Developing Alexa Skills

the big nerd ranch guide

Developing Alexa Skills: The Big Nerd Ranch Guide

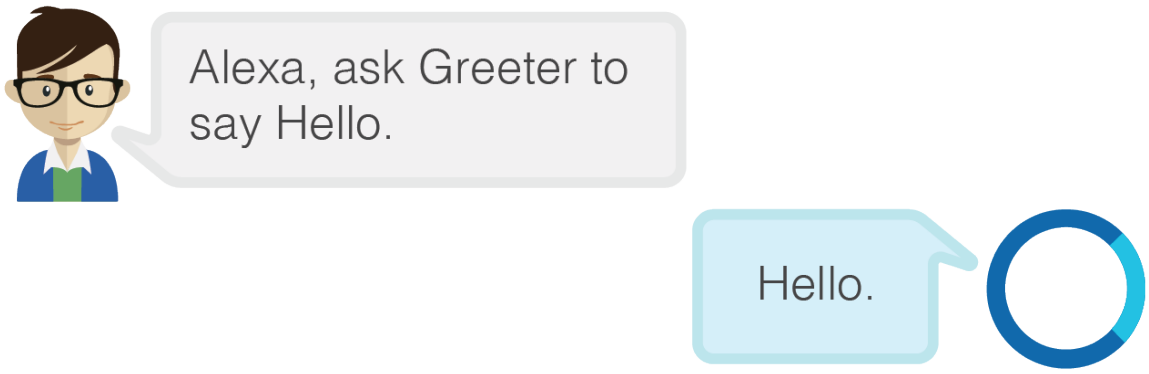
Table of Contents

1

Hello Alexa

In this chapter, you will build and deploy a basic Alexa skill called "Greeter". This exercise will leverage the basic requirements to get a skill up and running, and show you the basic components of a skill. The illustration shows an interaction with the Greeter skill. The user invokes Greeter by saying "Alexa, ask Greeter to say Hello" and Alexa responds with "Hello".

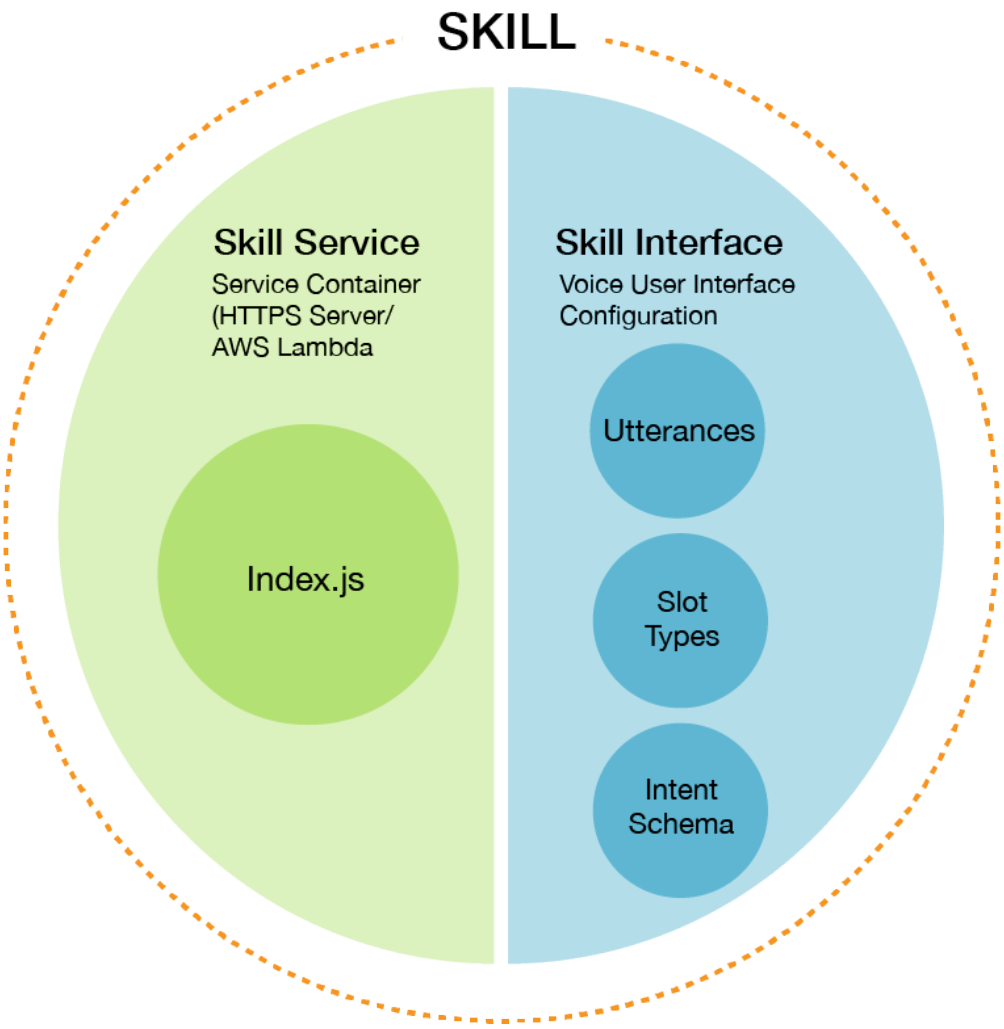
Figure 1.1



The Greeter skill will consist of two portions - a skill *service* you will deploy to a remote service container, and a skill *interface* you will register on Amazon's Alexa skill console. For the service portion, you will be working with AWS Lambda, a cloud-based service hosting platform and server environment.

While there are many options for hosting your skill's service code over HTTPS, you will be using AWS Lambda because of it's streamlined interface within the Alexa skill development pipeline. It is possible to use your own HTTPS server, but to do so it requires additional configuration to enable SSL and a signed certificate. No additional configuration is required with AWS Lambda.

Figure 1.2 A skill consists of a skill service and skill interface



Creating an Account

Before beginning development, you should ensure that you have registered an AWS account with Amazon first. Visit

```
https://aws.amazon.com/lambda/
```

and log in. Check that you have access to the AWS Lambda console located there and that your Lambda account is active. If not, register an account and follow the steps for enabling AWS Lambda on your AWS account as prompted in the signup process.

You should also download the course solutions from the github repository for your reference:

```
https://github.com/bignerdranch/developing-alexa-skills-solutions/raw/master/solutions.zip
```

Creating the Skill Service

You will begin by building the service portion of the skill. Now that you have verified an active AWS Lambda account, you will begin developing the skill by creating a new directory called `greeter`. Download the base `AlexaSkill` module, located at

```
https://raw.githubusercontent.com/amzn/alexa-skills-kit-js/master/samples/scoreKeeper/src/AlexaSkill.js
```

`AlexaSkill.js` will provide a base set of functionality for your skill that you will extend to flesh out a skill service with. Save this file to the `greeter` directory you created.

An Alexa skill service can be written using any platform that may be hosted on an HTTPS server endpoint. For this course, you will be working with JavaScript and the Node.js runtime. Node.js is a supported language on AWS Lambda and is supported by a vibrant open source community. Node.js is also convenient to develop in and debug, requiring a minimal toolchain for development with. The Node.js runtime enables JavaScript to run on the server. You will be deploying to AWS Lambda, a cloud based server environment that supports the Node.js runtime.

Defining index.js

Within the `greeter` directory add a new file called `index.js`, where you will now begin implementing the skill service for Greeter using Node.js/JavaScript.

Listing 1.1 Creating greeter/index.js

```
'use strict';
var APP_ID = undefined;
var AlexaSkill = require('./AlexaSkill');

var GreeterService = function() {
  AlexaSkill.call(this, APP_ID);
};
GreeterService.prototype = Object.create(AlexaSkill.prototype);
```

You have now defined a **`GreeterService`** function which inherits from the `AlexaSkill.js` class. In the next sections you will build upon this base, defining how the skill service will handle requests from the skill interface.

Notice you have also enabled strict mode in your JavaScript file. This will help catch common JavaScript programming blunders. For example when strict mode has been enabled an error will be thrown if you assign a string to an undefined variable. It is advised to always use strict mode in JavaScript programs. For a detailed (somewhat lengthy) write-up of how strict mode works, check out

```
http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/.
```

Defining the onLaunch handler

You will next add an **`onLaunch`** event handler to the GreeterService function. The **`onLaunch`** event handler will be invoked when the user first launches or opens the skill with it's *invocation name*, which you will specify later. To add the **`onLaunch`** event handler, add the following to `index.js`:

Listing 1.2 Adding an onLaunch Event Handler

```
'use strict';
var APP_ID = undefined;
var AlexaSkill = require('./AlexaSkill');
var SPEECH_OUTPUT = 'Hello';

var GreeterService = function() {
  AlexaSkill.call(this, APP_ID);
};
GreeterService.prototype = Object.create(AlexaSkill.prototype);

var helloResponseFunction = function(intent, session, response) {
  response.tell(SPEECH_OUTPUT);
};

GreeterService.prototype.eventHandlers.onLaunch = helloResponseFunction;
```

You also defined a **`helloResponseFunction`** function. This function will build a response to the Alexa skill interface that tells Alexa how to respond to the user's request. The **`onLaunch`** event will be fired by the skill interface and sent

to the service when the Alexa skill is started with nothing else said. For example, the **`onLaunch`** event is triggered if the skill is invoked with the phrase "Alexa, open Greeter" or "Alexa, start Greeter". You will learn more about the specifics of how the skill interface is configured soon.

Defining an Intent Handler

An *intent* is a description of what a user would like to accomplish that is sent to the skill service from the skill interface. To define how the service will handle the intent, you will add what is called an *intent handler*.

A user's request is resolved to the handler by providing the skill interface a list of "utterances" you will soon configure. In this simple example, the **`onLaunch`** handler will do the same thing your intent handler will do - respond with "Hello". You will assign the same **`helloResponseFunction`** to the new intent handler definition. Modify `index.js` to include a 'HelloWorldIntent' intent handler.

```
'use strict';
var APP_ID = undefined;
var AlexaSkill = require('./AlexaSkill');
var SPEECH_OUTPUT = 'Hello';

var GreeterService = function() {
  AlexaSkill.call(this, APP_ID);
};
GreeterService.prototype = Object.create(AlexaSkill.prototype);

var helloResponseFunction = function(intent, session, response) {
  response.tell(SPEECH_OUTPUT);
};
GreeterService.prototype.eventHandlers.onLaunch = helloResponseFunction;

GreeterService.prototype.intentHandlers = {
  'HelloWorldIntent': helloResponseFunction
};
```

Next, you will add a Lambda handler method definition. This definition will allow the skill service you have written to run on the AWS Lambda platform correctly.

Listing 1.3 Adding the AWS Lambda Handler

```
...
var helloResponseFunction = function(intent, session, response) {
  response.tell(SPEECH_OUTPUT);
};
GreeterService.prototype.eventHandlers.onLaunch = helloResponseFunction;

GreeterService.prototype.intentHandlers = {
  'HelloWorldIntent': helloResponseFunction
};

exports.handler = function(event, context) {
  var greeterService = new GreeterService();
  greeterService.execute(event, context);
};
```

Now that you have added the handler, AWS Lambda will be able to route the event and context information sent from the skill interface to your skill service as JSON data. This JSON payload includes session, environment, and information about the request from the Alexa account from which the skill was invoked. You will use more of these attributes as you work through the course.

Obtaining an Application ID

Before adding the skill service code you have written to AWS Lambda, you will need an Application ID from the Amazon skill interface. This will ensure the requests made from the skill interface to the skill service are from the correct source.

Go to the page

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

and click add a New Skill. Enter "Greeter" for Name, and "Greeter" for *invocation name*. The *invocation name* you specify here is how customers will address your skill when speaking with Alexa. In the diagram below, the example interaction shows the invocation name in use with the Greeter skill.

Figure 1.3 Defining the Invocation Name

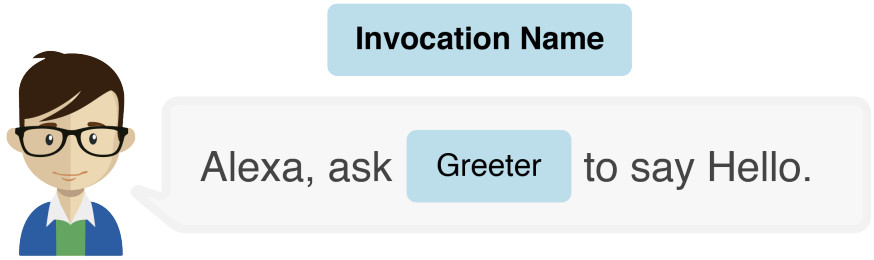
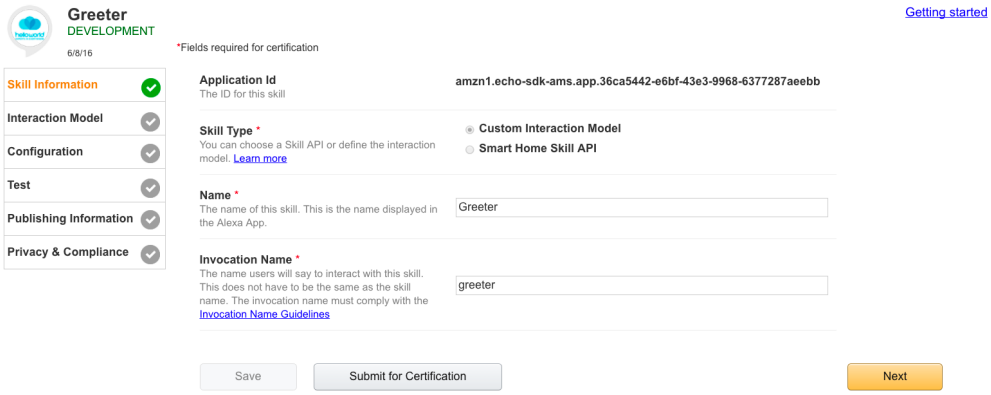


Figure 1.4 Copying the Application ID



Copy the Application ID value from the Skill Information step, and update the `APP_ID` variable in `index.js`.

Listing 1.4 Adding the Application ID

```
'use strict';
var APP_ID = undefined;
var APP_ID = 'amzn1.echo-sdk-ams.app.21133313-882b-4dcf-a90a-123123dd1ad';
var AlexaSkill = require('./AlexaSkill');
var SPEECH_OUTPUT = 'Hello World!';

var GreeterService = function() {
  AlexaSkill.call(this, APP_ID);
};
...
```

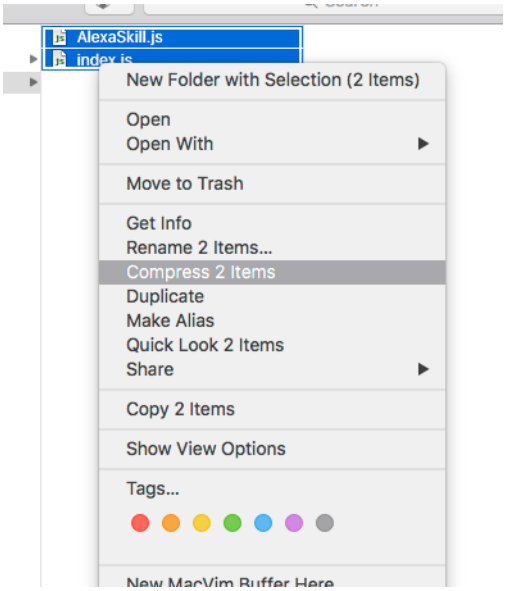
Deploying the Service to AWS Lambda

Next, you deploy the skill service to AWS Lambda. To begin deployment, visit:

```
https://console.aws.amazon.com/console/
```

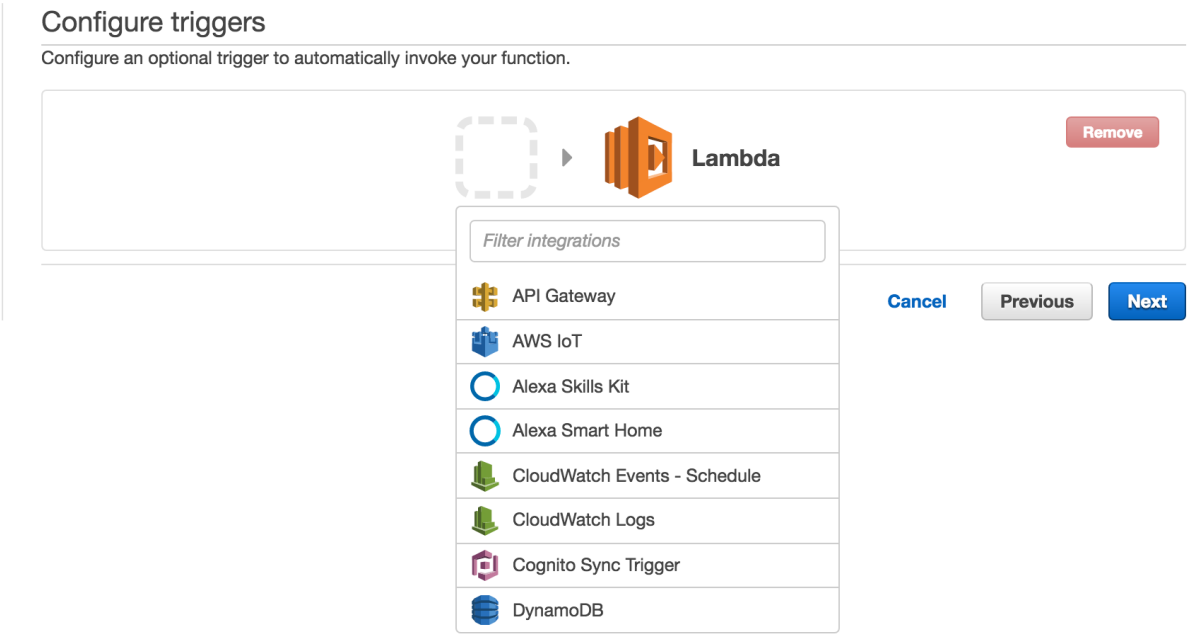
and click on Lambda. To upload the skill service, create an archive of the files within `greeter` by selecting all of the files in the directory, control clicking and selecting Compress.

Figure 1.5 Compressing the Skill Service Files



Next, click on Get Started Now or Create Lambda Function, depending on if you have used the interface before (Create Lambda Function will show if you have previously used AWS Lambda). On the Select blueprint page, click Next without selecting a blueprint. You will now be taken to the Configure Triggers page. Click the gray rounded square, select "Alexa Skills Kit" from the list, and click "Next".

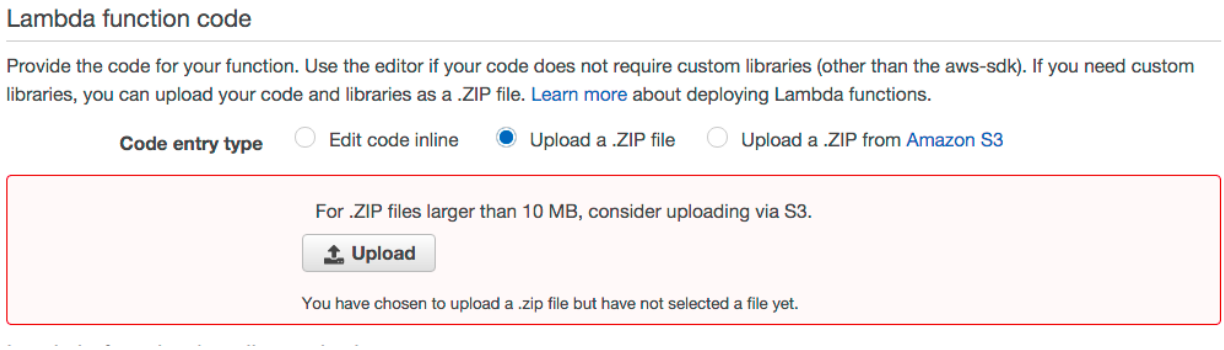
Figure 1.6 Selecting the Skills Kit Trigger



Now you will configure the AWS Lambda function. For the Name field, enter "GreeterService". For the Runtime field, select "Node.js 4.3".

Select Upload a .ZIP file under Code entry type and upload the archive you created previously by selecting it after clicking the Upload button.

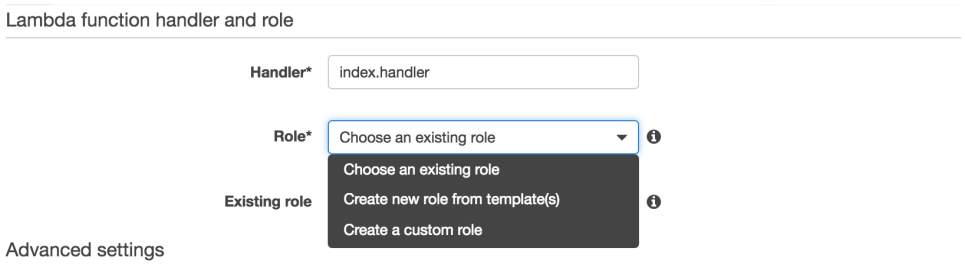
Figure 1.7 Uploading the Skill Service Archive



Next, for the Lambda instance to execute correctly, you must configure the correct permissions setttings. You first will check to see if the expected role is available under the Roles list. For Role, select Choose an existing role. Under the Existing role dropdown, check to see whether lambda_basic_execution is present or not as an option. If it is, select it.

If not present in the Existing role list, you must create a custom role. Under Lambda function handler and role select Create a custom role.

Figure 1.8 Select Create a Custom Role

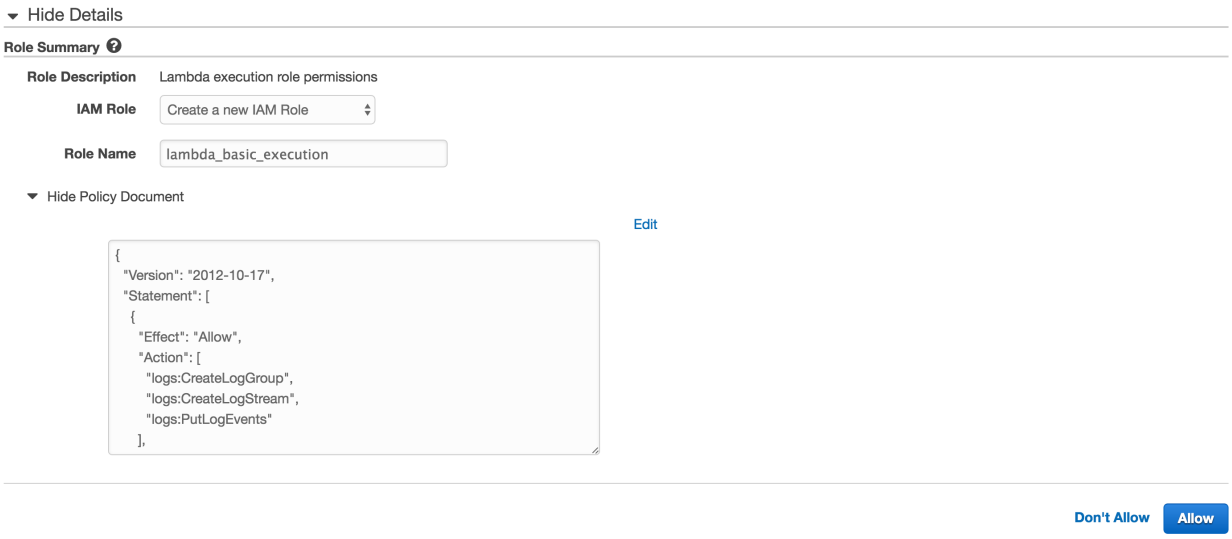


You will be redirected to a new page where you will specify the rules for the permission. In the IAM Role Dropdown, select Create new IAM Role. For Role Name, enter `lambda_basic_execution`, and click View Policy Document. Click the Edit text label that is now visible. Within the text entry field for the policy document, enter the following:

Listing 1.5 lambda_basic_execution Policy Document

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

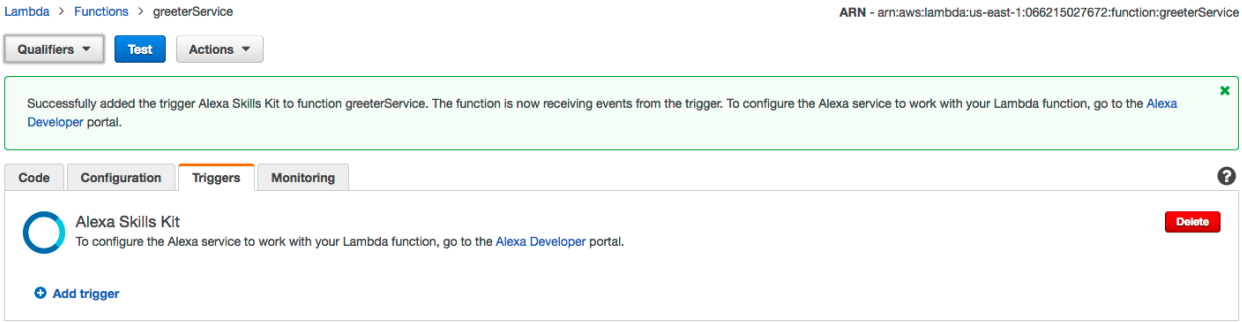
Figure 1.9 Create a New Role



Last, click Allow. You will be redirected to the lambda function creation page. Ensure that Choose an existing role is selected under Role and the new lambda_basic_execution role you created is selected under Existing role.

Now, click Next and Create Function. On the resulting screen, click Triggers and ensure Alexa Skills Kit is present. If it is not, click Add trigger. For Triggers, select Alexa Skills Kit and click Submit.

Figure 1.10 Alexa Skills Kit Trigger Enabled



Notice the value displayed in the top right of the AWS Lambda Management Console. This is the ARN, or *Amazon Resource Name*, which you will use to configure the skill interface. The ARN serves as an address that points to your skill service and indicates where requests should be routed. You will require the ARN value in several steps - copy the ARN down to a text file so that you have it ready when needed.

The skill service should now be ready to receive requests from the skill interface. Next, you verify the skill service works correctly. Click the Actions dropdown and select Configure test event.

Figure 1.11 Configuring the Test Event 1/2

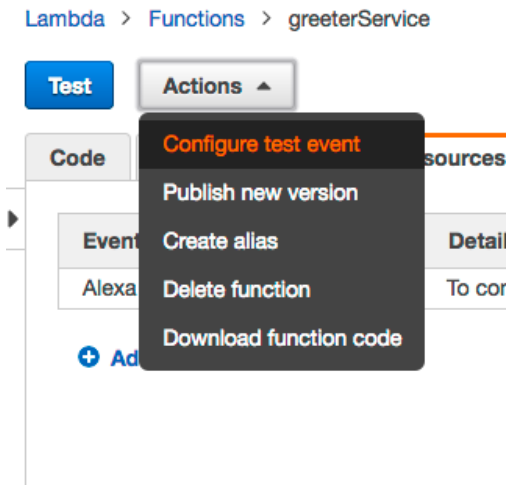
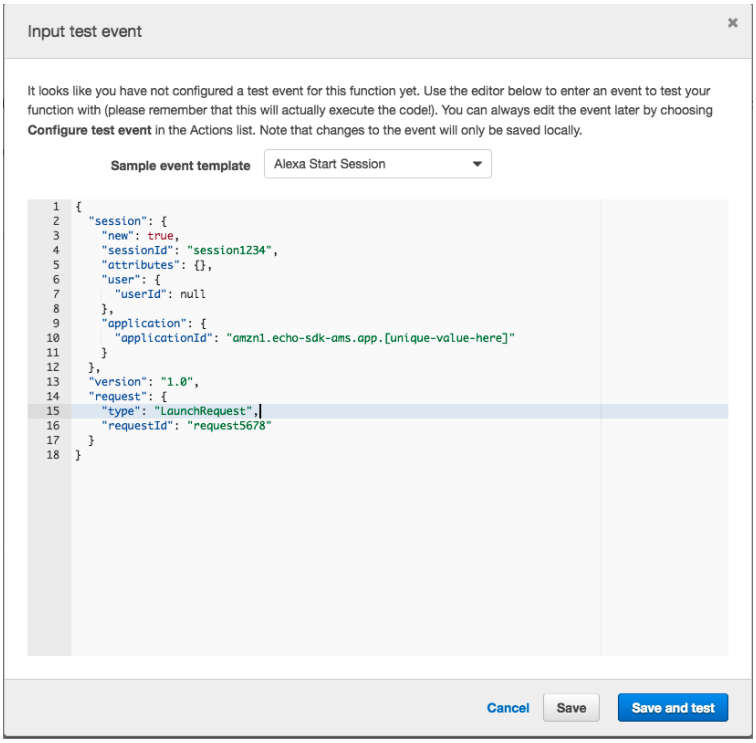


Figure 1.12 Configuring the Test Event 2/2



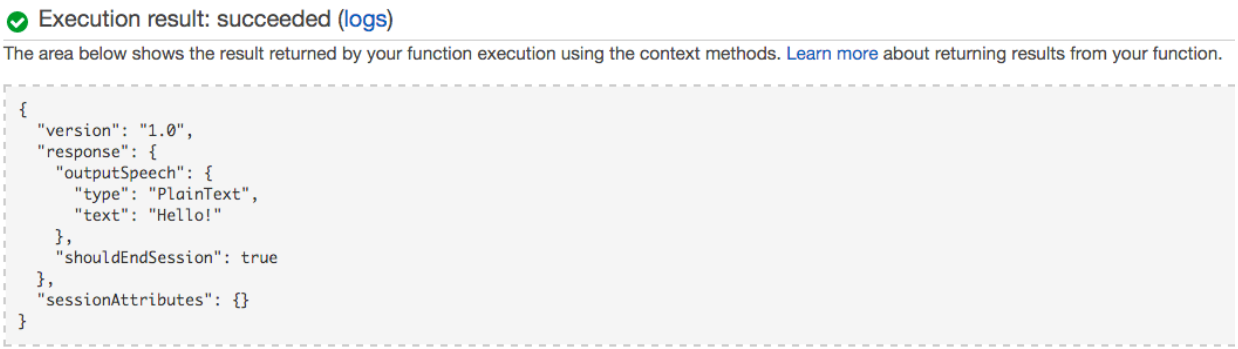
Select Alexa Start Session under the Sample event template. Before clicking Save and test you will need to change the `applicationId` value in the request template. Use the Application ID you obtained from the skill interface previously.

Listing 1.6 Adding Application ID to the Test Event

```
{
  "session": {
    "new": true,
    "sessionId": "session1234",
    "attributes": {},
    "user": {
      "userId": null
    },
    "application": {
      "applicationId": "amzn1.echo-sdk-ams.app.[unique-value-here]"
      "applicationId": "amzn1.echo-sdk-ams.app.36ca5442-e6bf-43e3-9968-6377287aeebb"
    }
  },
  "version": "1.0",
  "request": {
    "type": "LaunchRequest",
    "requestId": "request5678"
  }
}
```

Press the Save and test button. You should see something similar to the following appear in the Execution Result pane:

Figure 1.13 Inspecting the Execution Results



Configuring the Alexa Skill Interface

Notice the data that is displayed in the Execution Results pane. This is the JSON response from the skill service with a payload instructing the Alexa-enabled device to say "hello".

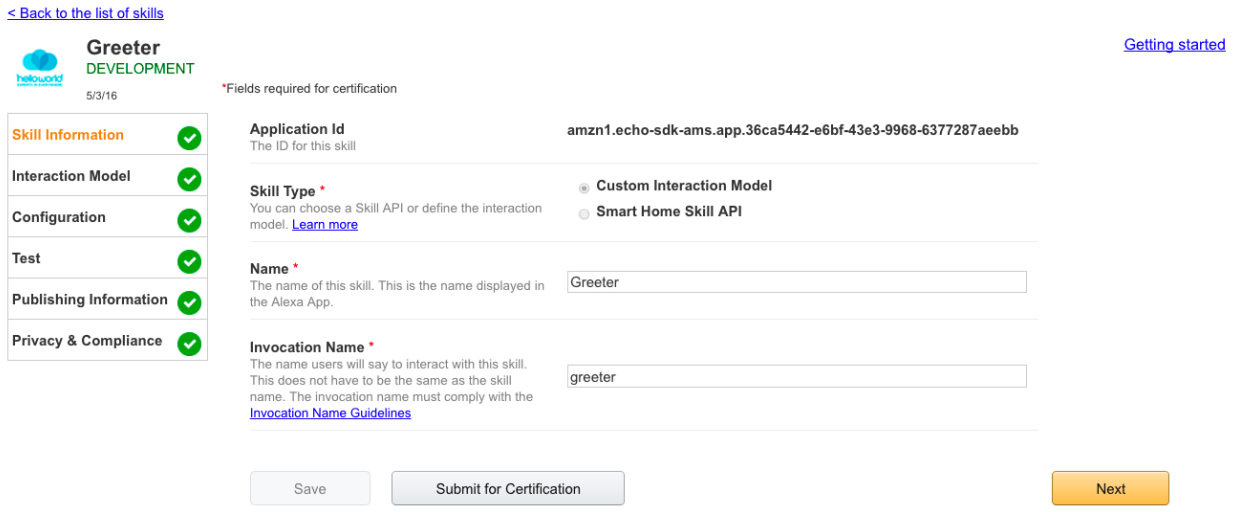
Now that the skill service is set up, you configure the skill interface in the Alexa Developer console. Visit

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

and click on the "Greeter" skill you began configuring earlier.

Verify that within Skill Information you have Name and Invocation name set to Greeter.

