Voxa Documentation

Release 2.0.0

Rain Agency

Mar 06, 2017

Contents:

CHAPTER 1

Installation

Voxa is distributed via `npm`

```
$ npm install voxa --save
```

CHAPTER 2

Initial Configuration

Instantiating a StateMachineSkill requires a configuration specifying your *Views and Variables*.

```
'use strict';
const Voxa = require('voxa');
const views = require('./views'):
const variables = require('./variables');

const skill = new Voxa({ variables, views });
```

CHAPTER 3

Responding to alexa events

Once you have your skill configured responding to events is as simple as calling the *skill.execute* method

```
const skill = require('./MainStateMachine');

exports.handler = function handler(event, context) {
  skill.execute(event, context)
    .then(context.succeed)
    .catch(context.fail);
};
```

CHAPTER 4

Responding to an intent event

```
skill.onIntent('HelpIntent', (alexaEvent) => {
  return { reply: 'HelpIntent.HelpAboutSkill' };
});

skill.onIntent('ExitIntent', (alexaEvent) => {
  return { reply: 'ExitIntent.Farewell' };
});
```

CHAPTER 5

Project Samples

To help you get started the state machine has a number of example projects you can use.

Starter Kit

This is the simplest project, it defines the default directory structure we recommend using with voxa projects and has an example `serverless.yml` file that can be used to deploy your skill to a lambda function.

My First Podcast

In this example you will see how to implement a podcast skill by having a list of audios in a file (*podcast.js*) with titles and urls. It implements all audio intents allowed by the audio background feature and handles all the playback requests dispatched by Alexa once an audio has started, stopped, failed, finished or nearly to finish. Keep in mind the audios must be hosted in a secure server.

Account Linking

A more complex project that shows how to work with account linking and make responses using the model state. It uses serverless to deploy your account linking server and skill to lambda, create a dynamodb table to store your account linking and create an s3 bucket to store your static assets. It also has a gulp task to upload your assets to S3

CHAPTER 6

Links

- search

Models

Models are the data structure that holds the current state of your application, the framework doesn't make many assumptions on it and only requires have a `fromEvent` method that should initialize it based on the alexaEvent session attributes and a `serialize` method that should return `JSON.stringify` able structure to then store in the session attributes

```
'use strict';

const _ = require('lodash');

class Model {
  constructor(data) {
    _.assign(this, data);
  }

  static fromEvent(alexaEvent) {
    return new this(alexaEvent.session.attributes.data);
  }

  serialize() {
    return this;
  }
}

module.exports = Model;
```

Views and Variables

Views

Views are the Voxa way of handling replies to the user, they're templates of responses that can have a context as defined by your variables and Model

There are 5 responses in the following snippet: `LaunchIntent.OpenResponse`, `ExitIntent.Farewell`, `HelpIntent.HelpAboutSkill`, `Count.Say` and `Count.tell`

```
const views = {
  LaunchIntent: {
    OpenResponse: { tell: 'Hello! Good {time}' },
  },
  ExitIntent: {
    Farewell: { tell: 'Ok. For more info visit {site} site.' },
  },
  HelpIntent: {
    HelpAboutSkill: { tell: 'For more help visit www.rain.agency' },
  },
  Count: {
    Say: { say: '{count}' },
    Tell: { tell: '{count}' },
  },
};
```

Variables

Variables are the rendering engine way of adding logic into your views. They're dessigned to be very simple since most of your logic should be in your *model* or *controllers*.

A variable signature is:

`variable`(*model*, *alexaEvent*)

Arguments

- **`model`** – The instance of your *model* for the current alexa event.
- **`AlexaEvent`** – The current *alexa event*.

Returns The value to be rendered or a promise resolving to a value to be rendered in the view.

```
const variables = {
  site: function site(model) {
    return Promise.resolve('example.com');
  },

  count: function count(model) {
    return model.count;
  },

  locale: function locale(model, alexaEvent) {
    return alexaEvent.locale;
  }
};
```

Controllers

Controllers in your application control the logic of your skill, they respond to alexa alexaEvents, external resources, manipulate the input and give proper responses using your *Model*, *Views and Variables*.

States come in one of two ways, they can be an object of mappings from intent name to state.

```
skill.onState('entry', {
  LaunchIntent: 'launch',
  'AMAZON.HelpIntent': 'help',
});
```

Or they can be a function that gets a *alexaEvent* object.

```
skill.onState('launch', (alexaEvent) => {
  return { reply: 'LaunchIntent.OpenResponse', to: 'die' };
});
```

Your state should respond with a *transition*. The transition is a plain object that can take `directives`, `to` and `reply` keys.

The `entry` controller

The `entry` controller is special in that it's the default state to go to at the beginning of your session and if your state returns no response.

