# Alexa 3: Conversational Movie Facts

Now that we’re comfortable with the Alexa communication paradigm, using Intents and Utterances, and Slots (and Custom Slots), let’s introduce another major component of the Alexa Skills Kit: Sessions.

We’re going to build an application that allows users to ask this:

Alexa, ask Movie Facts about Titanic

Alexa should respond with some facts about the movie ‘Titanic’. Then, our users should be able to ask context-based questions *without restating the Invocation Name 'Movie Facts'*, such as:

Who directed that

Who starred in that

Alexa should respond with the director of ‘Titanic’, and a cast list.

Alexa should remember that the user asked about ‘Titanic’ in the first request, and limit her response to subsequent requests to the context of the first.

Additionally, a user will be able to ask:

Start over

And then follow up with:

Ask about Beauty and the Beast

Again, Alexa should then answer questions about 'Beauty and the Beast'.

## 1. An introduction to Sessions

To build this Conversational Interface, we will need to make use of Alexa’s ability to manage Sessions.

A ‘Conversational Interface’ allows users to engage in dialogue with technology, with the technology providing meaningful responses based on the context of the dialogue.

A Session lives for the life of a user's conversation with our skill. As developers, we can control when to end or continue the Session. If we end the Session, the user will need to start their next phrase with 'Alexa, ask Movie Facts ...'. If we leave it open, the user has 8 seconds to respond and continue the conversation. If there is no reply after 8 seconds, Alexa will provide a reprompt (defined by us) and wait for another 8 seconds before closing the Session herself. During this Session we can persist attributes (more on that later).

#### Set up a new skill and application

Set up a new skill, with an Invocation Name of ‘Movie Facts’, and a new Sinatra application. Again, we’ll be using ngrok to tunnel our development server over HTTPS, and providing the ngrok HTTPS endpoint to our skill as our endpoint.

Feel free to use another method of connecting a Ruby application to Alexa via HTTPS. We’ll move forward assuming you’re using an ngrok Tunnel, but you can adapt as desired.

Before we try and build our Movie Facts skill, let’s get to grips with some key concepts regarding Sessions: what they are, how we use them, and why they’re handy. We’ll build a simple VUI that responds to the following:

Alexa, ask Movie Facts to talk to me

Alexa should respond with “This is the first question”, **but only on the first request**. On all subsequent requests, Alexa should respond with a count of how many questions the user has asked.

In other words, when a user asks:

Alexa, ask Movie Facts to talk to me

Alexa should respond with “this is question number number”, depending on how many times the user has asked Movie Facts to talk with them.

#### Set up a minimal interaction

Let’s set up a minimal Intent Schema, using the Intent name MovieFacts:

{  
 "intents": [  
 {  
 "intent": "MovieFacts"  
 }  
 ]  
}

We’ll add a simple Utterance:

MovieFacts talk to me

Now, in our Sinatra application, we can provide a simple minimal response. In addition, let’s print the request so we can have a look at it:

require 'sinatra'  
require 'json'  
  
post '/' do   
 parsed\_request = JSON.parse(request.body.read)  
  
 # Print the incoming request  
 p parsed\_request  
   
 # Send back a simple response  
 return {   
 version: "1.0",  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "This is the first question"  
 }  
 }  
 }.to\_json  
end

#### Investigating the Session

Let's run this in the Service Simulator, by typing "talk to me". In your server logs, take a look at the "session" key from parsed\_request:

"session"=>{  
 "new"=>true,   
 "sessionId"=>"SessionId.120a73d8-c1dc-437c-8c5b-fb2d1057f991",  
 "application"=>{"applicationId"=>"A long string"},   
 "attributes"=>{},   
 "user"=>{"userId"=>"A long string"}  
}

Notice that the session.new key tells us that this is a **new** session. Now let’s pass the same Utterance to the Service Simulator again. Notice how the session.new key changes with this second request:

"session"=>{  
 "new"=>false,   
 "sessionId"=>"SessionId.120a73d8-c1dc-437c-8c5b-fb2d1057f991",  
 "application"=>{"applicationId"=>"A long string"},   
 "attributes"=>{},   
 "user"=>{"userId"=>"A long string"}  
}

Alexa remembers that this user has already interacted with Movie Facts, and mentions that in the request: the session "new" value has changed from true to false.

