# Table of Contents

1

Sessions and Voice User Interfaces

In this chapter you will write a more elaborate skill, Madlib Builder. You will become familiar with how to work with *session states* within the skill, and you will also learn more about voice user interface guidelines and recommendations along the way.

Madlib Builder will accept inputs for a series of adverbs, adjectives, nouns and so forth and will build a madlib based on the completed set of steps. Once all steps have been completed, the madlib will be read back. Examine the diagrams below which further illustrate Madlib Builder's interaction flow.

Figure 1.1 Building a Madlib
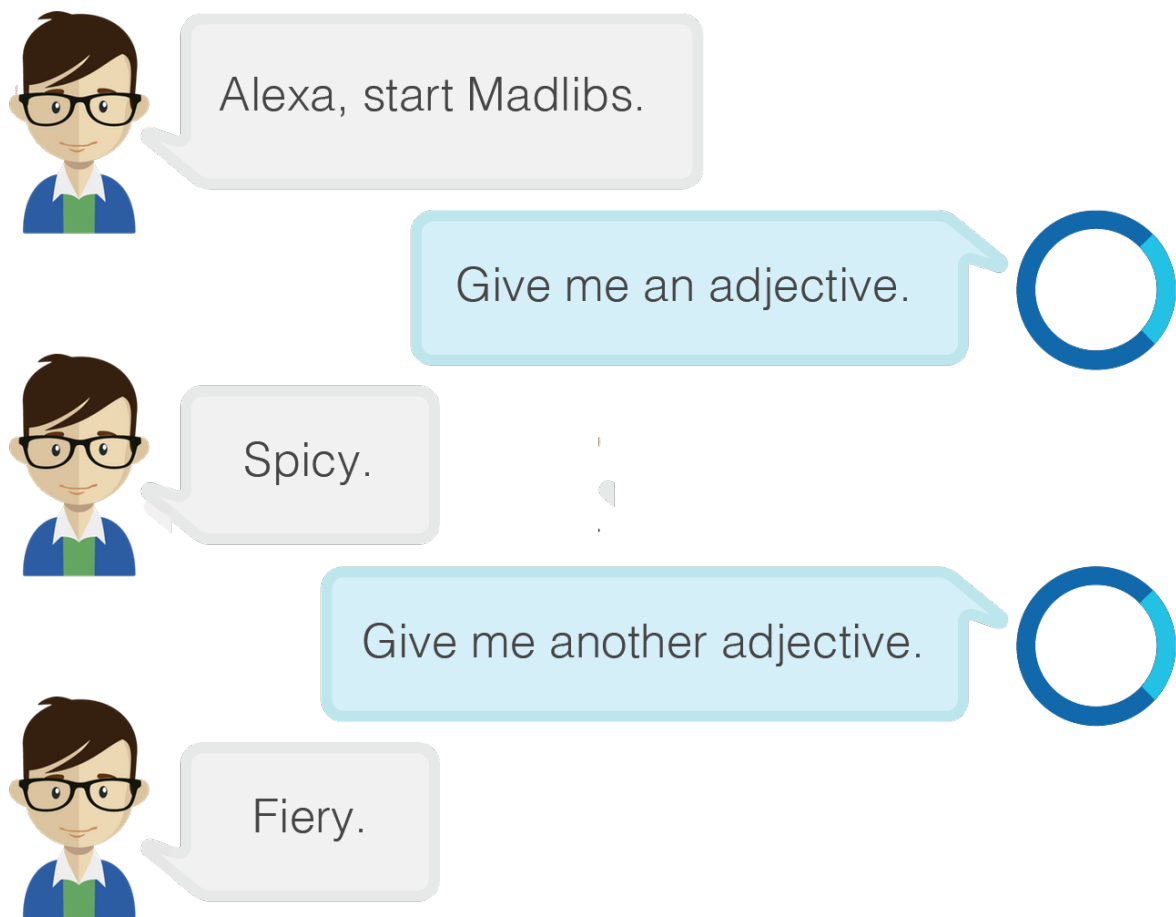
As the user provides words to Madlib Builder when prompted, the user progresses through a series of steps with their previous responses saved. You will learn about how to persist the data accumulated across requests via the session state. The session state is available in the request and response a skill service can use. This allows the skill to break a more complex set of data requirements into small steps a user can easily navigate and respond to.
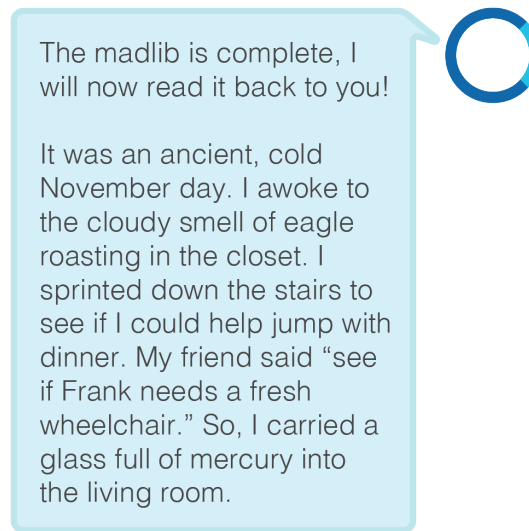
Figure 1.2 Requesting Help Along the Way

Notice that as a user completes each step, they are also able to request help (as shown in the interaction diagram above). This will guide users who may be confused about how to complete the current step. As part of creating the Madlib Builder skill, you will also learn how to implement a contextual help system to allow users to receive help information relevant to the particular step users are on in a multi-step process.

