Table of Contents

1. Persistence 1
- Getting Started 2
- Creating a DatabaseHelper 3
- Creating the madlibsData Table 4
- Adding Store/Load Methods to the Helper 4
- Creating the Development Database Table 5
- Adding a Save Intent 5
- Refactoring the madlibIntent Handler 7
- Adding a Load Intent 9
- Testing the Save and Load Handlers 10
- Deployment 13
- Updating the Skill Interface Intent Schema and Utterances 17
- Testing the Skill in the Service Simulator 18
- Challenge: Implicit Saves 20

1
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 1.1

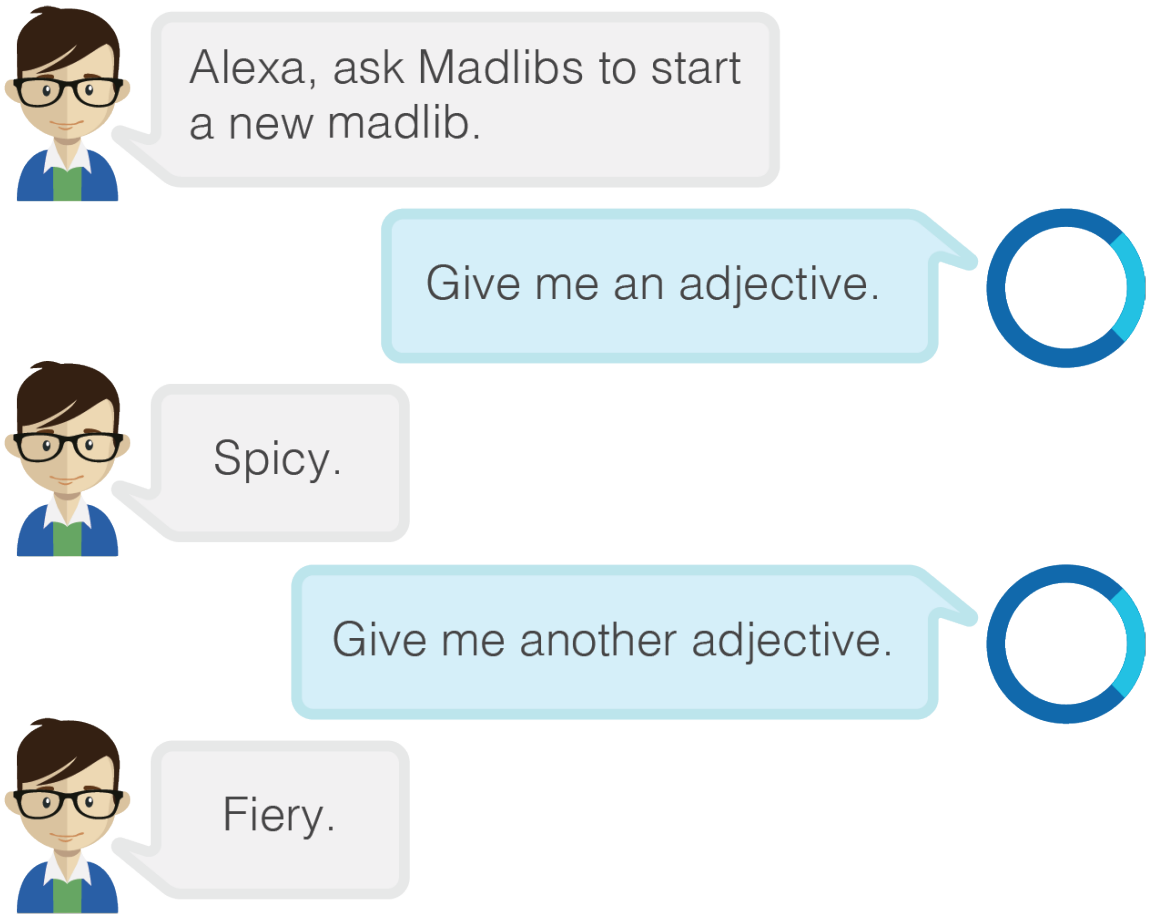
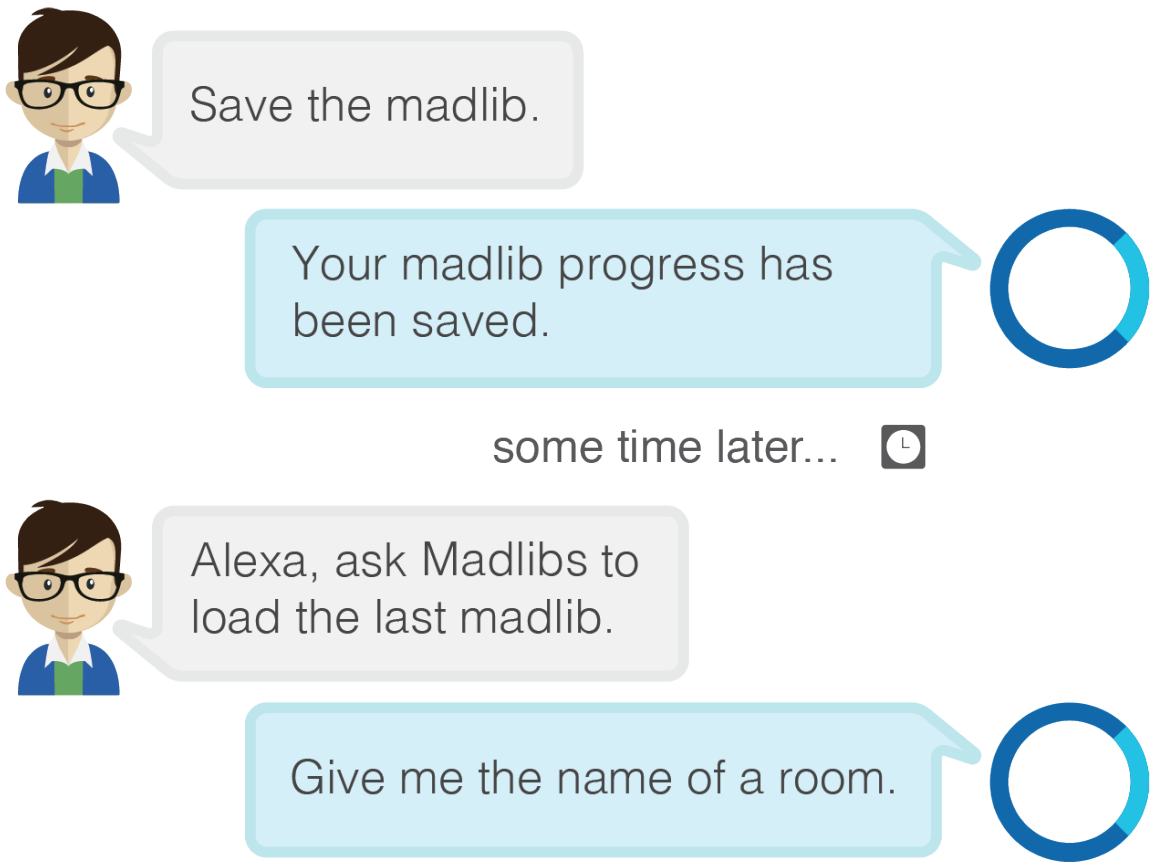


