Voxa Documentation

Release 3.0.0

Rain Agency

May 28, 2020

Contents:

CHAPTER 1

Summary

Voxa is an Alexa skill framework that provides a way to organize a skill into a state machine. Even the most complex voice user interface (VUI) can be represented through the state machine and it provides the flexibility needed to both be rigid when needed in specific states and flexible to jump around when allowing that also makes sense.

CHAPTER 2

Why Voxa vs other frameworks

Voxa provides a more robust framework for building Alexa skills. It provides a design pattern that wasn't found in other frameworks. Critical to Voxa was providing a pluggable interface and supporting all of the latest ASK features.

CHAPTER 3

Features

- MVC Pattern
- State or Intent handling (State Machine)
- Easy integration with several Analytics providers
- Easy to modify response file (the view)
- Compatibility with all SSML features
- Works with companion app cards
- Supports i18n in the responses
- Clean code structure with a unit testing framework
- Easy error handling
- Account linking support
- Several Plugins

CHAPTER 4

Installation

Voxa is distributed via `npm`

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

CHAPTER 5

Initial Configuration

Instantiating a Voxa Application requires a configuration specifying your *Views and Variables*.

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

const app = new voxa.VoxaApp({ variables, views });
```

CHAPTER 6

Platforms

Once you have instantiated a platform is time to create a plaform application. There are platform handlers for Alexa, Dialogflow (Google Assistant and Facebook Messenger) and Botframework (Cortana);

```
const alexaSkill = new voxa.AlexaPlatform(app);
const googleAction = new voxa.GoogleAssistantPlatform(app);
const facebookBot = new voxa.FacebookPlatform(app);

// botframework requires some extra configuration like the Azure Table Storage to use␣
↪and the Luis.ai endpoint
const storageName = config.cortana.storageName;
const tableName = config.cortana.tableName;
const storageKey = config.cortana.storageKey; // Obtain from Azure Portal
const azureTableClient = new azure.AzureTableClient(tableName, storageName,␣
↪storageKey);
const tableStorage = new azure.AzureBotStorage({ gzipData: false }, azureTableClient);
const botframeworkSkill = new voxa.BotFrameworkPlatform(app, {
  storage: tableStorage,
  recognizerURI: config.cortana.recognizerURI,
  applicationId: config.cortana.applicationId,
  applicationPassword: config.cortana.applicationPassword,
  defaultLocale: 'en',
});
```

CHAPTER 7

Using the development server

The framework provides a simple builtin server that's configured to serve all POST requests to your skill, this works great when developing, specially when paired with ngrok

```
// this will start an http server listening on port 3000
alexaSkill.startServer(3000);
```

CHAPTER 8

Responding to an intent event

```
app.onIntent('HelpIntent', (voxaEvent) => {
  return { tell: 'HelpIntent.HelpAboutSkill' };
});

app.onIntent('ExitIntent', (voxaEvent) => {
  return { tell: 'ExitIntent.Farewell' };
});
```

CHAPTER 9

Responding to lambda requests

Once you have your skill configured creating a lambda handler is as simple using the *alexaSkill.lambda* method

```
exports.handler = alexaSkill.lambda();
```

CHAPTER 10

Links

- search

10.1 New Alexa developer

If the skills development for alexa is a new thing for you, we have some suggestion to get you deep into this world.

10.1.1 Getting Started with the Alexa Skills

Alexa provides a set of built-in capabilities, referred to as skills. For example, Alexa's abilities include playing music from multiple providers, answering questions, providing weather forecasts, and querying Wikipedia.

The Alexa Skills Kit lets you teach Alexa new skills. Customers can access these new abilities by asking Alexa questions or making requests. You can build skills that provide users with many different types of abilities. For example, a skill might do any one of the following:

- Look up answers to specific questions ("Alexa, ask tide pooler for the high tide today in Seattle.")
- Challenge the user with puzzles or games ("Alexa, play Jeopardy.")
- Control lights and other devices in the home ("Alexa, turn on the living room lights.")
- Provide audio or text content for a customer's flash briefing ("Alexa, give me my flash briefing")

You can see the different types of skills here to got more deep reference.

How users interact with Alexa?

With Interaction Model.

End users interact with all of Alexa's abilities in the same way – by waking the device with the wake word (or a button for a device such as the Amazon Tap) and asking a question or making a request.

For example, users interact with the built-in Weather service like this:

User: Alexa, what's the weather? Alexa: Right now in Seattle, there are cloudy skies...

In the context of Alexa, an interaction model is somewhat analogous to a graphical user interface in a traditional app. Instead of clicking buttons and selecting options from dialog boxes, users make their requests and respond to questions by voice.

Here you can see how the interaction model works

10.1.2 Amazon Developer Service Account

Amazon Web Services provides a suite of solutions that enable developers and their organizations to leverage Amazon.com's robust technology infrastructure and content via simple API calls.

The first thing you need to do is create your own Amazon Developer Account.

10.1.3 Registering an Alexa skill

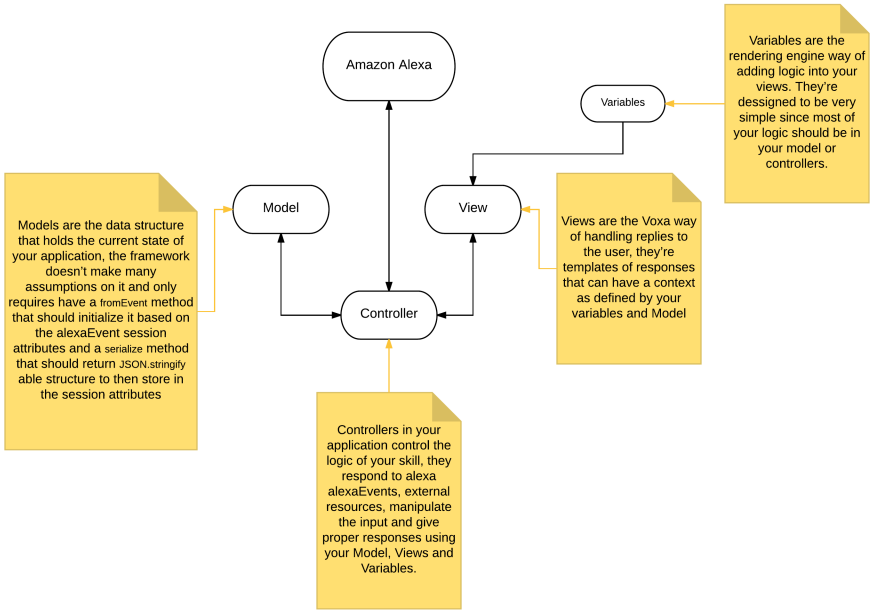
Registering a new skill or ability on the Amazon Developer Portal creates a configuration containing the information that the Alexa service needs to do the following:

- Route requests to the AWS Lambda function or web service that implements the skill, or for development purpose you can run it locally using ngrok.
- Display information about the skill in the Amazon Alexa App. The app shows all published skills, as well as all of your own skills currently under development.

You must register a skill before you can test it with the Service Simulator in the developer portal or an Alexa-enabled device.

Follow these instructions to register and managing your Alexa skill.

10.2 Voxa architecture pattern MVC



10.3 Voxa Application

class VoxaApp(*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 app = new VoxaApp.pp({ Model, variables, views, appIds });
```