For example in the next snipped there's a `waiting` state that expects an `AMAZON.NextIntent` or an `AMAZON.PreviousIntent`, in the case the users says something unexpected like an `AMAZON.HelpIntent` the state returns undefined, the State Machine framework handles this situations by redirecting to the `entry` state

```
skill.onState('waiting', (alexaEvent) => {
  if (alexaEvent.intent.name === 'AMAZON.NextIntent') {
    alexaEvent.model.index += 1;
    return { reply: 'Ingredients.Describe', to: 'waiting' }
  } else if (alexaEvent.intent.name === 'AMAZON.PreviousIntent') {
    alexaEvent.model.index -= 1;
    return { reply: 'Ingredients.Describe', to: 'waiting' }
  }
});
```

The `onIntent` helper

For the simple pattern of having a controller respond to an specific intent the framework provides the `onIntent` helper

```
skill.onIntent('LaunchIntent', (alexaEvent) => {
  return { reply: 'LaunchIntent.OpenResponse', to: 'die' };
});
```

Under the hood this creates a new key in the `entry` controller and a new state

Transition

A transition is the result of controller execution, it's simple object with some keys that control the flow of execution in your skill.

to

The `to` key should be the name of state in your state machine, when present it indicates to the framework that it should move to a new state. If absent it's assumed that the framework should move to the `die` state.

```
return { to: 'stateName' };
```

directives

Directives are used passed directly to the alexa response, the format is described in the alexa documentation

```
return {
  directives: {
    type: 'AudioPlayer.Play',
    playBehavior: 'REPLACE_ALL',
    url: lesson.Url,
    offsetInMilliseconds: 0,
    },
};
```

reply

The `reply` key can take 2 forms, a simple string pointing to one of your views or a *Reply* object.

```
return { reply: 'LaunchIntent.OpenResponse' };

const reply = new Reply(alexaEvent, { tell: 'Hi there!' });
return { reply };
```

The `alexaEvent` Object

class `AlexaEvent` (*event*, *lambdaContext*)

The `alexaEvent` object contains all the information from the Alexa event, it's an object kept for the entire lifecycle of the state machine transitions and as such is a perfect place for middleware to put information that should be available on every request.

`AlexaEvent.`**`model`**

The default middleware instantiates a `Model` and makes it available through `alexaEvent.model`

`AlexaEvent.intent.`**`params`**

The alexaEvent object makes `intent.slots` available through `intent.params` after aplying a simple transformation so `{ slots: [{ name: 'Dish', value: 'Fried Chicken' }] }` becomes `{ Dish: 'Fried Chicken' }`

The `reply` Object

class **`Reply`**(*alexaEvent*[, *message*])

The `reply` object is used by the framework to render Alexa responses, it takes all of your `statements`, `cards` and `directives` and generates a proper json response for Alexa

Arguments

- **`alexaEvent`** (`AlexaEvent`) –
- **`message`** – A message object

`Reply`.**`append`**(*message*)

Adds statements to the `Reply`

Arguments

- **`message`** – An object with keys `ask`, `tell`, `say`, `reprompt`, `card` or `directives` keys. Or another `reply` object

Returns the `Reply` object

`Reply`.**`toJSON`**()

Returns An object with the proper format to send back to Alexa, with statements wrapped in SSML tags, cards, reprompts and directives

Voxa

class **`Voxa`**(*config*)

Arguments

- **`config`** – Configuration for your skill, it should include *Views and Variables* and optionally a *model* and a list of appIds.

If appIds is present then the framework will check every alexa event and enforce the application id to match one of the specified application ids.

```
const skill = new Voxa({ Model, variables, views, appIds });
```

`Voxa`.**`execute`**(*event*)

The main entry point for the Skill execution

Arguments

- **`event`** – The event sent by alexa.
- **`context`** – The context of the lambda function

Returns Promise A response resolving to a javascript object to be sent as a result to Alexa.

```
skill.execute(event, context)
  .then(context.succeed)
  .catch(context.fail);
```

`Voxa`.**`onState`**(*stateName*, *handler*)

Maps a handler to a state

Arguments

- **stateName** (*string*) – The name of the state
- **handler** (*function/object*) – The controller to handle the state

Returns An object or a promise that resolves to an object that specifies a transition to another state and/or a view to render

```
skill.onState('entry', {
  LaunchIntent: 'launch',
  'AMAZON.HelpIntent': 'help',
});

skill.onState('launch', (alexaEvent) => {
  return { reply: 'LaunchIntent.OpenResponse', to: 'die' };
});
```

`Voxa.`**onIntent**(*intentName*, *handler*)

A shortcut for defininining state controllers that map directly to an intent

Arguments

- **intentName** (*string*) – The name of the intent
- **handler** (*function/object*) – The controller to handle the state

Returns An object or a promise that resolves to an object that specifies a transition to another state and/or a view to render

```
skill.onIntent('HelpIntent', (alexaEvent) => {
  return { reply: 'HelpIntent.HelpAboutSkill' };
});
```

`Voxa.`**onIntentRequest**(*callback*[, *atLast*])

This is executed for all `IntentRequest` events, default behavior is to execute the State Machine machinery, you generally don't need to override this.

Arguments

- **callback** (*function*) –
- **last** (*bool*) –

Returns Promise

`Voxa.`**onLaunchRequest**(*callback*[, *atLast*])

Adds a callback to be executed when processing a `LaunchRequest`, the default behavior is to fake the *alexa event* as an `IntentRequest` with a `LaunchIntent` and just defer to the `onIntentRequest` handlers. You generally don't need to override this.

`Voxa.`**onBeforeStateChanged**(*callback*[, *atLast*])

This is executed before entering every state, it can be used to track state changes or make changes to the *alexa event* object

`Voxa.`**onBeforeReplySent**(*callback*[, *atLast*])

Adds a callback to be executed just before sending the reply, internally this is used to add the serialized model and next state to the session.

