

Alexa Skill Builder's Guide

# Advanced Skill Building with Dialog Management



Justin Jeffress

Senior Solutions Architect

Amazon Alexa

# Table of Contents

You can navigate to each section of this guide by **clicking the page titles** listed below.

<b>Introduction</b>	<b>1</b>
<b>What Is Conversational User Interface Design?</b>	<b>2</b>
<b>Simplifying Slot Collection with Dialog Management</b>	<b>5</b>
<b>Implementing Dialog Management in 10 Steps</b>	<b>6</b>
Step 1: Defining Intent-Level Utterances	7
Step 2: Defining Custom Slot Types	9
Step 3: Marking Slots Required	11
Step 4: Providing Prompts for Each Slot	12
Step 5: Providing Utterances for Each Slot	13
Step 6: Delegating Slot Collection from Your Backend	15
Step 7: Understanding the Dialog State	16
Step 8: Setting Default Values	17
Step 9: Eliciting Slots	17
Step 10: Providing the Match Upon Completion	18
<b>Advanced Dialog Management Techniques</b>	<b>19</b>
Conditional Slot Collection	19
Context Switching	23

# Introduction

Conversations are dynamic, moving between topics and ideas fluidly. Through voice, we are able to converse with technology, which is far more engaging than clicking buttons on a screen. To create truly engaging conversational voice experiences, you'll need to incorporate flexibility and responsiveness in your Alexa skills. Your skills should be able to handle variations of conversation, conditional collection of data, and switching context mid-conversation without losing track of what's being said.

While it is entirely possible to add your own code to create and manage these conversational elements, you can dramatically reduce the amount of effort required to do so and improve automatic speech recognition with [dialog management](#), which is built into the [Alexa Skills Kit \(ASK\)](#).

Dialog management provides a set of directives for managing a conversation between your Alexa skill and the customer. Your skill can return the directives to have Alexa ask the customer for the information you need in order to perform a task. It keeps track of the state of the conversation, including what information has and hasn't been collected and will report the state back to you. From your skill's backend, you can decide what to do. You can delegate prompting the next slot, confirm a slot, confirm an intent, or elicit a different slot.

Dialog management also enables Alexa to narrow in on what slot values to expect next, which improves the accuracy of your skill because the Alexa service has the context of what intent is presently being resolved.

In this guide, we will show you how to leverage dialog management within your skills to enhance interactions with your customers. The code samples in this guide are all referenced from a sample skill called [Pet Match](#). Pet Match recommends a dog to the customer based on a conversation that uses dialog management. This document provides code samples for the [ASK Software Development Kit for Node.js](#).

# What Is Conversational User Interface Design?

Conversations aren't scripted. They are dynamic. They will move from topic to topic while the underlying context is tracked and remembered. Upon returning to a previously discussed topic, the participants will remember what has been discussed. For example, if Paul asks Jane a question and Jane only provides half an answer, Paul will follow up with questions until Jane has provided all the answers that Paul was looking for. On the other hand, when Paul asks Jane for answers and Jane provides extra relevant information that answers an unasked question, there is no need for a follow-up question. When designing your voice interactions you should be mindful of this experience and refrain from forcing the customer to follow your script exactly as you wrote it. It should be malleable in order to handle the variations of conversation.

Let's use the following conversation as an example. You want a dog, but don't really know much about them. There are many types of dogs: large, tiny, rowdy, docile, short-haired, hypoallergenic, and the list goes on and on. It can be very overwhelming to figure out what kind of dog is perfect for you. For example, if you live in an apartment, you probably don't want a high-energy dog that needs to roam around in a field for large periods of the day to stay healthy and happy. Lucky for you, your best friend, Mia, is an expert. To narrow down what kind of dog would match your situation, she asks you a series of questions:

**Mia:** What size dog are you looking for?

**You:** I want a large dog.

**Mia:** Are you looking for a dog to play with or relax with?

**You:** I want a dog to relax with.

**Mia:** Would you prefer a dog to hang out with kids or to protect you?

**You:** I want a dog to protect me.

**Mia:** So you want a large, low-energy guard dog. I think a Doberman Pinscher would be perfect for you!

Isn't Mia the best! She was able to find the perfect dog for you by asking three simple questions. Once she knows the **size**, **energy**, and **temperament** (large, low, and guard dog) that you want, she can provide a recommendation. Throughout the course of the conversation, she keeps track of the answers provided and asks questions until she has everything she needs. You can over-answer and give her much more information than she needs, but she really only needs to know three things. If you provide a different answer to even one of her questions, her recommendation will change accordingly. For example, if you said you want a **tiny** dog, she might recommend a "chihuahua" instead. (Do chihuahuas really make good guard dogs or is Mia joking?)

Let's say you want to make Mia's dog-matching abilities into an Alexa skill so anyone can find the perfect dog based on Mia's method. Just like Mia, your skill will need to be able to track the conversation and follow up with questions until it knows the desired **size**, **energy**, and **temperament**. Your skill will need to be able to handle a variety of interactions inherent to conversation, such as one-shot, underfilled, and overfilled utterances, described below.

### **One-Shot Utterances**

The first time a customer interacts with the skill, they most likely aren't familiar with Mia's method. As they interact with the skill and understand the method, they may provide all three pieces of information necessary for the match all in one sentence. For example, "I want a large, guard dog to relax with at my apartment." This type of utterance is called a "**one-shot utterance**" because it includes all of the slots necessary that an intent needs to perform its task.

