex V2

Supports service roles	Yes
------------------------	-----

A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

**Warning**

Changing the permissions for a service role might break Amazon Lex V2 functionality. Edit service roles only when Amazon Lex V2 provides guidance to do so.

## Service-linked roles for Amazon Lex V2

Supports service-linked roles	Partial
-------------------------------	---------

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see [AWS services that work with IAM](#). Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the [Yes](#) link to view the service-linked role documentation for that service.

## Identity-based policy examples for Amazon Lex V2

By default, users and roles don't have permission to create or modify Amazon Lex V2 resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform actions on the resources that they need. The administrator must then attach those policies for users that require them.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see [Creating IAM policies](#) in the *IAM User Guide*.

For details about actions and resource types defined by Amazon Lex V2, including the format of the ARNs for each of the resource types, see [Actions, resources, and condition keys for Amazon Lex V2](#) in the *Service Authorization Reference*.

## Topics

- [Policy best practices \(p. 325\)](#)
- [Using the Amazon Lex V2 console \(p. 326\)](#)
- [Allow users to add functions to a bot \(p. 326\)](#)
- [Allow users to add channels to a bot \(p. 326\)](#)
- [Allow users to create and update bots \(p. 327\)](#)
- [Allow users to use the Automated Chatbot Designer \(p. 327\)](#)
- [Allow users to use a AWS KMS key to encrypt and decrypt files \(p. 328\)](#)
- [Allow users to delete bots \(p. 328\)](#)
- [Allow users to have a conversation with a bot \(p. 328\)](#)
- [Allow a specific user to manage resource-based policies \(p. 329\)](#)
- [Allow a user to export bots and bot locales \(p. 329\)](#)
- [Allow a user to export a custom vocabulary \(p. 330\)](#)
- [Allow a user to import bots and bot locales \(p. 330\)](#)
- [Allow a user to import a custom vocabulary \(p. 331\)](#)
- [Allow a user to migrate a bot from Amazon Lex to Amazon Lex V2 \(p. 331\)](#)
- [Allow users to view their own permissions \(p. 332\)](#)
- [Allow a user to draw conversation flow with visual conversation builder in Amazon Lex V2 \(p. 333\)](#)

## Policy best practices

Identity-based policies determine whether someone can create, access, or delete Amazon Lex V2 resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see [AWS managed policies](#) or [AWS managed policies for job functions](#) in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see [Policies and permissions in IAM](#) in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions**
  - IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see [IAM Access Analyzer policy validation](#) in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or root users in your account, turn on MFA for additional security. To require MFA when API operations are

called, add MFA conditions to your policies. For more information, see [Configuring MFA-protected API access](#) in the *IAM User Guide*.

For more information about best practices in IAM, see [Security best practices in IAM](#) in the *IAM User Guide*.

## Using the Amazon Lex V2 console

To access the Amazon Lex V2 console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Amazon Lex V2 resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (IAM users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

To ensure that users and roles can still use the Amazon Lex V2 console, users need to have Console access. For more information about creating a user with Console access, see [Creating an IAM user in your AWS account](#) in the *IAM User Guide*.

## Allow users to add functions to a bot

This example shows a policy that allows IAM users to add Amazon Comprehend, sentiment analysis and Amazon Kendra query permissions to an Amazon Lex V2 bot.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "Id1",  
            "Effect": "Allow",  
            "Action": "iam:PutRolePolicy",  
            "Resource": "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/  
AWSServiceRoleForLexV2Bots*",  
        },  
        {  
            "Sid": "Id2",  
            "Effect": "Allow",  
            "Action": "iam:GetRolePolicy",  
            "Resource": "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/  
AWSServiceRoleForLexV2Bots*",  
        }  
    ]  
}
```

## Allow users to add channels to a bot

This example is a policy that allows IAM users to add a messaging channel to a bot. A user must have this policy in place before they can deploy a bot on a messaging platform.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "Id1",  
            "Effect": "Allow",  
            "Action": "iam:PutRolePolicy",  
        }  
    ]  
}
```

```

        "Resource": "arn:aws:iam::*:role/aws-service-role/channels.lexv2.amazonaws.com/
AWSServiceRoleForLexV2Channels"
    },
    {
        "Sid": "Id2",
        "Effect": "Allow",
        "Action": "iam:GetRolePolicy",
        "Resource": "arn:aws:iam::*:role/aws-service-role/channels.lexv2.amazonaws.com/
AWSServiceRoleForLexV2Channels"
    }
]
}

```

## Allow users to create and update bots

This example shows an example policy that allows IAM users to create and update any bot. The policy includes permissions to complete this action on the console or using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex>CreateBot",
                "lex:UpdateBot",
                "iam:PassRole"
            ],
            "Effect": "Allow",
            "Resource": ["arn:aws:lex:Region:123412341234:bot/*"]
        }
    ]
}
```

## Allow users to use the Automated Chatbot Designer

This example shows an example policy that allows IAM users to run the Automated Chatbot Designer.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3.GetObject",
                "s3>ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::<customer-bucket>/<bucketName>",
                # Resource should point to the bucket or an explicit folder.
                # Provide this to read the entire bucket
                "arn:aws:s3:::<customer-bucket>/<bucketName>/*",
                # Provide this to read a specific folder
                "arn:aws:s3:::<customer-bucket>/<bucketName>/<pathFormat>/*"
            ]
        },
        {
            # Use this if your S3 bucket is encrypted with a KMS key.
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": [

```

```
        "arn:aws:kms:<Region>:<customerAccountId>:key/<kmsKeyId>"  
    ]  
}
```

## Allow users to use a AWS KMS key to encrypt and decrypt files

This example shows an example policy that allows IAM users to use a AWS KMS customer managed key to encrypt and decrypt data.

```
{  
    "Version": "2012-10-17",  
    "Id": "sample-policy",  
    "Statement": [  
        {  
            "Sid": "Allow Lex access",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "lexv2.amazonaws.com"  
            },  
            "Action": [  
                # If the key is for encryption  
                "kms:Encrypt",  
                "kms:GenerateDataKey"  
                # If the key is for decryption  
                "kms:Decrypt"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

## Allow users to delete bots

This example shows an example policy that allows IAM users to delete any bot. The policy includes permissions to complete this action on the console or using the AWS CLI or AWS API.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "lex>DeleteBot",  
                "lex>DeleteBotLocale",  
                "lex>DeleteBotAlias",  
                "lex>DeleteIntent",  
                "lex>DeleteSlot",  
                "lex>DeleteSlottype"  
            ],  
            "Effect": "Allow",  
            "Resource": ["arn:aws:lex:<Region>:123412341234:bot/*",  
                        "arn:aws:lex:<Region>:123412341234:bot-alias/*"]  
        }  
    ]  
}
```

## Allow users to have a conversation with a bot

This example shows an example policy that allows IAM users have a conversation with any bot. The policy includes permissions to complete this action on the console or using the AWS CLI or AWS API.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "lex:StartConversation",  
                "lex:RecognizeText",  
                "lex:RecognizeUtterance",  
                "lex:GetSession",  
                "lex:PutSession",  
                "lex:DeleteSession"  
            ],  
            "Effect": "Allow",  
            "Resource": "arn:aws:lex:Region:123412341234:bot-alias/*"  
        }  
    ]  
}
```

Allow a specific user to manage resource-based policies

The following example grants permission for a specific user to manage the resource-based policies. It allows console and API access to the policies associated with bots and bot aliases.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ResourcePolicyEditor",  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "arn:aws:iam::123456789012:role/ResourcePolicyEditor"  
            },  
            "Action": [  
                "lex>CreateResourcePolicy",  
                "lex:UpdateResourcePolicy",  
                "lex>DeleteResourcePolicy",  
                "lex:DescribeResourcePolicy"  
            ]  
        }  
    ]  
}
```

## Allow a user to export bots and bot locales

The following IAM permission policy enables a user to create, update, and get an export for a bot or bot locale.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "lex>CreateExport",  
                "lex>UpdateExport",  
                "lex>DescribeExport",  
                "lex>DescribeBot",  
                "lex>DescribeBotLocale",  
                "lex>ListBotLocales",  
                "lex>DescribeIntent",  
                "lex>ListIntents".  
            ]  
        }  
    ]  
}
```

```
        "lex:DescribeSlotType",
        "lex>ListSlotTypes",
        "lex:DescribeSlot",
        "lex>ListSlots",
        "lex:DescribeCustomVocabulary"
    ],
    "Effect": "Allow",
    "Resource": ["arn:aws:lex:Region:123456789012:bot/*"]
}
]
}
```

## Allow a user to export a custom vocabulary

The following IAM permission policy allows a user to export a custom vocabulary from a bot locale.