#### Responding depending on the state of the Session

Now that we know this, let’s upgrade our application to respond with two different strings, depending on whether this is the first question the user has asked to Movie Facts:

require 'sinatra'  
require 'json'  
  
post '/' do   
 parsed\_request = JSON.parse(request.body.read)  
 this\_is\_the\_first\_question = parsed\_request["session"]["new"]  
  
 if this\_is\_the\_first\_question  
 return {   
 version: "1.0",  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "This is the first question."  
 }  
 }  
 }.to\_json  
 end  
  
 return {  
 version: "1.0",  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "This is question number 2"  
 }  
 }  
 }.to\_json  
end

Running in the Service Simulator, or on any Alexa-enabled device, we receive the first message first, and the second message for all subsequent requests.

If you’re using the Service Simulator, don’t forget to hit the 'Reset' button, or refresh the page, to start a new session with Alexa.

#### Persisting information to the Session Attributes

However, there’s a problem: at the moment, the user will first hear “This is the first question”, and then they’ll hear “This is question number 2” *forever* – regardless of how many times they ask. We need a way to **persist information** about how many questions the user has asked, and reference it between requests.

To persist information between requests in this way, we can use **Session Attributes**. Sessions can store information about what a user has said to Alexa in the past, and our Sinatra application can use that persisted information to construct a response.

First, we need to initialise the Session Attributes for our first response to include a new attribute, numberOfRequests:

# for brevity, here's just the Ruby code making the first response  
if this\_is\_the\_first\_question  
 return {   
 version: "1.0",  
 # here, we can persist data across multiple requests and responses  
 sessionAttributes: {  
 numberOfRequests: 1  
 },  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "This is the first question."  
 }  
 }  
}.to\_json  
end

Using puts to output the request body, notice how the request now contains a reference to the number of requests made, in an attribute called numberOfRequests:

"session"=>{  
 "new"=>true,   
 "sessionId"=>"a long string",   
 "application"=>{"applicationId"=>"a long string"},   
 "attributes"=>{"numberOfRequests"=>1}

Now that the user has initialised the number of requests in their first interaction with Movie Facts, we can increment it in each subsequent interaction:

# for brevity, here's just the Ruby code for subsequent responses  
  
# grab the numberOfRequests attribute from the Session Attributes,  
# and increment it by 1  
number\_of\_requests = parsed\_request["session"]["attributes"]["numberOfRequests"] + 1  
  
return {  
 version: "1.0",  
 sessionAttributes: {  
 numberOfRequests: number\_of\_requests  
 },  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "This is question number #{ number\_of\_requests }"  
 }  
 }  
}.to\_json

Now, we are persisting – and acting on – data across multiple interactions. Try it out in the Service Simulator!

In the Service Simulator, remember to hit the 'Reset' button, or refresh the page, to start a new session with Alexa.

#### Different ways of restarting a Session

One final thing – what about if we want to allow users to start the count over? To do that, we have two choices:

1. End the Session.
2. Clear the Session Attributes.

These should be used in two different circumstances:

1. **End the Session** when Alexa **'tells'** the user something. For instance, "Goodbye."
2. **Clear the Session Attributes** when Alexa **'asks'** the user something. For instance, "Okay, starting over. Would you like to talk to me?"

The user's experience is different in each case:

1. **End the Session**: The user has to start the interaction over from the beginning, by asking: "Alexa, ask Movie Facts to talk to me". This is similar to the user 'logging out' of a web app.
2. **Clear the Session Attributes**: The existing Session continues, but the application 'forgets' everything that's happened so far. The user can just say "Talk to me" (i.e. no Invocation Name is required). This is similar to the user restarting some process in a web app, *without 'logging out'*.