### **Underfilled Utterance**

Conversations are fluid and it's perfectly normal to skip over a detail or leave some vital information out of an answer, especially if it's answer to a two-part question. For example, when asked, "What **size** and **temperament** are you looking for?" The user may reply, "I want a **large** dog." In this case, they've only provided the size slot. This is referred to as underfilling. It would totally disorient the user to follow up with the exact same question. It requires a follow-up question to fill just the **temperament** slot.

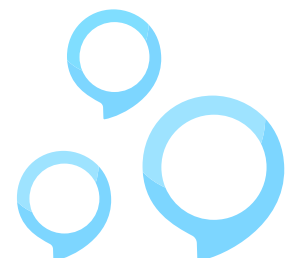
### **Overfilled Utterance**

Likewise, the customer could provide more information than what was asked. For example, to "What **size** of a dog would you like?" the customer replies, "I want a **large guard dog**." Here they've actually given more information than what was asked. The customer provided **size** and **temperament**. This is known as **overfilling** and it wouldn't make sense to follow up with a

question about the **temperament** since your skill should already know the answer. Your skill should be able to handle overfilled slots.

Tracking the information your skill needs to perform the match for the customer poses quite a challenge. Throughout every interaction with your skill, you would need to keep a list of required slots, collected slots, questions to ask in order to fill the slots, and the state of the conversation such as started, in progress, and completed and store them somewhere such as the session or a database. As each slot(s) is provided, you need to check your list of required slots with the collected ones and if there are any missing you need to prompt the customer to fill them by using your list of questions, repeating the process until all required slots have been collected.

While it is entirely possible to manage slot collection yourself, you'd be reinventing the wheel since the Alexa Skills Kit includes dialog management. It will do the heavy lifting for you, which frees you up to focus on consuming the slots, performing the task, and providing value to your customers. It will also improve the accuracy of your skill due to the increased focus on the slot that's being filled. In the next section, we will walk through the Pet Match sample skill, which uses dialog management to facilitate slot collection.

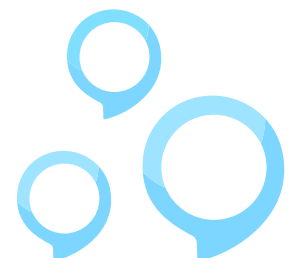


# Simplifying Slot Collection with Dialog Management

Dialog management is a powerful tool that reduces the necessary work to track the state of a conversation and the slots collected from the customer. It keeps track of the slots that you tell it to and it uses a state machine to manage the state (**dialogState**) of the conversation which includes three states: **STARTED**, **IN\_PROGRESS**, and **COMPLETED**. When the dialog starts between the customer and your skill, at each turn of the interaction the JSON request that is sent to your skill will include all the slots associated with the intent, and as well as the **dialogState**, which you can inspect and decide to:

- Set default values
- Delegate slot collection back to Alexa
- Elicit slots
- Elicit slot confirmations
- Elicit intent confirmations
- Manually override dialog management

You can think of the interaction as if you were playing a game of catch. Alexa sends JSON to your skill, and you send JSON back, but the JSON you provide will determine what type of pitch Alexa will throw back. Simply delegating back to Alexa is equivalent to asking for a fastball straight down the middle. Alexa will prompt the customer for the next missing slot and the JSON will include the answer. You can have Alexa throw secondary pitches through the **elicitSlot**, **confirmSlot**, and **confirmIntent** directives. You can even manually override dialog management. In the section titled [Conditional Slot Collection](#), you'll learn how to collect slots based on a condition.



# Implementing Dialog Management in 10 Steps

There are only a few steps that you need to take in order take advantage of dialog management. First you will need to activate dialog management in your voice user interface from the [Alexa Developer Portal](#) and add some code to your backend to hook into and send **dialog directives** to delegate slot collection to Alexa.

The process involves:

1. Defining Intent-Level Utterances
2. Defining Custom Slot Types
3. Marking Slots Required
4. Providing Prompts for Each Slot
5. Providing Utterances for Each Slot
6. Delegating Slot Collection from Your Backend
7. Understanding the Dialog State
8. Setting Default Values
9. Eliciting Slots
10. Providing the Match Upon Completion

Steps 1 through 5 are carried out from your skill's user interaction model and involves no coding at all. Steps 6 through 10 involve programming. You can simply delegate collecting the remaining slots to Alexa, or you can tap into the process in order to set default values, override prompts, elicit slots, confirm slots, and check a database or web service.



## Step 1: Defining Intent-Level Utterances

Let's start by defining our utterances based on our original conversation with Mia.

### Recommend a dog

I want a dog.

I want a large dog.

I want a dog to relax with.

I want a dog to protect me.

I want a guard dog.

I want a large family dog.

I want a large, guard dog to relax with at my apartment.

Next, we will identify our slots.

### Recommend a {pet}

I want a {pet}.

I want a {size} {pet}.

I want a {pet} {energy}.

I want a {pet} {temperament}.

I want a {temperament} {pet}.

I want a {size} {temperament} {pet}.

I want a {size} {temperament} {pet} {energy} at my apartment.