It can be used to alter the reply, or for example to track the final response sent to a user in analytics.

```
skill.onBeforeReplySent((alexaEvent, reply) => {
  const rendered = reply.write();
  analytics.track(alexaEvent, rendered)
});
```

`Voxa.`**`onAfterStateChanged`**(*callback*[, *atLast*])

Adds callbacks to be executed on the result of a state transition, this are called after every transition and internally it's used to render the *transition* `reply` using the *views and variables*

The callbacks get `alexaEvent`, `reply` and `transition` params, it should return the transition object

```
skill.onAfterStateChanged((alexaEvent, reply, transition) => {
  if (transition.reply === 'LaunchIntent.PlayTodayLesson') {
    transition.reply = _.sample(['LaunchIntent.PlayTodayLesson1', 'LaunchIntent.
↪PlayTodayLesson2']);
  }

  return transition;
});
```

`Voxa.`**`onUnhandledState`**(*callback*[, *atLast*])

Adds a callback to be executed when a state transition fails to generate a result, this usually happens when redirecting to a missing state or an entry call for a non configured intent, the handlers get a *alexa event* parameter and should return a *transition* the same as a state controller would.

`Voxa.`**`onSessionStarted`**(*callback*[, *atLast*])

Adds a callback to the `onSessinStarted` event, this executes for all events where `alexaEvent.session.new === true`

This can be useful to track analytics

```
skill.onSessionStarted((alexaEvent, reply) => {
  analytics.trackSessionStarted(alexaEvent);
});
```

`Voxa.`**`onRequestStarted`**(*callback*[, *atLast*])

Adds a callback to be executed whenever there's a `LaunchRequest`, `IntentRequest` or a `SessionEndedRequest`, this can be used to initialize your analytics or get your account linking user data. Internally it's used to initialize the model based on the event session

```
skill.onRequestStarted((alexaEvent, reply) => {
  alexaEvent.model = this.config.Model.fromEvent(alexaEvent);
});
```

`Voxa.`**`onSessionEnded`**(*callback*[, *atLast*])

Adds a callback to the `onSessionEnded` event, this is called for every `SessionEndedRequest` or when the skill returns a transition to a state where `isTerminal === true`, normally this is a transition to the `die` state. You would normally use this to track analytics

`Voxa.onSystem.`**`ExceptionEncountered`**(*callback*[, *atLast*])

This handles System.ExceptionEncountered event that are sent to your skill when a response to an `AudioPlayer` event causes an error

```
return Promise.reduce(errorHandlers, (result, errorHandler) => {
  if (result) {
    return result;
  }
  return Promise.resolve(errorHandler(alexaEvent, error));
}, null);
```

Error handlers

You can register many error handlers to be used for the different kind of errors the application could generate. They all follow the same logic where if the first error type is not handled then the default is to be deferred to the more general error handler that ultimately just returns a default error reply.

They're executed sequentially and will stop when the first handler returns a reply.

`Voxa.`**`onStateMachineError`**(*callback*[, *atLast*])

This handler will catch all errors generated when trying to make transitions in the stateMachine, this could include errors in the state machine controllers, , the handlers get `(alexaEvent, reply, error)` parameters

```
skill.onStateMachineError((alexaEvent, reply, error) => {
  // it gets the current reply, which could be incomplete due to an error.
  return new Reply(alexaEvent, { tell: 'An error in the controllers code' })
    .write();
});
```

`Voxa.`**`onError`**(*callback*[, *atLast*])

This is the more general handler and will catch all unhandled errors in the framework, it gets `(alexaEvent, error)` parameters as arguments

```
skill.onError((alexaEvent, error) => {
  return new Reply(alexaEvent, { tell: 'An unrecoverable error occurred.' })
    .write();
});
```

Playback Controller handlers

Handle events from the AudioPlayer interface

`audioPlayerCallback`(*alexaEvent*, *reply*)

All audio player middleware callbacks get a *alexa event* and a *reply* object

Arguments

- **`alexaEvent`** (`AlexaEvent`) – The *alexa event* sent by Alexa
- **`reply`** (*`object`*) – A reply to be sent as a response

Returns object write Your alexa event handler should return an appropriate response according to the event type, this generally means appending to the *reply* object

In the following example the alexa event handler returns a `REPLACE_ENQUEUED` directive to a *`PlaybackNearlyFinished()`* event.

```
skill['onAudioPlayer.PlaybackNearlyFinished']((alexaEvent, reply) => {
  const directives = {
    type: 'AudioPlayer.Play',
    playBehavior: 'REPLACE_ENQUEUED',
    token: "",
    url: 'https://www.dl-sounds.com/wp-content/uploads/edd/2016/09/Classical-Bed3-
↪preview.mp3',
    offsetInMilliseconds: 0,
  };

  return reply.append({ directives });
});
```

Voxa.onAudioPlayer.**PlaybackStarted**(*callback*[, *atLast*])

Voxa.onAudioPlayer.**PlaybackFinished**(*callback*[, *atLast*])

Voxa.onAudioPlayer.**PlaybackStopped**(*callback*[, *atLast*])

