Extracted from:

Build Talking Apps

Develop Voice-First Applications for Alexa

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Build Talking Apps

Develop Voice-First Applications for Alexa

Craig Walls

edited by Jacquelyn Carter

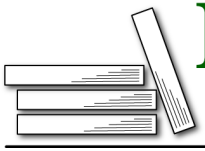
Build Talking Apps

Develop Voice-First Applications for Alexa

Craig Walls

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-725-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2022

CHAPTER 3

Parameterizing Intents with Slots

Much of human communication is made up of sentences that have parameters. Someone might say, "The weather is cold today." But on a different day, they may say, "The weather is warm today." The only difference between these two sentences is the words "cold" and "warm." They change the meaning of the sentence, but the purpose is the same with both: to describe the weather.

The same holds true when talking to Alexa. You might ask her to play Mozart. Or you might ask her to play Van Halen. Either way, the objective is the same (to play some music) although the specifics are different (Mozart vs. Van Halen).

The main thing that Star Port 75 Travel wants to offer to their clientele through their Alexa skill is the ability to schedule a trip to one of their several planetary destinations. For example, one potential space traveler might say something like "Schedule a trip to Mercury leaving Friday and returning next Thursday." Another user may have a different trip in mind and say, "Plan a trip between June 9th and June 17th to visit Mars."

Although these two requests are worded differently, their purpose is the same: to schedule a trip. The specifics—the destination, departure date, and return date—are parameters that define the trip that is to be scheduled. When handling such requests in an Alexa skill, these parameters are known as *slots*. We're going to use slots in this chapter to handle parameters in the utterances spoken by the skill's users.

Adding Slots to an Intent

The Star Port 75 Travel skill needs to be able to book trips for spacefaring adventurers based on three parameters: the destination, the departure date, and the return date. To enable this, we'll add a new intent to the skill that

accepts those parameters in slots. But first, let's start by writing a BST test specification that captures and asserts what we want the intent to do:

slots/starport-75/test/unit/schedule-trip.test.yml

```
---
configuration:
  locales: en-US

---
- test: Schedule a trip
- "Schedule a trip to Mercury leaving Friday and returning next Thursday":
  - prompt: I've got you down for a trip to Mercury,
            leaving on Friday and returning next Thursday.

---
- test: Plan a trip
- "Plan a trip between June 9th and June 17th to visit Mars":
  - prompt: I've got you down for a trip to Mars,
            leaving on June 9th and returning June 17th.
```

These tests cover two possible ways of wording a request to schedule a trip. But even more interesting, they each cover different values for the three parameters and assert that those values are reflected in the intent's response.

The first step toward making these tests pass is to define the new intent in the interaction model. As with the HelloWorldIntent, we'll add a new entry to skill-package/interactionModels/custom/en-US.json within the intents property. And, just like HelloWorldIntent, we'll give it a name and a list of sample utterances. But as you can see from this interaction model excerpt, ScheduleTripIntent has a few new tricks:

slots/starport-75/skill-package/interactionModels/custom/en-US.json

```
"intents": [

...

  {
    "name": "ScheduleTripIntent",
    "samples": [
      "schedule a trip to {destination} leaving {departureDate} and
        returning {returnDate}",
      "plan a trip between {departureDate} and {returnDate}
        to visit {destination}"
    ],
    "slots": [
      {
        "name": "destination",
        "type": "AMAZON.City"
      },
```

```
      {
        "name": "departureDate",
        "type": "AMAZON.DATE"
      },
      {
        "name": "returnDate",
        "type": "AMAZON.DATE"
      }
    ]
  }
]
```

The first thing you'll notice is that the sample utterances don't have any explicitly stated destination or date values. Instead, they have placeholders in the form of variable names wrapped in curly-braces. These represent the places in the utterance where the slots will be provided.

The slots themselves are defined in the slots property. Each has a name and a type. The name must match exactly with the placeholder in the utterances. As for type, Amazon provides several built-in types,[1] including types for dates, numbers, and phone numbers. Amazon also provides nearly 100 slot types that identify a list of potential values such movie names, sports, book titles, and cities.