Figure 1.14 Configuring the Skill Information



Defining an Intent Schema and Sample Utterances

Click on Interaction Model in the side area. You will provide an *intent schema* and *sample utterances* list the skill interface configuration. Under the Intent Schema field enter the following:

Listing 1.7 Adding application ID to the Test Event

```
{
  "intents": [
    {
      "intent": "HelloWorldIntent"
    }
  ]
}
```

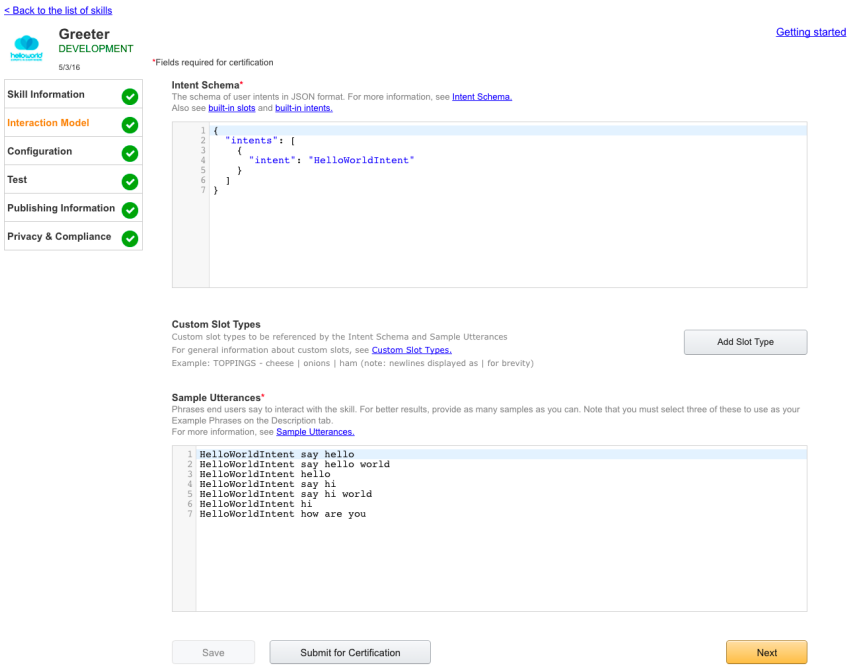
This schema will let the skill interface know that your skill can handle an `Intent` called `HelloWorldIntent` that can be resolved to by spoken utterance. The intent name must be included in the intent schema for it to be resolved by the skill interface.

Next, you provide *sample utterances* for the `HelloWorldIntent`. In the Sample Utterances field, enter the following:

Listing 1.8 Sample Utterances for Greeter

```
HelloWorldIntent say hello
HelloWorldIntent say hello world
HelloWorldIntent hello
HelloWorldIntent say hi
HelloWorldIntent say hi world
HelloWorldIntent say hey there world
HelloWorldIntent hi
HelloWorldIntent how are you
```

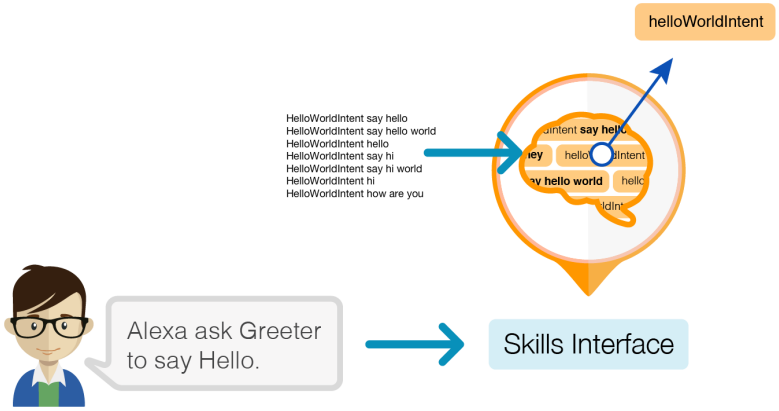
Figure 1.15 configuring utterance and interaction model



This list of sample utterances allows the Alexa skill interface to resolve spoken words to an intent. It is not required that all of the words that are specific to matching the intent are contained in the sample utterances list, though providing a more comprehensive set of samples to build the model from improves the chances of a match. Click the Next button to save this configuration.

The list of sample utterances that you defined allows the skill interface to resolve spoken words to intent events (see diagram below).

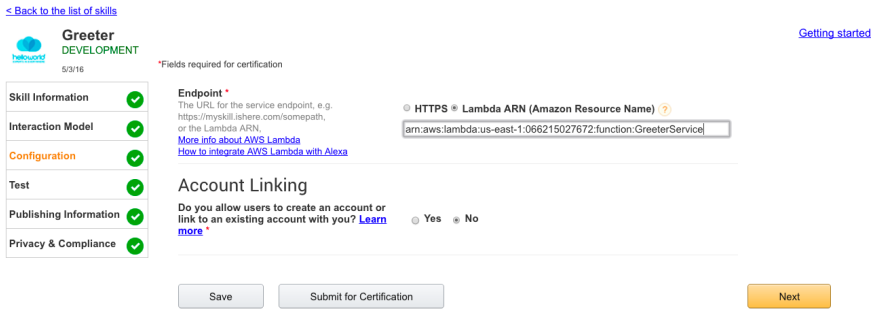
Figure 1.16 Sample Utterances Resolve Spoken Words to Intents



Specifying the ARN

Now you specify the endpoint for your skill service you uploaded by providing the ARN value you copied down earlier. Select Lambda ARN for Endpoint. Paste the ARN value into the Endpoint field. For the Account Linking option, select No and click Next. You will work with Account Linking at a later time.

Figure 1.17 Providing the Service ARN



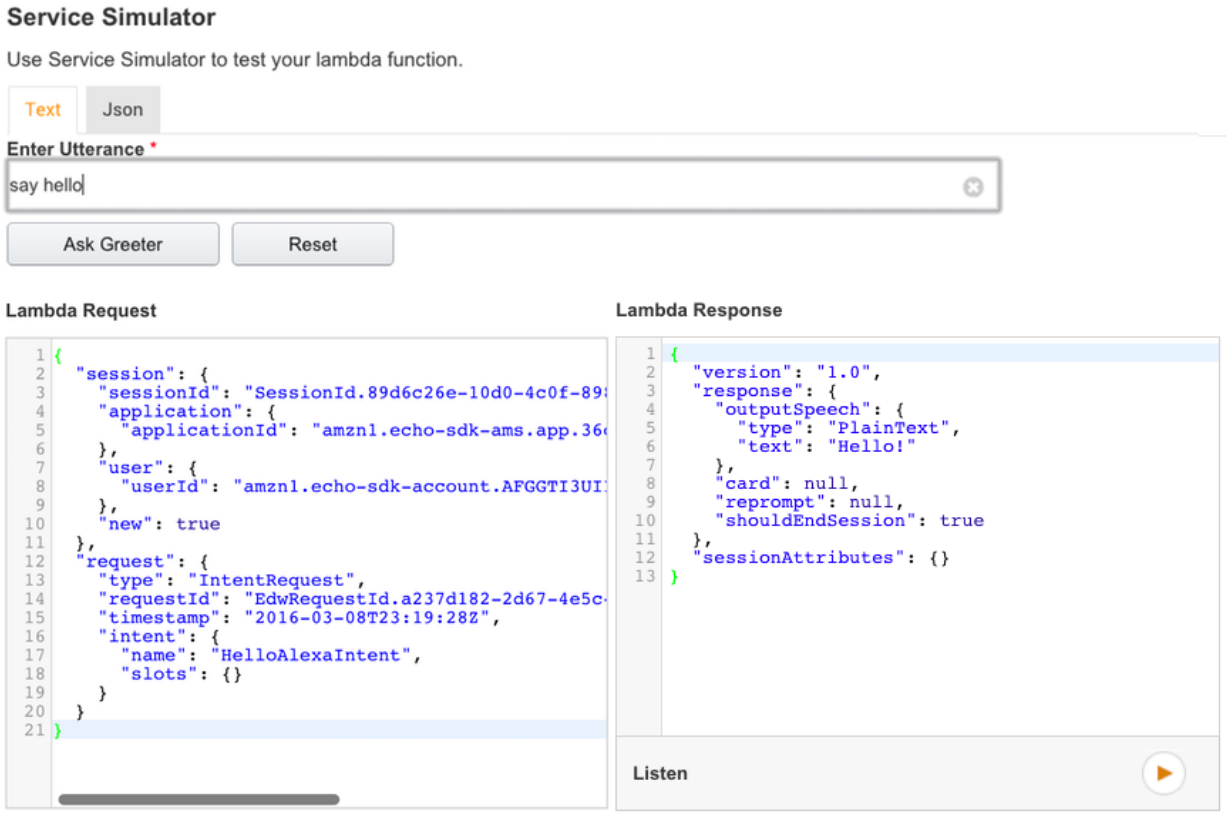
Testing the Interaction with the Service Simulator

Now that the skill service and skill interface are both configured, you may test an interaction with the skill. Make sure you are on the Test section within the Greeter skill interface configuration. You will see the Service Simulator on this page. Under the field Enter Utterance, enter "say hello" and click "ask Greeter". This would be equivalent to speaking to an Echo and saying "Alexa, ask Greeter to say hello". Verify that the Lambda response pane contains text similar to the following:

Listing 1.9

```
{
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "PlainText",
      "text": "Hello"
    },
    "card": null,
    "reprompt": null,
    "shouldEndSession": true
  },
  "sessionAttributes": {}
}
```

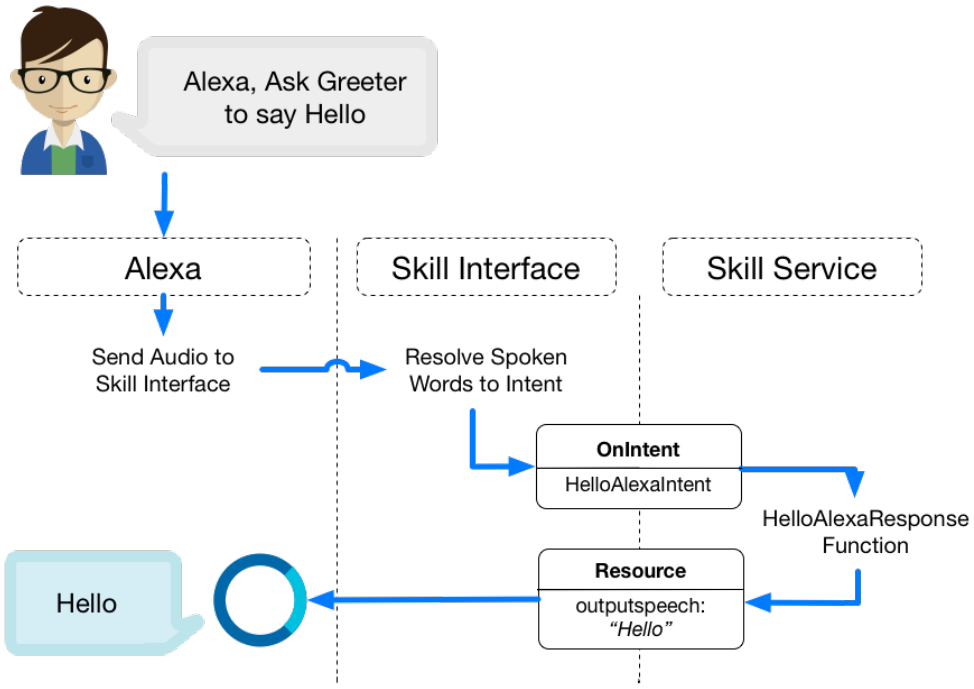
Figure 1.18 Service Simulator Results



Congratulations! You have successfully completed your first Alexa skill. If you have an Echo available, sign in with the developer account that you created this skill under to test it on an Alexa-enabled device.

Understanding the Greeter Skill

Figure 1.19 For the More Curious: A Skill's Typical Request Lifecycle



`Greeter` shows the typical situation with a skill interaction between the user, Echo device, skill interface, and skill service. Echo routes spoken words from the user to the skill interface, where a determination can be made about the intent from the sample utterances. At this point, a request containing a JSON payload describing what the skill interface resolved is sent to the skill service. The skill service's resulting JSON response is then returned to the skill interface where it is forwarded on to the Echo and finally played back.

The technology provided by the skill interface that resolves spoken words into intents is called the *Natural Language Processing* classifier. Natural Language Processing is an actively advancing field of research within the Machine Learning space. Simply stated, Natural Language Processing applies statistical techniques and the latest AI research to the problem of resolving a user's spoken words to intents the skill service can act upon. For a more in-depth summary of NLP, check out

```
https://en.wikipedia.org/wiki/Natural_language_processing
```

Silver Challenge: Bonjour, Alexa!

A skill can feature many intent handlers, each responding to different sample utterance mappings. For this extra challenge, register a new intent within the schema and provide sample utterances for handling a (slightly) French speaker's request to your skill: "Alexa, ask greeter to say Bonjour!". The new version of the skill should handle both old requests to say "hello" and the new request that returns "bonjour!" instead. Solving this challenge will require changes to both the skill service and the skill interface.

Gold Challenge: Good Morning, Good Afternoon, Good Night!

Saying Hello is good, but a skill that appropriately responds with "Good Morning", "Good Afternoon" and "Good Night" is even better! For this challenge, extend the HelloWorldIntent handler to check the current time on the

server. Use the following rules to determine the response: If the time is between 12 AM and 12 PM, the response should be "Good Morning!". If the time is between 12 PM and 5 PM, the response should be "Good Afternoon!". If the time is between 5 PM and 12 AM, the response should be "Good Night!". Do not worry about Timezone support in this exercise, the server's current time without accounting for a user's location will work for the challenge! As a starting place, JavaScript has the helpful `Date` class for doing things with time.

```
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date
```

Platinum Challenge: ES6 support

If you are already familiar with Node.js, you may also be familiar with the different features available in the most recent version of JavaScript, ES6. ES6 support was recently added to the AWS Lambda runtime, and is an option for further simplifying the required code to implement a skill in Node.js. Rewrite the skill service to make use of the new lambda shorthand syntax and const keyword where appropriate. For example, an ES6 rewrite of the `helloResponseFunction` would look like this:

Listing 1.10 Rewriting helloResponseFunction with ES6 Lambdas

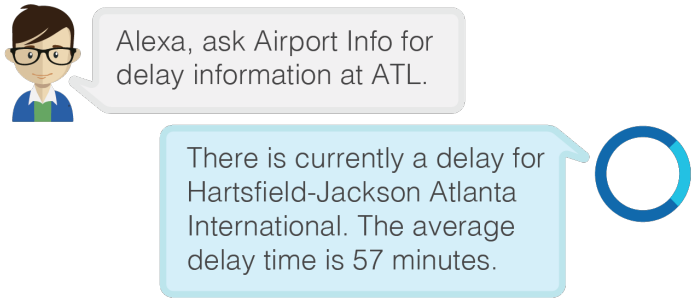
```
~~var helloResponseFunction = function(intent, session, response) {~~
~~  response.tell(SPEECH_OUTPUT);~~
~~};~~
var helloResponseFunction = (intent, session, response) => { response.tell(SPEECH_OUTPUT); };
```

2

Slots, Slot Types and Utterances

In this chapter, you will build a skill called *Airport Info*, which allows you to find out whether there are currently delays at a specified airport. The skill will talk to the Federal Aviation Administration's Airport Information service in order to retrieve the current status of an airport in response to a user's requested airport code. Here is how an interaction with the Airport Info skill will work:

Figure 2.1



In this chapter you will expand your knowledge of the interaction model that you worked with in the previous exercise. You will use a feature of the interaction model's utterances definition called *slots*. Slots are an aspect of the interaction model you have not seen yet. They allow you to pass values into an intent handler as a variable a user speaks with their voice. You will also see how to interact with a JSON-backed webservice.

Setting up a Local Development Environment

Before you begin building the Airport Info skill you will first set up a local development environment. The local development environment will allow you to author a skill without the need for uploading files to AWS Lambda each time a change is made, enable local debugging, and ease the process of diagnosing any bugs that may occur by allowing you to easily inspect stack traces and logs.

Your service is authored in JavaScript and runs on Node.js. The first step in getting a local development environment working is installing Node.js so that you can run JavaScript on the Node.js platform locally. Use Node Version Manager to install Node.js so that you can easily switch versions of Node.js as new versions become available.

Listing 2.1 installing nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.30.2/install.sh | bash
```

Close and reopen the terminal to ensure it is reloaded and check that nvm installed by running:

Listing 2.2 checking nvm is installed correctly

```
nvm ls
```

This will list the versions of Node.js that have been installed:

Listing 2.3 checking nvm is installed correctly

```
$ nvm ls
       v4.3.2
->     system
```

It is not important if your output from the nvm command looks different. What you are looking for is that the command completed successfully.

Next, install the version of Node.js currently supported on AWS Lambda. This is to ensure that the behavior of your skill locally will closely match the behavior of your skill when it is later deployed to the AWS Lambda environment.

At the time of this writing, the version of Node.js available on AWS Lambda is *v4.3.2*, though this may have changed, since AWS is under active development. You should first check the following url:

```
http://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html
```

to find out if the version of Node.js AWS Lambda supports is now a more recent version. Note the most recent version number of Node.js that is available on AWS Lambda and substitute *v4.3.2* for that version if applicable. The text will use *v4.3.2*, since it is currently the most recent version of Node.js available on AWS Lambda.

Install Node.js using the following command:

Listing 2.4 Installing node v4.3.2

```
nvm install v4.3.2
```

After this command completes, set *v4.3.2* as the default version of Node.js the system will use for executing JavaScript using the following command:

Listing 2.5 Setting v4.3.2 to default

```
nvm alias default v4.3.2
```

Adding alexa-app-server

Now that Node.js is set up you will check out an open source project that enables running a skill service locally called *alexa-app-server*. Run the following command:

Listing 2.6 Checking out alexa-app-server

```
git clone https://github.com/matt-kruse/alexa-app-server.git
```

Change to the `alexa-app-server` directory and install the dependencies with the following command:

Listing 2.7 Installing Node Package Manager

```
npm install
```

Change to the `examples/apps` directory within the `alexa-app-server` directory. Create a new folder called `airportinfo`. You will be building your skill and its related components within this directory. In the *Testing the Skill Service* section, you'll work more closely with *alexa-app-server* itself when you begin fleshing out the skill service portion of the skill.

Managing Dependencies with Node Package Manager

Your project will require library dependencies to support making the network request. Use the Node Package Manager -- npm -- to add these dependences. npm was installed with Node.js. **`npm`** generates a `package.json` file to manage a list of dependencies your skill service will require. Run the following command within the `airportinfo` directory to create the `package.json` file.

Listing 2.8 Generating the package.json file

```
npm init
```

You will be guided through a series of prompts that will construct a `package.json` file for the project:

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (foobar) airportinfo
version: (1.0.0)
description: an alexa skill for tracking airport information
entry point: (index.js)
test command:
git repository:
keywords:
author: josh skeen
license: (ISC)

About to write to /Users/joshskeen/code/airportinfo/package.json:
{
 "name": "airportinfo",
 "version": "1.0.0",
 "description": "an alexa skill for tracking airport information",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "josh skeen",
 "license": "ISC"
}
Is this ok? (yes) yes
```

Now that the `package.json` has been generated, you will add the dependencies the skill will require.

Listing 2.9 Installing dependencies

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

The **`--save`** flag tells **`npm`** to add the dependencies to the `package.json` file to keep track of them for installation in different environments.

Here is a brief description of each library you installed. You will learn more about each library as you work with them throughout the chapter.

alexa-app - simplified skill service development with support for running within alexa-app-server

request-promise - simplifies the task of requesting data from a REST endpoint. Handles asynchronous requests as a promise

lodash - a convenience library with many useful utility functions for JavaScript development

Building FAADataHelper

You will begin by creating a helper module for making requests to the FAA Airport Information service,

```
http://services.faa.gov/docs/services/airport/
```

Begin by adding the following to a new file within `airportinfo` called `faa_data_helper.js`.

Listing 2.10 Creating airportinfo/faa_data_helper.js

```
'use strict';
var _ = require('lodash');
var requestPromise = require('request-promise');
var ENDPOINT = 'http://services.faa.gov/airport/status/';

function FAADataHelper() {
}

module.exports = FAADataHelper;
```

You required the libraries needed to build the **`FAADataHelper`** module with. You also defined the service endpoint you will make GET requests to. Last, you defined **`FAADataHelper`**, which you will build upon as you flesh out functionality.

Requesting Airport Info

Next, define a method for requesting the status of an airport using an airport code. For example, requesting

```
http://services.faa.gov/airport/status/CLT?format=application/json
```

should return airport information for Charlotte Douglas International Airport similar to the following:

Listing 2.11 Airport information Service Response

```
{
  'delay': 'true',
  'IATA': 'CLT',
  'state': 'North Carolina',
  'name': 'Charlotte Douglas International',
  'weather': {
    'visibility': 8.00,
    'Weather': 'Light Rain',
    'meta': {
      'credit': 'NOAA\'s National Weather Service',
      'updated': '12:52 PM Local',
      'url': 'http://weather.gov/'
    },
    'temp': '63.0 F (17.2 C)',
    'wind': 'Southwest at 18.4mph'
  },
  'ICAO': 'KCLT',
  'city': 'Charlotte',
  'status': {
    'reason': 'WX:Low Ceilings',
    'closureBegin': '',
    'endTime': '',
    'minDelay': '46 minutes',
    'avgDelay': '',
    'maxDelay': '1 hour',
    'closureEnd': '',
    'trend': 'Increasing',
    'type': 'Arrival'
  }
};
```

Define the **`getAirportStatus(airportCode)`** as follows:

Listing 2.12 Defining getAirportStatus(airportCode)

```
...
function FAADataHelper() {
}

FAADataHelper.prototype.getAirportStatus = function(airportCode) {
  var options = {
    method: 'GET',
    uri: ENDPOINT + airportCode,
    json: true
  };
  return requestPromise(options);
};
module.exports = FAADataHelper;
```

This code instructs the *request-promise* library that you imported to make a GET request to the FAA Endpoint and return JSON data as a *promise*. For further reading on the concept of promises, check out:

```
https://www.promisejs.org/
```

A promise allows the asynchronous nature of a network request - making a request and waiting on the results to come back and then performing work - to be easily handled by abstracting the need for having complex callback logic or function nesting, which can become tough to reason about. Instead, the promise object returned by *request-promise* will provide a **`.then()`** method where you will define what happens when the request completes.

Testing the Request

Test that the **`FAADataHelper`** object works as expected from Node.js. First, open a terminal and change to the `airportinfo` directory. Next, type **`node`** on the commandline. This will start the interactive Node.js shell REPL.

Listing 2.13 Testing the getAirportStatus method

```
$ node
> var FAADataHelper = require('./faa_data_helper.js'); var faaHelper = new FAADataHelper();
undefined
> faaHelper.getAirportStatus('CLT').then(console.log);
Promise {
  _bitField: 0,
  _fulfillmentHandler0: undefined,
  _rejectionHandler0: undefined,
  _progressHandler0: undefined,
  _promise0: undefined,
  _receiver0: undefined,
  _settledValue: undefined }
> { delay: 'true',
  IATA: 'CLT',
  state: 'North Carolina',
  name: 'Charlotte Douglas International',
  weather:
   { visibility: 10,
     weather: 'Overcast',
     meta:
      { credit: 'NOAA\'s National Weather Service',
        updated: '1:52 PM Local',
        url: 'http://weather.gov/' },
     temp: '70.0 F (21.1 C)',
     wind: 'South at 18.4mph' },
  ICAO: 'KCLT',
  city: 'Charlotte',
  status:
   { reason: 'WX:Low Ceilings',
     closureBegin: '',
     endTime: '',
     minDelay: '15 minutes',
     avgDelay: '',
     maxDelay: '29 minutes',
     closureEnd: '',
     trend: 'Decreasing',
     type: 'Arrival' }
}
```

You should see JSON data from the FAA webservice logged out from the **`console.log`** method you passed into the **`then()`** method. Because *request-promise* returns a promise object, the **`then()`** method was available.

Formatting the Status Response

Next, you will add a method to **`FAADataHelper`** that formats the response from the webservice to be spoken by Alexa called **`formatAirportStatus(airportStatusObject)`**. This method will accept the `AirportStatusObject` you retrieved from the FAA, and return a string the service will send to Alexa to speak back. The `AirportStatusObject` contains delay information, average delay time, airport name, and more. You will pull the information from the `AirportStatusObject` and build a formatted message for Alexa to speak from it.

Listing 2.14 Adding the formatAirportStatus method

```
FAADataHelper.prototype.getAirportStatus = function(airportCode) {
  var options = {
    method: 'GET',
    uri: ENDPOINT + airportCode,
    json: true
  };
  return requestPromise(options);
};
FAADataHelper.prototype.formatAirportStatus = function(aiportStatusObject) {
  if (aiportStatusObject.delay === 'true') {
    var template = _.template('There is currently a delay for ${airport}. ' +
      'The average delay time is ${delay_time}.');
    return template({
      airport: aiportStatusObject.name,
      delay_time: aiportStatusObject.status.avgDelay
    });
  } else {
    //no delay
    var template = _.template('There is currently no delay at ${airport}.');
    return template({ airport: aiportStatusObject.name });
  }
};

module.exports = FAADataHelper;
```

Defining the Skill Service

Now that you have implemented a way to receive the data your skill will share with the user, you can begin defining the skill service. First, create a file called `index.js` in the `airportinfo` directory. The `index.js` file serves as an entry point into the service when requests are made from the skill interface to the skill service. Add the following to `index.js`:

Listing 2.15 Creating airportinfo/index.js

```
'use strict';
module.change_code = 1;
var Alexa = require('alexa-app');
var skill = new Alexa.app('airportinfo');
var FAADataHelper = require('./faa_data_helper');
module.exports = skill;
```

Here, you imported the **`FAADataHelper`** object you wrote and the **`alexa-app`** module that you installed earlier.

Notice the *alexa-app* library that was required - this library provides several conveniences you will leverage to implement the service. It also makes the skill service runnable locally, which you will see soon. The `module.change_code = 1;` declaration enables live reloading of the skill service when running it locally as changes are made.

Listing 2.16 Responding to the onLaunch Event

Next, you will define an **`onLaunch`** event handler that will allow the service to respond correctly to onLaunch events from the skill interface. The **`onLaunch`** event will be triggered if a user uses the phrase "Alexa, open Airport Info", "Alexa, launch AirportInfo", or "Alexa, start AirportInfo".

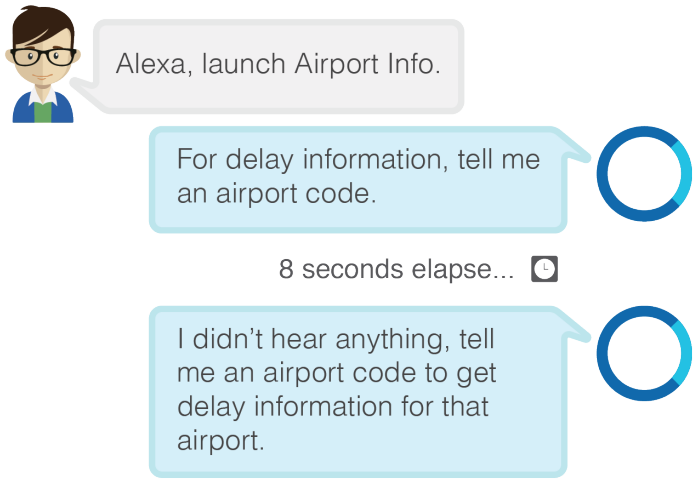
If you have followed the previous exercise for setting up the Greeter skill, this may seem a bit different. This is because of the helper functions provided by the `alexa-app` library you are now using, which has it's own design patterns that slightly differ from Amazon's implementation of the `AlexaSkill.js` class.

```
'use strict';
module.change_code = 1;
var _ = require('lodash');
var Alexa = require('alexa-app');
var skill = new Alexa.app('airportinfo');
var FAADataHelper = require('./faa_data_helper');
var reprompt = 'I didn\'t hear an airport code, tell me an Airport code to get delay '
    + 'information for that airport.';
skill.launch(function(request, response) {
  var prompt = 'For delay information, tell me an Airport code.';
  response.say(prompt).reprompt(reprompt).shouldEndSession(false);
});
module.exports = skill;
```

Here you defined the **`onLaunch`** event handler by specifying a function the **`launch()`** function expects as an argument. In this case, if a user launches our skill from the Echo, the launch event will be sent to the skill service and the launch definition will be used to determine how Alexa will behave. As a result of this definition, Alexa will say "For delay information, tell me an airport code" upon launch.

Also, notice you have specified a *reprompt* as part of the response. A reprompt defines what happens when nothing is heard in response to the last statement. The interaction would look like this:

Figure 2.2 An interaction with reprompts



Defining an Intent Handler

Next, you will define the intent handler for a user's requested airport status. As you recall, an intent handler responds to an intent request. An intent request is an indication of a particular task our user would like to perform.

Listing 2.17 Adding an airportInfoIntent handler

```
...
skill.launch(function(request, response) {
  var prompt = 'For delay information, tell me an Airport code.';
  response.say(prompt).reprompt(prompt).shouldEndSession(false);
});
skill.intent('airportInfoIntent', {
    'slots': {
      'AIRPORTCODE': 'FAACODES'
    },
    'utterances': ['{|flight|airport} {|delay|status} {|info} {|for} {-|AIRPORTCODE}']
  },
  function(request, response) {
        return false;
  }
);

module.exports = skill;
```

First, you declared that the skill can handle an intent called `airportInfoIntent`. This will be triggered on the server by a request from the skill interface every time the utterance that corresponds to it is matched by a phrase the user has spoken.

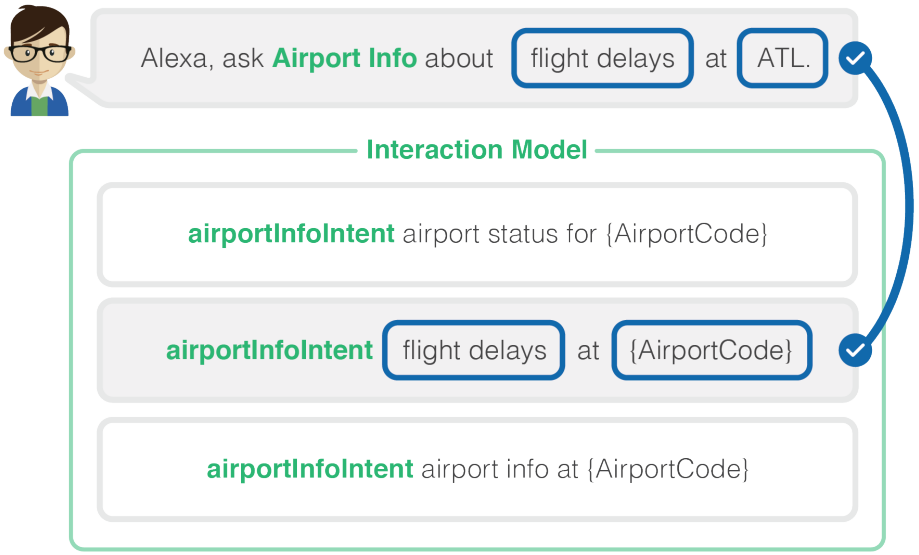
Next, you provided a *slot type* definition for the intent in the 'slots' definition list. A slot in an Alexa skill is a container for user-provided input our program requires. Think of a slot as defining a dictionary that maps a key onto a value. Here, you made a slot that defined an `AIRPORTCODE` key that will have values of type `FAACODES`.

Note that the definition here tells alexa-app that you would like to generate a new slot and slot type definition, but as you will see you will also need to publish this declaration to the skill interface. You will soon provide the values `AIRPORTCODE` can possibly take as well as define the slot on the skill interface, but alexa-app makes this work easier for you.

Defining Utterances

An utterance is a phrase a user could say when interacting with your skill, for example "ask airport info about flight delays at ATL". You declared the sample utterances in your code that will be used by the skill interface to match the intent to the user's spoken words.

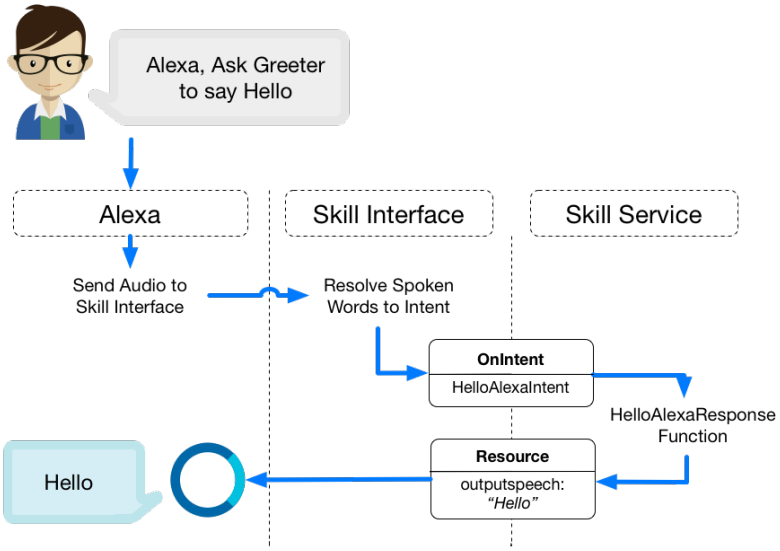
Figure 2.3 Sample Utterances Resolve Spoken Words to Intents



One clarification about how the skill interface resolves spoken utterances to an intent is that it is not a "one to one" matching operation. The list of sample utterances does not necessarily need to contain all of the possible variations of how the user may phrase his request. Behind the scenes, the list of sample utterances is used to "train" a Natural Language Understanding model that resolves spoken utterances to the Intent and slots.