Let’s allow users to say:

Start over.

Alexa should respond with:

Okay, starting over. Would you like to talk to me?

And the user should answer with:

Talk to me.

Since we don't want the user to restate the Invocation Name, we are going for **Option number 2: Clearing the Session Attributes**.

#### Starting a Session over using AMAZON.StartOverIntent

Amazon provides us with an Intent for starting an interaction from the beginning: AMAZON.StartOverIntent. Rather than defining our own, let's use the built-in Intent.

Before defining a new Intent, it's a good idea to check the [Amazon Built-in Intents](https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit/docs/built-in-intent-ref/standard-intents) first.

Because this is a Built-in Intent, we don't need to define an Utterance for it. In the Intent Schema, we add a new Intent, with an Intent name of AMAZON.StartOverIntent:

{  
 "intents": [  
 {  
 "intent": "MovieFacts"  
 },  
 {  
 "intent": "AMAZON.StartOverIntent"  
 }  
 ]  
}

In our Sinatra application, let’s add a response just for requests to clear the Session. In the response, we **clear the Session Attributes**, but **don't end the Session**:

if parsed\_request["request"]["intent"]["name"] == "AMAZON.StartOverIntent"  
 return {  
 version: "1.0",  
 # adding this line to a response will  
 # remove any Session Attributes  
 sessionAttributes: {},  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "Okay, starting over. What movie would you like to know about?"  
 },  
 # Let's be really clear that we're not  
 # ending the session, just restarting it  
 shouldEndSession: false  
 }  
 }.to\_json  
end

This response will now start the Session over. However, when the user next says "Alexa, talk to me", their session will not be "new": it'll just have **empty Session Attributes**. So, we need to upgrade our this\_is\_the\_first\_question variable:

# This is the 'first question' IF  
# the 'new' session key is true OR  
# the Session Attributes are empty  
this\_is\_the\_first\_question = parsed\_request["session"]["new"] || parsed\_request["session"]["attributes"].empty?

In fact, we can refactor this: **any 'new' session will have empty Session Attributes anyway**. So our final this\_is\_the\_first\_question variable looks like this:

this\_is\_the\_first\_question = parsed\_request["session"]["attributes"].empty?

#### Ending Sessions

You can end a session by setting shouldEndSession to true in the response. If you do this, you should **tell the user the session has ended**. In the example above, we could respond:

return {  
 version: "1.0",  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "Goodbye."  
 },  
 # End the session, and  
 # clear the Session Attributes  
 shouldEndSession: true  
 }  
}.to\_json

As well as restarting a session using a built-in Intent, users can **end** a session any time in one of three circumstances:

1. The user says “exit”,
2. The user does not respond, or says something that does not match an intent you have defined
3. An error occurs.

In either of these cases, your Sinatra Application will receive a special type of request: a SessionEndedRequest. Your application cannot return a response to SessionEndedRequests, but you may wish to use these requests to do some cleanup.

Now a user can reset their session, and start the question count over! Now let’s do something a little more complex.

## 2. Querying IMDb

First, we want users to be able to ask:

Alexa, ask Movie Facts about {some movie name}

Let’s upgrade our first Utterance to respond to information about movies:

MovieFacts about {Movie}

#### Adding a MOVIE slot

If your skill is an English (US) skill, you can use Amazon’s built-in AMAZON.Movie Slot Type to pass the name of the movie. If not, you’ll need to define a Custom Slot Type with the names of several movies, to guide voice recognition for whichever movie the user requests. Assuming the latter, let’s define a Custom Slot Type, named MOVIE, with a definition containing a few example movies:

titanic  
jaws  
the perfect storm

If you would prefer to use an exhaustive list of movies available on the Internet Movie Database (IMDb), you can find a list of every movie IMDb has listed [here](ftp://ftp.funet.fi/pub/mirrors/ftp.imdb.com/pub/).

Add a slot with the appropriate slot type to your Intent Schema, and test that your slot is filled appropriately by printing requests to your Sinatra application.