Once you've added your utterances, they should appear like the image on the following page:

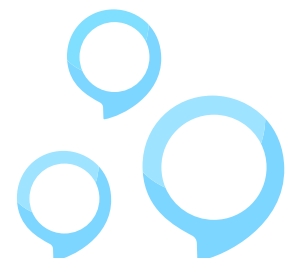
## Intents / PetMatchIntent

### Sample Utterances (11) <sup>?</sup>

What might a user say to invoke this intent?	+
recommend a dog	
I want a {pet}	
I want a {size} {pet}	
I want a {pet} {energy}	
I want a {pet} {temperament}	
I want a {temperament} {pet}	
I want {size} {temperament} {energy} {pet}	
I want a {size} {temperament} {energy} at my apartment	
I want a {size} {temperament} {pet} that's {energy}	
I want {energy} {pet}	
I want a {size} {temperament} {pet} to {energy}	

#### Developer Tip

Dialog management allows you to choose which slots should be required. Although we didn't do it in the example above, you can mix and match utterances with required and unrequired slots. You can even make a slot required that doesn't appear in your utterances. Since it's required, Alexa will prompt for it.



## Step 2: Defining Custom Slot Types

As you can see, we have four slots: **pet**, **size**, **energy**, and **temperament**. The pet slot is using the built-in **AMAZON.ANIMAL** slot type while the **size**, **energy**, and **temperament** slots are custom and their values are defined below.

size	energy	temperament
tiny	low	family
small	medium	guard
medium	high	
large		

One thing you may have noticed is that the **energy** slot values are “low,” “medium,” and “high.” However, when talking to Mia we were able to say things like “to relax with” and she was able to understand what it means and perform the match. In every day conversation, there are many ways to express the same idea. In our skill’s interaction model we can map synonyms to our values using **entity resolution**. Entity resolution allows you to set your slot values as entities and map a set of synonyms to them. Alexa will then try to match what the customer says to either an entity or one of your synonyms values.

Each of our custom slot types will include the following synonyms:

### sizeType

value	id	synonym
tiny	xs	little, petite, itty bitty
small	s	little, take on an airplane
medium	m	average, up to my knees
large	l	huge, waist height

Let’s take a look on the following page to see what that will look like in the developer portal:

## Slot Types / sizeType

Slot Values (4) ?

Search

Enter a new value for this slot type



VALUE ?	ID (OPTIONAL) ?	SYNONYMS (OPTIONAL) ?	
tiny	xs	Add synonym +	itty bitty x petite x little x
small	s	Add synonym +	take on an airplane x little x
medium	m	Add synonym +	up to my knees x average x
large	l	Add synonym +	waist height x huge x

### energyType

value	id	synonym
low	low	to relax with, to cuddle with
medium	med	fun to play with, plays tug of war
high	high	energetic, play frisbee, that I can run with

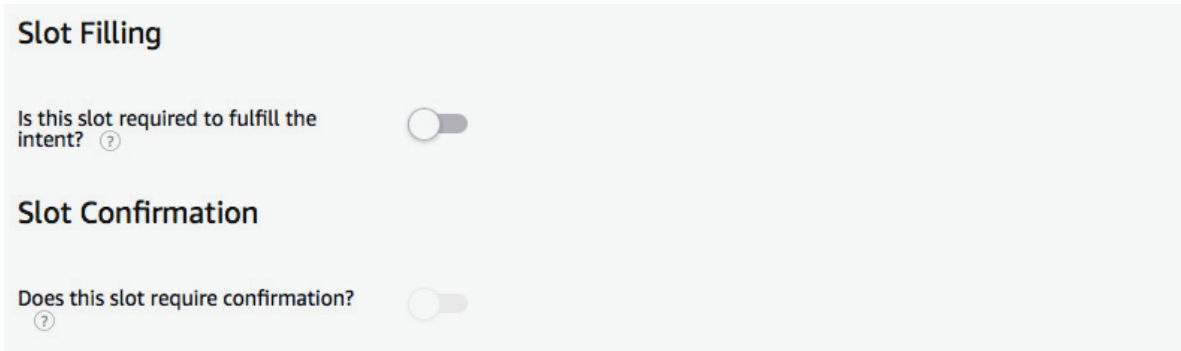
### temperamentType

value	id	synonym
family		to relax with, to cuddle with
guard		fun to play with, plays tug of war

While the “id” field is optional, it can come in handy when you have to make an API call or database lookup based on an id. Instead of writing code to map the synonyms to our id, we simply define the id with entity resolution and when the customer provides a slot value, we get the id for free.

### Step 3: Marking Slots Required

Now that we've defined our slots, it's now time to make them required. From the [skill builder](#) within the developer portal, click on one of the slots **size**, **energy**, or **temperament** and toggle the "Is this slot required to fulfill the intent?" switch to "on."