The slots defined in ScheduleTripIntent take advantage of two of Amazon's built-in types: AMAZON.City and AMAZON.DATE. It makes sense that the "departureDate" and "returnDate" slots are typed as AMAZON.DATE. But you might be wondering why "destination" is defined as AMAZON.City. Put simply, it's because Amazon doesn't define a built-in type for "planets" or any other astronomical locations. We'll create a custom type for planets in the next section. But until we get around to that, AMAZON.City will be a fine temporary stand-in.

Slot types are used as hints to help Alexa's natural language processing match what a user says to an intent. For example, suppose that the user asks to plan a trip to Seattle, but pronounces the city as "see cattle." The natural language processor may hear "see cattle," but since that sounds a lot like "Seattle," it can infer that the user meant "Seattle" based on the slot type AMAZON.City.

On the other hand, suppose that the user asks to plan a trip to "Jupiter," which is not a city included in the AMAZON.City type. Alexa's natural language processor will hear "Jupiter" and since no entry in the AMAZON.City type sounds

1. https://developer.amazon.com/docs/custom-skills/slot-type-reference.html

anything like that, it will give the user benefit of the doubt and give "Jupiter" as the slot's value.

Now that we've defined the intent, sample utterances, and slots in the interaction model, we need to write the intent handler. Rather than pile it on along with the other intent handlers in index.js, let's split it out into its own JavaScript module. This will help keep the skill's project code more organized and prevent index.js from growing unwieldy. The intent handler's code, defined in lambda/ScheduleTripIntentHandler.js, looks like this:

slots/starport-75/lambda/ScheduleTripIntentHandler.js

```
const Alexa = require('ask-sdk-core');

const ScheduleTripIntentHandler = {
  canHandle(handlerInput) {
    return Alexa.getRequestType(
              handlerInput.requestEnvelope) === 'IntentRequest'
      && Alexa.getIntentName(
              handlerInput.requestEnvelope) === 'ScheduleTripIntent';
  },
  handle(handlerInput) {
    const destination =
        Alexa.getSlotValue(handlerInput.requestEnvelope, 'destination');
    const departureDate =
        Alexa.getSlotValue(handlerInput.requestEnvelope, 'departureDate');
    const returnDate =
        Alexa.getSlotValue(handlerInput.requestEnvelope, 'returnDate');

    const speakOutput = handlerInput.t('SCHEDULED_MSG',
        {
          destination: destination,
          departureDate: departureDate,
          returnDate: returnDate
        });

    return handlerInput.responseBuilder
      .speak(speakOutput)
      .withShouldEndSession(true)
      .getResponse();
  },
};

module.exports=ScheduleTripIntentHandler;
```

You'll want to be sure to register this new intent handler with the skill builder, just like we did with HelloWorldIntentHandler. The intent handler can be imported into index.js using require() and then added to the list of handlers like this:

slots/starport-75/lambda/index.js

```
const HelloWorldIntentHandler = require('./HelloWorldIntentHandler');
const ScheduleTripIntentHandler = require('./ScheduleTripIntentHandler');
const StandardHandlers = require('./StandardHandlers');

  ...

exports.handler = Alexa.SkillBuilders.custom()
    .addRequestHandlers(
        HelloWorldIntentHandler,
        ScheduleTripIntentHandler,
        StandardHandlers.LaunchRequestHandler,
        StandardHandlers.HelpIntentHandler,
        StandardHandlers.CancelAndStopIntentHandler,
        StandardHandlers.FallbackIntentHandler,
        StandardHandlers.SessionEndedRequestHandler,
        StandardHandlers.IntentReflectorHandler)
    .addErrorHandlers(
        StandardHandlers.ErrorHandler)
    .addRequestInterceptors(
        LocalisationRequestInterceptor)
    .lambda();
```

As with the HelloWorldIntentHandler, ScheduleTripIntentHandler is defined by two functions: canHandle() to determine if the intent handler is capable of handling the request's intent, and handle() to handle the request if so.

The most significant and relevant difference in ScheduleTripIntentHandler, however, is in the first few lines of the handle() function. They use the Alexa.getSlotValue() function to extract the values of the "destination", "departureDate", and "returnDate" slots and assign them to constants of their own. Those constants are referenced in an object passed to the t() function when looking up the "SCHEDULED_MSG" message assigned to speakOutput.