VoxaApp.**execute**(*event*, *context*)

The main entry point for the Skill execution

Arguments

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

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

```
app.execute(event, context)
  .then(result => callback(null, result))
  .catch(callback);
```

VoxaApp.**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
- **intent** (*string/array*) – The intents that this state will handle

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

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

app.onState('launch', (voxaEvent) => {
  return { tell: 'LaunchIntent.OpenResponse', to: 'die' };
});
```

Also you can use a shorthand version to define a controller. This is very useful when having a controller that only returns a *transition*

```
voxaApp.onState('launch',
  {
    flow: 'yield'
    reply: 'LaunchIntent.OpenResponse',
    to: 'nextState'
  }
);
```

You can also set the intent that the controller will handle. If set, any other triggered intent will not enter into the controller.

```
voxaApp.onState("agreed?", {
  to: "PurchaseAccepted"
}, "YesIntent");

voxaApp.onState("agreed?", {
  to: "TransactionCancelled"
}, ["NoIntent", "CancelIntent"]);

voxaApp.onState("agreed?", {
  to: "agreed?",
  reply: "Help.ArticleExplanation",
  flow: "yield"
}, "HelpIntent");

voxaApp.onState("agreed?", {
  to: "agreed?",
  reply: "UnknownInput",
```

(continues on next page)

(continued from previous page)

```
  flow: "yield"
});
```

The order on how you declare your controllers matter in Voxa

You can set multiple *controllers* for a single state, so how do you know which code will be executed? The first one that Voxa finds. Take this example:

```
voxaApp.onState('ProcessUserRequest', (voxaEvent) => {
  // Some code
  return { tell: 'ThankYouResponse', to: 'die' };
});
voxaApp.onState('ProcessUserRequest', (voxaEvent) => {
  // Some other code
  return { tell: 'GoodbyeResponse', to: 'die' };
});
```

If the state machine goes to the *ProcessUserRequest*, the code running will always be the first one, so the user will always hear the *ThankYouResponse*.

The only scenario where this is overwritten is when you have more than one handler for the same state, and one of them has one or more intents defined. If the user triggers the intent that's inside the list of one-controller intents, Voxa will give it priority. For example, take this code:

```
voxaApp.onState("agreed?", {
  to: "PurchaseAccepted"
}, "YesIntent");

voxaApp.onState("agreed?", {
  to: "agreed?",
  reply: "UnknownInput",
  flow: "yield"
});

voxaApp.onState("agreed?", {
  to: "TransactionCancelled"
}, ["NoIntent", "CancelIntent"]);
```

If the user triggers the *NoIntent*, and the state machine goes to the *agreed?* state, the user will listen to the *TransactionCancelled* response, it doesn't matter if the controller is placed above or below a controller without defined intents, the priority will go to the controller with the defined intent.

`VoxaApp.`**`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

```
app.onIntent('HelpIntent', (voxaEvent) => {
  return { tell: 'HelpIntent.HelpAboutSkill' };
});
```

VoxaApp.**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

VoxaApp.**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.

VoxaApp.**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

VoxaApp.**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.

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

VoxaApp.**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 `voxaEvent`, `reply` and `transition` params, it should return the transition object

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

  return transition;
});
```

VoxaApp.**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.

VoxaApp.**onSessionStarted**(*callback*[, *atLast*])

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

This can be useful to track analytics

```
app.onSessionStarted((voxaEvent, reply) => {
  analytics.trackSessionStarted(voxaEvent);
});
```

`VoxaApp.`**`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

```
app.onRequestStarted((voxaEvent, reply) => {
  let data = ... // deserialized from the platform's session
  voxaEvent.model = this.config.Model.deserialize(data, voxaEvent);
});
```

`VoxaApp.`**`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

`VoxaApp.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(voxaEvent, error));
}, null);
```

10.3.1 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.

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

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

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

10.3.2 Playback Controller handlers

Handle events from the AudioPlayer interface

`audioPlayerCallback`(*voxaEvent*, *reply*)

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

Arguments

- **`voxaEvent`** (`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.

```
app['onAudioPlayer.PlaybackNearlyFinished']((voxaEvent, reply) => {
  const playAudio = new PlayAudio({
    behavior: "REPLACE_ALL",
    offsetInMilliseconds: 0,
    token: "",
    url: 'https://www.dl-sounds.com/wp-content/uploads/edd/2016/09/Classical-Bed3-
↪preview.mp3'
  });

  playAudio.writeToReply(reply);

  return reply;
});
```

`VoxaApp.onAudioPlayer.`**`PlaybackStarted`**(*callback*[, *atLast*])

`VoxaApp.onAudioPlayer.`**`PlaybackFinished`**(*callback*[, *atLast*])

`VoxaApp.onAudioPlayer.`**`PlaybackStopped`**(*callback*[, *atLast*])

`VoxaApp.onAudioPlayer.`**`PlaybackFailed`**(*callback*[, *atLast*])

`VoxaApp.onAudioPlayer.`**`PlaybackNearlyFinished`**(*callback*[, *atLast*])

`VoxaApp.onPlaybackController.`**`NextCommandIssued`**(*callback*[, *atLast*])

`VoxaApp.onPlaybackController.`**`PauseCommandIssued`**(*callback*[, *atLast*])

`VoxaApp.onPlaybackController.`**`PlayCommandIssued`**(*callback*[, *atLast*])

`VoxaApp.onPlaybackController.`**`PreviousCommandIssued`**(*callback*[, *atLast*])

10.3.3 Alexa Skill Event handlers

Handle request for the Alexa Skill Events

`alexaSkillEventCallback`(*alexaEvent*)

All the alexa skill event 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 the response

Returns object reply Alexa only needs an acknowledgement that you received and processed the event so it doesn't need to resend the event. Just returning the *reply* object is enough

This is an example on how your skill can process a *`SkillEnabled()`* event.