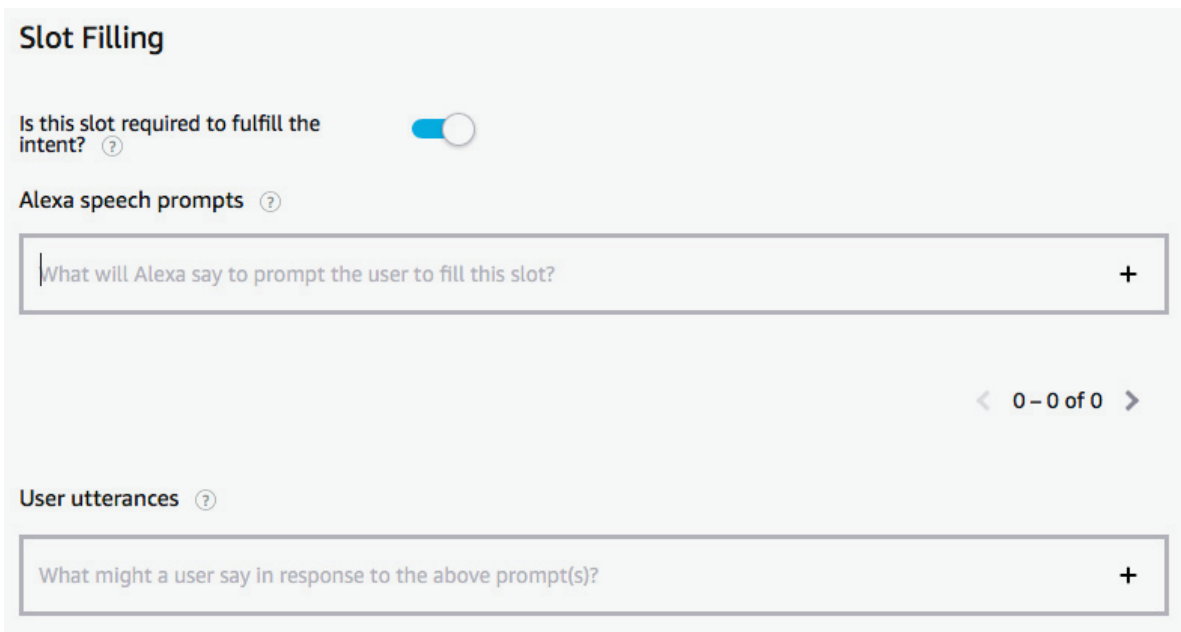
**Slot Filling**

Is this slot required to fulfill the intent? ☐

**Slot Confirmation**

Does this slot require confirmation? ☐

Upon toggling the switch, "elicitationRequired" is set to true, which will make Alexa track this slot and ensure that value has been provided and the **Alexa speech prompts** and **User utterances** boxes will appear.



**Slot Filling**

Is this slot required to fulfill the intent? ☒

**Alexa speech prompts**

What will Alexa say to prompt the user to fill this slot? +

< 0 - 0 of 0 >

**User utterances**

What might a user say in response to the above prompt(s)? +

Take a look at your Interaction Mode's JSON using the JSON Editor. It will now include the following JSON snippet.

```
...
"slots": [
{
  ...
  {
    "name": "size",
    "type": "sizeType",
    "confirmationRequired": false,
    "elicitationRequired": true,
    "prompts": {
      "elicitation": "Elicit.Intent-PetMatchIntent.IntentSlot-size"
    }
  }
  ...
}
...
```


The value **"Elicit.Intent-PetMatchIntent.IntentSlot-size"** is the prompt ID, which links the prompts to the slot.

Now we need to provide prompts.

## Step 4: Providing Prompts for Each Slot


For each one of our slots, **size**, **energy**, and **temperament**, we will need to provide a prompt so Alexa knows what question to ask the user to fill it. We can use the exact same questions that Mia asked for prompts. For example, "What size of a dog are you looking for?" or "There are dogs that are tiny, small, medium, and large. Which would you like?"

### Slot Filling

Is this slot required to fulfill the intent? 

Alexa speech prompts 

What will Alexa say to prompt the user to fill this slot? 

There are dogs that are tiny, small, medium, and large, which would you like? 

Upon providing prompts, your interaction model will be updated with the following JSON snippet:

```
...
"prompts": [
  ...
  {
    "id": "Elicit.Intent-PetMatchIntent.IntentSlot-size",
    "variations": [
      {
        "type": "PlainText",
        "value": "There are dogs that are tiny, small, medium, and large. Which would you like?"
      },
      {
        "type": "PlainText",
        "value": "What size of a dog would you like?"
      }
    ]
  },
  ...
]
...
```

As you can see our "Elicit.Intent-PetMatchIntent.IntentSlot-size" id that appeared in the slot definition is present in the prompt object. All of our prompt variations are contained in an array and Alexa will randomly choose one. Having several prompts is a way to add variation into your interactions.

#### Developer Tip

You can include slots in your prompts such as, "So you want a {size} dog. Do you want a dog that's good with kids or to protect you?" If the **size** was previously filled when prompting for the **temperament** slot, Alexa will fill the slot with the size value. You should at least provide one prompt without any slots; otherwise, if **size** hasn't been provided yet, Alexa will say, "So you want an undefined dog. Do you want a dog that's good with kids or to protect you?" Alexa will choose a prompt without a slot if it's undefined as long as you have one without prompts in it.

## Step 5: Providing Utterances for Each Slot

For each slot, you will need to provide utterances so the customer can fill the slots. Alexa will use these utterances when resolving what the customer says, but your other utterances will be active. This will allow the customer to switch context mid-conversation and trigger another intent, which is covered in the section [Context Switching](#). For example, to fill the size slot, the customer could say:

I want a small dog.  
Large.

Next, we will identify our slots and replace the values with them.

```
I want a {size} {pet}
{size}
```

It's quite possible the customer might say, "I want a large guard dog". To cater to this, we can also provide the overfilled utterances as well. For example:

```
I want a {size} {temperament} {pet}
I want a {size {energy} {temperament} {pet}
```

#### User utterances ?

What might a user say in response to the above prompt(s)?		+
I want a {size} {temperament} {pet}		🗑️
I want a {size} {pet}		🗑️
{size}	<b>Optional Utterance</b>	🗑️

#### Developer Tip

When you mark a slot required, Alexa will automatically add the slot-only utterance for your skill. Above we added {size} to the customer utterances, but we could have left it blank. If the customer said, "large" the size slot will be filled. Below we left out the slot-only utterance for {size} but our skill will still be able to understand "large" and resolve it to our size slot.