Voxa.onAudioPlayer.**PlaybackFailed**(*callback*[, *atLast*])

Voxa.onAudioPlayer.**PlaybackNearlyFinished**(*callback*[, *atLast*])

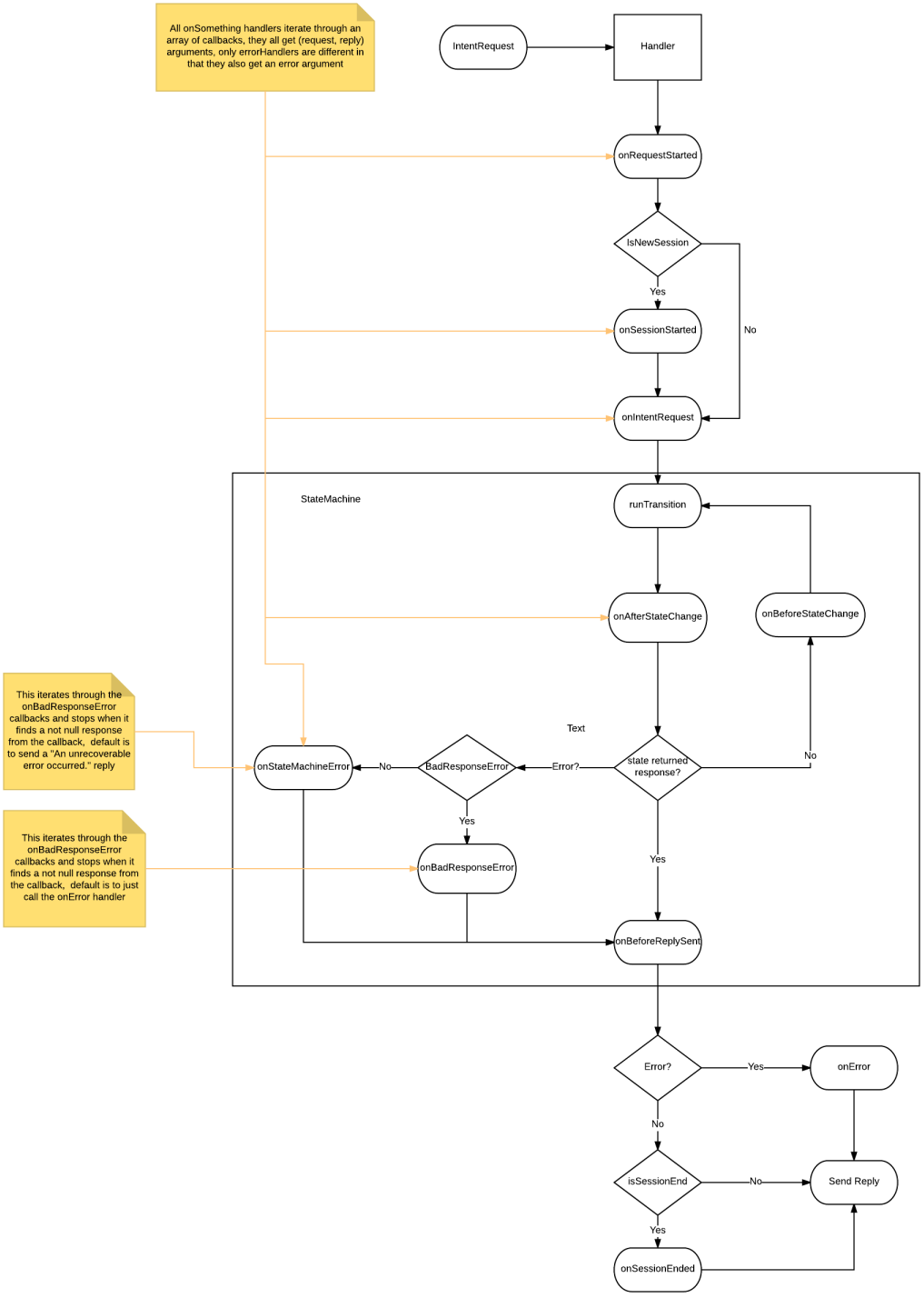
Voxa.onPlaybackController.**NextCommandIssued**(*callback*[, *atLast*])

Voxa.onPlaybackController.**PauseCommandIssued**(*callback*[, *atLast*])

Voxa.onPlaybackController.**PlayCommandIssued**(*callback*[, *atLast*])

Voxa.onPlaybackController.**PreviousCommandIssued**(*callback*[, *atLast*])

Request Flow



I18N

Internationalization support is done using the i18next library, the same the Amazon Alexa Node SDK uses.

Configuring the skill for I18N

To use it you need to configure your skill to use the `I18NRenderer` class instead of the default renderer class.

```
const Voxa = require('voxa');
const skill = new Voxa({ Model, variables, views, RenderClass: Voxa.I18NRenderer });
```

The framework takes care of selecting the correct locale on every alexa event by looking at the `alexaEvent.request.locale` property.