Once the user has completed all of the steps, the madlib will be read back by Alexa, often to humorous effect!
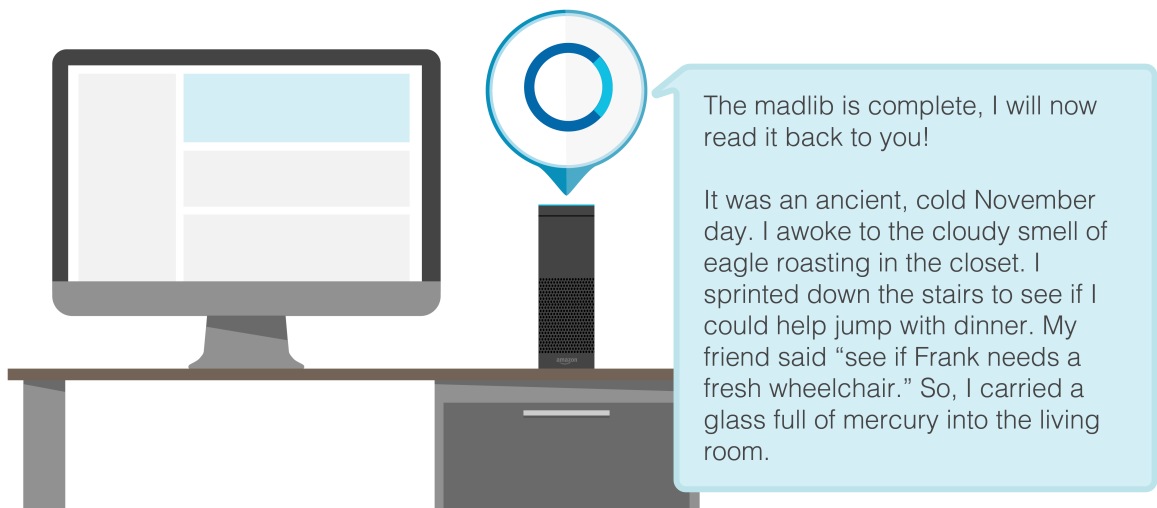
Figure 1.3 The Completed Madlib is Played Back

The madlib is complete, I will now read it back to you!

It was an ancient, cold November day. I awoke to the cloudy smell of eagle roasting in the closet. I sprinted down the stairs to see if I could help jump with dinner. My friend said "see if Frank needs a fresh wheelchair." So, I carried a glass full of mercury into the living room.

The completed madlib will also be sent as a *home card* to the Alexa app located at

```
http://alexa.amazon.com/#cards
```

Figure 1.4 A Card is Displayed in the Alexa App

The madlib is complete, I will now read it back to you!

It was an ancient, cold November day. I awoke to the cloudy smell of eagle roasting in the closet. I sprinted down the stairs to see if I could help jump with dinner. My friend said "see if Frank needs a fresh wheelchair." So, I carried a glass full of mercury into the living room.

A card is an element that can be displayed in the Alexa app and reviewed by the user at a later time. It includes a title and content body you specify. You will learn about how to display cards in the Alexa App using the Alexa Skills Kit in this chapter.
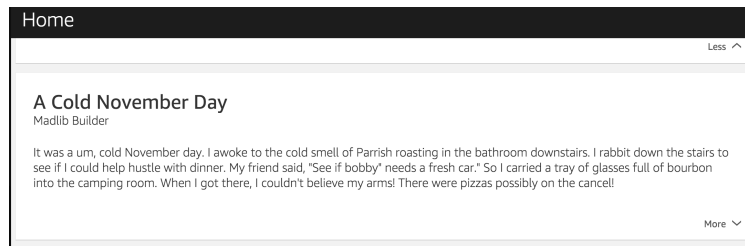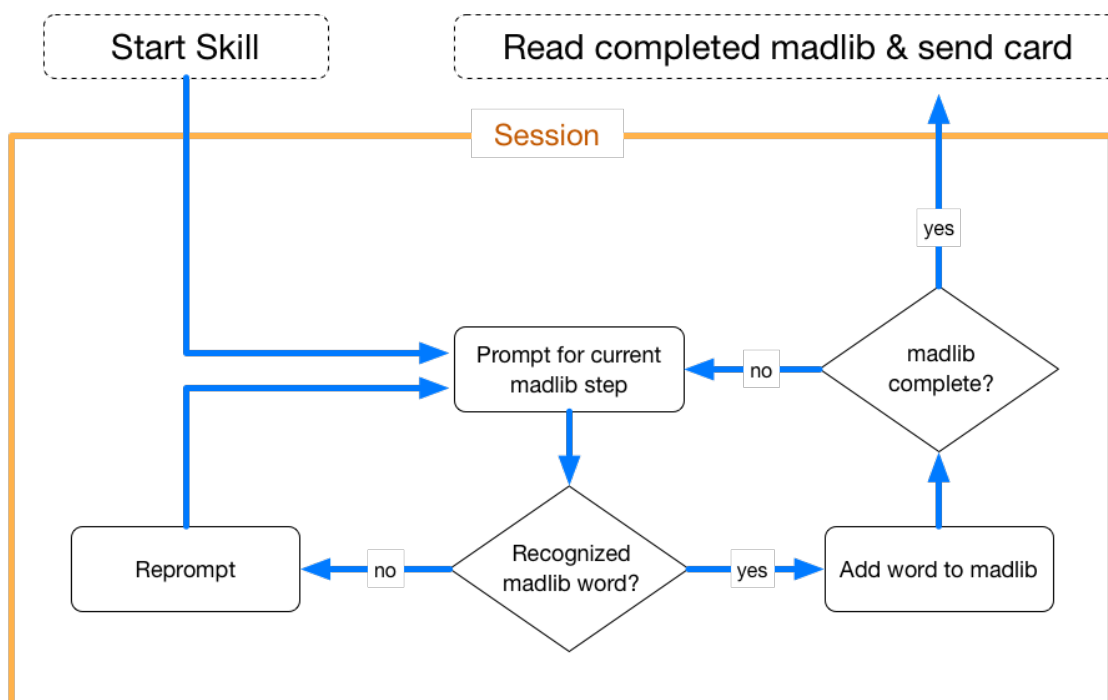
Figure 1.5 The Completed Madlib as a Card