```
app['onAlexaSkillEvent.SkillEnabled']((alexaEvent, reply) => {
  const userId = alexaEvent.user.userId;
  console.log(`skill was enabled for user: ${userId}`);
  return reply;
});
```

VoxaApp.onAlexaSkillEvent.**SkillAccountLinked**(*callback*[, *atLast*])

VoxaApp.onAlexaSkillEvent.**SkillEnabled**(*callback*[, *atLast*])

VoxaApp.onAlexaSkillEvent.**SkillDisabled**(*callback*[, *atLast*])

VoxaApp.onAlexaSkillEvent.**SkillPermissionAccepted**(*callback*[, *atLast*])

VoxaApp.onAlexaSkillEvent.**SkillPermissionChanged**(*callback*[, *atLast*])

10.3.4 Alexa List Event handlers

Handle request for the Alexa List Events

alexaListEventCallback(*alexaEvent*)

All the alexa list event 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 the response

Returns object reply Alexa only needs an acknowledgement that you received and processed the event so it doesn't need to resend the event. Just returning the *reply* object is enough

This is an example on how your skill can process a *ItemsCreated()* event.

```
app['onAlexaHouseholdListEvent.ItemsCreated']((alexaEvent, reply) => {
  const listId = alexaEvent.request.body.listId;
  const userId = alexaEvent.user.userId;
  console.log(`Items created for list: ${listId}` for user ${userId});
  return reply;
});
```

VoxaApp.onAlexaHouseholdListEvent.**ItemsCreated**(*callback*[, *atLast*])

VoxaApp.onAlexaHouseholdListEvent.**ItemsUpdated**(*callback*[, *atLast*])

VoxaApp.onAlexaHouseholdListEvent.**ItemsDeleted**(*callback*[, *atLast*])

10.4 Voxa Platforms

Voxa Platforms wrap your *VoxaApp* and allows you to define handlers for the different supported voice platforms.

class VoxaPlatform(*voxaApp*, *config*)

Arguments

- **voxaApp** (VoxaApp) – The app
- **config** – The config

VoxaPlatform.**startServer**([*port*])

Returns A promise that resolves to a running `http.Server` on the specified port number, if no port number is specified it will try to get a port number from the `PORT` environment variable or default to port 3000

This method can then be used in combination with a proxy server like ngrok or Bespoken tools proxy to enable local development of your voice application

VoxaPlatform.**lambda**()

Returns A lambda handler that will call the *app.execute* method

```
exports.handler = alexaSkill.lambda();
```

VoxaPlatform.**lambdaHTTP**()

Returns A lambda handler to use as an AWS API Gateway ProxyEvent handler that will call the *app.execute* method

```
exports.handler = dialogflowAction.lambdaHTTP();
```

VoxaPlatform.**azureFunction**()

Returns An azure function handler

```
module.exports = cortanaSkill.azureFunction();
```

10.4.1 Alexa

The Alexa Platform allows you to use Voxa with Alexa

```
const { AlexaPlatform } = require('voxa');
const { voxaApp } = require('./app');

const alexaSkill = new AlexaPlatform(voxaApp);
exports.handler = alexaSkill.lambda();
```

10.4.2 Dialogflow

The GoogleAssistant and Facebook Platforms allow you to use Voxa with these 2 type of bots

```
const { GoogleAssistantPlatform, FacebookPlatform } = require('voxa');
const { voxaApp } = require('./app');

const googleAction = new GoogleAssistantPlatform(voxaApp);
exports.handler = googleAction.lambdaHTTP();

const facebookBot = new FacebookPlatform(voxaApp);
exports.handler = facebookBot.lambdaHTTP();
```

10.4.3 Botframework

The BotFramework Platform allows you to use Voxa with Microsoft Botframework

```
const { BotFrameworkPlatform } = require('voxa');
const { AzureBotStorage, AzureTableClient } = require('botbuilder-azure');
const { voxaApp } = require('./app');
const config = require('./config');

const tableName = config.tableName;
const storageKey = config.storageKey; // Obtain from Azure Portal
const storageName = config.storageName;
```

(continues on next page)

(continued from previous page)

```
const azureTableClient = new AzureTableClient(tableName, storageName, storageKey);
const tableStorage = new AzureBotStorage({ gzipData: false }, azureTableClient);

const botframeworkSkill = new BotFrameworkPlatform(voxaApp, {
  storage: tableStorage,
  recognizerURI: process.env.LuisRecognizerURI,
  applicationId: process.env.MicrosoftAppId,
  applicationPassword: process.env.MicrosoftAppPassword,
  defaultLocale: 'en',
});

module.exports = botframeworkSkill.azureFunction();
```

10.5 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 `deserialize` method that should initialize it based on an object of attributes and a `serialize` method that should return a `JSON.stringify` able structure to then store in the session attributes.

```
/*
 * Copyright (c) 2018 Rain Agency <contact@rain.agency>
 * Author: Rain Agency <contact@rain.agency>
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy of
 * this software and associated documentation files (the "Software"), to deal in
 * the Software without restriction, including without limitation the rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
 * the Software, and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
 * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
 * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 */

import * as _ from "lodash";
import { IBag, IVoxaEvent } from "./VoxaEvent";

export class Model {
  [key: string]: any;

  public static deserialize(
    data: IBag,
    voxaEvent: IVoxaEvent,
  ): Promise<Model> | Model {
    return new this(data);
  }
```

(continues on next page)

(continued from previous page)

```
  public state?: string;

  constructor(data: any = {}) {
    _.assign(this, data);
  }

  public async serialize(): Promise<any> {
    return this;
  }
}

export interface IModel {
  new (data?: any): Model;
  deserialize(data: IBag, event: IVoxaEvent): Model | Promise<Model>;
  serialize(): Promise<any>;
}
```

10.6 Views and Variables

10.6.1 Views

Views are the Voxa way of handling replies to the user, they're templates of responses using a simple javascript DSL. They can contain ssml and include cards.

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

Also, there's a special type of view which can contain arrays of options, when Voxa finds one of those like the `LaunchIntent.OpenResponse` it will select a random sample and use it as the response.

10.6.2 I18N

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

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