```
{"Version": "2012-10-17",
"Statement": [
    {"Action": [
        "lex>CreateExport",
        "lex:UpdateExport",
        "lex:DescribeExport",
        "lex:DescribeCustomVocabulary"
    ],
    "Effect": "Allow",
    "Resource": ["arn:aws:lex:Region:123456789012:bot/*"]
}
]
```

## Allow a user to import bots and bot locales

The following IAM permission policy allows a user to import a bot or bot locale and to check the status of an import.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex>CreateUploadUrl",
                "lex:StartImport",
                "lex:DescribeImport",
                "lex>CreateBot",
                "lex:UpdateBot",
                "lex>DeleteBot",
                "lex>CreateBotLocale",
                "lex:UpdateBotLocale",
                "lex>DeleteBotLocale",
                "lex>CreateIntent",
                "lex:UpdateIntent",
                "lex>DeleteIntent",
                "lex>CreateSlotType",
                "lex:UpdateSlotType",
                "lex>DeleteSlotType",
                "lex>CreateSlot",
                "lex:UpdateSlot",
                "lex>DeleteSlot",
                "lex>CreateCustomVocabulary",
                "lex:UpdateCustomVocabulary",
                "lex>DeleteCustomVocabulary",
                "lex:DescribeBot",
                "lex:DescribeBotLocale",
                "lex:DescribeIntent",
                "lex:DescribeSlot",
                "lex:DescribeSlotType"
            ],
            "Effect": "Allow",
            "Resource": ["arn:aws:lex:Region:123456789012:bot/*"]
        }
    ]
}
```

```
        "iam:PassRole",
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:lex:Region:123456789012:bot/*",
        "arn:aws:lex:Region:123456789012:bot-alias/*"
    ]
}
]
```

## Allow a user to import a custom vocabulary

The following IAM permission policy allows a user to import a custom vocabulary to a bot locale.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "lex>CreateUploadUrl",
                "lex>StartImport",
                "lex>DescribeImport",
                "lex>CreateCustomVocabulary",
                "lex>UpdateCustomVocabulary",
                "lex>DeleteCustomVocabulary"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:lex:Region:123456789012:bot/*"
            ]
        }
    ]
}
```

## Allow a user to migrate a bot from Amazon Lex to Amazon Lex V2

The following IAM permission policy allows a user to start migrating a bot from Amazon Lex to Amazon Lex V2.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "startMigration",
            "Effect": "Allow",
            "Action": "lex>StartMigration",
            "Resource": "arn:aws:lex:>Region<:>123456789012<:bot:/*"
        },
        {
            "Sid": "passRole",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam:>123456789012<:role/>v2 bot role<"
        },
        {
            "Sid": "allowOperations",
            "Effect": "Allow",
            "Action": "lex>StartMigration"
        }
    ]
}
```

```

    "Action": [
        "lex>CreateBot",
        "lex>CreateIntent",
        "lex>UpdateSlot",
        "lex>DescribeBotLocale",
        "lex>UpdateBotAlias",
        "lex>CreateSlotType",
        "lex>DeleteBotLocale",
        "lex>DescribeBot",
        "lex>UpdateBotLocale",
        "lex>CreateSlot",
        "lex>DeleteSlot",
        "lex>UpdateBot",
        "lex>DeleteSlotType",
        "lex>DescribeBotAlias",
        "lex>CreateBotLocale",
        "lex>DeleteIntent",
        "lex>StartImport",
        "lex>UpdateSlotType",
        "lex>UpdateIntent",
        "lex>DescribeImport",
        "lex>CreateCustomVocabulary",
        "lex>UpdateCustomVocabulary",
        "lex>DeleteCustomVocabulary",
        "lex>DescribeCustomVocabulary",
        "lex>DescribeCustomVocabularyMetadata"
    ],
    "Resource": [
        "arn:aws:lex:>Region<:>123456789012<:bot/*",
        "arn:aws:lex:>Region<:>123456789012<:bot-alias/*/*"
    ]
},
{
    "Sid": "showBots",
    "Effect": "Allow",
    "Action": [
        "lex>CreateUploadUrl",
        "lex>ListBots"
    ],
    "Resource": "*"
}
]
}

```

## Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam>ListGroupsForUser",
                "iam>ListAttachedUserPolicies",
                "iam>ListUserPolicies",
                "iam GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        }
    ]
}
```

```
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam>ListAttachedGroupPolicies",
                "iam>ListGroupPolicies",
                "iam>ListPolicyVersions",
                "iam>ListPolicies",
                "iam>ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

## Allow a user to draw conversation flow with visual conversation builder in Amazon Lex V2

The following IAM permission policy allows a user to draw the conversation flow with visual conversation builder in Amazon Lex V2.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {"Action": [
            "lex:UpdateIntent",
            "lex:DescribeIntent"
        ],
        "Effect": "Allow",
        "Resource": ["arn:aws:lex:Region:123456789012:bot/*"]
    }
]
```

## Resource-based policy examples for Amazon Lex V2

A *resource-based policy* is attached to a resource, such as a bot or a bot alias. With a resource-based policy you can specify who has access to the resource and the actions that they can perform on it. For example, you can add resource-based policies that enable a user to modify a specific bot, or to allow a user to use runtime operations on a specific bot alias.

When you use a resource-based policy you can allow other AWS services to access resources in your account. For example, you can allow Amazon Connect to access an Amazon Lex bot.

To learn how to create a bot or bot alias, see [Building bots \(p. 17\)](#).

### Topics

- [Use the console to specify a resource-based policy \(p. 334\)](#)
- [Use the API to specify a resource-based policy \(p. 336\)](#)
- [Allow an IAM role to update a bot and list bot aliases \(p. 338\)](#)

- [Allow a user to have a conversation with a bot \(p. 338\)](#)
- [Allow an AWS service to use a specific Amazon Lex V2 bot \(p. 339\)](#)

## Use the console to specify a resource-based policy

You can use the Amazon Lex console to manage the resource-based policies for your bots and bot aliases. You enter the JSON structure of a policy and the console associates it with the resource. If there is a policy already associated with a resource, you can use the console to view and modify the policy.

When you save a policy with the policy editor, the console checks the syntax of the policy. If the policy contains errors, such as a non-existent user or an action that is not supported by the resource, it returns an error and doesn't save the policy.

The following shows the resource-based policy editor for a bot in the console. The policy editor for a bot alias is similar.

## Resource-based policy

You can use a resource-based policy to grant access permission to other AWS services, IAM users, and roles.

Resource ARN

arn:aws:lex:us-west-2: [REDACTED]:bot/AKWB8PVLD2

Policy

```
1 {  
2   "Version": "2012-10-17",  
3   "Statement": [  
4     {  
5       "Sid": "botRunners",  
6       "Effect": "Allow",  
7       "Principal": {  
8         "AWS": "arn:aws:iam::123456789012:user/botRunner"  
9       },  
10      "Action": [  
11        "lex:RecognizeText",  
12        "lex:RecognizeUtterance",  
13        "lex:StartConversaion"  
14      ],  
15      "Resource": [  
16        "arn:aws:lex:us-west-2:123456789012:bot/AKWB8PVLD2"  
17      ]  
18    }  
19  ]  
20}
```

Cancel Save

### To open the policy editor for a bot

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the **Bots** list, choose the bot whose policy you want to edit.
3. In the **Resource-based policy** section, choose **Edit**.

### To open the policy editor for a bot alias

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. From the **Bots** list, choose the bot that contains the alias that you want to edit.
3. From the left menu, choose **Aliases**, then choose the alias to edit.
4. In the **Resource-based policy** section, choose **Edit**.

## Use the API to specify a resource-based policy

You can use API operations to manage the resource-based policies for your bots and bot aliases. There are operations to create, update and delete policies.

### [CreateResourcePolicy](#)

Adds a new resource policy with the specified policy statements to a bot or bot alias.

### [CreateResourcePolicyStatement](#)

Adds a new resource policy statement to a bot or bot alias.

### [DeleteResourcePolicy](#)

Removes a resource policy from a bot or bot alias.

### [DeleteResourcePolicyStatement](#)

Removes a resource policy statement from a bot or bot alias.

### [DescribeResourcePolicy](#)

Gets a resource policy and the policy revision.

### [UpdateResourcePolicy](#)

Replaces the existing resource policy for a bot or bot alias with a new one.

## Examples

### Java

The following example shows how to use the resource-based policy operations to manage a resource-based policy.