Figure 1.6 Madlib Builder Overview

The flow chart above depicts the full path for a user interacting with the Madlib Builder skill from starting a madlib to completing one. Notice the orange "Session" rectangle. This represents the accumulation of a bank of answers for Madlib Builder. Once the session begins, the skill proceeds with prompting the user to complete the specified steps. After the steps are completed, the session ends and the completed result is spoken and displayed in a card in the companion app.

Why Use Session State?

Before beginning, a quick discussion of why using sessions can improve a skill interaction.

Consider a hypothetical skill that requires users to complete a user profile. A user profile might contain fields for first name, last name, username, twitter handle, and telephone number, for example. In a voice user interface, If all of this information was requested verbally in one large prompt, it would be overwhelming for users to answer.

Figure 1.7 A Confusing Interaction
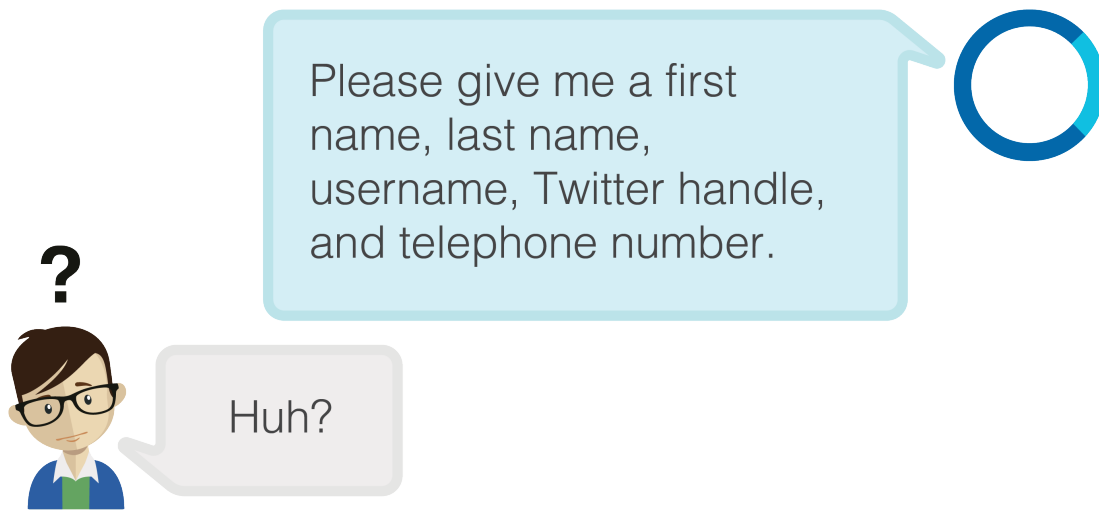
This is where using a session to keep track of the user's responses in individual steps improves the user experience of a skill. The sessions feature allows skills to break complicated data flows into a series of smaller, more focused steps so that users can follow along easily. Sessions provide the capability of keeping data across interactions for continued use within the skill. The default behavior is that the data on the skill service is removed after each customer interaction - data like the state of the madlib the customer is completing, for example.

Instead, we can store the data in the session response JSON that is sent from the skill service and it will be carried forward to the next interaction's request. When the user starts Madlib Builder, we will keep track of the state of the Madlib Builder responses in the session object. This will result in the user's responses being kept for each prompt. Answers to the prompt will be kept as long as the stream has not been closed.

Figure 1.8 Avoiding Confusion with Sessions

Getting Started

Begin by creating a new directory called `madlibbuilder` within the `alexa-app-server/examples/apps` directory.

As seen in the previous chapter, initialize a new npm `package.json` file by typing `npm init` within the `madlibbuilder` directory you created. For the name, enter `madlibbuilder` and press enter to select all of the default values for the new package.

Once completed, install the `alexa-app` and `lodash` dependencies as before.

Listing 1.1 Installing dependencies

```
npm install --save alexa-app lodash
```

The Madlib Builder skill will also make use of a **MadlibHelper** class. The **MadlibHelper** class contains the template for constructing a new madlib and formatting the output. Download `madlib_helper.js` at `https://goo.gl/oWrikM` and add it to the `madlibbuilder` directory you created.