```
const views = {
  en: {
    translaction: {
      LaunchIntent: {
        OpenResponse: [
          'Hello! <break time="3s"/> Good  {time}. Is there anything i can do to help␣
↪you today?',
          'Hi there! <break time="3s"/> Good  {time}. How may i be of service?',
          'Good  {time}, Welcome!. How can i help you?',
        ]
      },
      ExitIntent: {
        Farewell: 'Ok. For more info visit {site} site.',
      },
      HelpIntent: {
```

(continues on next page)

(continued from previous page)

```
        HelpAboutSkill: 'For more help visit example dot com'
      },
      Count: {
        Say: '{count}',
        Tell: '{count}',
      },
    }
  }
};
```

10.6.3 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*, *voxaEvent*)

Arguments

- **voxaEvent** – The current *voxa 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(voxaEvent) {
    return Promise.resolve('example.com');
  },

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

  locale: function locale(voxaEvent) {
    return voxaEvent.locale;
  }
};
```

10.7 Controllers

Controllers in your application control the logic of your skill, they respond to alexa voxaEvents, 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 with a transition:

```
app.onState('HelpIntent', {
  tell: "Help"
});
```

Or they can be a function that gets a *voxaEvent* object:

```
app.onState('launch', (voxaEvent) => {
  return { tell: 'LaunchIntent.OpenResponse' };
});
```

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

`onState` also takes a third parameter which can be used to limit which intents a controller can respond, for example

```
app.onState('shouldSendEmail?', {
  sayp: "All right! An email has been sent to your inbox",
  flowp: "terminate"
}, "YesIntent");

app.onState('shouldSendEmail?', {
  sayp: "No problem, is there anything else i can help you with?",
  flowp: "yield"
}, "NoIntent");
```

10.7.1 The `onIntent` helper

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

```
app.onIntent('LaunchIntent', (voxaEvent) => {
  return { tell: 'LaunchIntent.OpenResponse' };
});
```

If you receive a Display.ElementSelected type request, you could use the same approach for intents and state. Voxa receives this type of request and turns it into `DisplayElementSelected` Intent

```
app.onIntent('DisplayElementSelected', (voxaEvent) => {
  return { tell: 'DisplayElementSelected.OpenResponse' };
});
```

Keep in mind that controllers created with `onIntent` won't accept transitions from other states with a different intent

10.8 Transition

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

10.8.1 `to`

The `to` key should be the name of a 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' };
```

10.8.2 `directives`

Directives is an array of directive objects that implement the `IDirective` interface, they can make modifications to the reply object directly

```
const { PlayAudio } = require('voxa').alexa;

return {
  directives: [new PlayAudio(url, token)],
};
```

10.8.3 flow

The `flow` key can take one of three values:

continue: This is the default value if the flow key is not present, it merely continues the state machine execution with an internal transition, it keeps building the response until a controller returns a `yield` or a `terminate` flow.

yield: This stops the state machine and returns the current response to the user without terminating the session.

terminate: This stops the state machine and returns the current response to the user, it closes the session.

10.8.4 say

Renders a view and adds it as SSML to the response

10.8.5 sayp

Adds the passed value as SSML to the response

10.8.6 text

Renders a view and adds it as plain text to the response

10.8.7 textp

Adds the passed value as plain text to the response

10.8.8 reprompt

Used to render a view and add the result to the response as a reprompt

10.8.9 reply

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

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

10.9 The `voxaEvent` Object

class VoxaEvent (*event*, *context*)

The `voxaEvent` object contains all the information from the Voxa 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.

`VoxaEvent.`**rawEvent**
A plain javascript copy of the request object as received from the platform

`VoxaEvent.`**executionContext**
On AWS Lambda this object contains the context

`VoxaEvent.`**t**
The current translation function from i18next, initialized to the language of the current request

`VoxaEvent.`**renderer**
The renderer object used in the current request

`VoxaEvent.`**platform**
The currently running *Voxa Platform*

`VoxaEvent.`**model**
The default middleware instantiates a `Model` and makes it available through `voxaEvent.model`

`VoxaEvent.intent.`**params**
In Alexa the voxaEvent object makes `intent.slots` available through `intent.params` after aplying a simple transformation so

```
{ "slots": [{ "name": "Dish", "value": "Fried Chicken" }] }
```

becomes:

```
{ "Dish": "Fried Chicken" }
```

in other platforms it does it's best to make the intent params for each platform also available on `intent.params`

`VoxaEvent.`**user**
An object that contains the userId and accessToken if available

```
{
  "userId": "The platform specific userId",
  "id": "same as userId",
  "accessToken": "available if user has done account linking"
}
```

`VoxaEvent.`**model**
An instance of the *Voxa App Model*.

`VoxaEvent.`**log**
An instance of lambda-log

`VoxaEvent.`**supportedInterfaces**`()`
Array of supported interfaces

Returns Array A string array of the platform's supported interfaces

`VoxaEvent.`**getUserInformation**`()`
Object with user personal information from the platform being used.

```
{
  // Google specific fields
  "sub": 1234567890,        // The unique ID of the user's Google Account
  "iss": "https://accounts.google.com",        // The token's issuer
  "aud": "123-abc.apps.googleusercontent.com", // Client ID assigned to your␣
↪Actions project
  "iat": 233366400,         // Unix timestamp of the token's creation time
  "exp": 233370000,         // Unix timestamp of the token's expiration time
  "emailVerified": true,
  "givenName": "John",
  "familyName": "Doe",
  "locale": "en_US",

  // Alexa specific fields
  "zipCode": "98101",
  "userId": "amzn1.account.K2LI23KL2LK2",

  // Platforms common fields
  "email": "johndoe@gmail.com",
  "name": "John Doe"
}
```

Returns object A object with user's information

VoxaEvent.**getUserInformationWithGoogle**()
Object with user personal information from Google. Go *here* for more information.

```
{
  "sub": 1234567890,        // The unique ID of the user's Google Account
  "iss": "https://accounts.google.com",        // The token's issuer
  "aud": "123-abc.apps.googleusercontent.com", // Client ID assigned to your␣
↪Actions project
  "iat": 233366400,         // Unix timestamp of the token's creation time
  "exp": 233370000,         // Unix timestamp of the token's expiration time
  "givenName": "John",
  "familyName": "Doe",
  "locale": "en_US",
  "email": "johndoe@gmail.com",
  "name": "John Doe"
}
```

Returns object A object with user's information

VoxaEvent.**getUserInformationWithLWA**()
Object with user personal information from Amazon. Go *here* for more information.