#### User utterances ?

What might a user say in response to the above prompt(s)?		+
I want a {size} {temperament} {pet}		🗑️
I want a {size} {pet}		🗑️



These slot utterances will appear in our interaction model as samples associated with the slot.

```
...
"intents": [
  {
    "name": "PetMatchIntent",
    "slots": [
      ...
      {
        "name": "size",
        "type": "sizeType",
        "samples": [
          "I want a {size} {pet}",
          "I want a {size} {temperament} {pet}",
          "I want a {size} {energy} {temperament} {pet}",
        ]
      },
      ...
    ]
  },
  ...
]
```

Once you've activated dialog management from your skill's front end, you will need to create and add support to your backend so it can delegate the slot collection to Alexa when needed. It's important to note that your skill will receive a request during each turn of the interaction, so you can still control the dialog through your code.

## Step 6: Delegating Slot Collection from Your Backend

Up until this point, we've spent a lot of time setting up our front end. Steps 6-10 focus on your backend. You'll learn how tap into the dialog management state machine and delegate slot collection back to Alexa. The knowledge gained here is essential to understand the fundamentals and leverage its full power.

After each utterance spoken to your skill, Alexa sends a JSON request to your backend (typically an AWS Lambda function). Among other important information, this request contains the type of request and any slots that the user has provided. If the intent supports dialog management, the request will also include a key/value pair indicating the state of the dialog. You can check this state indicator and decide the next response. For example, you can:

- Set default values
- Delegate collecting the next slot back to Alexa (`addDelegateDirective`)
- Elicit slots (`addElicitSlotDirective`)
- Confirm slots (`addConfirmSlotDirective`)
- Confirm intents (`addConfirmIntentDirective`)

In addition to the above, you can choose not to return a **dialog directive**, which will end dialog management.

## Step 7: Understanding the Dialog State

The **dialogState** determines what state of the interaction that dialog management is in. There are three states, **STARTED**, **IN\_PROGRESS** and **COMPLETED**.

From our **PetMatchIntent** we can inspect the **dialogState** and determine what we want to do. To make Alexa handle prompting for the next slot until the multi-turn dialog has been completed, we will delegate to Alexa. To do that we will return the **Dialog.Delegate** directive.

```
return inputHandler.responseBuilder
    .addDelegateDirective()
    .getResponse();
```

Upon doing so, our skill will output JSON that includes the **delegate** directive.

```
{
  "body": {
    "version": "1.0",
    "response": {
      "directives": [
        {
          "type": "Dialog.Delegate",
          "updatedIntent": {
            "name": "PetMatchIntent",
            "confirmationStatus": "NONE",
            "slots": {
              "size": {
                "name": "size",
                "confirmationStatus": "NONE"
              },
              "temperament": {
                "name": "temperament",
                "confirmationStatus": "NONE"
              },
              "pet": {
                "name": "pet",
                "value": "dog",
                "confirmationStatus": "NONE"
              },
              "energy": {
                "name": "energy",
                "confirmationStatus": "NONE"
              }
            }
          }
        }
      ]
    },
    "sessionAttributes": {},
    "userAgent": "ask-node/2.0.0 Node/v8.10.0"
  }
}
```

## Step 8: Setting Default Values

Oftentimes, you may want to set a default value for your slot in case the customer does not provide it. In our Pet Match example, we can set the default size to **medium** then pass the **updatedIntent** to **addDelegateDirective()**, which will set the value.

```
let updatedIntent = handlerInput.requestEnvelope.intent;
let updatedSlots = updatedIntent.slots;

if ( !updatedSlots.size.value ) {
    updatedSlots.size.value = "medium";
}

return handlerInput.responseBuilder
    .addDelegateDirective(updatedIntent);
.getResponse();
```

### Developer Tip

While **updatedIntent** is an optional parameter, you **must** provide it if you're making any modifications to the slots. Otherwise, the modifications will not take effect.

## Step 9: Eliciting Slots

If at any point you want to override one of your predefined slots or you want to change what slot is elicited next, you can return the **ElicitSlot** directive. It will contain the slot to elicit as well as the **outputSpeech** and **reprompt**.

To return the **ElicitSlot** directive, you can simply call **addElicitSlotDirective** and provide the **slotName**.

```
let speechOutput = 'I\'m sorry but banana is a not a size. ';
speechOutput += 'Please choose either tiny or small?';
const reprompt = 'Which would you like tiny or small?';
const slotName = 'size';

return handlerInput.responseBuilder
    .speak(speechOutput)
    .reprompt(reprompt)
    .addElicitSlotDirective(slotName)
    .getResponse();
```

The output from your skill will include the **outputSpeech** as well as the **elicitSlot** directive.

```

{
  "body": {
    "version": "1.0",
    "response": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I'm sorry but banana is a not a size. Please ch
      },
      "directives": [
        {
          "type": "Dialog.ElicitSlot",
          "slotToElicit": "size"
        }
      ],
      "reprompt": {
        "outputSpeech": {
          "type": "SSML",
          "ssml": "<speak>Which would you like tiny or small?</speak>
        }
      },
      "shouldEndSession": false
    },
    "sessionAttributes": {},
    "userAgent": "ask-node/2.0.0 Node/v8.10.0"
  }
}

```

## Step 10: Providing the Match Upon Completion

Once your skill receives a request where **dialogState** is **COMPLETED**, all the required slots have been collected, so we are ready to fulfill the request and make the recommendation to the customer. Here we simply return a speech output.

```

let speechOutput = "So you want a large high energy guard dog.";
speechOutput += " You should consider a Doberman Pincher"
return inputHandler.responseBuilder
  .speak(speechOutput);
  .getResponse();

```

### Developer Tip

You're not required to return a **dialog directive** so you can stop collecting the rest of the required slots if you determine that you don't need them. However, the Alexa service will throw an error if your skill returns a **dialog directive** outside the context of dialog management.