Defining the Madlib Launch Handler

Create a new file called index.js within alexa-app-server/examples/apps/madlibbuilder. As seen in the previous chapter, you will create the skill service portion of Madlib Builder within a index.js file.

Begin by defining a new launch handler function. This will be triggered when the user speaks the utterance "Alexa, open {Invocation Name}" or "Alexa, start {Invocation Name}", where Invocation Name matches the phrase defined in the skill interface configuration.

Listing 1.2 Adding the Launch handler

```
'use strict';
module.change_code = 1;
var _ = require('lodash');
var Skill = require('alexa-app');
var skillService = new Skill.app('madlibbuilder');
var MadlibHelper = require('./madlib_helper');

skillService.launch(function(request, response) {
  var prompt = 'Welcome to Madlibs. '
    + 'To create a new madlib, say create a madlib';
  response.say(prompt).shouldEndSession(false);
});

module.exports = skillService;
```

Now that you have defined the launch handler, the skill service can handle a launch request in response to a user's spoken utterances such as "Alexa, open Madlibs", "Alexa launch Madlibs", or "Alexa, start Madlibs".

When considered from the perspective of Voice User Interface design, this type of invocation is called a *No Intent Invocation Type*.

Table 1.1 Invocation Types

Invocation Type	Description	Example
No Intent	user asks Alexa to talk to your skill without providing any further detail	Alexa open FAATracker, Alexa open madlibs
Partial Intent	user expresses just a subset of what is required for you to take action on their request	Alexa, Ask Horoscopes for a reading //needs more info - what sign?
Full Intent	user expresses everything required to complete their request (or intent) in a single utterance	Alexa, Ask Horoscopes for a Gemini reading

A No Intent handler is where you specify instructions for how to use the skill.

Figure 1.9 Handling the No Intent (onLaunch)
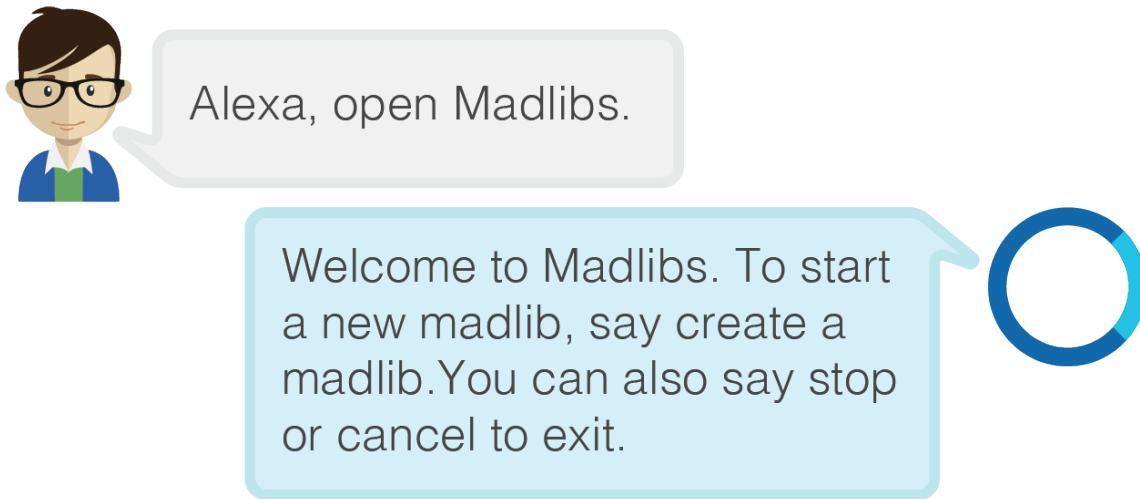
Adding an AMAZON.HelpIntent Handler

Amazon also provides "built-in" Intent Handlers for implementing common interactions. One such built-in intent is `AMAZON.HelpIntent`. You will add this intent to the intent schema definition on the skill interface, and implement a handler for it on the skill service.

To provide a concise experience that is easy for users to follow, keep the instructions the `AMAZON.NoIntent` handler returns short and to the point. As with all prompts in Voice User Interface design, it is best when the time a user must spend on a prompt is carefully considered and kept to a minimum during the interaction.

To add additional help for a more complex skill, you will implement a help system that allows users to get additional detail about a particular intent or feature. Implementing a HelpIntent is required for any skill that will be released on the Alexa Skills

Listing 1.3 Adding a HelpIntent Handler

```
skillService.launch(function(request, response) {
  var prompt = 'Welcome to Madlibs. '
    + 'To create a new madlib, say create a madlib';
  response.say(prompt).shouldEndSession(false);
});
skillService.intent('AMAZON.HelpIntent', {},
  function(request, response) {
    var help = 'Welcome to Madlibs. '
      + 'To start a new madlib, say create a madlib.'
      + 'You can also say stop or cancel to exit.';
    response.say(help).shouldEndSession(false);
});
module.exports = skillService;
```

Implementing a help system is a certification requirement outlined in the Voice User Interface guidelines. You will expand the functionality of the `AMAZON.HelpIntent` handler as you implement more of the features for the skill. The `AMAZON.HelpIntent` handler will respond with a message guiding users about the basics of using Madlib Builder.

Notice the `AMAZON.HelpIntent` handler's name is defined with an `AMAZON` prefix. The `AMAZON` prefix indicates it is a special type of intent called a *built-in intent*. There is no requirement to define utterances in the Interaction Model

settings to resolve spoken words to this special type of Intent within the skill interface. A user's spoken utterances such as "Help" or "Help Me" will resolve to the Intent and passed to the skill service without any configuration being necessary.