Figure 1.2



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 1.1 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 1.2 Installing dynamodb locally

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

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

Listing 1.3 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 1.4 Installing Dependencies

```
$ npm install --save dynasty
```

`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 1.5 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 1.6 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 1.7 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 1.8 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 1.9 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 1.10 Adding the pre Method

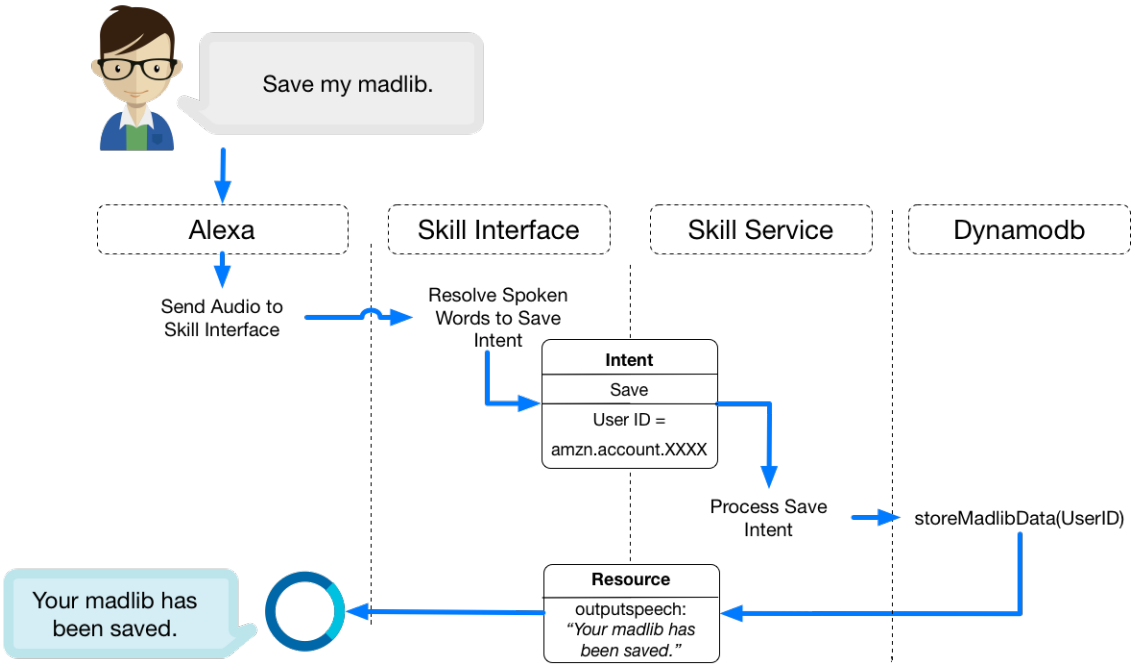
```
...
var DatabaseHelper = require('./database_helper');
skillService.pre = function(request, response, type) {
  databaseHelper.createMadlibsTable();
};
```

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 1.3 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 1.11 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 1.12 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 1.13 Modifying the madlibIntent Handler

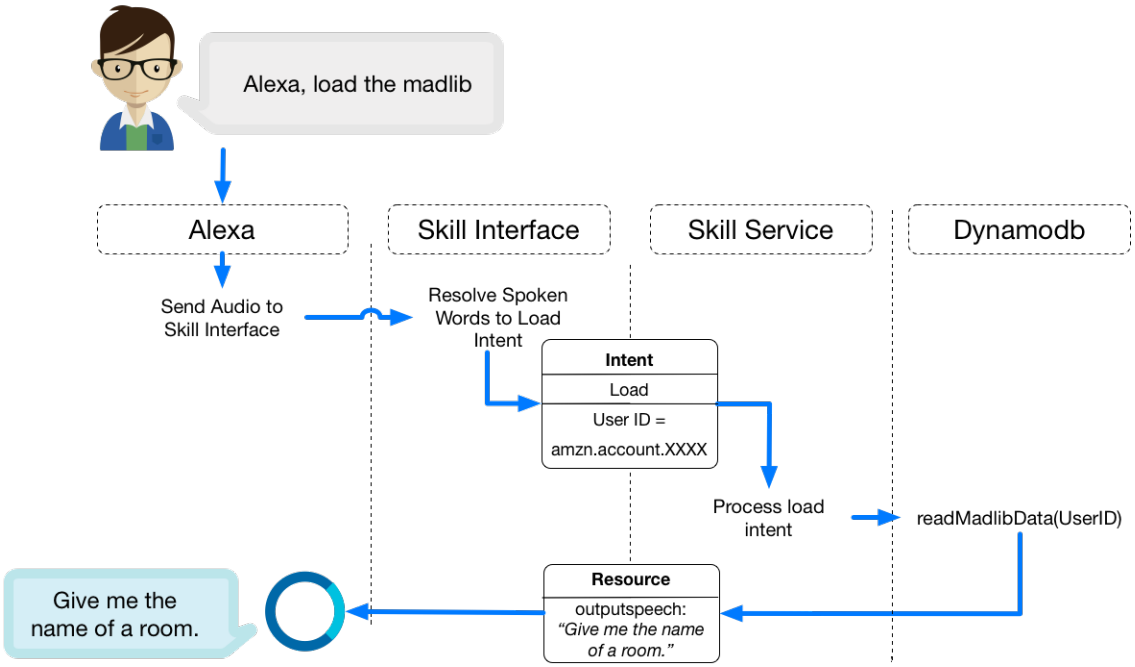
```
skillService.intent('madlibIntent', {
    'slots': {
      'STEPVALUE': 'STEPVALUES'
    },
    'utterances': ['{new|start|create|begin|build} {|a|the} madlib',
      '{-|STEPVALUE}'
    ]
  },
  function(request, response) {
    //check to see if a madlibbuilder exists in the request.
    var stepValue = 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 1.4 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 1.14 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 1.15 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 1.5 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": []
    }
  ]
}
```