```
{
  "email": "johndoe@gmail.com",
  "name": "John Doe",
  "zipCode": "98101",
  "userId": "amzn1.account.K2LI23KL2LK2"
}
```

Returns object A object with user's information

`IVoxaEvent` is an interface that inherits its attributes and function to the specific platforms, for more information about each platform's own methods visit:

- *AlexaEvent*
- *BotFrameworkEvent*
- *Dialogflow Event*
- *GoogleAssistantEvent*
- *FacebookEvent*

10.10 The `AlexaEvent` Object

class AlexaEvent(*event*, *lambdaContext*)

The `alexaEvent` object contains all the information from the Voxa 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.AlexaEvent.`**token**

A convenience getter to obtain the request's token, specially when using the `Display.ElementSelected`

`AlexaEvent.AlexaEvent.alexa.`**customerContact**

When a customer enables your Alexa skill, your skill can request the customer's permission to the their contact information, see *Customer Contact Information Reference*.

`AlexaEvent.AlexaEvent.alexa.`**deviceAddress**

When a customer enables your Alexa skill, your skill can obtain the customer's permission to use address data associated with the customer's Alexa device, see *Device Address Information Reference*.

`AlexaEvent.AlexaEvent.alexa.`**deviceSettings**

Alexa customers can set their timezone, distance measuring unit, and temperature measurement unit in the Alexa app, see *Device Settings Reference*.

`AlexaEvent.AlexaEvent.alexa.`**isp**

The in-skill purchasing feature enables you to sell premium content such as game features and interactive stories for use in skills with a custom interaction model, see *In-Skill Purchases Reference*.

`AlexaEvent.AlexaEvent.alexa.`**lists**

Alexa customers have access to two default lists: Alexa to-do and Alexa shopping. In addition, Alexa customer can create and manage custom lists in a skill that supports that, see *Alexa Shopping and To-Do Lists Reference*.

10.11 The `BotFrameworkEvent` Object

10.12 The `GoogleAssistantEvent` Object

class GoogleAssistantEvent(*event*, *lambdaContext*)

The `googleAssistantEvent` object contains all the information from the Voxa 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.

`GoogleAssistantEvent.GoogleAssistantEvent.google.`**conv**

The conversation instance that contains the raw input sent by Dialogflow

10.13 The `FacebookEvent` Object

class FacebookEvent(*event*, *lambdaContext*)

The `facebookEvent` object contains all the information from the Voxa event for the Facebook Messenger platform, just like Google Assistant events. Additionally you can access the *facebook* property to send Actions to the Chatbot conversation:

```
const { FacebookEvent, FacebookPlatform, VoxaApp } = require('voxa');

const config = {
  pageAccessToken: 'EAAaKuJF183EBAApxv.........',
};
const app = new VoxaApp({ views, variables });
const facebookBot = new FacebookPlatform(app, config);

app.onIntent("LaunchIntent", async (voxaEvent: FacebookEvent) => {
  await voxaEvent.facebook.sendTypingOffAction();
  await voxaEvent.facebook.sendMarkSeenAction();
  await voxaEvent.facebook.sendTypingOnAction();

  const info = await voxaEvent.getUserInformation(FACEBOOK_USER_FIELDS.ALL);

  voxaEvent.model.info = info;
  return {
    flow: "terminate",
    text: "Facebook.User.FullInfo",
    to: "die",
  };
});

const reply = await facebookBot.execute(event);
```

The `facebookEvent` object also gives you the necessary helpers to implement the Handover Protocol, very useful when you want to pass the conversation control from your bot to a person who manages your Facebook Page. The most common example is when user sends to your bot the following text: I want to talk to a representative. This means your bot is not understanding what user is saying, or the bot can't find what the user is looking for. So, it's necessary a person to talk directly to the user. You can pass the control to your Page Inbox like this:

```
const { FacebookEvent, FacebookPlatform, VoxaApp } = require('voxa');

const config = {
  pageAccessToken: 'EAAaKuJF183EBAApxv.........',
};
const app = new VoxaApp({ views, variables });
const facebookBot = new FacebookPlatform(app, config);

app.onIntent("PassControlIntent", async (voxaEvent: FacebookEvent) => {
  await voxaEvent.facebook.passThreadControlToPageInbox();

  return {
    flow: "terminate",
    text: "Facebook.RepresentativeWillGetInTouch.text",
    to: "die",
  };
});
```

Also, if the app you are working on is not the Primary Receiver, you can request control of the conversation like this:

```
const { FacebookEvent, FacebookPlatform, VoxaApp } = require('voxa');

const config = {
  pageAccessToken: 'EAAaKuJF183EBAApxv.........',
};
const app = new VoxaApp({ views, variables });
const facebookBot = new FacebookPlatform(app, config);

app.onIntent("CustomIntent", async (voxaEvent: FacebookEvent) => {
  await voxaEvent.facebook.requestThreadControl();

  return {
    flow: "terminate",
    text: "Facebook.ControlRequested.text",
    to: "die",
  };
});
```

Finally, if you detect the secondary receiver is not responding to the user, you can make your bot (Primary Receiver) take the control of the conversation like this:

```
const { FacebookEvent, FacebookPlatform, VoxaApp } = require('voxa');

const config = {
  pageAccessToken: 'EAAaKuJF183EBAApxv.........',
};
const app = new VoxaApp({ views, variables });
const facebookBot = new FacebookPlatform(app, config);

app.onIntent("CustomIntent", async (voxaEvent: FacebookEvent) => {
  await voxaEvent.facebook.takeThreadControl();

  return {
    flow: "terminate",
    text: "Facebook.ControlTaken.text",
    to: "die",
  };
});
```

10.14 Alexa Directives

10.14.1 HomeCard

Alexa Documentation

Interactions between a user and an Alexa device can include home cards displayed in the Amazon Alexa App, the companion app available for Fire OS, Android, iOS, and desktop web browsers. These are graphical cards that describe or enhance the voice interaction. A custom skill can include these cards in its responses.

In Voxa you can send cards using a view or returning a Card like structure directly from your controller

```
const views = {
  "de-DE": {
    translation: {
      Card: {
```

(continues on next page)

(continued from previous page)

```
        image: {
          largeImageUrl: "https://example.com/large.jpg",
          smallImageUrl: "https://example.com/small.jpg",
        },
        title: "Title",
        type: "Standard",
      },
    },
  };


  app.onState('someState', () => {
    return {
      alexaCard: 'Card',
    };
  });


  app.onState('someState', () => {
    return {
      alexaCard: {
        image: {
          largeImageUrl: "https://example.com/large.jpg",
          smallImageUrl: "https://example.com/small.jpg",
        },
        title: "Title",
        type: "Standard",
      },
    };
  });
```

10.14.2 AccountLinkingCard

Alexa Documentation

An account linking card is sent with the *alexaAccountLinkingCard* key in your controller, it requires no parameters.