Built-in Intents

You just saw one of the built-in intents, `AMAZON.HelpIntent`, available with the Alexa Skills Kit. A built in intent gives you specific interaction model behavior without requiring the definition of an example utterances list. All that is required to use a built-in intent is that you add the intent to the intent schema. The following table contains the built-in intents:

Table 1.2 Built-in Intents

Method	Allowed Utterances	Purpose
AMAZON.CancelIntent	cancel, never mind, forget it	Let the user cancel a transaction or task (but remain in the skill)
AMAZON.HelpIntent	help, help me, can you help me	Provide help about how to use the skill
AMAZON.NoIntent	no, no thanks	Let the user provide a negative response to a yes/no question for confirmation.
AMAZON.RepeatIntent	repeat, say that again, repeat that	Let the user request to repeat the last action.
AMAZON.StartOverIntent	start over, restart, start again	Let the user request to restart an action, such as restarting a game or a transaction.
AMAZON.StopIntent	stop, off, shut up	Let the user stop an action
AMAZON.YesIntent	yes, yes please, sure	Let the user provide a positive response to a yes/no question

Required Built-in Intents

Your skill must implement additional Voice User Interface guidelines for it to be accepted by Amazon for distribution to Alexa-enabled devices. For Madlib Builder to meet the guidelines, you must handle additional built-in intents. An acceptable voice user interface for Madlib Builder implements `AMAZON.HelpIntent` as you have done, and will also include `AMAZON.StopIntent` and `AMAZON.CancelIntent`. Handlers for the `StopIntent` and `CancelIntent` will be required because your skill implements a long-running process that users may wich to cancel or stop at some point in time. MadlibBuilder's user experience would fail to be easily usable without implementing handlers for `StopIntent` and `CancelIntent` because users may request stopping or canceling the workflow and be ignored.

You will now implement handlers for both the `AMAZON.StopIntent` and `AMAZON.CancelIntent`. Add the following code to handle the events directly above the `AMAZON.HelpIntent` handler:

Listing 1.4 Adding AMAZON.StopIntent and AMAZON.CancelIntent

```
var cancelIntentFunction = function(request, response) {
  response.say("Goodbye!").shouldEndSession(true);
};
skillService.intent("AMAZON.CancelIntent", {}, cancelIntentFunction);
skillService.intent("AMAZON.StopIntent", {}, cancelIntentFunction);
skillService.intent('AMAZON.HelpIntent', {}, function(request, response) {
  ....
});
```

For more information about the built-in intents, check out the developer docuementation at:

```
https://goo.gl/c0A5yA
```

Adding the MadlibIntent Handler

You will now define the utterances that will trigger the `madlibIntent`. You will also define a new slot called `StepValue` that uses a slot type called `StepValues` you will later register with the skill interface. The `StepValue` will be used to retrieve a user's response to each madlib step.

Listing 1.5 Defining the Intent Handler for MadlibIntent

```
skillService.intent('AMAZON.HelpIntent', {},
  function(request, response) {
    ....
  });
skillService.intent('madlibIntent', {
    'slots': {
      'StepValue': 'StepValues'
    },
    'utterances': ['{new|start|create|begin|build} {|a|the} madlib', '{-|StepValue}']
  },
    function(request, response) {
      // madlib functionality!
    }
);
module.exports = skillService;
```

Storing the Step Value

The `madlibIntent` handler should manage storing the state of the step values. If you inspect the `madlib_helper.js` file, you will see that the step values associated with a madlib can be retrieved using the **getStep()** method. You will add logic for adding the `StepValue` slot to the current madlib step.

Listing 1.6 Storing the Step Value

```
...
skillService.intent('madlibIntent', {
    'slots': {
      'StepValue': 'StepValues'
    },
    'utterances': ['{new|start|create|begin|build} {|a|the} madlib', '{-|StepValue}']
  },

    function(request, response) {
       // madlib functionality!
      var stepValue = request.slot('StepValue');
      var madlibHelper = new MadlibHelper(madlibHelperData);
      madlibHelper.started = true;
      if (stepValue !== undefined) {
        madlibHelper.getStep().value = stepValue;
      }
      if (madlibHelper.completed()) {
        var completedMadlib = madlibHelper.buildMadlib();
        response.card(madlibHelper.currentMadlib().title, completedMadlib);
        response.say("The madlib is complete! I will now read it to you. " + completedMadlib);
        response.shouldEndSession(true);
      } else {
        response.say('Give me ' + madlibHelper.getPrompt());
        response.reprompt('I didn\'t hear anything. Give me '
        + madlibHelper.getPrompt() + ' to continue.');
        response.shouldEndSession(false);
      }
    }
);
...
```

The code you have added reads the StepValue slot that you defined in the utterances definition section of the **intent** method. You then initialize a new **MadlibHelper** and update the started property to true, indicating to the rest of the program that the madlib construction has begun. The first time the madlibIntent handler is triggered, the stepValue should be undefined because the user has not yet been prompted for a specific step of the madlib. If a value is present in the slot, it is assigned to the value attribute of the current step of the **MadlibHelper** instance.

After storing the value, the relevant prompt for the current madlib step is played. These values are specifically defined on the madlib_helper.js file you downloaded earlier.