Once this resolution occurs, the skill interface sends the intent and additional info for the slots to the skill service. The service can then determine what alexa should say, and returns it to the skill interface. Finally, the skill interface forwards this on to the Echo device. To recap, remember the diagram from the following chapter:

Figure 2.4 Recap: Lifecycle of an Alexa Skill Request



The *alexa-app* module allows generating a list of utterances by providing a shorthand notation for creating a list of the expanded form. In the following *alexa-app* expression, the pipe character indicates a word or group of words are optional or can be substituted for an alternate within the same set of curly braces: `'{|flight|airport} {|delays|status} {|info} {|for} {-|AIRPORTCODE}'` This expression will be evaluated as:

```
airportInfoIntent  {AIRPORTCODE}
airportInfoIntent flight {AIRPORTCODE}
airportInfoIntent airport {AIRPORTCODE}
airportInfoIntent  delays {AIRPORTCODE}
airportInfoIntent flight delays {AIRPORTCODE}
airportInfoIntent airport delays {AIRPORTCODE}
airportInfoIntent  status {AIRPORTCODE}
airportInfoIntent flight status {AIRPORTCODE}
airportInfoIntent airport status {AIRPORTCODE}
airportInfoIntent  info {AIRPORTCODE}
airportInfoIntent flight info {AIRPORTCODE}
airportInfoIntent airport info {AIRPORTCODE}
airportInfoIntent  delays info {AIRPORTCODE}
airportInfoIntent flight delays info {AIRPORTCODE}
airportInfoIntent airport delays info {AIRPORTCODE}
airportInfoIntent  status info {AIRPORTCODE}
airportInfoIntent flight status info {AIRPORTCODE}
airportInfoIntent airport status info {AIRPORTCODE}
airportInfoIntent  for {AIRPORTCODE}
airportInfoIntent flight for {AIRPORTCODE}
airportInfoIntent airport for {AIRPORTCODE}
```

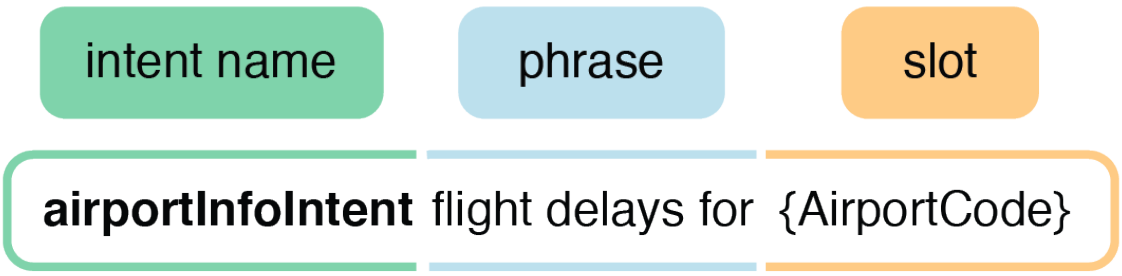
```
airportInfoIntent  delays for {AIRPORTCODE}
airportInfoIntent flight delays for {AIRPORTCODE}
airportInfoIntent airport delays for {AIRPORTCODE}
airportInfoIntent  status for {AIRPORTCODE}
airportInfoIntent flight status for {AIRPORTCODE}
airportInfoIntent airport status for {AIRPORTCODE}
airportInfoIntent  info for {AIRPORTCODE}
airportInfoIntent flight info for {AIRPORTCODE}
airportInfoIntent airport info for {AIRPORTCODE}
airportInfoIntent  delays info for {AIRPORTCODE}
airportInfoIntent flight delays info for {AIRPORTCODE}
airportInfoIntent airport delays info for {AIRPORTCODE}
airportInfoIntent  status info for {AIRPORTCODE}
airportInfoIntent flight status info for {AIRPORTCODE}
airportInfoIntent airport status info for {AIRPORTCODE}
```

Accepted user utterances with this utterance definition would be "Alexa, ask airport info for airport status info for ATL", "Alexa, ask airport info about delays for ATL", "Alexa, ask airport info about flight delays for ATL" for example.

alexa-app will automatically generate all of the permutations for this expression and is a convenient way to manage the list of utterances. You will continue to use the *alexa-app* module throughout the remainder of the course. A note about the `alexa-app` syntax for defining sample utterances: to specify a custom slot type to be emitted in the sample utterances output, the syntax '{-|SlotName}' is used, where 'SlotName' is the slot you would like to use in the sample utterance. The final output to be loaded into the interaction model will equal 'airportInfoIntent airport status info for {SlotName}'.

Notice the "{AIRPORTCODE}" portion of each emitted sample utterance. This defines where the slot you defined earlier should be expected in the user utterance. When the skill interface resolves the user's words to a particular intent, if a slot is defined it will also map the words spoken within the `AIRPORTCODE` slot. The following diagram shows the parts of the sample utterance you have specified:

Figure 2.5 Parts of an Example Utterance



Using the Slot Value

Now that you have defined a slot and determined how a user's phrase maps to an intent, you will put the intent and slot to use. The slot will be available on the request object that is passed from the skill interface to the Node.js service. It will be accessed by the name you defined it by in the 'slots' portion of your intent handler definition. In this case, the FAA Code the user would like to query for is available via '`AIRPORTCODE`'. The next step is to retrieve the `AIRPORTCODE` slot value and pass it in to **`FAADataHelper`**'s **`getAirportStatus(airportcode)`** method. Change the `index.js` file in the following way:

Listing 2.18 Retrieving the Slot Value

```
...
var reprompt = 'I didn\'t hear an airport code, tell me an airport code to get delay'
    + 'information for that airport.';

skill.launch(function(request, response) {
  var prompt = 'For delay information, tell me an airport code.';
  response.say(prompt).reprompt(reprompt).shouldEndSession(false);
});
skill.intent('airportInfoIntent', {
    'slots': {
      'AIRPORTCODE': 'FAACODES'
    },
    'utterances': ['{|flight|airport} {|delay|status} {|info} {|for} {-|AIRPORTCODE}']
  },
  function(request, response) {
      var airportCode = request.slot('AIRPORTCODE');
      var faaDataHelper = new FAADataHelper();
      faaDataHelper.getAirportStatus(airportCode).then(function(airportStatus) {
        console.log(airportStatus);
        response.say(faaDataHelper.formatAirportStatus(airportStatus)).send();
      });
      return false;
  }
);
module.exports = skill;
```

The line `var airportCode = request.slot('AIRPORTCODE');` pulls the value the user spoke into the intent so that it can be used to retrieve the appropriate airport information from the FAA.

Keep in mind that the data the skill service is acting upon is simply a specially formatted JSON payload from the skill interface. The data that was received by your skill service resembles the following JSON structure:

Listing 2.19 It's Just JSON

```
"request": {
    "type": "IntentRequest",
    "requestId": "EdwRequestId.xxxx",
    "intent": {
      "name": "airportInfoIntent",
      "slots": {
        "AIRPORTCODE": {
          "name": "AIRPORTCODE",
          "value": "ATL"
        }
      }
    },
    "locale": "en-US"
  }
```

As you can see, the slot information and intent information is sent as a JSON formatted payload from the skill interface once it is resolved. The `alexa-app` library makes it more convenient to work with this information in a systematic way.

Handling Edge Cases

Your service will now work in most cases, but there are two additional cases you need to address. The first is if the FAA Airport Service is not working correctly or doesn't understand the code a user provided. In this case the server will return a `404` error. Fortunately, the promise object returned from **`FAADataHelper`** has a built in mechanism for dealing with such a failure called **`catch(function(error){})`** that will be called when these cases occur.

Add the following **`catch(function(error){})`** definition to the method above:

Listing 2.20 Handling a Failed Response

```
...
faaDataHelper.getAirportStatus(airportCode).then(function(airportStatus) {
      console.log(airportStatus);
      response.say(faaDataHelper.formatAirportStatus(airportStatus)).send();
}).catch(function(err) {
    console.log(err.statusCode);
    var prompt = 'I didn\'t have data for an airport code of ' + airportCode;
    response.say(prompt).reprompt(reprompt).shouldEndSession(false).send();
});
return false;
...
```

The next case you will address is when the airport code was not heard or submitted by the Alexa-enabled device. This can happen if the user misspoke or a network interruption occured. Add the following to the skill server code:

Listing 2.21 Gracefully recovering from Bad Input

```
'use strict';
module.change_code = 1;
var Alexa = require('alexa-app');
var skill = new Alexa.app('airportinfo');
var FAADataHelper = require('./faa_data_helper');
var _ = require('lodash');
...
function(request, response) {
  var airportCode = request.slot('AIRPORTCODE');
  if (_.isEmpty(airportCode)) {
    var prompt = 'I didn\'t hear an airport code. Tell me an airport code.';
    response.say(prompt).reprompt(reprompt).shouldEndSession(false);
    return true;
  } else {
      var faaDataHelper = new FAADataHelper();
      faaDataHelper.getAirportStatus(airportCode).then(function(airportStatus) {
            console.log(airportStatus);
            response.say(faaDataHelper.formatAirportStatus(airportStatus)).send();
      }).catch(function(err) {
          console.log(err.statusCode);
          var prompt = 'I didn\'t have data for an airport code of ' + airportCode;
          response.say(prompt).reprompt(reprompt).shouldEndSession(false).send();
      });
      return false;
  }
...
```

Here you handle the second case - an empty value for the airport code. The *lodash* module you imported provides a number of JavaScript utilities, including a method called **`_.isEmpty`** , which you use to determine if the AIRPORTCODE slot had a value or was empty. If it was empty, you reprompt to instruct the user to try again.

Notice the true/false values being returned. These indicate to the alexa-app framework whether the operation being performed is synchronous or asynchronous. Whenever an asynchronous request is made, false must be returned. When the request is synchronous, true should be returned.

Testing the Skill Service

Now that you have implemented the skill service, you will test it by starting the **`alexa-app-server`**, which will run the skill service you have built locally. Change to the `examples/` directory where you should see a `server.js` file, and run the command:

Listing 2.22 Starting alexa-app-server

```
node server
```

You should see output similar to the following:

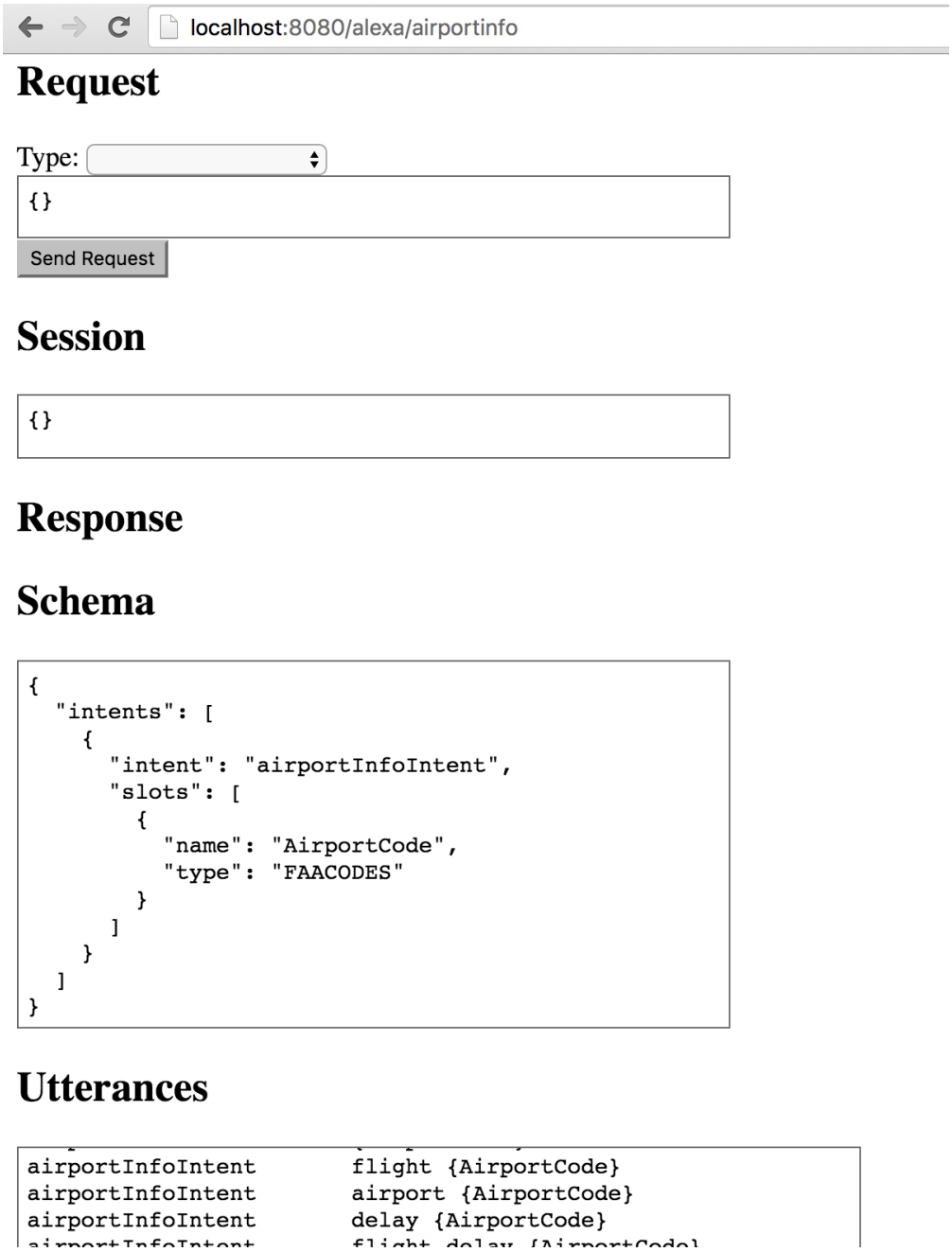
```
$ node server
Serving static content from: public_html
Loading server-side modules from: server
   Loaded /Users/joshskeen/code/alexa-app-server/examples/server/login.js
Loading apps from: apps
   Loaded app [airportinfo] at endpoint: /alexa/airportinfo
Listening on HTTP port 8080
```

Now that you have started the server, visit the following url in your web browser:

```
http://localhost:8080/alexa/airportinfo
```

You should see a web page that includes the intent schema, slot type, and utterances you defined earlier.

Figure 2.6 Alexa App Server Web Interface



First, test that the launch intent handler behaves as expected. In the Type dropdown, select Launch Request and click Send Request. Check that the "Response" window contains text similar to the following:

```
{
  "version": "1.0",
  "sessionAttributes": {},
  "response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>For delay information, tell me an airport code.</speak>"
    },
    "reprompt": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I didn't hear an airport code,
        tell me an Airport code to get delay information for that airport.</speak>"
      }
    }
  }
}
```

Intent Requests

Next, you will test the behavior of the skill service when it is sent an intent request from the skill interface.

The first case you will address is when no `AIRPORTCODE` is submitted to the intent handler. Select IntentRequest from the Type dropdown and airportInfoIntent from the Intent dropdown. leave the AIRPORTCODE field under Slot Values empty, and press Send Request.

Figure 2.7 Testing an Empty AIRPORTCODE Slot

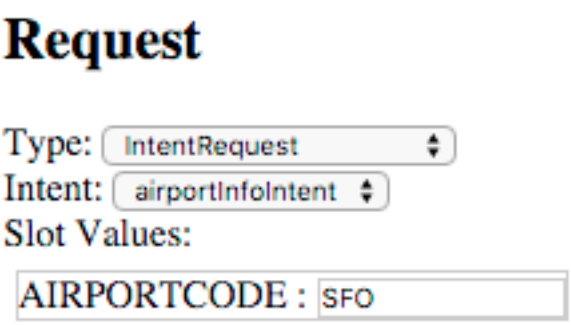
In the response section of the web page, you should see the following output:

```
{
  "version": "1.0",
  "sessionAttributes": {},
  "response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>I didn't hear an airport code. Tell me an airport code.</speak>"
    },
    "reprompt": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I didn't hear an airport code, tell me an Airport code to get delay information for t
      }
    }
  }
```

```
}
```

Test that an `airportInfoIntent` with a valid `AIRPORTCODE` responds with the airport status information. Enter 'SFO' for the AIRPORTCODE field.

Figure 2.8 Testing a valid AIRPORTCODE

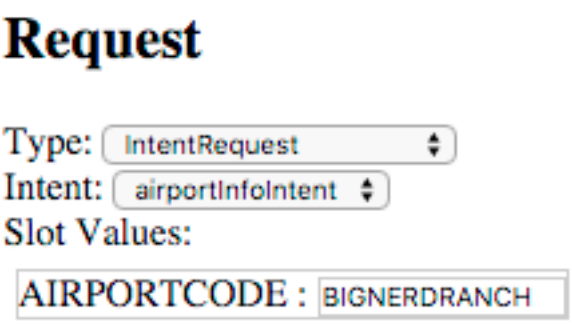


You should get a response from the skill service indicating the airport status for San Francisco International Airport:

```
{
  "version": "1.0",
  "sessionAttributes": {},
  "response": {
    "shouldEndSession": true,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>There is currently no delay at San Francisco International.</speak>"
    }
  }
}
```

Test an `airportInfoIntent` with an invalid `AIRPORTCODE` responds with the airport status information. Enter 'BIGNERDRANCH' for the `AIRPORTCODE` field.

Figure 2.9 Testing an invalid AIRPORTCODE



You should get a response from the skill service indicating there was no data for an airport code of BIGNERDRANCH:

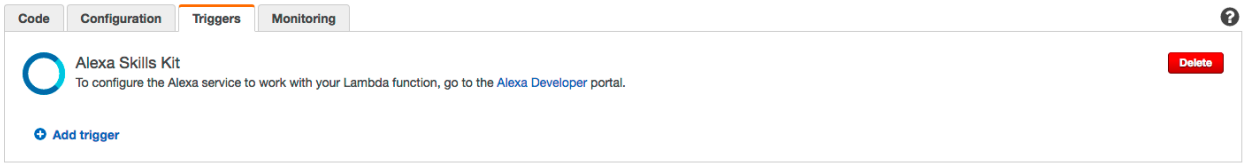
```
{
  "version": "1.0",
  "sessionAttributes": {},
  "response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
```

```
      "ssml": "<speak>I didn't have data for an airport code of BIGNERDRANCH</speak>"
    },
    "reprompt": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I didn't hear an airport code,
        tell me an Airport code to get delay information for that airport.</speak>"
      }
    }
  }
```

Deploying the Skill

Now that you have tested the skill service locally, it is time to deploy it and set up a skill configuration so that it can be used on a real Echo. Compress all files in the `airportinfo` directory including the `node_modules` directory and create a new AWS Lambda instance using the same steps from the previous exercise. Ensure that you enable Alexa Skills Kit in the Trigger section of the AWS Lambda page.

Figure 2.10 Enabling the Trigger on AWS Lambda

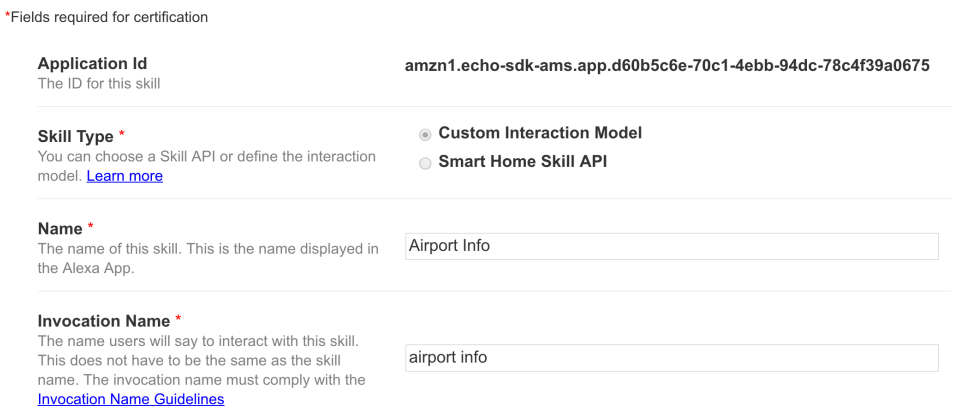


Copy the ARN from the Lambda instance, and create a new skill configuration in the Skill Configuration console at

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

For the invocation name, enter "airport info".

Figure 2.11 registering the skill information

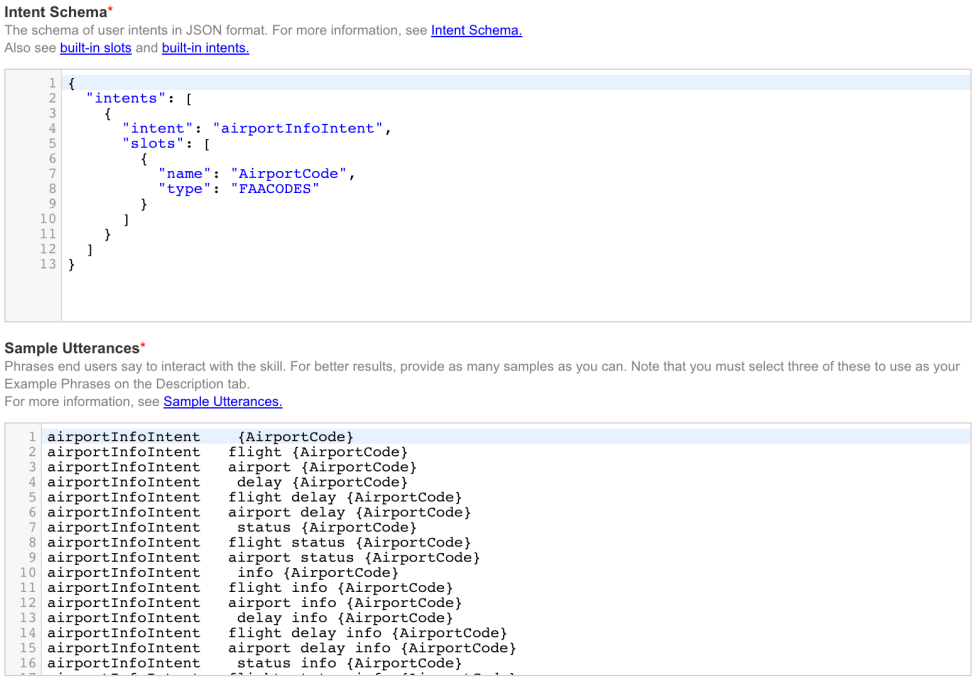


Advance to the Interaction Model page. You will now copy the Intent Schema and Sample Utterances information from your local skill server to the relevant fields. Refer to

```
http://localhost:8080/alexa/airportinfo
```

and copy the Intent Schema field into the Amazon Skill Configuration Console's Intent Schema field. Copy the Utterances field into the Amazon Skill Configuration Console's Utterances field.

Figure 2.12 Setting the Intent Schema and Utterances

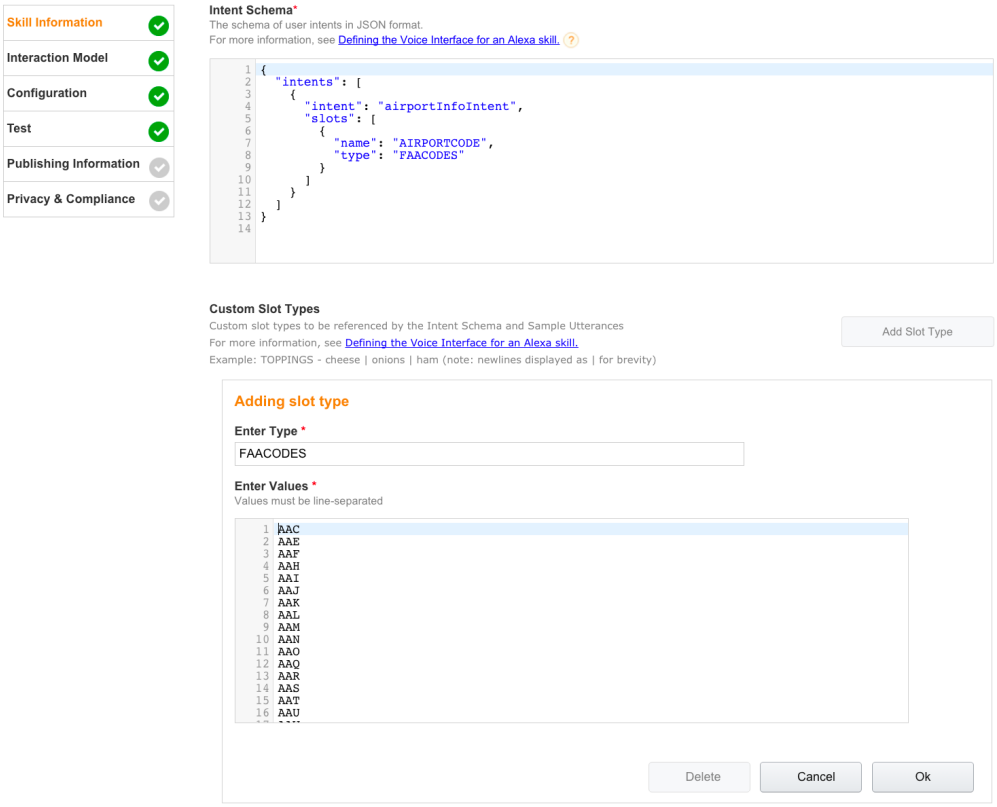


Your skill makes use of a custom slot type called `FAACODES`, which must be registered in the skill interface to work correctly. You must provide a set of example data the Slot Type will use to improve the chances of matching a phrase to the set of data. The example data you will use is a list list of FAA Airport Codes. Copy the list here at

```
https://raw.githubusercontent.com/bignerdranch/alexa-airportinfo/master/resources/FAACODES.txt
```

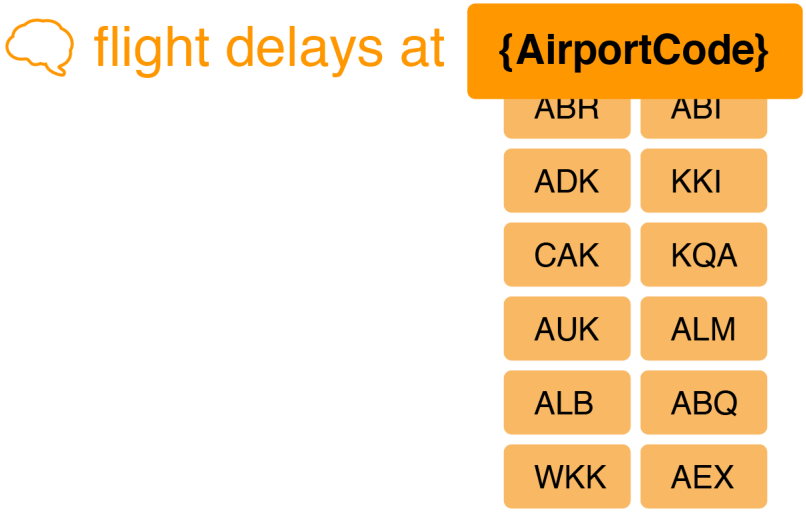
Under the Custom Slot Types area, click Add Slot Type. Enter `FAACODES` for Enter Type, and for Enter Values, paste the list of `FAACODES` you downloaded.

Figure 2.13 Adding the Custom Slot Type



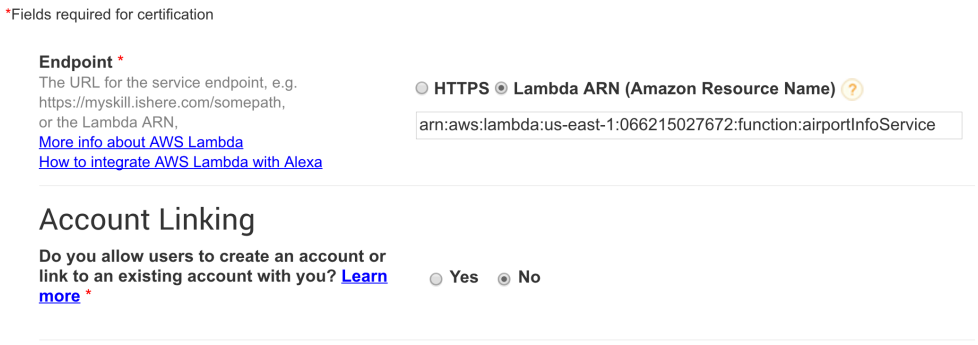
The custom slot type defines the list of acceptable three letter `FAACODES` the skill interface can resolve a user's spoken words to. You added the `AIRPORTCODE` slot type to the intent schema and used the slot in the sample utterances, so the values from the custom slot type could be used to resolve the user's utterance to an `AIRPORTCODE`.

Figure 2.14 Making Use of the FAACODES Custom Slot Type



Last, advance to the configuration page. For the Endpoint field, enter the ARN value that you copied during the AWS Lambda setup step.

Figure 2.15 Providing the Amazon Resource Name

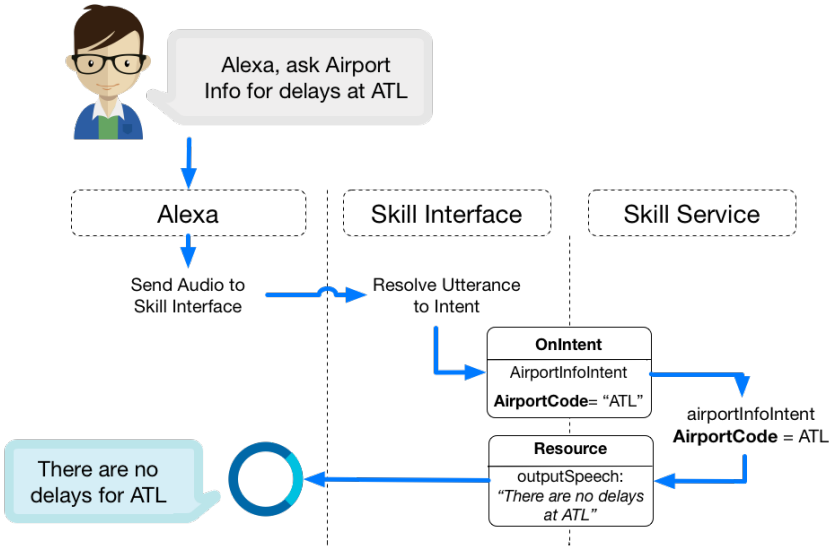


Congratulations, you have successfully developed and deployed the Airport Info Skill to the Developer Portal and AWS Lambda! You may now test the skill in the Developer Portal Test page or on an Alexa-enabled device.

Understanding Airport Info

Examine the following interaction diagram for the Airport Info skill.

Figure 2.16 Airport Info Diagram



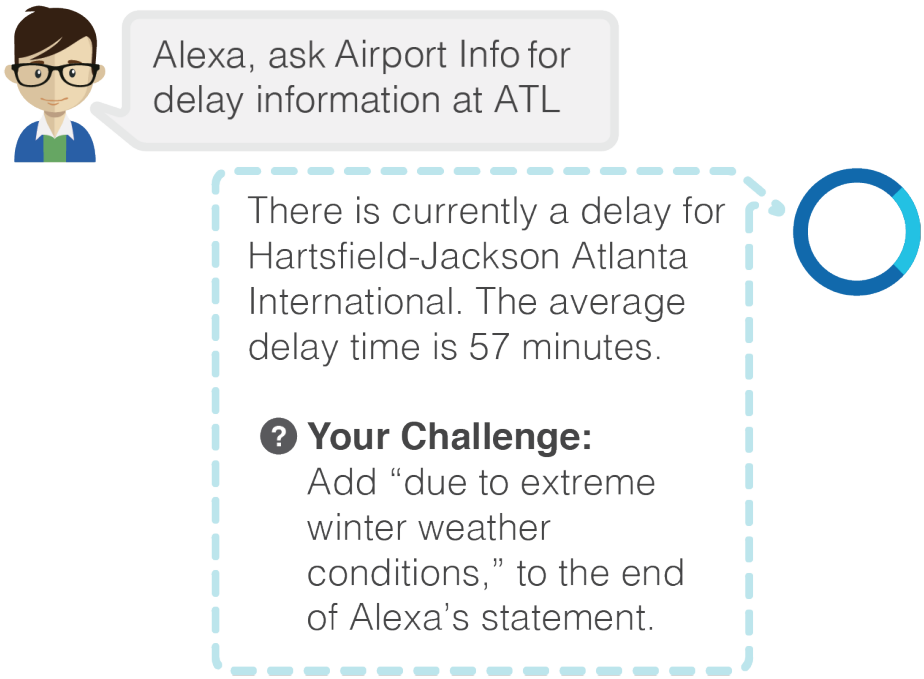
This diagram will appear similar to the Greeter skill, with a key difference. On the skill interface, the variable `AIRPORTCODE` was defined and sent to the skill service as part of the intent resolution the skill interface provided. Because you defined the custom slot type `FAACODES` you were able to update the skill interface's interaction model with an intent schema that defined a new slot. The slot definition you created on the intent schema mapped `FAACODES` to `AIRPORTCODE`, which you then made use of in your sample utterances. The skill interface resolved the user's words to an airport code because of the custom slot type definition, intent schema, and sample utterances in the interaction model. As you can see in the diagram, the skill interface passes the resolved `AIRPORTCODE` to the skill

service as part of the JSON payload for the intent, so that the skill service can make use of it in it's business logic. This allows a user to speak a variable with their voice so that your program can make use of it.