# Advanced Dialog Management Techniques

## Conditional Slot Collection

There are times when you might not need to capture all the required slots. For example, the required slots may differ based on a selection.

In our Pet Match example, let's say in addition to recommending a pet to the customer, you want to help them find an animal shelter that has the dog you've recommended. You can add a **FindAnimalShelterIntent** and prompt the customer for location information including the **zip**, **city**, and **state** to narrow the search. Let's say the API you're using supports a lookup by either the **zip** or **city** and **state**. If you have the **zip**, then you don't need to wait until the **dialogState** is **COMPLETED**. If the customer provides the **city**, you'll need to prompt for the **state** but you don't need the **zip**. Since dialog management sets up a state machine and we can inspect the slots that have been provided at each turn in the conversation, we can programmatically interrupt the slot collection lookup and provide the results. Why ask the customer for information that we don't technically need?

In this case, we are looking to satisfy, **zip** or **city** and **state**. We can translate that into a condition **zip || city && state**. Further simplifying the condition, we get **A || B && C**. Follow these steps to enable conditional slot selection.

### 1. Setting Up Your Interaction Model

First, you need to set up an **intent** and some **utterances** and **slots**. Our intent will be called **FindAnimalShelterIntent**. The utterances are:

```
find a shelter
find a shelter in {city}
find a shelter in {state}
find a shelter in {zip}
find a shelter close to {city} {state}
find a shelter close to {city} {state} {zip}
```

As you can see, the utterances for the intent have been set up to capture the slots in several ways:

### No Slots

The “find a shelter” utterance will invoke the **FindAnimalShelterIntent**. Since it provides no slots, dialog management will kick in and ask the customer to fill all the required slots until all have been filled or that our backend has identified that **A || B && C** has been met so we can look up a shelter.

### Single Slots

“find a shelter in {city}”, “find a shelter in {state}” and “find a shelter in {zip}” will collect a single slot and invoke the **FindAnimalShelterIntent**. When you delegate slot collection back to Alexa, the next elicited slot will be asked based upon the order you defined in your interaction model.

### Multiple Slots

The “find a shelter close to {city} {state}” utterance can capture multiple slots. If the customer says, “find a shelter close to Seattle, Washington,” the **city** and **state** slots will be filled. Normally dialog management will follow up by asking for the missing **zip** as we have marked it required. However, at this point our condition **A || B && C** has been met (undefined || Seattle && Washington) so our backend will jump out of dialog management and perform the look up.

### One-Shot

The “find a shelter close to {city} {state} {zip}” utterance is an example of a **one-shot invocation**, where all the information is provided in one breathe. For example, if I were to say, “Find a shelter in Seattle, Washington, 98121,” the **dialogState** will be **STARTED**, but our condition **A || B && C** has been met (98121 || Seattle && Washington), which will evaluate to true. At this point, we can either delegate back to Alexa, which will then set the **dialogState** to **COMPLETED**, or we can look up the shelter and provide the result. In the example below, we are using the former approach, which will allow us to isolate the lookup code based on the collected slots.

## 2. Activating Dialog Management

Once you’ve set up your **intent**, **utterances** and **slots**, you’ll need to activate dialog management. To do that from the developer portal, make each slot required by selecting them one-by-one and activating the **Is this slot required to fulfill the intent?** slider.

Then you will need to provide **prompts** and **utterances** just like when we set up the **PetMatchIntent**, which are phrases the user might say when prompted to fill the slots.

### 3. Check for A or B and C

Now that the frontend has been configured, now it's time to set up the backend so we can hook into the state machine and check if **A || B && C** is true and manually set **dialogState** to **COMPLETED**.

We will create four handlers:

- StartedInProgressFindAnimalShelterHandler
- HasZipFindAnimalShelterHandler
- HasCityStateFindAnimalShelterHandler
- CompletedFindAnimalShelterHandler

#### StartedInProgressFindAnimalShelterHandler

This handler will handle the request when the **dialogState** is either **STARTED** or **IN\_PROGRESS** and our condition **A || B && C** hasn't been met.

```
canHandle(handlerInput) {  
    return handlerInput.requestEnvelope.request.type === 'IntentRequest'  
        && handlerInput.requestEnvelope.request.intent.name === 'FindAnimalShelterIntent'  
        && handlerInput.requestEnvelope.request.dialogState !== 'COMPLETED';  
}
```

#### HasZipFindAnimalShelterHandler

Here we will handle requests where we have a **zip**. Our A has now been filled, which means that **A || B && C** is true! Now we can find a shelter.

```
canHandle(handlerInput) {  
    return handlerInput.requestEnvelope.request.type === 'IntentRequest'  
        && handlerInput.requestEnvelope.request.intent.name === 'FindAnimalShelterIntent'  
        && handlerInput.requestEnvelope.request.dialogState !== 'COMPLETED'  
        && handlerInput.requestEnvelope.request.intent.slots.zip.value;  
}
```