Changes in your views

The other change you will need is to define your views using the i18next translate format:

```
'use strict';

const views = (function views() {
  return {
    'en-us': {
      translation: {
        LaunchIntent: {
          OpenResponse: { tell: 'Hello! Good {time}' },
        },
        Question: {
          Ask: { ask: 'What time is it?' },
        },
        ExitIntent: {
          Farewell: { tell: 'Ok. For more info visit {site} site.' },
        },
        Number: {
          One: { tell: '{numberOne}' },
        },
      },
    },
    'de-de': {
      translation: {
        LaunchIntent: {
          OpenResponse: { tell: 'Hallo! guten {time}' },
        },
        Question: {
          Ask: { ask: 'wie spät ist es?' },
        },
        ExitIntent: {
          Farewell: { tell: 'Ok für weitere Infos besuchen {site} Website' },
        },
        Number: {
          One: { tell: '{numberOne}' },
        },
      },
    },
  };
```

```
}());

module.exports = views;
```

Variables

Variables should work mostly the same as with the `DefaultRenderer`, with the exception that variables will now get a `locale` key

```
'use strict';

/**
 * Variables for tests
 *
 * Copyright (c) 2016 Rain Agency.
 * Licensed under the MIT license.
 */

const Promise = require('bluebird');

const variables = {
  time: function time() {
    const today = new Date();
    const curHr = today.getHours();

    if (curHr < 12) {
      return Promise.resolve('Morning');
    }
    if (curHr < 18) {
      return Promise.resolve('Afternoon');
    }
    return Promise.resolve('Evening');
  },

  site: function site() {
    return Promise.resolve('example.com');
  },

  count: function count(model) {
    return model.count;
  },

  numberOne: function numberOne(model, request) {
    if (request.request.locale === 'en-us') {
      return 'one';
    } else if (request.request.locale === 'de-de') {
      return 'ein';
    }

    return 1;
  },
};

module.exports = variables;
```

Plugins

Plugins allow you to modify how the StateMachineSkill handles an alexa event. When a plugin is registered it will use the different hook points in your skill to add functionality. If you have several skills with similar behavior then your answer is to create a plugin.

Using a plugin

After instatiating a StateMachineSkill you can register plugins on it. Built in plugins can be accessed through `Voxa.plugins`

```
'use strict';
const Voxa = require('voxa');
const Model = require('./model');
const views = require('./views'):
const variables = require('./variables');

const skill = new Voxa({ Model, variables, views });

Voxa.plugins.replaceIntent(skill);
```

State Flow plugin

Stores the state transitions for every alexa event in an array.

stateFlow(*skill*)

State Flow attaches callbacks to *onRequestStarted()*, *onBeforeStateChanged()* and *onBeforeReplySent()* to track state transitions in a `alexaEvent.flow` array

Arguments

- **skill** (Voxa) – The skill object

Usage

```
const alexa = require('alexa-statemachine');
alexa.plugins.stateFlow.register(skill)

skill.onBeforeReplySent((alexaEvent) => {
  console.log(alexaEvent.flow.join(' > ')); // entry > firstState > secondState > die
});
```

Replace intent plugin

It allows you to rename an intent name based on a regular expression. By default it will match `/(.*)OnlyIntent$/` and replace it with `$1Intent`.

replaceIntent(*skill*[, *config*])

Replace Intent plugin uses *onIntentRequest()* to modify the incomming request intent name

Arguments

- **skill** (Voxa) – The stateMachineSkill

- **config** – An object with the `regex` to look for and the `replace` value.

Usage

```
const skill = new Voxa({ Model, variables, views });

Voxa.plugins.replaceIntent(skill, { regex: /(.*)OnlyIntent$/, replace: '$1Intent' });
Voxa.plugins.replaceIntent(skill, { regex: /^VeryLong(.*)/, replace: 'Long$1' });
```

Why onlyIntents?

A good practice is to isolate an utterance into another intent if it's contain a single slot. By creating the only intent, alexa will prioritize this intent if the user says only the slot.

Let's explain with the following scenario. You need the user to provide a zipcode. so you should have an *intent* called `ZipCodeIntent`. But you still have to manage if the user only says its zipcode with no other words on it. So that's when we create an OnlyIntent. Let's called `ZipCodeOnlyIntent`.

Our utterance file will be like this:

```
ZipCodeIntent here is my {ZipCodeSlot}
ZipCodeIntent my zip is {ZipCodeSlot}
...

ZipCodeOnlyIntent {ZipCodeSlot}
```

But now we have two states which are basically the same. Replace intent plugin will rename all incoming requests intents from `ZipCodeOnlyIntent` to `ZipCodeIntent`.

Cloudwatch plugin

It logs a CloudWatch metric when the skill catches an error.

Params

cloudwatch (*skill*, *cloudwatch*[, *eventMetric*])

Cloudwatch plugin uses `skill.onError` to log a metric

Arguments

- **skill** (Voxa) – The stateMachineSkill
- **cloudwatch** – A new AWS.CloudWatch object.
- **putMetricDataParams** – Params for putMetricData

How to use it

```
const AWS = require('aws-sdk');
const skill = new Voxa({ Model, variables, views });

const cloudwatch = new AWS.CloudWatch({});
const eventMetric = { Namespace: 'fooBarSkill' };
```

```
Voxa.plugins.cloudwatch(skill, cloudwatch, eventMetric);
```

Debugging

Voxa uses the debug module internally to log a number of different internal events, if you want have a look at those events you have to declare the following environment variable