Silver Challenge: Adding a Delay Reason

Users would like to know why the delay at the airport they specified occured. Add the delay reason information to Alexa's response, which should have been included as part of the response from the FAA service. Inspect the JSON the FAA webservice returns for detail.

Figure 2.17



Gold Challenge: Weather Information

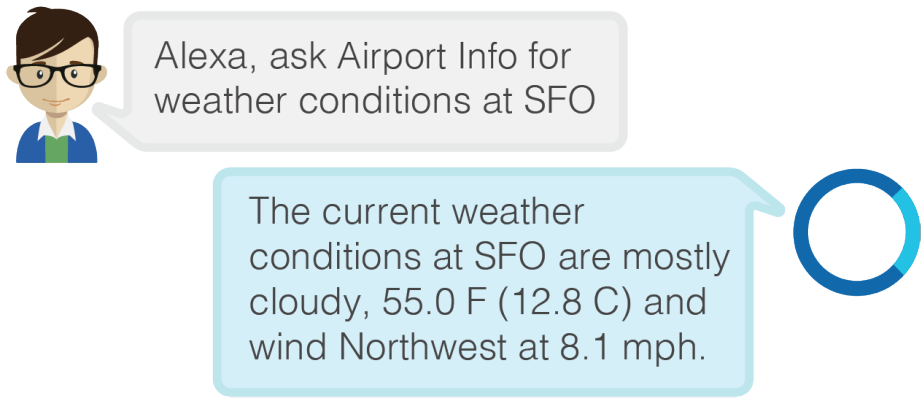
Users would like to know what the weather is like at the airport they have specified. If you inspect the FAA server response this information is included as part of the data:

Listing 2.23 Weather Information

```
"weather": {
   "visibility": 10,
   "weather": "A Few Clouds",
   "meta": {
      "credit": "NOAA's National Weather Service",
      "updated": "6:56 AM Local",
      "url": "http://weather.gov/"
   },
   "temp": "58.0 F (14.4 C)",
   "wind": "West at 4.6mph"
}
```

Use the "weather" portion of the response from the FAA service to create a new intent response for your skill. Weather information should include the weather conditions, temperature, and wind speed. Add a new intent called "airportWeatherIntent" to handle this request. The intent should respond to utterances matching "weather conditions at {AIRPORTCODE}".

Figure 2.18



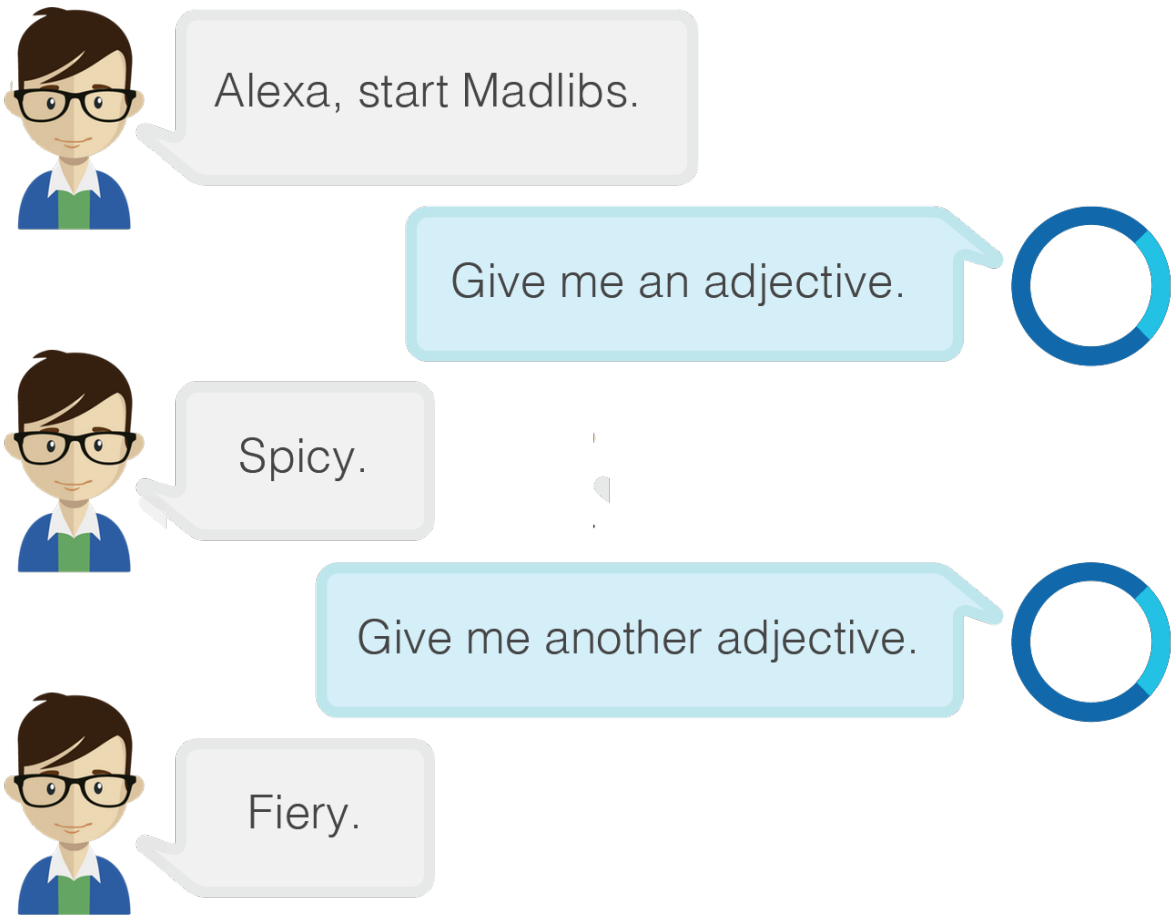
3

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.

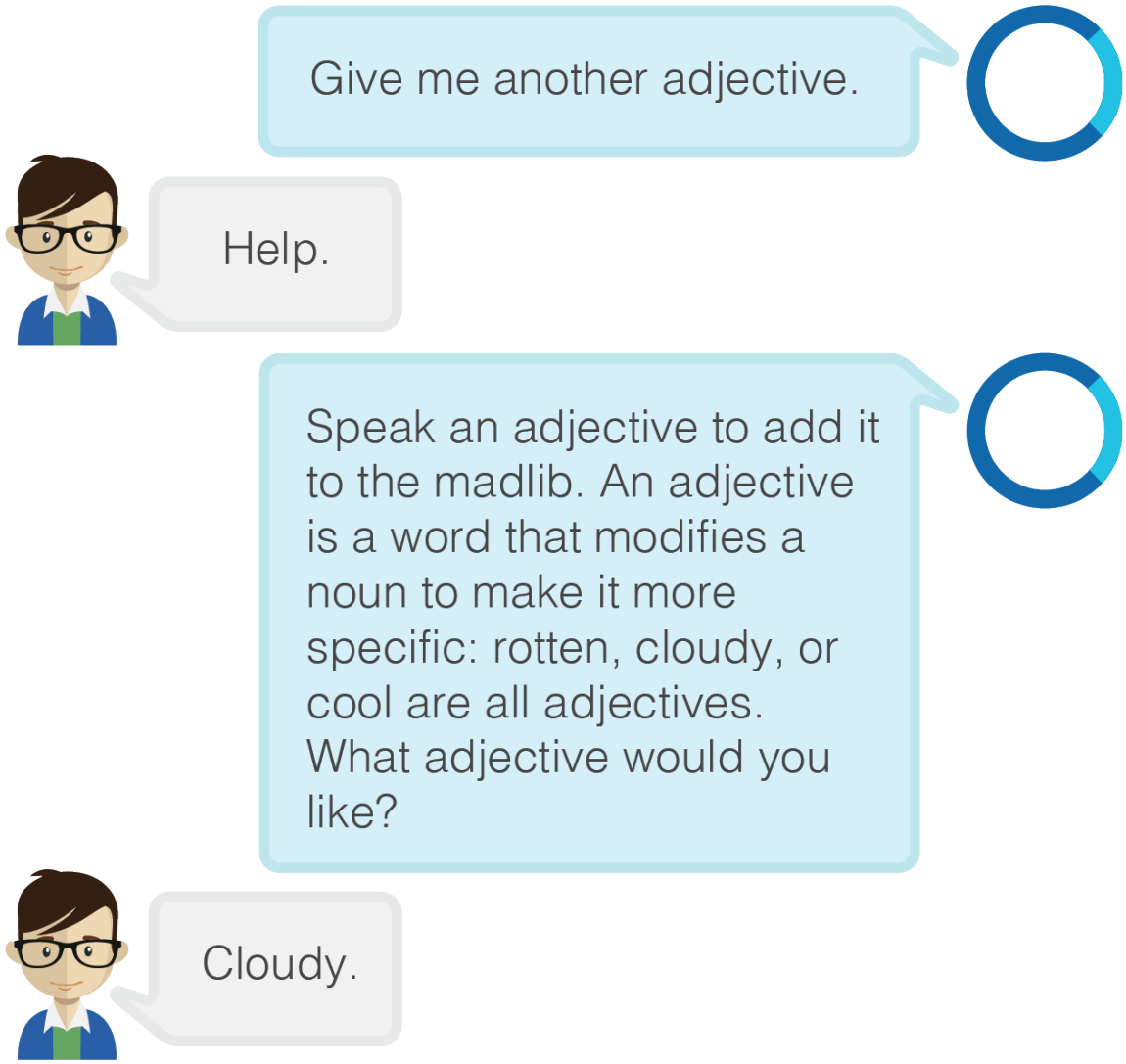
Madlib Builder will accept inputs for a series of adverbs, adjectives, nouns and so forth and will build a madlib based on the completed set of steps. Once all steps have been completed, the madlib will be read back. Examine the diagrams below which further illustrate Madlib Builder's interaction flow.

Figure 3.1 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.

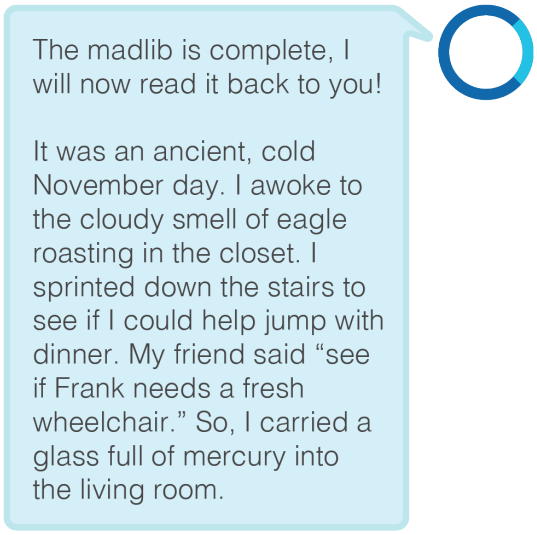
Figure 3.2 Requesting Help Along the Way



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!

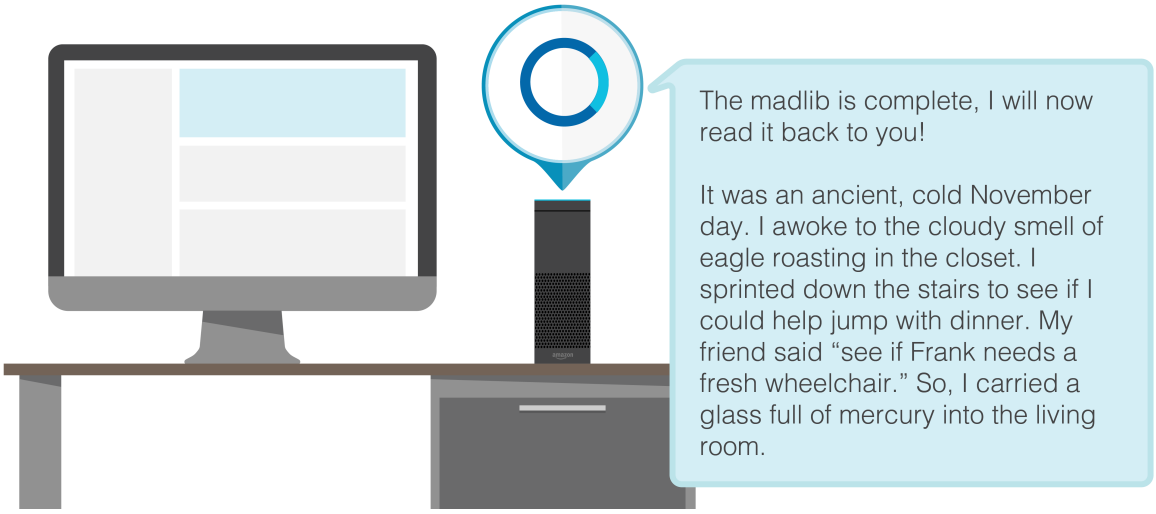
Figure 3.3 The Completed Madlib is Played Back



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

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

Figure 3.4 A Card is Displayed in the Alexa App



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 3.5 The Completed Madlib as a Card

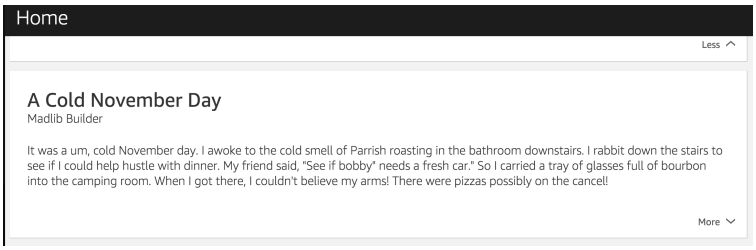
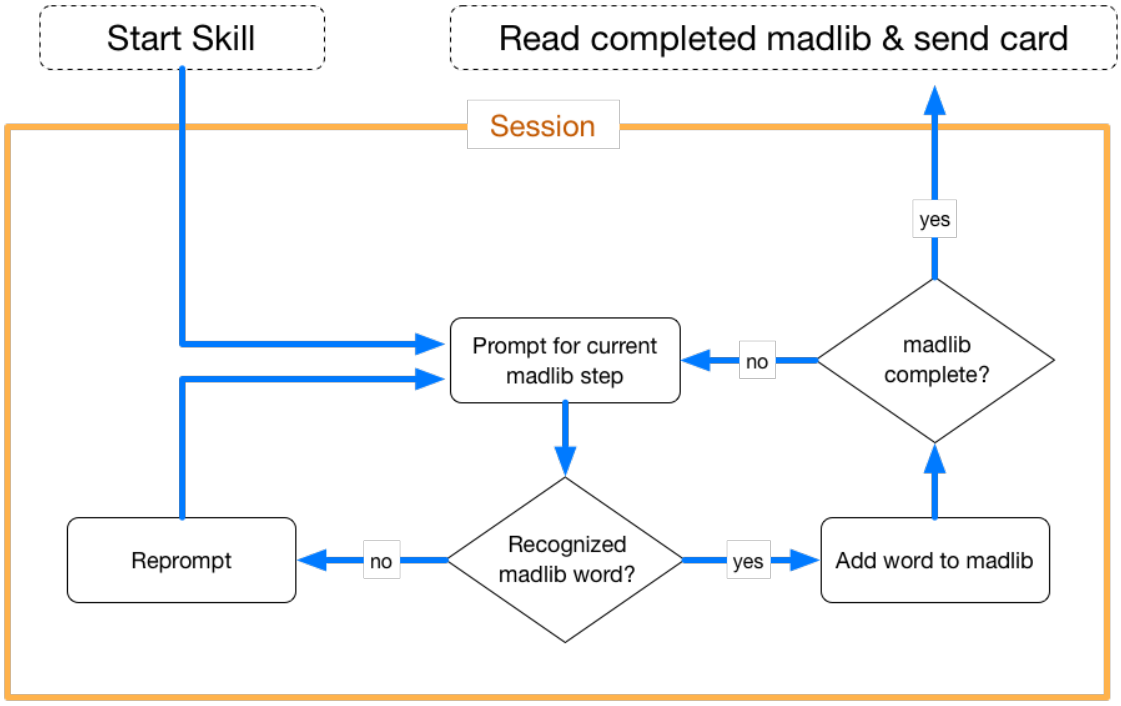


Figure 3.6 Madlib Builder Overview



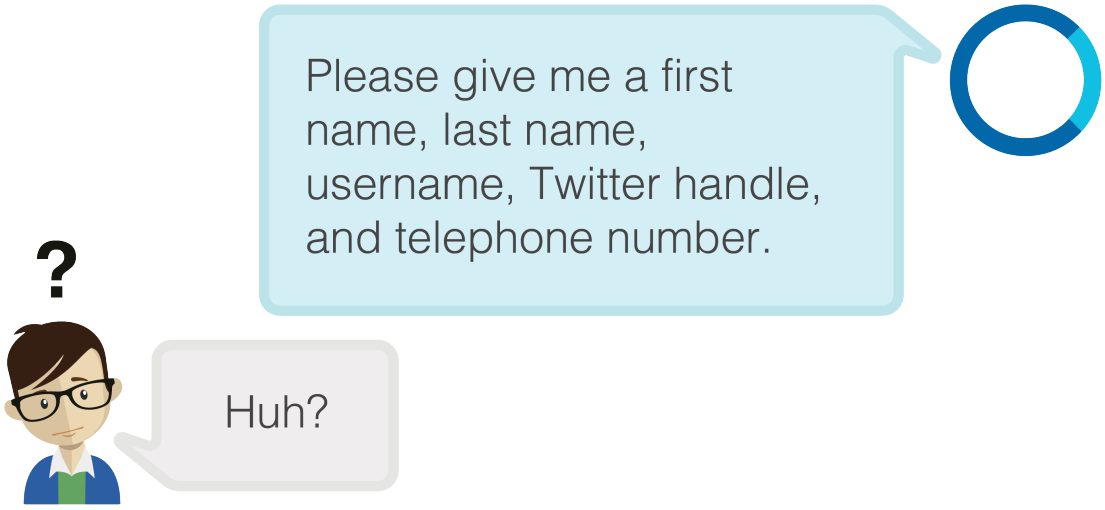
The flow chart above depicts the full path for a user interacting with the Madlib Builder skill from starting a madlib to completing one. Notice the orange "Session" rectange. This represents the accumulation of a bank of answers for Madlib Builder. Once the session begins, the skill proceeds with prompting 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 3.7 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 3.8 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 3.1 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 3.2 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".

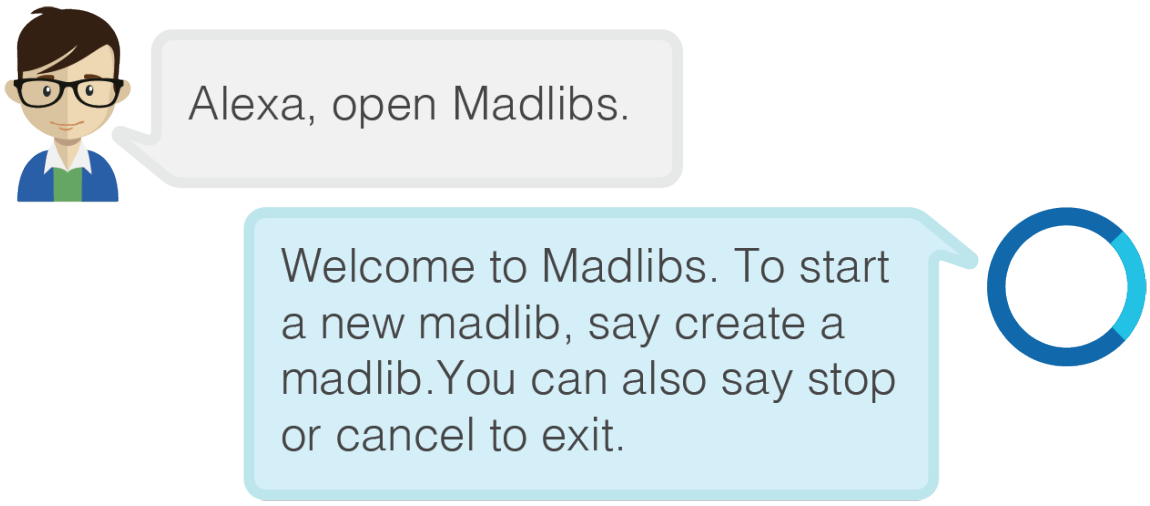
When considered from the perspective of Voice User Interface design, this type of invocation is called a *No Intent Invocation Type*.

Table 3.1 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 3.9 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 3.3 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 3.2 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 wish 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 3.4 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 `STEPVALUE` you will later register with the skill interface. The `STEPVALUE` will be used to retrieve a user's response to each madlib step.

Listing 3.5 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 3.6 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();
        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 3.7 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 3.8 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();
        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 3.9 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 3.10 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",
```

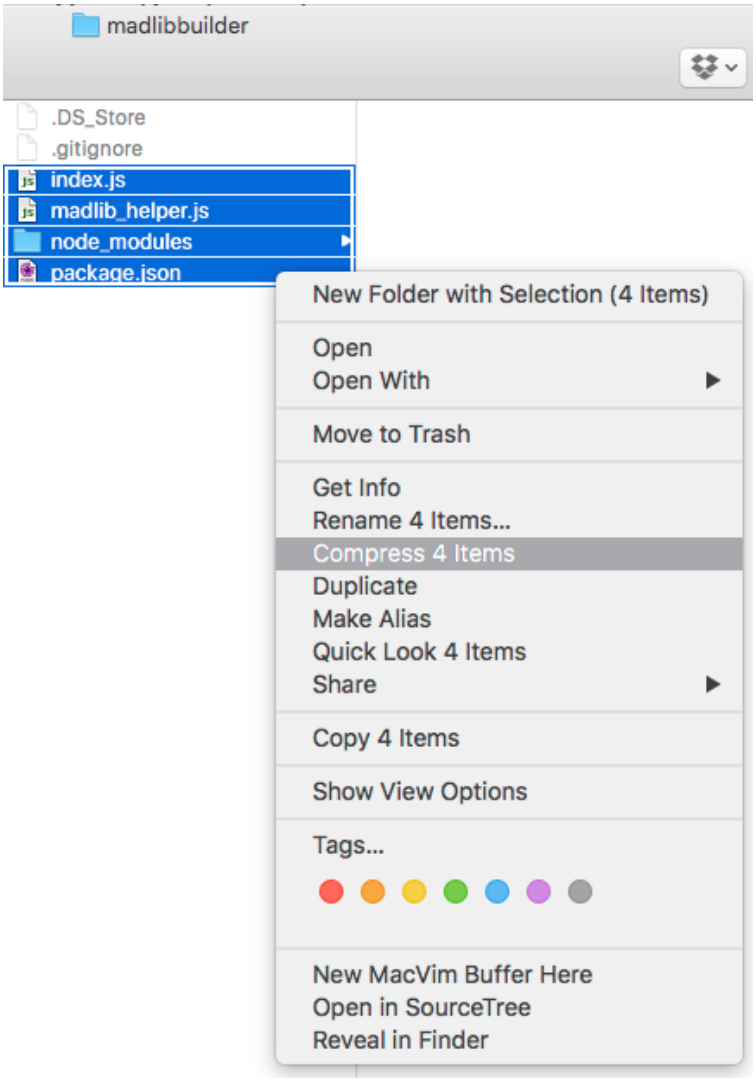
```
        "content": "It was a Test, cold November day.
        I awoke to the Test smell of Test roasting in the Test downstairs.
        I Test down the stairs to see if I could help Test with dinner.
        My friend said, \"See if Test\" needs a fresh Test.\"
        So I carried a tray of glasses 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 3.10 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.