```
/*
 * Create a new policy for the specified bot alias
 * that allows a role to invoke lex:UpdateBotAlias on it.
 * The created policy will have revision id 1.
 */

CreateResourcePolicyRequest createPolicyRequest =
    CreateResourcePolicyRequest.builder()
```

```

        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
        .policy("{\"Version\": \"2012-10-17\", \"Statement\":
[{\\"Sid\\": \\"BotAliasEditor\\\", \\"Effect\\": \\"Allow\\\", \\"Principal\\": \\"AWS\":
\\arn:aws:iam::123456789012:role/BotAliasEditor\"}, \\"Action\\": [\\\"lex:UpdateBotAlias
\\\"], \\"Resource\\": [\\\"arn:aws:lex:Region:123456789012:bot-alias/MYBOTALIAS/TSTALIASID
\\\"]}]})")

lexmodelsv2Client.createResourcePolicy(createPolicyRequest);

/*
 * Overwrite the policy for the specified bot alias with a new policy.
 * Since no expectedRevisionId is provided, this request overwrites the current
revision.
 * After this update, the revision id for the policy is 2.
 */
UpdateResourcePolicyRequest updatePolicyRequest =
UpdateResourcePolicyRequest.builder()
        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
        .policy("{\"Version\": \"2012-10-17\", \"Statement\":
[{\\"Sid\\": \\"BotAliasEditor\\\", \\"Effect\\": \\"Deny\\\", \\"Principal\\": \\"AWS\":
\\arn:aws:iam::123456789012:role/BotAliasEditor\"}, \\"Action\\": [\\\"lex:UpdateBotAlias
\\\"], \\"Resource\\": [\\\"arn:aws:lex:Region:123456789012:bot-alias/MYBOTALIAS/TSTALIASID
\\\"]}]})")

lexmodelsv2Client.updateResourcePolicy(updatePolicyRequest);

/*
 * Creates a statement in an existing policy for the specified bot alias
 * that allows a role to invoke lex:RecognizeText on it.
 * This request expects to update revision 2 of the policy. The request will
fail
 * if the current revision of the policy is no longer revision 2.
 * After this request, the revision id for this policy will be 3.
 */
CreateResourcePolicyStatementRequest createStatementRequest =
CreateResourcePolicyStatementRequest.builder()
        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
        .effect("Allow")

.principal(Principal.builder().arn("arn:aws:iam::123456789012:role/
BotRunner").build())
        .action("lex:RecognizeText")
        .statementId("BotRunnerStatement")
        .expectedRevisionId(2)
        .build();

lexmodelsv2Client.createResourcePolicyStatement(createStatementRequest);

/*
 * Deletes a statement from an existing policy for the specified bot alias by
statementId.
 * Since no expectedRevisionId is supplied, the request will remove the
statement from
 * the current revision of the policy for the bot alias.
 * After this request, the revision id for this policy will be 4.
 */
DeleteResourcePolicyRequest deleteStatementRequest =
DeleteResourcePolicyRequest.builder()
        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
        .statementId("BotRunnerStatement")
        .build();

```

```

lexmodelsv2Client.deleteResourcePolicy(deleteStatementRequest);

/*
 * Describe the current policy for the specified bot alias
 * It always returns the current revision.
 */
DescribeResourcePolicyRequest describePolicyRequest =
    DescribeResourcePolicyRequest.builder()
        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
        .build();

lexmodelsv2Client.describeResourcePolicy(describePolicyRequest);

/*
 * Delete the current policy for the specified bot alias
 * This request expects to delete revision 3 of the policy. Since the revision
id for
 * this policy is already at 4, this request will fail.
 */
DeleteResourcePolicyRequest deletePolicyRequest =
    DeleteResourcePolicyRequest.builder()
        .resourceArn("arn:aws:lex:Region:123456789012:bot-
alias/MYBOTALIAS/TSTALIASID")
        .expectedRevisionId(3);
        .build();

lexmodelsv2Client.deleteResourcePolicy(deletePolicyRequest);

```

## Allow an IAM role to update a bot and list bot aliases

The following example grants permissions for a specific IAM role to call Amazon Lex V2 model building API operations to modify an existing bot. The user can list aliases for a bot and update the bot, but can't delete the bot or bot aliases.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "botBuilders",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/BotBuilder"
            },
            "Action": [
                "lex>ListBotAliases",
                "lex:UpdateBot"
            ],
            "Resource": [
                "arn:aws:lex:Region:123456789012:bot/MYBOT"
            ]
        }
    ]
}
```

## Allow a user to have a conversation with a bot

The following example grants permission for a specific user to call Amazon Lex V2 runtime API operations on a single alias of a bot.

The user is specifically denied permission to update or delete the bot alias.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "botRunners",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:user/botRunner"
            },
            "Action": [
                "lex:RecognizeText",
                "lex:RecognizeUtterance",
                "lex:StartConversation",
                "lex:DeleteSession",
                "lex:GetSession",
                "lex:PutSession"
            ],
            "Resource": [
                "arn:aws:lex:Region:123456789012:bot-alias/MYBOT/MYBOTALIAS"
            ]
        },
        {
            "Sid": "botRunners",
            "Effect": "Deny",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:user/botRunner"
            },
            "Action": [
                "lex:UpdateBotAlias",
                "lex:DeleteBotAlias"
            ],
            "Resource": [
                "arn:aws:lex:Region:123456789012:bot-alias/MYBOT/MYBOTALIAS"
            ]
        }
    ]
}
```

## Allow an AWS service to use a specific Amazon Lex V2 bot

The following example grants permission for AWS Lambda and Amazon Connect to call Amazon Lex V2 runtime API operations.

The condition block is required for service principals, and must use the global context keys AWS:SourceAccount and AWS:SourceArn.

The AWS:SourceAccount is the account ID that is calling the Amazon Lex V2 bot.

The AWS:SourceArn is the resource ARN of the Amazon Connect service instance or Lambda function that the call to the Amazon Lex V2 bot alias originates from.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "connect-bot-alias",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "connect.amazonaws.com"
                ]
            }
        }
    ]
}
```

```

        ],
    },
    "Action": [
        "lex:RecognizeText",
        "lex:StartConversation"
    ],
    "Resource": [
        "arn:aws:lex:Region:123456789012:bot-alias/MYBOT/MYBOTALIAS"
    ],
    "Condition": {
        "StringEquals": {
            "AWS:SourceAccount": "123456789012"
        },
        "ArnEquals": {
            "AWS:SourceArn":
                "arn:aws:connect:Region:123456789012:instance/instance-id"
        }
    }
},
{
    "Sid": "lambda-function",
    "Effect": "Allow",
    "Principal": {
        "Service": [
            "lambda.amazonaws.com"
        ]
    },
    "Action": [
        "lex:RecognizeText",
        "lex:StartConversation"
    ],
    "Resource": [
        "arn:aws:lex:Region:123456789012:bot-alias/MYBOT/MYBOTALIAS"
    ],
    "Condition": {
        "StringEquals": {
            "AWS:SourceAccount": "123456789012"
        },
        "ArnEquals": {
            "AWS:SourceArn": "arn:aws:lambda:Region:123456789012:function/function-name"
        }
    }
}
]
}

```

## AWS managed policies for Amazon Lex V2

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to [create IAM customer managed policies](#) that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see [AWS managed policies](#) in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched.

or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the `ViewOnlyAccess` AWS managed policy provides read-only access to many AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see [AWS managed policies for job functions](#) in the *IAM User Guide*.