Notice that you also made use of the **shouldEndSession** method. **shouldEndSession** will promptly close the interaction with the user and remove the values from the session array which is present on the request object if you pass true to this method. If you pass false on the other hand, the session array will be carried over to the next request as long as the skill has not been closed or exited explicitly. Remember, there is nothing magical about the **shouldEndSession** method. It simply adds an additional part to the JSON payload from the skill service to the skill interface that instructs Alexa to behave as instructed.

Storing and Retrieving the Session State

The next task in writing Madlib Builder will be introducing persistence of the steps a user has completed between requests. We will use the session state that is available on the skill service to provide this simple persistence of values. In the code above, you used **shouldEndSession** with a false value to express that you would like the session kept alive and the interaction with the user to continue. As you have set this value to false, you can now store the **MadlibHelper** in the session, and retrieve it each time the request is made for the madlibIntent handler. Add the following to index.js just after where you imported **MadlibHelper**.

Listing 1.7 Creating the getMadlibHelper(request) method

```
...
var MadlibHelper = require('./madlib_helper');
var MADLIB_BUILDER_SESSION_KEY = 'madlib_builder';
var getMadlibHelper = function(request) {
  var madlibHelperData = request.session(MADLIB_BUILDER_SESSION_KEY);
  if (madlibHelperData === undefined) {
    madlibHelperData = {};
  }
  return new MadlibHelper(madlibHelperData);
};
skillService.launch(function(request, response) {
...
```

The **getMadlibHelper** method you defined does two things. First, the method checks to see if the request object that you passed in from madlibIntent handler has an object defined on its session array under the key MADLIB_BUILDER_SESSION_KEY. It will then pass the data if it exists to the **MadlibHelper** object, where its state will be set from the previous values in the session.

Retrieving the Session State

Update the madlibIntent handler function to use the **getMadlibHelper** method you defined:

Listing 1.8 Incorporating getMadlibHelper

```
...
skillService.intent('madlibIntent', {
      'slots': {
        'StepValue': 'StepValues'
      },
      'utterances': ['{new|start|create|begin|build} {|a|the} madlib', '{-|StepValue}']
    },

    function(request, response) {
        var stepValue = request.slot('StepValue');
        var madlibHelper = new MadlibHelper(madlibHelperData);
        var madlibHelper = getMadlibHelper(request);
        madlibHelper.started = true;
        if (stepValue !== undefined) {
          madlibHelper.getStep().value = stepValue;
        }
        if (madlibHelper.completed()) {
          var completedMadlib = madlibHelper.buildMadlib();
          response.card(madlibHelper.currentMadlib().title, completedMadlib);
          response.say("The madlib is complete! I will now read it to you. " + completedMadlib);
          response.shouldEndSession(true);
        } else {
          if (stepValue !== undefined) {
            madlibHelper.currentStep++;
          }
          response.say('Give me ' + madlibHelper.getPrompt());
          response.reprompt('I didn\'t hear anything. Give me ' + madlibHelper.getPrompt() + ' to continue.');
          response.shouldEndSession(false);
        }
        response.session(MADLIB_BUILDER_SESSION_KEY, madlibHelper);
    }
);
...
```

The previous changes increment the current step if the step value is defined, and persists **MadlibHelper** by adding it to the response session. When **getMadlibBuilder** is called with the request object as a parameter, the **MadlibHelper** object you added in the previous request can be retrieved if available. The result is that the steps of the madlib can now be advanced as each request is made. As the previous state is preserved, the data that is added with each request will be used when all of the steps are completed to build the madlib.

Making Help Contextually-Aware

You can use the **getMadlibHelper** method from the AMAZON.HelpIntent handler to provide help relevant to the particular step a user is on as they complete a madlib. Update the AMAZON.HelpIntent handler method to the following:

Listing 1.9 Reading the help data from MadlibHelper

```
skillService.intent('AMAZON.HelpIntent', {},
  function(request, response) {
    var madlibHelper = getMadlibHelper(request);
    var help = 'Welcome to Madlibs. '
      + 'To start a new madlib, say create a madlib.'
      + 'You can also say stop or cancel to exit.';
    if (madlibHelper.started) {
      help = madlibHelper.getStep().help;
    }
    response.say(help).shouldEndSession(false);
});
```

Now, if a user requests help while completing a madlib, the relevant help data for that particular step will be given by Alexa.

Testing the Skill

Before deploying, test that the skill works correctly in the local development environment. Start up `alexa-app-server` as you did previously using the command: `node server` from the `alexa-app-server/examples` directory. After running the command, visit `http://localhost:8080/alexa/madlibbuilder`. You should see the Alexa skill testing interface.