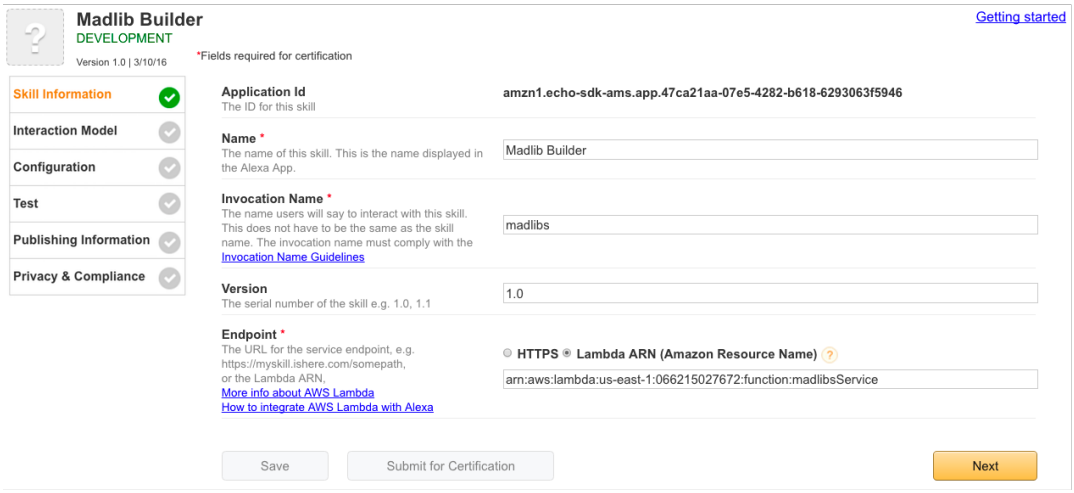
Configuring the Skill Interface

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
```

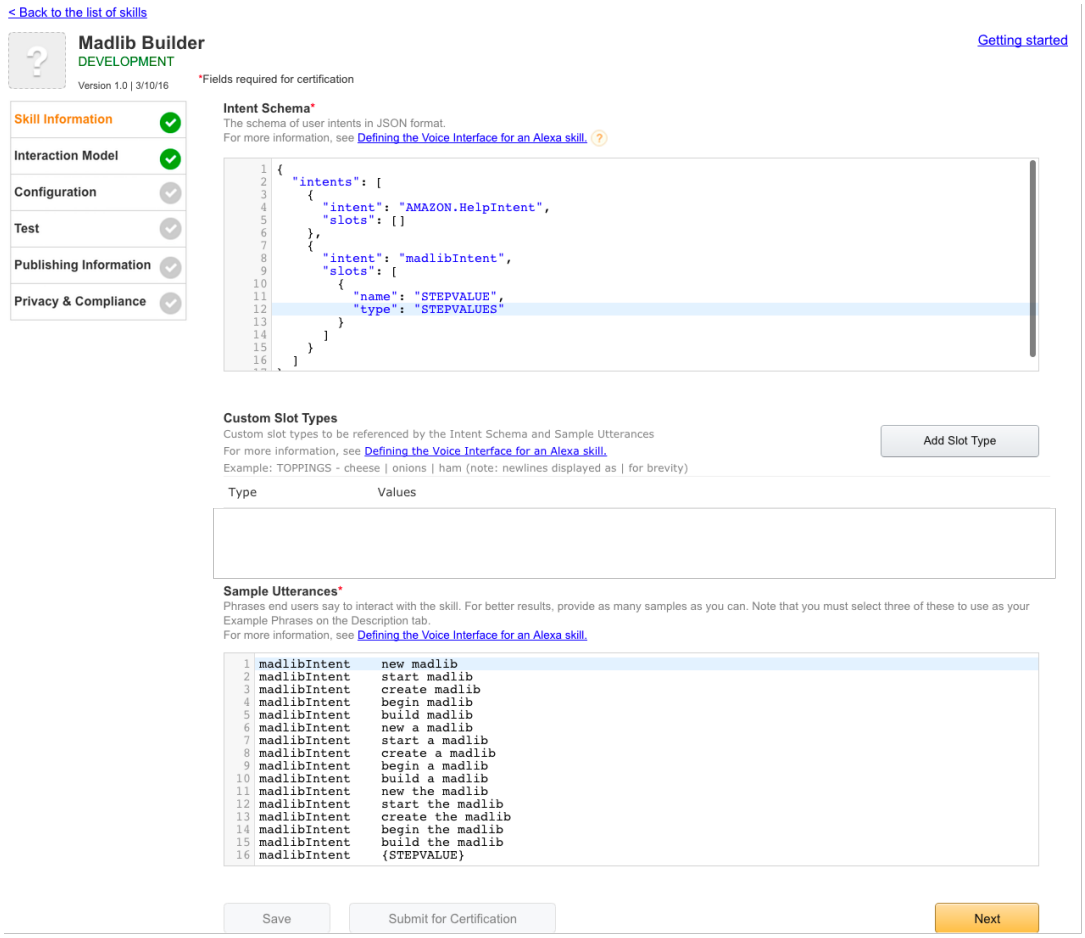
and click Add a New Skill. Enter "Madlib Builder" for Name and "madlibs" for Invocation Name. Paste the `ARN` that you copied down from the Lambda configuration in the field for Endpoint.

Figure 3.11 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 3.12 Configuring the Interaction Model

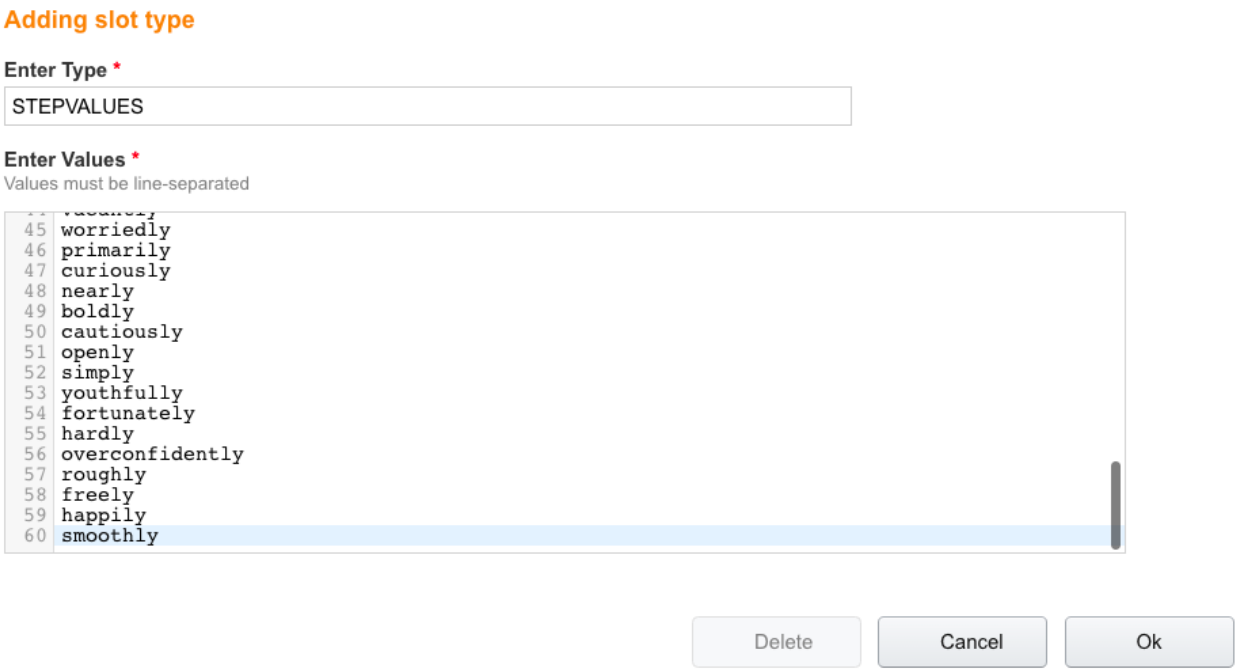


Last, define a custom slot type for the `STEPVALUE` value. Download the content for the `STEPVALUES` here :

`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.

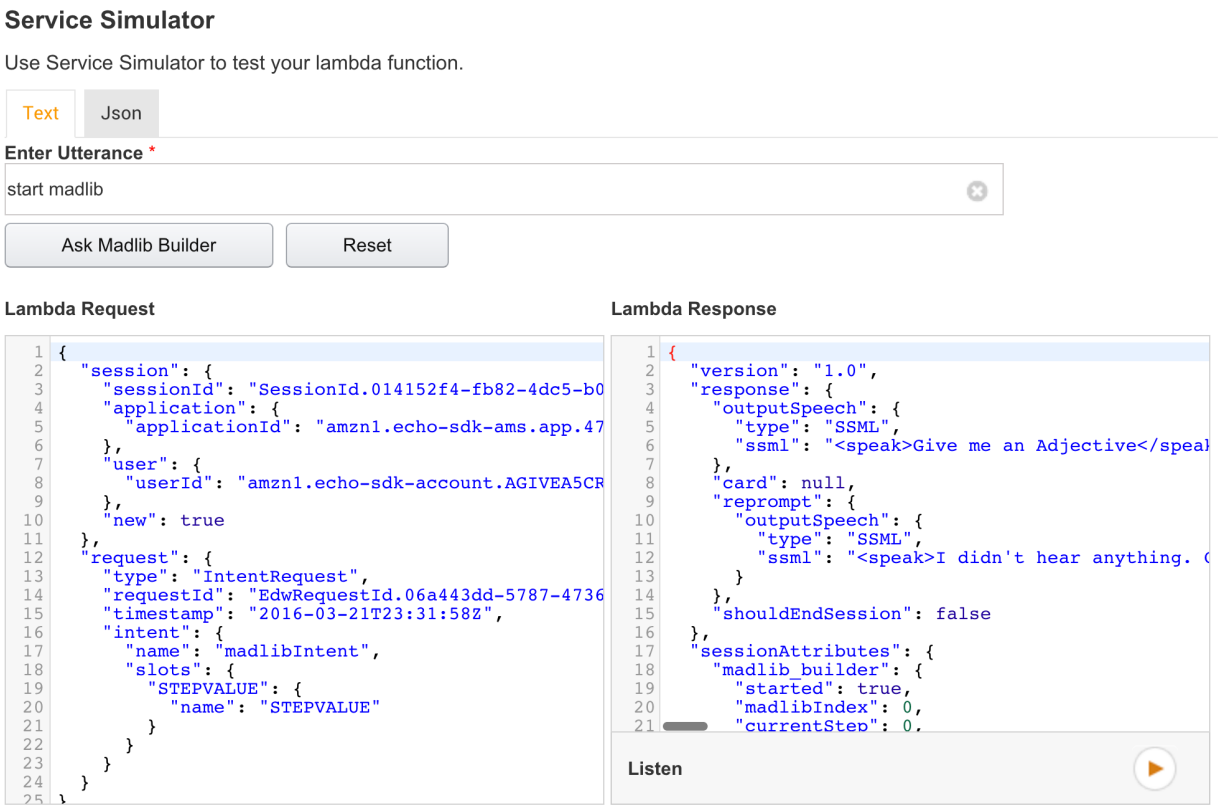
Figure 3.13 Defining the STEPVALUES Slot Type



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 3.14 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 3.15 Challenge - Madlib Template Chooser

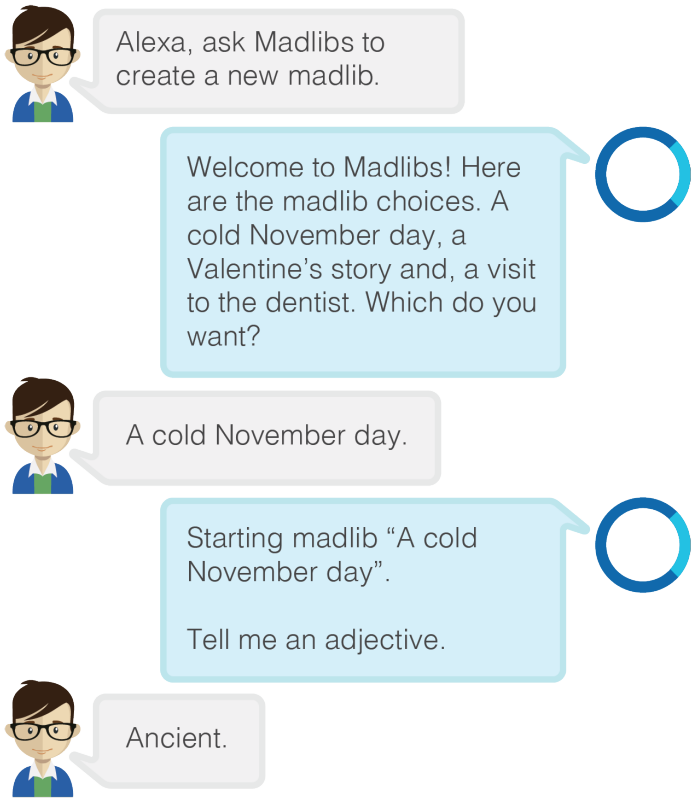
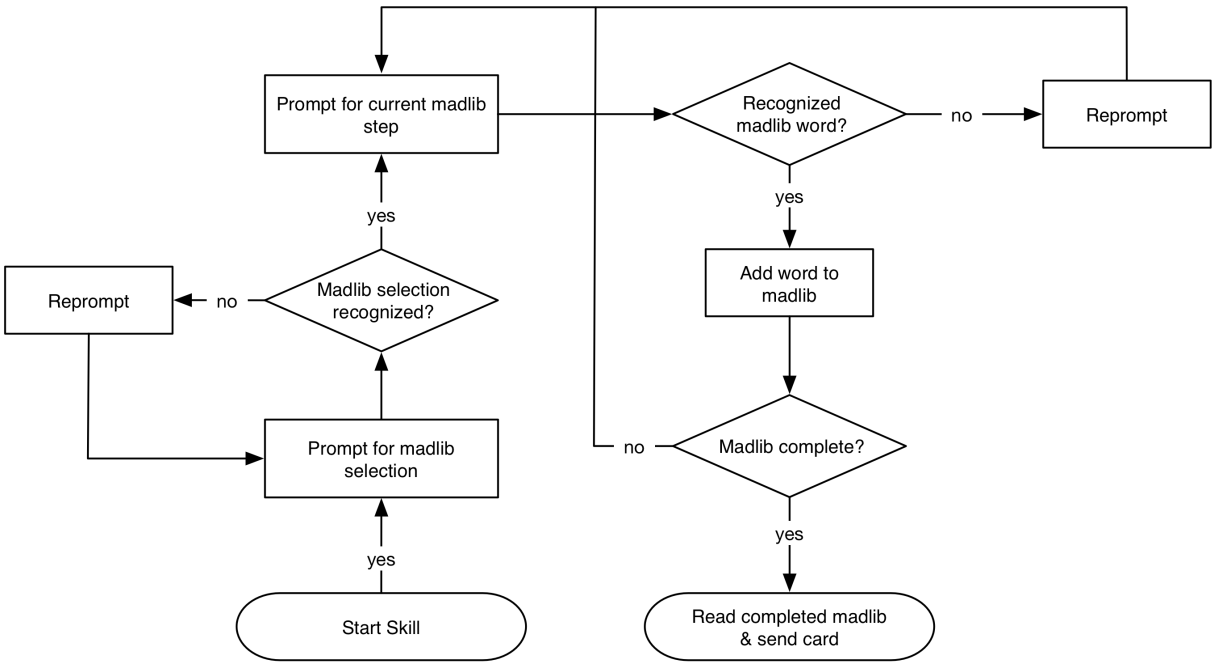


Figure 3.16 The Madlib Template Chooser is Added to the Workflow



4

Persistence

In this chapter you will implement save and load intent handlers for the Madlib Builder skill. These handlers will use a database to persist the state of the madlib, allowing users to save a madlib and resume work at a later time. Once you have implemented the save and load intent handlers, the following interaction will be possible:

Figure 4.1

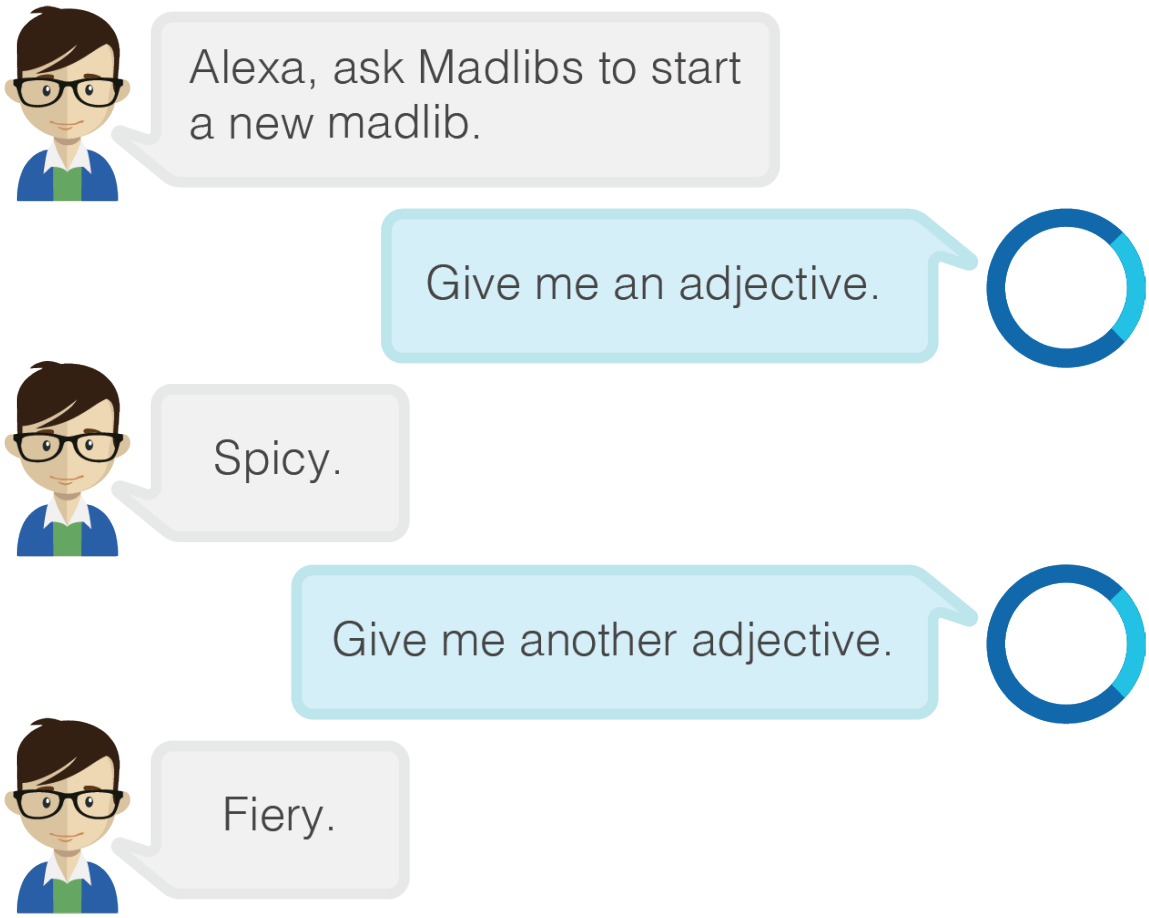
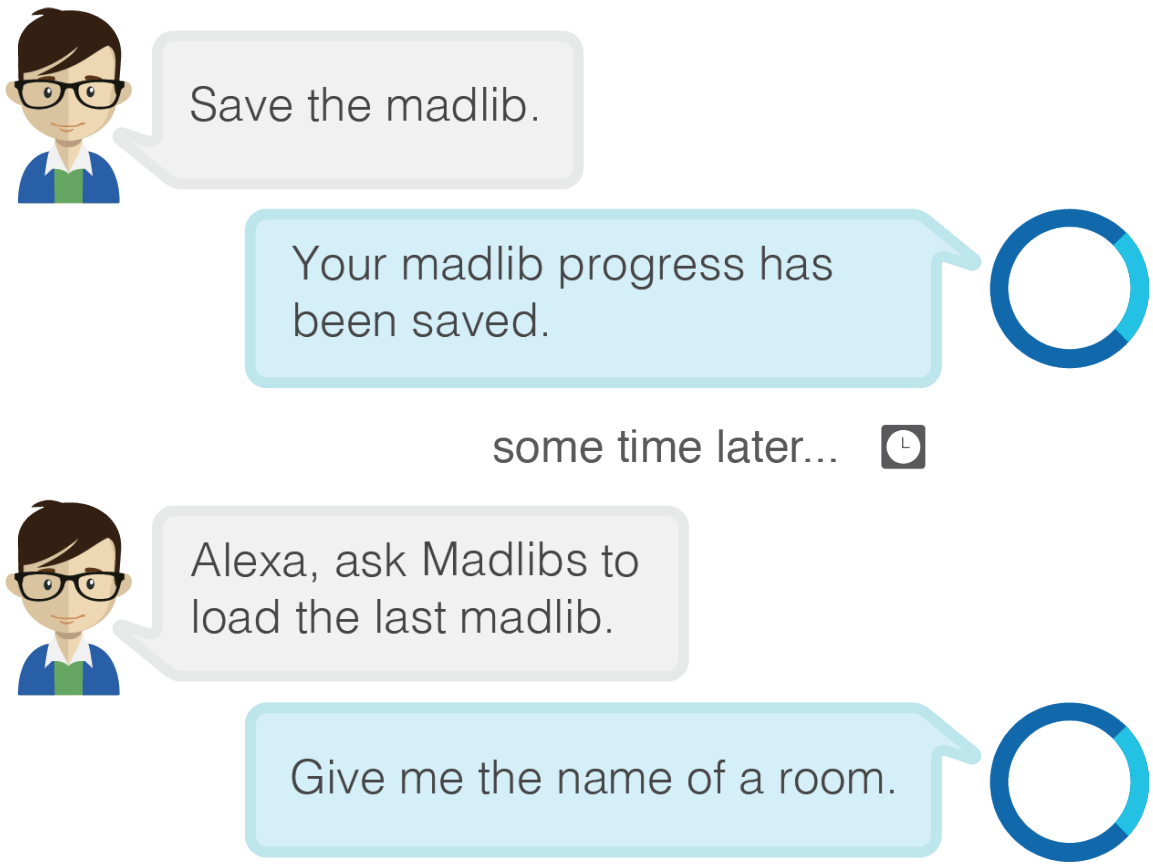


Figure 4.2



Notice, the madlib progress can now be saved and loaded by spoken command, allowing to resume work on a particular madlib at a later time.

For the database that will hold the state of the Madlib Builder progress, you will use `DynamoDB`, a key-value store that easily integrates with an `Amazon Lambda` hosted webservice. `DynamoDB` is a cloud-based Amazon service that also offers quick read/write times and NoSQL style schema. This means it requires no schema for the object that is stored in the table, only a definition for a key to access the data with.

Getting Started

Before implementing persistence, you first need to install `DynamoDB` on your system to support local development and testing. You will use the `brew` package manager to install `DynamoDB`. If you have not installed `brew` before, run the following command in your terminal:

Listing 4.1 Installing brew

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once the command completes, install `DynamoDB` locally by running the following command in your terminal:

Listing 4.2 Installing dynamodb locally

```
  $ brew install dynamodb-local
```

Now, start up `DynamoDB` locally by opening a new terminal tab and running the following command:

Listing 4.3 Start DynamoDB Locally

```
  $ dynamodb-local -sharedDb -inMemory -port 4000
```

Running the command should result in the following output:

```
Initializing DynamoDB Local with the following configuration:
Port:   4000
InMemory:       true
DbPath: null
SharedDb:       true
shouldDelayTransientStatuses:   false
CorsParams:     *
```

You will be building upon the existing `madlibbuilder` project from the last chapter. Change directories to the `madlibbuilder` project directory, and install the additional `Node.js` dependencies `database_helper.js` will require with the following command:

Listing 4.4 Installing Dependencies

```
$ npm install --save dynasty@0.2.4
```

Note the `@0.2.4` portion of the statement - this ensures installation of the most recent version of `Dynasty` that works correctly with a local dynamoDB instance. `Dynasty` is a `Node.js` library that lets you use `DynamoDB` from your skill. It also wraps query results in promise objects that allow you to handle asynchronous results easily.

Creating a DatabaseHelper

Within the `madlibbuilder` directory, add a new file called `database_helper.js` and add the following:

Listing 4.5 Defining DatabaseHelper's Local Database Config

```
'use strict';
module.change_code = 1;

var MadlibHelper = require('./madlib_helper');
var MADLIBS_DATA_TABLE_NAME = 'madlibsData';
var localUrl = 'http://localhost:4000';
var localCredentials = {
  region: 'us-east-1',
  accessKeyId: 'fake',
  secretAccessKey: 'fake'
};
var localDynasty = require('dynasty')(localCredentials, localUrl);
var dynasty = localDynasty;

function DatabaseHelper() {}

var madlibTable = function() {
  return dynasty.table(MADLIBS_DATA_TABLE_NAME);
};

module.exports = DatabaseHelper;
```

The code you have added constructs a new Dynasty object and sets it up to talk to the local `DynamoDB` instance you set up earlier. You will use this configuration in the local environment, and change it when deploying to the live environment.

You also added a method to look for a table called `madlibsData`, using the **`dynasty.table(tablename)`** method.

Creating the madlibsData Table

No definition for creating a `madlibsData` table in the development environment exists, so you add one to the helper.

Listing 4.6 Defining the createMadlibsTable function

```
var madlibTable = function() {
  return dynasty.table(MADLIBS_DATA_TABLE_NAME);
};
DatabaseHelper.prototype.createMadlibsTable = function() {
  return dynasty.describe(MADLIBS_DATA_TABLE_NAME)
    .catch(function(error) {
      console.log('createMadlibsTable::error: ', error);
      return dynasty.create(MADLIBS_DATA_TABLE_NAME, {
        key_schema: {
          hash: ['userId', 'string']
        }
      });
    });
};
module.exports = DatabaseHelper;
```

You have defined a method for creating a table when none is present. The describe method checks to see if a table exists and returns an error if one does not - at which point you instruct `DynamoDB` to create one. You defined the key for objects that will be written to the `madlibsData` table, an attribute called `userId` of type **`String`**. The key is what will be used for retrieving `madlibsData`.

Adding Store/Load Methods to the Helper

You will now add methods for saving and loading the madlib data to the `DynamoDB` database table. The data to write will be from the `MadlibHelper` object and written to the database as stringified `JSON`. Add the following to the end of the `database_helper.js` file:

Listing 4.7 Defining the storeMadlibData function

```
...
        key_schema: {
          hash: ['userId', 'string']
        }
      });
    });
};
DatabaseHelper.prototype.storeMadlibData = function(userId, madlibData) {
  console.log('writing madlibdata to database for user ' + userId);
  return madlibTable().insert({
    userId: userId,
    data: JSON.stringify(madlibData)
  }).catch(function(error) {
    console.log(error);
  });
};
```

Notice the `userId` that is passed to **`storeMadlibData(userId, madlibData)`**. This value represents the user's Alexa skill account id that the Alexa-enabled device is associated with and is sent with the request from the skill interface.

You pass the `userId` value along with the `madlibData` to the `DatabaseHelper` object in order to save it to the database. The `userId` is used to uniquely associate the madlibData with the user account. You also "stringified" (converted the object to a JSON string representation) the madlibData object so that it can be properly written to the database.

Next, you will add a **`readMadlibData(userId)`** method to `database_helper.js`. **`readMadlibData`** will return the saved data for the madlib as a `MadlibHelper` object.

Listing 4.8 Adding the readMadlibData Method

```
    console.log(error);
  });
};
DatabaseHelper.prototype.readMadlibData = function(userId) {
  console.log('reading madlib with user id of : ' + userId);
  return madlibTable().find(userId)
    .then(function(result) {
      var data = (result === undefined ? {} : JSON.parse(result['data']));
      return new MadlibHelper(data);
    }).catch(function(error) {
    console.log(error);
  });
};
module.exports = DatabaseHelper;
```

Creating the Development Database Table

You will now use the `DatabaseHelper` to create the `madlibData` table in the local developement environment. Open `index.js` and add the following initialization to the top of the file, just after `var MadlibHelper = require('./madlib_helper');`

Listing 4.9 Initializing the databaseHelper

```
  'use strict';
  module.change_code = 1;

  var skill = require('alexa-app');
  var MADLIB_BUILDER_SESSION_KEY = 'madlib_builder';
  var skillservice = new skill.app('madlibbuilder');
  var MadlibHelper = require('./madlib_helper');
  var DatabaseHelper = require('./database_helper');
  var databaseHelper = new DatabaseHelper();
```

Next, add the following code just after the DatabaseHelper variable declaration:

Listing 4.10 Adding the pre Method

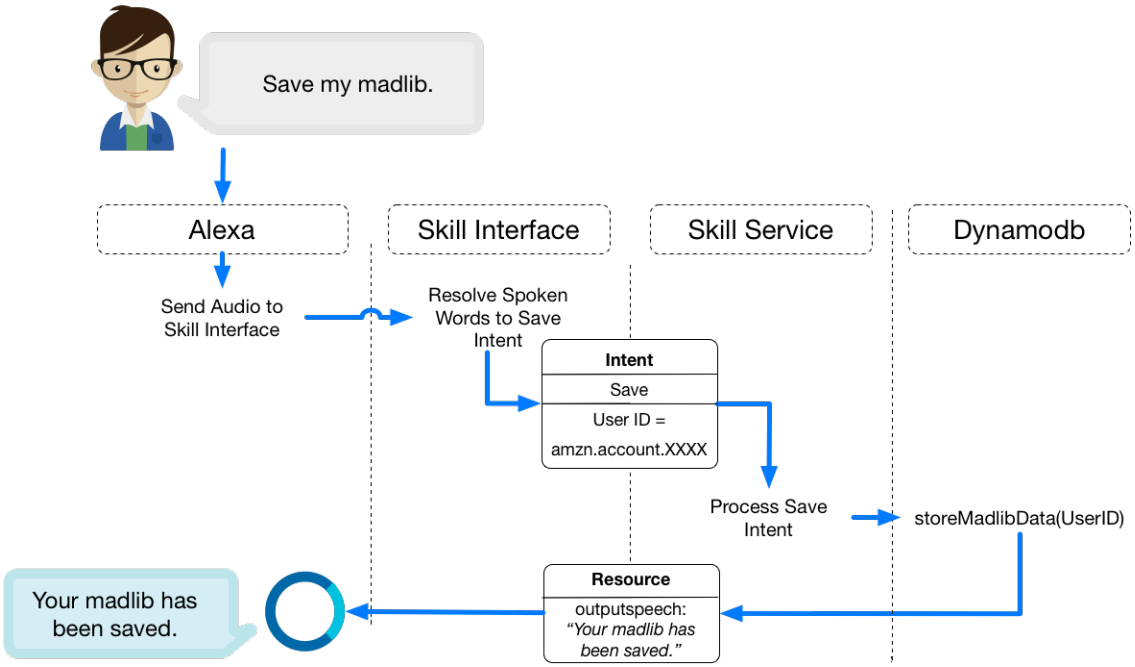
```
...
var DatabaseHelper = require('./database_helper');
skillService.pre = function(request, response, type) {
  databaseHelper.createMadlibsTable();
};
```

the **`pre`** method is called before any intent handler, and will ensure a table to hold the data is created before any function/code tries to access it in the development environment.

Adding a Save Intent

You next will implement an intent handler to enable users to request the skill and save the `madlib` data to the database.

Figure 4.3 Saving the madlib Data to the Database



The diagram above shows a user's spoken words resolved to the skill service. Once resolved to the `saveMadlibIntent` by the skill interface, the matching intent on the service is called and the `userId` and MadlibHelper are passed to the `DatabaseHelper`. The result is that the data for the `MadlibHelper` is persisted to the `DynamoDB` instance and associated with the user's unique ID.

Modify the **`getMadlibHelper`** method as follows:

Listing 4.11 Modifying the getMadlibHelper method

```
var getMadlibHelper = function(~~request~~madlibHelperData) {
  ~~var madlibHelperData = request.session(MADLIB_BUILDER_SESSION_KEY);~~
  if (madlibHelperData === undefined) {
    madlibHelperData = {};
  };
  return new MadlibHelper(madlibHelperData);
};
```

Next, add the following before the `module.exports = skillService;` line at the end of `index.js`:

Listing 4.12 Adding the saveMadLibIntent Handler

```
var getMadlibHelperFromRequest = function(request) {
  var madlibHelperData = request.session(MADLIB_BUILDER_SESSION_KEY);
  return getMadlibHelper(madlibHelperData);
};

skillService.intent('saveMadlibIntent', {
    'utterances': ['{save} {|a|the|my} madlib']
  },
  function(request, response) {
    var userId = request.userId;
    var madlibHelper = getMadlibHelperFromRequest(request);
    databaseHelper.storeMadlibData(userId, madlibHelper).then(
      function(result) {
        return result;
      }).catch(function(error) {
      console.log(error);
    });
    response.say('Your madlib progress has been saved.');
    response.shouldEndSession(true).send();
    return false;
  }
);

module.exports = skillService;
```

The **saveMadlibIntent** you added passes the `madlibHelper` and `userId` to the `databaseHelper` so that it can be saved, and lets users know the save completed by speaking a response of "The madlib has been saved". You also added a convenience method for fetching the **madlibHelperData** from the session and constructing a new **MadlibHelper** from that data if present.

Refactoring the madlibIntent Handler

Before proceeding with implementing the loading feature, some changes are needed to the existing `madlibIntent` handler to support the desired behavior of users being able to pick back up where they left off. The logic that madlibIntent holds for dealing with a `MadlibHelper` and generating the response should be pulled out into a method, so that you can call it from the `loadMadlibIntent` intent handler you will soon define. Update the **madlibIntent** method as follows:

Listing 4.13 Modifying the madlibIntent Handler

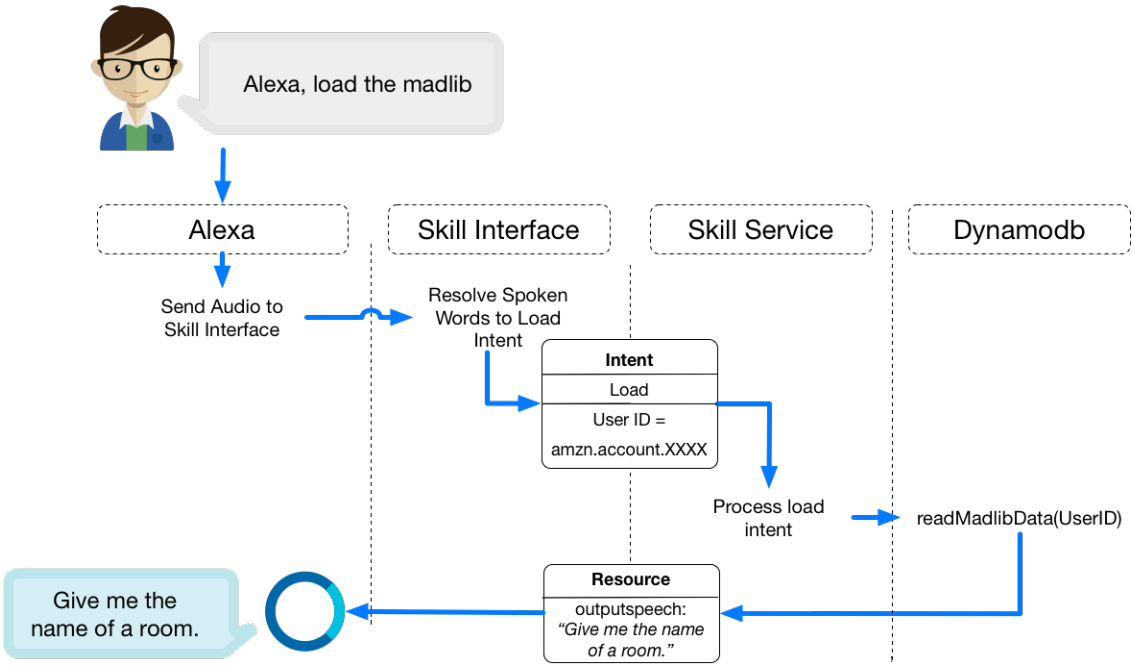
```
skillService.intent('madlibIntent', {
    'slots': {
      'STEPVALUE': 'STEPVALUES'
    },
    'utterances': ['{new|start|create|begin|build} {|a|the} madlib',
      '{-|STEPVALUE}'
    ]
  },
  function(request, response) {
    //check to see if a madlibbuilder exists in the request.
    var stepValue = request.slot('STEPVALUE');
    var madlibHelper = getMadlibHelper(request);
    madlibHelper.started = true;
    if (stepValue !== undefined) {
      madlibHelper.getStep().value = stepValue;
    }
    if (madlibHelper.completed()) {
      var completedMadlib = madlibHelper.buildMadlib();
      console.log('madlib completed! Result: ' + completedMadlib);
      response.card(madlibHelper.currentMadlib().title, completedMadlib, 'your completed madlib');
      response.say('The madlib is complete! I will now read it to you. ' + madlibHelper.buildMadlib());
      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);
    madlibIntentFunction(getMadlibHelperFromRequest(request), request, response);
  }
);
var madlibIntentFunction = function(madlibHelper, request, response) {
  var stepValue = request.slot('STEPVALUE');
  madlibHelper.started = true;
  if (stepValue !== undefined) {
    madlibHelper.getStep().value = stepValue;
  }
  if (madlibHelper.completed()) {
    var completedMadlib = madlibHelper.buildMadlib();
    response.card(madlibHelper.currentMadlib().title, completedMadlib,
      'your completed madlib');
    response.say('The madlib is complete! I will now read it to you. ' +
        madlibHelper.buildMadlib());
    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);
  response.send();
};
...
```

You have extracted the logic from the `madlibLibIntent` handler into a new function that can now be called from multiple intent handlers.

Adding a Load Intent

Now that the madlib can be saved and you have extracted the logic for responding with the **`MadlibHelper`**'s current step into a reusable method, we will implement an intent that allows users to resume or load the madlib they were working on. In diagram form, here is what that request will look like:

Figure 4.4 Loading the Madlib Data from the Database



The data that was persisted to the `DynamoDB` database was keyed on the `userId` associated with the enabled skill. To retrieve it, you will pass the `userId` value from the request to the `readMadlibData(userId)` so that the `madlibData` can be fetched from the database. Add the following before the `module.exports = skillService;` line at the end of `index.js`:

Listing 4.14 Adding the loadMadlibIntent Handler

```
  ...
  response.say('Your madlib progress has been saved.');
  response.shouldEndSession(true).send();
  return false;
}
);
skillService.intent('loadMadlibIntent', {
    'utterances': ['{load|resume} {|a|the} {|last} madlib']
  },
  function(request, response) {
    var userId = request.userId;
    databaseHelper.readMadlibData(userId).then(
      function(loadedMadlibHelper) {
        console.log("got", loadedMadlibHelper);
        return madlibIntentFunction(loadedMadlibHelper, request, response);
      });
    return false;
  }
);

module.exports = skillService;
```

Testing the Save and Load Handlers

Now you test the intent handlers you implemented in the `alexa-app-server` test page. Ensure that `alexa-app-server` is running by changing to the `alexa-app-server/examples` directory and running the following:

Listing 4.15 Starting alexa-app-server

```
$ node server
```

Visit the `alexa-app-server` test page, at `http://localhost:8080/alexa/madlibbuilder`. You will test that a `madlib` can be saved at its current step and resumed when loaded. Advance the madlib 3 steps forward by selecting IntentRequest for Type, MadlibIntent for Intent, and enter "test" for STEPVALUE. Click Send Request 3 times. In the Response portion of the page, verify the following text:

```
    {
  "version": "1.0",
  "sessionAttributes": {
    "madlib_builder": {
      "started": true,
      "madlibIndex": 0,
      "currentStep": 3,
      "madlibs": [
        {
          "title": "A Cold November Day",
          "template": "It was a ${adjective_1}, cold November day.
          //removed for brevity
          "steps": [
            {
              "value": "ATL",
              "template_key": "adjective_1",
              "prompt": "an Adjective",
              //removed for brevity
            },
          ...
          ]
        }
      ]
    }
  },
  "response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Give me a name of Room in a house</speak>"
    },
    "reprompt": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I didn't hear anything. Give me a name of Room in a house to continue.</speak>"
      }
    }
  },
  "dummy": "text"
}
```

The state of the Madlib Builder progress is now on step 4. Test that the **`saveMadlibIntent`** handler works correctly by selecting saveMadlibIntent in the Intent dropdown and pressing Send Request. In the Response area, you should see the following:

```
{
  "version": "1.0",
  "sessionAttributes": {
    "madlib_builder": {
      "started": true,
      "madlibIndex": 0,
      "currentStep": 3,
```

```
      "madlibs": [
        {
          "title": "A Cold November Day",
          "template": "It was a ${adjective_1}, cold November day.
          //shortened for brevity
          "steps": [
            {
              "value": "test",
              "template_key": "adjective_1",
              "prompt": "an Adjective",
              //shortened
            },
            ...
          ]
        }
      ]
    }
  },
  "response": {
    "shouldEndSession": true,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Your madlib progress has been saved.</speak>"
    }
  },
  "dummy": "text"
}
```

Next, test loading the `madlib` by selecting loadMadlibIntent in the Intent dropdown and pressing Send Request. You should observe the following response in the Server Test page's Response area:

```
{
  "version": "1.0",
  "sessionAttributes": {
    "madlib_builder": {
      "started": true,
      "madlibIndex": 0,
      "currentStep": 3,
      "madlibs": [
        {
          "title": "A Cold November Day",
          "template": "It was a ${adjective_1}, cold November day.
          //shortened for brevity
          "steps": [
            {
              "value": "test",
              "template_key": "adjective_1",
              "prompt": "an Adjective",
            },
            {
              "value": "test",
              "template_key": "adjective_2",
              "prompt": "another Adjective",
            },
            {
              "value": "test",
              "template_key": "type_of_bird",
              "prompt": "a Type of bird",
            },
            {
              "value": null,
              "template_key": "room_in_house",
              "prompt": "a name of Room in a house",
            },
            //shortened for brevity
            {
              "value": null,
```

```
              "template_key": "noun_2",
              "prompt": "a noun",
            }
          ]
        }
      ]
    }
  },
  "response": {
    "shouldEndSession": false,
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Give me a name of Room in a house</speak>"
    },
    "reprompt": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I didn't hear anything. Give me a name of Room in a house to continue.</speak>"
      }
    }
  },
  "dummy": "text"
}
```

Your next step will be to update the skill interface with the updated Schema and Utterances information. Before continuing on to the next step, copy the updated Schema and Utterances information on the Test page to a text file.

Figure 4.5 Copying the Updated Schema and Utterances from the Test Page

Schema