## AWS managed policy: AmazonLexReadOnly

You can attach the `AmazonLexReadOnly` policy to your IAM identities.

This policy grants read-only permissions that allow users to view all actions in the Amazon Lex V2 and Amazon Lex model building service.

### Permissions details

This policy includes the following permissions:

- `lex` – Read-only access to Amazon Lex V2 and Amazon Lex resources in the model building service.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "lex:GetBot",  
                "lex:GetBotAlias",  
                "lex:GetBotAliases",  
                "lex:GetBots",  
                "lex:GetBotChannelAssociation",  
                "lex:GetBotChannelAssociations",  
                "lex:GetBotVersions",  
                "lex:GetBuiltInIntent",  
                "lex:GetBuiltInIntents",  
                "lex:GetBuiltInSlotTypes",  
                "lex:GetIntent",  
                "lex:GetIntents",  
                "lex:GetIntentVersions",  
                "lex:GetSlotType",  
                "lex:GetSlotTypes",  
                "lex:GetSlotTypeVersions",  
                "lex:GetUtterancesView",  
                "lex:DescribeBot",  
                "lex:DescribeBotAlias",  
                "lex:DescribeBotChannel",  
                "lex:DescribeBotLocale",  
                "lex:DescribeBotRecommendation",  
                "lex:DescribeBotVersion",  
                "lex:DescribeCustomVocabulary",  
                "lex:DescribeCustomVocabularyMetadata",  
                "lex:DescribeExport",  
                "lex:DescribeImport",  
                "lex:DescribeIntent",  
                "lex:DescribeResourcePolicy",  
                "lex:ListBotAliases",  
                "lex:ListBotVersions",  
                "lex:ListIntents",  
                "lex:ListSlotTypes",  
                "lex:ListUtterances",  
                "lex:UpdateBot",  
                "lex:UpdateBotAlias",  
                "lex:UpdateBotChannel",  
                "lex:UpdateBotLocale",  
                "lex:UpdateBotRecommendation",  
                "lex:UpdateBotVersion",  
                "lex:UpdateCustomVocabulary",  
                "lex:UpdateCustomVocabularyMetadata",  
                "lex:UpdateExport",  
                "lex:UpdateImport",  
                "lex:UpdateIntent",  
                "lex:UpdateResourcePolicy"  
            ]  
        }  
    ]  
}
```

```
        "lex:DescribeSlot",
        "lex:DescribeSlotType",
        "lex>ListBotRecommendations",
        "lex>ListBots",
        "lex>ListBotLocales",
        "lex>ListBotAliases",
        "lex>ListBotChannels",
        "lex>ListBotVersions",
        "lex>ListBuiltInIntents",
        "lex>ListBuiltInSlotTypes",
        "lex>ListExports",
        "lex>ListImports",
        "lex>ListIntents",
        "lex>ListRecommendedIntents",
        "lex>ListSlots",
        "lex>ListSlotTypes",
        "lex>ListTagsForResource",
        "lex>SearchAssociatedTranscripts"
    ],
    "Resource": "*"
}
]
}
```

## AWS managed policy: AmazonLexRunBotsOnly

You can attach the `AmazonLexRunBotsOnly` policy to your IAM identities.

This policy grants read-only permissions that allow access to run Amazon Lex V2 and Amazon Lex conversational bots..

### Permissions details

This policy includes the following permissions:

- `lex` – Read-only access to all actions in the Amazon Lex V2 and Amazon Lex runtime.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lex:PostContent",
                "lex:PostText",
                "lex:PutSession",
                "lex:GetSession",
                "lex>DeleteSession",
                "lex:RecognizeText",
                "lex:RecognizeUtterance",
                "lex:StartConversation"
            ],
            "Resource": "*"
        }
    ]
}
```

## AWS managed policy: AmazonLexFullAccess

You can attach the `AmazonLexFullAccess` policy to your IAM identities.

This policy grants administrative permissions that allow the user permission to create, read, update, and delete Amazon Lex V2 and Amazon Lex resources; and to run Amazon Lex V2 and Amazon Lex conversational bots.

### Permissions details

This policy includes the following permissions:

- **lex** – Allows principals read and write access to all actions in the Amazon Lex V2 and Amazon Lex model building and runtime services.
- **cloudwatch** – Allows principals to view Amazon CloudWatch metrics and alarms.
- **iam** – Allows principals to create and delete service-linked roles, pass roles, and attach and detach policies to a role. The permissions are restricted to "lex.amazonaws.com" for Amazon Lex operations and to "lexv2.amazonaws.com" for Amazon Lex V2 operations.
- **kendra** – Allows principals to list Amazon Kendra indexes.
- **kms** – Allows principals to describe AWS KMS keys and aliases.
- **lambda** – Allows principals to list AWS Lambda functions and manage permissions attached to any Lambda function.
- **polly** – Allows principals to describe Amazon Polly voices and synthesize speech.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "cloudwatch:GetMetricStatistics",  
                "cloudwatch:DescribeAlarms",  
                "cloudwatch:DescribeAlarmsForMetric",  
                "kms:DescribeKey",  
                "kms>ListAliases",  
                "lambda:GetPolicy",  
                "lambda>ListFunctions",  
                "lambda>ListAliases",  
                "lambda>ListVersionsByFunction"  
                "lex:*",  
                "polly:DescribeVoices",  
                "polly:SynthesizeSpeech",  
                "kendra>ListIndices",  
                "iam>ListRoles",  
                "s3>ListAllMyBuckets",  
                "logs:DescribeLogGroups",  
                "s3:GetBucketLocation"  
            ],  
            "Resource": [  
                "*"  
            ]  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "lambda:AddPermission",  
                "lambda:RemovePermission"  
            ],  
            "Resource": "arn:aws:lambda:*:*:function:AmazonLex*",  
            "Condition": {  
                "StringEquals": {  
                    "lambda:Principal": "lex.amazonaws.com"  
                }  
            }  
        }  
    ]  
}
```

```

        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:GetRolePolicy"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots",
                "arn:aws:iam::*:role/aws-service-role/channels.lex.amazonaws.com/
AWSServiceRoleForLexChannels",
                "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexV2Bots*",
                "arn:aws:iam::*:role/aws-service-role/channels.lexv2.amazonaws.com/
AWSServiceRoleForLexV2Channels*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "lex.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/channels.lex.amazonaws.com/
AWSServiceRoleForLexChannels"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "channels.lex.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole"
            ],
            "Resource": [
                "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSServiceRoleForLexV2Bots*"
            ],
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "lexv2.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [

```

```

        "iam:CreateServiceLinkedRole"
    ],
    "Resource": [
        "arn:aws:iam::*:role/aws-service-role/channels.lexv2.amazonaws.com/
AWSERVICERoleForLexV2Channels*"
    ],
    "Condition": {
        "StringEquals": {
            "iam:AWSServiceName": "channels.lexv2.amazonaws.com"
        }
    }
},
{
    "Effect": "Allow",
    "Action": [
        "iam:DeleteServiceLinkedRole",
        "iam:GetServiceLinkedRoleDeletionStatus"
    ],
    "Resource": [
        "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSERVICERoleForLexBots",
        "arn:aws:iam::*:role/aws-service-role/channels.lex.amazonaws.com/
AWSERVICERoleForLexChannels",
        "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSERVICERoleForLexV2Bots*",
        "arn:aws:iam::*:role/aws-service-role/channels.lexv2.amazonaws.com/
AWSERVICERoleForLexV2Channels*"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iam:PassRole"
    ],
    "Resource": [
        "arn:aws:iam::*:role/aws-service-role/lex.amazonaws.com/
AWSERVICERoleForLexBots"
    ],
    "Condition": {
        "StringEquals": {
            "iam:PassedToService": [
                "lex.amazonaws.com"
            ]
        }
    }
},
{
    "Effect": "Allow",
    "Action": [
        "iam:PassRole"
    ],
    "Resource": [
        "arn:aws:iam::*:role/aws-service-role/lexv2.amazonaws.com/
AWSERVICERoleForLexV2Bots*"
    ],
    "Condition": {
        "StringEquals": {
            "iam:PassedToService": [
                "lexv2.amazonaws.com"
            ]
        }
    }
},
{
    "Effect": "Allow",
    "Action": [

```