`DEBUG=voxa`

This is an example of the log output

```
voxa Received new event: {"version":"1.0","session":{"new":true,"sessionId":
↪"SessionId.09162f2a-cf8f-414f-92e6-1e3616ecaa05","application":{"applicationId":
↪"amzn1.ask.skill.1fe77997-14db-409b-926c-0d8c161e5376"},"attributes":{},"user":{
↪"userId":"amzn1.ask.account.","accessToken":""}},"request":{"type":"LaunchRequest",
↪"requestId":"EdwRequestId.0f7b488d-c198-4374-9fb5-6c2034a5c883","timestamp":"2017-
↪01-25T23:01:15Z","locale":"en-US"}} +0ms
voxa Initialized model like {} +8ms
voxa Starting the state machine from entry state +2s
voxa Running simpleTransition for entry +1ms
voxa Running onAfterStateChangeCallbacks +0ms
voxa entry transition resulted in {"to":"launch"} +0ms
voxa Running launch enter function +1ms
voxa Running onAfterStateChangeCallbacks +0ms
voxa launch transition resulted in {"reply":"Intent.Launch","to":"entry","message":{
↪"tell":"Welcome mail@example.com!"},"session":{"data":{},"reply":null}} +7ms
```

Starter Kit

This project is designed to be a simple template for your new skills. With some well thought defaults that have proven useful when developing real life skills.

Directory Structure

It has the following directory structure

```
.
- README.md
- config
|   - env.js
|   - index.js
|   - local.json.example
|   - production.json
|   - staging.json
- gulpfile.js
- package.json
- serverless.yml
- services
- skill
|   - MainStateMachine.js
|   - index.js
```

```
|   - variables.js
|   - views.js
- speechAssets
|   - IntentSchema.json
|   - SampleUtterances.txt
|   - customSlotTypes
- test
- www
    - infrastructure
    |   - mount.js
    - routes
    |   - index.js
    |   - skill.js
    - server.js
```

config

By default your skill will have the following environments:

- local
- staging
- production

What environment you're is determined in the `config/env.js` module using the following code:

```
'use strict';

function getEnv() {
  if (process.env.NODE_ENV) return process.env.NODE_ENV;
  if (process.env.AWS_LAMBDA_FUNCTION_NAME) {
    // TODO put your own lambda function name here
    if (process.env.AWS_LAMBDA_FUNCTION_NAME === '') return 'production';
    return 'staging';
  }

  return 'local';
}

module.exports = getEnv();
```

skill

This is where your code to handle alexa events goes, you will usually have a State Machine definition, this will include *states*, *middleware* and a *Model*, *Views and Variables*

speechAssets

This should be a version controlled copy of your intent schema, sample utterrances and custom slots.

www

A standard express project configured to serve your skill in the `/skill` route. Combined with ngrok this is a great tool when developing or debugging.

services

Just a common place to put models and libraries

test

You write tests right?

gulpfile

A gulp runner configured with a watch task that starts your express server and listens for changes to reload your application.

serverless.yml

The serverless framework is a tool that helps you manage your lambda applications, assuming you have your AWS credentials setup properly this starter kit defines the very minimum needed so you can deploy your skill to lambda with the following command:

```
$ sls deploy
```

Running the project

1. Clone the voxa repository
2. Create a new skill project using the `samples/starterKit` directory as a basis
3. Make sure you're running node 4.3, this is easiest with nvm
4. Create a `config/local.json` file using `config/local.json.example` as an example
5. Run the project with `gulp watch`
6. At this point you should start `ngrok http 3000` and configure your skill in the Amazon Developer panel to use the ngrok https endpoint.

My First Podcast

This project will help you build a podcast skill using the Audio directives template. You will be able to manage loop, shuffle requests as well as offer the user the possibility to start an audio over, pause, stop it or play the next or previous audio from a podcast list.

Directory Structure

It has the following directory structure

```
.
- README.md
- config
|   - env.js
|   - index.js
|   - local.json.example
|   - production.json
|   - staging.json
- gulpfile.js
- package.json
- serverless.yml
- services
- skill
|   - data
|   |   - podcast.js
|   - MainStateMachine.js
|   - index.js
|   - states.js
|   - variables.js
|   - views.js
- speechAssets
|   - IntentSchema.json
|   - SampleUtterances.txt
|   - customSlotTypes
- test
- www
    - infrastructure
    |   - mount.js
    - routes
    |   - index.js
    |   - skill.js
    - server.js
```

config

By default your skill will have the following environments:

- local
- staging
- production

What environment you're is determined in the `config/env.js` module using the following code:

```
'use strict';

function getEnv() {
  if (process.env.NODE_ENV) return process.env.NODE_ENV;
  if (process.env.AWS_LAMBDA_FUNCTION_NAME) {
    // TODO put your own lambda function name here
    if (process.env.AWS_LAMBDA_FUNCTION_NAME === '') return 'production';
    return 'staging';
  }
```

```
  return 'local';
}

module.exports = getEnv();
```

skill

index.js

First file invoked by the lambda function, it initializes the state machine. You don't need to modify this file.

MainStateMachine.js

State machine is initialized with your model, views and variables. The class *states.js* will be in charge to handle all intents and events coming from Alexa. You don't need to modify this file.

states.js

All events and intents dispatched by the Alexa Voice Service to your skill are handled here. You can integrate any other module or API calls to third party services, call database resources or just simply reply a Hello or Goodbye response to the user.