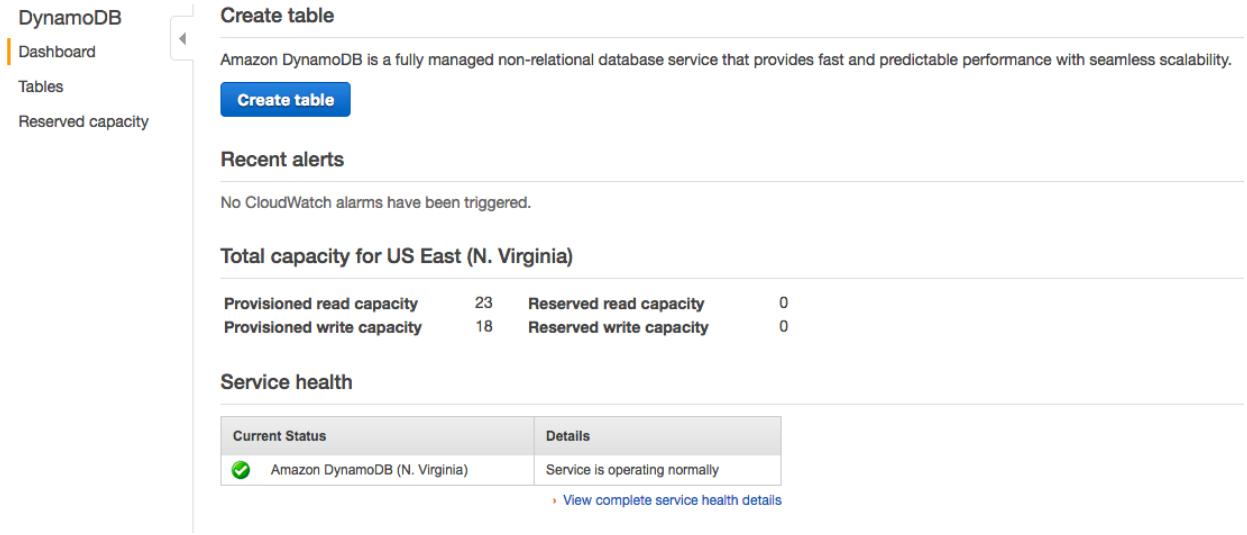
Utterances

```
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

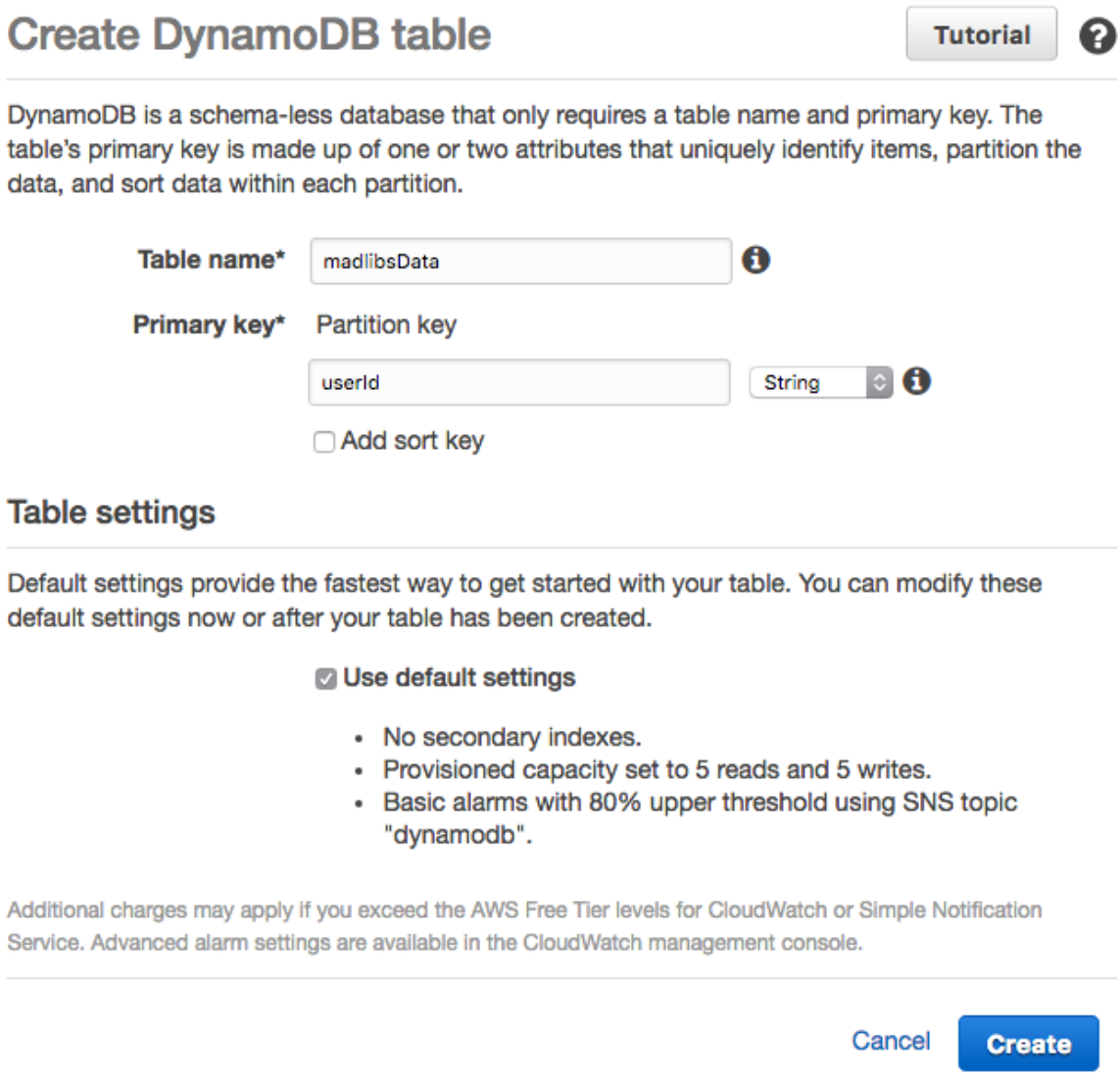
To begin deployment, you first need to configure the AWS DynamoDB to work correctly with your skill. To configure the database, visit `https://console.aws.amazon.com/dynamodb/home?region=us-east-1`.

Figure 1.6 The AWS DynamoDB Page