```

        "iam:PassRole"
    ],
    "Resource": [
        "arn:aws:iam::*:role/aws-service-role/channels.lexv2.amazonaws.com/
AWSLambdaRoleForLexV2Channels*"
    ],
    "Condition": {
        "StringEquals": [
            "iam:PassedToService": [
                "channels.lexv2.amazonaws.com"
            ]
        ]
    }
}
]
}
}

```

## Amazon Lex V2 updates to AWS managed policies

View details about updates to AWS managed policies for Amazon Lex V2 since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Amazon Lex V2 [Document history for Amazon Lex V2 \(p. 362\)](#) page.

Change	Description	Date
<a href="#">AmazonLexReadOnly (p. 341)</a> – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to information about custom vocabularies.	November 16, 2021
<a href="#">AmazonLexFullAccess (p. 342)</a> – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 model building service operations.	August 18, 2021
<a href="#">AmazonLexReadOnly (p. 341)</a> – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 Automated Chatbot Designer operations.	December 1, 2021
<a href="#">AmazonLexFullAccess (p. 342)</a> – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 model building service operations.	August 18, 2021
<a href="#">AmazonLexReadOnly (p. 341)</a> – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 model building service operations.	August 18, 2021
<a href="#">AmazonLexRunBotsOnly (p. 342)</a> – Update to an existing policy	Amazon Lex V2 added new permissions to allow read-only access to Amazon Lex V2 runtime service operations.	August 18, 2021

Change	Description	Date
Amazon Lex V2 started tracking changes	Amazon Lex V2 started tracking changes for its AWS managed policies.	August 18, 2021

## Troubleshooting Amazon Lex V2 identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Amazon Lex V2 and IAM.

### Topics

- [I am not authorized to perform an action in Amazon Lex V2 \(p. 347\)](#)
- [I am not authorized to perform iam:PassRole \(p. 347\)](#)
- [I want to view my access keys \(p. 348\)](#)
- [I'm an administrator and want to allow others to access Amazon Lex V2 \(p. 348\)](#)
- [I want to allow people outside of my AWS account to access my Amazon Lex V2 resources \(p. 348\)](#)

## I am not authorized to perform an action in Amazon Lex V2

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional `my-example-widget` resource but does not have the fictional `lex:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:  
lex:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the `my-example-widget` resource using the `lex:GetWidget` action.

## I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Amazon Lex V2.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Amazon Lex V2. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

## I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, AKIAIOSFODNN7EXAMPLE) and a secret access key (for example, wJalrXUtnFEMI/K7MDENG/bPxRfCYEXAMPLEKEY). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

**Important**

Do not provide your access keys to a third party, even to help [find your canonical user ID](#). By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see [Managing access keys](#) in the *IAM User Guide*.

## I'm an administrator and want to allow others to access Amazon Lex V2

To allow others to access Amazon Lex V2, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in Amazon Lex V2.

To get started right away, see [Creating your first IAM delegated user and group](#) in the *IAM User Guide*.

## I want to allow people outside of my AWS account to access my Amazon Lex V2 resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Amazon Lex V2 supports these features, see [How Amazon Lex V2 works with IAM \(p. 317\)](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

## Using service-linked roles for Amazon Lex V2

Amazon Lex V2 uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to Amazon Lex V2. Service-linked roles are

predefined by Amazon Lex V2 and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up Amazon Lex V2 easier because you don't have to manually add the necessary permissions. Amazon Lex V2 defines the permissions of its service-linked roles, and unless defined otherwise, only Amazon Lex V2 can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting their related resources. This protects your Amazon Lex V2 resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

## Service-linked role permissions for Amazon Lex V2

Amazon Lex V2 uses these service-linked roles.

- **AWSServiceRoleForLexV2Bots\_** – Gives permission to connect your bot to other required services..
- **AWSServiceRoleForLexV2Channels\_** – Gives permission to list bots in an account and to call conversation APIs for a bot..

The AWSServiceRoleForLexV2Bots\_ service-linked role trusts the following service to assume the role:

- `lexv2.amazonaws.com`

The AWSServiceRoleForLexV2Channels\_ service linked role trusts the following service to assume the role:

- `channels.lexv2.amazonaws.com`

The AWSServiceRoleForLexV2Bots\_ role allows Amazon Lex V2 to complete the following actions on the specified resources:

- Action: Use Amazon Polly to synthesize speech on all Amazon Lex V2 resources that the action supports.
- Action: If a bot is configured to use Amazon Comprehend sentiment analysis, detect sentiment on all Amazon Lex V2 resources that the action supports.
- Action: If a bot is configured to store audio logs in an S3 bucket, put object in a specified bucket.
- Action: If a bot is configured to store audio and text logs, create a log stream in and put logs into a specified log group.
- Action: If a bot is configured to use a AWS KMS key to encrypt data, generate a specific data key.
- Action: If a bot is configured to use the KendraSearchIntent intent, query access to a specified Amazon Kendra index.

If a bot is configured to use a channel to communicate with a messaging service, the AWSServiceRoleForLexV2Channels\_ role permissions policy allows Amazon Lex V2 to complete the following actions on the specified resources:

- Action: List permissions on all bots in an account.
- Action: Recognize text, get session and put session permissions on a specified bot alias.

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

## Creating a service-linked role for Amazon Lex V2

You don't need to manually create a service-linked role. When you create or modify a bot, or create a channel integration with a messaging service in the AWS Management Console, the AWS CLI, or the AWS API, Amazon Lex V2 creates the service-linked role for you.

Amazon Lex V2 creates a new service-linked role in your account for each bot. When you create a channel integration to deploy a bot on a messaging platform, Amazon Lex V2 creates a new service-linked role in your account for each channel.

If you delete this service-linked role, and then need to create one again, you can use the same process to create a new role in your account. When you create or modify a bot, or create a channel integration with a messaging service, Amazon Lex V2 creates the service-linked role with a new random suffix.

## Editing a service-linked role for Amazon Lex V2

Amazon Lex V2 does not allow you to edit the `AWSServiceRoleForLexV2Bots_` or `AWSServiceRoleForLexV2Channels_` service-linked roles. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. For `AWSServiceRoleForLexV2Bots_`, Amazon Lex V2 modifies the role when you add additional capabilities to a bot. For example, if you add Amazon Comprehend sentiment analysis to a bot, Amazon Lex V2 adds permission for the `DetectSentiment` action to `AWSServiceRoleForLexV2Bots_`. You can edit the description of either role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

## Deleting a service-linked role for Amazon Lex V2

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must clean up the resources for your service-linked role before you can manually delete it.

### Note

If the Amazon Lex V2 service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

### To delete Amazon Lex V2 resources used by `AWSServiceRoleForLexV2Bots_`

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Choose to bot to delete.
3. From the left menu, choose **Settings**.
4. From **IAM permissions**, record the **Runtime role identifier**.
5. From the left menu, choose **Bots**.
6. From the list of bots, choose the radio button next to the bot to delete.
7. From the **Action**, choose **Delete**.

### To delete Amazon Lex V2 resources used by `AWSServiceRoleForLexV2Channels_`

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.

2. Choose to bot to delete.
3. From the left menu, choose **Bot versions**, then choose **Channel integrations**.
4. Choose the channel to delete.
5. From **General configuration**, choose the choose the role name. The IAM management console opens in a new tab with the role chosen.
6. In the Amazon Lex V2 console, choose **Delete**, then choose **Delete** again to delete the channel.
7. In the IAM management console, choose **DeleteRole** to remove the channel integration role.

#### To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the AWSServiceRoleForLexV2Bots\_ or AWSServiceRoleForLexV2Channels\_ service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

## Supported regions for Amazon Lex V2 service-linked roles

Amazon Lex V2 supports using service-linked roles in all of the regions where the service is available. For more information, see [AWS Regions and Endpoints](#).

## Logging and monitoring in Amazon Lex V2

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Lex V2 and your other AWS solutions. AWS provides the following monitoring tools to watch Amazon Lex V2, report when something is wrong, and take automatic actions when appropriate:

- *Amazon CloudWatch* monitors your AWS resources and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the [Amazon CloudWatch User Guide](#).
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the [AWS CloudTrail User Guide](#).

## Compliance validation for Amazon Lex V2

Third-party auditors assess the security and compliance of Amazon Lex V2 as part of multiple AWS compliance programs. Amazon Lex V2 is a HIPAA eligible service. It is PCI, SOC, and ISO compliant.

To learn whether an AWS service is within the scope of specific compliance programs, see [AWS services in Scope by Compliance Program](#) and choose the compliance program that you are interested in. For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- [Architecting for HIPAA Security and Compliance on Amazon Web Services](#) – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

**Note**

Not all AWS services are HIPAA eligible. For more information, see the [HIPAA Eligible Services Reference](#).

- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

## Resilience in Amazon Lex V2

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

In addition to the AWS global infrastructure, Amazon Lex V2 offers several features to help support your data resiliency and backup needs.

## Infrastructure security in Amazon Lex V2

As a managed service, Amazon Lex V2 is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of Security Processes](#) whitepaper.

You use AWS published API calls to access Amazon Lex V2 through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

## Amazon Lex V2 and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon Lex V2 by creating an *interface VPC endpoint*. Interface endpoints are powered by [AWS PrivateLink](#), a technology that enables you to

privately access Amazon Lex V2 APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Amazon Lex V2 APIs. Traffic between your VPC and Amazon Lex V2 does not leave the Amazon network.

Each interface endpoint is represented by one or more [Elastic Network Interfaces](#) in your subnets.

For more information, see [Interface VPC endpoints \(AWS PrivateLink\)](#) in the *Amazon VPC User Guide*.

## Considerations for Amazon Lex V2 VPC endpoints

Before you set up an interface VPC endpoint for Amazon Lex V2, ensure that you review [Interface endpoint properties and limitations](#) in the *Amazon VPC User Guide*.

Amazon Lex V2 supports making calls to all of its API actions from your VPC.

## Creating an interface VPC endpoint for Amazon Lex V2

You can create a VPC endpoint for the Amazon Lex V2 service using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see [Creating an interface endpoint](#) in the *Amazon VPC User Guide*.

Create a VPC endpoint for Amazon Lex V2 using the following service name:

- com.amazonaws.*region*.models-v2-lex
- com.amazonaws.*region*.runtime-v2-lex

If you enable private DNS for the endpoint, you can make API requests to Amazon Lex V2 using its default DNS name for the Region, for example, `runtime-v2-lex.us-east-1.amazonaws.com`.

For more information, see [Accessing a service through an interface endpoint](#) in the *Amazon VPC User Guide*.

## Creating a VPC endpoint policy for Amazon Lex V2

You can attach an endpoint policy to your VPC endpoint that controls access to Amazon Lex V2. The policy specifies the following information:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see [Controlling access to services with VPC endpoints](#) in the *Amazon VPC User Guide*.

### Example: VPC endpoint policy for Amazon Lex V2 actions

The following is an example of an endpoint policy for Amazon Lex V2. When attached to an endpoint, this policy grants access to the listed Amazon Lex V2 actions for all principals on all resources.