#### Querying IMDb using a gem

In our Sinatra application, let’s use the Open-Source [IMDb gem](https://github.com/ariejan/imdb) to query IMDb for information about whichever movie the user wants to know more about:

require 'sinatra'  
require 'json'  
# include the IMDb gem to query IMDb easily  
require 'imdb'  
  
post '/' do   
 parsed\_request = JSON.parse(request.body.read)  
 this\_is\_the\_first\_question = parsed\_request["session"]["attributes"].empty?  
  
 if this\_is\_the\_first\_question  
 # Fetch the name of the movie the user wanted information about  
 requested\_movie = parsed\_request["request"]["intent"]["slots"]["Movie"]["value"]  
 # Search IMDb for all movies matching that name  
 movie\_list = Imdb::Search.new(requested\_movie).movies  
 # Pick the first one  
 movie = movie\_list.first  
  
 return {   
 version: "1.0",  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 # Return the plot synopsis for that movie to the user  
 text: movie.plot\_synopsis  
 }  
 }  
 }.to\_json  
 end  
end

Remember to run the command-line command gem install imdb before you try to run your Sinatra application (or use a more rigorous dependency management system such as [Bundler](http://bundler.io/gemfile.html)).

Once you’ve verified this is all working in the Service Simulator, let’s move on to the final section: using the session to make a Conversational Interface.

## 3. Building a dialogue

So far, our users can ask Alexa:

Alexa, ask Movie Facts about {some movie name}

Alexa will respond with the plot synopsis for the first movie matching the name the user provides. For example, if a user asks “Alexa, ask Movie Facts about Titanic”, Alexa will respond with a plot synopsis for the 1997 movie *Titanic*.

We’d love our users to ask follow-up questions about the movie they initially queried – but how can we do that without requiring the user give the movie name a second time? Let’s use Session Attributes!

#### Remembering the movie the user asked about

We can persist the title of the requested movie after our initial request, using the Session Attributes:

if this\_is\_the\_first\_question  
 requested\_movie = parsed\_request["request"]["intent"]["slots"]["Movie"]["value"]  
 movie\_list = Imdb::Search.new(requested\_movie).movies  
 movie = movie\_list.first  
  
 return {   
 version: "1.0",  
 sessionAttributes: {  
 # Persist the movie name to the Session  
 movieTitle: requested\_movie  
 },  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: movie.plot\_synopsis  
 }  
 }  
 }.to\_json  
end

Now we can access the movie title on subsequent requests.

#### Querying for more information about the movie

We want our users to be able to query for information about the movie, such as:

Who directed that

Who starred in that

Since finding out more about a movie is a new 'intent' on the part of the user, let's define a new Intent in our Intent Schema, called FollowUp.

Let’s create an Utterance for this:

FollowUp who {Role} that

And a Custom Slot Type for possible Roles people might have in the movie, called ROLE:

directed  
starred in

Now let’s add that Custom Slot to our Intent Schema:

"intents": [  
 {  
 "intent": "MovieFacts",  
 "slots": [  
 {  
 "name": "Movie",  
 "type": "MOVIE"  
 }  
 ]  
 },  
 {  
 "intent": "FollowUp"  
 "slots": [  
 {  
 "name": "Role",  
 "type": "ROLE"  
 }  
 ]  
 },  
 {  
 "intent": "AMAZON.StartOverIntent"  
 }  
]

Now, let's ensure our Sinatra application can respond to these subsequent requests:

# After the block that handles the first request  
if parsed\_request["request"]["intent"]["name"] == "FollowUp"  
 # Fetch the movie title from the Session Attributes  
 movie\_title = session["attributes"]["movieTitle"]  
  
 # Search again for this movie, and pull out the first one  
 movie\_list = Imdb::Search.new(movie\_title).movies  
 movie = movie\_list.first  
  
 # Find out which Role the user was interested in  
 # this could be 'directed' or 'starred in' (or any other Values  
 # we provided to our Custom Slot Type)  
 role = parsed\_request["request"]["intent"]["slots"]["Role"]["value"]  
  