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

Figure 1.7 Creating a DynamoDB table



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

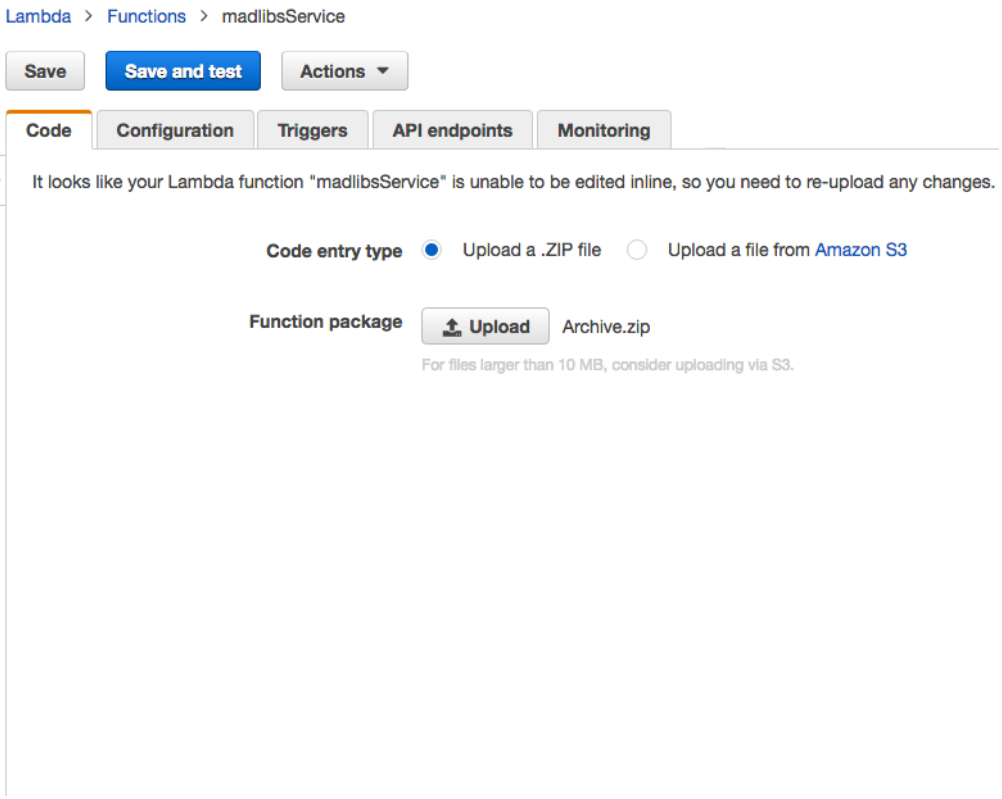
Next, edit the `database_helper.js` database configuration so that it is ready for the live environment. Edit `database_helper.js`:

Listing 1.16 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);
};
```