The Pet Match sample does not actually make a webservice call but it's set up so you could easily add one yourself.

```

handle(handlerInput) {

    const zip = handlerInput.requestEnvelope.request.intent.slots.zip.value;
    let outputSpeech = '';
    // make the api call.
    ...

    return handlerInput.responseBuilder
        .speak(outputSpeech)
        .getResponse();
}

```

### HasCityStateFindAnimalShelterHandler

Here we will handle requests where we have a **city** and a **state**, which means we have the **B** and **C** in **A || B && C** and that will evaluate to true so we can proceed to find a shelter.

```

canHandle(handlerInput) {
    return handlerInput.requestEnvelope.request.type === 'IntentRequest'
        && handlerInput.requestEnvelope.request.intent.name === 'FindAnimalShelterIntent'
        && handlerInput.requestEnvelope.request.dialogState !== 'COMPLETED'
        && handlerInput.requestEnvelope.request.intent.slots.city.value
        && handlerInput.requestEnvelope.request.intent.slots.state.value;
}

```

Like `HasZipFindAnimalShelterHandler` the `HasCityStateFindAnimalShelterHandler`'s `canHandle` function returns true when we have the **city** and **state**. At this point, we can simply pass these values to our web service.

```

handle(handlerInput) {

    const city = handlerInput.requestEnvelope.request.intent.slots.city.value;
    const state = handlerInput.requestEnvelope.request.intent.slots.state.value;
    let outputSpeech = '';
    // make the api call.
    ...

    return handlerInput.responseBuilder
        .speak(outputSpeech)
        .getResponse();
}

```



### CompletedFindAnimalShelterHandler

We actually don't need this handler, since our **dialogState** will never be **COMPLETED**. While we are using dialog management to manage collecting, **the zip, city and state**, it's up to us to decide when to return a **delegate** directive. Our **HasZipFindAnimalShelterHandler** and **HasCityStateFindAnimalShelterHandler** will not return a **Dialog.Delegate**. They will return a **speechOutput** since we have collected the minimum necessary slots from the customer.

Dialog management is flexible. Since it's up to you to decide when to delegate back to Alexa to collect the next slot, you can short-circuit prompting the customer upon capturing a necessary subset of data.

### Context Switching

When you have a natural conversation with another person, you might find the conversation can take different directions. Therefore, the context of the conversation can change quite rapidly. With dialog management, you can ensure your skill is able to handle context switching.

One thing to note, is that **you** are responsible for keeping track of the filled slots when switching contexts. If you don't, the dialog will start over at the beginning instead of where the customer left off. Requiring the customer to repeat the same information all over again is not a great experience, so you'll want to be sure your skill remembers the collected information in order to not exhaust the customer.

### Maintaining Context While Switching Between Intents

When using Mia's method to recommend a dog, during the conversation the customer may seek clarification. For many people, choosing one of the four options is straightforward, but it would be good to provide some help if the customer wanted to know what **large** means. For example, when Alexa asks, "There are tiny, small, medium, and large dogs, which would you like?" The customer may ask, "How many pounds is a large dog?"

To deliver an answer to the customer, we can provide a new intent called **ExplainSizeIntent** that allows the customer to ask, "How many pounds is a large dog?" which will context switch out of dialog management and provide an answer. At this point, the customer could say, "I want a large dog," which would then re-activate the **PetMatchIntent** and continue prompting the customer for the required slots.

Being able to jump back and forth between intents is a great voice-first experience because the customer isn't stuck in a menu. We refer to this approach as **frames**, in which each intent is a frame. While you're in the middle of a dialog, you can speak an utterance that allows you to access another intent and then go back to the one you were on before. Interactive voice response (IVR), which is the backbone of many automated call centers, forces the caller into a rigid path often referred to as a **tree**. There's no way move across branches without going all the way back to the top-level menu, which is unnatural and should be avoided when designing a voice experience.

Let's start with the conversation and work back to the solution:

**Customer:** Alexa, tell Pet Match I want a dog.

**Alexa:** What are two things you're looking for in a dog?

**Customer:** I want a low-energy guard dog.

**Alexa:** There are dogs that are tiny, small, medium, and large. Which would you like?

**Customer:** How many pounds is a large dog?

**Alexa:** A large dog is 55 to 77 pounds. What size dog would you like?

**Customer:** I want a large dog.

**Alexa:** So a large, low-energy guard dog sounds good for you. You should consider a Doberman Pincher.

To allow the customer to ask for clarification on size, we will need to:

1. Add a new intent
2. Save dialog management context
3. Restore dialog management context

### Add a New Intent

First, we are going to add a new intent called `ExplainSizeIntent` and define the following utterances:

How many {unitOfMeasurement} is a {size} dog

What is a {size} dog

Upon doing so, the skill will be able to identify the size the customer has inquired about so we can respond with an explanation. The **unitOfMeasurement** slot is assigned a custom slot type called **unitOfMeasurementType** that allows us to identify what unit of measurement the customer asked for. The values are **pounds**, and **kilograms**. In our backend we can look up the **size** based on the **unitOfMeasurement** using the following data structure:

```
let sizeChart = {
  "tiny": {
    "pounds": "4 to 6",
    "kilograms": "1.8 to 2.7",
  },
  "small": {
    "pounds": "7 to 20",
    "kilograms": "3.18 to 9",
  },
  "medium": {
    "pounds": "21 to 54",
    "kilograms": "9.53 to 24.49",
  },
  "large": {
    "pounds": "55 to 80",
    "kilograms": "24.94 to 38.28",
  }
}
```

We'll plug **size** and **unitOfMeasurementType** into the **sizeChart**, **sizeChart[size][unitOfMeasurement]**, to look up the size. We will also want to determine a default **unitOfMeasurement** because the **What is a {size} dog** utterance allows the user to omit it.