 # Construct response text if the user wanted to know  
 # who directed the movie  
 if role == "directed"  
 response\_text = "#{movie\_title} was directed by #{movie.director.join}"  
 end  
  
 # Construct response text if the user wanted to know  
 # who starred in the movie  
 if role == "starred in"  
 response\_text = "#{movie\_title} starred #{movie.cast\_members.join(", ")}"  
 end  
  
 # Pass the response text to the response, and remember to  
 # store the movie title in the Session Attributes so users  
 # can make subsequent requests about role in this movie  
 return {  
 version: "1.0",  
 sessionAttributes: {  
 movieTitle: movie\_title  
 },  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: response\_text  
 }  
 }  
 }.to\_json  
end

#### Routing multiple Intents

We now have three possible Intents (as well as numerous Built-in Intents) the user can use: AMAZON.StartOverIntent, MovieFacts, and FollowUp. In each case, our Sinatra application does something different:

* AMAZON.StartOverIntent: Clear the session and start again, ready to ask about a new movie.
* MovieFacts: Retrieve the synopsis of a movie, ready for follow-up questions about that movie.
* FollowUp: Give more information about a given movie.

Your Intent Schema will generally tie one-to-one with actions in your application. In other words, **our post / route is acting as a kind of router, with Intents as the possible routes**.

As a result of this three-intent system, we no longer need to know if this\_is\_the\_first\_question. Let's upgrade our code to reflect that:

require 'sinatra'  
require 'json'  
require 'imdb'  
  
post '/' do   
 parsed\_request = JSON.parse(request.body.read)  
  
 # Route 1: Starting Over  
 if parsed\_request["request"]["intent"]["name"] == "AMAZON.StartOverIntent"  
 return {  
 version: "1.0",  
 sessionAttributes: {},  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "OK, what movie would you like to know about?"  
 }  
 }  
 }.to\_json  
 end  
  
 # Route 2: MovieFacts Intent  
 if parsed\_request["request"]["intent"]["name"] == "MovieFacts"  
 requested\_movie = parsed\_request["request"]["intent"]["slots"]["Movie"]["value"]  
 movie\_list = Imdb::Search.new(requested\_movie).movies  
 movie = movie\_list.first  
  
 return {   
 version: "1.0",  
 sessionAttributes: {  
 movieTitle: requested\_movie  
 },  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: movie.plot\_synopsis  
 }  
 }  
 }.to\_json  
 end  
  
 # Route 3: FollowUp Intent  
 if parsed\_request["request"]["intent"]["name"] == "FollowUp"  
 movie\_title = parsed\_request["session"]["attributes"]["movieTitle"]  
 movie\_list = Imdb::Search.new(movie\_title).movies  
 movie = movie\_list.first  
  
 role = parsed\_request["request"]["intent"]["slots"]["Role"]["value"]  
  
 if role == "directed"  
 response\_text = "#{movie\_title} was directed by #{movie.director.join}"  
 end  
  
 if role == "starred in"  
 response\_text = "#{movie\_title} starred #{movie.cast\_members.join(", ")}"  
 end  
  
 return {  
 version: "1.0",  
 sessionAttributes: {  
 movieTitle: movie\_title  
 },  
 response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: response\_text  
 }  
 }  
 }.to\_json  
 end  
end

It's no coincidence that this routing system could be represented by a switch statement: in module 4, we'll use OO principles to extract a more readable representation of a router.

#### Testing conversations in the Service Simulator

Let’s test this out in the Service Simulator, or on any Alexa-enabled device. First, the user can ask:

Alexa, ask Movie Facts about Titanic

Alexa responds with “In 1996, treasure hunter Brock Lovett…”: the plot synopsis for Titanic. But who directed it?