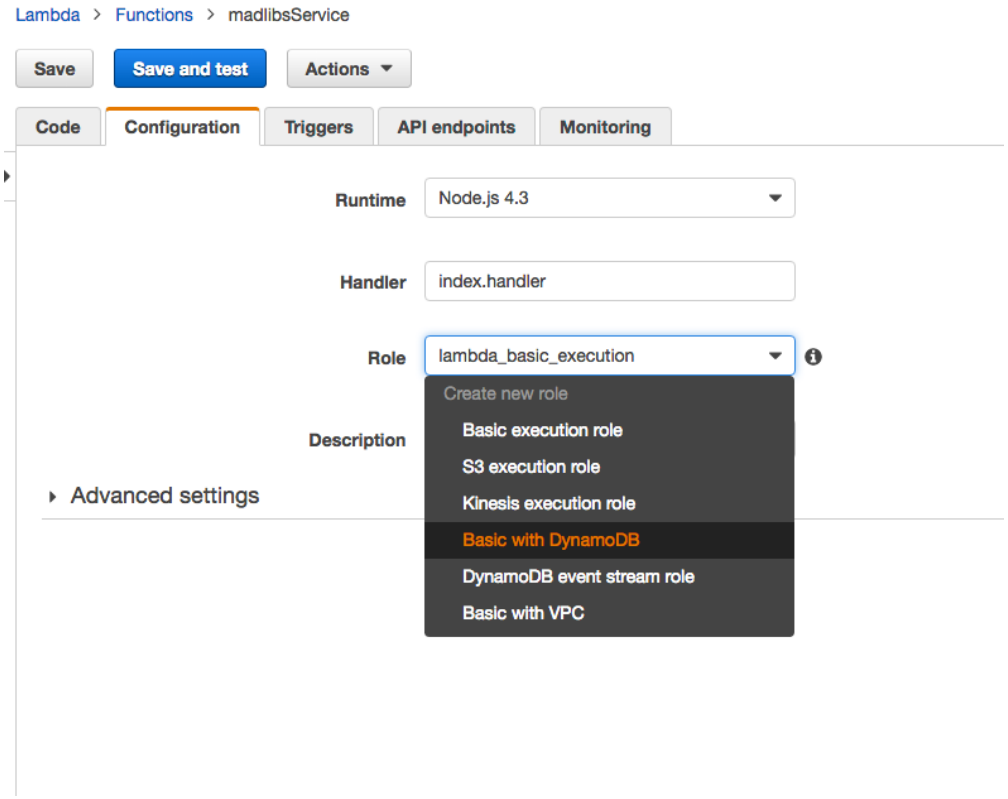
To deploy the skill to AWS, you update the source code that is running on the AWS Lambda function you set up earlier for the Madlib Builder skill. Compress the contents within the `/madlibsbuilder` directory to create a new Archive. Next, return to the "madlibsService" AWS Lambda function you set up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