Dialog management supports context switching so the customer may interact with another intent during the slot collection, but you must manage the context, otherwise when the customer switches back the previously collected slots will disappear and the customer will have to provide all the slots over again.

### Save Dialog Management Context

To save the dialog management context, we will leverage session attributes. Session attributes provide a way to keep track of information throughout the life cycle of your skill. To store something into the session, simply define a key and set the desired value, **sessionAttributes['myKey'] = value**.

To support a context switch at any point, we will need to save the collected slots into the session attributes. To do this, we will hook into the dialog management state machine and update the **delegateSlotCollection** function to save the state with the following code.

Let's revisit our `canHandle` function for our `startedInProgressPetMatchHandler`.

## canHandle

```
canHandle(handlerInput) {  
    return handlerInput.requestEnvelope.request.type === 'IntentRequest'  
    && handlerInput.requestEnvelope.request.intent.name === 'PetMatchIntent'  
    && handlerInput.requestEnvelope.request.dialogState !== "COMPLETED";  
},
```

The `canHandle` will return true for the `PetMatchIntent` as long as the `dialogState` is either `STARTED` or `IN_PROGRESS`.

## Handle

The `handle` function will save the collected slots into the session attributes and restore our collected slots when we context switch back to the intent. We will name the session attribute after the intent name, which will allow us to build a skill that has multiple intents that use dialog management that can context switch and not lose any previously collected slots.

```
let currentIntent = handlerInput.requestEnvelope.request.intent;  
const { attributesManager, responseBuilder } = handlerInput;  
const sessionAttributes = attributesManager.getSessionAttributes();  
  
sessionAttributes[currentIntent.name] = currentIntent;  
attributesManager.setSessionAttributes(sessionAttributes);
```

Simply saving the session attributes alone won't help us. We have to restore the state when we get back to the `PetMatchIntent`.

## Restore Dialog Management Context

When the customer asks, "How many pounds is a large dog?" the `ExplainSizeIntent` will explain to the customer that, "A large dog is 55 to 80 pounds. What size dog would you like?" When the customer responds, "I would like a large dog", the skill will context switch back to the `PetMatchIntent`, but the `dialogState` will be `STARTED` and none of the slots that we have previously captured will be present in the request, so we will need to restore them from the session.

We will add code to restore the slots from the session attributes. To determine if we need to do so, we will check to see if the session attributes contain an attribute named after the intent. If we were returning to the `PetMatchIntent`, the session attributes would contain - **PetMatchIntent**.

```
if (sessionAttributes[currentIntent.name]) {  
    // add logic here  
}
```

Next, we loop through the set of slots that we saved in the session attributes, and for each one that has a value, restore it into the request. Doing so enables the skill to continue on through dialog management where it left off before the customer invoked the **ExplainSizeIntent**.

```
if (sessionAttributes[currentIntent.name]) {  
    const tempSlots = sessionAttributes[currentIntent.name].slots;  
    for (key in tempSlots) {  
        if (tempSlots[key].value && !currentIntent.slots[key].value) {  
            currentIntent.slots[key] = tempSlots[key]  
        }  
    }  
}
```

### Returning the Delegate Directive

In this case we **MUST** pass `currentIntent` because we modified the collected slots. If you don't, the slots will come out of sync. Passing them here will tell Alexa that you modified them and Alexa will sync its version of the collected slot values with yours.

```
return responseBuilder  
    .addDelegateDirective(currentIntent)  
    .getResponse();
```

### Complete Function

On the following page is the `ExplainSizeIntent`'s handle function in its entirety.

```
handle(handlerInput) {  
  
    let currentIntent = handlerInput.requestEnvelope.request.intent;  
    const { attributesManager, responseBuilder } = handlerInput;  
    const sessionAttributes = attributesManager.getSessionAttributes();  
  
    if (sessionAttributes[currentIntent.name]) {  
        const tempSlots = sessionAttributes[currentIntent.name].slots;  
        for (key in tempSlots) {  
            if (tempSlots[key].value && !currentIntent.slots[key].value) {  
                currentIntent.slots[key] = tempSlots[key]  
            }  
        }  
    }  
  
    sessionAttributes[currentIntent.name] = currentIntent;  
    attributesManager.setSessionAttributes(sessionAttributes);  
  
    return responseBuilder  
        .addDelegateDirective(currentIntent)  
        .getResponse();  
}
```

## Create Conversational Experiences and Delight Your Customers

Dialog management is a flexible and easy-to-use feature built into the [Alexa Skills Kit](#) that facilitates handling complex multi-turn interactions while improving the accuracy of your skill. Now that you understand how to use dialog management to its full potential, add it to your Alexa skills to create more delightful and engaging voice experiences for your customers.

### Additional Resources

[Build Advanced Alexa Skills Using Dialog Management](#)

[Codecademy: Conversational Design with Alexa](#)

[Technical Documentation: Dialog Interface Reference](#)

[Alexa Skill-Building Cookbook](#)

[Pet Match Sample Skill](#)

### Find Us on Social

Twitter: [@AlexaDevs](#), [@sleepydeveloper](#)

Facebook: [Alexa Developers](#)

LinkedIn: [Amazon Alexa Developers](#)

YouTube: [Alexa Developers](#)

Twitch: [Amazon Alexa](#)

