

---

# Amazon Lex

## Developer Guide





## **Amazon Lex: Developer Guide**

Copyright © 2017 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? .....	1
Are You a First-time User of Amazon Lex? .....	2
How It Works .....	3
Programming Model .....	5
Runtime API Operations .....	5
Lambda Functions as Code Hooks .....	6
Managing Messages (Prompts and Statements) .....	7
Types of Messages .....	8
Contexts for Configuring Messages .....	9
Supported Message Formats .....	12
Response Cards .....	12
Managing Conversation Context .....	14
Session Timeout .....	14
Cross-Intent Information Sharing .....	15
Deployment Options .....	15
Built-in Intents and Slot Types .....	15
Built-in Intents .....	16
Built-in Slots .....	16
Getting Started .....	17
Exercise 1: Create a Bot Using a Blueprint .....	17
Amazon Lex Bot: Blueprint Overview .....	18
AWS Lambda Function: Blueprint Summary .....	18
Step 1: Prepare .....	19
Step 2: Create an Amazon Lex Bot .....	21
Step 3 (Optional): Review the Details of Information Flow .....	22
Step 4: Create a Lambda Function .....	27
Step 5: Update Intent Configuration: Add the Lambda Function as Code Hook .....	27
Step 6 (Optional): Review the Details of Information Flow .....	29
Exercise 2: Create a Custom Bot .....	39
Step 1: Prepare .....	39
Step 2: Create a Bot .....	42
Step 3: Create Slot Types .....	43
Step 4: Create an Intent .....	44
Step 5: Configure Error Handling .....	46
Step 6: Build and Test the Bot .....	47
Exercise 3: Publish a Version and Create an Alias .....	48
Versioning and Aliases .....	50
Versioning .....	50
Creating an Amazon Lex Bot (the \$LATEST version) .....	50
Publishing an Amazon Lex Bot Version .....	51
Updating an Amazon Lex Resource .....	51
Deleting an Amazon Lex Resource and a Specific Version .....	52
Aliases .....	52
Using Lambda Functions .....	54
Lambda Function Input Event and Response Format .....	54
Input Event Format .....	54
Response Format .....	57
Amazon Lex and AWS Lambda Blueprints .....	59
Deploying Bots .....	61
Deploying an Amazon Lex Bot on a Messaging Platform .....	61
Integrating with Facebook .....	61
Deploying an Amazon Lex Bot in Mobile Applications .....	65
Bot Examples .....	66
Example Bot: ScheduleAppointment .....	66
Overview of the Bot Blueprint (ScheduleAppointment) .....	67

Overview of the Lambda Function Blueprint (lex-make-appointment) .....	68
Step 1: Prepare .....	68
Step 2: Create an Amazon Lex Bot .....	69
Step 3: Create a Lambda function .....	71
Step 4: Update the Intent: Configure a Code Hook .....	71
Example Bot: BookTrip .....	72
Step 1: Blueprint Review .....	73
Step 2: Prepare .....	75
Step 3: Create an Amazon Lex Bot .....	75
Step 4: Create a Lambda function .....	78
Step 5: Add the Lambda Function as a Code Hook .....	78
Details of Information Flow .....	81
Example: Using a Response Card .....	96
Guidelines and Limits .....	99
General Guidelines .....	99
Limits .....	101
API Reference .....	105
Actions .....	105
PostContent .....	106
PostText .....	113
Data Types .....	117
Button .....	119
GenericAttachment .....	120
ResponseCard .....	121
Document History .....	122
AWS Glossary .....	123

# What Is Amazon Lex?

---

*This is prerelease documentation for a service in preview release. It is subject to change.*

Amazon Lex is an AWS service for building conversational interfaces for any applications using voice and text. With Amazon Lex, the same conversational engine that powers Amazon Alexa is now available to any developer, enabling you to build sophisticated, natural language chatbots into your new and existing applications. Amazon Lex provides the deep functionality and flexibility of natural language understanding (NLU) and automatic speech recognition (ASR) to enable you to build highly engaging user experiences with lifelike, conversational interactions and create new categories of products.

Amazon Lex enables any developer to build conversational chatbots quickly. With Amazon Lex, no deep learning expertise is necessary—you just specify the basic conversation flow in the Amazon Lex console to create a bot. Amazon Lex manages the dialogue 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 provides pre-built integration with AWS Lambda, and you can easily integrate with many other services on the AWS platform including Amazon Cognito, AWS Mobile Hub, Amazon CloudWatch, and Amazon DynamoDB. Integration with AWS Lambda provides bots access to pre-built serverless enterprise connectors, to link to data in SaaS applications, such as Salesforce, HubSpot or Marketo.

Some of the benefits of using Amazon Lex include:

- **Simplicity** – Amazon Lex provides a simple to use console to guide you through creating your own chatbot in minutes. You supply just a few example phrases and Amazon Lex 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** – Powered by the same technology as Alexa, Amazon Lex provides ASR and NLU technologies to create a Speech Language Understanding (SLU) system. Through SLU, Amazon Lex 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 puts deep learning technologies within reach of all developers, powered by the same technology as Alexa. Amazon Lex chatbots convert incoming speech to text and understand the user intent to generate an intelligent response, so you can focus on building your bots with differentiated value-add for your customers, to define entirely new categories of products made possible through conversational interfaces.

- **Seamlessly deploy and scale** – With Amazon Lex, you can build, test, and deploy your chatbots directly from the Amazon Lex console. Amazon Lex enables you to easily publish your voice or text chatbots for use on mobile devices, web apps, and chat services (for example, Facebook Messenger). Amazon Lex scales automatically so 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 has native interoperability with other AWS services such as Amazon Cognito, AWS Lambda, Amazon CloudWatch, and AWS Mobile Hub. You can take advantage of the power of the AWS platform for security, monitoring, user authentication, business logic, storage and mobile app development.
- **Cost-effective** – With Amazon Lex, there are no upfront costs or minimum fees. You are only charged for the text or speech requests that are made. The pay-as-you-go pricing and low cost per request make the service a cost-effective way to build conversational interfaces. With the Amazon Lex free tier, you can easily try Amazon Lex without any initial investment.

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

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

1. [Getting Started \(p. 17\)](#) – In this section you set your account and test Amazon Lex.
2. [API Reference \(p. 105\)](#) – This section provides additional examples that you can use to explore Amazon Lex.

# Amazon Lex: How It Works

---

*This is prerelease documentation for a service in preview release. It is subject to change.*

Amazon Lex enables you to build applications using a speech or text interface powered by the same technology that powers Amazon Alexa. Following are the typical steps you perform when working with Amazon Lex:

1. Create a bot and configure it with one or more intents that you want to support. You add the configuration so that the bot is able to understand the user's goal (intent), engage in conversation with the user to elicit information, and, after the user provides the necessary data, fulfill the user's intent.
2. Test the bot. You can use the test window client provided by the Amazon Lex console.
3. Publish a version and create an alias.
4. Deploy the bot. You can deploy the bot on platforms such as mobile applications or messaging platforms such as Facebook Messenger.

Before you get started, familiarize yourself with the following Amazon Lex 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 bot is powered by Automatic Speech Recognition (ASR) and Natural Language Understanding (NLU) capabilities. This is the same technology that powers Amazon Alexa.

Amazon Lex bots can understand user input via text or speech and converse in natural language. You can create Lambda functions and add them as code hook in your intent configuration to perform user data validation and perform fulfillment tasks.



- **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 a bot that performs ordering pizza 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 please" 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 (for example, place order with a local pizza shop). You can create a Lambda function (recommended) to fulfill the intent. The Lambda function is also referred to as a code hook.

Optionally, you can configure the intent to simply return the information back to the client application to do the necessary fulfillment.

In addition to custom intents such as the one to order pizza, Amazon Lex also provides built-in intents to quickly set up your bot. For more information, see [Built-in Intents and Slot Types \(p. 15\)](#).

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

For example, the `OrderPizza` intent requires slots such as pizza size, crust type, and number of pizzas. In the intent configuration, you add these slots. For each slot you add to the intent configuration, you provide slot type and a prompt for Amazon Lex to send to the client to elicit data from the user. A user can reply with a slot value that includes additional words, such as "large pizza please" or "lets stick with small," and Amazon Lex can still understand the intended slot value.

- **Slot type** – Each slot has a type. You can create your custom slot types or 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 also provides built-in slot types. For example, `AMAZON.NUMBER` is a built-in slot type that you might use with the number of pizzas ordered. For more information, see [Built-in Intents and Slot Types \(p. 15\)](#).

The following topics provide additional information. We recommend that you review them in order and then explore the [Getting Started \(p. 17\)](#) exercises.

Topics

- [Programming Model \(p. 5\)](#)
- [Managing Messages \(Prompts and Statements\) \(p. 7\)](#)
- [Managing Conversation Context \(p. 14\)](#)
- [Bot Deployment Options \(p. 15\)](#)
- [Built-in Intents and Slot Types \(p. 15\)](#)

## Programming Model

A *bot* is the primary resource type in Amazon Lex. In addition, the other resource types in Amazon Lex are *intent*, *slot type*, *alias*, and *bot channel association*. You create and manage these resource using the Amazon Lex console.

### Note

Currently, the build-time API that the Amazon Lex console uses to create and manage resources is not publicly available.

After you create a bot you can deploy it on one of the [supported platforms](#). At runtime (when a user interacts with the bot), these application clients can send requests to Amazon Lex using the runtime API provided by Amazon Lex. For example, when a user says "I want to order pizza", your client can send a request to Amazon Lex using one of the runtime API operations and include user input in the request. Note that users can provide both speech and text input.

You can also create Lambda functions and add them as code hooks in your intent configuration to perform runtime activities such as initialization/validation of user data and intent fulfillment. The following sections provide additional information.

### Topics

- [Runtime API Operations \(p. 5\)](#)
- [Lambda Functions as Code Hooks \(p. 6\)](#)

## Runtime API Operations

Client applications use the following runtime API operations to communicate with Amazon Lex at runtime:

- `PostContent` – Takes speech or text input and returns current intent information and a message to convey to the user, which can be text or speech. Currently, Amazon Lex supports the following audio formats:

Input audio formats – LPCM and Opus

Output audio formats – MPEG, OGG, and PCM

- `PostText` – Takes text as input and returns the current intent information and a message to convey to the user as text.

For example, the test window client in the Amazon Lex console uses `PostText` API to send requests to Amazon Lex. You use this test window in the [Getting Started \(p. 17\)](#) exercises.

Your client application uses the runtime API against a specific Amazon Lex bot to understand user utterances (that is, user input text or voice). For example, suppose a user says "I want pizza." The

client sends this user input to Amazon Lex using one of the runtime API operations. From the user input, Amazon Lex recognizes that the user request is for the OrderPizza intent (one of the intents defined in the bot). Amazon Lex follows up and engages the user in conversation to elicit the required information (such as pizza size, toppings, and number of pizzas). After the user provides all of the necessary slot data, Amazon Lex invokes the code hook to fulfill the intent or returns the intent data to the client (according to the intent configuration).

## Lambda Functions as Code Hooks

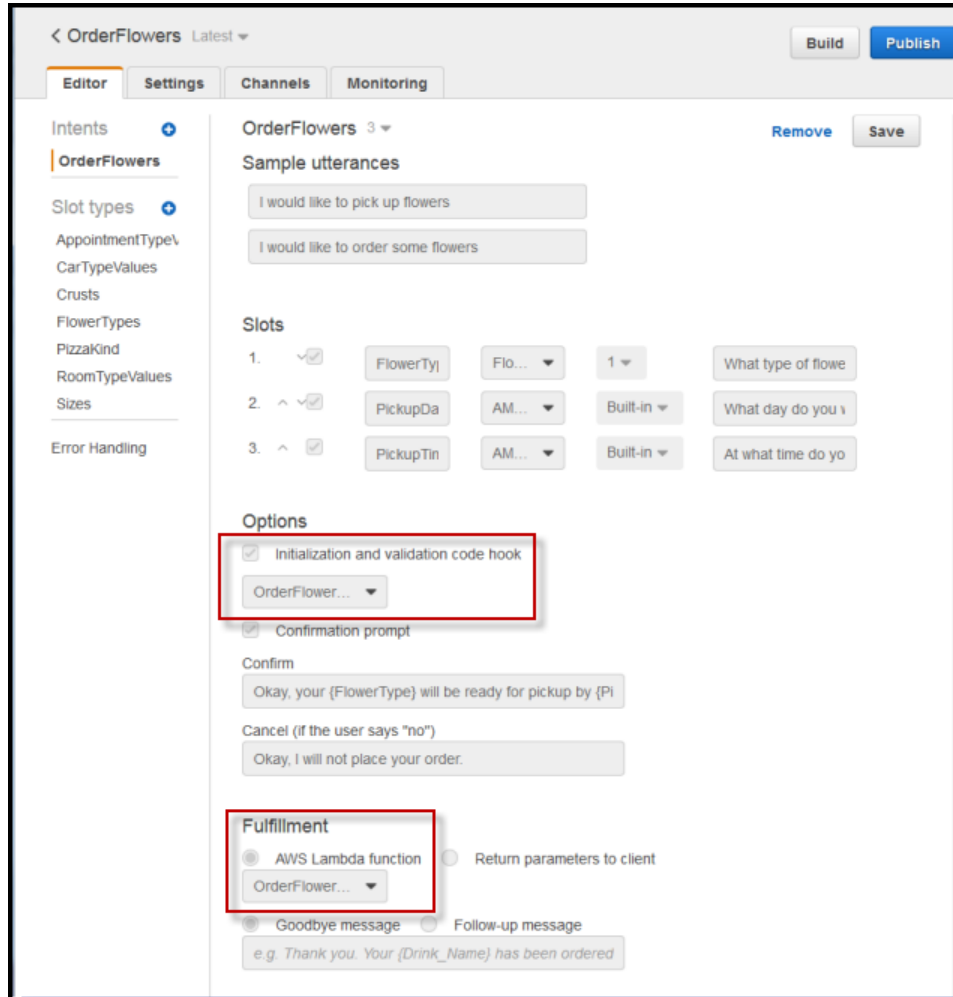
You can configure your Amazon Lex bot to invoke your Lambda function as a code hook. The code hook can serve multiple purposes:

- Customize user interaction based on prior knowledge about the user – For example, when Joe asks for the available pizza toppings, you can use prior knowledge of Joe's dietary preferences (assuming you have a back-end database) to display a subset of toppings.
- Validate the user's input – Suppose Jen wants to pick up flowers after hours. You can validate the time that Jen input and send an appropriate response.
- Fulfill the user's intent – After Joe provides all of the information for his pizza order, you can configure Amazon Lex to invoke a Lambda function that can place the order with a local pizzeria.

When you configure an intent, you can specify Lambda functions as code hooks in the following places:

- Dialog code hook (initialization/validation) – This Lambda function is invoked on each user input (assuming Amazon Lex understood the user intent correctly).
- Fulfillment code hook – This Lambda function is invoked after the user provides all of the slot data required to fulfill the intent.

In the Amazon Lex console, you choose the intent and set these code hooks, as shown in the following example screen shot:



You can also have one Lambda function do both. The event data that the Lambda function receives has a field that identifies the source of invocation (that is, a dialog or fulfillment code hook). You can use it to execute the appropriate portion of your code.

For more information, see [Using Lambda Functions](#) (p. 54).

## Managing Messages (Prompts and Statements)

### Topics

- [Types of Messages](#) (p. 8)
- [Contexts for Configuring Messages](#) (p. 9)
- [Supported Message Formats](#) (p. 12)
- [Response Cards](#) (p. 12)

When you create a bot, you can configure messages that you want the bot to send in the relevant context. Consider the following examples:

- You might configure your bot with the following clarification prompt:

```
I didn't understand, what would you like to do?
```

Then, Amazon Lex can send this message to the client if it doesn't understand the user's intent.

- Suppose you create a bot to support an intent called OrderPizza. For a pizza order, you want users to provide information such as pizza size, toppings, and crust type. For example, you can configure prompts such as the following:

```
What size pizza you want?  
What toppings you want on the pizza?  
Do you want thick or thin crust?
```

After Amazon Lex determines the user's intent to order pizza, it sends these messages to the client to elicit data from the user.

This section explains designing user interactions in your bot configuration.

## Types of Messages

You can classify the messages as follows:

- Prompt – A prompt expects a user response (typically in the form of a question).
- Statement – A statement does not expect any response.

The messages you configure can have dynamic components:

- Messages can use the following syntax to refer to slot values of the intent that Amazon Lex is currently aware of:

```
{SlotName}
```

- Messages can use the following syntax to refer to session attributes:

```
[AttributeName]
```

You can also have messages that include both slots and session attributes.

At runtime, Amazon Lex substitutes these references with actual values. For example, suppose you configure the following message in the OrderPizza intent of your bot:

```
"Hey [FirstName], your {PizzaTopping} pizza will arrive in [DeliveryTime]  
minutes"
```

This message refers to both slot (`PizzaTopping`) and session attributes (`FirstName` and `DeliveryTime`). At runtime, Amazon Lex replaces these placeholders with values and returns the following message to the client :

```
"Hey John, your cheese pizza will arrive in 30 minutes"
```

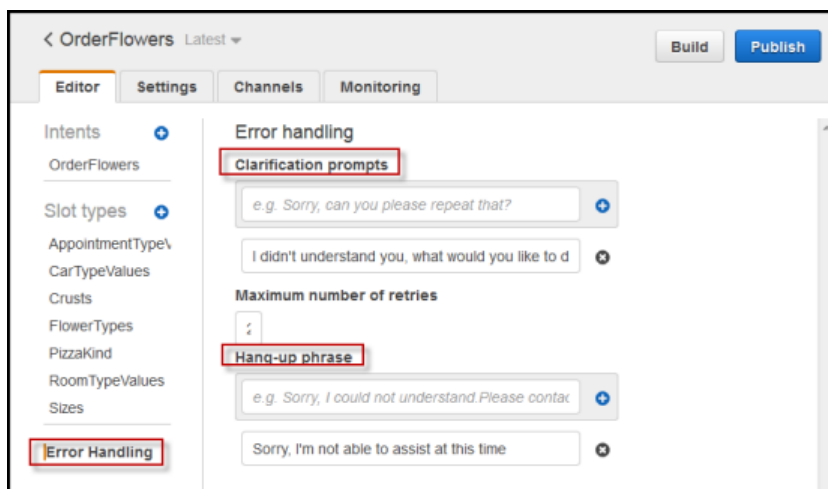
For information about session attributes, see the runtime API operations ([PostText \(p. 113\)](#), and [PostContent \(p. 106\)](#)). For an example, see [Example Bot: BookTrip \(p. 72\)](#).

If you add code hooks using Lambda functions in your intent configuration, you can create messages dynamically. Lambda functions can generate messages and return them to Amazon Lex to send to the user. By providing the messages while configuring your bot, you can eliminate the need to construct a prompt in code hooks.

## Contexts for Configuring Messages

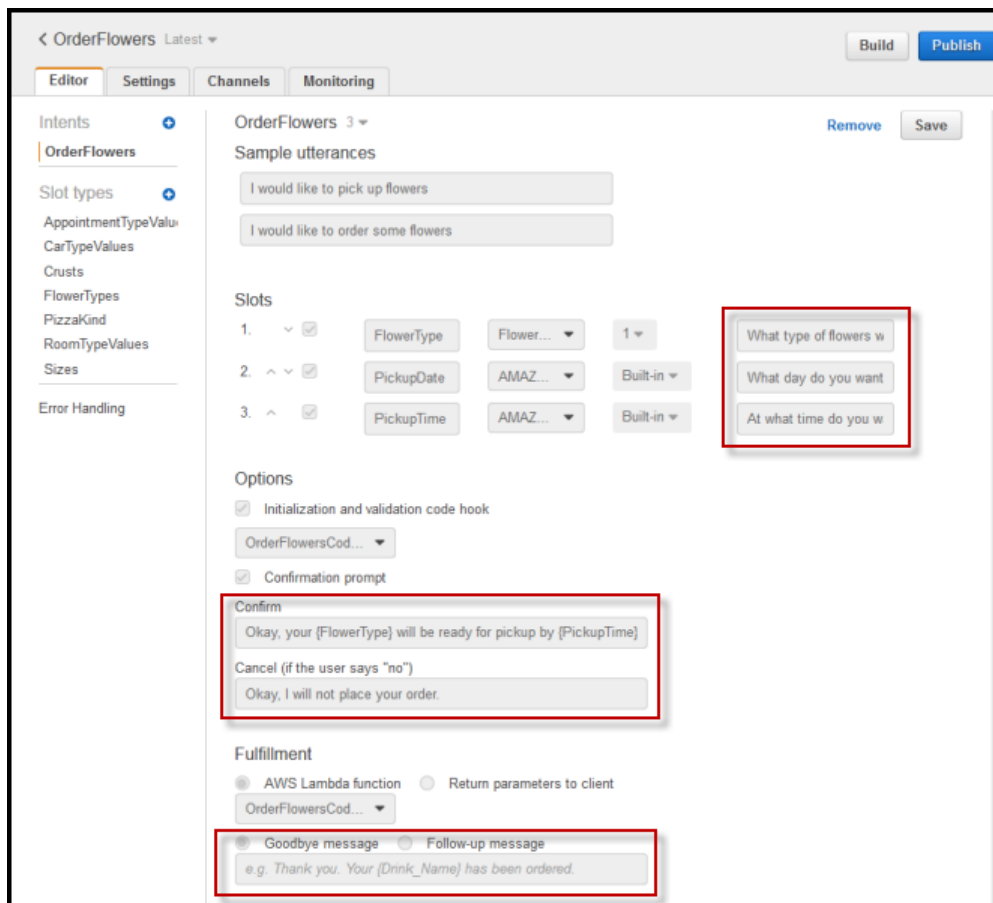
In your bot configuration, you can add messages in the following contexts. You can use the Amazon Lex console to configure your bot:

- Bot-level messages – You can configure your bot with clarification prompts and hang-up messages. At runtime, Amazon Lex uses the clarification prompts if it does not understand the user's intent. You can also configure the number of times that Amazon Lex requests clarification before hanging up with a hang-up message. You configure these bot-level messages in the **Error Handling** section in the Amazon Lex console, as shown in the following screen shot:



### Note

- If you have a code hook (that is, a Lambda function) configured for an intent, the Lambda function might return a response directing Amazon Lex to elicit user intent. If the Lambda function does not provide a message to convey to the user, then Amazon Lex uses the clarification prompt you configured.
- Amazon Lex uses the hang-up statement whenever the user does not respond with an appropriate answer for a prompt (including intent elicitation, slot elicitation, follow-up prompt, intent confirmation, etc.) after the maximum permissible attempts. You configure the maximum permissible attempts in the **Error Handling** section in the console.
- Intent-level messages – You can configure the intent-level messages such as confirmation prompts, cancel statements, goodbye messages (conclusion statements), and prompts (that Amazon Lex can use to elicit slot values), as shown in the following screen shot:



- Confirmation prompts and cancel statements – After a user provides all of the required data, Amazon Lex asks the user for confirmation using the specified message before fulfilling the intent. If the user replies with a "No" to a confirmation prompt, Amazon Lex returns the cancel statement to the client.
- Goodbye message (conclusion statement) or follow-up prompts – If you add a code hook (that is, a Lambda function) to fulfill the intent, you can configure one of these messages as backup messages. If the Lambda function succeeds but does not provide a message to send to the user, Amazon Lex sends the message that you configured.
- The following is an example of a conclusion statement (the example assumes that the application maintains the `DeliveryTime` session attribute):

```
"I have placed your order for pizza. It will arrive in [DeliveryTime] minutes."
```

- The following is an example of a follow up prompt:

```
"I have placed your order for pizza. Do you want me to do anything else?"
```

If you configure a follow-up prompt, you must also configure a cancel (rejection) statement. If the user's reply to a follow-up prompt is a "Yes," Amazon Lex recognizes the user's confirmation and also recognizes the user's intent (OrderDrink), and then follows up accordingly. For example:

```
"Yes, I also want to order a drink."
```

If the user says "No," Amazon Lex sends the cancel statement. For example:

```
"Alright. Ping me if you need anything else."
```

- Prompts to elicit value slot values – You must specify at least one prompt message for each of the required slots in an intent. At runtime, Amazon Lex uses one of these messages to prompt the user to provide value for this slot. For example, for a `cityName` slot, the following is a valid prompt:

```
"Which city would you like to fly to?"
```

#### Note

In your Lambda function that you create as code hook for an intent, you can override any of the messages that you configured at build-time.

You can configure more than one message for a specific context. At runtime, Amazon Lex picks the message with the maximum possible substitutions. For example, to elicit a value for crust type in OrderPizza intent, you can configure multiple messages, as shown following:

```
Hey [FirstName], what topping would you like for your {PizzaSize} pizza?  
Hey [FirstName], what topping would you like for your pizza?  
What topping would you like?  
Tell me the topping you would like on your pizza
```

Then, Amazon Lex uses the following order of selection:

- If both the session attribute (`FirstName`) and slot value (`PizzaSize`) are available, Amazon Lex uses the first prompt.
- If the session attribute (`FirstName`) is available but the user doesn't provide a slot value (`PizzaSize`) that's available, Amazon Lex uses the second prompt.
- If both the session attribute and the slot value are not available, Amazon Lex uses either the third or fourth prompt (random selection).

At runtime, Amazon Lex disregards messages with references to unresolved slot values. If all of the messages for a given context have unresolved references, Amazon Lex throws a `BadRequestException` error. Therefore, we recommend that you have at least one message without references.



## Supported Message Formats

Amazon Lex supports messages in the following formats: plain text and Speech Synthesis Markup Language (SSML).

If the output mode is text (that is, a client sends requests using the `PostText` API operation or the `PostContent` API operation with `Accept` HTTP header set to `text/plain; charset=utf-8`), Amazon Lex selects only plain text messages and SSML messages are disregarded.

### Note

- If you configure your bot with only SSML messages and a text client tries to communicate with your bot, Amazon Lex returns a `BadRequestException` error. We recommend that you give at least one `PlainText` message for each context.
- If `outputDialogMode` in the incoming event is text, you must return a `PlainText` message from your AWS Lambda function. For more information, see [Lambda Function Input Event and Response Format \(p. 54\)](#).

Amazon Lex also supports synthesizing audio from SSML. For more information, see [Using SSML](#).

## Response Cards

For each interaction context, you can also configure a response card. There can be many messages for a given context, but there can only be one response card.

You can use response cards with text-based clients, including messaging platforms such as Facebook Messenger.

When you configure a response card, Amazon Lex can include the response card (along with the corresponding message) in its response to the client. The client can then show the message and also display the response card. The user just chooses an option in the response card.

Each option in a response card has a value that you can configure to match the training data. For example, while building a taxi application, you can configure an option to read "Home" on the response card and set the value to an address "111 Maple Street, Seattle 98101". When the user selects this option, Amazon Lex receives the entire address as the input text. Thus, response cards simplify interaction for your users and increase your bot accuracy by reducing typographical errors.

You can define response cards statically (that is, at build-time) or dynamically (that is, at runtime) in a Lambda function. If both static and dynamic definition of response card exists dynamic response card definition takes precedence over static definitions.

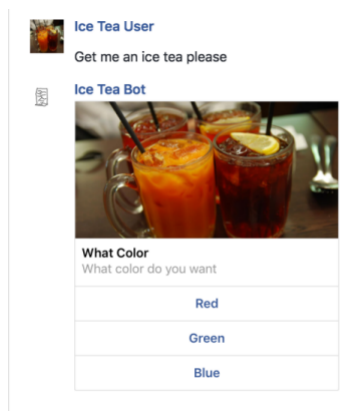
Amazon Lex understands these response cards and sends responses in a format understandable to client applications. For example, Amazon Lex transforms the response card to a generic template if the client is Facebook Messenger. For more information, see [Generic Template](#) on the Facebook website.

## Defining Response Cards Statically

You can define response cards at the bot build-time. You can use the Amazon Lex console to define the response card. Suppose you are creating a bot with some intent. Suppose one of the slots for this intent is color. When defining the color slot, you specify prompts, as shown in the following example screen shot:

In describing prompts you can optionally associate a response card and define details in the Amazon Lex console, as shown in the following example screen shot:

Now suppose you integrate your bot with Facebook Messenger, you see the following. User can then click the buttons to choose a color as shown in the following illustration:



You can refer to session attributes in response cards. At runtime, Amazon Lex substitutes these references with appropriate values from the session attributes.

## Defining Response Cards Dynamically

You can use a code hook (that is, a Lambda function) to generate response cards dynamically in response to user input and return the details as part of the `dialogAction` in the response. For more information, see [Response Format \(p. 57\)](#).