For that to work, we'll need to define "SCHEDULED_MSG" to languageStrings.js.

slots/starport-75/lambda/languageStrings.js

```
module.exports = {
  en: {
    translation: {
      ...
      SCHEDULED_MSG: "I've got you down for a trip to {{destination}}, " +
            "leaving on {{departureDate}} and returning {{returnDate}}",
      ...
    }
  }
}
```

As you can see, embedded within the "SCHEDULED_MSG" string are placeholders, denoted by double-curly-braces, that will be replaced with the properties from the object we passed to the t() function.

With the new intent defined in the interaction model and its corresponding intent handler and language string written, we've just added basic trip scheduling support to the skill. Let's run the tests and see if it works:

```
% bst test --jest.collectCoverage=false schedule-trip.test.yml

BST: v2.6.0  Node: v17.6.0
Did you know? You can use the same YAML syntax for both your end-to-end
and unit tests. Find out more at https://read.bespoken.io.

 PASS  test/unit/schedule-trip.test.yml
  en-US
    Schedule a trip
      ✓ Schedule a trip to Mercury leaving Friday and returning next Thursday
    Plan a trip
      ✓ Plan a trip between June 9th and June 17th to visit Mars

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.128s, estimated 2s
Ran all test suites.
```

As you can see, both tests passed! We've successfully used slots to handle variable input from the user when booking a trip! For brevity's sake, we can run bst test with --jest.collectCoverage=false so that the test coverage report is not in the output. And, the test specification's name is passed as a parameter in order to focus on the tests relevant to our new intent.

Although it works, there's still room for improvement. Ultimately our skill is for planning interplanetary travel. Therefore, AMAZON.City isn't really the best slot type for our needs. But before we swap it out for a custom planets slot type, let's take a quick look at how some entities may offer more information than just the entity's name.

Fetching Entity Information

While testing the previous example, if you inspected the intent request closely enough, you may have spotted something very interesting about the resolved slot value. Specifically, not only did the value have a name, it also had an id property whose value is a URL.

For example, if the city spoken in place of the city slot were Paris, the request might look a little like this:

```
"city": {
  "name": "city",
  "value": "Paris",
  "resolutions": {
    "resolutionsPerAuthority": [
      {
        "authority": "AlexaEntities",
        "status": {
          "code": "ER_SUCCESS_MATCH"
        },
        "values": [
          {
            "value": {
              "name": "Paris",
➤             "id": "https://ld.amazonalexa.com/entities/v1/1z1ky..."
            }
          }
        ]
      }
    ]
  },
  "confirmationStatus": "NONE",
  "source": "USER",
  "slotValue": {
    "type": "Simple",
    "value": "Paris",
    "resolutions": {
      "resolutionsPerAuthority": [
        {
          "authority": "AlexaEntities",
          "status": {
            "code": "ER_SUCCESS_MATCH"
          },
          "values": [
            {
              "value": {
                "name": "Paris",
➤               "id": "https://ld.amazonalexa.com/entities/v1/1z1ky..."
              }
            }
          ]
        }
      ]
    }
  }
}
```

As it turns out, the URL in the `id` property can be fetched with an HTTP GET request to lookup additional information about the resolved entity. This is a

relatively new feature called *Alexa Entities* and at this time is currently in Beta for skills deployed in the following locales:

- English (AU)
- English (CA)
- English (IN)
- English (UK)
- English (US)
- French (FR)
- German (DE)
- Italian (IT)
- Spanish (ES)

(We'll talk more about locales in Chapter 8, Localizing Responses, on page ?.)

In the case of a slot whose type is `AMAZON.City`, that includes details such as the average elevation, which larger government boundaries the city is contained within (for example, metroplex, state, country), and the human population of the city. While not all skills will need this extra information, it can come in very handy for skills that do.