Who directed that

Alexa responds with "Titanic was directed by James Cameron”. Great! And, because we’re storing the movie title in the Session Attributes, our users can continue querying:

Who starred in that

Alexa responds with a list of cast members for the 1997 movie *Titanic*. And, because we’ve added a Session-clearing intent, users can ask:

Start over

And they’ll be offered the chance to start querying a new movie. When they query the new movie, the user doesn't have to state the Invocation Name, or 'Alexa':

Ask about Beauty and the Beast

Awesome!

## 4. Improving the User Experience (UX)

Let's look at some ways we can improve the user's interaction with this application.

#### Limiting response text

At the moment, the user can ask:

Alexa, ask Movie Facts about Titanic

Alexa will respond with the entire plot synopsis for the 1997 movie *Titanic*. It's pretty long! The user will be waiting around for a while before they get a chance to query the movie further. Let's improve the user experience by chopping it off after the first 140 characters of synopsis:

# inside our initial response to this first question  
  
...  
response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: movie.plot\_synopsis.slice(0, 140)  
 }  
}  
...

We can do the same with the director and cast lists:

# Construct response text if the user wanted to know  
# who directed the movie  
if role == "directed"  
 response\_text = "#{movie\_title} was directed by #{movie.director.join.slice(0, 140)}"  
end  
  
# Construct response text if the user wanted to know  
# who starred in the movie  
if role == "starred in"  
 response\_text = "#{movie\_title} starred #{movie.cast\_members.join(", ").slice(0, 140)}"  
end

That slightly improves the UX!

EXTRA CREDIT: Extracting sentences from strings is a tough task. However, there are regexes which can approximate it. Upgrade this response-shortening to extract the first few sentences of each response, rather than arbitrarily chopping off the response in the middle of a word.

#### Adding prompts

The user may not know that they can query Alexa for further information about a movie. Alexa should prompt them. Let's append some strings to our responses, giving the user prompts for their next action:

# inside our initial response to this first question  
  
...  
response: {  
 outputSpeech: {  
 type: "PlainText",  
 text: "#{movie.plot\_synopsis.slice(0, 140)}. You can ask who directed that, or who starred in it."  
 }  
}  
...

We can do the same with the director and cast lists:

# Construct response text if the user wanted to know  
# who directed the movie  
if role == "directed"  
 response\_text = "#{movie\_title} was directed by #{movie.director.join.slice(0, 140)}. You can ask who directed #{movie\_title}, ask who starred in it, or start over."  
end  
  
# Construct response text if the user wanted to know  
# who starred in the movie  
if role == "starred in"  
 response\_text = "#{movie\_title} starred #{movie.cast\_members.join(", ").slice(0, 140)}. You can ask who directed #{movie\_title}, ask who starred in it, or start over."  
end

Now that we've implemented some more signposting for the user, our skill is easier for them to use.

### Extra Credits

EXTRA CREDIT 1: It can take a while to search IMDb and then whittle down the response to a single movie. Using a more sophisticated set of Session Attributes, try persisting information relevant to the movie in the session, and extracting subsequent user requests from the session instead of querying IMDb.

EXTRA CREDIT 2: Our codebase is looking pretty scrappy, and it’s highly procedural. There are a few things that feel like they’re violating the ‘Don’t Repeat Yourself’ rule by duplicating knowledge about the system at several points. Try refactoring the procedural codebase into something a little more OO. If you do it right, you’ll wind up with the start of a useful framework that could abstract some of the messy JSON manipulation we’ve been doing. This will be the subject of module 4.

EXTRA CREDIT 3: It’s important to know that the request to your application is coming from Alexa, and not from anywhere else (say, a user trying to access your application via cURL from the command-line). To do this, Amazon recommend that before taking any action on a request, developers first **verify** the request you receive comes from the application you expect. JSON requests from the Amazon Alexa Service come with a key for doing just this: the session.application.applicationId. The value for this key is a string. For extra credit, add a guard clause to verify that the request came from your application, and return an appropriate HTTP error if it does not.

EXTRA CREDIT 4: It's pretty easy to crash our application: say, if the user asks for a movie that doesn't exist. Upgrade the application handling of the MovieFacts Intent to handle the case where the user's requested movie cannot be found.