The following is an example Lambda function with a partial response showing the `responseCard` element that can generate a user experience similar to the one shown in the preceding section.

```
...
...
responseCard: {
  "version": 1,
  "contentType": "application/vnd.amazonaws.card.generic",
  "genericAttachments": [
    {
      "title": "What Color",
      "subtitle": "What color do you want",
      "imageUrl": "https://s3.amazonaws.com/lex-box/ice-tea.jpeg",
      "attachmentLinkUrl": "https://s3.amazonaws.com/lex-box/ice-tea.html",
      "buttons": [
        {
          "text": "Red",
          "value": "red"
        },
        {
          "text": "Green",
          "value": "green"
        },
        {
          "text": "Blue",
          "value": "blue"
        }
      ]
    }
  ]
}
...
...
```

## Managing Conversation Context

***This is prerelease documentation for a service in preview release. It is subject to change.***

This section explains the following:

- Session timeout – How long Amazon Lex maintains context information of an in-progress intent activity.
- Cross-intent information sharing – How you can share context information across intents.

### Session Timeout

For each Amazon Lex bot, you configure session timeout. Amazon Lex maintains context information of each in-progress intent activity for the duration of the session.

Suppose you create a bot (OrderShoes) that supports intents such as OrderShoesIntent and GetStatusIntent. When Amazon Lex detects that the user's intent is to order shoes, it asks the user for

slot data (for example, shoe size, color, brand, etc.). Suppose the user provides some of the slot data, but does not complete the shoe purchase. Amazon Lex remembers all of the prior intent information for the duration of the session (by default, session duration is five minutes). If the user returns within the session duration, the user can continue the conversation where it was left off, provide remaining slot data, and complete the purchase.

## Cross-Intent Information Sharing

Amazon Lex supports cross-intent information sharing. For example, suppose a user of the OrderShoes bot starts with ordering shoes. Amazon Lex engages in conversation with the user, elicits information (slot data such as shoe size, color, and brand), and the user successfully places a shoe order.

Then, the user switches the intent. Now the user wants to know the order status (GetOrderStatusIntent). Amazon Lex can ask the user for order information (slot data such as order number, order date, etc.). However, if the user switched the intent soon after ordering shoes, within the same session, you might design your bot to return the status of the latest order. That is, instead of asking the user for order information again, you can use session attributes to share cross-intent information (in this case, the order number) and fulfill the intent (that is, return the status of the last order the user placed).

You can use session attributes in this cross-intent information sharing scenario. Your application can save any information (such as order number in the preceding example). Every request that the client sends using `PostText` or `PostContent` runtime API includes the `sessionAttributes` field. Amazon Lex passes these session attributes to the code hook configured for the intent. If your application maintains the order number as a session attribute, it can be shared across intents.

For an example of cross-intent information sharing, see [Example Bot: BookTrip \(p. 72\)](#).

### Note

Amazon Lex does not store or log session attributes anywhere. Amazon Lex passes them between the client and code hook configured for the intent. If a message refers to a session attribute (using `[AttributeName]` notation), Amazon Lex substitutes the corresponding value while building the message. For more information, see [Managing Messages \(Prompts and Statements\) \(p. 7\)](#).

## Bot Deployment Options

Currently, Amazon Lex provides the following bot deployment options:

- [AWS Mobile SDK](#) – You can build mobile applications that communicate with Amazon Lex using the AWS Mobile SDKs.
- [Facebook Messenger](#) – You can integrate your Facebook Messenger page with your Amazon Lex bot so that end users on Facebook can communicate with the bot. In the current implementation, this integration supports only text input messages.

For examples, see [Deploying Amazon Lex Bots on Various Platforms \(p. 61\)](#).

## Built-in Intents and Slot Types

Topics

- [Built-in Intents](#) (p. 16)
- [Built-in Slots](#) (p. 16)

## Built-in Intents

Amazon Lex has a large built-in intent library. To create an intent from a built-in intent, select one of the existing built-in intents in the console, and add a custom name.

Then all the base intent configuration (such as sample utterances and slots) are available to the intent you are creating.

For a list of built-in intents, see [Built-in Intent Library](#) in the *Alexa Skills Kit*.

### Note

- Amazon Lex doesn't support `AMAZON.YesIntent` and `AMAZON.NoIntent`.
- In the current implementation, you can't add or remove sample utterances or slots from the base intent. Also, you cannot configure slots for built-in intents.

## Built-in Slots

Amazon Lex also has several built-in slot types (from the Alexa Skills Kit). You can create slots of these types in your intents. They eliminate the need to create enumeration values for commonly used slot data such as date, time and location. The built-in slot-types do not have versions. For a list of available built-in slot types, see [Slot Type Reference](#) in the *Alexa Skills Kit*.

### Note

Amazon Lex doesn't support `AMAZON.LITERAL` built-in slot type (it's deprecated in the *Alexa Skills Kit*).

# Getting Started

---

***This is prerelease documentation for a service in preview release. It is subject to change.***

This getting started section provides the following introductory exercises:

- Exercise 1 – Create an Amazon Lex bot using a blueprint. The bot blueprint provides all the necessary bot configuration. You only do minimum work to test the end-to-end setup.

In addition, you use the Lambda function blueprint provided by AWS Lambda to create a Lambda function (that is, a code hook), with predefined code that is compatible with your bot.

- Exercise 2 – Create a custom bot where you manually create and configure the bot. You also create a Lambda function as a code hook. Sample code is provided.
- Exercise 3 – Publish a bot and create a new version. As part of this exercise, you create an alias pointing to the bot version.

## Topics

- [Exercise 1: Create an Amazon Lex Bot Using a Blueprint \(p. 17\)](#)
- [Exercise 2: Create a Custom Amazon Lex Bot \(p. 39\)](#)
- [Exercise 3: Publish a Version and Create an Alias \(p. 48\)](#)

## Exercise 1: Create an Amazon Lex Bot Using a Blueprint

***This is prerelease documentation for a service in preview release. It is subject to change.***

In this exercise, you do the following:

- Create your first Amazon Lex bot and test it in the Amazon Lex console.

For this exercise, you use the **OrderFlowers** blueprint. For information about blueprints, see [Amazon Lex and AWS Lambda Blueprints \(p. 59\)](#).

- Create an AWS Lambda function and test it in the Lambda console. This Lambda function is the code hook for your bot. For this exercise, you use a Lambda blueprint (**lex-order-flowers-python**) provided in the AWS Lambda console to create your Lambda function. The blueprint code illustrates how you can use the same Lambda function to perform both initialization/validation and also fulfill the OrderFlowers intent.
- Update the bot to add the Lambda function as the code hook to fulfill the intent. Test the end-to-end experience.

The following sections explain the blueprints. You can review these settings now, and then revisit them after you have created and tested the end-to-end experience.

## Amazon Lex Bot: Blueprint Overview

You use the **OrderFlowers** blueprint to create an Amazon Lex bot. The bot is preconfigured as follows:

- **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?"

## AWS Lambda Function: Blueprint Summary

The Lambda function in this exercise performs both initialization/validation and fulfillment tasks. Therefore, after creating the Lambda function, you then update the intent configuration by specifying the same Lambda function as a code hook to handle both the initialization/validation and fulfillment tasks.

- As an initialization and validation code hook, the Lambda function performs a basic validation. For example, if the user provides a time for pickup that is outside of normal business hours, the Lambda function directs Amazon Lex to re-prompt the user for the time again.

- As part of fulfillment code hook, the Lambda function returns a summary message indicating that the flower order has been placed (that is, the intent is fulfilled).

Next Step

[Step 1: Prepare \(p. 19\)](#)

## Step 1: Prepare

In this section, you create two IAM roles:

- IAM role that Amazon Lex can assume. You grant this role necessary permissions so that Amazon Lex can invoke your Lambda function on your behalf.
- IAM role that AWS Lambda can assume. You grant this role necessary permissions for the CloudWatch actions so that AWS Lambda can write any logs that your Lambda function generates.

For more information about IAM roles, see [IAM Roles](#) in the *IAM User Guide*. Use the following procedures to create the IAM roles.

### To create the first IAM role (lex-exec-role)

You create this role for Amazon Lex to assume.

1. Sign in to the Identity and Access Management (IAM) console at <https://console.aws.amazon.com/iam/>.
2. Follow the steps in [Creating a Role to Delegate Permissions to an AWS Service](#) in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:

- In **Role Name**, use a name that is unique within your AWS account (for example, **lex-exec-role**).
- In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.

#### Note

In the current implementation, Amazon Lex service role is not available. Therefore, you first create a role using the AWS Lambda as the AWS service role. After you create the role, you update the trust policy and specify Amazon Lex as the service principal to assume the role.

- In **Attach Policy**, choose **Next Step** (that is, you create a role without any permissions). Create the role.
- Choose the role you created and update policies as follows:
  - In the **Permissions** tab, choose **Inline Policies**, and then attach the following custom policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "lambda:InvokeFunction",
        "polly:SynthesizeSpeech"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```



```
}  
]  
}
```

In this exercise you use a text-based client. Therefore, Amazon Lex does not require permission for the `polly:SynthesizeSpeech` action, but we include this so that you can use the same IAM role with a speech-based client that uses the `PostContent` runtime API operation.

- In the **Trust Relationships** tab, choose **Edit Trust Relationship**, and specify the Amazon Lex service principal ("`lex.amazonaws.com`"). The updated policy should look as shown:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Principal": {  
        "Service": "lex.amazonaws.com"  
      },  
      "Action": "sts:AssumeRole"  
    }  
  ]  
}
```

### To create the second IAM role (lambda-exec-role-for-lex-get-started)

You create this role for AWS Lambda to assume.

1. Sign in to the Identity and Access Management (IAM) console at <https://console.aws.amazon.com/iam/>.
2. Follow the steps in [Creating a Role to Delegate Permissions to an AWS Service](#) in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
  - In **Role Name**, use a name that is unique within your AWS account (for example, **lambda-exec-role-for-lex-get-started**).
  - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**. This grants the AWS Lambda service permissions to assume the role.
  - In **Attach Policy**, choose **Next Step** (that is, you create a role without any permissions). Create the role.
  - Choose the role you created. In the **Permissions** tab, choose **Inline Policies**, and then attach the following custom policy, which allows AWS Lambda permissions to write CloudWatch logs when it assumes the role.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "logs:CreateLogGroup",  
        "logs:CreateLogStream",  
        "logs:PutLogEvents"  
      ]  
    }  
  ],  
}
```

```
        "Resource": [
            "*"
        ]
    }
}
```

Next Step

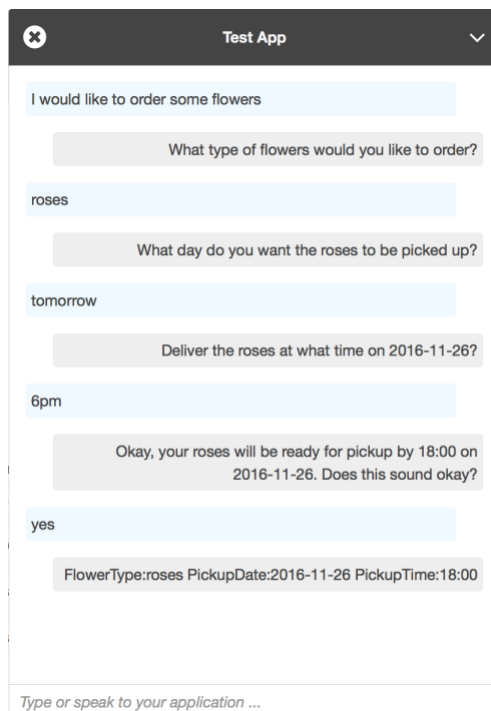
[Step 2: Create an Amazon Lex Bot \(p. 21\)](#)

## Step 2: Create an Amazon Lex Bot

In this section, you create an Amazon Lex bot (OrderFlowersBot).

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. On the **Bots** page, choose **Create**.
3. On the **Create your Lex bot** page, provide the following information and then choose **Create**.
  - Choose **OrderFlowers** blueprint.
  - Leave the default bot name (OrderFlowers).
  - Choose the IAM role (**lex-exec-role**)
4. The console makes the necessary requests to Amazon Lex to save the configuration and build the bot. The console then displays the **Test Bot** window.
5. Test the bot.

Use the following example text to engage in conversation with the bot to order flowers:



From this input, the bot infers the OrderFlowers intent and prompts for slot data. When you provide all the required slot data, the bot fulfills the intent (OrderFlowers) by simply returning all the information back to the client application (in this case, the console). The console simply shows the information in the test windows.

Note the following:

- In the statement "What day do you want the roses to be picked up?", the term "roses" appears because the prompt for the `pickupDate` slot is configured using substitutions, `{FlowerType}`. You can verify this in the console.
- The "Okay, your roses will be ready..." statement is what you configured as the confirmation prompt.
- The last statement ("FlowerType:roses...") is simply the slot data that is returned to the client, in this case the test window. In the next exercise, you use a Lambda function to fulfill the intent, in which case you get a message indicating the order is fulfilled.

Next Step

[Step 3 \(Optional\): Review the Details of Information Flow \(p. 22\)](#)

## Step 3 (Optional): Review the Details of Information Flow

This section explains the flow of information between the client and Amazon Lex for each user input. The client in the Amazon Lex console makes the [PostText \(p. 113\)](#) runtime API requests to Amazon Lex. The API topic describes details of the requests/responses shown in the following steps.

1. User: I would like to order some flowers
  - a. The client (console) sends the following [PostText \(p. 113\)](#) request to Amazon Lex:

```
POST /bot/OrderFlowers/alias/$LATEST/
user/4o9wwdhx6nlheferh6a73fujd3118f5w/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

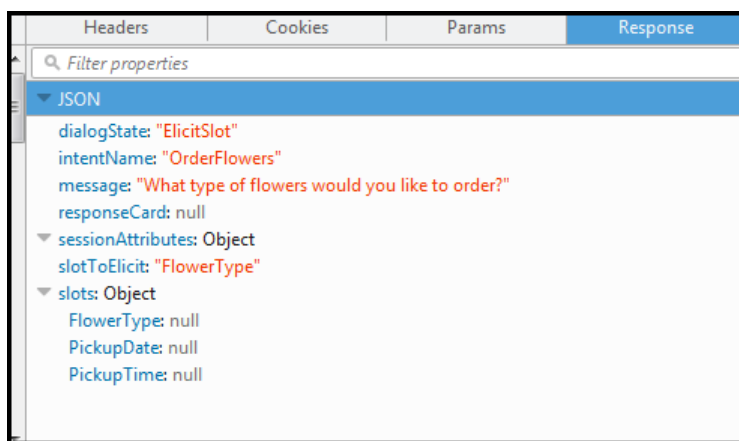
{
  "inputText": "I would like to order some flowers",
  "sessionAttributes": {}
}
```

Note that both the request URI and the body provide information to Amazon Lex:

- Request URI – Provides bot name (`OrderFlowers`), bot alias (`$LATEST`), and user name (a random string identifying the user). The trailing `text` indicates that it is a `PostText` API request (and not `PostContent`).

- Request body – Includes the user input (`inputText`) and empty `sessionAttributes`. When the client makes the first request, there are no session attributes. The Lambda function initiates them later.
- b. From the `inputText`, Amazon Lex detects the intent (`OrderFlowers`). This intent does not have any code hooks (that is, the Lambda functions) for initialization/validation of user input or fulfillment.

Amazon Lex chooses one of the intent's slots (`FlowerType`) to elicit the value. It also selects one of the value elicitation prompts for the slot (all part of the intent configuration) and then sends the following response back to the client. The client (that is, the console) displays the message in the response to the user.



The client displays the message in the response.

2. User: roses
- a. The client (console) sends the following `PostText` (p. 113) request to Amazon Lex:

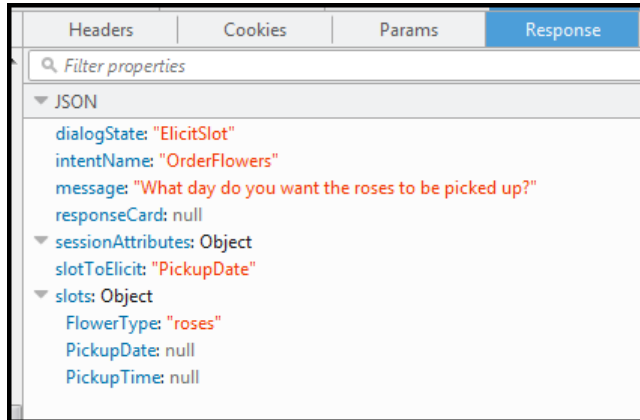
```
POST /bot/OrderFlowers/alias/$LATEST/
user/4o9wwdhx6nlheferh6a73fujd3118f5w/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "roses",
  "sessionAttributes": {}
}
```

Note that the `inputText` in the request body provides user input. The `sessionAttributes` remains empty.

- b. Amazon Lex first interprets the `inputText` in the context of the current intent (the service remembers that it had asked the specific user for information about `FlowerType` slot). Amazon Lex first updates the slot value for the current intent and chooses another slot (`PickupDate`) along with one of its prompt messages (What day do you want the roses to be picked up?) for the slot.

Then, Amazon Lex returns the following response:



The client displays the message in the response.

3. User: tomorrow
  - a. The client (console) sends the following `PostText` (p. 113) request to Amazon Lex:

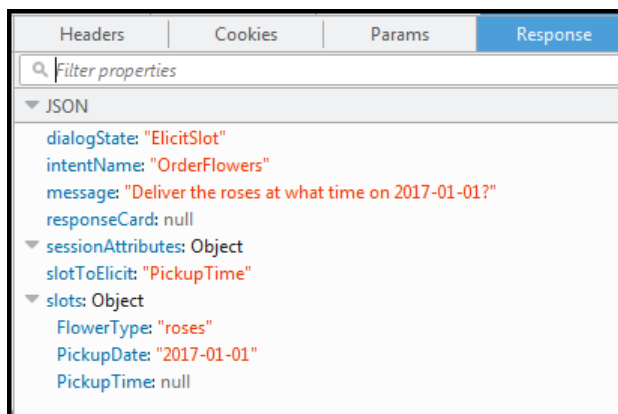
```
POST /bot/OrderFlowers/alias/$LATEST/
user/4o9wwdhx6nlheferh6a73fujd3118f5w/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "tomorrow",
  "sessionAttributes": {}
}
```

Note that the `inputText` in the request body provides user input. The `sessionAttributes` remains empty.

- b. Amazon Lex first interprets the `inputText` in the context of the current intent (the service remembers that it had asked the specific user for information about `PickupDate` slot). Amazon Lex updates the slot (`PickupDate`) value for the current intent. It chooses another slot to elicit value for (`PickupTime`). It returns one of the value elicitation prompts (Deliver the roses at what time on 2017-01-01?) to the client.

Amazon Lex then returns the following response:



The client displays the message in the response.

4. User: 6 pm
  - a. The client (console) sends the following [PostText](#) (p. 113) request to Amazon Lex:

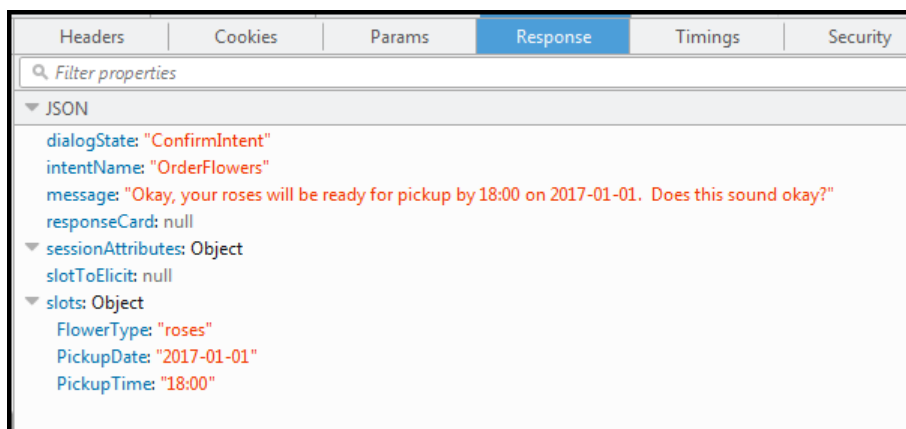
```
POST /bot/OrderFlowers/alias/$LATEST/
user/4o9wwdhx6nlheferh6a73fujd3118f5w/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "6 pm",
  "sessionAttributes": {}
}
```

Note that the `inputText` in the request body provides user input. The `sessionAttributes` remains empty.

- b. Amazon Lex first interprets the `inputText` in the context of the current intent (the service remembers that it had asked the specific user for information about `PickupTime` slot). Amazon Lex first updates the slot value for the current intent. Now Amazon Lex detects that it has information for all the slots.

The `OrderFlowers` intent is configured with a confirmation message. Therefore, Amazon Lex needs an explicit confirmation from the user before it can proceed to fulfill the intent. Amazon Lex sends the following message to the client requesting confirmation before ordering the flowers:



The client displays the message in the response.

5. User: Yes
  - a. The client (console) sends the following [PostText](#) (p. 113) request to Amazon Lex:

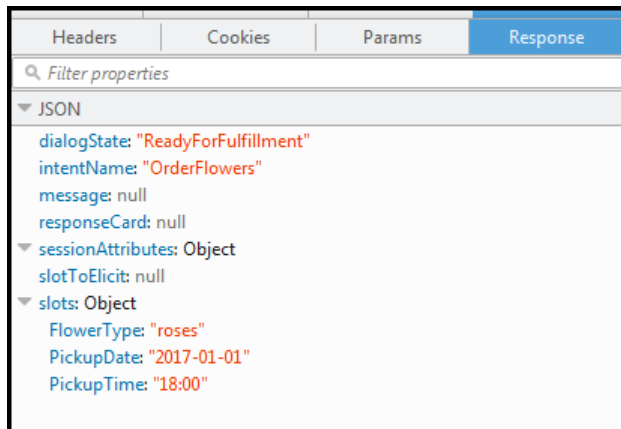
```
POST /bot/OrderFlowers/alias/$LATEST/
user/4o9wwdhx6nlheferh6a73fujd3118f5w/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
```

```
"inputText": "Yes",  
"sessionAttributes": {}  
}
```

Note that the `inputText` in the request body provides user input. The `sessionAttributes` remains empty.

- b. Amazon Lex interprets the `inputText` in the context of confirming the current intent. It understands that the user want to proceed with the order. `OrderFlowers` intent is configured with `ReturnIntent` as the fulfillment activity (there is no code hook, a Lambda function to fulfill the intent). Therefore, Amazon Lex returns the slot data back to the client.



Note that Amazon Lex set the `dialogState` to `ReadyForFulfillment`. The client can then fulfill the intent.

6. Now you need to test the bot again. To do that, you must choose the **Clear** link in the console to establish a new (user) context. Now as you provide data for the order flowers intent, try to provide invalid data. For example:
  - Jasmine as the flower type (it is not one of the supported flower types).
  - Yesterday as the day when you want to pick up the flowers.

Notice that the bot accepts these values because you don't have any code to initialize/validate user data. In the next section, you add a Lambda function to do this. Note the following about the Lambda function:

- The Lambda function validates slot data after every user input. It fulfills the intent at the end (that is, the bot processes the flowers order and returns a message to the user instead of simply returning slot data back to the client). For more information, see [Using Lambda Functions \(p. 54\)](#).
- The Lambda function also sets the session attributes. For more information about session attributes, see [PostText \(p. 113\)](#).

After you complete the Getting Started section, you can do the additional exercises ([Additional Examples: Creating Amazon Lex Bots \(p. 66\)](#)). [Example Bot: BookTrip \(p. 72\)](#) uses session attributes to share cross-intent information to engage in a dynamic conversation with the user.

Next Step

[Step 4: Create a Lambda Function \(p. 27\)](#)

## Step 4: Create a Lambda Function

In this section, you create a Lambda function (using the **lex-order-flowers-python** blueprint) and perform test invocation using sample event data in the AWS Lambda console. This Lambda function is written in Node.js.

In the next section you go back to the Amazon Lex console and add the Lambda function as the code hook to fulfill the OrderFlowers intent (in your OrderFlowersBot you created in the preceding section).

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose **Create a Lambda function**.
3. On **Select blueprint**, type **lex** to find the blueprint, choose the `lex-order-flowers-python` blueprint.

Lambda function blueprints are provided in both Node.js and Python. For this exercise, you use the Python-based blueprint.

4. Choose **Next** on the **Configure Triggers** page.
5. On the **Configure function** page, do the following, and then choose **Next**.
  - Type a Lambda function name (`OrderFlowersCodeHook`).
  - For the IAM role, choose the **Choose an existing role** and then select `lambda-exec-role-for-lex-get-started` from the **Existing role list**.
  - Leave other default values.
6. On the **Review** page, choose **Create function**.
7. Test the Lambda function.
  - a. Choose **Actions, Configure test event**.
  - b. Choose **Lex-Order Flowers (preview)** from the **Sample event template** list. This sample event matches the Amazon Lex request/response model (see [Using Lambda Functions \(p. 54\)](#)).
  - c. Choose **Save and test**.
  - d. Verify that the Lambda function successfully executed. The response in this case matches the Amazon Lex response model.

Next Step

[Step 5: Update Intent Configuration: Add the Lambda Function as Code Hook \(p. 27\)](#)

## Step 5: Update Intent Configuration: Add the Lambda Function as Code Hook

In this section, you update the configuration of the OrderFlowers intent to use the Lambda function as follows:

- First use the Lambda function as a code hook to perform fulfillment of the OrderFlowers intent. You test the bot and verify that you received a fulfillment message from the Lambda function. Amazon Lex invokes the Lambda function only after you provide data for all the required slots for ordering flowers.



- Configure the same Lambda function as a code hook to perform initialization and validation. You test and verify that the Lambda function performs validation (as you provide slot data).
1. In the Amazon Lex console, select the **OrderFlowers** bot. The console shows the **OrderFlowers** intent. Make sure that the intent version is set to \$LATEST because this is the only version that we can modify.
  2. Add the Lambda function as the fulfillment code hook and test it.

- a. In the Editor, choose **AWS Lambda function as Fulfillment**, and select the Lambda function that you created in the preceding step (OrderFlowersCodeHook).

Note that you are configuring this Lambda function as a code hook to fulfill the intent. Amazon Lex invokes this function only after it has all the necessary slot data from the user to fulfill the intent.

- b. Specify a **Goodbye message**.
- c. Choose **Save** and then choose **Build**.
- d. Test the bot using the same previous conversation.

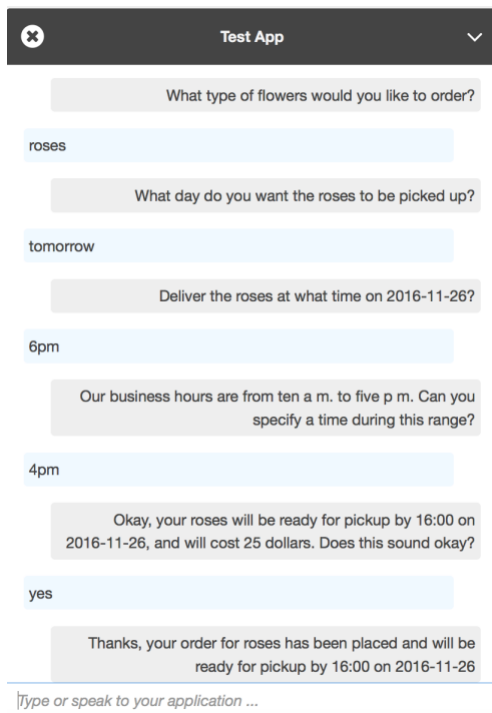
Note that the last statement "Thanks, your order for roses...." is a response from the Lambda function that you configured as a code hook. In the preceding section, there was no Lambda function. Now you are using a Lambda function to actually fulfill the OrderFlowers intent.

3. Add the Lambda function as an initialization and validation code hook, and test.

The sample Lambda function code that you are using can do both (perform user input validation and fulfillment). The input event the Lambda function receives has a field (`invocationSource`) that the code uses to determine what portion of the code to execute. For more information, see [Lambda Function Input Event and Response Format \(p. 54\)](#).

- a. Make sure the intent (OrderFlowers) version \$LATEST because that is the only version you can update.
- b. In the Editor, choose **Initialization and validation** in **Options**.
- c. Again, select the same Lambda function.
- d. Choose **Save** and then choose **Build**.
- e. Test the bot.

You completed the setup. You are now ready to converse with Amazon Lex as follows. To test the validation portion, choose time 6 PM, and your Lambda function returns a response ("Our business hours are from ten AM to five PM."), and prompts you again. After you provide all the valid slot data, the Lambda function fulfills the order.