For example, suppose that we were building a skill with an intent that provided population information for a city. Such an intent might be defined like this:

slots/city-population/skill-package/interactionModels/custom/en-US.json

```
{
  "name": "CityPopulationIntent",
  "slots": [
    {
      "name": "city",
      "type": "AMAZON.City"
    }
  ],
  "samples": [
    "what is the population of {city}",
    "tell me about {city}",
    "how many people live in {city}",
    "how big is {city}"
  ]
},
```

The expectation is that if the user were to ask, "What is the population of Paris?" then Alexa would respond with the number of people living in Paris.

Without Alexa Entities, you'd have to maintain a database of city population data or perhaps delegate out to some API that provides such information. But

with Alexa Entities, the information is readily available to your skill, just for the asking.

The way to ask for entity data is to make an HTTP GET request to the URL in the id property, providing an API access token in the Authorization header of the request. The API access token is made available in the intent's request envelope and can be easily be obtained with Alexa.getApiAccessToken() like this:

```
const apiAccessToken =
      Alexa.getApiAccessToken(handlerInput.requestEnvelope);
```

You'll also need the request's locale, which is just as readily available from the request envelope:

```
const locale = Alexa.getLocale(handlerInput.requestEnvelope);
```

With the entity URL, locale, and an access token in hand, making the request for entity information can be done using any JavaScript client library that you like. For our project, we'll use the Axios client library. You can install it by issuing the following command from the project's root directory:

```
$ npm install --prefix=lambda axios
```

With Axios installed, the following snippet shows how to request entity information:

slots/city-population/lambda/index.js

```
const resolvedEntity = resolutions.values[0].value.id;
const headers = {
  'Authorization': `Bearer ${apiAccessToken}`,
  'Accept-Language':locale
};

const response =
    await axios.get(resolvedEntity, { headers: headers });
```

Here, the first resolution is chosen and its ID is assigned to a constant named resolvedEntity. The value of resolvedEntity is not just a simple ID, but also the URL of the entity to be fetched. Therefore, it is passed in as the URL parameter to axios.get() to retrieve entity details.

Assuming that the request is successful, the response will include a JSON document with several properties that further define the resolved entity.

As an example, here's a sample of what you'll get if the entity is the city of Paris:

```
{
    "@context": {
      ...
```

```
        },
        "@id": "https://ld.amazonalexa.com/entities/v1/1z1kyo7XwxYGAKcx5F3TCf",
        "@type": [ "City" ],
        "averageElevation": [{ "@type": "unit:Meter", "@value": "28" }],
        "capitalOf": [
            {
                "@id": "https://ld.amazonalexa.com/entities/v1/LGYtKPDONTW...",
                "@type": [ "Country" ],
                "name": [{ "@language": "en", "@value": "France" }]
            }
        ],
        "countryOfOrigin": {
            "@id": "https://ld.amazonalexa.com/entities/v1/LGYtKPDONTWCtt6...",
            "@type": [ "Country" ],
            "name": [{ "@language": "en", "@value": "France" }]
        },
➤       "humanPopulation": [{ "@type": "xsd:integer", "@value": "2140000" }],
        "locatedWithin": [
            {
                "@id": "https://ld.amazonalexa.com/entities/v1/DAy2cvRGvSB...",
                "@type": [ "Place" ],
                "name": [{ "@language": "en", "@value": "Paris" }]
            },
            {
                "@id": "https://ld.amazonalexa.com/entities/v1/1NlBgtwDmHb...",
                "@type": [ "Place" ],
                "name": [{ "@language": "en", "@value": "Île-de-France" }]
            },
            {
                "@id": "https://ld.amazonalexa.com/entities/v1/LGYtKPDONTW....",
                "@type": [ "Country" ],
                "name": [{ "@language": "en", "@value": "France" }]
            }
        ],
        "name": [{ "@language": "en", "@value": "Paris" }]
    }
```

Without looking any further, your skill can use any of this information as it sees fit, including reporting the population of the city. But also notice that some of the properties include their own URLs in `@id` properties. So, for example, if you wanted your skill to dig even deeper into the country that Paris is the capital of, you could make another request, following the URL in the `@id` property from the `countryOfOrigin` property.