Figure 1.8 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. You will be redirected to a new page. On this page, click Allow at the bottom right of the screen.

Figure 1.9 Updating the Role

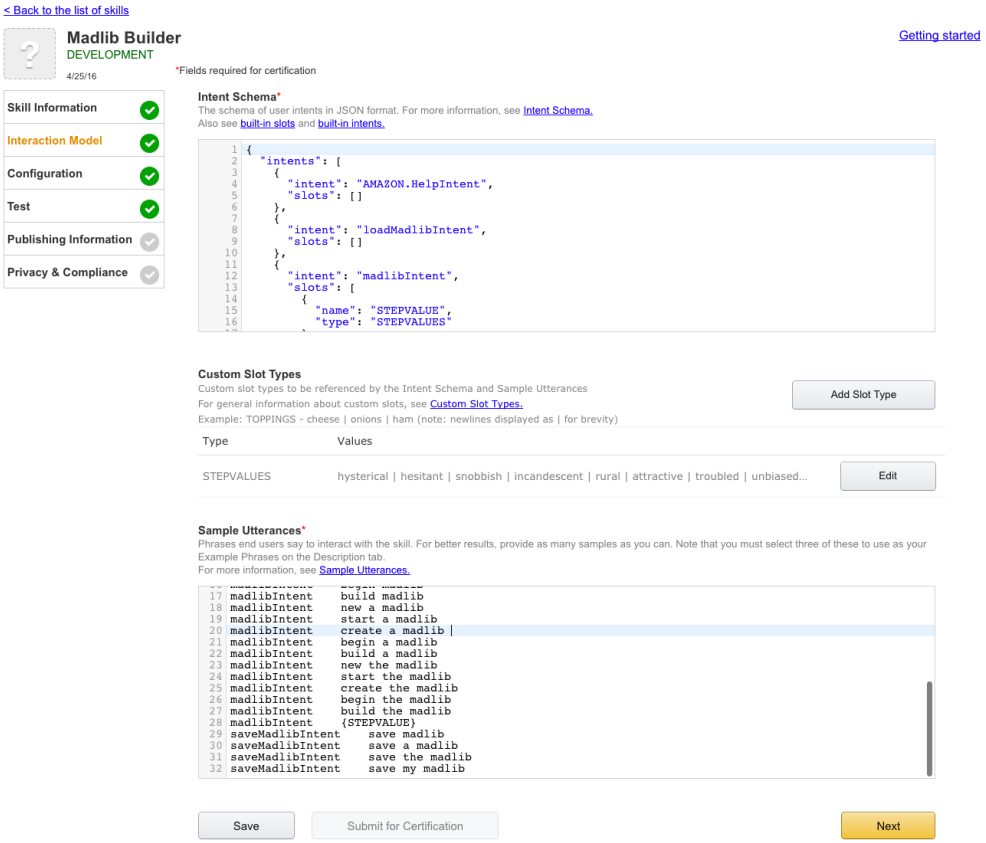


Finally, once you are redirected to the service page, click Save at the top left of the page.