#### Next Step

[Step 6 \(Optional\): Review the Details of Information Flow \(p. 29\)](#)

## Step 6 (Optional): Review the Details of Information Flow

This section explains the flow of information between the client and Amazon Lex for each user input. The client in the Amazon Lex console uses the [PostText \(p. 113\)](#) runtime API to send requests to Amazon Lex. The API topic describes details of the requests/responses shown in the following steps:

1. User: I would like to order some flowers
  - a. The client (console) sends the following [PostText \(p. 113\)](#) request to Amazon Lex:

```
POST /bot/OrderFlowers/alias/$LATEST/
user/ignw84y6seypre4xly5rimopuri2xwnd/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "I would like to order some flowers",
  "sessionAttributes": {}
}
```

Note that both the request URI and the body provides information to Amazon Lex:

- Request URI – Provides bot name (OrderFlowers), bot alias (\$LATEST), and user name (a random string identifying the user). The trailing `text` indicates that it is a `PostText` API request (and not `PostContent`).
  - Request body – Includes the user input (`inputText`) and empty `sessionAttributes`. When the client makes the first request, there are no session attributes. Lambda function initiates them later.
- b. From the `inputText`, Amazon Lex detects the intent (OrderFlowers). This intent is configured with a Lambda function as code hook for user data initialization/validation. Therefore, Amazon Lex invokes that Lambda function by passing the following information as event data:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "ignw84y6seypre4xly5rimopuri2xwnd",
  "sessionAttributes": {},
  "bot": {
    "name": "OrderFlowers",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "OrderFlowers",
    "slots": {
      "PickupTime": null,
      "FlowerType": null,
      "PickupDate": null
    }
  },
  "confirmationStatus": "None"
}
```

For more information, see [Input Event Format \(p. 54\)](#).

Note that, in addition to the information client sent, Amazon Lex also includes the following additional data.

- `messageVersion` – Currently Amazon Lex supports only the 1.0 version.
  - `invocationSource` – Indicates the purpose of Lambda function invocation. In this case, it is to perform user data initialization and validation (at this time Amazon Lex knows that the user has not provided all the slot data to fulfill the intent).
  - `currentIntent` information, which has all the slot values set to null.
- c. At this time, all the slot values are null. There is nothing for the Lambda function to validate. The Lambda function returns the following response to Amazon Lex:

```
{
  "sessionAttributes": {},
  "dialogAction": {
    "type": "Delegate",
    "slots": {
      "PickupTime": null,
      "FlowerType": null,

```

```
        "PickupDate": null
      }
    }
  }
```

For information about the response format, see [Response Format \(p. 57\)](#).

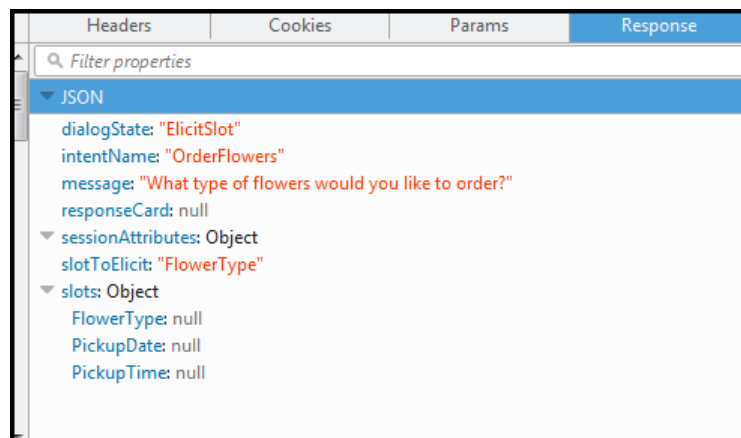
Note the following:

- `dialogAction.type` – By setting this value to `Delegate`, Lambda function delegates the responsibility of deciding the next course of action to Amazon Lex.

**Note**

If Lambda function detects anything in the user data validation, it instructs Amazon Lex what to do next, as shown in the next few steps.

- d. According to the `dialogAction.type`, Amazon Lex decides the next course of action. Because none of the slots are filled, it decides to elicit the value for the `FlowerType` slot. It selects one of the value elicitation prompts ("What type of flowers would you like to order?") for this slot, as per the intent configuration, and sends the following response back to the client:



The client displays the message in the response.

2. User: roses

- a. The client (console) sends the following [PostText \(p. 113\)](#) request to Amazon Lex:

```
POST /bot/OrderFlowers/alias/$LATEST/
user/ignw84y6seypre4xly5rimopuri2xwnd/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "roses",
  "sessionAttributes": {}
}
```

Note that, in the request body the `inputText` provides user input. The `sessionAttributes` remains empty.

- b. Amazon Lex first interprets the `inputText` in the context of the current intent (the service remembers that it had asked the specific user for information about `FlowerType` slot). It updates the slot value in the current intent and invokes the Lambda function with the following event data:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "ignw84y6seypre4xly5rimopuri2xwnd",
  "sessionAttributes": {},
  "bot": {
    "name": "OrderFlowers",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "OrderFlowers",
    "slots": {
      "PickupTime": null,
      "FlowerType": "roses",
      "PickupDate": null
    },
    "confirmationStatus": "None"
  }
}
```

Note the following:

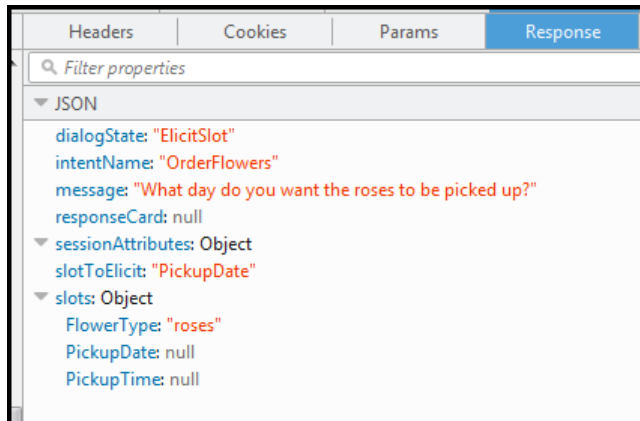
- `invocationSource` – continues to be `DialogCodeHook` (we are simply validating user data).
  - `currentIntent.slots` – Amazon Lex has updated the `FlowerType` slot to `roses`.
- c. According to the `invocationSource` value of `DialogCodeHook`, the Lambda function performs user data validation. It recognizes `roses` as a valid slot value (and sets `Price` as a session attribute) and returns the following response to Amazon Lex.

```
{
  "sessionAttributes": {
    "Price": 25
  },
  "dialogAction": {
    "type": "Delegate",
    "slots": {
      "PickupTime": null,
      "FlowerType": "roses",
      "PickupDate": null
    }
  }
}
```

Note the following:

- `sessionAttributes` – Lambda function has added `Price` (of the roses) as a session attribute.

- `dialogAction.type` – is set to `Delegate`. The user data was valid so the Lambda function directs Amazon Lex to choose the next course of action.
- d. According to the `dialogAction.type`, Amazon Lex chooses the next course of action. Amazon Lex knows it needs more slot data so it picks the next unfilled slot (`PickupDate`) with the highest priority according to the intent configuration. Amazon Lex selects one of the value elicitation prompt messages ("What day do you want the roses to be picked up?") for this slot according to the intent configuration and then sends the following response back to the client:



The client simply displays the message in the response – "What day do you want the roses to be picked up?"

### 3. User: tomorrow

- a. The client (console) sends the following [PostText](#) (p. 113) request to Amazon Lex:

```
POST /bot/OrderFlowers/alias/$LATEST/  
user/ignw84y6seypre4xly5rimopuri2xwnd/text  
"Content-Type": "application/json"  
"Content-Encoding": "amz-1.0"  
  
{  
  
  "inputText": "tomorrow",  
  "sessionAttributes": {  
    "Price": "25"  
  }  
}
```

Note that the `inputText` in the request body provides user input and the client passes the session attributes back to the service.

- b. Amazon Lex remembers the context—that it was eliciting data for the `PickupDate` slot. In this context, it knows the `inputText` value is for the `PickupDate` slot. Amazon Lex then invokes the Lambda function by sending the following event:

```
{  
  "messageVersion": "1.0",  
  "invocationSource": "DialogCodeHook",
```

```
"userId": "ignw84y6seypre4xly5rimopuri2xwnd",
"sessionAttributes": {
  "Price": "25"
},
"bot": {
  "name": "OrderFlowersCustomWithRespCard",
  "alias": null,
  "version": "$LATEST"
},
"outputDialogMode": "Text",
"currentIntent": {
  "name": "OrderFlowers",
  "slots": {
    "PickupTime": null,
    "FlowerType": "roses",
    "PickupDate": "2017-01-05"
  },
  "confirmationStatus": "None"
}
}
```

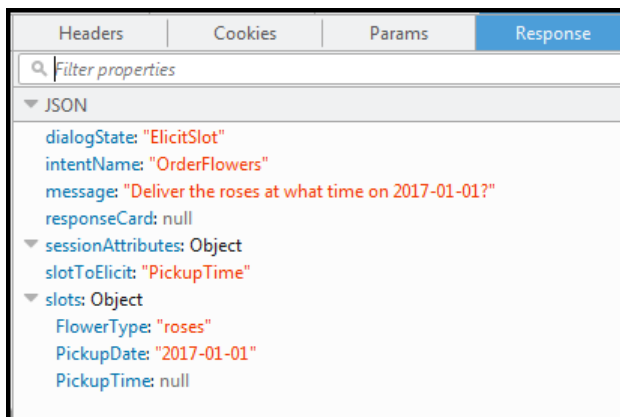
Note that Amazon Lex has updated the `currentIntent.slots` by setting the `PickupDate` value. Also note that the services passes the `sessionAttributes` as it is to the Lambda function.

- c. As per `invocationSource` value of `DialogCodeHook`, the Lambda function performs user data validation. It recognizes `PickupDate` slot value is valid and returns the following response to Amazon Lex:

```
{
  "sessionAttributes": {
    "Price": 25
  },
  "dialogAction": {
    "type": "Delegate",
    "slots": {
      "PickupTime": null,
      "FlowerType": "roses",
      "PickupDate": "2017-01-05"
    }
  }
}
```

Note the following:

- `sessionAttributes` – No change.
  - `dialogAction.type` – is set to `Delegate`. The user data was valid, and the Lambda function directs Amazon Lex to choose the next course of action.
- d. According to the `dialogAction.type`, Amazon Lex chooses the next course of action. Amazon Lex knows it needs more slot data so it picks the next unfilled slot (`PickupTime`) with the highest priority according to the intent configuration. Amazon Lex selects one of the prompt messages ("Deliver the roses at what time on 2017-01-01?") for this slot according to the intent configuration and sends the following response back to the client:



The client displays the message in the response – "Deliver the roses at what time on 2017-01-01?"

4. User: 4 pm
  - a. The client (console) sends the following [PostText](#) (p. 113) request to Amazon Lex:

```
POST /bot/OrderFlowers/alias/$LATEST/
user/ignw84y6seypre4xly5rimopuri2xwnd/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "4 pm",
  "sessionAttributes": {
    "Price": "25"
  }
}
```

Note that the `inputText` in the request body provides user input. The client passes the `sessionAttributes` in the request.

- b. Amazon Lex knows context, that it was eliciting data for the `PickupTime` slot. In this context, it knows that the `inputText` value is for the `PickupTime` slot. Amazon Lex then invokes the Lambda function by sending the following event:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "ignw84y6seypre4xly5rimopuri2xwnd",
  "sessionAttributes": {
    "Price": "25"
  },
  "bot": {
    "name": "OrderFlowersCustomWithRespCard",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "OrderFlowers",
    "slots": {
```



```
        "PickupTime": "16:00",  
        "FlowerType": "roses",  
        "PickupDate": "2017-01-05"  
    },  
    "confirmationStatus": "None"  
  }  
}
```

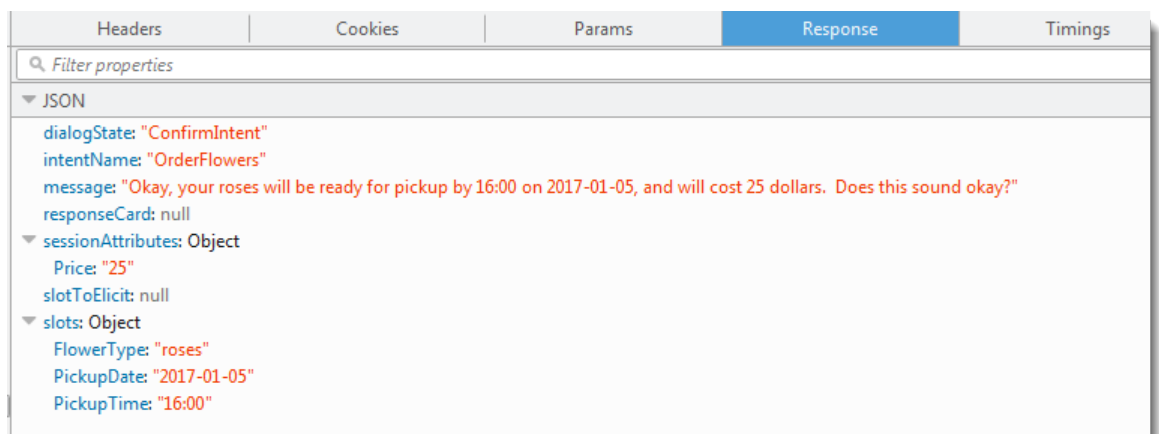
Note that Amazon Lex has updated the `currentIntent.slots` by setting the `PickupTime` value.

- c. According to the `invocationSource` value of `DialogCodeHook`, the Lambda function performs user data validation. It recognizes `PickupDate` slot value is valid and returns the following response to Amazon Lex.

```
{  
  "sessionAttributes": {  
    "Price": 25  
  },  
  "dialogAction": {  
    "type": "Delegate",  
    "slots": {  
      "PickupTime": "16:00",  
      "FlowerType": "roses",  
      "PickupDate": "2017-01-05"  
    }  
  }  
}
```

Note the following:

- `sessionAttributes` – No change in session attribute.
  - `dialogAction.type` – is set to `Delegate`. The user data was valid so the Lambda function directs Amazon Lex to choose the next course of action.
- d. At this time Amazon Lex knows it has all the slot data. This intent is configured with a confirmation prompt. Therefore, Amazon Lex sends the following response to the user asking for confirmation before fulfilling the intent:



The client simply displays the message in the response and waits for the user response.

5. User: Yes

- a. The client (console) sends the following [PostText \(p. 113\)](#) request to Amazon Lex:

```
POST /bot/OrderFlowers/alias/$LATEST/
user/ignw84y6seypre4xly5rimopuri2xwnd/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "yes",
  "sessionAttributes": {
    "Price": "25"
  }
}
```

- b. Amazon Lex interprets the `inputText` in the context of confirming the current intent. Amazon Lex understands that the user wants to proceed with the order. This time Amazon Lex invokes the Lambda function to fulfill the intent by sending the following event, which sets the `invocationSource` to `FulfillmentCodeHook` in the event it sends to the Lambda function. Amazon Lex also sets the `confirmationStatus` to `Confirmed`.

```
{
  "messageVersion": "1.0",
  "invocationSource": "FulfillmentCodeHook",
  "userId": "ignw84y6seypre4xly5rimopuri2xwnd",
  "sessionAttributes": {
    "Price": "25"
  },
  "bot": {
    "name": "OrderFlowersCustomWithRespCard",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "OrderFlowers",
    "slots": {
      "PickupTime": "16:00",
      "FlowerType": "roses",
      "PickupDate": "2017-01-05"
    }
  },
  "confirmationStatus": "Confirmed"
}
```

Note the following:

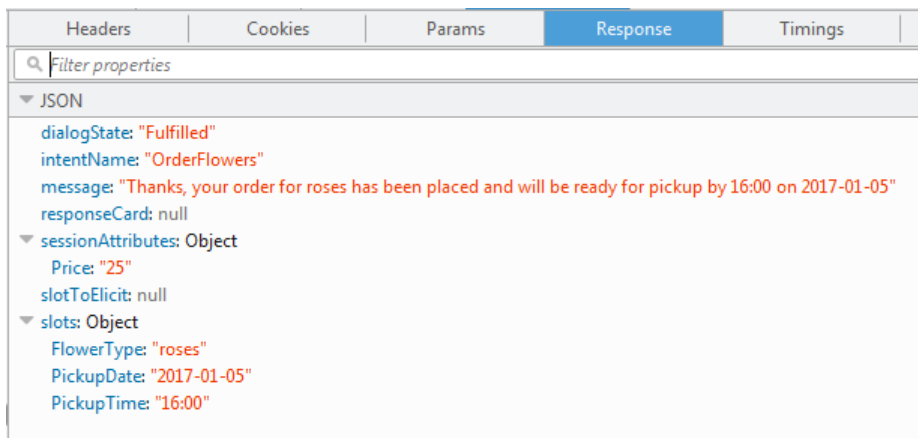
- `invocationSource` – This time Amazon Lex set this value to `FulfillmentCodeHook`, directing the Lambda function to fulfill the intent.
  - `confirmationStatus` – is set to `Confirmed`.
- c. This time, the Lambda function fulfills the `OrderFlowers` intent, and returns the following response:

```
{
  "sessionAttributes": {
    "Price": "25"
  },
  "dialogAction": {
    "type": "Close",
    "fulfillmentState": "Fulfilled",
    "message": {
      "contentType": "PlainText",
      "content": "Thanks, your order for roses has been placed
and will be ready for pickup by 16:00 on 2017-01-05"
    }
  }
}
```

Note the following:

- Sets the `dialogAction.type` – The Lambda function sets this value to `Close`, directing Amazon Lex to not expect a user response.
  - `dialogAction.fulfillmentState` – is set to `Fulfilled` and includes an appropriate message to convey to the user.
- d. Amazon Lex reviews the `fulfillmentState` and sends the following response back to the client.

Amazon Lex then returns the following to the client:



Note that,

- `dialogState` – Amazon Lex sets this value to `fulfilled`.
- `message` – is the same message that the Lambda function provided.

The client displays the message.

6. Now test the bot again. You must choose the **Clear** link in the test window to establish a new (user) context. Now try to provide invalid slot data for the `OrderFlowers` intent. This time the Lambda function performs the data validation, resets invalid slot data value to null, and asks Amazon Lex to prompt the user for valid data. For example, try the following:

- Jasmine as the flower type (it is not one of the supported flower types).
- Yesterday as the day when you want to pick up the flowers.
- After placing your order, choose to enter flower type instead of replying "yes" to confirm the order. In response, the Lambda function updates the `Price` in the session attribute, keeping a running total of flower orders.

The Lambda function also performs the fulfillment activity.

## Exercise 2: Create a Custom Amazon Lex Bot

In this exercise, you create a custom bot (OrderPizzaBot). You do all the necessary configuration including adding a custom intent (OrderPizza), defining custom slot types, and defining the slots required to fulfill a pizza order (pizza crust, size, and so on). For more information about slot types and slots, see [Amazon Lex: How It Works \(p. 3\)](#).

First, you create a bot and test it in the Amazon Lex console without any code hook. Then, you configure the intent by adding a code hook, which is a Lambda function to fulfill the intent. Lambda function code is provided for this exercise.

### Topics

- [Step 1: Prepare \(p. 39\)](#)
- [Step 2: Create an Amazon Lex Bot \(p. 42\)](#)
- [Step 3: Create Slot Types \(p. 43\)](#)
- [Step 4: Create an Intent \(p. 44\)](#)
- [Step 5: Configure Error Handling \(p. 46\)](#)
- [Step 6: Build and Test the Bot \(p. 47\)](#)

## Step 1: Prepare

Follow the sections to first create two IAM roles and a Lambda function.

### Topics

- [Step 1.1: Create IAM Roles \(p. 39\)](#)
- [Step 1.2: Create a Lambda Function \(p. 39\)](#)

## Step 1.1: Create IAM Roles

You can use the IAM roles that you created for Getting Started Exercise 1 for this exercise. If you created these roles already, go to the next section. For instructions to create these roles, see [Step 1: Prepare \(p. 19\)](#).

## Step 1.2: Create a Lambda Function

In this section you create a Lambda function. You specify this Lambda function in your Amazon Lex bot, which you create in the next section, to fulfill a pizza order.

## Create a Lambda Function

In this section you create a Lambda function.

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose the US East (N. Virginia) Region (us-east-1).
3. Choose **Create a Lambda function**.
4. On the **Select blueprint** page, choose **Blank function**.

You are creating a Lambda function using custom code provided to you in this exercise, therefore you choose the blank function option.

5. On the **Configure triggers** page, choose **Next**.
6. On the **Configure function** page, do the following:
  - a. Type name (`PizzaOrderProcessor`) and choose the Node.js.4.3 as the **runtime**.
  - b. In the **Lambda function code** section, choose **Edit code inline**, and then copy the following Lambda function code and paste it in the window.

```
'use strict';

// Close dialog with the customer, reporting fulfillmentState of
// Failed or Fulfilled ("Thanks, your pizza will arrive in 20 minutes")
function close(sessionAttributes, fulfillmentState, message) {
    return {
        sessionAttributes,
        dialogAction: {
            type: 'Close',
            fulfillmentState,
            message,
        },
    };
}

// ----- Events -----

function dispatch(intentRequest, callback) {
    console.log('request received for userId=${intentRequest.userId},
intentName=${intentRequest.currentIntent.intentName}');
    const sessionAttributes = intentRequest.sessionAttributes;
    const slots = intentRequest.currentIntent.slots;
    const crust = slots.crust;
    const size = slots.size;
    const pizzaKind = slots.pizzaKind;

    callback(close(sessionAttributes, 'Fulfilled',
    {'contentType': 'PlainText', 'content': `Okay, I have ordered your
    ${size} ${pizzaKind} pizza on ${crust} crust`}));
}

// ----- Main handler -----

// Route the incoming request based on intent.
```

```
// The JSON body of the request is provided in the event slot.
exports.handler = (event, context, callback) => {
  try {
    dispatch(event,
      (response) => {
        callback(null, response);
      });
  } catch (err) {
    callback(err);
  }
};
```

- c. In the **Lambda function handler and role** section, choose **Choose an existing role** for the **Role**, and then select the IAM role `lambda-exec-role-for-lex-get-started` that you created in the preceding section.
- d. Choose **Next**.

7. On the **Review** page, choose **Create function**.

## Test the Lambda Function in the Lambda Console Using Sample Event Data

You can test the Lambda function in the console by using sample event data to manually invoke the function. Follow the steps to test the Lambda function:

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. On the Lambda function page, choose **Actions**, and then choose **Configure test event**.
3. Choose the Lambda function.
4. On the **Input test event** page, copy the following Amazon Lex event into the window.

```
{
  "messageVersion": "1.0",
  "invocationSource": "FulfillmentCodeHook",
  "userId": "user-1",
  "sessionAttributes": {},
  "bot": {
    "name": "PizzaOrderingApp",
    "alias": "$LATEST",
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "OrderPizza",
    "slots": {
      "size": "large",
      "pizzaKind": "meat",
      "crust": "thin"
    },
    "confirmationStatus": "None"
  }
}
```

5. Choose **Save and test**.

AWS Lambda executes your Lambda function. The following output appears in the **Execution result** pane.