First, test that the launch request behaves as expected. In the Type dropdown, select LaunchRequest and click Send Request. You should see `outputSpeech` matching the following in the Response area:

```
{
  "version": "1.0",
  "sessionAttributes": {},
  "response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Welcome to Madlibs.
      To create a new madlib, say create a madlib</speak>"
    }
  },
  "dummy": "text"
}
```

This tests the "No Intent" invocation you configured earlier. Next, test that the steps for the `madLibIntent` can be completed as you would expect. Select Intent Request in the Type dropdown, `madlibIntent` for the Intent, and click "Send Request" without entering a value for `StepValue`. Inspect the Response area. You should see text similar to the following:

```
{
 "version": "1.0",
 "sessionAttributes": {
   "madlib_builder": {
     "started": true,
     "madlibIndex": 0,
     "currentStep": 0,
     "madlibs": [
       {
         "title": "A Cold November Day",
         "template": "It was a ${adjective_1}, cold November day. I awoke to the ${adjective_2}
         smell of ${type_of_bird} roasting in the ${room_in_house} downstairs.
          I ${verb_past_tense} down the stairs to see if I could help ${verb} with dinner.
           My friend said, \"See if ${relative_name}\" needs a fresh ${noun_1}.\"
           So I carried a tray of glasses full of ${a_liquid} into the ${verb_ending_in_ing} room.
            When I got there, I couldn't believe my ${part_of_body_plural}!
             There were ${plural_noun} ${verb_ending_in_ing_2} on the ${noun_2}!",
         "steps": [
           {
             "value": null,
             "template_key": "adjective_1",
             "prompt": "an Adjective",
             "help": "Speak an adjective to add it to the madlib.
              An adjective is a word that modifies a noun (or pronoun) to make it more
              specific: a rotten egg, a cloudy day, or a tall, cool glass of water.
              What adjective would you like?"
           },
           ....
           {
             "value": null,
             "template_key": "verb_ending_in_ing_2",
             "prompt": "a verb ending in ing",
             "help": "Speak a verb ending in ing to add it to the madlib.
              Running, living, or singing are all examples. What verb ending in ing do you want to add?"
           },
           {
```

```
                    "value": null,
                    "template_key": "noun_2",
                    "prompt": "a noun",
                    "help": "Speak a noun to add it to the madlib.
                    A noun used to identify any of a class of people, places, or things.
                    What noun would you like?"
                }
            ]
        }
    ]
    }
},
"response": {
  "shouldEndSession": false,
  "outputSpeech": {
    "type": "SSML",
    "ssml": "<speak>Give me an Adjective</speak>"
  },
  "reprompt": {
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>I didn't hear anything. Give me an Adjective to continue.</speak>"
    }
  }
},
"dummy": "text"
}
```

Notice the state of the MadlibHelper has been copied into the sessionAttributes portion of the response. The currentStep value is 0, and should remain 0 on subsequent requests if no value is provided to StepValue. The message contained in the outputSpeech portion of the response should also contain a message containing the prompt text matching the current step of the madlib.

Next, test that providing a value adds the value to the steps array. Enter "test" for StepValue and press Send Request. You should observe the currentStep was incremented by 1 and that the value "test" was added to the steps in the Response box in the sessionAttributes part of the response.

Listing 1.10 The Session Attributes (partial listing)

```
{
  "version": "1.0",
  "sessionAttributes": {
    "madlib_builder": {
      "started": true,
      "madlibIndex": 0,
      "currentStep": 1,
      "madlibs": [
        {
          "title": "A Cold November Day",
          "template": "It was a ${adjective_1}, cold November day...",
          "steps": [
            {
              "value": "Test",
              "template_key": "adjective_1",
              "prompt": "an Adjective",
              "help": "Speak an adjective to add it to the madlib.
              An adjective is a word that modifies a noun (or pronoun)
              to make it more specific: a rotten egg, a cloudy day, or a tall, cool glass of water.
              What adjective would you like?"
            },
            {
             "value": null,
             "template_key": "adjective_2",
             "prompt": "another Adjective",
             "help": "An adjective is a word that modifies a noun (or pronoun)
              to make it more specific: a rotten egg, a cloudy day, or a tall, cool glass of water.
              What adjective would you like?"
            },
```

...listing continues (abbreviated)...

You should also notice that with each step taken, the outputSpeech changes to the relevant prompt for the subsequent step. If the currentStep value is 2 for example, the response outputSpeech should look like:

```
"response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Give me a Type of bird</speak>"
    },
```

Advancing the step to 3 by hitting Send Request again should show:

```
"response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Give me a name of Room in a house</speak>"
    },
```

Continue to advance steps to step 13 by pressing Send Request. When you reach step 13 the intent should have received all of the values to complete the madlib. The response should contain the completed madlib. It should also contain the data for creating the card you requested.

```
    "response": {
        "shouldEndSession": true,
        "card": {
          "type": "Simple",
          "title": "A Cold November Day",
```

```
          "content": "It was a Test, cold November day.
          I awoke to the Test smell of Test roasting in the Test downstairs.
          I Test down the stairs to see if I could help Test with dinner.
          My friend said, \"See if Test\" needs a fresh Test.\"
          So I carried a tray of glasses full of Test into the Test room.
          When I got there, I couldn't believe my Test! There were Test Test on the Test!",
          "subtitle": "your completed madlib"
        },
        "outputSpeech": {
          "type": "SSML",
          "ssml": "<speak>The madlib is complete! I will now read it to you.
          It was a Test, cold November day.
          I awoke to the Test smell of Test roasting in the Test downstairs.
          I Test down the stairs to see if I could help Test with dinner.
          My friend said, \"See if Test\" needs a fresh Test.\"
          So I carried a tray of glasses full of Test into the Test room.
          When I got there, I couldn't believe my Test!
          There were Test Test on the Test!</speak>"
        }
      },
```

Before continuing to the next step, copy the values from Schema and Utterances at the bottom of the test page to your clipboard - you will require them in the next step.