```
{  
  "Statement": [  
    {  
      "Principal": "*",  
      "Action": "lex:  
        -  
        *  
      ",  
      "Resource": "  
        *  
      "}  
    ]  
}
```

```
    "Effect": "Allow",
    "Action": [
        "lex:RecognizeText",
        "lex:RecognizeUtterance",
        "lex:StartConversation",
        "lex>DeleteSession",
        "lex:GetSession",
        "lex:DeleteSession"
    ],
    "Resource": "*"
}
```

# Amazon Lex V1 to V2 migration guide

The Amazon Lex V2 console and APIs make it easier to build and manage bots. Use this guide to learn about the improvements in the Amazon Lex V2 API as you migrate bots.

You migrate a bot using the Amazon Lex console or API. For more information see [Migrating a bot](#) in the *Amazon Lex developer guide*.

## Amazon Lex V2 overview

Multiple languages can be added to a bot so you can manage them as a single resource. A simplified information architecture lets you efficiently manage your bot versions. Capabilities such as a 'conversation flow', partial saving of bot configuration and bulk upload of utterances give you more flexibility.

### Multiple languages in a bot

You can add multiple languages with the Amazon Lex V2 API. You add, modify, and build each language independently. Resources such as slot types are scoped at the language level. You can quickly move between different languages to compare and refine the conversations. You can use one dashboard in the console to review utterances for all languages for faster analysis and iterations. A bot operator can manage permissions and logging operations for all languages with one bot configuration. You must provide a language as a runtime parameter to converse with a Amazon Lex V2 bot. For more information, see [Languages and locales supported by Amazon Lex V2 \(p. 4\)](#).

### Simplified information architecture

The Amazon Lex V2 API follows a simplified information architecture (IA) with intent and slot types scoped to a language. You version at the bot level so that resources such as intents and slot types aren't versioned individually. By default, a bot is created with a *Draft* version that is mutable and used for testing changes. You can create numbered snapshots from the draft version. You choose the languages to include in a version. All resources within the bot (languages, intents, slot types) are archived as part of creating a bot version. For more information, see [Creating versions \(p. 48\)](#).

### Improved builder productivity

You have additional builder productivity tools and capabilities that give you more flexibility and control of your bot design process.

### Save partial configuration

The Amazon Lex V2 API enables you to save partial changes during development. For example, you can save a slot that references a deleted slot type. This flexibility enables you to save your work and return to it later. You can resolve these changes before building the bot. In Amazon Lex V2 the partial save can be applied to slots, versions, and aliases.

## Renaming resources

With Amazon Lex V2 you can rename a resource after it's created. Use a resource name to associate user-friendly metadata with each resource. The Amazon Lex V2 API assigns every resource a unique 10-character resource ID. All resources have a resource name. You can rename the following resources:

- Bot
- Intent
- Slot type
- Slot
- Alias

You can use resource IDs to read and modify your resources. If you are using the AWS Command Line Interface or the Amazon Lex V2 API to work with Amazon Lex V2, resource IDs are required for certain commands.

## Simplified management of Lambda functions

In the Amazon Lex V2 API you define one Lambda function per language instead of a function for each intent. The Lambda function is configured in the alias for the language and is used for both the dialog and fulfillment code hook. You can still choose to enable or disable the dialog and fulfillment code hooks independently for each intent. For more information, see [Using an AWS Lambda function \(p. 194\)](#).

## Granular settings

The Amazon Lex V2 API moves the voice and intent classification confidence score threshold from the bot to the language scope. The sentiment analysis flag moves from bot scope to alias scope. Session time out and privacy settings at the bot scope, and conversation logs at the alias scope, remain unchanged.

## Default fallback intent

The Amazon Lex V2 API adds a default fallback intent when you create a language. Use it to configure error handling for your bot instead of specific error-handling prompts.

## Optimized session variable update

With the Amazon Lex V2 API you can update session state directly with the [RecognizeText](#) and [RecognizeUtterance](#) operations without any dependency on session APIs.

# Creating Amazon Lex V2 resources with AWS CloudFormation

Amazon Lex V2 is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want (such as Amazon Lex V2 chatbots), and AWS CloudFormation provisions and configures those resources for you.

When you use AWS CloudFormation, you can reuse your template to set up your Amazon Lex V2 resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

## Amazon Lex V2 and AWS CloudFormation templates

To provision and configure resources for Amazon Lex V2 and related services, you must understand [AWS CloudFormation templates](#). Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see [What is AWS CloudFormation Designer?](#) in the [AWS CloudFormation User Guide](#).

Amazon Lex V2 supports creating the following resources in AWS CloudFormation:

- [AWS::Lex::Bot](#)
- [AWS::Lex::BotAlias](#)
- [AWS::Lex::BotVersion](#)
- [AWS::Lex::ResourcePolicy](#)

For more information, including examples of JSON and YAML templates for these resources, see the [Amazon Lex V2 resource type reference](#) in the [AWS CloudFormation User Guide](#).

## Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- [AWS CloudFormation](#)
- [AWS CloudFormation user guide](#)
- [AWS CloudFormation API reference](#)
- [AWS CloudFormation Command line interface user guide](#)

# Guidelines and quotas

## Topics

- [Regions \(p. 358\)](#)
- [General guidelines \(p. 358\)](#)
- [Quotas \(p. 359\)](#)

## Regions

For a list of AWS Regions where Amazon Lex V2 is available, see [AWS regions and endpoints](#) in the [AWS general reference](#).

## General guidelines

This topic describes general guidelines when using Amazon Lex V2.

- **Signing requests** – All Amazon Lex V2 model-building and runtime requests in the [API Reference](#) use signature V4 for authenticating requests. For more information about authenticating requests, see [Signature Version 4 signing process](#) in the [Amazon Web Services general reference](#).
- Note the following about how Amazon Lex V2 captures slot values from user utterances:

Amazon Lex V2 uses the enumeration values that you provide in a slot type definition to train its machine learning models. Suppose that you define an intent called GetPredictionIntent with the following sample utterance:

```
"Tell me the prediction for {sign}"
```

where {sign} is a slot with the custom type ZodiacSign. It has 12 enumeration values, Aries through Pisces. From the user utterance "Tell me the prediction for ..." Amazon Lex V2 understands that the following is a zodiac sign.

When the valueSelectionStrategy field is set to OriginalValue using the [CreateSlotType](#) operation, or if **Expand values** is selected in the console, if the user says "Tell me the prediction for earth", Amazon Lex V2 infers that "earth" is a ZodiacSign value and passes it to your client application or Lambda function. You must check that slot values are valid values before using them in your fulfillment activity.

If you set the valueSelectionStrategy field to TopResolution using the [CreateSlotType](#) operation or if **Restrict to slot values and synonyms** is selected in the console, the values that are returned are limited to the values that you defined for the slot type. For example, if the user says "Tell me the prediction for earth", the value would not be recognized because it is not one of the values defined for the slot type. When you define synonyms for slot values, they are recognized the same as a slot value, however, the slot value is returned instead of the synonym.

When Amazon Lex V2 calls a Lambda function or returns the result of a speech interaction with your client, the case of the slot values is not guaranteed. In text interactions, the case of the slot values matches the text entered or the slot value, depending on the value of the valueResolutionStrategy field.

- When defining slot values that contain acronyms, use the following patterns:

- Capital letters separated by periods (D.V.D.)
- Capital letters separated by spaces (D V D)
- The [AMAZON.Date \(p. 78\)](#) and [AMAZON.Time \(p. 81\)](#) built-on slot types capture both absolute and relative dates and times. Relative dates and times are resolved in the region where Amazon Lex V2 is processing the request.

For the AMAZON.Time built-in slot type, if the user doesn't specify that a time is before or after noon, the time is ambiguous and Amazon Lex V2 will prompt the user again. We recommend prompts that elicit an absolute time. For example, use a prompt such as "When do you want your pizza delivered? You can say 6 PM or 6 in the evening."

- Providing confusable training data in your bot reduces the ability of Amazon Lex V2 to understand user input. Consider these examples:

Suppose you have two intents (`OrderPizza` and `OrderDrink`) in your bot and both are configured with an "I want to order" utterance. This utterance does not map to a specific intent that Amazon Lex V2 can learn from while building the language model for the bot at build time. As a result, when a user inputs this utterance at runtime, Amazon Lex V2 can't pick an intent with a high degree of confidence.

When you have two intents with the same utterance, use input contexts to help Amazon Lex distinguish between the two intents at runtime. For more information, see [Setting intent context](#).

- When you use the `TSTALIASID` alias, keep in mind the following:
  - The `TSTALIASID` alias of your bot points to the Draft version and should only be used for manual testing. Amazon Lex limits the number of runtime requests that you can make to the `TSTALIASID` alias of the bot.
  - When you update the Draft version of the bot, Amazon Lex terminates any in-progress conversations for any client application using the `TSTALIASID` alias of the bot. Generally, you should not use the `TSTALIASID` alias of a bot in production because the Draft version can be updated. You should publish a version and an alias and use them instead.
  - When you update an alias, Amazon Lex takes a few minutes to pick up the changes. When you modify the Draft version of the bot, the change is picked up by the `TSTALIASID` alias immediately.
  - The runtime API operations [RecognizeText](#) and [RecognizeUtterance](#) take a session ID as a required parameter. Developers can set this to any value that meets the constraints described in the API. We recommend that you don't use this parameter to send any confidential information such as user logins, emails, or social security numbers. This ID is primarily used to uniquely identify a conversation with a bot.

## Quotas

Service quotas, also referred to as limits, are the maximum number of service resources for your AWS account. For more information, see [AWS service quotas](#) in the [AWS general reference](#).

Service quotas can be adjusted or increased. Contact AWS customer support to increase a quota. It can take a few days to increase a service quota. If you're increasing your quota as part of a larger project, be sure to add this time to your plan.

## Build-time quotas

The following maximum quotas are enforced when you are creating a bot.

Description	Default
Bots per AWS account	100

Description	Default
Bot channel associations per AWS account	5,000
Versions per bot	100
Intents per locale in each bot	<ul style="list-style-type: none"> <li>1,000 in en-AU, en-GB, and en-US</li> <li>250 in all other locales</li> </ul>
Slots per locale in each bot	<ul style="list-style-type: none"> <li>4,000 in en-AU, en-GB, and en-US</li> <li>2,000 in all other locales</li> </ul>
Custom slot types per bot locale	<ul style="list-style-type: none"> <li>250 in en-AU, en-GB, and en-US</li> <li>100 in all other locales</li> </ul>
Custom slot type values and synonyms per locale in each bot	50,000
Total characters in sample utterances per locale in each bot	<ul style="list-style-type: none"> <li>2,000,000 in en-AU, en-GB, and en-US</li> <li>200,000 in all other locales</li> </ul>
Channel associations per bot alias	10
Slots per intent	100
Sample utterances per intent	1,500
Characters per sample utterance	500
Text response length	4,000
Sample utterances per slot	10
Characters per sample slot utterance	500
Prompts per slot	30
Values and synonyms per custom slot type	10,000
Characters per custom slot type value	500
Characters in a channel association name	100
Number of concurrent Automated Chatbot Designer analysis jobs across all bots in your account per region	10

## Runtime quotas

The following maximum quotas are enforced at runtime.

Description	Default
Input text size for <a href="#">RecognizeText</a> and <a href="#">RecognizeUtterance</a>	1024 Unicode characters
Speech input length for <a href="#">RecognizeUtterance</a> operation	15 seconds

Description	Default
Size of RecognizeUtterance headers	16 KB
Size of combined request and session headers for RecognizeUtterance	12 KB
Maximum number of concurrent text-mode conversations for RecognizeText, RecognizeUtterance, or StartConversation for the TestBotAlias	2
Maximum number of concurrent text-mode conversations for RecognizeText, RecognizeUtterance, or StartConversation for other aliases	50
Maximum number of concurrent voice-mode conversations for RecognizeUtterance for the TestBotAlias	2
Maximum number of concurrent voice-mode conversations for RecognizeUtterance for other aliases	125
Maximum number of concurrent voice-mode conversations for StartConversation for the TestBotAlias	2
Maximum number of concurrent voice-mode conversations for StartConversation for other aliases	200
Maximum number of concurrent session management operations (PutSession, GetSession, or DeleteSession) when using the TestBotAlias	2
Maximum number of concurrent session management operations (PutSession, GetSession, or DeleteSession) when using other aliases	50
Maximum input size to a Lambda function	12 KB
Maximum output size of a Lambda function	25 KB
Maximum size of session attributes in Lambda function output	12 KB

# Document history for Amazon Lex V2

- **Latest documentation update:** November 7, 2022

The following table describes important changes in each release of Amazon Lex V2. For notification about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Date
New feature	Amazon Lex V2 can display an alternative representation for a phrase or word by using the console or APIs to customize the speech to text output. For more information, see <a href="#">Creating a custom vocabulary for speech recognition</a> .	November 7, 2022