```
{
  "sessionAttributes": {},
  "dialogAction": {
    "type": "Close",
    "fulfillmentState": "Fulfilled",
    "message": {
      "contentType": "PlainText",
      "content": "Okay, I have ordered your large meat pizza on thin crust"
    }
  }
}
```

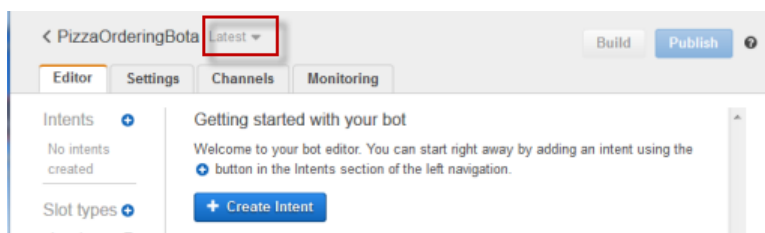
## Step 2: Create an Amazon Lex Bot

In this section, you create an Amazon Lex bot with minimum required information (name, output voice, and the IAM role that Amazon Lex can assume invoke your Lambda function). In the next section, you add intents to the bot.

In this section, you will create an Amazon Lex bot with minimum required information: name, output voice, and the IAM role that Amazon Lex can assume invoke your Lambda function. In the next section, you will add intents to the bot.

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. Create a bot
  - a. If you are creating your first bot, choose **Get Started**. Otherwise, on the **Bots** tab, choose **Create**.
  - b. On the **Create your Lex bot** page, choose **Custom bot** and provide the following information:
    - **App name** : PizzaOrderingBot
    - **Output voice** : Salli
    - **Session timeout** : 5 minutes.
    - **IAM role** : Choose the `lex-exec-role` from the list.
  - c. Choose **Create**.

The console sends Amazon Lex a request to create a new bot. Amazon Lex sets the bot version to \$LATEST. After the bot is created, console shows the bot editor tab as shown:



### Note

- Next to the bot name in the console is the bot version, Latest. New Amazon Lex resources have \$LATEST as version. For more information, see [Versioning and Aliases \(p. 50\)](#).

- There are no **Intents** or **Slot types** created at this time.
- The **Build** and **Publish** are bot level activities. After you configure the entire bot, you'll learn more about these activities.

In the next section, you add an intent (OrderPizza) to the bot you created in this section.

Next Step

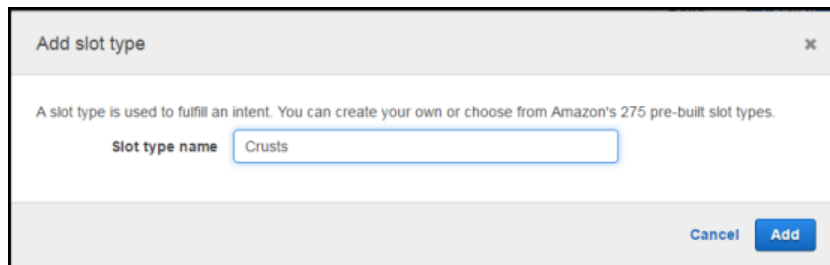
[Step 3: Create Slot Types \(p. 43\)](#)

## Step 3: Create Slot Types

In this section you create slot types (`Crusts`, `Sizes`, and `PizzaKind`) with specified enumeration values:

You use these types in the next section where you create the `PizzaOrder` intent.

1. Choose the plus sign (+) next to **Slot types**.
2. In the **Add slot type** wizard, type a slot type name (`Crusts`), and then choose **Add**.



The console sends the request to Amazon Lex to create the slot type.

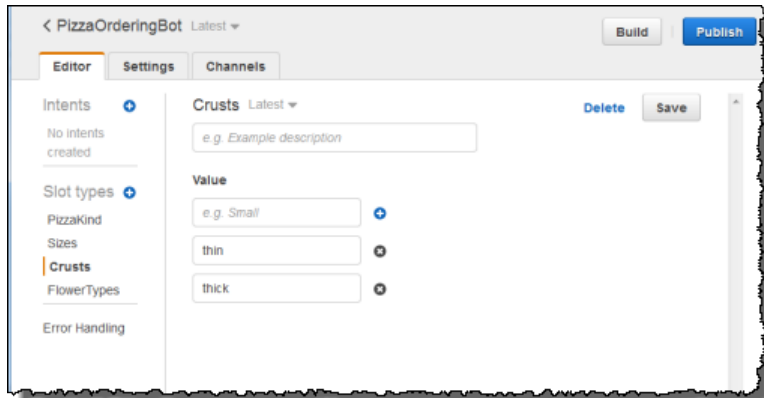
### Note

Slot type is an resource in Amazon Lex and it is similar to other Amazon Lex resources. Amazon Lex assigns `$LATEST` as the version for the newly created parameter type. For more information about versioning, see [Versioning and Aliases \(p. 50\)](#).

3. Repeat the preceding step and add the following slot types:
    - `Sizes`
    - `PizzaKind`
  4. Assign enumeration values to the slot types:
    - `Crusts` slot type with the enumeration values `thick` and `thin`.
    - `Sizes` slot type with the enumeration values `small`, `medium`, and `large`.
    - `PizzaKind` slot type with the enumeration values `cheese` and `veg`.
- a. Select a slot type.



- b. Add enumeration values. Choose the plus sign (+) next to the **Value** box and type a value. Press Enter or choose the plus sign (+) to add the next value.



- c. After you've added the enumeration values, choose **Save**.  
The console sends a request to Amazon Lex to update the slot type.
- d. Repeat the step to assign enumeration values to all of the slot types.

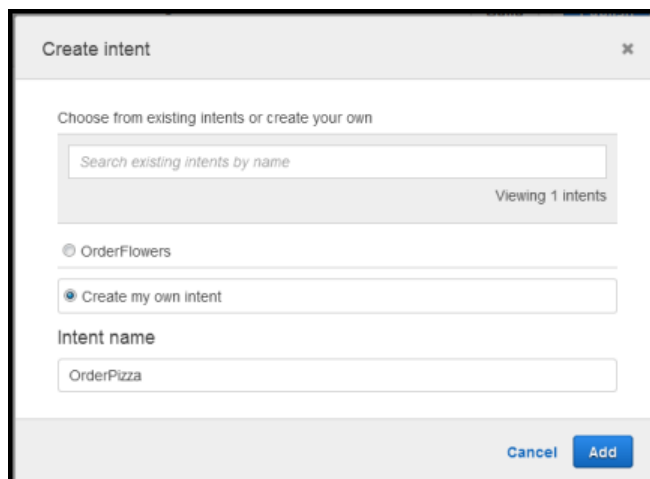
Next step

[Step 4: Create an Intent \(p. 44\)](#)

## Step 4: Create an Intent

In this section, you create an intent. You only specify a name (OrderPizza) when you create the intent. You then update the intent by adding the necessary configuration information.

1. In the Amazon Lex console, choose the plus sign (+) next to **Intents**.
2. On the **Add intent** wizard, type the intent name (OrderPizza) and then choose **Add**.

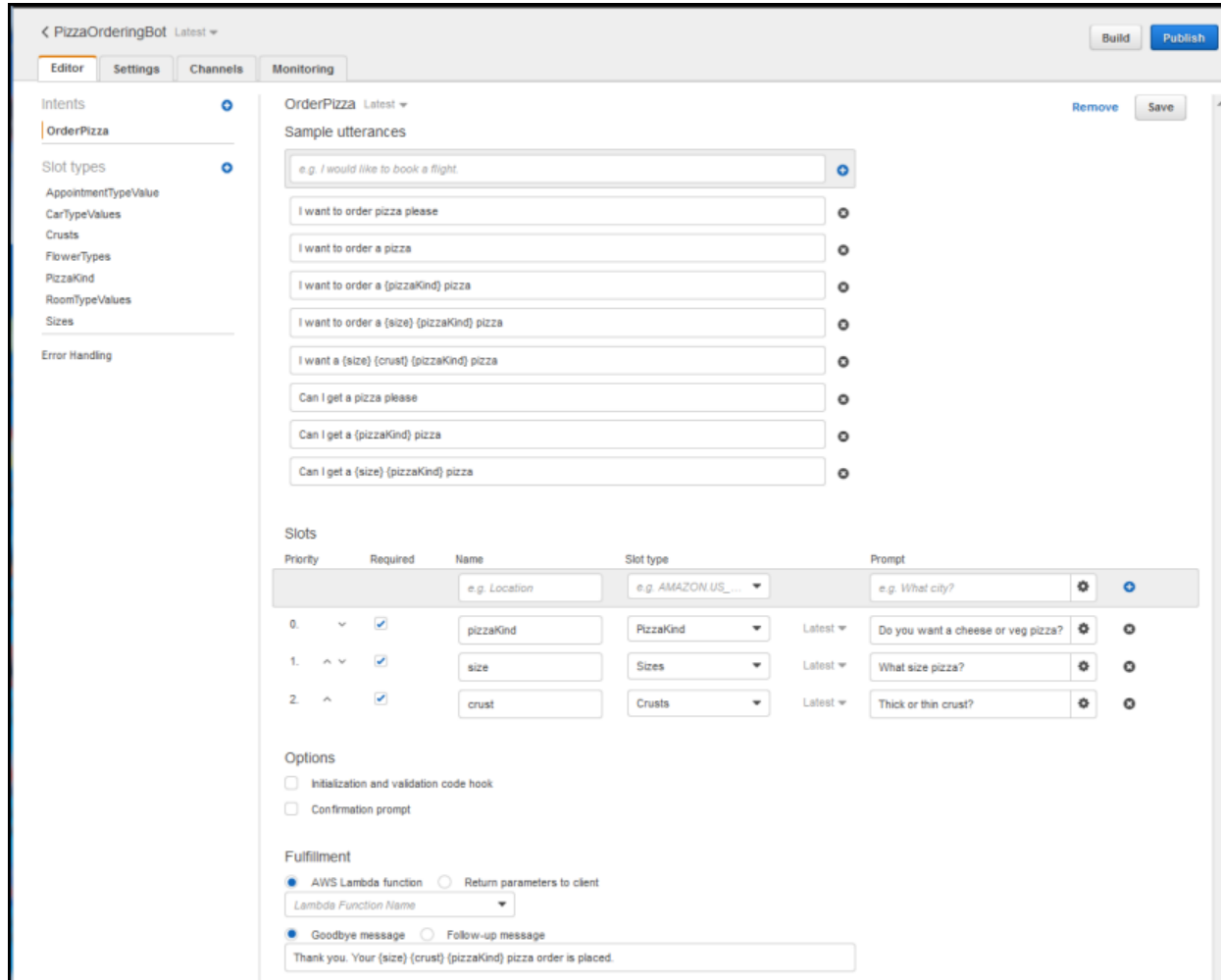


The console sends a request (`CreateIntent`) to Amazon Lex to create the OrderPizza intent.

3. The **Editor** tab shows the OrderPizza configuration page. Configure the intent as follows:

- **Sample utterances** – Type the following strings. Note that the curly braces {} provide slot names.
  - I want to order pizza please
  - I want to order a pizza
  - I want to order a {pizzaKind} pizza
  - I want to order a {size} {pizzaKind} pizza
  - I want a {size} {crust} crust {pizzaKind} pizza
  - Can I get a pizza please
  - Can I get a {pizzaKind} pizza
  - Can I get a {size} {pizzaKind} pizza
- **Slots** – Add the following slots:
  - **Name:** pizzaKind
    - **Slot Type:** PizzaKind
    - **Prompt:** Do you want a cheese or veg pizza?
  - **Name:** size
    - **Slot Type:** Sizes
    - **Prompt:** What size pizza?
  - **Name:** crust
    - **Slot Type:** Crusts
    - **Prompt:** Thick or thin crust?
- **Options** – Leave the check boxes unselected.
- **Fulfillment:** Leave the check boxes unselected.

An example screen shot is shown :



4. Verify that you have selected the **Required** check box for all slots.
5. Choose **Save**.

The console sends Amazon Lex a request to update the OrderPizza intent.

**Note**

Updates are always made to the \$LATEST version.

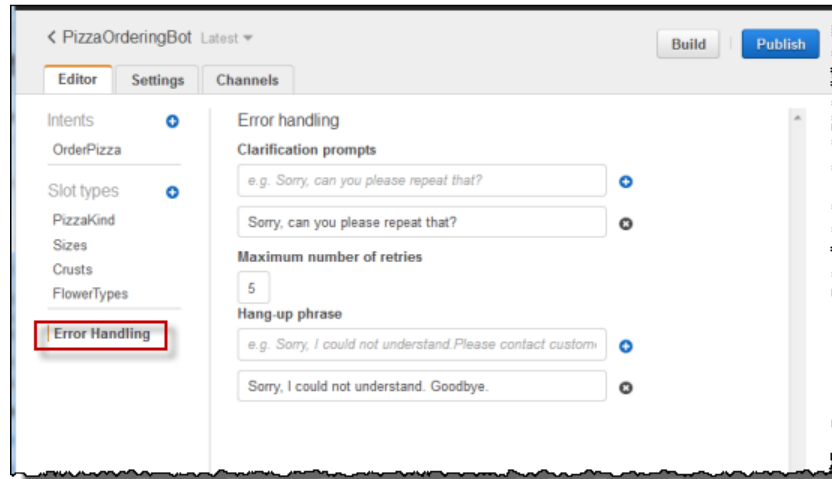
Next Step

[Step 5: Configure Error Handling \(p. 46\)](#)

## Step 5: Configure Error Handling

In this section you configure error handling information for the bot.

1. On the **Editor** tab of your PizzaOrderingBot bot, choose **Error Handling**.



2. The **Editor** tab displays information you can configure for bot error handling.

Note that the UI elements map to the bot configuration as follows:

- Information you provide in **Clarification Prompts**, maps to the bot's `clarificationPrompt` configuration.

When Amazon Lex can't determine the user intent, the service returns response with this message

- Information you provide in the **Hang-up** phrase, maps to the bot's `abortStatement` configuration.

Amazon Lex returns response with this message if the service can't determine the user's intent after a set number of consecutive requests.

Leave the defaults.

Next Step

[Step 6: Build and Test the Bot \(p. 47\)](#)

## Step 6: Build and Test the Bot

In this section, you build the bot and test it. During the build process, Amazon Lex creates a new version of each new resource (intents and slot types).

1. Choose **Build** to build the `PizzaOrderingBot` bot.

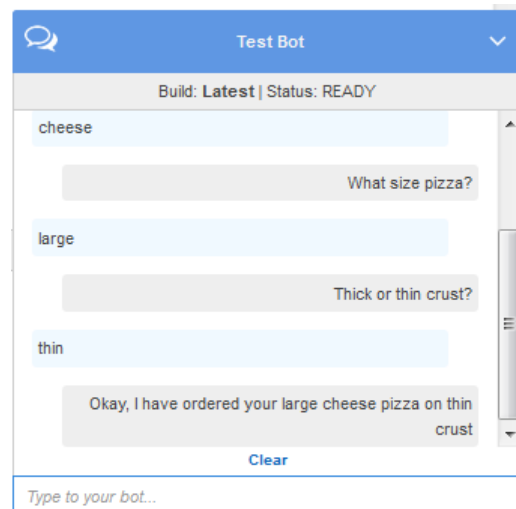
When you choose **Build**, the console sends the following requests to Amazon Lex:

- Request to create a new version (from the `$LATEST` version) of the slot types.
- Request to create a version of the intent (from the `$LATEST` version). In some cases, the console sends a request for the update API operation before creating a new version.
- Update the `$LATEST` version of the bot.

At this time, Amazon Lex builds a machine learning model for the bot. When you test the bot in the console, the console uses the runtime API to send the user input back to Amazon Lex, which then uses the machine learning model to interpret the user input.

Note that it can take some time to complete the build.

2. Test the bot in the Amazon Lex console. In the test window, start communicating with your Amazon Lex bot.
  - An example is shown:



- The PizzaOrder intent is configured with sample utterances that allow you to specify all all intent data in a single utterance. For example, the following is one of the sample utterance you configured for the PizzaOrder intent

```
I want a {size} {crust} crust {pizzaKind} pizza
```

So you specify the following utterance:

```
I want a large thin crust cheese pizza
```

#### Note

- When you type "I want to order a pizza", Amazon Lex detects the intent (OrderPizza). Then, Amazon Lex engages with the user get information for the slots.
- After the user provides all of the slot information, Amazon Lex invokes the code hook (that is, the Lambda function you configured for the intent).
- The Lambda function returns a message ("Okay, I have ordered your .....") to Amazon Lex, which Amazon Lex returns to the client.

## Exercise 3: Publish a Version and Create an Alias

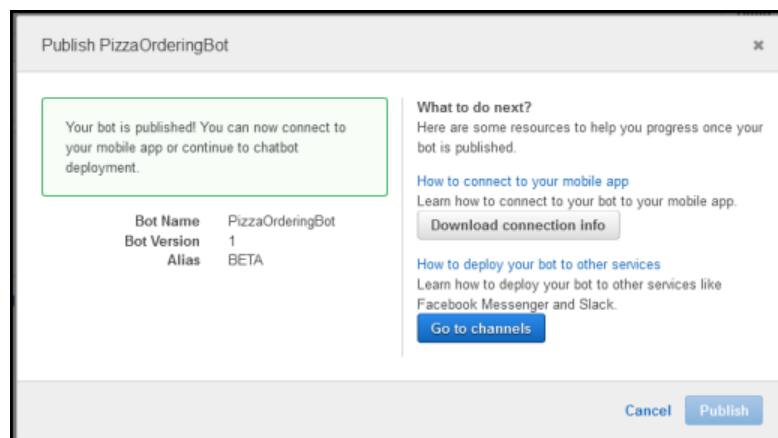
In Getting Started Exercises 1 and 2, you created a bot (\$LATEST version) and tested it. In this exercise, you do the following:

- Publish a new version of the bot. Amazon Lex takes a snapshot copy of the \$LATEST version to publish a new version.
- Create an alias (BETA) that points to it.

For more information about versioning and aliases, see [Versioning and Aliases \(p. 50\)](#).

Do the following to publish a version of a bot you created for this exercise:

1. In the Amazon Lex console, choose one of the bots you created.  
  
Verify that the console shows the \$LATEST as the bot version next to the bot name.
2. Choose **Publish**.
3. On the **Publish *botname*** wizard, specify an alias (BETA), and then choose **Publish**.
4. Verify that the Amazon Lex console shows the new version next to the bot name.



Now that you have a working bot with published version and an alias, you can deploy the bot (in your mobile application or integrate the bot with Facebook Messenger). For an example, see [Integrating an Amazon Lex Bot with Facebook Messenger \(p. 61\)](#).

# Versioning and Aliases

---

*This is prerelease documentation for a service in preview release. It is subject to change.*

Versioning enables you to manage your Amazon Lex production resources without breaking existing client applications. Amazon Lex supports publishing versions of bots, intents, and slot types. You can work with different variations of these resources in your development workflow, such as development, beta, and production. After you publish a version, it is immutable (that is, it can't be changed).

Amazon Lex bot can also have aliases. With aliases, deploying (or rolling back) your Amazon Lex bot is just one click away. Most importantly, the client applications do not have to upgrade. Conceptually, an alias is a pointer to a specific Amazon Lex bot version. Each bot alias points to a specific bot version (note that an alias can only point to a bot version, not to another alias).

## Topics

- [Versioning \(p. 50\)](#)
- [Aliases \(p. 52\)](#)

## Versioning

This section explains versioning. The examples use bot to create versions, but note that Amazon Lex supports versioning for bots, intents, and slot types.

### Creating an Amazon Lex Bot (the \$LATEST version)

When you create an Amazon Lex bot, there is only one version. It is the \$LATEST version.

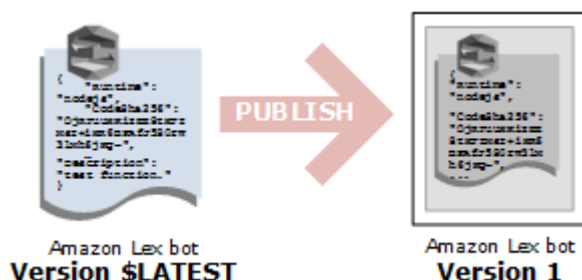


Amazon Lex bot  
Version \$LATEST

You can only update the \$LATEST version. Unless you choose to publish versions, the \$LATEST version is the only bot version you have.

## Publishing an Amazon Lex Bot Version

When you publish a version, Amazon Lex makes a snapshot copy of the Amazon Lex bot (with all its configuration) in the \$LATEST version. A published version is immutable. That is, you can't change the configuration. The new version has a version number:



You can publish versions using the console.

You can publish multiple versions. Each time you publish a version, Amazon Lex copies \$LATEST version to create a new version. When you publish additional versions, Amazon Lex assigns a strictly increasing sequence number for versioning, even if the resource was deleted and re-created. Version numbers are never reused. Suppose you delete the last published version, version 10, of a resource and then you publish a new version—the new version is 11.



### Note

Version numbers are never reused.

Amazon Lex only publishes a new version if the last published version is different from the \$LATEST version (that is, you updated the \$LATEST version after you published the last version). If you try to publish a version without any modifications to the \$LATEST, the last version is returned.

## Updating an Amazon Lex Resource

You can update only the \$LATEST version of the Amazon Lex resources (bot, intent, and slot types). Published versions are immutable. You cannot update any configuration information associated with a published version. You can publish a version anytime after you update a resource.



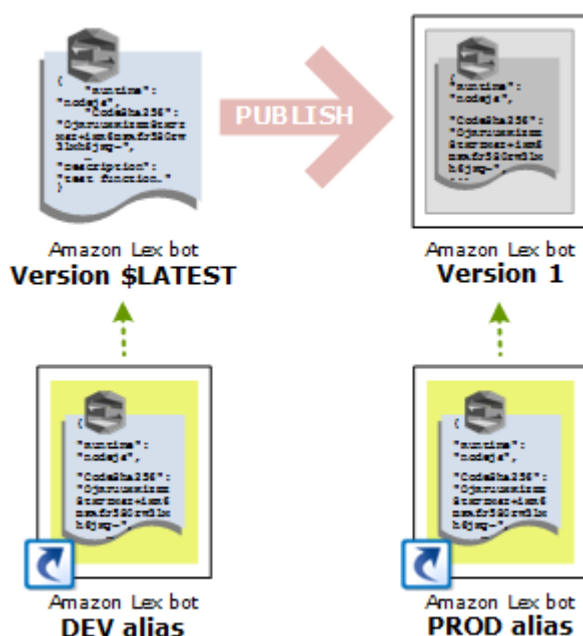
## Deleting an Amazon Lex Resource and a Specific Version

In the current implementation, Amazon Lex does not support deleting resources.

### Aliases

You can create aliases for your Amazon Lex bot. An Amazon Lex alias is like a pointer to a specific Amazon Lex bot version. By using aliases, you can access the Amazon Lex bot it is pointing to without the caller having to know the specific version the alias is pointing to.

The following example shows two versions of an Amazon Lex bot (version `$LATEST`, and version 1). Each of these bot versions has an alias (DEV and PROD) pointing to it.



Bot aliases enable the following:

- **Easier support for promotion of new versions of Amazon Lex bots and roll back when needed**
  - For example, after initially creating an Amazon Lex bot (the `$LATEST` version) you first publish a version 1 of it. You create an alias (named PROD) that points to version 1 of the bot. You can then use the PROD alias to refer to the version 1 of the bot.

For example, mobile clients send runtime API requests to a specific bot alias. Similarly, you specify an alias in bot channel association to deploy a specific version of your bot on Facebook Messenger. For more information, see [Bot Deployment Options \(p. 15\)](#).

Now, you can continue to update the bot (you can only update the `$LATEST` version) with all of your improvements, and then publish another stable and improved version (version 2). You can promote version 2 to production by remapping the PROD alias so that it points to version 2. If you find something wrong, you can easily roll back the production version to version 1 by remapping the PROD alias so that it points to version 1.

**Note**

In this context, the terms *promotion* and *roll back* refer to the remapping of aliases to different bot versions.

- **Simplify management of bot channel association** – Because you use an alias in the bot channel association you ensure that you don't need to update your event source mappings when you promote a new version or roll back to a previous version.

**Note**

When you create a bot, you can test the `$LATEST` version of the bot in the console. However, when integrating the bot with your client applications, we recommend that you first publish a version and create an alias. You then use the specific bot alias in the integration with your client application for reasons explained in this section. If you update an alias, it might take a few minutes for the alias propagation to complete.

# Using Lambda Functions

---

***This is prerelease documentation for a service in preview release. It is subject to change.***

You can create AWS Lambda functions for use as a code hook with your Amazon Lex bot. In your intent configuration, you can identify your Lambda function to perform initialization/validation, fulfillment, or both.

We recommend that you use a Lambda function as a code hook for your bot. Without a Lambda function, you configure your bot to simply return the intent information to the client application for fulfilling the intent.

The following sections provide additional information:

## Topics

- [Lambda Function Input Event and Response Format \(p. 54\)](#)
- [Amazon Lex and AWS Lambda Blueprints \(p. 59\)](#)

## Lambda Function Input Event and Response Format

This section describes the structure of event data that Amazon Lex provides to a Lambda function. You can use this information to parse the input in your Lambda code. The section also explains the format of the response that Amazon Lex expects your Lambda function to return.

## Topics

- [Input Event Format \(p. 54\)](#)
- [Response Format \(p. 57\)](#)

## Input Event Format

The following is the general format of an Amazon Lex event passed to a Lambda function. When you write your Lambda function code, you can use this information to parse the incoming event.

```
{
```

```
"messageVersion": "1.0",
"invocationSource": "FulfillmentCodeHook or DialogCodeHook",
"userId": "user-id specified in the POST request to Amazon Lex.",
"sessionAttributes": {
  "key1": "value1",
  "key2": "value2",
},
"bot": {
  "name": "bot-name",
  "alias": "bot-alias",
  "version": "bot-version"
},
"outputDialogMode": "Text or Voice, based on ContentType request header in runtime API request",
"currentIntent": {
  "name": "intent-name",
  "slots": {
    "slot-name": "value",
    "slot-name": "value",
    "slot-name": "value"
  },
  "confirmationStatus": "None, Confirmed, or Denied (intent confirmation, if configured)"
}
}
```

Note the following additional information about the event fields:

- **messageVersion** – The version identifying the format of the event data going into the Lambda function and expected format of the response from a Lambda function.

**Note**

You configure this value when defining an intent. In the current implementation, only message version 1.0 is supported. Therefore, the console doesn't show the message version and it assumes the default value of 1.0.

- **invocationSource** – Amazon Lex sets this to one of the following values to indicate why it is invoking the Lambda function.
- **DialogCodeHook** – Amazon Lex sets this value to direct the Lambda function to perform initialization/validation of the user's data input.

If the intent is configured to invoke a Lambda function as an initialization/validation code hook, Amazon Lex invokes the specified Lambda function on each user input (utterance) after Amazon Lex is aware of the intent.

**Note**

If the intent was not clear, Amazon Lex can't invoke the Lambda function.

- **FulfillmentCodeHook** – Amazon Lex sets this value to direct the Lambda function to fulfill an intent.

If the intent is configured to invoke a Lambda function as fulfillment code hook, Amazon Lex sets the `invocationSource` to this value only after it has all the slot data to fulfill the intent.

In your intent configuration, you can have two separate Lambda functions for the initialization/validation of user data and to fulfill the intent. You can also use one Lambda function to do both, in which case your Lambda function can use the `invocationSource` value to follow the proper code path.

- **userId** – This value is provided by the client application. Amazon Lex passes it to the Lambda function.
- **outputDialogMode** – For each user input, the client sends the request to Amazon Lex using one of the runtime API operations, [PostContent \(p. 106\)](#) or [PostText \(p. 113\)](#). From the API request parameters, Amazon Lex determines whether the response to the client (user) is text or voice, and sets this field accordingly.

Lambda function can use this information to generate an appropriate message. For example, if the client expects a voice response, your Lambda function could return SSML (instead of text).

- **sessionAttributes** – Application-specific session attributes that client sent in the request. Your Lambda function should send these back to Amazon Lex in response, if you want Amazon Lex to include them in the response to the client. For more information, see the runtime API operations, [PostContent \(p. 106\)](#) and [PostText \(p. 113\)](#).
- **currentIntent** – Provides the intent `name`, `slots`, and `confirmationStatus` fields.

`slots` provide list of slots configured for the intent and values, recognized by Amazon Lex in the user conversation from the beginning (otherwise, the values are null).

Is a map of slot names (configured for the intent) to slot values, recognized by Amazon Lex in the user conversation. Note that a slot value remains null until the user provides a value.

`confirmationStatus` provides the user response to a confirmation prompt, if any. For example, if Amazon Lex asks "Do you want to order a large cheese pizza?", depending on the user response, this field value can be `Confirmed` or `Denied`. Otherwise, this field value is `None`.

If the user confirms the intent, Amazon Lex sets this field to `Confirmed`. If the user denies the intent, Amazon Lex sets this value to `Denied`.

In the confirmation response, a user utterance might provide slot updates. For example, the user might say "yes, change size to medium." In this case, the subsequent Lambda event has the updated slot value (`PizzaSize` set to `medium`). In this case Amazon Lex sets the `confirmationStatus` to `None`, because the user modified some slot data, requiring the Lambda function to perform the user data validation.

## Response Format

Amazon Lex expects a response from a Lambda function in the following format:

```
{
  "sessionAttributes": {
    "key1": "value1",
    "key2": "value2"
    ...
  },
  "dialogAction": {
    "type": "ElicitIntent, ElicitSlot, ConfirmIntent, Delegate, or Close",
    "fulfillmentState": "Fulfilled or Failed",
    "message": {
      "contentType": "PlainText or SSML",
      "content": "message to convey to the user"
    },
    "intentName": "intent-name",
    "slots": {
      "slot-name": "value",
      "slot-name": "value",
      "slot-name": "value"
    },
    "slotToElicit": "slot-name",
    "responseCard": {
      "version": integer-value,
      "contentType": "application/vnd.amazonaws.card.generic",
      "genericAttachments": [
        {
          "title": "card-title",
          "subTitle": "card-sub-title",
          "imageUrl": "URL of the image to be shown",
          "attachmentLinkUrl": "URL of the attachment to be associated with
the card",
          "buttons": [
            {
              "text": "button-text",
              "value": "value sent to server on button click"
            }
          ]
        }
      ]
    }
  }
}
```

Note the following additional information about the fields in your Lambda function response:

- **sessionAttributes** – Application-specific session attributes.

This is an optional field. If you include it, it can be empty. If you want Amazon Lex to include any session attributes in its response to the client, your Lambda function must return them in this field. For more information, see the runtime API operations, [PostContent \(p. 106\)](#) and [PostText \(p. 113\)](#).

- **dialogAction** – Your Lambda function must return this field in the response. Only the `dialogAction.type` field is required. The value of the `dialogAction.type` directs Amazon Lex

to the next course of action, and what to expect from user after Amazon Lex returns a response to the client.

This value also determines what other fields the Lambda function needs to provide as part of the `dialogAction` value. The `dialogAction.type` value also determines what other `dialogAction` fields are required.

**Note**

With each of these `dialogAction` types, except the `Delegate` type, you include a message. If you don't provide a message, Amazon Lex picks an appropriate message from the bot configuration. If the bot also is not configured with a message that Amazon Lex can use, Amazon Lex returns an exception to the client.

The `dialogAction.type` values can be one of the following:

- `ElicitIntent` – Informs Amazon Lex that the user is expected to respond with an utterance that includes an intent. For example, "I want a large pizza" (which indicates `OrderPizzaIntent`) as opposed to just the utterance "large" (which is not sufficient for Amazon Lex to infer the intent).

The `message` and `responseCard` fields are optional for this type. If you don't provide a message, Amazon Lex uses one of the bot's clarification prompts (see the **Error Handling** section in the console).

- `ConfirmIntent` – Informs Amazon Lex that the user is expected to give a Yes or No answer to confirm or deny the current intent.

`intentName` is required for this `dialogAction.type`. `slots` must include all of the slots configured for the intent. If the value of a slot is unknown, it must be explicitly set to null (similar to Lambda function request).

The `message` and `responseCard` fields are optional. If you don't provide a message, Amazon Lex uses the intent's confirmation prompt.

- `Delegate` – Directs Amazon Lex to choose the next course of action based on the bot configuration. `slots` must include all of the slots configured for the intent. If the value of a slot is unknown, it must be explicitly set to null (similar to the Lambda function request). All other fields are ignored.
- `Close` – Informs Amazon Lex to not expect a response from the user. For example, the Lambda function might want to convey to the user "Your pizza order has been placed." For this message, you don't expect a user response.

The `fulfillmentState` field is required for this type. Amazon Lex uses this value to set the `dialogState` in its response to the client.

The `slots`, `slotsToElicit`, and `intentName` fields should not be included. The `message` and `responseCard` fields are optional. If you don't provide a message, Amazon Lex uses the goodbye message or the follow-up message that is configured for the intent.

- `ElicitSlot` – Informs Amazon Lex that the user is expected to provide a slot value in response. For example, a value for the `pizzaSize` or `pizzaKind` slots.

The `intentName` and `slotToElicit` fields are required for this `dialogAction.type.slots` must include all of the slots configured for the intent. If the value of a slot is unknown, it must be explicitly set to null (like in the Lambda function request).

The `message` and `responseCard` fields are optional. If you don't provide a message, Amazon Lex uses one of the slot value elicitation prompts configured for the slot.

#### Note

- A `message` or a `responseCard` generated in a Lambda function cannot have substitutions. For more information, see [Managing Messages \(Prompts and Statements\) \(p. 7\)](#).
- If you do not provide a message for any `dialogAction`, Amazon Lex looks for a suitable message in the bot configuration.
- You can fill some of the slot values in your Lambda function. For example, if you determine that the current user is named Joe (from session attributes), you might automatically populate some of the slots based on what you know about Joe's preferences (assuming you have a back-end database that contains information about users and what they like or dislike). For an example that showcases using session attributes for cross-intent data sharing, see [Example Bot: BookTrip \(p. 72\)](#).
- Depending on the capabilities, a client application can use response card to draw an interactive UI to establish a rich conversational engagement. You can use a Lambda function to generate response cards dynamically. For more information, see [Response Cards \(p. 12\)](#).

## Amazon Lex and AWS Lambda Blueprints

The Amazon Lex console provides example bots (called bot blueprints) that are preconfigured so you can quickly create and test a bot in the console. For each of these bot blueprints, Lambda function blueprints are also available. These blueprints provide sample code that works with the specific bots. With these blueprints, you can quickly create a bot that is configured with a Lambda function as a code hook, and test the end-to-end setup without having to write any code.

The following is a list of Amazon Lex bot blueprints and corresponding AWS Lambda function blueprints that you can use as a code hook for bots.

- Amazon Lex blueprint – `OrderFlowers`
  - AWS Lambda blueprint – `lex-order-flowers (Node.js code)`, and `lex-order-flowers-python`
- Amazon Lex blueprint – `ScheduleAppointment`
  - AWS Lambda blueprint – `lex-make-appointment (Node.js code)` and `lex-make-appointment-python`



- Amazon Lex blueprint – BookTrip
  - AWS Lambda blueprint – `lex-book-trip` (Node.js code) and `lex-book-trip-python`

To create a bot using a blueprint and configure it to use a Lambda function as a code hook, see [Exercise 1: Create an Amazon Lex Bot Using a Blueprint \(p. 17\)](#). For example of using other blueprints, see [Additional Examples: Creating Amazon Lex Bots \(p. 66\)](#).

# Deploying Amazon Lex Bots on Various Platforms

---

***This is prerelease documentation for a service in preview release. It is subject to change.***

This section provides examples of deploying your Amazon Lex bot on various platforms.

## Topics

- [Deploying an Amazon Lex Bot on a Messaging Platform \(p. 61\)](#)
- [Deploying an Amazon Lex Bot in Mobile Applications \(p. 65\)](#)

## Deploying an Amazon Lex Bot on a Messaging Platform

The following exercise provides instructions on how to associate your Amazon Lex bot with Facebook Messenger.

## Topics

- [Integrating an Amazon Lex Bot with Facebook Messenger \(p. 61\)](#)

## Integrating an Amazon Lex Bot with Facebook Messenger

***This is prerelease documentation for a service in preview release. It is subject to change.***

## Topics

- [Step 1: Create an Amazon Lex bot \(p. 62\)](#)
- [Step 2: Create an IAM Role \(p. 62\)](#)

- [Step 3: Create a Facebook Application](#) (p. 63)
- [Step 4: Integrate Facebook Messenger With Amazon Lex Bot](#) (p. 63)
- [Step 5: Test the integration](#) (p. 64)

This exercise shows how to integrate Facebook Messenger with your Amazon Lex bot. You perform the following steps:

- Create an Amazon Lex bot.
- Create an IAM role that the Amazon Lex integration service can assume to invoke the Amazon Lex runtime service.
- Integrate Facebook Messenger with your Amazon Lex bot.

## Step 1: Create an Amazon Lex bot

In this section, you create an Amazon Lex bot.

1. Create an Amazon Lex bot. For instructions, see [Getting Started](#) (p. 17).
2. Deploy the bot and create an alias. For instructions, see [Exercise 3: Publish a Version and Create an Alias](#) (p. 48).

## Step 2: Create an IAM Role

Create an IAM role (say `LexChannelExecutionRole`) that the Amazon Lex channel service can assume. This role will need permissions to invoke Amazon Lex runtime service.

### To create an IAM role 1 (`LexChannelExecutionRole`)

1. Sign in to the Identity and Access Management (IAM) console at <https://console.aws.amazon.com/iam/>.
2. Follow the steps in [Creating a Role to Delegate Permissions to an AWS Service](#) in the *IAM User Guide* to create an IAM role (execution role). As you follow the steps to create a role, note the following:
  - In **Role Name**, use a name that is unique within your AWS account (for example, **LexChannelExecutionRole**).
  - In **Select Role Type**, choose **AWS Service Roles**, and then choose **AWS Lambda**.

#### Note

In the current implementation, Amazon Lex service role is not available. Therefore, you first create a role using the AWS Lambda as the AWS service role. After you create the role, you update the trust policy and specify Amazon Lex runtime service as the service principal to assume the role.

- In **Attach Policy**, choose **Next Step** (that is, you create a role without any permissions). Create the role.
- Choose the role you created and update policies as follows:
  - In the **Permissions** tab, choose **Inline Policies**, and then attach the following custom policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "lex:PostText"
      ]
    }
  ]
}
```

```
    ],  
    "Effect": "Allow",  
    "Resource": "*"    
  }  
]  
}
```