The audio intents handled in this example are:

- AMAZON.CancelIntent
- AMAZON.LoopOffIntent
- AMAZON.LoopOnIntent
- AMAZON.NextIntent
- AMAZON.PauseIntent
- AMAZON.PreviousIntent
- AMAZON.RepeatIntent
- AMAZON.ResumeIntent
- AMAZON.ShuffleOffIntent
- AMAZON.ShuffleOnIntent
- AMAZON.StartOverIntent

You can track the values for loop, shuffle and current URL playing in the token property of the Alexa event in the path *alexaEvent.context.AudioPlayer.token*:

```
skill.onState('loopOff', (alexaEvent) => {
        if (alexaEvent.context) {
          const token = JSON.parse(alexaEvent.context.AudioPlayer.token);
          const shuffle = token.shuffle;
          const loop = 0;
          const offsetInMilliseconds = alexaEvent.context.AudioPlayer.
↪offsetInMilliseconds;
          let index = token.index;
```

```
          if (index === podcast.length) {
            index = 0;
          }

          const directives = buildPlayDirective(podcast[index].url, index, shuffle,␣
↪loop, offsetInMilliseconds);

          return { reply: 'Intent.LoopDeactivated', to: 'die', directives };
        }

        return { reply: 'Intent.Exit', to: 'die' };
});
```

For any of these events you can make Alexa to speak after user's action with a reply object, optionally you can define the *die* state and pass through the directives object with either a *AudioPlayer.Play* or *AudioPlayer.Stop* directive type.

You can also handled the following playback request events:

- AudioPlayer.PlaybackStarted
- AudioPlayer.PlaybackFinished
- AudioPlayer.PlaybackStopped
- AudioPlayer.PlaybackNearlyFinished
- AudioPlayer.PlaybackFailed

You're not allowed to respond with a reply object since it's just an event most for trackign purposes, so it's optional to implement and you can do the following syntax:

```
skill['onAudioPlayer.PlaybackStarted']((alexaEvent) => {
        console.log('onAudioPlayer.PlaybackStarted', JSON.stringify(alexaEvent, null,␣
↪2));
});
```

In case the user has activated the loop mode by dispatching the *AMAZON.LoopOnIntent* intent, you can implement a queue list in the *AudioPlayer.PlaybackNearlyFinished* this way:

```
skill['onAudioPlayer.PlaybackNearlyFinished']((alexaEvent, reply) => {
        const token = JSON.parse(alexaEvent.context.AudioPlayer.token);

        if (token.loop === 0) {
          return reply;
        }

        const shuffle = token.shuffle;
        const loop = token.loop;
        let index = token.index + 1;

        if (shuffle === 1) {
          index = randomIntInc(0, podcast.length - 1);
        } else if (index === podcast.length) {
          index = 0;
        }

        const directives = buildEnqueueDirective(podcast[index].url, index, shuffle,␣
↪loop);
        return reply.append({ directives });
```

```
});

function buildEnqueueDirective(url, index, shuffle, loop) {
        const directives = {};
        directives.type = 'AudioPlayer.Play';
        directives.playBehavior = 'REPLACE_ENQUEUED';
        directives.token = createToken(index, shuffle, loop);
        directives.url = podcast[index].url;
        directives.offsetInMilliseconds = 0;

        return directives;
}
```

The *buildEnqueueDirective* function is in charge to build a directive object with a queue behavior, which will allow the skill to play the next audio as soon as the current one is finished.

This is where your code to handle alexa events goes, you will usually have a State Machine definition, this will include *states*, *middleware* and a *Model*, *Views and Variables*.

data/podcast.js

A JSON variable with titles and urls for 5 audio examples hosted in a secure server, all along play a podcast which the user can shuffle or loop. You can modify this file with whatever other audio to add to your playlist. Keep in mind that they must be hosted in a secure server. The supported formats for the audio file include AAC/MP4, MP3, HLS, PLS and M3U. Bitrates: 16kbps to 384 kbps.

speechAssets

This should be a version controlled copy of your intent schema, sample utterrances and custom slots.

www

A standard express project configured to serve your skill in the `/skill` route. Combined with ngrok this is a great tool when developing or debugging.

services

Just a common place to put models and libraries

test

You write tests right?

gulpfile

A gulp runner configured with a watch task that starts your express server and listens for changes to reload your application.

serverless.yml

The serverless framework is a tool that helps you manage your lambda applications, assuming you have your AWS credentials setup properly this starter kit defines the very minimum needed so you can deploy your skill to lambda with the following command:

```
$ sls deploy
```

Running the project

1. Clone the voxa repository
2. Create a new skill project using the `samples/my-first-podcast` directory as a basis
3. Make sure you're running node 4.3, this is easiest with nvm
4. Create a `config/local.json` file using `config/local.json.example` as an example
5. Run the project with `gulp watch`
6. Create a skill in your Amazon Developer Portal account under the ALEXA menu.
7. Go to the interaction model tab and copy the intent schema and utterances from the the speechAssets folder.
8. At this point you should start `ngrok http 3000` and configure your skill in the Amazon Developer panel to use the ngrok https endpoint.