```
app.onState('someState', () => {
  return {
    alexaAccountLinkingCard: null,
  };
});
```

10.14.3 RenderTemplate

Alexa Documentation

Voxa provides a *DisplayTemplate* builder that can be used with the *alexaRenderTemplate* controller key to create Display templates for the echo show and echo spot.

```
const voxa = require('voxa');
const { DisplayTemplate } = voxa;

app.onState('someState', () => {
```

(continues on next page)

(continued from previous page)

```
  const template = new DisplayTemplate("BodyTemplate1")
    .setToken("token")
    .setTitle("This is the title")
    .setTextContent("This is the text content", "secondaryText", "tertiaryText")
    .setBackgroundImage("http://example.com/image.jpg", "Image Description")
    .setBackButton("HIDDEN");

  return {
    alexaRenderTemplate: template,
  };
});
```

10.14.4 Alexa Presentation Language (APL) Templates

Alexa Documentation

An APL Template is sent with the *alexaAPLTemplate* key in your controller. You can pass the directive object directly or a view name with the directive object.

One important thing to know is that is you sent a Render Template and a APL Template in the same response but the APL Template will be the one being rendered if the device supports it; if not, the Render Template will be one being rendered.

```
// variables.js

  exports.MyAPLTemplate = (voxaEvent) => {
    // Do something with the voxaEvent, or not...

    return {
      datasources: {},
      document: {},
      token: "SkillTemplateToken",
      type: "Alexa.Presentation.APL.RenderDocument",
    };
  });

// views.js

  const views = {
    "en-US": {
      translation: {
        MyAPLTemplate: "{MyAPLTemplate}"
      },
    };
  };

// state.js

  app.onState('someState', () => {
    return {
      alexaAPLTemplate: "MyAPLTemplate",
    };
  });

  // Or you can do it directly...
```

(continues on next page)

(continued from previous page)

```
app.onState('someState', () => {
  return {
    alexaAPLTemplate: {
      datasources: {},
      document: {},
      token: "SkillTemplateToken",
      type: "Alexa.Presentation.APL.RenderDocument",
    },
  };
});
```

10.14.5 Alexa Presentation Language (APL) Commands

Alexa Documentation

An APL Command is sent with the *alexaAPLCommand* key in your controller. Just like the APL Template, you can pass the directive object directly or a view name with the directive object.

```
// variables.js

  exports.MyAPLCommand = (voxaEvent) => {
    // Do something with the voxaEvent, or not...

    return {
      token: "SkillTemplateToken",
      type: "Alexa.Presentation.APL.ExecuteCommands";
      commands: [{
        type: "SpeakItem", // Karaoke type command
        componentId: "someAPLComponent";
      }],
    };
  });

// views.js

  const views = {
    "en-US": {
      translation: {
        MyAPLCommand: "{MyAPLCommand}"
      },
    };
  };

// state.js

  app.onState('someState', () => {
    return {
      alexaAPLCommand: "MyAPLCommand",
    };
  });

  // Or you can do it directly...

  app.onState('someState', () => {
```

(continues on next page)

(continued from previous page)

```
    return {
      alexaAPLCommand: {
        token: "SkillTemplateToken",
        type: "Alexa.Presentation.APL.ExecuteCommands";
        commands: [{
          type: "SpeakItem", // Karaoke type command
          componentId: "someAPLComponent";
        }],
      },
    };
  });



Alexa Presentation Language - T (APLT) Templates
```

Alexa Documentation

Alexa Presentation Language is supported on devices with character displays. Use the APLT document format to send text to these devices. The APLT document format is smaller and simpler than the APL document format supported by devices with screens.

One important thing to know is that if you sent a Render Template and a APLT Template in the same response but the APLT Template will be the one being rendered if the device supports it; if not, the Render Template will be one being rendered.

```
// variables.js

  exports.MyAPLTTemplate = (voxaEvent) => {
    // Do something with the voxaEvent, or not...

     return {
      datasources: {},
      document: {},
      targetProfile: "FOUR_CHARACTER_CLOCK",
      token: SkillTemplateToken,
      type: "Alexa.Presentation.APLT.RenderDocument"
    };
  });

// views.js

  const views = {
    "en-US": {
      translation: {
        MyAPLTTemplate: "{MyAPLTTemplate}"
      },
    };
  };

// state.js

  app.onState('someState', () => {
    return {
      alexaAPLTTemplate: "MyAPLTTemplate",
```

(continues on next page)

(continued from previous page)

```
  };
});

// Or you can do it directly...

app.onState('someState', () => {
  return {
    alexaAPLTTemplate: {
      datasources: {},
      document: {},
      targetProfile: "FOUR_CHARACTER_CLOCK",
      token: "SkillTemplateToken",
      type: "Alexa.Presentation.APLT.RenderDocument",
    },
  };
});
```

10.14.6 Alexa Presentation Language - T (APLT) Commands

Alexa Documentation

An APLT Command is sent with the *alexaAPLTCommand* key in your controller. Just like the APLT Template, you can pass the directive object directly or a view name with the directive object.

// variables.js

exports.MyAPLTCommand = (voxaEvent) => { // Do something with the voxaEvent, or not...

return { token: "SkillTemplateToken", type: "Alexa.Presentation.APLT.ExecuteCommands"; commands: [{

type: "SetValue", description:

"Changes the text property value on the 'myTextId' component.",

componentId: "myTextId", property: "text", value: "New text value!", delay: 3000

}]

};

});

// views.js

const views = {

"en-US": {

translation: { MyAPLTCommand: "{MyAPLTCommand}"

},

};

};

// state.js

app.onState('someState', () => {

return { alexaAPLTCommand: “MyAPLTCommand”,

};

});

// Or you can do it directly...

app.onState(‘someState’, () => {

return {

alexaAPLTCommand: { token: “SkillTemplateToken”, type: “Alexa.Presentation.APLT.ExecuteCommands”; commands: [{

type: “SetValue”, description:

“Changes the text property value on the ‘myTextId’ component.”,

componentId: “myTextId”, property: “text”, value: “New text value!”, delay: 3000

}]

},

};

});

10.14.7 PlayAudio

Alexa Documentation

```
function register(app) {
  app.onState('someState', () => {
    const url = 'http://example.com/example.mp3';
    const token = '{}';
    const offsetInMilliseconds = 0;
    const behavior = 'REPLACE_ALL';
    const playAudio = new PlayAudio(url, token, offsetInMilliseconds, behavior);

    return {
      directives: [playAudio],
    };
  });
}
```

Add metadata for your audio

The *PlayAudio* directive has a fifth parameter to set metadata for an audio, just pass it when creating a *PlayAudio* instance following the correct structure required by Amazon (refer to the Alexa documentation link above).