Deploying the Skill Service

The skill service has been tested locally and it is now time to deploy it to AWS Lambda to make it live. Create an archive of the contents of the madlibbuilder directory.

Figure 1.10 Creating an Archive of the Skill Service
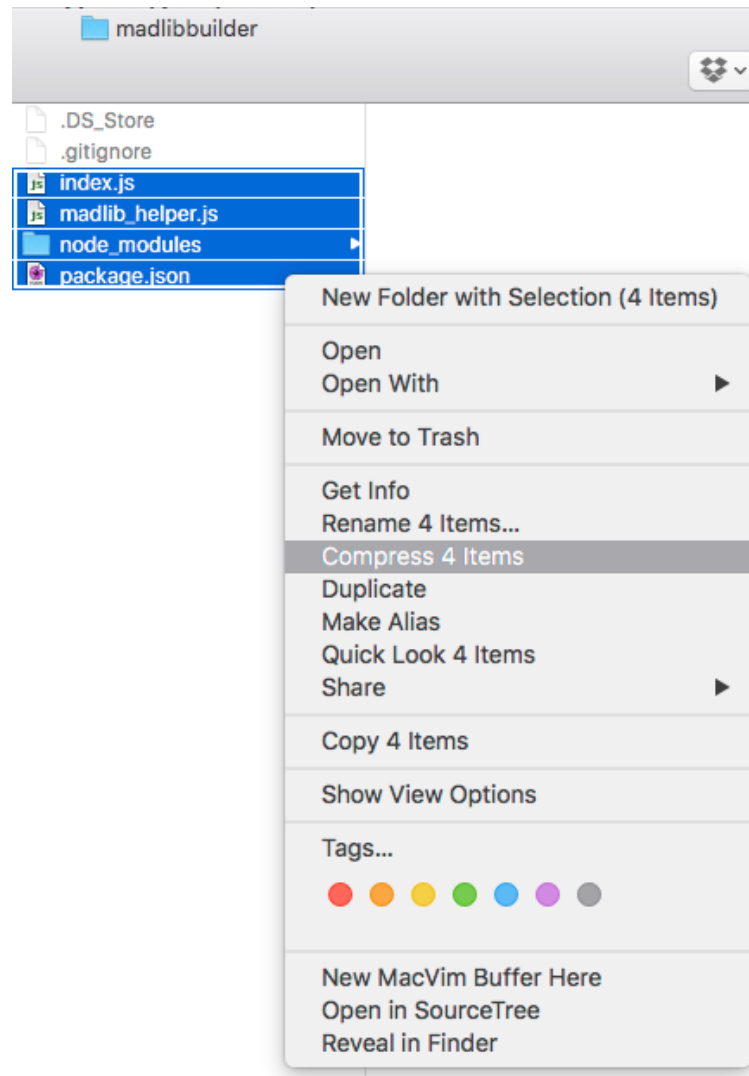
Next, you need to create an AWS Lambda function to host the archive. Visit the url

```
https://console.aws.amazon.com/lambda/home?region=us-east-1#
```

and create a new Lambda function and upload the archive using the steps from chapter one, giving the function a name of "madlibService". Remember to copy the ARN down that appears at the top right of the screen as you did before, as it will be needed in the skill interface configuration.

Configuring the Skill Interface

Now that you have completed setting up the skill service, you will set up the skill interface. Visit the url

```
https://developer.amazon.com/edw/home.html#/skills/list
```

and click Add a New Skill. Enter "Madlib Builder" for Name and "madlibs" for Invocation Name. Paste the ARN that you copied down from the Lambda configuration in the field for Endpoint.

Figure 1.11 Configuring the Skill Interface

Click the Next button to advance to the Interaction Model step. Here, you will paste the values into the Intent Schema and Sample Utterances fields that you copied to your clipboard.

Figure 1.12 Configuring the Interaction Model

Last, define a custom slot type for the StepValue value. Download the content for the StepValues here :

https://goo.gl/mzWMY4

The values are a composite of nouns and adjectives that will provide training data to the natural understanding capabilities of the interaction model. The interaction model will use this sample data to correctly recognize the words users provide for the slot during each madlib step. Once you have downloaded the list, click Add Slot Type. Name the Slot Type "StepValues" and paste the content of the list you downloaded in the Enter Values area.

Figure 1.13 Defining the StepValues Slot Type

Testing the Skill

Advance to the Test Skill page of the Skill Interface. Here you will be presented a dialog that allows you to test the interaction for the skill. Enter "start madlib" under the Enter Utterance input area and click Ask Madlib Builder. Inspect the Lambda response. You should receive text similar to the following:

Figure 1.14 Testing in the Service Simulator

Test that advancing through the steps behaves the same as how the local development environment behaved when testing the skill.

Congratulations, you have completed Madlib Builder! You have learned about the session management capabilities of the Alexa Skills Kit, and you have also expanded your knowledge of some voice interface design practices.

Osmium Challenge

Your user enjoys completing the madlib, but desires more variety in the selection of madlibs! Create your own madlib template and prompts, and implement the ability for the user to select a madlib before starting the step process for their selection. This will involve adding a new intent for specifying which madlib a user would like, a new madlib template, and logic for indexing into the correct madlib using the selection that the user makes initially. Good Luck!

Figure 1.15 Challenge - Madlib Template Chooser

Figure 1.16 The Madlib Template Chooser is Added to the Workflow