Account Linking

This project is designed to be a simple template for your new skills with account linking. User's information is stored in a DynamoDB table so you can fetch it from the skill once users are authenticated.

Directory Structure

It has the following directory structure

```
.
- README.md
- config
|   - env.js
|   - index.js
|   - local.json.example
|   - production.json
|   - staging.json
- gulpfile.js
- package.json
- serverless.yml
- services
|   - model.js
|   - userStorage.js
- skill
|   - MainStateMachine.js
|   - index.js
|   - states.js
|   - variables.js
```

```
|   - views.js
- speechAssets
|   - IntentSchema.json
|   - SampleUtterances.txt
|   - customSlotTypes
- test
- www
    - infrastructure
    |   - mount.js
    - routes
    |   - index.js
    |   - skill.js
    - server.js
```

config

By default your skill will have the following environments:

- local
- staging
- production

What environment you're is determined in the `config/env.js` module using the following code:

```
'use strict';

function getEnv() {
  if (process.env.NODE_ENV) return process.env.NODE_ENV;
  if (process.env.AWS_LAMBDA_FUNCTION_NAME) {
    // TODO put your own lambda function name here
    if (process.env.AWS_LAMBDA_FUNCTION_NAME === '') return 'production';
    return 'staging';
  }

  return 'local';
}

module.exports = getEnv();
```

skill

index.js

First file invoked by the lambda function, it initializes the state machine. You don't need to modify this file.

MainStateMachine.js

State machine is initialized with your model, views and variables. The class *states.js* will be in charge to handle all intents and events coming from Alexa. You don't need to modify this file.

states.js

All events and intents dispatched by the Alexa Voice Service to your skill are handled here. You can integrate any other module or API calls to third party services, call database resources or just simply reply a Hello or Goodbye response to the user. Before the very beginning of the lesson, you can implement the method *onRequestStarted* to fetch user's data from DynamoDB based on the accessToken coming from Alexa

```
skill.onRequestStarted((alexaEvent) => {
  if (!alexaEvent.session.user.accessToken) {
    return alexaEvent;
  }
  const storage = new UserStorage();

  return storage.get(alexaEvent.session.user.accessToken)
    .then((user) => {
      alexaEvent.model.user = user;
      return alexaEvent;
    });
});
```

If the user is not authenticated you can also send a *LinkingAccount* card to the Alexa app so users know that before using your skill, they must get authenticated.

speechAssets

This should be a version controlled copy of your intent schema, sample utterrances and custom slots.

www

A standard express project configured to serve your skill in the `/skill` route. Combined with ngrok this is a great tool when developing or debugging.

routes/index.js

You can handle all GET and POST requests for your account linking projects here. The most common one will be the POST call of the form after users hit the submit button. In this example, we gather user's information and create a row in DynamoDB for their information. For example you can generate an UUID to identify the users as the primary key and send it back to Alexa as the accessToken so you can easily fetch user's information later on.

```
router.post('/', (req, res, next) => {
  const md = new MobileDetect(req.headers['user-agent']);
  const db = new Storage();
  const email = req.body.email;
  const code = uuidV4().replace(/-/g, '');

  const params = {
    id: code,
    email,
  };

  return db.put(params)
    .then(() => {
      const redirect = `${req.query.redirect_uri}#state=${req.query.state}&access_
↪token=${code}&token_type=Bearer`;
```

```
      if (md.is('AndroidOS')) {
        console.log(`redirecting android to: ${redirect}`);
        res.redirect(redirect);
      } else {
        console.log(`redirecting web to: ${redirect}`);
        res.render('auth/success', {
          page: 'success',
          title: 'Success',
          redirectUrl: redirect,
        });
      }
    })
    .catch(next);
});
```

To finish the authentication process you have to make a redirection to the *redirect_uri* Amazon sends to our service. Since there could be 2 origins to redirect to, we create this URL dynamically; these endpoints could look like this:

- https://pitangui.amazon.com/spa/skill/account-linking-status.html?vendorId=xxx -> For United States store
- https://layla.amazon.com/spa/skill/account-linking-status.html?vendorId=xxxxxx -> For UK and Germany store

The other parameters to send are:

- access_token=YOUR-TOKEN
- token_type=Bearer

services

Just a common place to put models and libraries

userStorage.js

Use this file as an example to handle database logic. Since we use DynamoDB for this example, we included 2 methods, a put and a get, so user's information get stored from the account linking project and get fetched from the alexa skill side. For reaching out DynamoDB you need some permissions for your lambda function. Make sure to grant your lambda function with a role with DynamoDB access.

test

You write tests right?

gulpfile

A gulp runner configured with a watch task that starts your express server and listens for changes to reload your application.

serverless.yml

The serverless framework is a tool that helps you manage your lambda applications, assuming you have your AWS credentials setup properly this starter kit defines the very minimum needed so you can deploy your skill to lambda with the following command:

```
$ sls deploy
```

Running the project

1. Clone the voxa repository
2. Create a new skill project using the `samples/starterKit` directory as a basis
3. Make sure you're running node 4.3, this is easiest with nvm
4. Create a `config/local.json` file using `config/local.json.example` as an example
5. Run the project with `gulp watch`
6. At this point you should start `ngrok http 3000` and configure your skill in the Amazon Developer panel to use the ngrok https endpoint.

Index