```
function register(app) {
  app.onState('someState', () => {
    const url = 'http://example.com/example.mp3';
    const token = '{}';
    const offsetInMilliseconds = 0;
    const behavior = 'REPLACE_ALL';
    const metadata = {
      title: 'title of the track to display',
```

(continues on next page)

(continued from previous page)

```
      subtitle: 'subtitle of the track to display',
      art: {
        sources: [
          {
            url: 'https://cdn.example.com/url-of-the-album-art-image.png'
          }
        ]
      },
      backgroundImage: {
        sources: [
          {
            url: 'https://cdn.example.com/url-of-the-background-image.png'
          }
        ]
      }
    };
    const playAudio = new PlayAudio(url, token, offsetInMilliseconds, behavior,␣
↪metadata);

    return {
      directives: [playAudio],
    };
  });
}
```

10.14.8 StopAudio

Alexa Documentation

```
function register(app) {
  app.onState("PauseIntent", {
    alexaStopAudio: true,
    reply: "SomeViewWithAPauseText",
    to: "die"
  });
}
```

10.14.9 Resume an Audio

Resuming an audio works using the *PlayAudio* directive, the only thing that need to change is the *offsetInMilliseconds* to, of course, start the audio where it stopped. The *offsetInMilliseconds* comes from the context attribute in the raw event coming from Alexa.

You can also use the *token* to pass important information since the AudioPlayer context is outside of the skill session, therefore you can't access the session variables. In this example, the information of the audio is returned with the *alexaPlayAudio* key from Voxa.

```
function register(app) {
  app.onState("playSomeAudio", () => {
    const url = 'http://example.com/example.mp3';
    const token = JSON.stringify({ url });
    const offsetInMilliseconds = 0;
    const behavior = 'REPLACE_ALL';
```

(continues on next page)

(continued from previous page)

```
    const metadata = {
      art: {
        sources: [
          {
            url: "http://example.com/image.png",
          },
        ],
      },
      backgroundImage: {
        sources: [
          {
            url: "http://example.com/image.png",
          },
        ],
      },
      subtitle: "Subtitle",
      title: "Title",
    };

    return {
      alexaPlayAudio: {
        behavior,
        metadata,
        offsetInMilliseconds,
        token
        url,
      },
    };
  });

  app.onIntent("ResumeIntent", (voxaEvent: IVoxaEvent) => {
    if (voxaEvent.rawEvent.context) {
      const token = JSON.parse(voxaEvent.rawEvent.context.AudioPlayer.token);
      const offsetInMilliseconds = voxaEvent.rawEvent.context.AudioPlayer.
↪offsetInMilliseconds;
      const url = token.url;

      const playAudio = new PlayAudio(url, token, offsetInMilliseconds);

      return {
        reply: "SomeViewSayingResumingAudio",
        to: "die",
        directives: [playAudio]
      };
    }

    return { flow: "terminate", reply: "SomeGoodbyeMessage" };
  });
}
```

10.14.10 ElicitSlot Directive

Alexa Documentation

When there is an active dialog you can use the `alexaElicitDialog` to tell alexa to prompt the user for a specific slot in the next turn. A prompt passed in as a `say`, `reply` or another statement is required and will replace the

prompt that is provided to the interaction model for the dialog. The `flow` and `to` keys should not be used or should always be `flow: "yield"` and `to: "{current_intent}"` since dialogs loop the same intent until all of the parameters are filled.

The only required parameter is the `slotToElicit`, but you can also pass in the values for slots to update the current values. If a slot isn't declared in the interaction model it will be ignored or cause an error.

```
// simplest example
app.onIntent('someDialogIntent', () => {
  // check if the dialog is complete and do some cool stuff here //

  // if we need to ask the user for something //
  return {
    alexaElicitDialog: {
      slotToElicit: "favoriteColor",
    },
    sayp: ["What is your favorite color?"],
  };
});

// updating slots example
app.onIntent('someDialogIntent', () => {
  // check if the dialog is complete and do some cool stuff here //

  // if we need to ask the user for something //
  return {
    alexaElicitDialog: {
      slotToElicit: "favoriteColor",
      slots: {
        bestLetter: {
          value: "W",
          confirmationStatus: "CONFIRMED",
        },
      },
    },
    sayp: ["What is your favorite color?"],
  };
});

// This is still OK
app.onIntent('someDialogIntent', () => {
  return {
    alexaElicitDialog: {
      slotToElicit: "favoriteColor",
    },
    sayp: ["What is your favorite color?"],
    to: "someDialogIntent",
  };
});

// This will break
app.onIntent('someDialogIntent', () => {
  return {
    alexaElicitDialog: {
      slotToElicit: "favoriteColor",
    },
    sayp: ["What is your favorite color?"],
    to: "someOtherThing",
```

(continues on next page)

(continued from previous page)

```
  };
});
```

10.14.11 Dynamic Entities

Alexa Documentation

Dynamic entities are sent with the *alexaDynamicEntities* key in your controller. You need to pass a view name with the types array.