- In the **Trust Relationships** tab, choose Edit Trust Relationship, and specify the Amazon Lex service principal ("channels.lex.amazonaws.com"). The updated policy should look as shown:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Principal": {  
        "Service": "channels.lex.amazonaws.com"  
      },  
      "Action": "sts:AssumeRole"  
    }  
  ]  
}
```

### Step 3: Create a Facebook Application

On the Facebook developer portal, create a Facebook application and a Facebook page. For instructions, see [Quick Start](#) in the Facebook Messenger platform documentation. Write down the following:

- **App Secret** for the Facebook App.
- **Page Access Token** for the Facebook page.

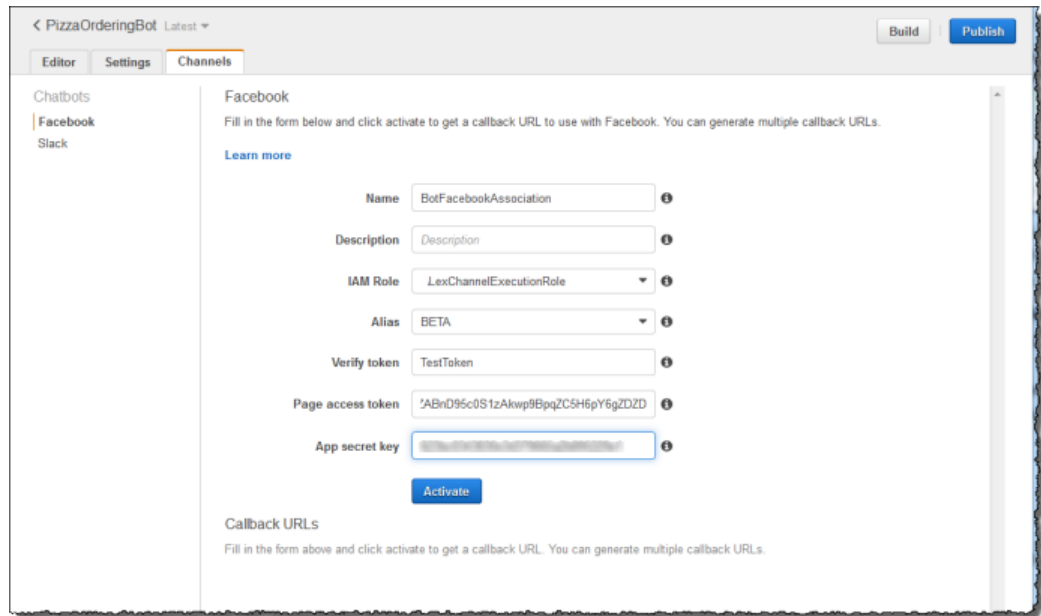
### Step 4: Integrate Facebook Messenger With Amazon Lex Bot

In this section, you integrate Facebook Messenger with your Amazon Lex bot.

1. Open the Amazon Lex console, and then associate Facebook Messenger with your Amazon Lex bot.

After you complete this step, the console provides a callback URL. Write down this URL.

- a. Choose your Amazon Lex bot.
- b. Choose the **Channels** tab.
- c. Choose **Facebook** under For **Chatbots**. The console displays the Facebook integration page.
- d. On the **Facebook** integration page, provide the following information:
  - Type a name: `BotFacebookAssociation`
  - Choose the IAM role you created in the preceding section.
  - Choose the bot alias from the drop-down.
  - Type the verify token. This can be any string you choose (for example, `ExampleToken`). You use this same token in the Facebook developer portal in the Webhook setup step.
  - Type the page access token and the app secret key you obtained from Facebook in the preceding step.



- e. Choose **Activate**.

The console creates the bot channel association and returns a callback URL. Write down this URL.

2. On the Facebook developer portal, choose your app. Then, select the **Messenger** product and choose **Setup webhooks** in the **Webhooks** section of the page.

For instructions, see [Quick Start](#) in the Facebook Messenger platform documentation.

On the webhook page subscription wizard, do the following:

- For **Callback URL**, type the callback URL provided in the Amazon Lex console in the preceding section.
  - For **Verify Token**, type the same token that you used in Amazon Lex.
  - Choose **Subscription Fields** (messages, messaging\_postbacks, and messaging\_options).
  - Choose **Verify and Save**. This results in a handshake between Facebook and Amazon Lex.
3. Enable Webhooks integration. Choose the page you created, and then choose **subscribe**.

**Note**

If you update or recreate a webhook, you must unsubscribe and then subscribe to the page again.

## Step 5: Test the integration

You can now start conversation from Facebook Messenger with your Amazon Lex bot.

1. Open your Facebook page and choose, **Message**.
2. In the Messenger window that opens, use the same test utterances provided in getting started with your Amazon Lex bot.

## Deploying an Amazon Lex Bot in Mobile Applications

Using AWS SDKs, you can integrate your Amazon Lex bot with your mobile applications. For more information, see the following topics:

- Android SDK – [Getting Started with Amazon Lex Android SDK](#)
- iOS SDK – [Getting Started with Amazon Lex iOS SDK](#)

# Additional Examples: Creating Amazon Lex Bots

---

*This is prerelease documentation for a service in preview release. It is subject to change.*

The following sections provide additional Amazon Lex exercises with step-by-step instructions.

## Topics

- [Example Bot: ScheduleAppointment](#) (p. 66)
- [Example Bot: BookTrip](#) (p. 72)
- [Example: Using a Response Card](#) (p. 96)

## Example Bot: ScheduleAppointment

*This is prerelease documentation for a service in preview release. It is subject to change.*

The example bot in this exercise schedules appointments for a dentist's office. After creating the bot, you test it using the client provided in the console.

In this exercise you do the following:

- Create and test a bot (using the **ScheduleAppointment** blueprint). For this exercise, you use bot blueprint to quickly setup and test the bot. For a list of available blueprints, see [Amazon Lex and AWS Lambda Blueprints](#) (p. 59). This bot is preconfigured with one intent (MakeAppointment).
- Create and test an AWS Lambda function (using the **lex-make-appointment-python** blueprint provided by AWS Lambda). You configure the MakeAppointment intent to use this Lambda function as code hook to perform both initialization/validation and fulfillment tasks.

### Note

The example Lambda function provided showcases dynamic conversation based upon mocked availability of a dentist appointment. In a real application, you might use a real calendar to set an appointment.

- Update the MakeAppointment intent configuration to use the Lambda function as code hook. Then, you test the end-to-end experience.

The following sections provide summary information about the blueprints you use in this exercise.

### Topics

- [Overview of the Bot Blueprint \(ScheduleAppointment\) \(p. 67\)](#)
- [Overview of the Lambda Function Blueprint \(lex-make-appointment\) \(p. 68\)](#)
- [Step 1: Prepare \(p. 68\)](#)
- [Step 2: Create an Amazon Lex Bot \(p. 69\)](#)
- [Step 3: Create a Lambda function \(p. 71\)](#)
- [Step 4: Update the Intent: Configure a Code Hook \(p. 71\)](#)

## Overview of the Bot Blueprint (ScheduleAppointment)

The **ScheduleAppointment** blueprint that you use to create a bot for this exercise provides the following preconfiguration:

- **Slot types** – One custom slot type called `AppointmentTypeValue` with the enumeration values `root canal`, `cleaning`, and `whitening`.
- **Intent** – The bot supports one intent (`MakeAppointment`). It is preconfigured as follows:

- **Slots** – The intent is configured with the following slots:
  - Slot `AppointmentType`, of the `AppointmentTypes` custom type.
  - Slot `Date`, of the `AMAZON.DATE` built-in type.
  - Slot `Time`, of the `AMAZON.TIME` built-in type.
- **Utterances** – The intent is pre-configured with the following utterances:
  - "I would like to book an appointment"
  - "Book an appointment"
  - "Book a {AppointmentType}"

If user utters any of these, Amazon Lex determines `MakeAppointment` is the intent and then uses the prompts to elicit slot data.

- **Prompts** – The intent is preconfigured with the following prompts:
  - Prompt for the `AppointmentType` slot – "What type of appointment would you like to schedule?"
  - Prompt for the `Date` slot – "When should I schedule your {AppointmentType}?"
  - Prompt for the `Time` slot – "At what time do you want to schedule the {AppointmentType}?" and "At what time on {Date}?"
  - Confirmation prompt – "{Time} is available, should I go ahead and book your appointment?"
  - Cancel message– "Okay, I will not schedule an appointment."



## Overview of the Lambda Function Blueprint (lex-make-appointment)

The Lambda function blueprint (**lex-make-appointment-python**) is specifically provided for use as a code hook for bots you create using the **ScheduleAppointment** bot blueprint.

This Lambda function blueprint code can perform both the initialization/validation and fulfillment tasks.

- The Lambda function code showcases a dynamic conversation that is based on example availability for a dentist appointment (in real applications you might use a calendar). For the day or date that the user specifies, the code is configured as follows:
  - If there are no appointments available, the Lambda function returns a response directing Amazon Lex to prompt user for another day or date (by setting dialogAction type to `ElicitSlot`). For more information, see [Response Format \(p. 57\)](#).
  - If there is only one appointment available, the Lambda function suggests the available time in the response and directs Amazon Lex to obtain user confirmation by setting the dialogAction in the response to `ConfirmIntent`. This illustrates how you can improve the user experience, by proactively suggesting the available time for an appointment.
  - If there are multiple appointments available, the Lambda function returns list of available times in the response to Amazon Lex. Amazon Lex returns response back to the client with message from the Lambda function.
- As the fulfillment code hook, the Lambda function returns a summary message indicating that an appointment is scheduled (that is, the intent is fulfilled).

### Note

In this example, we show use of response card. That is, the Lambda function also constructs and returns a response card to Amazon Lex. The response card shows a list of available days and times as buttons to choose from. But you cannot see the response card when testing the bot using the client provided by the Amazon Lex console. To see the response card, you must integrate the bot with messaging platforms such as Facebook messenger. For instructions, see [Integrating an Amazon Lex Bot with Facebook Messenger \(p. 61\)](#). For more information about response cards, see [Managing Messages \(Prompts and Statements\) \(p. 7\)](#).

When Amazon Lex invokes the Lambda function, it passes event data as input. One of the event fields is `invocationSource`, which the Lambda function uses to choose between an input validation and fulfillment activity. For more information, see [Input Event Format \(p. 54\)](#).

Next Step

[Step 1: Prepare \(p. 68\)](#)

## Step 1: Prepare

In this section, you create two IAM roles:

- IAM role that Amazon Lex can assume to invoke your Lambda function on your behalf.
- IAM role that AWS Lambda can assume. This role has permissions for the CloudWatch actions to write any logs that your Lambda function generates.

You can use the IAM roles that you created for the Getting Started Exercise 1 for this exercise. If you created these roles already, go to the next section. For instructions to create these roles, see [Step 1: Prepare \(p. 19\)](#).

Next Step

[Step 2: Create an Amazon Lex Bot](#) (p. 69)

## Step 2: Create an Amazon Lex Bot

In this section, you create an Amazon Lex bot using the blueprint (ScheduleAppointment) provided in by Amazon Lex console.

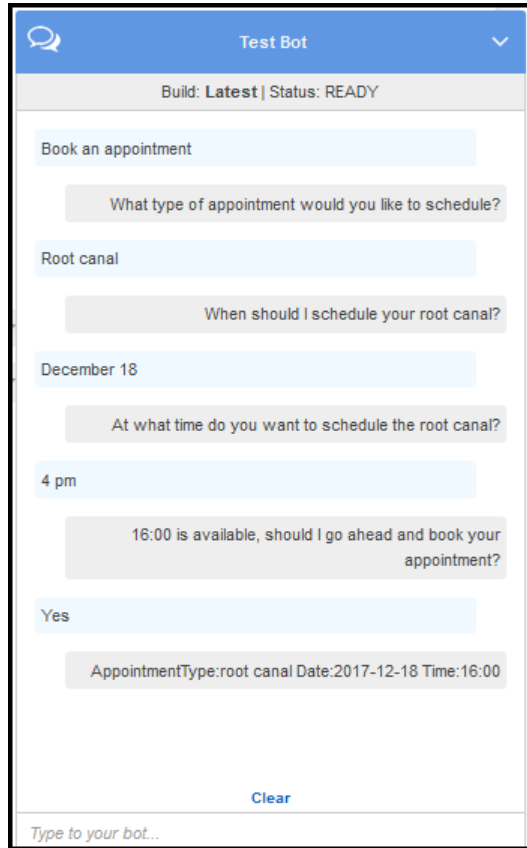
1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.
2. On the **Bots** page, choose **Create**.
3. On the **Create your Lex bot** page, do the following:
  - Choose **ScheduleAppointment** blueprint.
  - Leave the default bot name (ScheduleAppointment).
  - Choose the IAM role (**lex-exec-role**), role you created in the preceding section for Amazon Lex to assume.
4. Choose **Create**.

This step saves and builds the bot. The console sends the following requests to Amazon Lex as part of the build process:

- Create a new version of the slot types (from the \$LATEST version). For information about slot types defined in this bot blueprint, see [Overview of the Bot Blueprint \(ScheduleAppointment\)](#) (p. 67).
- Create a version of the MakeAppointment intent (from the \$LATEST version). In some cases, the console sends a request for the update API operation before creating a new version.
- Update the \$LATEST version of the bot.

At this time, Amazon Lex builds a machine learning model for the bot. When you test the bot in the console, the console uses the runtime API to send user input back to Amazon Lex, which then uses the machine learning model to interpret user input.

5. The console shows the **ScheduleAppointment** bot. On the **Editor** tab, review the preconfigured intent (MakeAppointment) details.
6. Test the bot in the test window. Use the following to engage in a test conversation with your bot:



Note the following:

- From the initial user input ("Book an appointment"), the bot infers the intent (MakeAppointment).
- The bot then uses the configured prompts to get slot data from the user.
- Note that in the bot blueprint has the MakeAppointment intent configured with the following confirmation prompt:

```
{Time} is available, should I go ahead and book your appointment?
```

Therefore, after user provides all slot data, Amazon Lex returns a response to the client with confirmation prompt as the message. The client displays the message for the user:

```
16:00 is available, should I go ahead and book your appointment?
```

Notice that the bot accepts any appointment date and time values because you don't have any code to initialize/validate user data. In the next section, you add a Lambda function to do this.

Next Step

[Step 3: Create a Lambda function \(p. 71\)](#)

## Step 3: Create a Lambda function

In this section, you create a Lambda function using a blueprint (**lex-make-appointment-python**) provided in the AWS Lambda console. You also test the Lambda function by invoking it using a sample Amazon Lex event data provided by the console.

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose **Create a Lambda function**.
3. On **Select blueprint**, type **lex** to find the blueprint, choose the `lex-make-appointment-python` blueprint.
4. Configure the Lambda function as follows and then choose **Create Function**.
  - Type a Lambda function name (`MakeAppointmentCodeHook`).
  - For the IAM role, choose the **Choose an existing role** and then select `lambda-exec-role-for-lex-get-started` from the **Existing role** list.
  - Leave other default values.
5. Test the Lambda function.
  - a. Choose **Actions, Configure test event**.
  - b. Choose **Lex-Make Appointment (preview)** from the **Sample event template** list. This sample event matches the Amazon Lex request/response model (see [Using Lambda Functions \(p. 54\)](#)).

This sample event matches the Lex request/response model (see [Using Lambda Functions \(p. 54\)](#)), with values set to match a request from your Lex bot.
  - c. Choose **Save and test**.
  - d. Verify that the Lambda function successfully executed. The response in this case matches the Amazon Lex response model.

Next Step

[Step 4: Update the Intent: Configure a Code Hook \(p. 71\)](#)

## Step 4: Update the Intent: Configure a Code Hook

In this section, you update the configuration of the `MakeAppointment` intent configuration to use the Lambda function as a code hook for the validation and fulfillment activities.

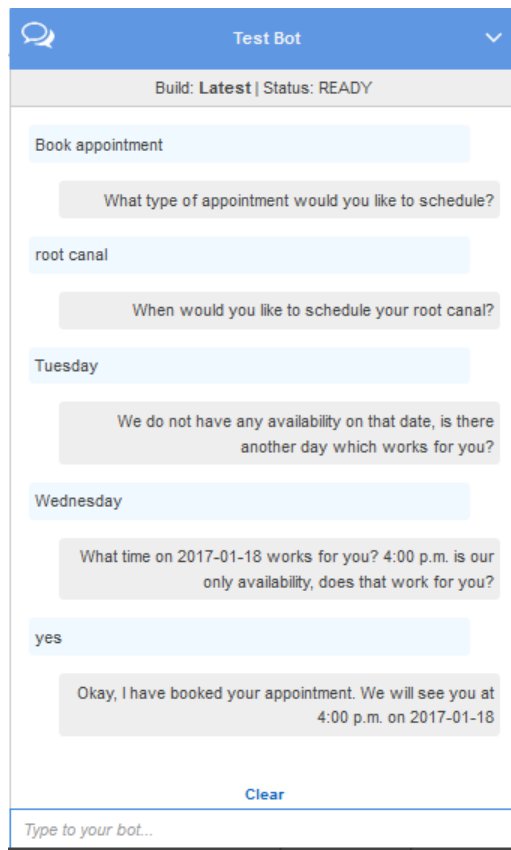
1. In the Amazon Lex console, select the **ScheduleAppointment** bot. The console shows the **MakeAppointment** intent. Modify the intent configuration as follows:

### Note

You can update only the `$LATEST` versions of any of the Amazon Lex resources including the intents. Make sure that the intent version is set to `$LATEST`. You have not published a version of your bot yet. It should still be `$LATEST` in the console.

- a. Choose **Initialization and validation code hook** in the **Options** section, and choose the Lambda function from the list.
- b. Choose **AWS Lambda function** in the **Fulfillment** section, and choose the Lambda function from the list.
- c. Choose **Goodbye message** and type a message.

2. Choose **Save** and then **Build**.
3. Test the bot.



## Example Bot: BookTrip

***This is prerelease documentation for a service in preview release. It is subject to change.***

This example illustrates creating a bot that is configured to support multiple intents. The example also illustrates how you can use session attributes for cross-intent information sharing. After creating the bot, you use a test client in the Amazon Lex console to test the bot (BookTrip). Note that the client uses the [PostText \(p. 113\)](#) runtime API operation to send requests to Amazon Lex for each user input.

The BookTrip bot in this example is configured with two intents (BookHotel and BookCar). For example, suppose a user first books a hotel. During the interaction, the user provides information such as check-in dates, location, and number of nights. After the intent is fulfilled, the client can persist this information using session attributes. For more information about session attributes, see [PostText \(p. 113\)](#).

Now suppose the user continues to book a car. Using information that the user provided in the previous BookHotel intent (that is, destination city, check-in and check-out dates), the code hook (Lambda function) you configured to initialize/validate the BookCar intent, initializes slot data for the BookCar intent (that is, destination, pick-up city, pick-up date, and return date). This illustrates how the cross-intent information sharing enables you to build bots that can engage in dynamic conversation with the user.

In this example, we use the following session attributes. Note that only the client and the Lambda function can set and update session attributes. Amazon Lex only passes these between the client and the Lambda function. Amazon Lex doesn't maintain or modify any session attributes.:

- `currentReservation` – Contains slot data for an in-progress reservation and other relevant information. For example, the following is a sample request from the client to Amazon Lex. It shows the `currentReservation` session attribute in the request body.

```
POST /bot/BookTrip/alias/$LATEST/user/wch89kjqcpkds8seny7dly5x3otq68j3/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "Chicago",
  "sessionAttributes": {
    "currentReservation": "{ \"ReservationType\": \"Hotel\",
                          \"Location\": \"Moscow\",
                          \"RoomType\": null,
                          \"CheckInDate\": null,
                          \"Nights\": null }"
  }
}
```

- `lastConfirmedReservation` – Contains similar information for a previous intent, if any. For example, if the user booked a hotel and then is in process of booking a car, this session attribute stores slot data for the previous `BookHotel` intent.
- `confirmationContext` – The Lambda function sets this to `AutoPopulate` when it prepopulates some of the slot data based on slot data from the previous reservation (if there is one). This enables cross-intent information sharing. For example, if the user previously booked a hotel and now wants to book a car, Amazon Lex can prompt the user to confirm (or deny) that the car is being booked for the same city and dates as their hotel reservation

In this exercise you use blueprints to create an Amazon Lex bot and a Lambda function. For more information about blueprints, see [Amazon Lex and AWS Lambda Blueprints \(p. 59\)](#).

#### Next Step

[Step 1: Review the Blueprints Used in this Exercise \(p. 73\)](#)

## Step 1: Review the Blueprints Used in this Exercise

#### Topics

- [Overview of the Bot Blueprint \(BookTrip\) \(p. 73\)](#)
- [Overview of the Lambda Function Blueprint \(lex-book-trip-python\) \(p. 75\)](#)

### Overview of the Bot Blueprint (BookTrip)

The blueprint (**BookTrip**) you use to create a bot provides the following preconfiguration:

- **Slot types** – Two custom slot types:
  - `RoomTypes` with enumeration values: `king`, `queen`, and `deluxe`, for use in the `BookHotel` intent.
  - `CarTypes` with enumeration values: `economy`, `standard`, `midsize`, `full size`, `luxury`, and `minivan`, for use in the `CarTypes` intent.

- **Intent 1 (`BookHotel`)** – It is preconfigured as follows:

- **Preconfigured slots**

- `RoomType`, of the `RoomTypes` custom slot type
- `Location`, of the `AMAZON.US_CITY` built-in slot type
- `CheckInDate`, of the `AMAZON.DATE` built-in slot type
- `Nights`, of the `AMAZON.NUMBER` built-in slot type

- **Preconfigured utterances**

- "Book a hotel"
- "I want to make hotel reservations"
- "Book a {Nights} stay in {Location}"

If the user utters any of these, Amazon Lex determines that `BookHotel` is the intent and then prompts the user for slot data.

- **Preconfigured prompts**

- Prompt for the `Location` slot – "What city will you be staying in?"
- Prompt for the `CheckInDate` slot – "What day do you want to check in?"
- Prompt for the `Nights` slot – "How many nights will you be staying?"
- Prompt for the `RoomType` slot – "What type of room would you like, queen, king, or deluxe?"
- Confirmation statement – "Okay, I have you down for a {Nights} night stay in {Location} starting {CheckInDate}. Shall I book the reservation?"
- Denial – "Okay, I have cancelled your reservation in progress."

- **Intent 2 (`BookCar`)** – It is preconfigured as follows:

- **Preconfigured slots**

- `PickUpCity`, of the `AMAZON.US_CITY` built-in type
- `PickUpDate4`, of the `AMAZON.DATE` built-in type
- `ReturnDate`, of the `AMAZON.DATE` built-in type
- `DriverAge`, of the `AMAZON.NUMBER` built-in type
- `CarType`, of the `CarTypes` custom type

- **Preconfigured utterances**

- "Book a car"
- "Reserve a car"
- "Make a car reservation"

If the user utters any of these, Amazon Lex determines `BookCar` is the intent and then prompts the user for slot data.

- **Preconfigured prompts**

- Prompt for the `PickUpCity` slot – "In what city do you need to rent a car?"

- Prompt for the `PickUpDate` slot – "What day do you want to start your rental?"
- Prompt for the `ReturnDate` slot – "What day do you want to return this car?"
- Prompt for the `DriverAge` slot – "How old is the driver for this rental?"
- Prompt for the `CarType` slot – "What type of car would you like to rent? Our most popular options are economy, midsize, and luxury"
- Confirmation statement – "Okay, I have you down for a {CarType} rental in {PickUpCity} from {PickUpDate} to {ReturnDate}. Should I book the reservation?"
- Denial – "Okay, I have cancelled your reservation in progress."

## Overview of the Lambda Function Blueprint (lex-book-trip-python)

In addition to the bot blueprint, AWS Lambda provides a blueprint (**lex-book-trip-python**) that you can use as a code hook with the bot blueprint. For a list of bot blueprints and corresponding Lambda function blueprints, see [Amazon Lex and AWS Lambda Blueprints \(p. 59\)](#).

When you create a bot using the BookTrip blueprint, you update configuration of both the intents (BookCar and BookHotel) by adding this Lambda function as a code hook for both initialization/validation of user data input and fulfillment of the intents.

This Lambda function code provided showcases dynamic conversation using previously known information (persisted in session attributes) about a user to initialize slot values for an intent. For more information, see [Managing Conversation Context \(p. 14\)](#).

Next Step

[Step 2: Prepare \(p. 75\)](#)

## Step 2: Prepare

In this section, you create two IAM roles:

- IAM role that Amazon Lex can assume to invoke your Lambda function on your behalf.
- IAM role that AWS Lambda can assume. This role has permissions for the CloudWatch actions to write any logs that your Lambda function generates.

You can use the IAM roles that you created for the Getting Started Exercise 1 for this exercise. If you created these roles already, go to the next section. For instructions to create these roles, see [Step 1: Prepare \(p. 19\)](#).

Next Step

[Step 3: Create an Amazon Lex Bot \(p. 75\)](#)

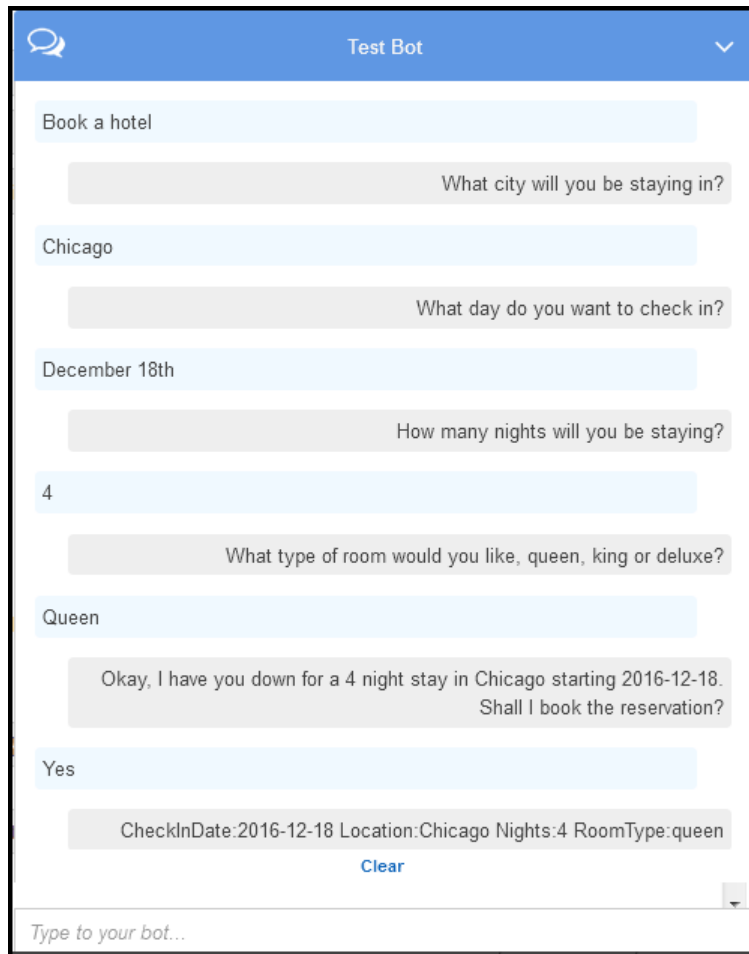
## Step 3: Create an Amazon Lex Bot

In this section, you create an Amazon Lex bot (BookTrip).

1. Sign in to the AWS Management Console and open the Amazon Lex console at <https://console.aws.amazon.com/lex/>.



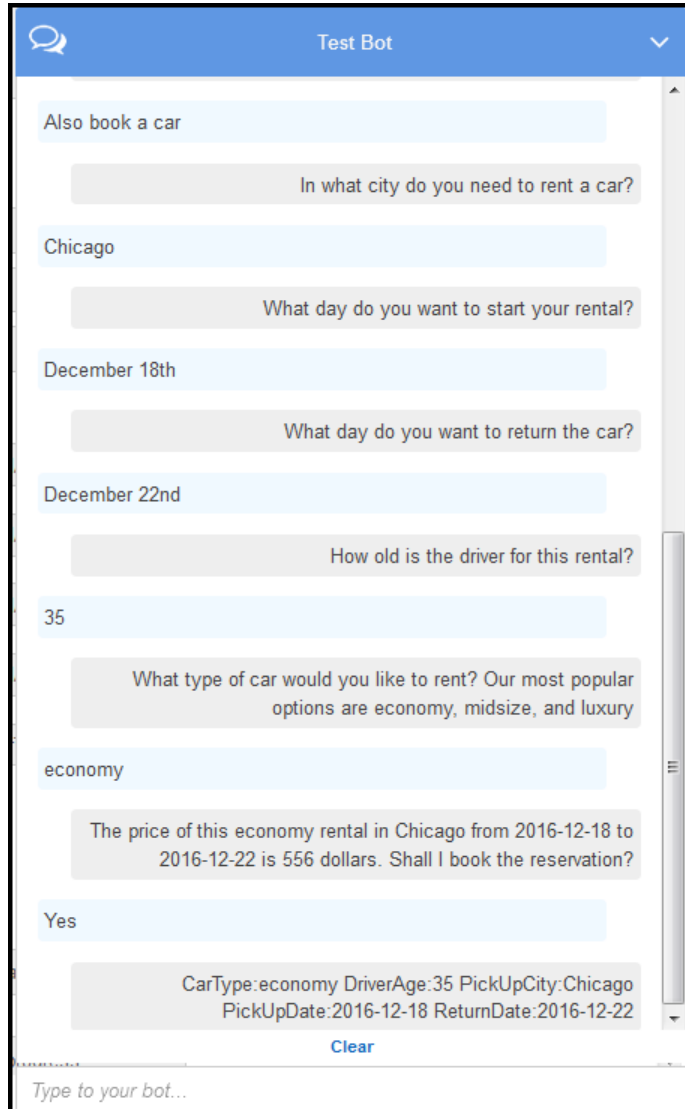
2. On the **Bots** page, choose **Create**.
3. On the **Create your Lex bot** page,
  - Choose **BookTrip** blueprint.
  - Leave the default bot name (BookTrip).
  - Choose the IAM role (**lex-exec-role**), one of the two roles you created in the preceding section.
4. Choose **Create**. The console sends a series of requests to Amazon Lex to create the bot. Note the following:
5. The console shows the BookTrip bot. On the **Editor** tab, review the details of the preconfigured intents (BookCar and BookHotel).
6. Test the bot in the test window. Use the following to engage in a test conversation with your bot:



From the initial user input ("Book a hotel"), Amazon Lex infers the intent (BookHotel). The bot then uses the prompts preconfigured in this intent to elicit slot data from the user. After user provide all of the slot data, Amazon Lex returns a response back to the client with a message that includes all the user input as a message. The client displays the message in the response as shown.