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.

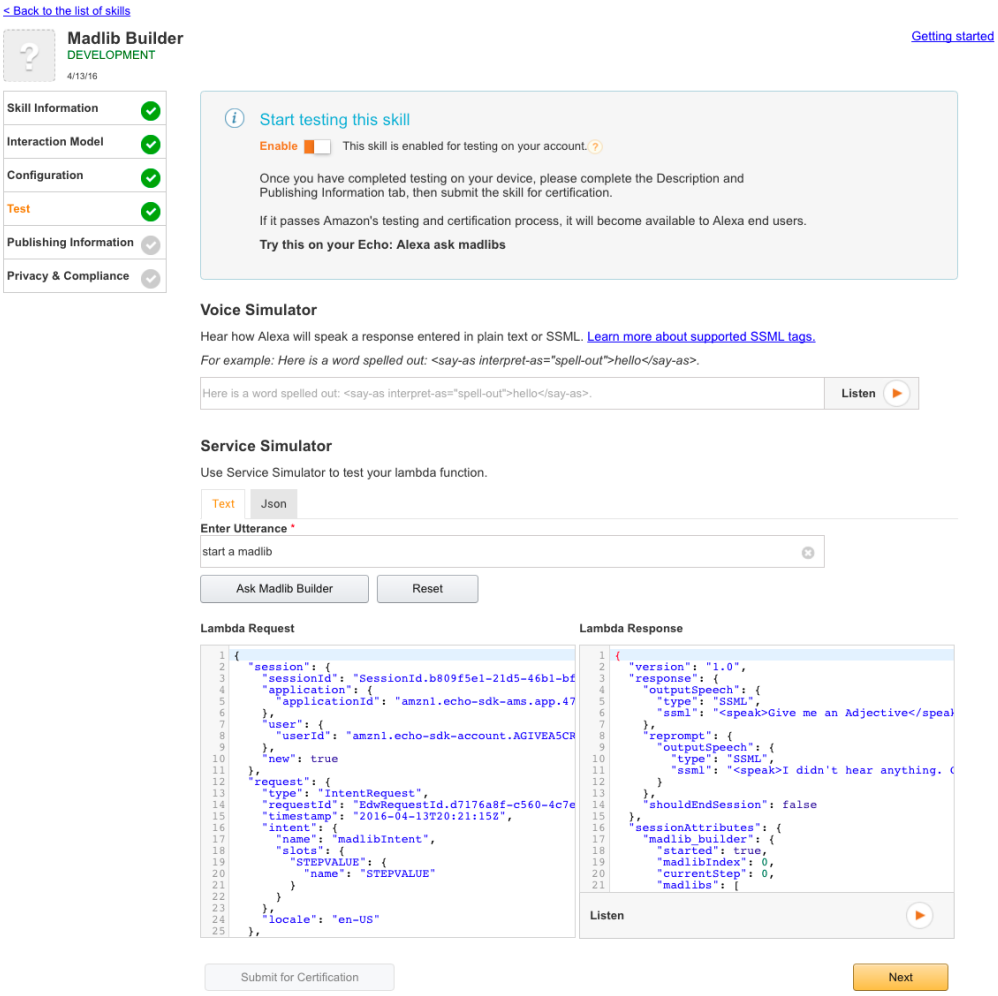
Figure 1.10 Updating the Interaction Model



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 1.11 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 1.17 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! 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.