All we need for a simple city population skill, however, is the value from the `humanPopulation` property. The following `fetchPopulation()` function shows how we might fetch the population for a given set of resolutions and API access token:

slots/city-population/lambda/index.js
```
const fetchPopulation = async (resolutions, locale, apiAccessToken) => {
  const resolvedEntity = resolutions.values[0].value.id;
  const headers = {
    'Authorization': `Bearer ${apiAccessToken}`,
    'Accept-Language':locale
  };

  const response =
      await axios.get(resolvedEntity, { headers: headers });
  if (response.status === 200) {
    const entity = response.data;
    if ('name' in entity && 'humanPopulation' in entity) {
      const cityName = entity.name[0]['@value'];
      const population = entity.humanPopulation[0]['@value'];
      const popInfo = {
        cityName: cityName,
        population: population
      };
      return popInfo;
    }
  } else {
    return null;
  }
};
```

After sending the GET request for the entity, if the response is an HTTP 200 (OK) response, then it extracts the value of the humanPopulation property from the response. We'll also need to know the fully resolved entity name for our intent's response, so while fetching the population, we also fetch the value of the name property. Both are packed up in an object and returned to the caller.

As for how fetchPopulation() is used, here's the intent handler which asks for the population and uses the city name and population from the returned object to produce a response to the user:

slots/city-population/lambda/index.js
```
const CityPopulationIntentHandler = {
  canHandle(handlerInput) {
    return Alexa.getRequestType(
            handlerInput.requestEnvelope) === 'IntentRequest'
        && Alexa.getIntentName(
            handlerInput.requestEnvelope) === 'CityPopulationIntent';
  },
  async handle(handlerInput) {
    const apiAccessToken =
          Alexa.getApiAccessToken(handlerInput.requestEnvelope);
    const slot =
          Alexa.getSlot(handlerInput.requestEnvelope, 'city');
```

```
    const resolutions = getSlotResolutions(slot);
    const locale = Alexa.getLocale(handlerInput.requestEnvelope);

    if (resolutions) {
        const popInfo =
            await fetchPopulation(resolutions, locale, apiAccessToken);

        if (popInfo !== null) {
          const speechResponse =
              `${popInfo.cityName}'s population is ${popInfo.population}.`
          return handlerInput.responseBuilder
                .speak(speechResponse)
                .getResponse();
        }
   }

    const reprompt = 'What city do you want to know about?';
    const speakOutput =
          "I don't know what city you're talking about. Try again. "
          + reprompt;
    return handlerInput.responseBuilder
        .speak(speakOutput)
        .reprompt(reprompt)
        .getResponse();
  }
};
```

This handler leans on a couple of helper functions to extract the slot resolutions from the given slot:

slots/city-population/lambda/index.js

```
const getSlotResolutions = (slot) => {
    return slot.resolutions
        && slot.resolutions.resolutionsPerAuthority
        && slot.resolutions.resolutionsPerAuthority.find(resolutionMatch);
};

const resolutionMatch = (resolution) => {
    return resolution.authority === 'AlexaEntities'
        && resolution.status.code === 'ER_SUCCESS_MATCH';
};
```

With all of this in place, if the user were to ask for the population of Paris, Alexa will respond by saying, “Paris’s population is 2,140,000.”

Not all built-in slot types support Alexa entities. Several slot types do, however, including:

- AMAZON.Person
- AMAZON.Movie
- AMAZON.Animal
- AMAZON.City

- AMAZON.Country
- AMAZON.Book
- AMAZON.Author
- AMAZON.TVSeries
- AMAZON.Actor
- AMAZON.Director
- AMAZON.Food
- AMAZON.MusicGroup
- AMAZON.Musician
- AMAZON.MusicRecording
- AMAZON.MusicAlbum

Of course, each slot type will have information relevant to that type. `AMAZON.Movie`, for example, won't have a `humanPopulation` property, but it will have a property named `entertainment:castMember` that is an array of actors who were in the movie. Each entry in the `entertainment:castMember` array is itself a reference to a person with an `@id` that you can use to look up additional information about the actor, such as their birthday.

Now let's take our skill beyond the confines of Earth and create a custom type that represents planetary destinations instead of relying on the built-in `AMAZON.City` type.