```
CheckInDate:2016-12-18 Location:Chicago Nights:4 RoomType:queen
```

Now you continue the conversation and try to book a car.



Note that,

- There is no user data validation at this time. For example, you can provide any city to book a hotel.
- You are providing some of the same information again (destination, pick-up city, pick-up date, and return date) to book a car. In a dynamic conversation, your bot should initialize some of this information based on prior input user provided for booking hotel.

In this next section, you create a Lambda function to do some of the user data validation, and initialization using cross-intent information sharing via session attributes. Then you update the intent configuration by adding the Lambda function as code hook to perform initialization/validation of user input and fulfill intent.

Next Step

[Step 4: Create a Lambda function \(p. 78\)](#)

## Step 4: Create a Lambda function

In this section you create a Lambda function using a blueprint (**lex-book-trip-python**) provided in the Amazon Lex console. You also test the Lambda function by invoking it using sample event data provided by the console.

This Lambda function is written in Node.js.

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.
2. Choose **Create a Lambda function**.
3. On **Select blueprint**, type `lex` to find the blueprint, choose the `lex-book-trip-python` blueprint.
4. Configure the Lambda function as follows and then choose **Create Function**.
  - Type a Lambda function name (`BookTripCodeHook`).
  - For the IAM role, choose the **Choose an existing role** and then select `lambda-exec-role-for-lex-get-started` from the **Existing role** list.
  - Leave other default values.
5. Test the Lambda function. You invoke the Lambda function twice, using sample data for both booking a car and booking a hotel.
  - a. Choose **Actions, Configure test event**.
  - b. Choose **Lex-Book Hotel (preview)** from the **Sample event template** list.

This sample event matches the Amazon Lex request/response model. For more information, see [Using Lambda Functions \(p. 54\)](#).
  - c. Choose **Save and test**.
  - d. Verify that the Lambda function successfully executed. The response in this case matches the Amazon Lex response model.
  - e. Repeat the step. This time you choose the **Lex-Book Car (preview)** from the **Sample event template** list. The Lambda function processes the car reservation.

Next Step

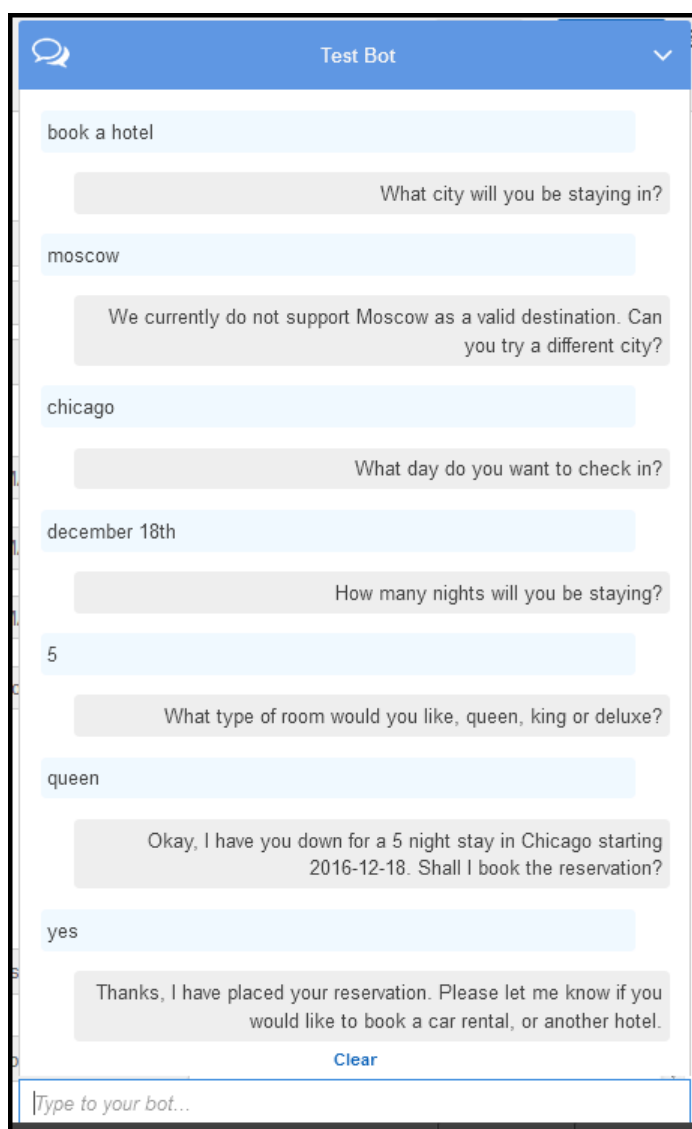
[Step 5: Add the Lambda Function as a Code Hook \(p. 78\)](#)

## Step 5: Add the Lambda Function as a Code Hook

In this section, you update the configurations of both the `BookCar` and `BookHotel` intents by adding the Lambda function as a code hook for initialization/validation and fulfillment activities. Make sure you choose the `$LATEST` version of the intents because you can only update the `$LATEST` version of your Amazon Lex resources.

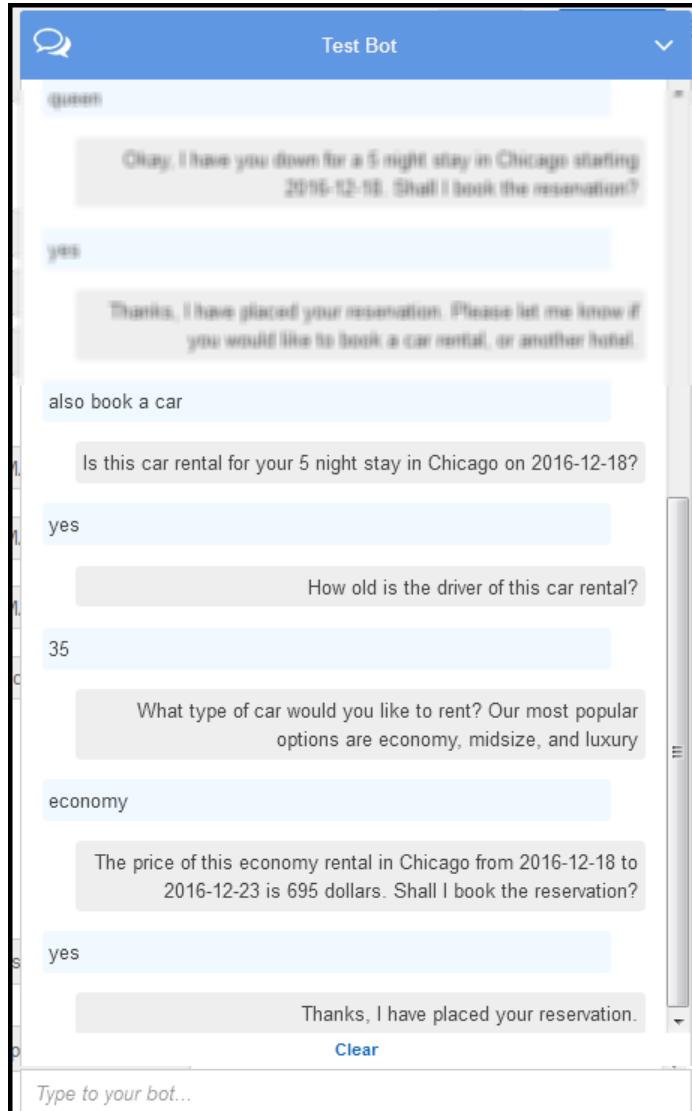
1. In the Amazon Lex console, choose the **BookTrip** bot.
2. On the **Editor** tab, choose the **BookHotel** intent. Update the intent configuration as follows:
  - a. Make sure the intent version (next to the intent name) is `$LATEST`.
  - b. Add the Lambda function as an initialization and validation code hook as follows:
    - In **Options**, choose **Initialization and validation code hook**.
    - Choose your Lambda function from the list.

- c. Add the Lambda function as a fulfillment code hook as follows:
  - In **Fulfillment**, choose **AWS Lambda function**.
  - Choose your Lambda function from the list.
  - Choose **Goodbye message** and type a message.
- d. Choose **Save**.
3. On the **Editor** tab, choose the BookCar intent. Follow the preceding step to add your Lambda function as validation and fulfillment code hook.
4. Choose **Build**. The console sends a series of requests to Amazon Lex to save the configurations.
5. Test the bot. Now that you have a Lambda function performing the initialization, user data validation and fulfillment, you can see the difference in the user interaction.



For more information about the data flow from the client (console) to Amazon Lex, and from Amazon Lex to the Lambda function, see [Data Flow: Book Hotel Intent \(p. 81\)](#).

6. Continue the conversation and book a car as shown following:



When you choose to book a car, the client (console) sends a request to Amazon Lex that includes the session attributes (from the previous conversation, BookHotel). Amazon Lex passes this information to the Lambda function, which then initializes (that is, it prepopulates) some of the BookCar slot data (that is, PickupDate, ReturnDate, and PickupCity).

**Note**

This illustrates how session attributes can be used to maintain context across intents. The console client provides the **Clear** link in the test window that a user can use to clear any prior session attributes.

For more information about the data flow from the client (console) to Amazon Lex, and from Amazon Lex to the Lambda function, see [Data Flow: Book Car Intent \(p. 91\)](#).

## Details of Information Flow

In this exercise, you engaged in conversation with the Amazon Lex BookTrip bot using the client provided in the Amazon Lex console test window. For each user input, several things happen. This section explains the information flow in detail including:

1. The requests that the client sends to Amazon Lex
2. How Amazon Lex invokes the Lambda function
  - What information Amazon Lex provides as the event parameter
  - What the Lambda function does with the incoming event and what it returns in response back to Amazon Lex
3. What Amazon Lex does with the response it receives from the Lambda function and what it returns back to the client.

In this example, both the bot intents (BookHotel and BookCar) are configured to invoke a Lambda function, as a code hook, for user data initialization/validation and fulfillment activities. Therefore, Amazon Lex invokes the Lambda function for each user input. In addition, Amazon Lex also invokes the Lambda function to fulfill the intent after the user provides all of the slot data for the intent.

Before you go into the details, note the following:

- The test window client provided in the console uses the [PostText \(p. 113\)](#) runtime API to communicate with Amazon Lex. This API describes request/response format of information exchange between the client and Amazon Lex.
- [Lambda Function Input Event and Response Format \(p. 54\)](#) describes the format of information exchange between Amazon Lex and the Lambda function.

### Topics

- [Data Flow: Book Hotel Intent \(p. 81\)](#)
- [Data Flow: Book Car Intent \(p. 91\)](#)

## Data Flow: Book Hotel Intent

This section explains details of what happens after each user input.

1. User: "book a hotel"
  - a. The client (console) sends the following [PostText \(p. 113\)](#) request to Amazon Lex:

```
POST /bot/BookTrip/alias/$LATEST/  
user/wch89kjqcpkds8seny7dly5x3otq68j3/text  
"Content-Type": "application/json"  
"Content-Encoding": "amz-1.0"  
  
{  
  "inputText": "book a hotel",  
  "sessionAttributes": {}  
}
```

Both the request URI and the body provides information to Amazon Lex:

- Request URI – Provides bot name (BookTrip), bot alias (\$LATEST) and the user name (???). The trailing text indicates that it is a PostText API request (and not PostContent).
  - Request body – Includes the user input (inputText) and empty sessionAttributes. Initially, this is an empty object and the Lambda function first sets the session attributes.
- b. From the inputText, Amazon Lex detects the intent (BookHotel). This intent is configured with a Lambda function as a code hook for user data initialization/validation. Therefore, Amazon Lex invokes that Lambda function by passing the following information as the event parameter (see [Input Event Format \(p. 54\)](#)):

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "wch89kjgcpkds8seny7dly5x3otq68j3",
  "sessionAttributes": {
  },
  "bot": {
    "name": "BookTrip",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "BookHotel",
    "slots": {
      "RoomType": null,
      "CheckInDate": null,
      "Nights": null,
      "Location": null
    },
    "confirmationStatus": "None"
  }
}
```

In addition to the information sent by the client, Amazon Lex also includes the following additional data:

- messageVersion – Currently Amazon Lex supports only the 1.0 version.
  - invocationSource – Indicates the purpose of Lambda function invocation. In this case, it is to perform user data initialization and validation (at this time Amazon Lex knows that the user has not provided all the slot data to fulfill the intent).
  - currentIntent – All of the slot values are set to null.
- c. At this time, all the slot values are null. There is nothing for the Lambda function to validate. The Lambda function returns the following response to Amazon Lex. For information about response format, see [Response Format \(p. 57\)](#).

```
{
  "sessionAttributes": {
    "currentReservation": "{\"ReservationType\": \"Hotel\", \"Location\": null, \"RoomType\": null, \"CheckInDate\": null, \"Nights\": null}"
  },
  "dialogAction": {
    "type": "Delegate",
  }
}
```

```
    "slots":{  
      "RoomType":null,  
      "CheckInDate":null,  
      "Nights":null,  
      "Location":null  
    }  
  }  
}
```

### Note

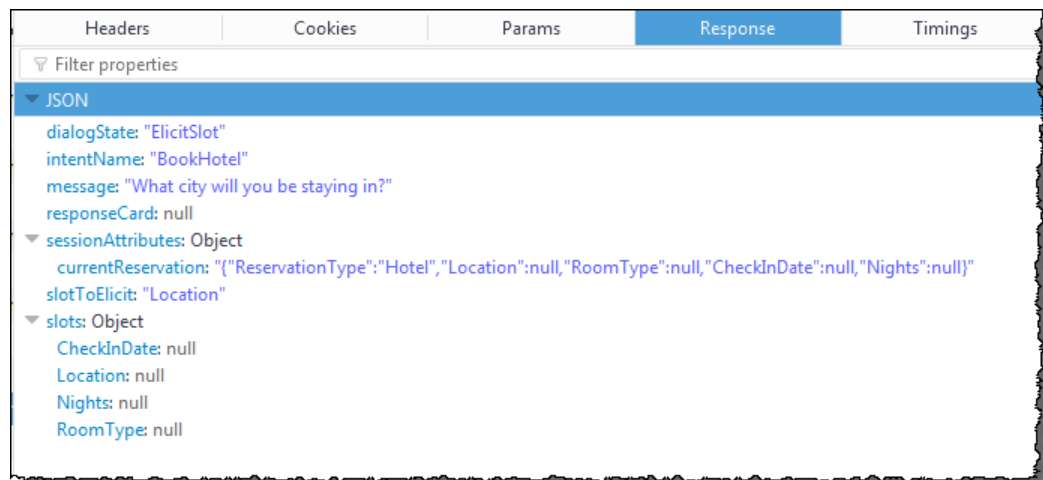
- `currentReservation` – The Lambda function includes this session attribute. Its value is a copy of the current slot information and the reservation type.

Only the Lambda function and the client can update these session attributes. Amazon Lex simply passes these values.

- `dialogAction.type` – By setting this value to `Delegate`, the Lambda function delegates the responsibility for the next course of action to Amazon Lex.

If the Lambda function detected anything in the user data validation, it instructs Amazon Lex what to do next.

- d. As per the `dialogAction.type`, Amazon Lex decides the next course of action—elicit data from the user for the `Location` slot. It selects one of the prompt messages ("What city will you be staying in?") for this slot, according to the intent configuration, and then sends the following response to the user:



The session attributes are passed to the client.

The client reads the response and then displays the message: "What city will you be staying in?"

2. User: "Moscow"

- a. The client sends the following `PostText` request to Amazon Lex (line breaks added for readability):

```
POST /bot/BookTrip/alias/$LATEST/  
user/wch89kjqqpkds8seny7dly5x3otq68j3/text  
"Content-Type": "application/json"
```



```
"Content-Encoding": "amz-1.0"

{
  "inputText": "Moscow",
  "sessionAttributes": {
    "currentReservation": "{ \"ReservationType\": \"Hotel\",
                          \"Location\": null,
                          \"RoomType\": null,
                          \"CheckInDate\": null,
                          \"Nights\": null }"
  }
}
```

In addition to the `inputText`, the client includes the same `currentReservation` session attributes it received.

- b. Amazon Lex first interprets the `inputText` in the context of the current intent (the service remembers that it had asked the specific user for information about `Location` slot). It updates the slot value for the current intent and invokes the Lambda function using the following event:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "wch89kjqcpkds8seny7dly5x3otq68j3",
  "sessionAttributes": {
    "currentReservation": "{ \"ReservationType\": \"Hotel\",
                          \"Location\": null, \"RoomType\": null, \"CheckInDate\": null, \"Nights\"
                          \": null }"
  },
  "bot": {
    "name": "BookTrip",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "BookHotel",
    "slots": {
      "RoomType": null,
      "CheckInDate": null,
      "Nights": null,
      "Location": "Moscow"
    }
  },
  "confirmationStatus": "None"
}
```

### Note

- `invocationSource` continues to be `DialogCodeHook`. In this step, we are just validating user data.
  - Amazon Lex is just passing the session attribute to the Lambda function.
  - For `currentIntent.slots`, Amazon Lex has updated the `Location` slot to `Moscow`.
- c. The Lambda function performs the user data validation and determines that `Moscow` is an invalid location.

**Note**

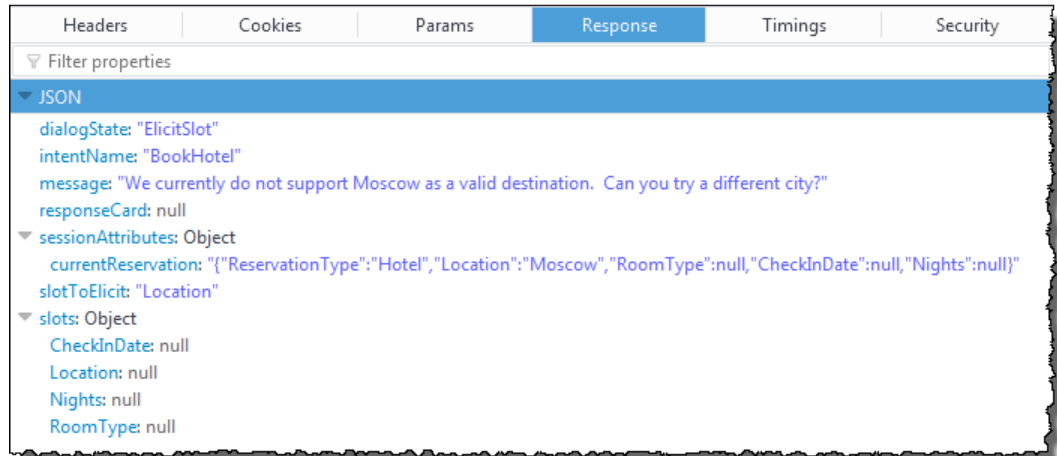
The Lambda function in this exercise has a simple list of valid cities and `MOSCOW` is not on the list. In a production application, you might use a back-end database to get this information.

It resets the slot value back to null and directs Amazon Lex to prompt the user again for another value by sending the following response:

```
{
  "sessionAttributes": {
    "currentReservation": "{\"ReservationType\": \"Hotel\",
    \"Location\": \"Moscow\", \"RoomType\": null, \"CheckInDate\": null,
    \"Nights\": null}"
  },
  "dialogAction": {
    "type": "ElicitSlot",
    "intentName": "BookHotel",
    "slots": {
      "RoomType": null,
      "CheckInDate": null,
      "Nights": null,
      "Location": null
    },
    "slotToElicit": "Location",
    "message": {
      "contentType": "PlainText",
      "content": "We currently do not support Moscow as a valid
      destination. Can you try a different city?"
    }
  }
}
```

**Note**

- `currentIntent.slots.Location` is reset to null.
  - `dialogAction.type` is set to `ElicitSlot`, which directs Amazon Lex to prompt the user again by providing the following:
    - `dialogAction.slotToElicit` – slot for which to elicit data from the user.
    - `dialogAction.message` – a message to convey to the user.
- d. Amazon Lex notices the `dialogAction.type` and passes the information to the client in the following response:



The client simply displays the message: "We currently do not support Moscow as a valid destination. Can you try a different city?"

3. User: "Chicago"
  - a. The client sends the following `PostText` request to Amazon Lex:

```
POST /bot/BookTrip/alias/$LATEST/
user/wch89kjqcpkds8seny7dly5x3otq68j3/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "Chicago",
  "sessionAttributes": {
    "currentReservation": {"ReservationType": "Hotel",
                          "Location": "Moscow",
                          "RoomType": null,
                          "CheckInDate": null,
                          "Nights": null}
  }
}
```

- b. Amazon Lex knows the context, that it was eliciting data for the `Location` slot. In this context, it knows the `inputText` value is for the `Location` slot. It then invokes the Lambda function by sending the following event:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "wch89kjqcpkds8seny7dly5x3otq68j3",
  "sessionAttributes": {
    "currentReservation": {"ReservationType": "Hotel",
                          "Location": "Moscow",
                          "RoomType": null,
                          "CheckInDate": null,
                          "Nights": null}
  },
  "bot": {
    "name": "BookTrip",
    "alias": null,
  }
}
```

```
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "BookHotel",
    "slots": {
      "RoomType": null,
      "CheckInDate": null,
      "Nights": null,
      "Location": "Chicago"
    },
    "confirmationStatus": "None"
  }
}
```

Amazon Lex updated the `currentIntent.slots` by setting the `Location` slot to `Chicago`.

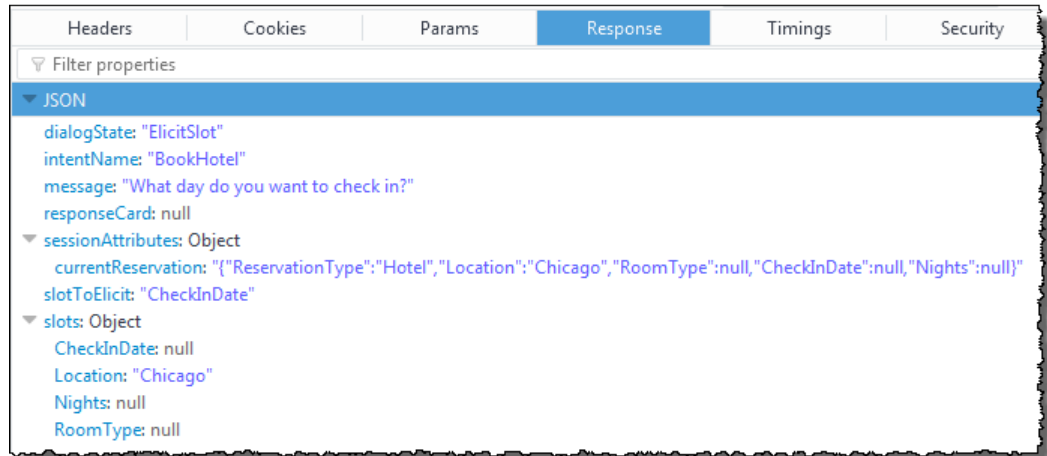
- c. According to the `invocationSource` value of `DialogCodeHook`, the Lambda function performs user data validation. It recognizes `Chicago` as a valid slot value, updates the session attribute accordingly, and then returns the following response to Amazon Lex.

```
{
  "sessionAttributes": {
    "currentReservation": "{\"ReservationType\": \"Hotel\",
    \"Location\": \"Chicago\", \"RoomType\": null, \"CheckInDate\": null,
    \"Nights\": null}"
  },
  "dialogAction": {
    "type": "Delegate",
    "slots": {
      "RoomType": null,
      "CheckInDate": null,
      "Nights": null,
      "Location": "Chicago"
    }
  }
}
```

### Note

- `currentReservation` – The Lambda function updates this session attribute by setting the `Location` to `Chicago`.
- `dialogAction.type` – Is set to `Delegate`. User data was valid, and the Lambda function directs Amazon Lex to choose the next course of action.