```
{
  "intents": [
    {
      "intent": "AMAZON.HelpIntent",
      "slots": []
    },
    {
      "intent": "loadMadlibIntent",
      "slots": []
    },
    {
      "intent": "madlibIntent",
      "slots": [
        {
          "name": "STEPVALUE",
          "type": "STEPVALUES"
        }
      ]
    },
    {
      "intent": "saveMadlibIntent",
      "slots": []
    }
  ]
}
```

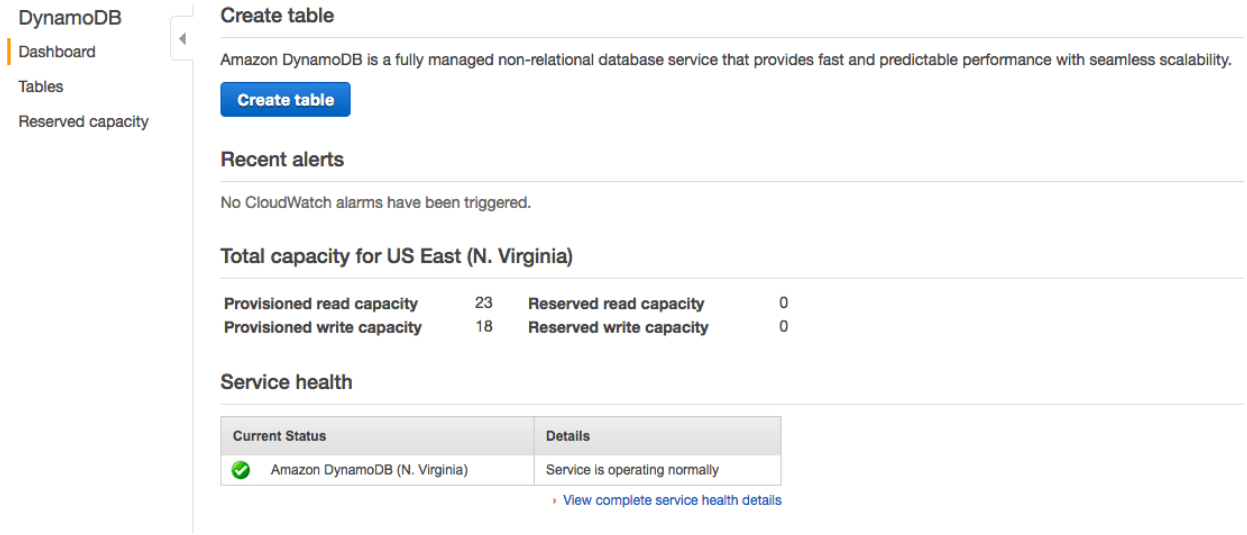
Utterances

```
loadMadlibIntent        load madlib
loadMadlibIntent        resume madlib
loadMadlibIntent        load a madlib
loadMadlibIntent        resume a madlib
loadMadlibIntent        load the madlib
loadMadlibIntent        resume the madlib
loadMadlibIntent        load last madlib
loadMadlibIntent        resume last madlib
loadMadlibIntent        load a last madlib
loadMadlibIntent        resume a last madlib
loadMadlibIntent        load the last madlib
loadMadlibIntent        resume the last madlib
madlibIntent    new madlib
madlibIntent    start madlib
madlibIntent    create madlib
madlibIntent    begin madlib
madlibIntent    build madlib
madlibIntent    new a madlib
madlibIntent    start a madlib
madlibIntent    create a madlib
madlibIntent    begin a madlib
madlibIntent    build a madlib
madlibIntent    new the madlib
madlibIntent    start the madlib
madlibIntent    create the madlib
madlibIntent    begin the madlib
madlibIntent    build the madlib
madlibIntent    {STEPVALUE}
saveMadlibIntent        save madlib
saveMadlibIntent        save a madlib
saveMadlibIntent        save the madlib
saveMadlibIntent        save my madlib
```

Deployment

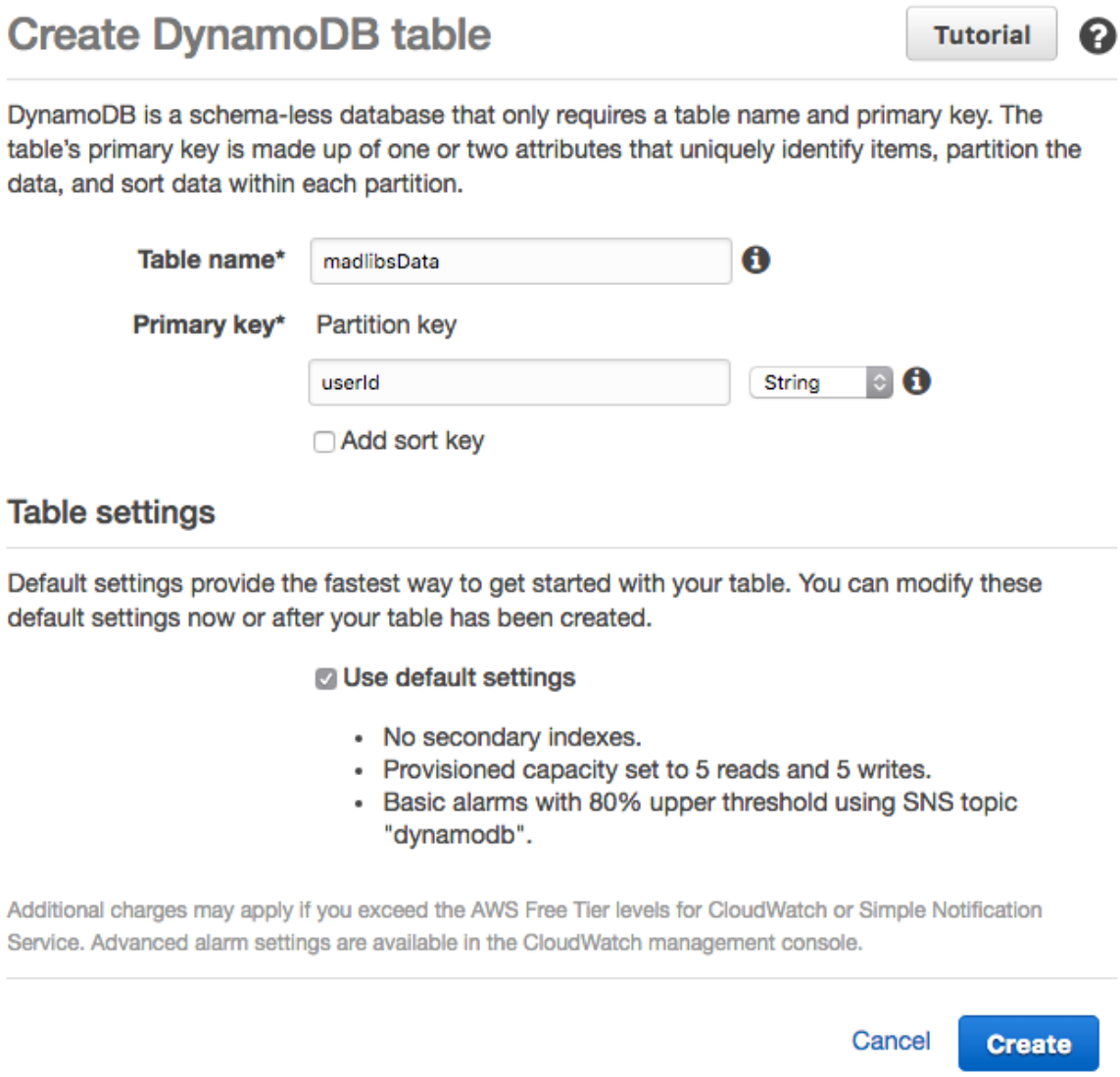
To begin deployment, you first need to configure the AWS DynamoDB to work correctly with your skill. To configure the database, visit `https://console.aws.amazon.com/dynamodb/home?region=us-east-1`.

Figure 4.6 The AWS DynamoDB Page



Click Create table. You now enter the details for the new `DynamoDB` table.

Figure 4.7 Creating a DynamoDB table



For the Table name* field, enter `madlibsData`. For the Primary key* field, enter `userId`. Now, click "Create".

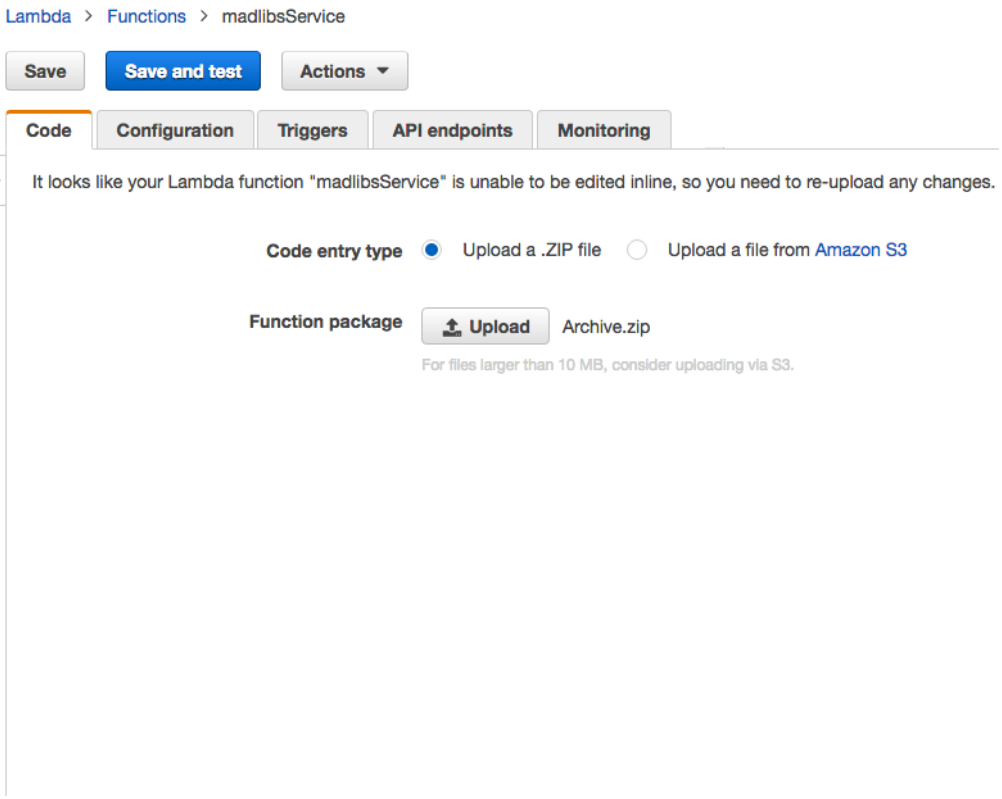
Next, edit the `database_helper.js` database configuration so that it is ready for the live environment. Edit `database_helper.js`:

Listing 4.16 Modifying the Database Connection Settings

```
'use strict';
module.change_code = 1;
var _ = require('lodash');
var MadlibHelper = require('./madlib_helper');
var MADLIBS_DATA_TABLE_NAME = 'madlibsData';
var localUrl = 'http://localhost:4000',
var localCredentials = {
  region: 'us-east-1',
  accessKeyId: 'fake',
  secretAccessKey: 'fake'
};
var localDynasty = require('dynasty')(localCredentials, localUrl),
var dynasty = localDynasty;
var dynasty = require('dynasty')({});
function DatabaseHelper() {}
var madlibTable = function() {
  return dynasty.table(MADLIBS_DATA_TABLE_NAME);
};
```

To deploy the skill to AWS, you update the source code that is running on the AWS Lambda function you set up earlier for the Madlib Builder skill. Compress the contents within the `/madlibsbuilder` directory to create a new Archive. Next, return to the "madlibsService" AWS Lambda function you set up earlier under `https://console.aws.amazon.com/lambda/`. Within the Code tab, click Upload and select the updated code archive you created.

Figure 4.8 Updating the Lambda Function Code



Next, click on the Configuration tab. You will update the Role so that access to a DynamoDB database is allowed from the Lambda function. Select Basic with DynamoDB under the Role dropdown. If Basic with DynamoDB is not shown in the available Roles list, you will need to create a new role that allows access to Dynamodb. To create a

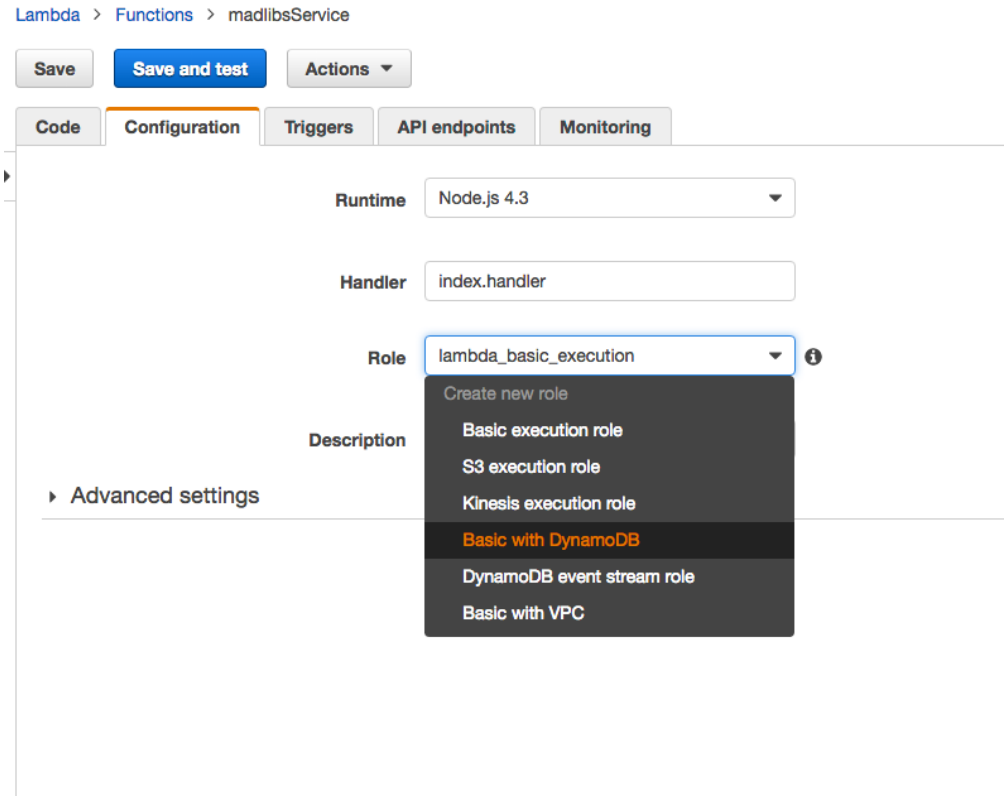
role, refer to the steps in chapter one for creating a new role. Name the new Role "basic_with_dynamodb", using the following for the role document:

Listing 4.17 Modifying the Database Connection Settings

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Refer to `https://gist.github.com/unitygirl/3b0bdc0f0826fb88448cf17ac4a7293b` for a detailed summary of creating the new dynamodb role document if needed. You will be redirected to a new page. On this page, click Allow at the bottom right of the screen. After creating the role and clicking allow, ensure the new role is selected for the Role field in the Configuration tab.

Figure 4.9 Updating the Role

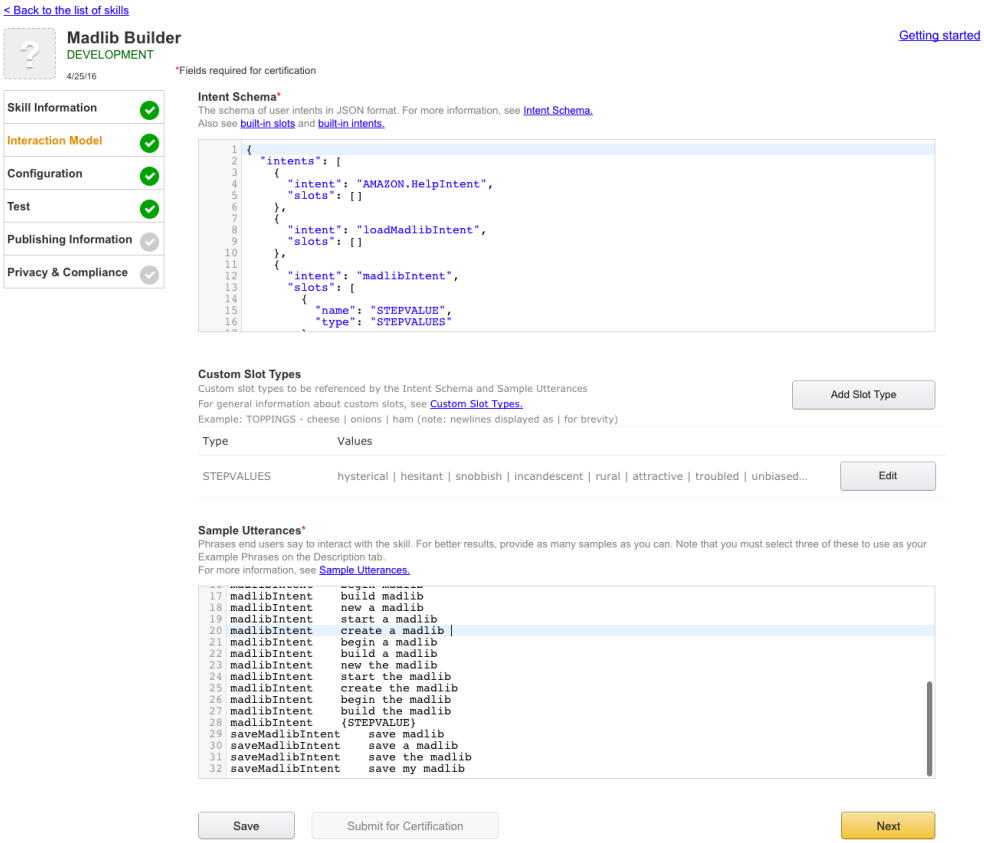


Finally, once you are redirected to the service page, click Save at the top left of the page.

Updating the Skill Interface Intent Schema and Utterances

Visit the skill interface you set up earlier for Madlib Builder, under `https://developer.amazon.com/edw/home.html#/skills/list` and advance to the Interaction Model section. Update the Intent Schema and Sample Utterances fields from the respective Schema and Utterance values you copied earlier from the alexa-app-server Test page, and click Save at the bottom right of the page.

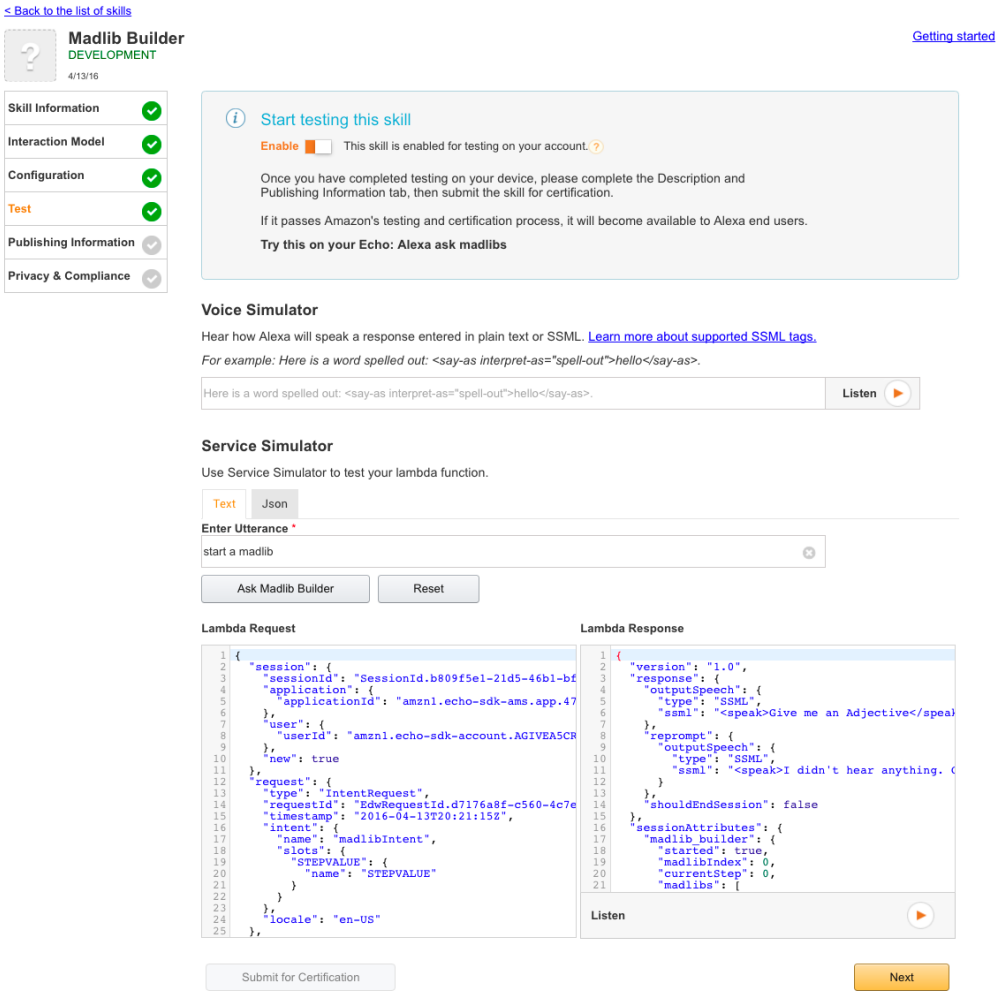
Figure 4.10 Updating the Interaction Model



Testing the Skill in the Service Simulator

Now that the skill service and skill interface have been updated, you can test the skill in the "Test" section of the skill interface. Return to the Madlib Builder skill you configured earlier under `https://developer.amazon.com/edw/home.html#/skills/list` and advance to the Test section in the skill interface. In the Enter Utterance section, enter "start a madlib" and press Ask Madlib Builder.

Figure 4.11 Testing the Madlib



Next, enter "test" in the Enter Utterance field and press Ask Madlib Builder two times. This should advance the state of the Madlib Builder progress to step three. Enter "save the madlib" in the Enter Utterance field. Now enter "load the madlib" in the Enter Utterance field. You should observe the following Lambda Response output:

Listing 4.18 Testing the Store/Load Functionality

```
     {
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>Give me a Type of bird</speak>"
    },
    "reprompt": {
      "outputSpeech": {
        "type": "SSML",
        "ssml": "<speak>I didn't hear anything. Give me a Type of bird to continue.</speak>"
      }
    },
    "shouldEndSession": false
  },
  "sessionAttributes": {
    "madlib_builder": {
      "started": true,
      "madlibIndex": 0,
      "currentStep": 2,
      "madlibs": [
        {
          "title": "A Cold November Day",
          "template": //removed for brevity
          "steps": [
            {
              "value": "test",
              "template_key": "adjective_1",
              "prompt": "an Adjective",
              "help": //removed for brevity
            },
            {
              "value": "test",
              "template_key": "adjective_2",
              "prompt": "another Adjective",
              "help": //removed for brevity
            },
            {
              "template_key": "type_of_bird",
              "prompt": "a Type of bird",
              "help": //removed for brevity
            },
            ....//removed for brevity
          ]
        }
      ]
    }
  }
}
```

Congratulations, you have successfully implemented persistence in the Madlib Builder skill and deployed to AWS! You may now test the Madlib Builder database functionality on a real device if one is available.

Challenge: Implicit Saves

In addition to providing the option to explicitly save a madlib, change the skill service so that the Madlib Builder progress is saved implicitly as users advance through completing the madlib.

5
Account Linking

In this chapter you will extend the Airport Info skill you previously built, granting users the ability to tweet the status of the airport they have requested to their personal Twitter timeline.

Figure 5.1 Requesting Airport Status is Posted to Twitter

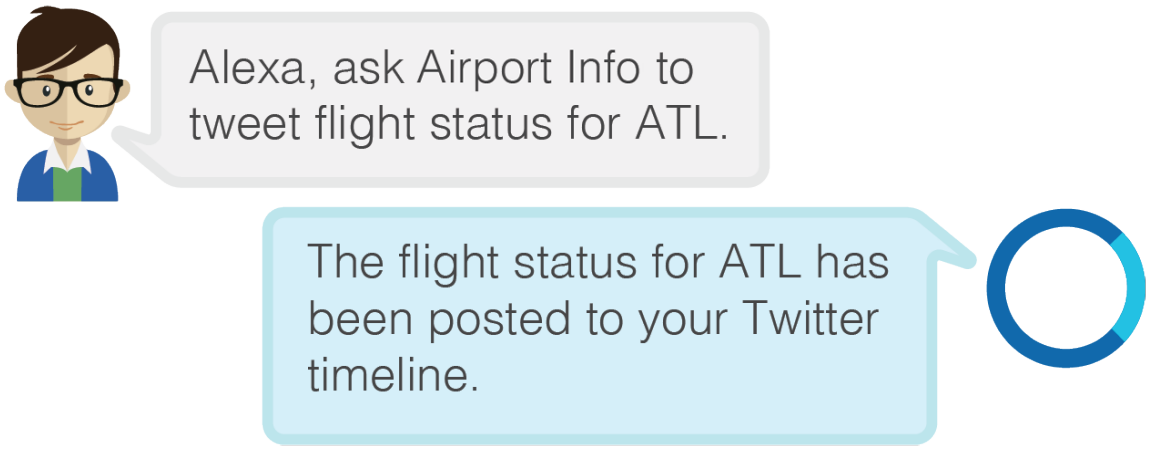
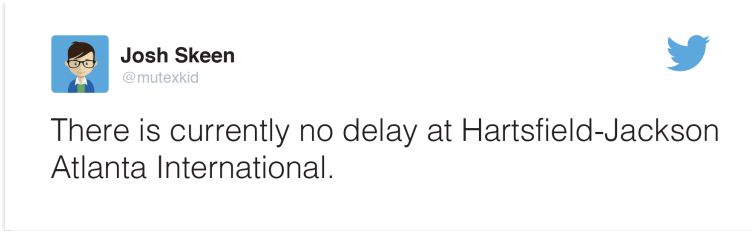


Figure 5.2



You will leverage the account linking feature of the Alexa Skills Kit platform to enable users to sign in to their personal Twitter account in a web browser and securely link their account with your skill. This action will share an access token from Twitter with the skill so that using the Twitter API from the skill is possible once authentication is complete. Upon initially enabling your skill in the in the skills section of the Alexa App, users will be prompted to authenticate with their Twitter credentials so that an access token may be generated and shared with the skill. In this chapter you will also take a look at OAuth web services, which enable services to share credentials with one antoher.

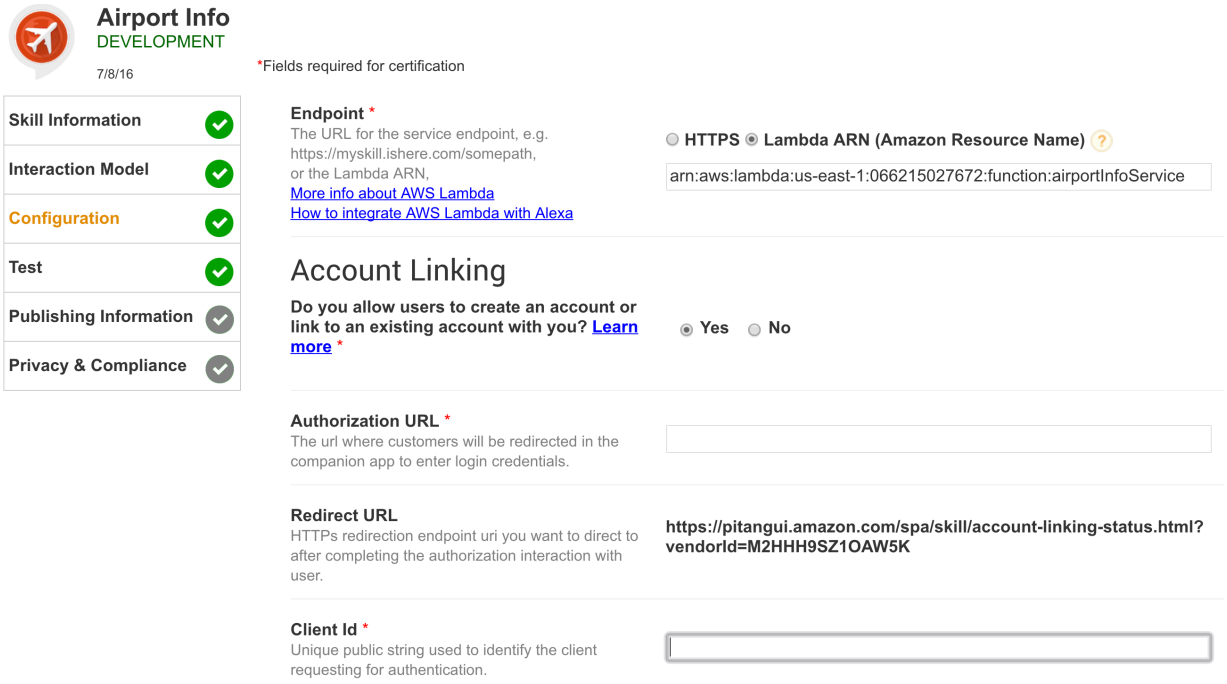
Skill Configuration

Begin by returning to the skill interface page for the Airport Info skill you configured earlier in the Amazon Developer Console, accessible from

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

Advance to the Configuration step in the skill interface for Airport Info and select "Yes" for Account Linking.

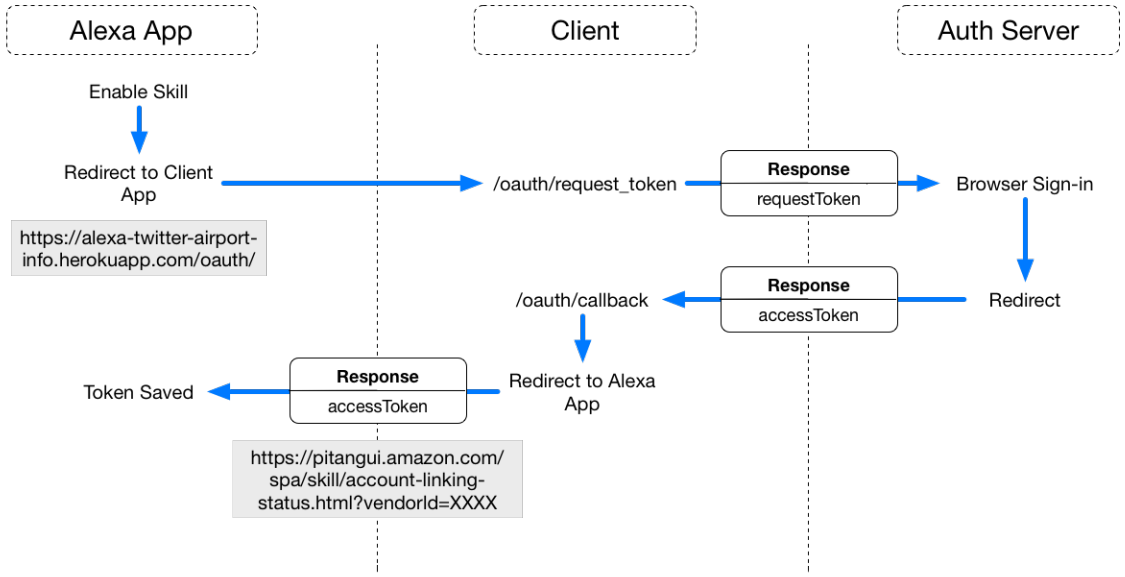
Figure 5.3 Enabling the Account Linking Option



Once you have made this selection, you will see additional fields for Authorization URL and Privacy Policy URL presented. You will return to complete these fields soon, but first you must configure the OAuth flow to work with the Twitter API you will be integrating with.

Understanding the OAuth Flow

Figure 5.4 OAuth flow diagram



The account linking feature will leverage an OAuth web flow within the Alexa App's card interface to allow a user to sign in using their Twitter credentials. This token grants the skill access to make posts to the Twitter timeline.

OAuth, an industry standard protocol for authentication, is widely used for authentication and securely sharing permissions between different user accounts on different servers. If you are new to OAuth, you can learn more about the protocol here:

```
http://oauth.net/
```

Prior to the workshop, a basic client OAuth integration for the Airport Info skill was written and deployed to the cloud. Because building the server code for the OAuth client leg of the authentication flow is beyond the scope of the course, you will use the already live implementation of the OAuth client needed to support the account linking behavior.

You may explore the source code for the client portion of the OAuth flow you will integrate with by downloading it here:

```
https://github.com/bignerdranch/alexa-airportinfo
```

The deployed OAuth client lives at `https://alexa-twitter-airport-info.herokuapp.com`, and handles the redirect to the login page on Twitter and fetching a requestToken. Upon logging in, users will be redirected to the OAuth client integration with an `accessToken`. Once the `accessToken` is received, the OAuth client integration redirects users to the skills section in the Alexa App page, where the token is saved for use in the skill.

The OAuth client is a small web application that hosts two endpoints which are necessary to complete the OAuth flow with the skill interface.

Table 5.1 OAuth Client Endpoints

Endpoint	Purpose
/oauth/request_token	generates a request token via the Twitter API and redirects to the Twitter account sign in page.
/oauth/callback	redirected to with access token, once visitors sign in via the Twitter account sign in page. Then, redirects to the Alexa App page, saving the token to the skill.

Understanding OAuth Versions

There are two primary versions of OAuth in use with OAuth-based authentication services today: 2.0, and 1.0a. For Account Linking, Amazon officially supports 2.0, the latest version of protocol. Some services, like Twitter, still make use the 1.0a version of the protocol. Integrating with either a 2.0 or 1.0a endpoint will be a similar process, with 1.0a requiring some additional work.

To integrate with an OAuth 2.0 endpoint, you provide the Authorization URL for the service in the Alexa Skill Developer Portal. Then, you specify whether the OAuth service you're integrating with uses Implicit Grants or Auth Code grants for returning the access token. If the OAuth 2.0 service you're integrating with requires the Auth Code grant type, you'll provide additional values, client secret and authorization code values, in the Alexa Skill Developer portal. Refer to the documentation for the service you are integrating with for these values to determine if the Auth Code grant type is required.

In the Twitter integration example, you will integrate with an OAuth 1.0a based endpoint (as Twitter requires it), so we need an OAuth client implementation of our own to make the integration work correctly. The OAuth client in the Twitter example is an additional web service. When you configure the OAuth 1.0a client for your skill, you provide the redirect URL, the twitter session key, and session secret values. This is an unneeded step for OAuth 2.0 endpoints, but required for the older OAuth 1.0a endpoints. An Oauth 2.0 integration will not require configuring the session key and secret values in the redirect URL, or including the ruby client to make the integration work. The following diagram shows the simplified authentication flow when working with an OAuth 2.0 based endpoint. Notice, it does not require a Oauth client to function correctly.

Figure 5.5 An example OAuth 2.0 Flow

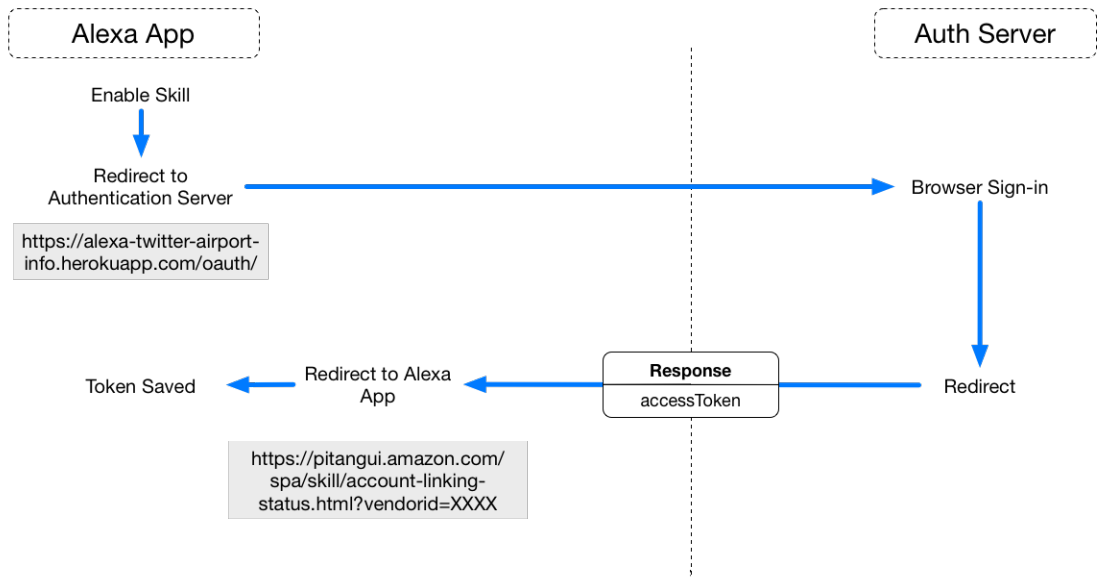
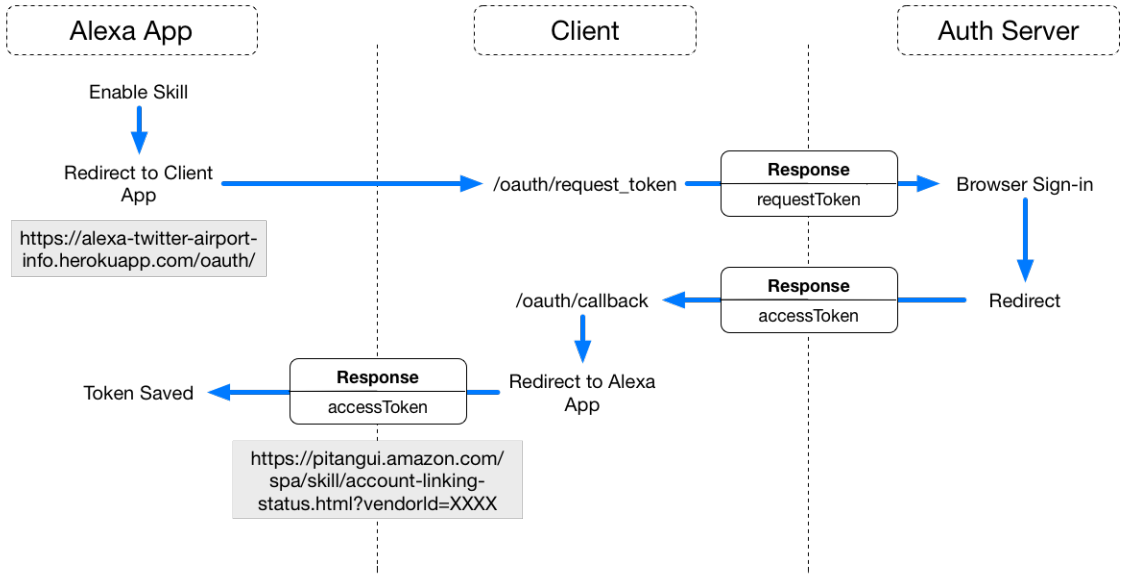


Figure 5.6 OAuth 1.0a flow diagram



If you compare the above diagrams you will notice that the OAuth 1.0a endpoint requires an additional "client" step to allow the generation of an authorization url and handle the redirect with token.

Registering a New Twitter Application

In order for the OAuth flow to work correctly against Twitter's server, you must register a new Twitter App to retrieve a consumer key and secret. If you do not have a Twitter account yet, you will require one to proceed — visit `https://twitter.com/signup?lang=en` and follow the signup process. To create a Twitter App, you also must associate a phone number with your Twitter account if you have not already done so. This can be found on `https://twitter.com/settings/devices`, once you have created an account and signed in.

Visit `https://apps.twitter.com/` and sign in with your Twitter account.

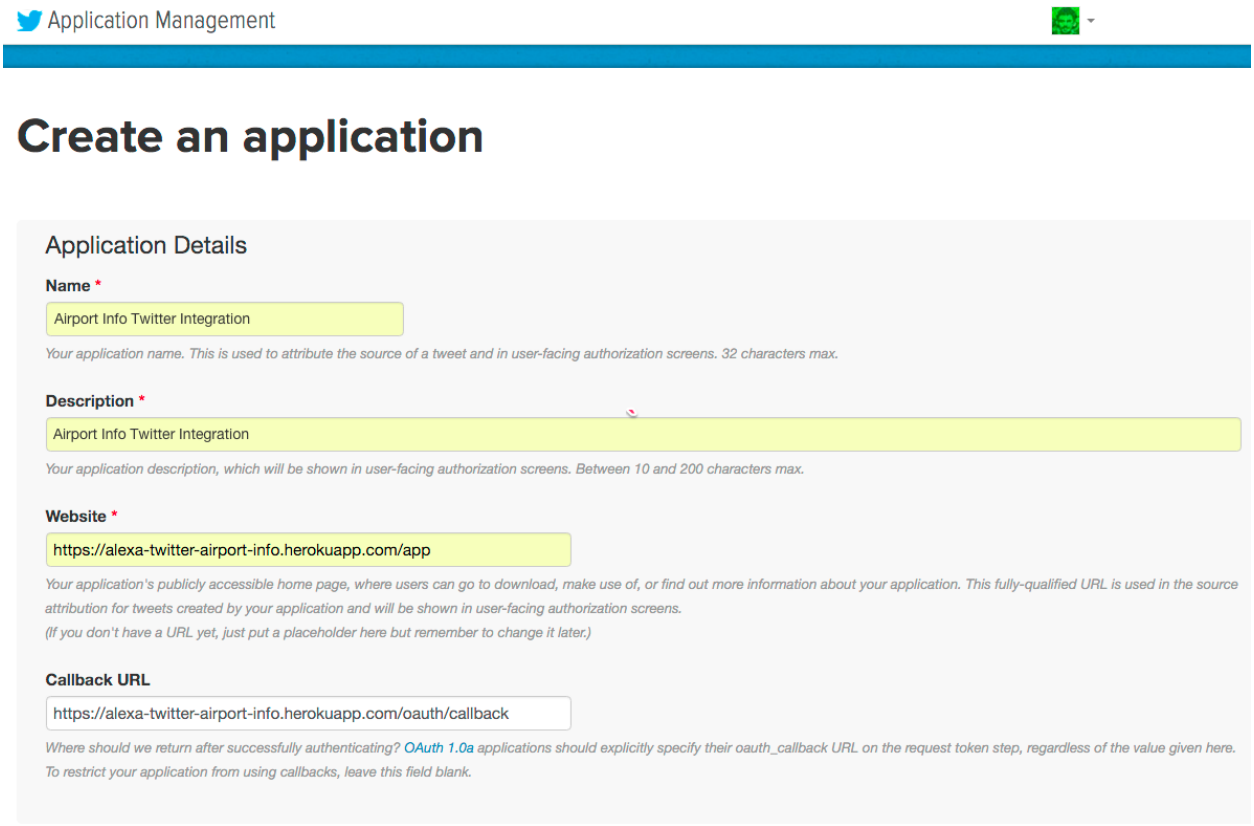
Figure 5.7 Twitter Apps

Click Create New App. On the subsequent page, enter "Airport Info Twitter App" for Name, "The Alexa skills Twitter integration for Airport Information" for Description, "`https://alexa-twitter-airport-info.herokuapp.com/app`" for Website, and enter

```
https://alexa-twitter-airport-info.herokuapp.com/oauth/callback
```

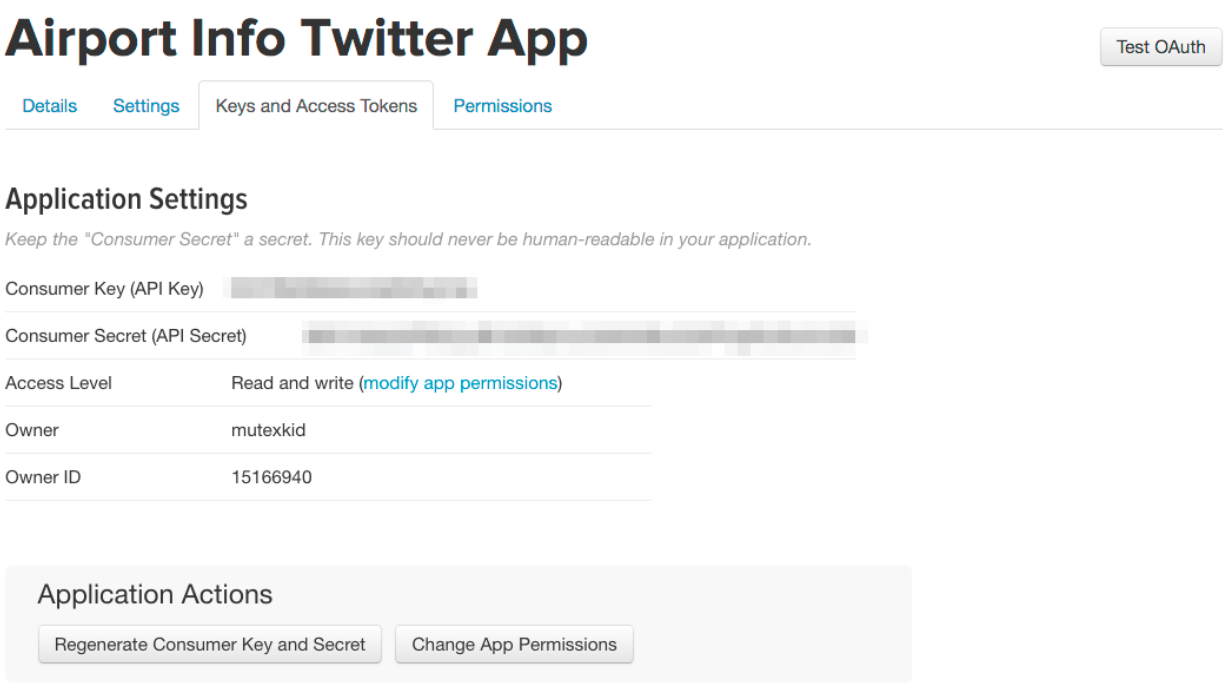
for the Callback URL. Click "Yes, I agree" to the "Developer Agreement", and click Create your Twitter Application.

Figure 5.8 Registering a new Twitter App



You will be redirected to the new Twitter App management page. Click on the Keys and Access Tokens tab, and copy the Consumer Key (API Key) and Consumer Secret (API Secret) values down, as they will be needed in the next step.

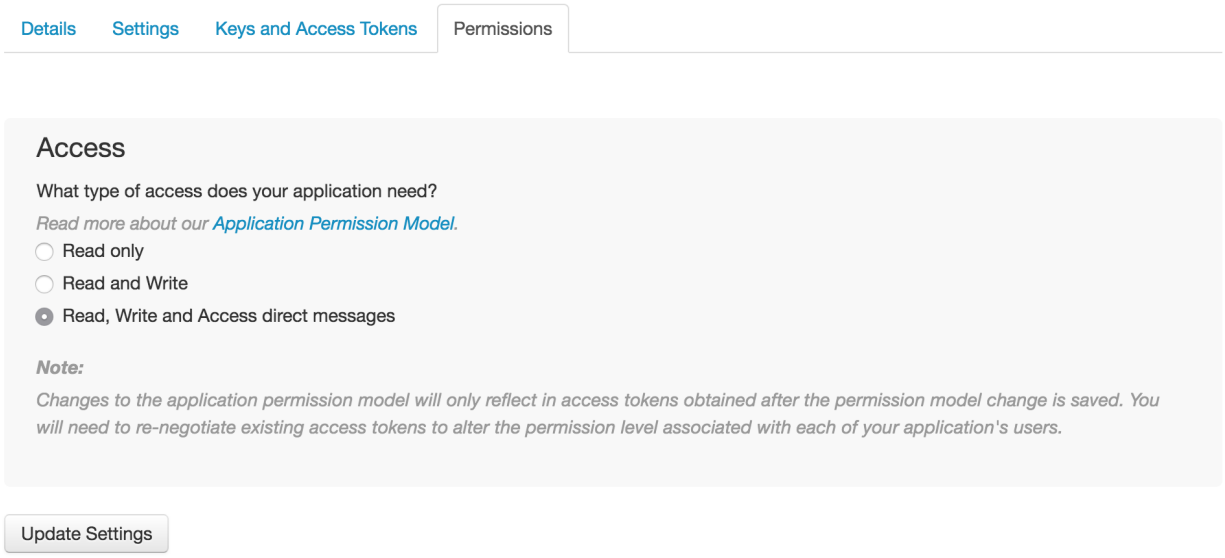
Figure 5.9 Copying the Application Consumer Key and Consumer Secret



Setting the Twitter App Permissions

For the Twitter app to successfully post to the timeline, the custom Twitter app permissions must be set to "Read, Write and Access direct messages". Click on the Permissions tab, and select Read, Write, and Access direct messages..

Figure 5.10 Setting the Twitter App Permissions



After you have done this, click Update Settings.

Adding the Authorization URL

Note the Redirect URL field listed on the skill interface configuration page - this address is where the OAuth flow will return the token that is needed to authenticate with Twitter. This is a unique URL that allows Amazon to associate a resulting `accessToken` with your user's skill.

Figure 5.11 The Skill Redirect URL

Note the `vendorId` attribute provided in the Redirect URL - this value will be passed to the OAuth client in order to successfully redirect upon completing the auth flow.

In a production OAuth client implementation, the `consumer_key`, `consumer_secret`, and `vendorId` values would be defined as environment variables on the server. In this learning example you will pass the values to the server as URL parameters instead, so that you are not required to deploy your own OAuth server. The OAuth client is hosted at

```
https://alexa-twitter-airport-info.herokuapp.com .
```

On the Configuration page, under Authorization URL, enter

```
https://alexa-twitter-airport-info.herokuapp.com/oauth/request_token?vendor_id=XXXXXX&consumer_key=YYYYYY
   &consumer_secret=ZZZZZZ
```

Replace "XXXXXXXX" with the value for "vendorId" found for the redirectURL. Replace "YYYYYYYY" and "ZZZZZZZZ" with the Consumer key and Consumer secret values you noted down on the Twitter App Management page.

For the Privacy Policy URL, enter `https://alexa-twitter-airport-info.herokuapp.com/policy`.

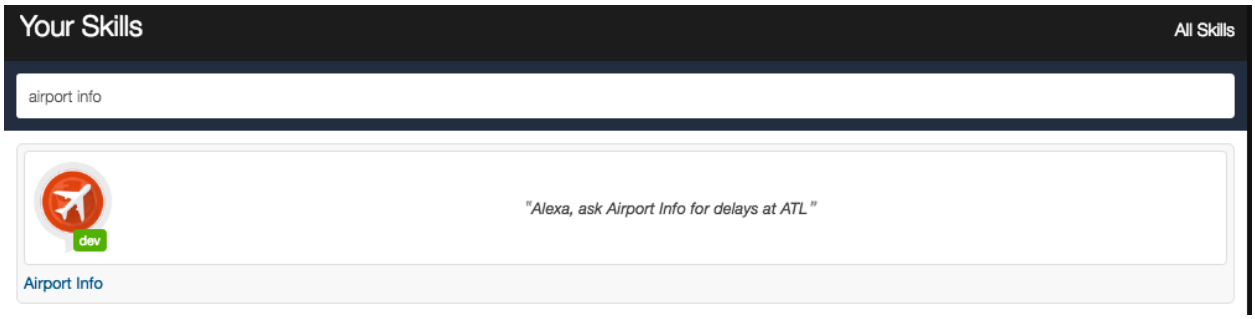
Testing the Account Linking Flow

You can now test that the account linking flow for the Airport Info skill works as expected. Visit

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

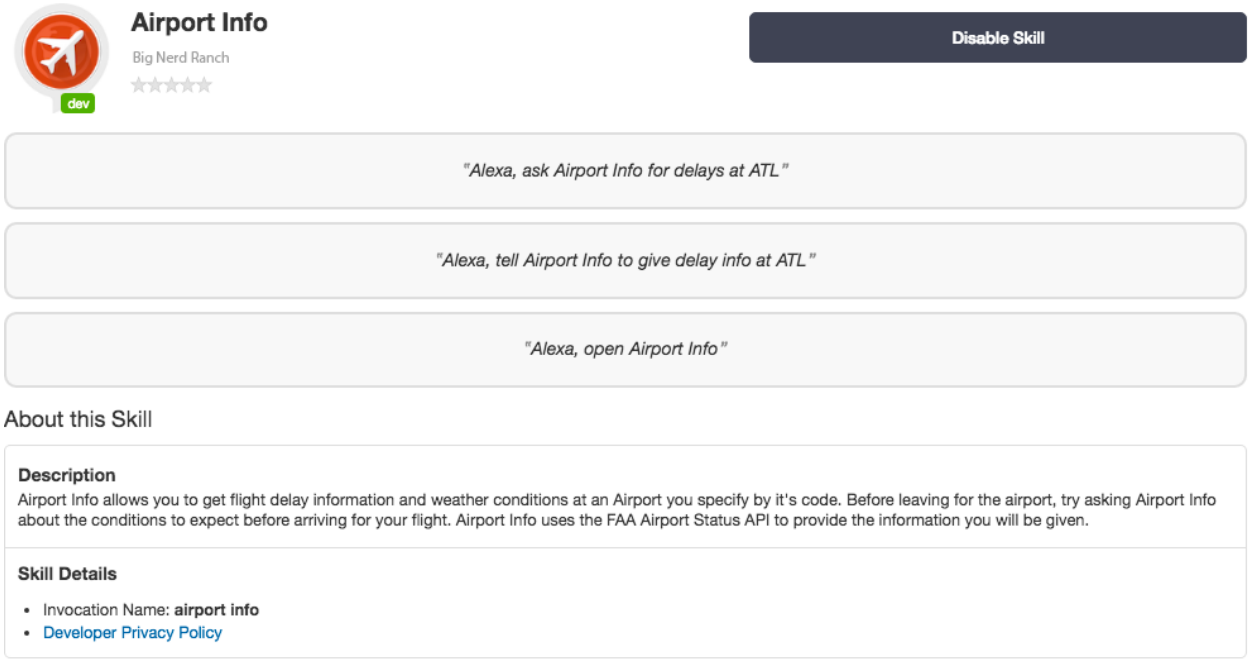
and search for Airport Info. Ensure you have clicked Your Skills at the top right of the interface.

Figure 5.12 Searching for the Airport Info Skill



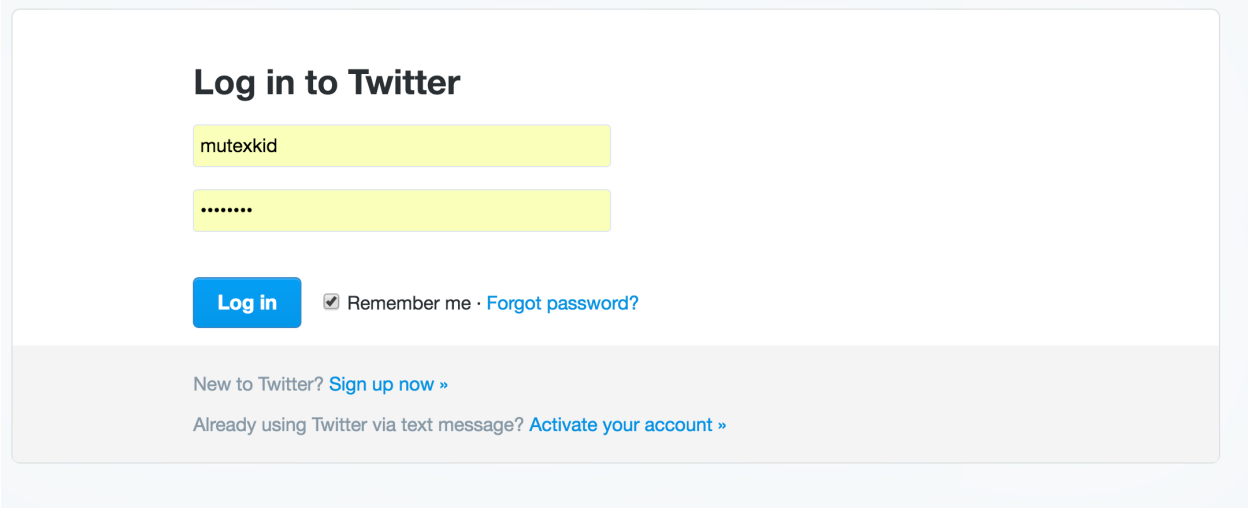
Once you have found the skill, click on it. The skill will currently be enabled, but you would like to test that account linking occurs upon an initial installation. Click Disable.

Figure 5.13 Disabling the Skill



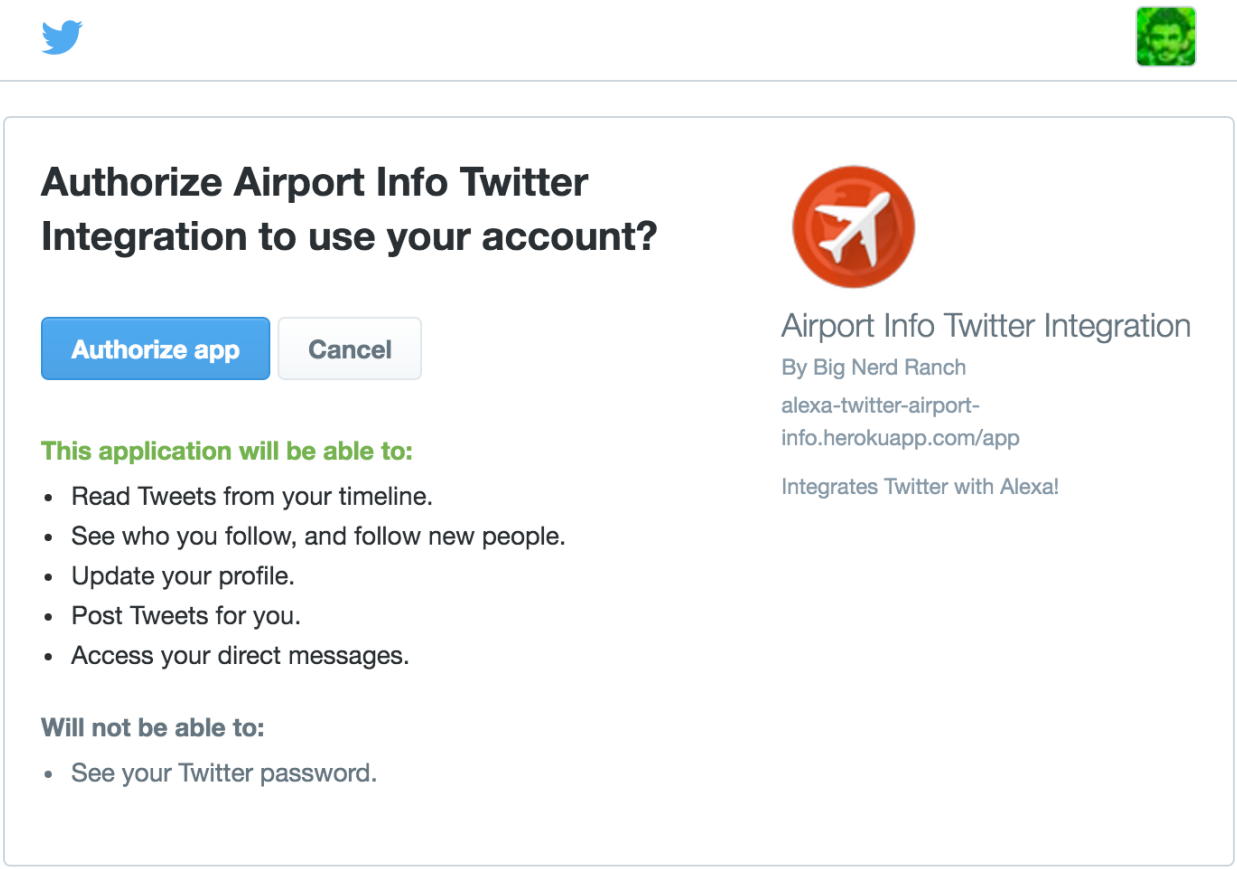
Now, Click Enable. Upon enabling the skill, the skill should redirect to the Twitter Sign In page.

Figure 5.14 Twitter Sign In



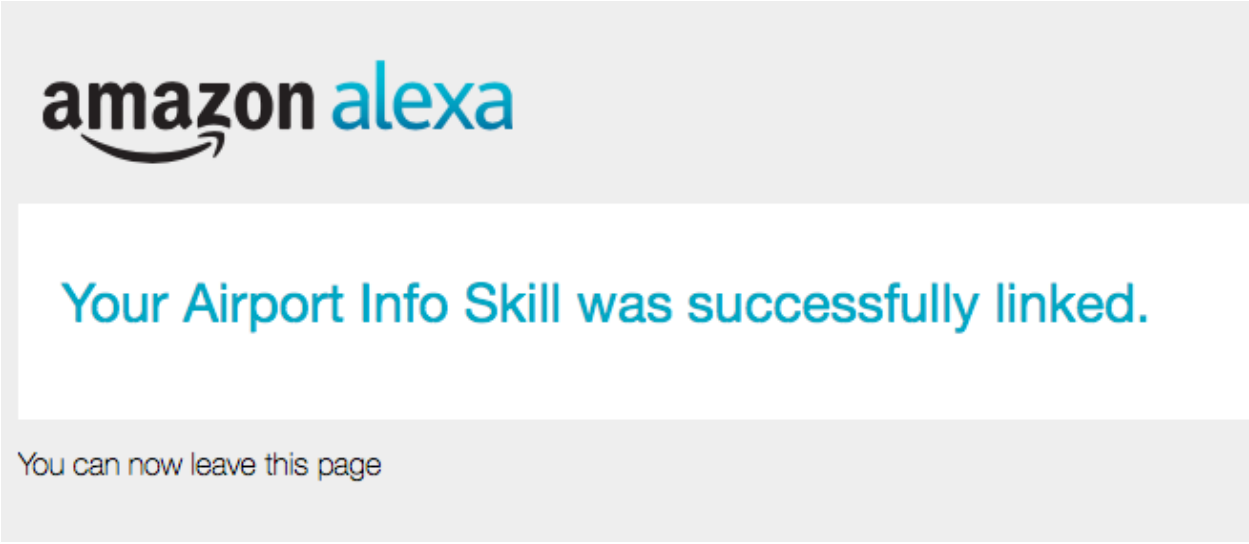
After logging in, you click "Authorize App" to accept the Airport Info Twitter Integration can use your account.

Figure 5.15 Authorizing Twitter



Click Authorize App. The browser should redirect to the skill page in the Alexa App, and display a success message.

Figure 5.16 Account Linking Success



Implementing a TweetAirportStatusIntent Handler

Now you can use the `accessToken` which has been associated with the skill since the account has been linked. You will use the `accessToken` to post a message to the user's Twitter account via the Twitter API. The twitter functionality will be an extension to the Airport Info skill you wrote earlier. Open the existing `index.js` file in the `airportinfo` directory you created during the Slots, Slot Types, and Utterances chapter. Add a new intent handler, called `TweetAirportStatusIntent` to the file. Add the new handler right below the launch intent handler. This handler will respond to the utterance "Alexa, ask airport info to tweet status for {AIRPORTCODE}".

Listing 5.1 Defining the tweetAirportStatusIntent Handler

```
skill.launch(function(request, response) {
  var prompt = 'For delay information, tell me an Airport code.';
  response.say(prompt).reprompt(prompt).shouldEndSession(false);
});
skill.intent('tweetAirportStatusIntent', {
'slots': {
  'AIRPORTCODE': 'FAACODES'
},
'utterances': ['tweet {|delay|status} {|info} {|for} {-|AIRPORTCODE}']
},
function(request, response) {
  var accessToken = request.sessionDetails.accessToken;
  if (accessToken === null) {
  //no token! display card and let user know they need to sign in
  } else {
  //has a token, post the tweet!
  }
});
```

Upon a user triggering the `tweetAirportStatusIntent` with their voice, the skill pulls the `request.sessionDetails.accessToken` the user has linked with the skill, if present. This is the OAuth value that was returned from the authentication flow.

If present, the skill should post a tweet of the status for the airport they requested. If not present, the skill should display a card instructing the user to log in with Twitter. For handling this situation, the Alexa Skills Kit offers a special card called a `LinkAccount` card. Next, update the case where no token is found with the following code:

Listing 5.2 Handling a Missing Token

```
skill.launch(function(request, response) {
  var prompt = 'For delay information, tell me an Airport code.';
  response.say(prompt).reprompt(prompt).shouldEndSession(false);
});

skill.intent('tweetAirportStatusIntent', {
'slots': {
  'AIRPORTCODE': 'FAACODES'
},
'utterances': ['tweet {|delay|status} {|info} {|for} {-|AIRPORTCODE}']
},
function(request, response) {
  var accessToken = request.sessionDetails.accessToken;
  if (accessToken === null) {
    response.linkAccount().shouldEndSession(true).say('Your Twitter account is not linked.
    Please use the Alexa app to link the account.');
    return true;
  } else {
    //has a token, post the tweet!
  }
});
```

The above code above handles a case where the skill currently finds no token for the user. The **`response.linkAccount()`** method displays a special card in the user's web browser within the Alexa App that guides them to login to link their Twitter account.

Implementing a TwitterHelper class

Before we can implement the logic to post the tweet, we will first define a new class for managing the Twitter-posting functionality. The new class will make use of an open source library, `twit`. The `twit` library integrates Twitter's REST API in a convenient Node.js library. To install `twit`, navigate to the `airportinfo` directory in the terminal, and run the following command:

Listing 5.3 Installing Twit

```
$ npm install twit --save
```

Now that `twit` is installed, create a new file called `twitter_helper.js` within the `airportinfo` directory. Add the following code to the file:

Listing 5.4 Building the TwitterHelper class

```
'use strict';
module.change_code = 1;
var _ = require('lodash');
var Twitter = require('twit');
var CONSUMER_KEY = 'XXXXX';
var CONSUMER_SECRET = 'XXXXX';
function TwitterHelper(accessToken) {
  this.accessToken = accessToken.split(',');
  this.client = new Twitter({
    consumer_key: CONSUMER_KEY,
    consumer_secret: CONSUMER_SECRET,
    access_token: this.accessToken[0],
    access_token_secret: this.accessToken[1]
  });
}

TwitterHelper.prototype.postTweet = function(message) {
  return this.client.post('statuses/update', {
    status: message
  }).catch(function(err) {
      console.log('caught error', err.stack);
    });
};
module.exports = TwitterHelper;
```

Replace `CONSUMER_KEY` and `CONSUMER_SECRET` values with the matching values you copied during the Twitter App creation step.

The **`TwitterHelper`** class accepts an access token, and implements the **`postTweet`** method, which will update the timeline associated with the token. The `twit` library also helpfully returns the results in the form of a `promise` object, because it is an asynchronous call.

Using the TwitterHelper

Open your `index.js` file. At the top of the file, and import the newly created **`TwitterHelper`**:

Listing 5.5 Posting the Tweet