New feature	Amazon Lex V2 can add a weight attribute to an item element that will represent the degree to which the phrase is boosted during speech recognition. For more information, see <a href="#">Grammar Weights</a> .	October 28, 2022
New feature	Amazon Lex V2 can be used to capture free form input from the end user made up of words or characters using AMAZON.FreeFormInput. For more information, see <a href="#">Built-in Slot Types</a> .	October 19, 2022
New feature	Amazon Lex V2 can display an alternative representation for a phrase or word in the final speech to text output. For more information, see <a href="#">Creating a custom vocabulary for speech recognition</a> .	October 19, 2022
New feature	Amazon Lex V2 now supports the Hindi (India) and Dutch (The Netherlands) locales. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	October 14, 2022
New feature	There was an update to the way Amazon Lex V2 manages user input. Now you can select	September 22, 2022

	whether Amazon Lex V2 accepts text, audio or DTMF input at any point in the conversation flow. For more information, see <a href="#">Configurable Attributes</a> .	
New feature	There was an update to the way Amazon Lex V2 manages conversation flows. Visual conversation builder is a drag and drop conversation builder to easily design and visualize conversation paths. For more information, see <a href="#">Visual Conversation Builder</a> .	September 14, 2022
New feature	There was an update to the way Amazon Lex V2 builds complex slots. You can now create complex subslots within slots to manage intents in complex conversation design. For more information, see <a href="#">Building composite slots</a> .	September 9, 2022
New feature	There was an update to the way Amazon Lex V2 manages the flow of conversational paths with your users. You can now create complex conversational paths by ordering the next step in the conversation. For more information, see <a href="#">Creating conversation paths</a> .	August 17, 2022
New feature	There was an update to the way Amazon Lex V2 manages the flow of conversations with your users. You can now create complex conversations by ordering the prompts. For more information, see <a href="#">Configuring prompts</a> .	July 5, 2022
New feature	There was an update to the way Amazon Lex V2 manages the flow of conversations with your users. You can now create complex conversations using conditions. For more information, see <a href="#">Understanding the new conversation flows</a> .	May 3, 2022
New feature	Added example industry grammars for the built-in grammar slot type. For more information, see <a href="#">Industry grammars</a> .	March 22, 2022

New feature	Added documentation about integrating Amazon Lex V2 with the Amazon Chime SDK. For more information, see <a href="#">Amazon Chime SDK</a> .	March 18, 2022
New feature	Amazon Lex V2 now provides confidence scores for voice transcriptions. Use the score to help determine the correct response from the user. For more information, see <a href="#">Using voice transcription confidence scores</a> .	January 27, 2022
New feature	You can now add contextual and dynamic hints to slots to improve the accuracy of your bot. For more information, see <a href="#">Using hints to improve accuracy</a> .	January 13, 2022
New feature	Amazon Lex V2 adds support for custom vocabularies to improve speech recognition for audio input. For more information, see <a href="#">Creating a custom vocabulary to improve speech recognition</a> .	January 12, 2022
New feature	Amazon Lex V2 now supports AWS PrivateLink. For more information, see <a href="#">VPC endpoints (AWS PrivateLink)</a> .	January 7, 2022
New feature	Amazon Lex V2 now supports the Catalan (Spain) locale. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	January 3, 2022
New feature	You can now create slot types using your own custom grammar. For more information, see <a href="#">Using a custom grammar slot type</a> .	December 20, 2021
New feature	AWS CloudFormation now supports Amazon Lex V2. For more information, see <a href="#">AWS CloudFormation resources</a> .	December 20, 2021
New feature	Amazon Lex V2 now supports the Portuguese (Brazil), Portuguese (Portugal), and Mandarin (PRC) locales. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	December 16, 2021

New feature	Amazon Lex V2 now provides a preview of the Automated Chatbot Designer to help you get started creating a chatbot from contact center transcripts. For more information, see <a href="#">Using the Automated Chatbot Designer (Preview)</a> .	December 1, 2021
New feature	You can now use spell-by-letter and spell-by-word styles for entering slot values that Amazon Lex V2 has difficulty understanding. For more information, see <a href="#">Using spelling styles to capture slot values</a> .	November 19, 2021
New feature	You can now use Amazon Polly neural text to speech (NTTS) voices for audio conversations with your users. For more information, see <a href="#">Voices in Amazon Polly</a> .	November 19, 2021
New feature	Amazon Lex V2 now supports the English (South Africa) locale. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	November 9, 2021
New feature	Amazon Lex V2 now supports the German (Austria) locale. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	November 5, 2021
New feature	You can now provide users with update messages that play at the start of a fulfillment function and periodically while the function runs. You can also create messages that inform the user of status of fulfillment when the function is complete. For more information, see <a href="#">Configuring fulfillment progress updates</a> .	October 7, 2021
Region expansion	Amazon Lex V2 is now available in Africa (Cape Town) (af-south-1) and Asia Pacific (Seoul) (ap-northeast-2).	September 22, 2021
New feature	You can now view statistics for the utterances that your users send to your bot. For more information, see <a href="#">Viewing utterance statistics</a> .	September 22, 2021

New feature	Amazon Lex V2 now supports the Korean (Korea) locale. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	September 9, 2021
New feature	Amazon Lex V2 now provides a built-in slot type for UK postal codes. For more information, see <a href="#">AMAZON.UKPostalCode</a> .	July 27, 2021
New feature	Amazon Lex V2 now supports the English (Indian) locale. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	July 15, 2021
New feature	Amazon Lex V2 now provides a tool to migrate a bot from Amazon Lex V1 to the Amazon Lex V2 API. For more information, see <a href="#">Migrating a bot</a> in the <i>Amazon Lex Developer Guide</i> .	July 13, 2021
New feature	Amazon Lex V2 now enables you to accept multiple values for a single slot in the English (US) language. For more information, see <a href="#">Using multiple values in a slot</a> .	June 15, 2021
New feature	You can now build larger bots for English languages. For more information, see <a href="#">Quotas</a> .	June 11, 2021
New feature	Use Amazon Lex V2 resource-based policies to manage access to your bots and bot aliases. For more information, see <a href="#">Resource-based policies within Amazon Lex V2</a> .	May 20, 2021
New feature	Amazon Lex V2 now enables you to import and export bots and bot locales. You can use this feature to copy bots and bot locales between accounts and AWS Regions. For more information, see <a href="#">Importing and exporting</a> .	May 18, 2021
Region expansion	Amazon Lex V2 is now available in Canada (Central) (ca-central-1).	May 17, 2021

New feature	Amazon Lex V2 now supports the Japanese (Japan) locale. For more information, see <a href="#">Languages and locales supported by Amazon Lex V2</a> .	April 1, 2021
New feature	Amazon Lex V2 now supports three new built-in slot types: AMAZON.City, AMAZON.Country, and AMAZON.State.	March 12, 2021
New guide (p. 362)	This is the first release of the <i>Amazon Lex V2 user guide</i> .	January 21, 2021

# API reference

The [API Reference](#) is now a separate document.

# AWS glossary

For the latest AWS terminology, see the [AWS glossary](#) in the *AWS General Reference*.