- d. According to `dialogAction.type`, Amazon Lex chooses the next course of action. Amazon Lex knows that it needs more slot data and picks the next unfilled slot (`CheckInDate`) with the highest priority according to the intent configuration. It selects one of the prompt messages ("What day do you want to check in?") for this slot according to the intent configuration and then sends the following response back to the client:



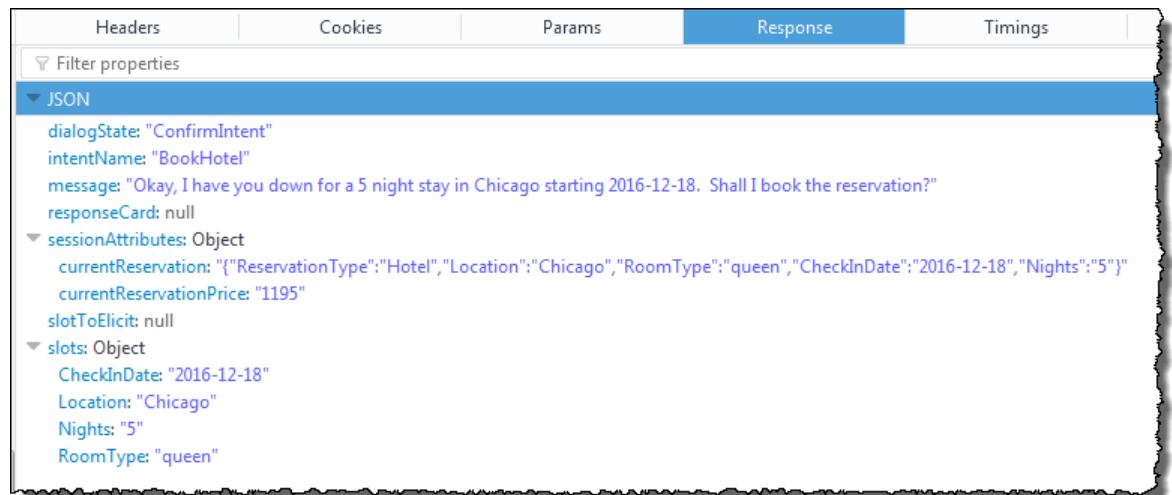
The client displays the message: "What day do you want to check in?"

- The user interaction continues—the user provides data, the Lambda function validates data, and then delegates the next course of action to Amazon Lex. Eventually the user provides all of the slot data, the Lambda function validates all of the user input, and then Amazon Lex recognizes it has all the slot data.

**Note**

In this exercise, after the user provides all of the slot data, the Lambda function computes the price of the hotel reservation and returns it as another session attribute (`currentReservationPrice`).

At this point, the intent is ready to be fulfilled, but the `BookHotel` intent is configured with a confirmation prompt requiring user confirmation before Amazon Lex can fulfill the intent. Therefore, Amazon Lex sends the following message to the client requesting confirmation before booking the hotel:



The client display the message: "Okay, I have you down for a 5 night in Chicago starting 2016-12-18. Shall I book the reservation?"

- User: "yes"

- a. The client sends the following `PostText` request to Amazon Lex:

```
POST /bot/BookTrip/alias/$LATEST/
user/wch89kjgcpkds8seny7dly5x3otq68j3/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "Yes",
  "sessionAttributes": {
    "currentReservation": "{ \"ReservationType\": \"Hotel\",
                          \"Location\": \"Chicago\",
                          \"RoomType\": \"queen\",
                          \"CheckInDate\": \"2016-12-18\",
                          \"Nights\": \"5\" }",
    "currentReservationPrice": "1195"
  }
}
```

- b. Amazon Lex interprets the `inputText` in the context of confirming the current intent. Amazon Lex understands that the user wants to proceed with the reservation. This time Amazon Lex invokes the Lambda function to fulfill the intent by sending the following event. By setting the `invocationSource` to `FulfillmentCodeHook` in the event, it sends to the Lambda function. Amazon Lex also sets the `confirmationStatus` to `Confirmed`.

```
{
  "messageVersion": "1.0",
  "invocationSource": "FulfillmentCodeHook",
  "userId": "wch89kjgcpkds8seny7dly5x3otq68j3",
  "sessionAttributes": {
    "currentReservation": "{ \"ReservationType\": \"Hotel\",
                          \"Location\": \"Chicago\",
                          \"RoomType\": \"queen\",
                          \"CheckInDate\":
                          \"2016-12-18\",
                          \"Nights\": \"4\" }",
    "currentReservationPrice": "956"
  },
  "bot": {
    "name": "BookTrip",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "BookHotel",
    "slots": {
      "RoomType": "queen",
      "CheckInDate": "2016-12-18",
      "Nights": "4",
      "Location": "Chicago"
    }
  },
  "confirmationStatus": "Confirmed"
}
```

### Note

- `invocationSource` – This time, Amazon Lex set this value to `FulfillmentCodeHook`, directing the Lambda function to fulfill the intent.

- `confirmationStatus` – Is set to `Confirmed`.
- c. This time, the Lambda function fulfills the `BookHotel` intent, Amazon Lex completes the reservation, and then it returns the following response:

```
{
  "sessionAttributes": {
    "lastConfirmedReservation": "{\"ReservationType\":\"Hotel\", \"Location\":\"Chicago\", \"RoomType\":\"queen\", \"CheckInDate\":\"2016-12-18\", \"Nights\":\"4\"}"
  },
  "dialogAction": {
    "type": "Close",
    "fulfillmentState": "Fulfilled",
    "message": {
      "contentType": "PlainText",
      "content": "Thanks, I have placed your reservation. Please let me know if you would like to book a car rental, or another hotel."
    }
  }
}
```

### Note

- `lastConfirmedReservation` – Is a new session attribute that the Lambda function added (instead of the `currentReservation`, `currentReservationPrice`).
- `dialogAction.type` – The Lambda function sets this value to `Close`, indicating that Amazon Lex to not expect a user response.
- `dialogAction.fulfillmentState` – Is set to `Fulfilled` and includes an appropriate message to convey to the user.

- d. Amazon Lex reviews the `fulfillmentState` and sends the following response to the client:

Headers	Cookies	Params	Response	Timings
Filter properties				
JSON				
dialogState: "Fulfilled"				
intentName: "BookHotel"				
message: "Thanks, I have placed your reservation. Please let me know if you would like to book a car rental, or another hotel."				
responseCard: null				
sessionAttributes: Object				
lastConfirmedReservation: "{\"ReservationType\":\"Hotel\", \"Location\":\"Chicago\", \"RoomType\":\"queen\", \"CheckInDate\":\"2016-12-18\", \"Nights\":\"5\"}"				
slotToElicit: null				
slots: Object				
CheckInDate: "2016-12-18"				
Location: "Chicago"				
Nights: "5"				
RoomType: "queen"				

### Note

- `dialogState` – Amazon Lex sets this value to `Fulfilled`.

- `message` – Is the same message that the Lambda function provided.

The client displays the message.

## Data Flow: Book Car Intent

The BookTrip bot in this exercise supports two intents (BookHotel and BookCar). After booking a hotel, the user can continue the conversation to book a car. As long as the session hasn't timed out, in each subsequent request the client continues to send the session attributes (in this example, the `lastConfirmedReservation`). The Lambda function can use this information to initialize slot data for the BookCar intent. This is an example of how you can use session attributes in cross-intent data sharing.

More specifically, when the user chooses the BookCar intent, the Lambda function uses relevant information in the session attribute to prepopulate slots (PickUpDate, ReturnDate, and PickUpCity) for the BookCar intent.

### Note

The Amazon Lex console provides the **Clear** link that you can use to clear any prior session attributes.

Follow the steps in this procedure to continue the conversation.

1. User: "also book a car"
  - a. The client sends the following `PostText` request to Amazon Lex.

```
POST /bot/BookTrip/alias/$LATEST/
user/wch89kjqcpkds8seny7dly5x3otq68j3/text
"Content-Type": "application/json"
"Content-Encoding": "amz-1.0"

{
  "inputText": "also book a car",
  "sessionAttributes": {
    "lastConfirmedReservation": "{ \"ReservationType\": \"Hotel\",
                                \"Location\": \"Chicago\",
                                \"RoomType\": \"queen\",
                                \"CheckInDate\": \"2016-12-18\",
                                \"Nights\": \"5\" }"
  }
}
```

The client includes the `lastConfirmedReservation` session attribute.

- b. Amazon Lex detects the intent (BookCar) from the `inputText`. This intent is also configured to invoke the Lambda function to perform the initialization and validation of the user data. Amazon Lex invokes the Lambda function with the following event:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "wch89kjqcpkds8seny7dly5x3otq68j3",
  "sessionAttributes": {
    "lastConfirmedReservation": "{ \"ReservationType\": \"Hotel
                                \", \"Location\": \"Chicago\", \"RoomType\": \"queen\", \"CheckInDate\":
                                \"2016-12-18\", \"Nights\": \"4\" }"
  }
}
```



```
    },
    "bot": {
      "name": "BookTrip",
      "alias": null,
      "version": "$LATEST"
    },
    "outputDialogMode": "Text",
    "currentIntent": {
      "name": "BookCar",
      "slots": {
        "PickUpDate": null,
        "ReturnDate": null,
        "DriverAge": null,
        "CarType": null,
        "PickUpCity": null
      }
    },
    "confirmationStatus": "None"
  }
}
```

### Note

- `messageVersion` – Currently Amazon Lex supports the 1.0 version only.
  - `invocationSource` – Indicates the purpose of invocation is to perform initialization and user data validation.
  - `currentIntent` – It includes the intent name and the slots. At this time, all slot values are null.
- c. The Lambda function notices all null slot values with nothing to validate. However, it uses session attributes to initialize some of the slot values (`PickUpDate`, `ReturnDate`, and `PickUpCity`), and then returns the following response:

```
{
  "sessionAttributes": {
    "lastConfirmedReservation": "{ \"ReservationType\": \"Hotel\", \"Location\": \"Chicago\", \"RoomType\": \"queen\", \"CheckInDate\": \"2016-12-18\", \"Nights\": \"4\" }",
    "currentReservation": "{ \"ReservationType\": \"Car\", \"PickUpCity\": null, \"PickUpDate\": null, \"ReturnDate\": null, \"CarType\": null }",
    "confirmationContext": "AutoPopulate"
  },
  "dialogAction": {
    "type": "ConfirmIntent",
    "intentName": "BookCar",
    "slots": {
      "PickUpCity": "Chicago",
      "PickUpDate": "2016-12-18",
      "ReturnDate": "2016-12-22",
      "CarType": null,
      "DriverAge": null
    }
  },
  "message": {
    "contentType": "PlainText",
    "content": "Is this car rental for your 4 night stay in Chicago on 2016-12-18?"
  }
}
```

```
}  
  }  
}
```

### Note

- In addition to the `lastConfirmedReservation`, the Lambda function includes more session attributes (`currentReservation` and `confirmationContext`).
- `dialogAction.type` is set to `ConfirmIntent`, which informs Amazon Lex that a yes, no reply is expected from the user (the `confirmationContext` set to `AutoPopulate`, the Lambda function knows that the yes/no user reply is to obtain user confirmation of the initialization the Lambda function performed (auto populated slot data).

The Lambda function also includes in the response an informative message in the `dialogAction.message` for Amazon Lex to return to the client.

### Note

The term `ConfirmIntent` (value of the `dialogAction.type`) is not related to any bot intent. In the example, Lambda function uses this term to direct Amazon Lex to get a yes/no reply from the user.

- d. According to the `dialogAction.type`, Amazon Lex returns the following response to the client:

```
{  
  "dialogState": "ConfirmIntent",  
  "intentName": "BookCar",  
  "message": "Is this car rental for your 5 night stay in Chicago on 2016-12-18?",  
  "responseCard": null,  
  "sessionAttributes": {  
    "confirmationContext": "AutoPopulate",  
    "currentReservation": {"ReservationType": "Car", "PickUpCity": null, "PickUpDate": null, "ReturnDate": null, "CarType": null},  
    "lastConfirmedReservation": {"ReservationType": "Hotel", "Location": "Chicago", "RoomType": "queen", "CheckInDate": "2016-12-18", "Nights": "5"},  
    "slotToElicit": null  
  },  
  "slots": {  
    "CarType": null,  
    "DriverAge": null,  
    "PickUpCity": "Chicago",  
    "PickUpDate": "2016-12-18",  
    "ReturnDate": "2016-12-23"  
  }  
}
```

The client displays the message: "Is this car rental for your 5 night stay in Chicago on 2016-12-18?"

2. User: "yes"
  - a. The client sends the following `PostText` request to Amazon Lex.

```
POST /bot/BookTrip/alias/$LATEST/  
user/wch89kjqcpkds8seny7dly5x3otq68j3/text  
"Content-Type": "application/json"  
"Content-Encoding": "amz-1.0"
```

```
{
  "inputText": "yes",
  "sessionAttributes": {
    "confirmationContext": "AutoPopulate",
    "currentReservation": {
      "ReservationType": "Car",
      "PickUpCity": null,
      "PickUpDate": null,
      "ReturnDate": null,
      "CarType": null
    },
    "lastConfirmedReservation": {
      "ReservationType": "Hotel",
      "Location": "Chicago",
      "RoomType": "queen",
      "CheckInDate": "2016-12-18",
      "Nights": "5"
    }
  }
}
```

- b. Amazon Lex reads the `inputText` and it knows the context (asked the user to confirm the auto population). Amazon Lex invokes the Lambda function by sending the following event:

```
{
  "messageVersion": "1.0",
  "invocationSource": "DialogCodeHook",
  "userId": "wch89kjqcpgds8seny7dly5x3otq68j3",
  "sessionAttributes": {
    "confirmationContext": "AutoPopulate",
    "currentReservation": {
      "ReservationType": "Car",
      "PickUpCity": null, "PickUpDate": null, "ReturnDate": null, "CarType": null
    },
    "lastConfirmedReservation": {
      "ReservationType": "Hotel",
      "Location": "Chicago", "RoomType": "queen", "CheckInDate": "2016-12-18", "Nights": "4"
    }
  },
  "bot": {
    "name": "BookTrip",
    "alias": null,
    "version": "$LATEST"
  },
  "outputDialogMode": "Text",
  "currentIntent": {
    "name": "BookCar",
    "slots": {
      "PickUpDate": "2016-12-18",
      "ReturnDate": "2016-12-22",
      "DriverAge": null,
      "CarType": null,
      "PickUpCity": "Chicago"
    }
  },
  "confirmationStatus": "Confirmed"
}
```

Because the user replied Yes, Amazon Lex sets the `confirmationStatus` to `Confirmed`.

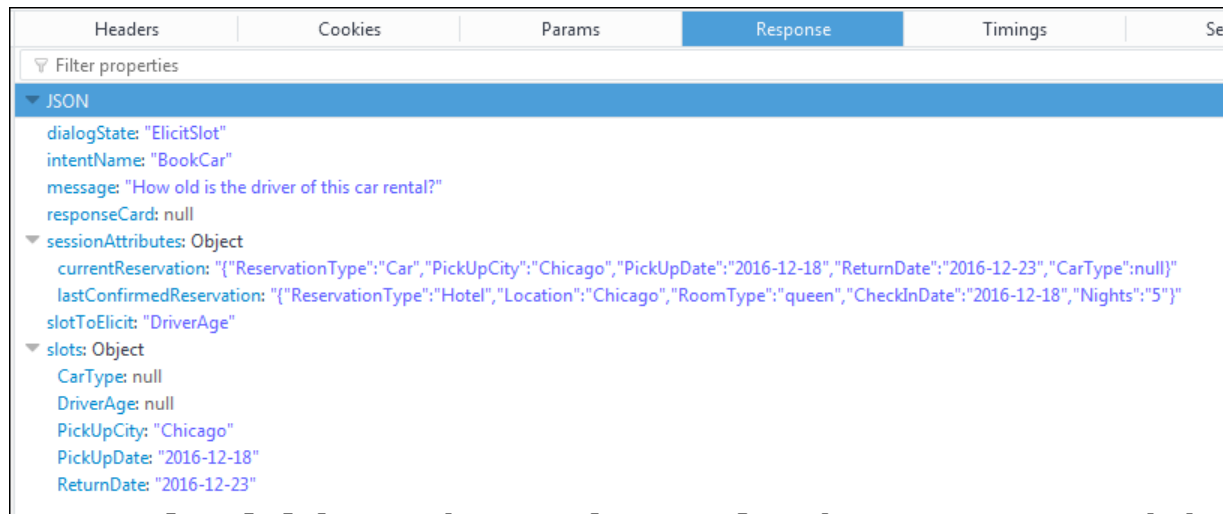
- c. From the `confirmationStatus`, the Lambda function knows that the prepopulated values are correct. The Lambda function does the following:
- Updates the `currentReservation` session attribute to slot value it had prepopulated.

- Sets the `dialogAction.type` to `ElicitSlot`
- Sets the `slotToElicit` value to `DriverAge`.

The following response is sent:

```
{
  "sessionAttributes": {
    "currentReservation": "{\"ReservationType\":\"Car\",
    \\\"PickUpCity\\\":\\\"Chicago\\\",\\\"PickUpDate\\\":\\\"2016-12-18\\\",\\\"ReturnDate
    \\\":\\\"2016-12-22\\\",\\\"CarType\\\":null}\",
    "lastConfirmedReservation": "{\"ReservationType\":\"Hotel
    \\\",\\\"Location\\\":\\\"Chicago\\\",\\\"RoomType\\\":\\\"queen\\\",\\\"CheckInDate\\\":
    \\\"2016-12-18\\\",\\\"Nights\\\":\\\"4\\\"}"
  },
  "dialogAction": {
    "type": "ElicitSlot",
    "intentName": "BookCar",
    "slots": {
      "PickUpDate": "2016-12-18",
      "ReturnDate": "2016-12-22",
      "DriverAge": null,
      "CarType": null,
      "PickUpCity": "Chicago"
    },
    "slotToElicit": "DriverAge",
    "message": {
      "contentType": "PlainText",
      "content": "How old is the driver of this car rental?"
    }
  }
}
```

- d. Amazon Lex returns following response:



The client displays the message "How old is the driver of this car rental?" and the conversation continues.

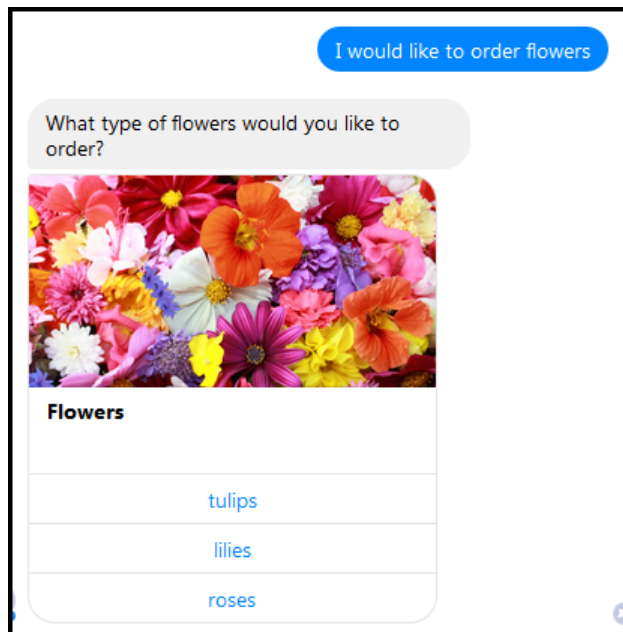
## Example: Using a Response Card

*This is prerelease documentation for a service in preview release. It is subject to change.*

In this exercise, you extend the Getting Started Exercise 1 by adding a response card. You create a bot that supports the OrderFlowers intent and then update the intent by adding a response card for the FlowerType slot. In addition to the following prompt for the FlowerType slot, the user can choose the type of flowers from the response card:

What type of flowers would you like to order?

The response card is shown following:



The bot user can either type the text or choose from the list of flower types. This response card is configured with an image, which appears in the client as shown. For more information about response cards, see [Response Cards \(p. 12\)](#).

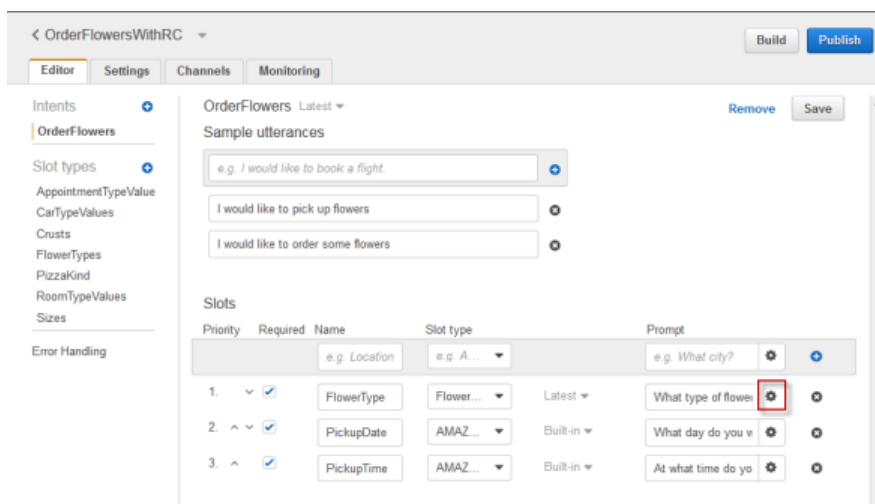
### Note

At this time, response cards are supported only with the Facebook Messenger platform. Therefore, you need to deploy your bot on the Facebook Messenger platform to use this feature. You test the bot on the Facebook message to see the response card.

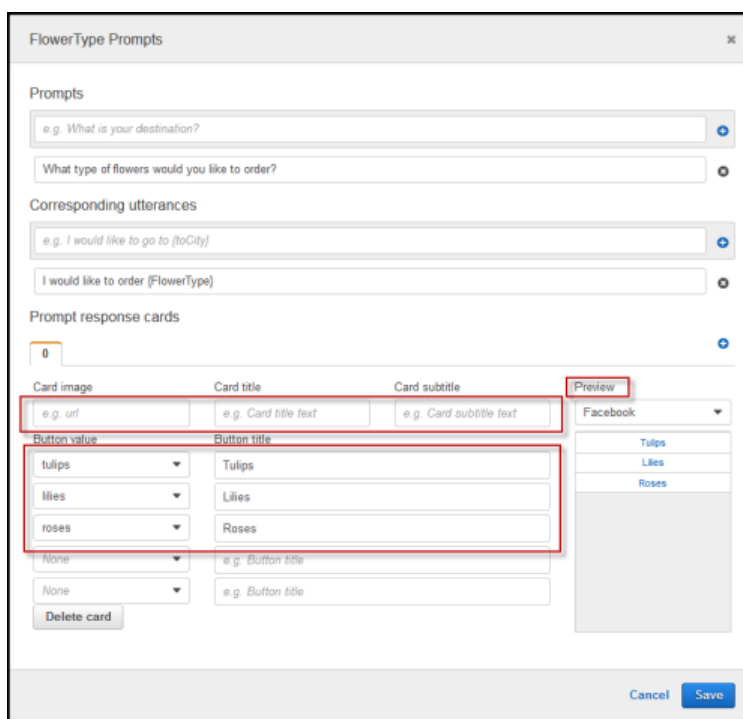
Do the following to create and test a bot with a response card:

1. Follow the Getting Started Exercise 1 to create and test an OrderFlowers bot. You must complete steps 1, 2, and 3. You don't need to add a Lambda function to test the response card. For instructions, see [Exercise 1: Create an Amazon Lex Bot Using a Blueprint \(p. 17\)](#).
2. Update the bot by adding the response card and then publish a version. When you publish a version, specify an alias (BETA) pointing to it.
  - a. In the Amazon Lex console, choose your bot.

- b. Choose the `OrderFlowers` intent.
- c. Choose the settings gear icon next to the "What type of flowers" **Prompt** to configure a response card for the `FlowerType`.



- d. Configure three buttons as shown in the following screen shot. You can optionally add an image to the response card, provided you have an image URL.



- e. Verify the button values in the **Preview** and then choose **Save**.
  - f. On the **Editor** tab, choose **Save** to save the intent configuration.
  - g. Choose **Build** to build the bot.
  - h. Choose **Publish** to publish a bot version. Specify BETA as an alias that points to the bot version. For information about versioning, see [Versioning and Aliases \(p. 50\)](#).
3. Deploy the bot on the Facebook Messenger platform and test the integration. For instructions, see [Integrating an Amazon Lex Bot with Facebook Messenger \(p. 61\)](#).

When you order flowers, the message window shows the response card for you to choose a flower type.

# Guidelines and Limits in Amazon Lex

---

***This is prerelease documentation for a service in preview release. It is subject to change.***

The following sections provide guidelines and limits when using Amazon Lex.

## Topics

- [General Guidelines \(p. 99\)](#)
- [Limits \(p. 101\)](#)

## General Guidelines

This section describes general guidelines when using Amazon Lex.

- Signing requests – All Amazon Lex model-building and runtime API operations in the [API Reference \(p. 105\)](#) 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*. The only difference is in the signing of the `PostContent` runtime API. Because this API starts processing speech or text sent by the client as it is being streamed, you cannot include the HTTP request body in the signature calculation, and it is therefore not required for signing.

- Note the following about how Amazon Lex captures slot values from user utterances:

Amazon Lex uses the enumeration values you provide in a slot type definition to train its machine learning models. Suppose you define an intent called `GetPredictionIntent` with the following sample utterance:

```
"Tell me the prediction for {Sign}"
```

Where `{Sign}` is a slot of custom type `ZodiacSign`. `ZodiacSign` has 12 enumeration values (`Aries` through `Pisces`). From the user utterance "Tell me the prediction for ..." Amazon Lex understands that what follows is a zodiac sign.



If the user says "Tell me the prediction for earth", Amazon Lex infers that "earth" is possibly another `ZodiacSign` and passes it to your fulfillment activity. Therefore, your fulfillment activity must validate the slot values.

Amazon Lex ignores non-alphanumeric characters (punctuation marks such as question mark, period, and hyphen) in slot values. For examples, when a users types "US-EAST-1", your Lambda function might recognize "us east 1" as the slot value.

- Amazon Lex does not support `AMAZON.LITERAL` built-in slot type that Alexa Skills Kit supports. However, Amazon Lex supports creating custom slot types that you can use to implement this functionality. As mentioned in the previous bullet, you can capture values outside the custom slot type definition. You can add more and diverse enumeration values to boost the automatic speech recognition (ASR) and natural language understanding (NLU) accuracy.
- Amazon Lex does not support the built-in intents `AMAZON.YesIntent` and `AMAZON.NoIntent` that Alexa Skills Kit supports. However, Amazon Lex supports the `ConfirmIntent` dialog action in the code hook interface. Using this, you can implement the same functionality.
- Providing confusable training data in your bot reduces Amazon Lex's ability 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 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 can't pick an intent with a high degree of confidence.

Consider another example where you define a custom intent for getting a confirmation from the user (for example, `MyCustomConfirmationIntent`) and configure the intent with the utterances `Yes` and `No`. Note that Amazon Lex also has a language model for understanding user confirmations. This can create conflicting situation. When the user responds with a `Yes`, does this mean that this is a confirmation for the ongoing intent or that the user is requesting the custom intent that you created?

In general, the sample utterances you provide should map to a specific intent and, optionally, to specific slot values.

- The runtime API operations [PostContent \(p. 106\)](#) and [PostText \(p. 113\)](#) take a user ID as the required parameter. Developers can set this to any value that meets the constraints described in the API. We recommend 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 conversation with a bot (there can be multiple users ordering pizza).
- If your client application uses Amazon Cognito for authentication, you might use the Amazon Cognito user ID as Amazon Lex user ID. Note that the Lambda function (code hook configured for your bot) must have its own authentication mechanism to identify the user on whose behalf Amazon Lex is invoking the Lambda function.

- We encourage you to define an intent that captures a user's intention to discontinue the conversation. For example, you can define an intent (NothingIntent) with sample utterances ("I don't want anything", "exit", "bye bye"), no slots, and no Lambda function configured as a code hook. This would let users gracefully close a conversation.
- Note the following about this *preview* release:
  - The Amazon Lex model building API is not available for public consumption. You can use the console to create and manage bots.

## Limits

This section describes current limits in Amazon Lex. These limits are grouped by categories.

- Amazon Lex general limits
  - Currently, Amazon Lex supports US English language. That is, Amazon Lex trains your bots to understand only US English.
  - Currently, Amazon Lex is available in `us-east-1` region.
- Amazon Lex runtime service limits - In addition to the limits described in the API reference, note the following:
  - API
    - Input speech in the [PostContent \(p. 106\)](#) can be up to 15 seconds long.
    - In both the runtime API operations [PostContent \(p. 106\)](#) and [PostText \(p. 113\)](#), the input text size can be up to 1024 Unicode characters.
    - The total size of the session attributes in a `PostContent` request and response can be up to 12 KB.
  - Currently, Amazon Lex is available in `us-east-1` region.

Region Name	Region	Endpoint	Protocol
US East (N. Virginia)	us-east-1	runtime.lex.us-east-1.amazonaws.com	HTTPS

- Model building limits

Model building refers to creating and managing bots. This includes, for example, creating/managing bots, intents, and slot types, slots, and bot channel associations. Currently, Amazon Lex supports creating and managing bots via the Amazon Lex console.

You configure prompts and statements throughout the model building API. Each of these prompts or statements can have up to five messages and each message can contain from 1 to 1000 UTF-8 characters.

- Bots

- Bot, alias, and bot channel association names are case insensitive at the time of creation. That is, if you create PizzaBot and then again try to create another pizzaBot, you will get an error. However, when accessing a resource, the resource names are case sensitive (that is, you must specify PizzaBot and not pizzaBot).

These names must be between 2 and 50 ASCII characters.

- Maximum number of versions you can publish, Amazon Lex resource types, is 100. Note that, there is no versioning for aliases.
- Within a bot, intent names and slot names must be unique (that is, you can't have an intent and a slot by the same name).
- You can create a bot that is configured to support multiple intents. If two intents have a slot by the same name, then the corresponding slot type must be the same.

For example, suppose you create a bot to support two intents (OrderPizza and OrderDrink). If both these intents have the `size` slot, then the slot type must be the same in both places.

In addition, sample utterances you provide for a slot (in one of the intents), applies to the same name slot in other intents.

- At the time of creating a bot, you specify a session timeout. Session timeout can be between one minute and one day. Five minutes is the default.

This timeout determines how long the bot can retain the context, such as current user intent and slot data.

In addition, note that after a user starts the conversation with your bot and until the session expires, Amazon Lex uses the same bot version (even if you update the bot alias to point to another version).

- When you update the `$LATEST` version of the bot, Amazon Lex terminates any in-progress user conversations with the bot (if your client application is using the `$LATEST` version of the bot).

Generally, you should not use the \$LATEST version of a bot in production because \$LATEST version can be updated. You should publish a version and use it instead.

- You can create up to five aliases for a bot.
- You can create up to 100 bots per AWS account.
- In a bot, you cannot create multiple intents that extend from the same built-in intent.
- Intents related limits
  - Intent and slot names are case insensitive at the time of creation. That is, if you create OrderPizza intent and then again try to create another orderPizza intent, you will get an error. However, when accessing these resources, the resource names are case sensitive (you must specify OrderPizza and not orderPizza).

These names must be between 1 and 100 ASCII characters.

- An intent can have up to 1,000 sample utterances (a minimum of one sample utterance is required). Each sample utterance you configure for an intent can be up to 200 UTF-8 characters long. A sample utterance:
  - Can refer to zero or more slot names.
  - Can refer to a slot name only once.

For example:

```
I want a pizza
I want a {pizzaSize} pizza
I want a {pizzaSize} {pizzaTopping} pizza
```

**Note**

pizzaSize and pizzaTopping refer to slot names (not slot types).

- Each slot can have up to 10 sample utterances. Each sample utterance must refer to the slot name exactly once. For example:

```
{pizzaSize} please
```

- You cannot provide utterances for intents that extend from built-in intents. For all other intents you must provide at least one sample utterance. Intents contain slots, but the slot level sample utterances are optional.
- You must publish a version of an intent before you can use it in a bot.

- Currently, Amazon Lex does not support slot elicitation for built-in intents. You cannot create Lambda functions to return the `ElicitSlot` directive in the response with an intent that is derived from built-in intents. For more information, see [Response Format \(p. 57\)](#).
  - The service does not support adding sample utterances to built-in intents. Similarly, you cannot add or remove slots to built-in intents.
- 
- You can create up to 1,000 intents per AWS account. You can create up to 100 slots in an intent.
- 
- Slot type related limits
    - Slot type names are case insensitive at the time of creation. That is, if you create `PizzaSize` slot type and then again try to create another `pizzaSize` slot type, you will get an error. However, when accessing these resources, the resource names are case sensitive (you must specify `PizzaSize` and not `pizzaSize`).

These names must be between 1 and 100 ASCII characters.

- Resource (bot, intent, alias, slot, slot type) names are case insensitive.

A custom slot type you create can have a maximum of 10,000 enumeration values, and each enumeration value can be up to 140 UTF-8 characters long. The enumeration values cannot contain duplicates.

- You must first publish a version of a slot type before you can use it in intent.
- For a slot type value, where appropriate, specify both upper and lower case. For example, for a slot type called Procedure, if value is MRI, specify both MRI and mir as values.
- Built-in slot types – Currently, Amazon Lex doesn't support adding enumeration values for the built-in slot types.

# API Reference

---

*This is prerelease documentation for a service in preview release. It is subject to change.*

This section provides documentation for the Amazon Lex API operations. Currently, Amazon Lex is available in the following AWS region.

Region Name	Region	Endpoint	Protocol
US East (N. Virginia)	us-east-1	runtime.lex.us-east-1.amazonaws.com	HTTPS

## Topics

- [Actions \(p. 105\)](#)
- [Data Types \(p. 117\)](#)

## Actions

The following actions are supported:

- [PostContent \(p. 106\)](#)
- [PostText \(p. 113\)](#)

## PostContent

Sends user input (text or speech) to Amazon Lex. Clients use this API to send requests to Amazon Lex at runtime. Amazon Lex interprets the user input using the machine learning model that it built for the bot.

In response, Amazon Lex returns the next message to convey to the user. Consider the following example messages:

- For a user input "I would like a pizza," Amazon Lex might return a response with a message eliciting slot data (for example, `PizzaSize`): "What size pizza would you like?".
- After the user provides all of the pizza order information, Amazon Lex might return a response with a message to get user confirmation: "Order the pizza?".
- After the user replies "Yes" to the confirmation prompt, Amazon Lex might return a conclusion statement: "Thank you, your cheese pizza has been ordered."

Not all Amazon Lex messages require a response from the user. For example, conclusion statements do not require a response. Some messages require only a yes or no response. In addition to the `message`, Amazon Lex provides additional context about the message in the response that you can use to enhance client behavior, such as displaying the appropriate client user interface. Consider the following examples:

- If the message is to elicit slot data, Amazon Lex returns the following context information:
  - `x-amz-lex-dialog-state` header set to `ElicitSlot`
  - `x-amz-lex-intent-name` header set to the intent name in the current context
  - `x-amz-lex-slot-to-elicited` header set to the slot name for which the message is eliciting information
  - `x-amz-lex-slots` header set to a map of slots configured for the intent with their current values
- If the message is a confirmation prompt, the `x-amz-lex-dialog-state` header is set to `Confirmation` and the `x-amz-lex-slot-to-elicited` header is omitted.
- If the message is a clarification prompt configured for the intent, indicating that the user intent is not understood, the `x-amz-lex-dialog-state` header is set to `ElicitIntent` and the `x-amz-lex-slot-to-elicited` header is omitted.

In addition, Amazon Lex also returns your application-specific `sessionAttributes`. For more information, see [Managing Conversation Context](#).

## Request Syntax

```
POST /bot/botName/alias/botAlias/user/userId/content HTTP/1.1
x-amz-lex-session-attributes: sessionAttributes
Content-Type: contentType
Accept: accept

inputStream
```

## URI Request Parameters

The request requires the following URI parameters.

### **accept** (p. 106)

You pass this value as the `Accept` HTTP header.

The message Amazon Lex returns in the response can be either text or speech based on the `Accept` HTTP header value in the request.

- If the value is `text/plain; charset=utf-8`, Amazon Lex returns text in the response.
- If the value begins with `audio/`, Amazon Lex returns speech in the response. Amazon Lex uses Amazon Polly to generate the speech (using the configuration you specified in the `Accept` header). For example, if you specify `audio/mpeg` as the value, Amazon Lex returns speech in the MPEG format.

The following are the accepted values:

- `audio/mpeg`
- `audio/ogg`
- `audio/pcm`
- `text/plain; charset=utf-8`
- `audio/*` (defaults to `mpeg`)

#### **botAlias** (p. 106)

Alias of the Amazon Lex bot.

#### **botName** (p. 106)

Name of the Amazon Lex bot.

#### **contentType** (p. 106)

You pass this values as the `Content-Type` HTTP header.

Indicates the audio format or text. The header value must start with one of the following prefixes:

- `audio/l16; rate=16000; channels=1`
- `audio/x-l16; sample-rate=16000; channel-count=1`
- `text/plain; charset=utf-8`
- `audio/x-cbr-opus-with-preamble; preamble-size=0; bit-rate=1; frame-size-milliseconds=1.1`

#### **sessionAttributes** (p. 106)

You pass this value in the `x-amz-lex-session-attributes` HTTP header. The value must be map (keys and values must be strings) that is JSON serialized and then base64 encoded.

A session represents dialog between a user and Amazon Lex. At runtime, a client application can pass contextual information, in the request to Amazon Lex. For example,

- You might use session attributes to track the requestID of user requests.
- In Getting Started Exercise 1, the example application uses the price session attribute to maintain the price of flowers ordered (for example, "price":25). The code hook (Lambda function) sets this attribute based on the type of flowers ordered. For more information, see [Review the Details of Information Flow](#).
- In the BookTrip bot exercise, the application uses the `currentReservation` session attribute to maintains the slot data during the in-progress conversation to book a hotel or book a car. For more information, see [Details of Information Flow](#).

Amazon Lex passes these session attributes to the Lambda functions configured for the intent In the your Lambda function, you can use the session attributes for initialization and customization (prompts). Some examples are:

- Initialization - In a pizza ordering application, if you pass user location (for example, "Location : 111 Maple Street"), then your Lambda function might use this information to determine the closest pizzeria to place the order (and perhaps set the `storeAddress` slot value as well).

Personalized prompts - For example, you can configure prompts to refer to the user by name (for example, "Hey [firstName], what toppings would you like?"). You can pass the user's name as a session attribute ("firstName": "Joe") so that Amazon Lex can substitute the placeholder to provide a personalized prompt to the user ("Hey Joe, what toppings would you like?").

#### **Note**

Amazon Lex does not persist session attributes.

If you configured a code hook for the intent, Amazon Lex passes the incoming session attributes to the Lambda function. The Lambda function must return these session attributes if you want Amazon Lex to return them to the client.



If there is no code hook configured for the intent Amazon Lex simply returns the session attributes to the client application.

#### **userId (p. 106)**

ID of the client application user. Typically, each of your application users should have a unique ID. The application developer decides the user IDs. At runtime, each request must include the user ID. Note the following considerations:

- If you want a user to start conversation on one device and continue the conversation on another device, you might choose a user-specific identifier, such as the user's login, or Amazon Cognito user ID (assuming your application is using Amazon Cognito).
- If you want the same user to be able to have two independent conversations on two different devices, you might choose device-specific identifier, such as device ID, or some globally unique identifier.

Length Constraints: Minimum length of 2. Maximum length of 50.

Pattern: [0-9a-zA-Z.\_:-]+

## Request Body

The request accepts the following data in JSON format.

#### **inputStream (p. 106)**

User input, in the format as described in the `Content-Type` HTTP header.

Type: Binary data object

Required: Yes

## Response Syntax

```
HTTP/1.1 200
Content-Type: contentType
x-amz-lex-intent-name: intentName
x-amz-lex-slots: slots
x-amz-lex-session-attributes: sessionAttributes
x-amz-lex-message: message
x-amz-lex-dialog-state: dialogState
x-amz-lex-slot-to-elicit: slotToElicit

audioStream
```

## Response Elements

If the action is successful, the service sends back an HTTP 200 response. The response returns the following HTTP headers.

#### **contentType (p. 108)**

Content type as specified in the `Accept` HTTP header in the request.

#### **dialogState (p. 108)**

Identifies the current state of the user interaction. Amazon Lex returns one of the following values as `dialogState`. The client can optionally use this information to customize the user interface.

- `ElicitIntent` – Amazon Lex wants to elicit the user's intent. Consider the following examples:  
For example, a user might utter an intent ("I want to order a pizza"). If Amazon Lex cannot infer the user intent from this utterance, it will return this dialog state.
- `ConfirmIntent` – Amazon Lex is expecting a "yes" or "no" response.  
For example, Amazon Lex wants user confirmation before fulfilling an intent. Instead of a simple "yes" or "no" response, a user might respond with additional information. For example, "yes,

but make it a thick crust pizza" or "no, I want to order a drink." Amazon Lex can process such additional information (in these examples, update the crust type slot or change the intent from OrderPizza to OrderDrink).

- `ElicitSlot` – Amazon Lex is expecting the value of a slot for the current intent. For example, suppose that in the response Amazon Lex sends this message: "What size pizza would you like?". A user might reply with the slot value (e.g., "medium"). The user might also provide additional information in the response (e.g., "medium thick crust pizza"). Amazon Lex can process such additional information appropriately.
- `Fulfilled` – Conveys that the Lambda function has successfully fulfilled the intent.
- `ReadyForFulfillment` – Conveys that the client has to fulfill the request.
- `Failed` – Conveys that the conversation with the user failed.

This can happen for various reasons, including that the user does not provide an appropriate response to prompts from the service (you can configure how many times Amazon Lex can prompt a user for specific information), or if the Lambda function fails to fulfill the intent.

Valid Values: `ElicitIntent` | `ConfirmIntent` | `ElicitSlot` | `Fulfilled` | `ReadyForFulfillment` | `Failed`

### **intentName** (p. 108)

Current user intent that Amazon Lex is aware of.

### **message** (p. 108)

Message to convey to the user. It can come from the bot's configuration or a code hook (Lambda function). If the current intent is not configured with a code hook or if the code hook returned `Delegate` as the `dialogAction.type` in its response, then Amazon Lex decides the next course of action and selects an appropriate message from the bot configuration based on the current user interaction context. For example, if Amazon Lex is not able to understand the user input, it uses a clarification prompt message (For more information, see the Error Handling section in the Amazon Lex console). Another example: if the intent requires confirmation before fulfillment, then Amazon Lex uses the confirmation prompt message in the intent configuration. If the code hook returns a message, Amazon Lex passes it as-is in its response to the client.

Length Constraints: Minimum length of 1. Maximum length of 1024.

### **sessionAttributes** (p. 108)

Map of key/value pairs representing the session-specific context information.

### **slots** (p. 108)

Map of zero or more intent slots (name/value pairs) Amazon Lex detected from the user input during the conversation.

### **slotToElicit** (p. 108)

If the `dialogState` value is `ElicitSlot`, returns the name of the slot for which Amazon Lex is eliciting a value.

The response returns the following as the HTTP body.

<varlistentry> **audioStream** (p. 108)

The prompt (or statement) to convey to the user. This is based on the application configuration and context. For example, if Amazon Lex did not understand the user intent, it sends the `clarificationPrompt` configured for the application. If the intent requires confirmation before taking the fulfillment action, it sends the `confirmationPrompt`. Another example: Suppose that the Lambda function successfully fulfilled the intent, and sent a message to convey to the user. Then Amazon Lex sends that message in the response.

</varlistentry>

## **Errors**

### **BadGatewayException**

Either the Amazon Lex bot is still building, or one of the dependent services (Amazon Polly, AWS Lambda) failed with an internal service error.

HTTP Status Code: 502

**BadRequestException**

Request validation failed, there is no usable message in the context, or the bot build failed.

HTTP Status Code: 400

**ConflictException**

Two clients are using the same AWS account, Amazon Lex bot, and user ID.

HTTP Status Code: 409

**DependencyFailedException**

One of the downstream dependencies, such as AWS Lambda or Amazon Polly, threw an exception. For example, if Amazon Lex does not have sufficient permissions to call a Lambda function, it results in Lambda throwing an exception.

HTTP Status Code: 424

**InternalFailureException**

Internal service error. Retry the call.

HTTP Status Code: 500

**LimitExceededException**

HTTP Status Code: 429

**LoopDetectedException**

Lambda fulfilment function returned `DelegateDialogAction` to Amazon Lex without changing any slot values.

HTTP Status Code: 508

**NotAcceptableException**

The accept header in the request does not have a valid value.

HTTP Status Code: 406

**NotFoundException**

The resource (such as the Amazon Lex bot or an alias) that is referred to is not found.

HTTP Status Code: 404

**RequestTimeoutException**

The input speech is too long.

HTTP Status Code: 408

**UnsupportedMediaTypeException**

The Content-Type header (`PostContent` API) has an invalid value.

HTTP Status Code: 415

## Example

### Example 1

In this request, the URI identifies a bot (`Traffic`), bot version (`$LATEST`), and end user name (`someuser`). The `Content-Type` header identifies the format of the audio in the body. Amazon Lex also supports other formats. To convert audio from one format to another, if necessary, you can use SoX open source software. You specify the format in which you want to get the response by adding the `Accept` HTTP header.

In the response, the `x-amz-lex-message` header shows the response that Amazon Lex returned. The client can then send this response to the user. The same message is sent in audio/MPEG format through chunked encoding (as requested).

### Sample Request

```
"POST /bot/Traffic/alias/$LATEST/user/someuser/content HTTP/1.1[\r][\n]"
"x-amz-lex-session-attributes: eyJlc2VyTmFtZSI6IkpvYiJ9[\r][\n]"
"Content-Type: audio/x-l16; channel-count=1; sample-rate=16000f[\r][\n]"
```



- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

## PostText

Sends user input (text-only) to Amazon Lex. Clients can use this API to send requests to Amazon Lex at runtime. Amazon Lex then interprets the user input using the machine learning model it built for the bot.

In response, Amazon Lex returns the next `message` to convey to the user an optional `responseCard` to display. Consider the following example messages:

- For a user input "I would like a pizza", Amazon Lex might return a response with a message eliciting slot data (for example, `PizzaSize`): "What size pizza would you like?"
- After the user provides all of the pizza order information, Amazon Lex might return a response with a message to obtain user confirmation "Proceed with the pizza order?".
- After the user replies to a confirmation prompt with a "yes", Amazon Lex might return a conclusion statement: "Thank you, your cheese pizza has been ordered."

Not all Amazon Lex messages require a user response. For example, a conclusion statement does not require a response. Some messages require only a "yes" or "no" user response. In addition to the `message`, Amazon Lex provides additional context about the message in the response that you might use to enhance client behavior, for example, to display the appropriate client user interface. These are the `slotToElicit`, `dialogState`, `intentName`, and `slots` fields in the response. Consider the following examples:

- If the message is to elicit slot data, Amazon Lex returns the following context information:
  - `dialogState` set to `ElicitSlot`
  - `intentName` set to the intent name in the current context
  - `slotToElicit` set to the slot name for which the `message` is eliciting information
  - `slots` set to a map of slots, configured for the intent, with currently known values
- If the message is a confirmation prompt, the `dialogState` is set to `ConfirmIntent` and `SlotToElicit` is set to null.
- If the message is a clarification prompt (configured for the intent) that indicates that user intent is not understood, the `dialogState` is set to `ElicitIntent` and `slotToElicit` is set to null.

In addition, Amazon Lex also returns your application-specific `sessionAttributes`. For more information, see [Managing Conversation Context](#).

## Request Syntax

```
POST /bot/botName/alias/botAlias/user/userId/text HTTP/1.1
Content-type: application/json

{
  "inputText": "string",
  "sessionAttributes": {
    "string" : "string"
  }
}
```

## URI Request Parameters

The request requires the following URI parameters.

### **botAlias** (p. 113)

The alias of the Amazon Lex bot.

### **botName (p. 113)**

The name of the Amazon Lex bot.

### **userId (p. 113)**

The ID of the client application user. The application developer decides the user IDs. At runtime, each request must include the user ID. Typically, each of your application users should have a unique ID. Note the following considerations:

- If you want a user to start a conversation on one device and continue the conversation on another device, you might choose a user-specific identifier, such as a login or Amazon Cognito user ID (assuming your application is using Amazon Cognito).
- If you want the same user to be able to have two independent conversations on two different devices, you might choose a device-specific identifier, such as device ID, or some globally unique identifier.

Length Constraints: Minimum length of 2. Maximum length of 50.

Pattern: [0-9a-zA-Z.\_:-]+

## Request Body

The request accepts the following data in JSON format.

### **inputText (p. 113)**

The text that the user entered (Amazon Lex interprets this text).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: Yes

### **sessionAttributes (p. 113)**

By using session attributes, a client application can pass contextual information in the request to Amazon Lex. For example,

- In *Getting Started Exercise 1*, the example application uses the `price` session attribute to maintain the price of the flowers ordered (for example, "Price":25). The code hook (the Lambda function) sets this attribute based on the type of flowers ordered. For more information, see [Review the Details of Information Flow](#).
- In the *BookTrip bot exercise*, the application uses the `currentReservation` session attribute to maintain slot data during the in-progress conversation to book a hotel or book a car. For more information, see [Details of Information Flow](#).
- You might use the session attributes (key, value pairs) to track the requestID of user requests. Amazon Lex simply passes these session attributes to the Lambda functions configured for the intent.

In your Lambda function, you can also use the session attributes for initialization and customization (prompts and response cards). Some examples are:

- **Initialization** - In a pizza ordering application, if you can pass the user location as a session attribute (for example, "Location" : "111 Maple street"), then your Lambda function might use this information to determine the closest pizzeria to place the order (perhaps to set the `storeAddress` slot value).
- **Personalize prompts** - For example, you can configure prompts to refer to the user name. (For example, "Hey [FirstName], what toppings would you like?"). You can pass the user name as a session attribute ("FirstName" : "Joe") so that Amazon Lex can substitute the placeholder to provide a personalized prompt to the user ("Hey Joe, what toppings would you like?").

#### **Note**

Amazon Lex does not persist session attributes.

If you configure a code hook for the intent, Amazon Lex passes the incoming session attributes to the Lambda function. If you want Amazon Lex to return these session attributes back to the client, the Lambda function must return them.

If there is no code hook configured for the intent, Amazon Lex simply returns the session attributes back to the client application.

Type: String to String map  
Required: No

## Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "dialogState": "string",
  "intentName": "string",
  "message": "string",
  "responseCard": {
    "contentType": "string",
    "genericAttachments": [
      {
        "attachmentLinkUrl": "string",
        "buttons": [
          {
            "text": "string",
            "value": "string"
          }
        ],
        "imageUrl": "string",
        "subTitle": "string",
        "title": "string"
      }
    ],
    "version": "string"
  },
  "sessionAttributes": {
    "string": "string"
  },
  "slots": {
    "string": "string"
  },
  "slotToElicit": "string"
}
```

## Response Elements

If the action is successful, the service sends back an HTTP 200 response. The following data is returned in JSON format by the service.

### **dialogState** (p. 115)

Identifies the current state of the user interaction. Amazon Lex returns one of the following values as `dialogState`. The client can optionally use this information to customize the user interface.

- `ElicitIntent` – Amazon Lex wants to elicit user intent.  
For example, a user might utter an intent ("I want to order a pizza"). If Amazon Lex cannot infer the user intent from this utterance, it will return this `dialogState`.
- `ConfirmIntent` – Amazon Lex is expecting a "yes" or "no" response.  
For example, Amazon Lex wants user confirmation before fulfilling an intent. Instead of a simple "yes" or "no," a user might respond with additional information. For example, "yes, but make it thick crust pizza" or "no, I want to order a drink". Amazon Lex can process



such additional information (in these examples, update the crust type slot value, or change intent from OrderPizza to OrderDrink).

- **ElicitSlot** – Amazon Lex is expecting a slot value for the current intent.  
For example, suppose that in the response Amazon Lex sends this message: "What size pizza would you like?". A user might reply with the slot value (e.g., "medium"). The user might also provide additional information in the response (e.g., "medium thick crust pizza"). Amazon Lex can process such additional information appropriately.
- **Fulfilled** – Conveys that the Lambda function configured for the intent has successfully fulfilled the intent.
- **ReadyForFulfillment** – Conveys that the client has to fulfill the intent.
- **Failed** – Conveys that the conversation with the user failed.  
This can happen for various reasons including that the user did not provide an appropriate response to prompts from the service (you can configure how many times Amazon Lex can prompt a user for specific information), or the Lambda function failed to fulfill the intent.

Type: String

Valid Values: `ElicitIntent` | `ConfirmIntent` | `ElicitSlot` | `Fulfilled` | `ReadyForFulfillment` | `Failed`

#### **intentName (p. 115)**

The current user intent that Amazon Lex is aware of.

Type: String

#### **message (p. 115)**

A message to convey to the user. It can come from the bot's configuration or a code hook (Lambda function). If the current intent is not configured with a code hook or the code hook returned `Delegate` as the `dialogAction.type` in its response, then Amazon Lex decides the next course of action and selects an appropriate message from the bot configuration based on the current user interaction context. For example, if Amazon Lex is not able to understand the user input, it uses a clarification prompt message (for more information, see the Error Handling section in the Amazon Lex console). Another example: if the intent requires confirmation before fulfillment, then Amazon Lex uses the confirmation prompt message in the intent configuration. If the code hook returns a message, Amazon Lex passes it as-is in its response to the client.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

#### **responseCard (p. 115)**

Represents the options that the user has to respond to the current prompt. Response Card can come from the bot configuration (in the Amazon Lex console, choose the settings button next to a slot) or from a code hook (Lambda function).

Type: [ResponseCard \(p. 121\)](#) object

#### **sessionAttributes (p. 115)**

A map of key-value pairs representing the session-specific context information.

Type: String to String map

#### **slots (p. 115)**

The intent slots (name/value pairs) that Amazon Lex detected so far from the user input in the conversation.

Type: String to String map

#### **slotToElicit (p. 115)**

If the `dialogState` value is `ElicitSlot`, returns the name of the slot for which Amazon Lex is eliciting a value.

Type: String

## Errors

**BadGatewayException**

Either the Amazon Lex bot is still building, or one of the dependent services (Amazon Polly, AWS Lambda) failed with an internal service error.

HTTP Status Code: 502

**BadRequestException**

Request validation failed, there is no usable message in the context, or the bot build failed.

HTTP Status Code: 400

**ConflictException**

Two clients are using the same AWS account, Amazon Lex bot, and user ID.

HTTP Status Code: 409

**DependencyFailedException**

One of the downstream dependencies, such as AWS Lambda or Amazon Polly, threw an exception. For example, if Amazon Lex does not have sufficient permissions to call a Lambda function, it results in Lambda throwing an exception.

HTTP Status Code: 424

**InternalFailureException**

Internal service error. Retry the call.

HTTP Status Code: 500

**LimitExceededException**

HTTP Status Code: 429

**LoopDetectedException**

Lambda fulfillment function returned `DelegateDialogAction` to Amazon Lex without changing any slot values.

HTTP Status Code: 508

**NotFoundException**

The resource (such as the Amazon Lex bot or an alias) that is referred to is not found.

HTTP Status Code: 404

## See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V2](#)

## Data Types

The following data types are supported:

- [Button](#) (p. 119)
- [GenericAttachment](#) (p. 120)
- [ResponseCard](#) (p. 121)



## Button

Represents an option to be shown on the client platform (Facebook, Slack, etc.)

### Contents

#### **text**

Text that is visible to the user on the button.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 15.

Required: Yes

#### **value**

The value sent to Amazon Lex when a user chooses the button. For example, consider button text "NYC." When the user chooses the button, the value sent can be "New York City."

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1000.

Required: Yes

### See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

# GenericAttachment

Represents an option rendered to the user when a prompt is shown. It could be an image, a button, a link, or text.

## Contents

### **attachmentLinkUrl**

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

### **buttons**

The list of options to show to the user.

Type: array of [Button \(p. 119\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 5 items.

Required: No

### **imageUrl**

The URL of an image that is displayed to the user.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 2048.

Required: No

### **subTitle**

The subtitle shown below the title.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 80.

Required: No

### **title**

The title of the option.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 80.

Required: No

## See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

## ResponseCard

If you configure a response card when creating your bots, Amazon Lex substitutes the session attributes and slot values that are available, and then returns it. The response card can also come from a Lambda function ( `dialogCodeHook` and `fulfillmentActivity` on an intent).

### Contents

#### **contentType**

The content type of the response.

Type: String

Valid Values: `application/vnd.amazonaws.card.generic`

Required: No

#### **genericAttachments**

An array of attachment objects representing options.

Type: array of [GenericAttachment](#) (p. 120) objects

Array Members: Minimum number of 0 items. Maximum number of 10 items.

Required: No

#### **version**

The version of the response card format.

Type: String

Required: No

### See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java](#)
- [AWS SDK for Ruby V2](#)

# Document History for Amazon Lex

---

*This is prerelease documentation for a service in preview release. It is subject to change.*

The following table describes the documentation for this release of Amazon Lex.

- **Latest documentation update:** November 30, 2016

Change	Description	Date
Preview release	Preview release of the Amazon Lex Developer Guide.	November 30, 2016

# AWS Glossary

---

For the latest AWS terminology, see the [AWS Glossary](#) in the *AWS General Reference*.