```
'use strict';
module.change_code = 1;
var _ = require('lodash');
var Alexa = require('alexa-app');
var skill = new Alexa.app('airportinfo');
var FAADataHelper = require('./faa_data_helper');
var TwitterHelper = require('./twitter_helper');
...
```

Update the case where the token is found to retrieve the airport status and call the **`postTweet(status)`** method on a **`TwitterHelper`** instance.

Listing 5.6 Posting the Tweet

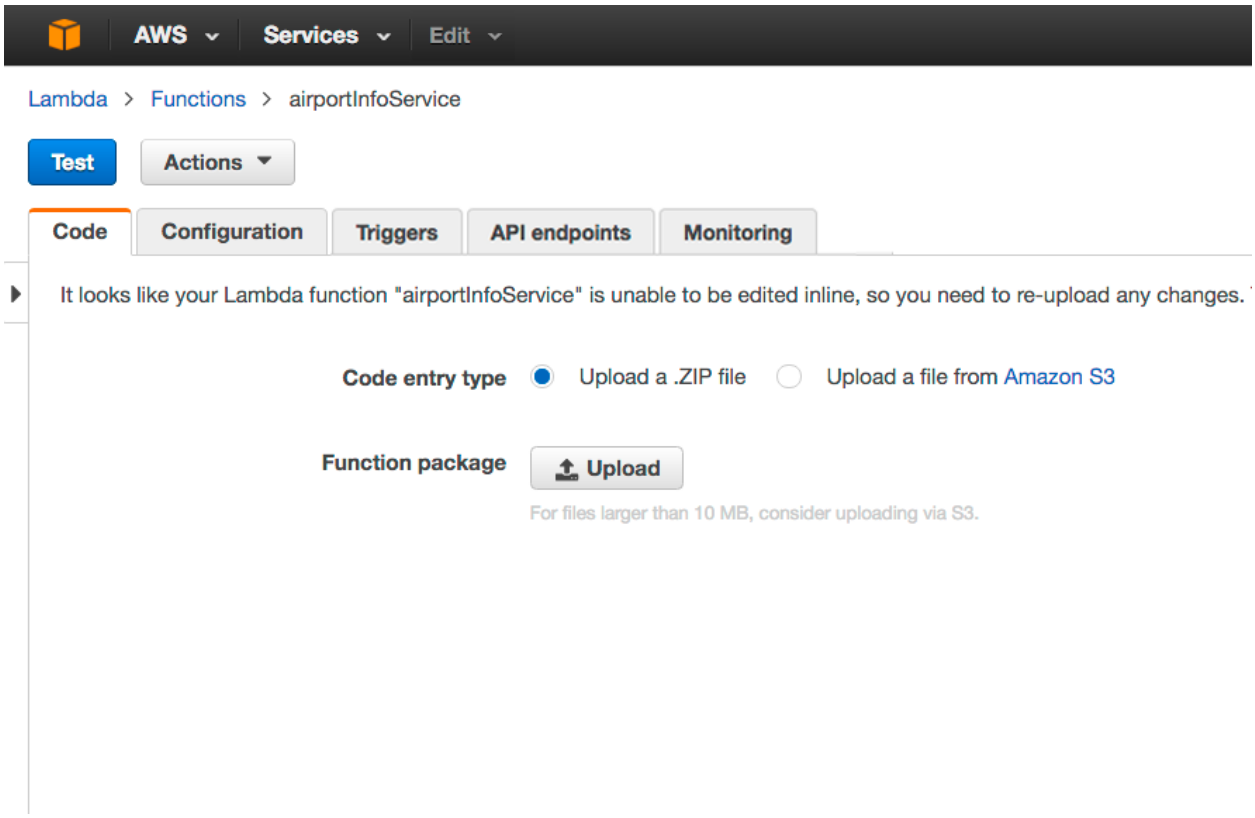
```
skill.launch(function(request, response) {
  var prompt = 'For delay information, tell me an Airport code.';
  response.say(prompt).reprompt(prompt).shouldEndSession(false);
});

skill.intent('tweetAirportStatusIntent', {
'slots': {
  'AIRPORTCODE': 'FAACODES'
},
'utterances': ['tweet {|delay|status} {|info} {|for} {-|AIRPORTCODE}']
},
function(request, response) {
  var accessToken = request.sessionDetails.accessToken;
  if (accessToken === null) {
    response.linkAccount().shouldEndSession(true).say('Your Twitter account is not linked.
    Please use the companion app to link the account.');
    return true;
  } else {
    var airportCode = request.slot('AIRPORTCODE');
    if (_.isEmpty(airportCode)) {
        //i've got a token! make the tweet
        var twitterHelper = new TwitterHelper(request.sessionDetails.accessToken);
        var faaHelper = new FAADataHelper();
        var prompt = 'i didn\'t have data for an airport code of ' + airportCode;
        response.say(prompt).send();
      } else {
        faaHelper.getAirportStatus(airportCode).then(function(airportStatus) {
          return faaHelper.formatAirportStatus(airportStatus);
        }).then(function(status) {
          return twitterHelper.postTweet(status);
        }).then(
          function(result) {
            response.say('I\'ve posted the status to your timeline').send();
          }
        );
        return false;
      }
  }
});
```

Updating the Skill Service

Now you will update the AWS Lambda function with your changes to the skill service. Create a new archive including the contents of the `airportinfo` directory. Visit the existing Airport Info Service Lambda function and click Upload, selecting the archive you created of the updated skill service source code.

Figure 5.17 Updating the Service



Updating the Skill Interface

You next update the intent schema and utterances for the AirportInfo Skill. Visit the skill interface in the Amazon Developer Console for the existing AirportInfo skill. Go to the Interaction Model page within the console. Update the Intent Schema section to include the `tweetAirportStatusIntent` intent:

Listing 5.7 Updating the Intent Schema

```
{
  "intents": [
  {
      "intent": "tweetAirportStatusIntent",
      "slots": [
        {
          "name": "AIRPORTCODE",
          "type": "FAACODES"
        }
      ]
    },
    {
      "intent": "airportInfoIntent",
      "slots": [
        {
          "name": "AIRPORTCODE",
          "type": "FAACODES"
        }
      ]
    }
  ]
}
```

Update the Sample Utterances to include the tweetAirportStatusIntent:

Listing 5.8 Updating the Sample Utterances

```
tweetAirportStatusIntent tweet {AIRPORTCODE}
tweetAirportStatusIntent tweet delay {AIRPORTCODE}
tweetAirportStatusIntent tweet status {AIRPORTCODE}
tweetAirportStatusIntent tweet info {AIRPORTCODE}
tweetAirportStatusIntent tweet delay info {AIRPORTCODE}
tweetAirportStatusIntent tweet status info {AIRPORTCODE}
tweetAirportStatusIntent tweet for {AIRPORTCODE}
tweetAirportStatusIntent tweet delay for {AIRPORTCODE}
tweetAirportStatusIntent tweet status for {AIRPORTCODE}
tweetAirportStatusIntent tweet info for {AIRPORTCODE}
tweetAirportStatusIntent tweet delay info for {AIRPORTCODE}
tweetAirportStatusIntent tweet status info for {AIRPORTCODE}
airportInfoIntent  {AIRPORTCODE}
airportInfoIntent flight {AIRPORTCODE}
airportInfoIntent airport {AIRPORTCODE}
airportInfoIntent  delay {AIRPORTCODE}
airportInfoIntent flight delay {AIRPORTCODE}
airportInfoIntent airport delay {AIRPORTCODE}
airportInfoIntent  status {AIRPORTCODE}
airportInfoIntent flight status {AIRPORTCODE}
airportInfoIntent airport status {AIRPORTCODE}
airportInfoIntent  info {AIRPORTCODE}
airportInfoIntent flight info {AIRPORTCODE}
airportInfoIntent airport info {AIRPORTCODE}
airportInfoIntent  delay info {AIRPORTCODE}
airportInfoIntent flight delay info {AIRPORTCODE}
airportInfoIntent airport delay info {AIRPORTCODE}
airportInfoIntent  status info {AIRPORTCODE}
airportInfoIntent flight status info {AIRPORTCODE}
airportInfoIntent airport status info {AIRPORTCODE}
airportInfoIntent  for {AIRPORTCODE}
airportInfoIntent flight for {AIRPORTCODE}
airportInfoIntent airport for {AIRPORTCODE}
airportInfoIntent  delay for {AIRPORTCODE}
airportInfoIntent flight delay for {AIRPORTCODE}
airportInfoIntent airport delay for {AIRPORTCODE}
airportInfoIntent  status for {AIRPORTCODE}
airportInfoIntent flight status for {AIRPORTCODE}
airportInfoIntent airport status for {AIRPORTCODE}
airportInfoIntent  info for {AIRPORTCODE}
airportInfoIntent flight info for {AIRPORTCODE}
airportInfoIntent airport info for {AIRPORTCODE}
airportInfoIntent  delay info for {AIRPORTCODE}
airportInfoIntent flight delay info for {AIRPORTCODE}
airportInfoIntent airport delay info for {AIRPORTCODE}
airportInfoIntent  status info for {AIRPORTCODE}
airportInfoIntent flight status info for {AIRPORTCODE}
airportInfoIntent airport status info for {AIRPORTCODE}
```

Once you have updated the Intent Schema and Sample Utterances fields, click Save.

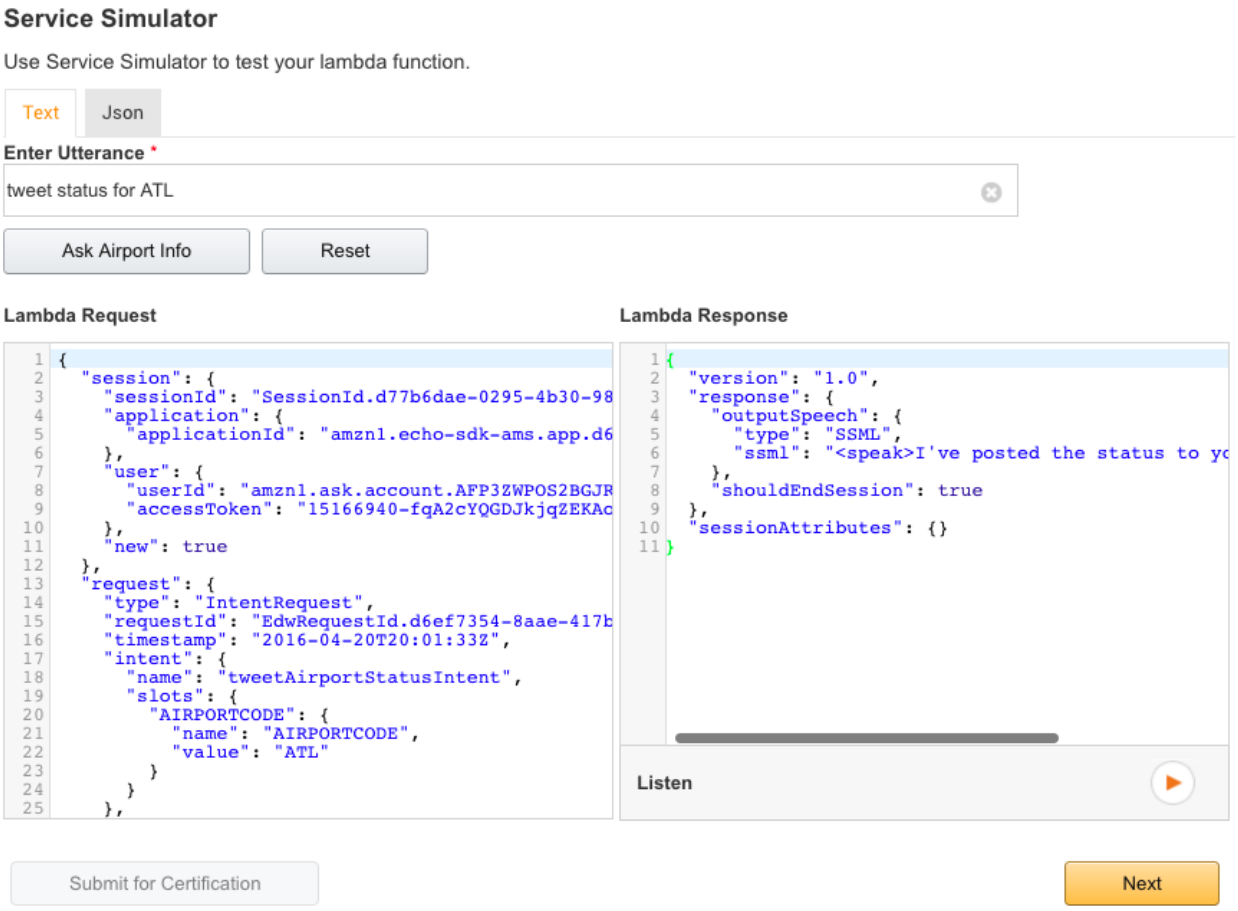
Testing the TweetAirportStatus Intent

To test the new `TweetAirportStatus` intent handler, visit the skill interface in the Amazon Developer Console and go to the Test page for the AirportInfo skill. Under the Service Simulator section, enter "tweet status for ATL" and click Ask Airport Info. Within the Lambda Response area, you should see the following response:

Listing 5.9 Posting the Tweet

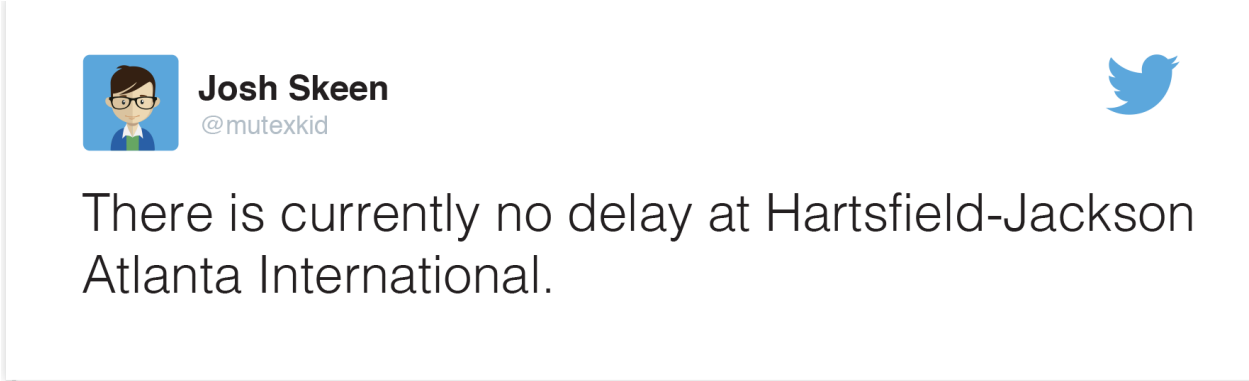
```
{
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speak>I've posted the status to your timeline</speak>"
    },
    "shouldEndSession": true
  },
  "sessionAttributes": {}
}
```

Figure 5.18 Testing the Twitter Integration via Service Simulator



Now, visit your Twitter timeline. You should see a new tweet indicating the status of the airport you specified.

Figure 5.19 Airport Status Tweet



Congratulations! You have successfully implemented account linking to integrate an external user account with your skill.

For more information and use cases with the account linking feature, check out:

`https://goo.gl/7xlGwa`

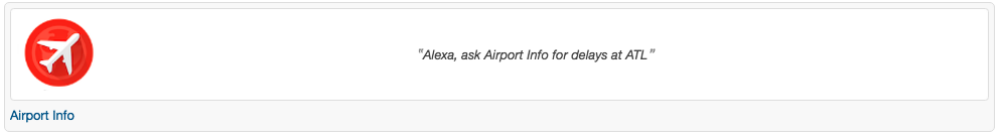
6

Certification, Testing, and SSML

Skill Approval and the Submission Process

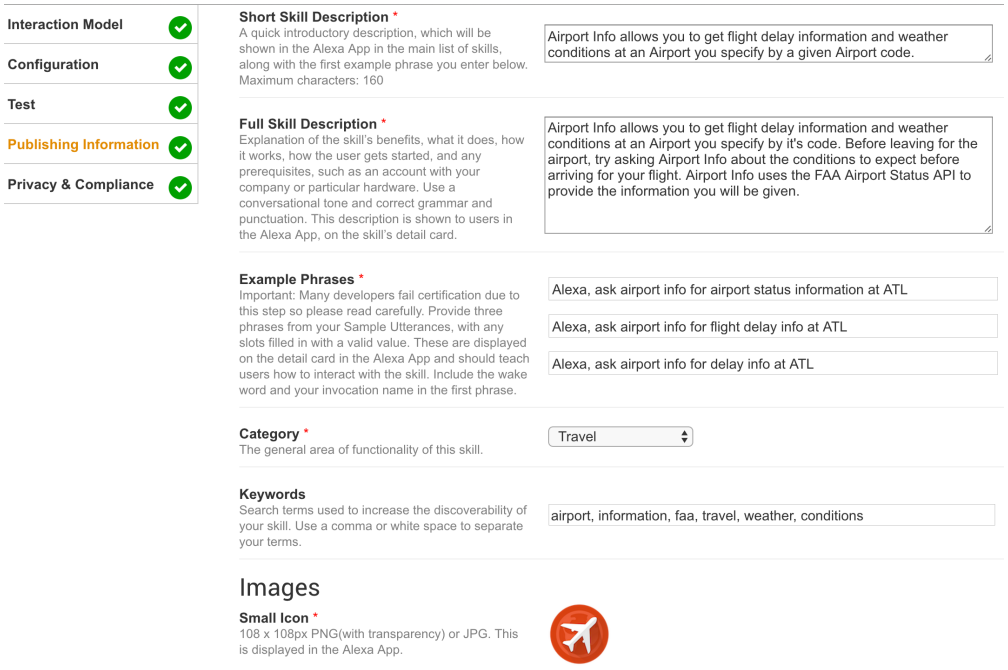
Completing the Skill Submission Process allows your skill to be used on any alexa-enabled device after it has passed Certification.

Figure 6.1 An Approved Skill in the Alexa App Skills Tab



When publishing a skill, the Publishing Information and Privacy & Compliance steps in the skill interface must be fully completed to move forward with the skill approval process. The information configured on the Publishing Information screen will be shown on the installation card for the skill in the alexa app skills tab and used in aiding users when searching for your skill.

Figure 6.2 The Publishing Information Screen

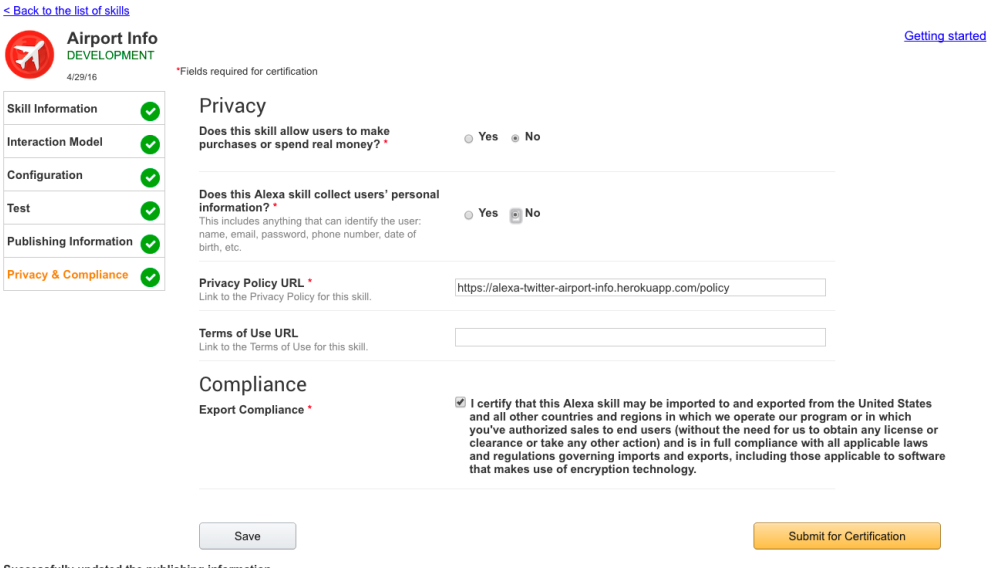


You should provide an icon, category, keywords, and short and full description for your skill. Also, three example phrases should be provided that show users how to interact with the skill. The example phrases should substitute any slots used in the intent with appropriate values. The example should also include the wake word and invocation name.

It is important that the example phrase be based upon an actual example utterance that can be found in the skill's interaction model. If it is not, Amazon will reject your skill during the approval process.

Next, the privacy and compliance details need to be provided to begin the approval process. Here you indicate whether the skill allows purchases or collects personal data. You also must provide a link to the privacy policy for your skill. A sample privacy policy can be found at `https://www.bbb.org/dallas/for-businesses/bbb-sample-privacy-policy1/` for an idea of how to create one.

Figure 6.3 Privacy and Compliance Screen



Once you have completed these fields, click the Submit for Certification button. Amazon responds within 5 - 7 days about the status of the approval process. Amazon will indicate whether the skill is approved or rejected. If rejected, Amazon will explain for what reasons so that you can correct the issues and resubmit.

Approval Guidelines and Common Rejection Reasons

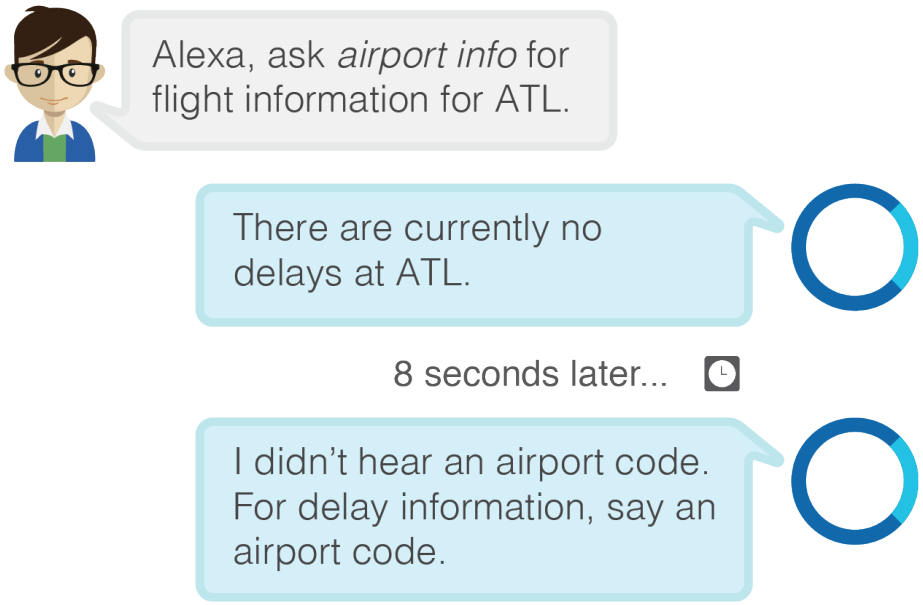
With the Amazon skill approval process, there are guidelines that should be kept in mind when developing a skill. A skill should contain no profanity or obscene content. A skill cannot target children under the age of 13 as an audience. A skill must also adhere to the voice user experience guidelines that Amazon requires a skill implement.

Figure 6.4 A Skill Must Follow Compliance Guidelines to be Approved

A common mistake is to not handle the required built-in intents. These required built-in intents are the `AMAZON.HelpIntent`, `AMAZON.CancelIntent`, and `AMAZON.StopIntent`. If you do not handle these in the skill service, the skill will be rejected because the skill will be difficult to use intuitively.

Recall that a built-in intent must be specified in the intent schema, as with any intent. The term built-in refers to the fact that the example utterances that resolve the user's spoken words to the intent and name of intent are pre-defined by Amazon. Built-in does not mean that you automatically gain certain intent handlers or behavior - you must implement handlers for these intents as with any other intent.

Figure 6.5 An Example of Incorrect Stream Management



Another reason for a skill being rejected is due to incorrect "stream management". Stream management is the act of closing or continuing the interaction with a user. The stream management for a skill must be done in a way that is considered intuitive from a user's perspective. In the example above, re-prompting after answering a user's question would be considered incorrect stream management and would result in a rejected submission.

A skill could also be rejected if it is hosted on a server other than AWS Lambda and the platform does not meet security requirements. The platform must implement SSL, respond on port 443, and the certificate used to sign requests must be from an Amazon-approved authority. For the list of Amazon-approved authorities, see `https://wiki.mozilla.org/CA:IncludedCAs`.

Testing

Listing 6.1 Mocha and Chai Tests for the FAADataHelper

```
→  faa-info ✗ moc
  FAADataHelper
    #getAirportStatus
      with an invalid airport code
        ✓ returns invalid airport code (314ms)
      with a valid airport code
success - received airport info for SFO
        ✓ returns airport code (276ms)
    #formatAirportStatus
      with a status containing no delay
        ✓ formats the status as expected
      with a status containing a delay
        ✓ formats the status as expecte
```

You may have wondered how to go about building unit tests to ensure your skill service works correctly as your skill grows more complex. For larger skills, a test suite will be an invaluable aid to producing code that behaves predictably and is maintainable. On Node.js, there are many solutions for building such tests. For the following examples you will be seeing the `mocha` and `chai` test framework and assertion libraries. `Mocha` and `chai` are popularly used for writing unit tests for Node.js. If you have ever done Ruby on Rails or Java development, `mocha` and `chai` are very similar to `RSpec` or `jUnit` and `AssertJ`. If you haven't done Ruby on Rails or Java development, don't worry - the `mocha` and `chai` style are very "plain english" and easy to adopt.

Figure 6.6 Mocha and Chai are Popular Test Libraries for Node.js Development

As an example, you will see how to write unit tests for **`FAADataHelper`**. First, you install `mocha`, `chai`, and an extension to `chai` that lets you easily test asynchronous methods. You install the test libraries in your project directory, in this case, `/faa-info`.

Listing 6.2 Installing the Test Libraries

```
$ npm install --save mocha chai chai-as-promised
```

All of the tests should be added within a new folder in your project called `/test`, which you create. Within `/test` you typically will add a new file to match each of the classes you would like to test. The example will show testing **`FAADataHelper`**, so you will add a new file called `test_faa_data_helper.js`.

Next, you import the libraries that are needed to write the tests - the class under test, and the `chai` assertion library. You also declare the name of the test, using the method **`describe`**. Typically, you will add a new description for each method you want.

Listing 6.3 Setting up the Test

```
'use strict';
var chai = require('chai');
var expect = chai.expect;
var FAADataHelper = require('../faa_data_helper');
describe('#getAirportStatus', function() {
//test goes here!
});
```

Listing 6.4 Setting up the Test

```
'use strict';
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');
chai.use(chaiAsPromised);

var expect = chai.expect;
var FAADataHelper = require('../faa_data_helper');
describe('FAADataHelper', function() {
    var subject = new FAADataHelper();
    var airport_code;
    //tests go here!
});
```

Now that you have described the name of the test (the class you want to test), you next describe the methods that should be tested. These should be all of the publicly visible methods a class offers. The first method, **`getAirportStatus(airportCode)`** should be tested. There are two situations - an invalid `airportCode` is passed to the method, or a valid one is passed to the method. These are referred to as "contexts", because they define the situation or context a method or group of methods have in common. Add a description for the `getAirportStatus` method test and the two contexts:

Listing 6.5 Testing the getAirportStatus Method

```
'use strict';
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');
chai.use(chaiAsPromised);

var expect = chai.expect;
var FAADataHelper = require('../faa_data_helper');

describe('FAADataHelper', function() {
    var subject = new FAADataHelper();
    var airport_code;
    describe('#getAirportStatus', function() {
        context('with an invalid airport code', function() {
            //assertions go here
        });
        context('with a valid airport code', function() {
            //assertions go here
        });
    });
});
```

Now that you have defined the contexts for the **`getAirportStatus(airportCode)`** method, you can make assertions about what you can expect the method to do. With an invalid `airportCode`, the method should raise an error. With a valid airport code, the result from the FAA server should be returned, including a matching `IATA` code. You can assert that it raises an error and returns the expected code using the following `chai` syntax:

Listing 6.6 Testing the getAirportStatus Method

```
'use strict';
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');
chai.use(chaiAsPromised);

var expect = chai.expect;
var FAADataHelper = require('../faa_data_helper');

describe('FAADataHelper', function() {
    var subject = new FAADataHelper();
    var airport_code;
    describe('#getAirportStatus', function() {
        context('with an invalid airport code', function() {
        it('returns invalid airport code', function() {
              airport_code = 'PUNKYBREWSTER';
              return expect(subject.requestAirportStatus(airport_code)).to.be.rejectedWith(Error);
            });
        });
        context('with a valid airport code', function() {
        it('returns airport code', function() {
            airport_code = 'SFO';
            var value = subject.requestAirportStatus(airport_code).then(function(obj) {
              return obj.IATA;
            });
            return expect(value).to.eventually.eq(airport_code);
          });
        });
    });
});
```

Here you check that calling the method with particular arguments results in the behavior you expected. You can now run the test by calling `mocha` from the command line within the `faa-info` directory:

Listing 6.7 Running the Test

```
$  mocha

  FaaDataHelper
    #getAirportStatus
      with an invalid airport code
        ✓ returns invalid airport code (1276ms)
      with a valid airport code
success - received airport info for SFO
        ✓ returns airport code (252ms)

  2 passing (2s)
```

Notice the test output indicates that the assertions match the behavior of the class. To see the rest of the **`FAADataHelper`** test, check out the test directory in

```
https://github.com/bignerdranch/alexa-airportinfo
```

To learn more about testing a skill with `mocha` and `chai`, visit

```
https://www.bignerdranch.com/blog/developing-alexa-skills-locally-with-nodejs-setting-up-your-local-environment/.
```

SSML

`SSML`, or *Speech Synthesis Markup Language*, is a markup syntax that can be used to customize how Alexa will pronounce sounds. `SSML` also supports requesting mp3 audio files to be played by Alexa. When you build an SSML-formatted `outputSpeech` command for Alexa, you configure the response from the skill service to include the following:

Listing 6.8 An SSML Formatted Speech Command

```
"outputspeech": {
    "type": "ssml",
    "ssml": <speak> you say <phoneme alphabet="ipa" ph="təˈmeɪtəʊ">tomato</phoneme>,
    i say <phoneme alphabet="ipa" ph="təˈmɑtəʊ">tomato</phoneme>! </speak>
}
```

The `outputSpeech` element should include a `type` attribute of `SSML`, and an `ssml` element with the additional `SSML` tags you would like to embed. The first step in using the SSML features.

`SSML` extends Alexa to support a number of additional capabilities. First, an `SSML` `outputSpeech` element's `ssml` attribute can contain <audio> elements that point to an mp3 file. Mp3 is supported only, and must be encoded to 16k bitrate resolution.

Listing 6.9 Playing an audio file with SSML

```
"outputSpeech": {
    "type": "SSML",
    "ssml": "<speak>
    <audio src='https://s3.amazonaws.com/ask-storage/tidePooler/OceanWaves.mp3'/>
    </speak>"
}
```

Additionally, the mp3 file must be hosted on an https endpoint. To encode a file to the correct format, you can use `ffmpeg` to correctly transcode the file. If you have **`brew`**, you may install `ffmpeg` via the following command :

Listing 6.10 Installing ffmpeg

```
brew install ffmpeg
```

Use the following invocation to encode a file to an mp3 file Alexa Skills Kit will support the playback of:

Listing 6.11 Encoding an audio file to an ASK supported format

```
  ffmpeg -y -i name_of_file.wav -ar 16000 -ab 48k -codec:a libmp3lame -ac 1 name_of_file.mp3
```

Another useful element when writing an SSML-flavored outputSpeech is the `<phoneme>` element. The phoneme element accepts a `ph` element which can be a series of either `ipa` (international phoenetic alphabet) or `x-sampa` (The Extended Speech Assessment Methods Phonetic Alphabet) notation characters. For example:

Listing 6.12 An SSML Formatted Speech Command

```
"outputSpeech": {
    "type": "SSML",
    "ssml": <speak> you say <phoneme alphabet="ipa" ph="təˈmeɪtəʊ">tomato</phoneme>,
    i say <phoneme alphabet="ipa" ph="təˈmɑtəʊ">tomato</phoneme>! </speak>
}
```

The above listing would pronounce tomato in subtly different ways. Using phonemes allows you to fine tune pronunciation of words. For further reading on IPA and X-Sampa please visit :

```
  https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit/docs/
  speech-synthesis-markup-language-ssml-reference
```

Conclusion

Now that you have completed the course you've gained a solid understanding of the capabilities of the Alexa Skills Kit platform. You're well on your way to building your own great skills for Alexa-enabled devices everywhere!

Should you ship a skill to the Alexa app skills tab, the authors of this course would love to hear about it! Contact them at developingalexaskills@gmail.com. We can't wait to see what you build!

Silver Challenge: Testing Airport Info

To take the next step in understanding unit testing an alexa skill, explore the `test/` directory within the Chapter 6 solutions repository found here:

```
https://github.com/bignerdranch/amazon-alexa-course/tree/master/exercises/faa-info
```

Using what you have learned in the chapter, run the test suite for the project. Are there additional tests you could write to expand the test coverage for Airport Info?

Gold Challenge: Playing a Sound using SSML

Encode, upload, and play back an mp3 as a SSML `<audio>` message to Alexa. This "1-up" sound can be used if you are looking for a free sound to use:

`http://themushroomkingdom.net/sounds/wav/smb/smb_1-up.wav`.

It will need to be encoded with ffmpeg using the encoding settings from the chapter.

Hints: For https storage for the file, use an AWS S3 bucket. To format an SSML message from alexa-app, you can use the SSML node library:

`https://www.npmjs.com/package/ssml`.

For an extra hint, you can take a look at the SSML example in the chapter 6 solutions repository:

`https://github.com/bignerdranch/developing-alexa-skills-solutions/tree/master/6_certificationTesting/beatmaster`