```
// variables.js

  exports.dynamicNames = (voxaEvent) => {
    return [
      {
        name: "LIST_OF_AVAILABLE_NAMES",
        values: [
          {
            id: "nathan",
            name: {
              synonyms: ["nate"],
              value: "nathan"
            }
          }
        ]
      }
    ];
  });

// views.js

  const views = {
    "en-US": {
      translation: {
        MyAvailableNames: "{dynamicNames}"
      },
    };
  };

// state.js

  app.onState('someState', () => {
    return {
      alexaDynamicEntities: "MyAvailableNames",
    };
  });

  // Or you can pass the types directly...

  app.onState('someState', () => {
    return {
      alexaDynamicEntities: [
        {
          name: "LIST_OF_AVAILABLE_NAMES",
          values: [
```

(continues on next page)

(continued from previous page)

```
            {
              id: "nathan",
              name: {
                synonyms: ["nate"],
                value: "nathan"
              }
            }
          ]
        }
      ],
    };
  });

  // Or you can pass the whole directive directly...

  app.onState('someState', () => {
    return {
      alexaDynamicEntities: {
        type: "Dialog.UpdateDynamicEntities",
        updateBehavior: "REPLACE",
        types: [
          {
            name: "LIST_OF_AVAILABLE_NAMES",
            values: [
              {
                id: "nathan",
                name: {
                  synonyms: ["nate"],
                  value: "nathan"
                }
              }
            ]
          }
        ]
      },
    };
  });
```

10.14.12 Entity Directive

The Entity Directive allows to create Dynamic Entities based on a generic structure.

Entities are sent with the *entities* key in your controller. You need to pass a view name with the types array or object.

If the updateBehavior is ommited, the REPLACE value is used by default. Use only one updateBehavior property. In case each entity has an updateBehavior property, just the first coincident will be used.

The id property is optional an only used for Alexa platform.

The value in alexaEntityName should be the same as the one used for the Slot Types in the Alexa Developer Console.

```
// variables.js

   export function myEntity(voxaEvent: VoxaEvent) {
     // Do something with the voxaEvent, or not...
```

(continues on next page)

(continued from previous page)

```
    const entityType = [
      {
        "entities": [
          {
            "synonyms": ["apple", "green apple", "crabapple"],
            "value": "APPLE_KEY"
          },
          {
            "id": 1,
            "synonyms": ["orange"],
            "value": "ORANGE_KEY"
          }
        ],
        "alexaEntityName": "LIST_OF_ANIMALS"
      }
    ];

    return entityType;
  }

// views.js

  const views = {
    "en-US": {
      translation: {
        myEntity: "{myEntity}"
      },
    };
  };

// state.js

  app.onState('someState', {
    entities: "myEntity",
    flow: "yield",
    sayp: "Hello!",
    to: "entry",
  });
```

// Or you can do it directly...

app.onState('someState', {

entities: [

{

"entities": [

{ "synonyms": ["apple", "green apple", "crabapple"], "value": "APPLE_KEY"

}, {

"id": 1, "synonyms": ["orange"], "value": "ORANGE_KEY"

}

], "alexaEntityName": "LIST_OF_ANIMALS"

}

], flow: “yield”, sayp: “Hello!”, to: “entry”,

});

10.15 Google Assistant Directives

Dialog Flow directives expose google actions functionality that’s platform specific. In general they take the same parameters you would pass to the Actions on Google Node JS SDK.

10.15.1 List

Actions on Google Documentation

The single-select list presents the user with a vertical list of multiple items and allows the user to select a single one. Selecting an item from the list generates a user query (chat bubble) containing the title of the list item.

```
app.onState('someState', () => {
  return {
    dialogflowList: {
      title: 'List Title',
      items: {
        // Add the first item to the list
        [SELECTION_KEY_ONE]: {
          synonyms: [
            'synonym of title 1',
            'synonym of title 2',
            'synonym of title 3',
          ],
          title: 'Title of First List Item',
          description: 'This is a description of a list item.',
          image: new Image({
            url: IMG_URL_AOG,
            alt: 'Image alternate text',
          }),
        },
        // Add the second item to the list
        [SELECTION_KEY_GOOGLE_HOME]: {
          synonyms: [
            'Google Home Assistant',
            'Assistant on the Google Home',
        ],
          title: 'Google Home',
          description: 'Google Home is a voice-activated speaker powered by ' +
            'the Google Assistant.',
          image: new Image({
            url: IMG_URL_GOOGLE_HOME,
            alt: 'Google Home',
          }),
        },
        // Add the third item to the list
        [SELECTION_KEY_GOOGLE_PIXEL]: {
          synonyms: [
            'Google Pixel XL',
            'Pixel',
            'Pixel XL',
```

(continues on next page)

(continued from previous page)

```
          ],
          title: 'Google Pixel',
          description: 'Pixel. Phone by Google.',
          image: new Image({
            url: IMG_URL_GOOGLE_PIXEL,
            alt: 'Google Pixel',
          }),
        },
      },
    }
  }
});
```

10.15.2 Carousel

Actions on Google Documentation

The carousel scrolls horizontally and allows for selecting one item. Compared to the list selector, it has large tiles-allowing for richer content. The tiles that make up a carousel are similar to the basic card with image. Selecting an item from the carousel will simply generate a chat bubble as the response just like with list selector.

```
app.onState('someState', () => {
  return {
    dialogflowCarousel: {
      items: {
        // Add the first item to the carousel
        [SELECTION_KEY_ONE]: {
          synonyms: [
            'synonym of title 1',
            'synonym of title 2',
            'synonym of title 3',
          ],
          title: 'Title of First Carousel Item',
          description: 'This is a description of a carousel item.',
          image: new Image({
            url: IMG_URL_AOG,
            alt: 'Image alternate text',
          }),
        },
        // Add the second item to the carousel
        [SELECTION_KEY_GOOGLE_HOME]: {
          synonyms: [
            'Google Home Assistant',
            'Assistant on the Google Home',
        ],
          title: 'Google Home',
          description: 'Google Home is a voice-activated speaker powered by ' +
            'the Google Assistant.',
          image: new Image({
            url: IMG_URL_GOOGLE_HOME,
            alt: 'Google Home',
          }),
        },
        // Add third item to the carousel
        [SELECTION_KEY_GOOGLE_PIXEL]: {
```

(continues on next page)

(continued from previous page)

```
          synonyms: [
            'Google Pixel XL',
            'Pixel',
            'Pixel XL',
          ],
          title: 'Google Pixel',
          description: 'Pixel. Phone by Google.',
          image: new Image({
            url: IMG_URL_GOOGLE_PIXEL,
            alt: 'Google Pixel',
          }),
        },
      },
    }
  }
});
```

10.15.3 Browse Carousel

Actions on Google Documentation

A browsing carousel is a rich response that allows users to scroll vertically and select a tile in a collection. Browsing carousels are designed specifically for web content by opening the selected tile in a web browser.

```
app.onState('someState', () => {
  return {
    dialogflowBrowseCarousel: {
      items: [
        {
          title: 'Title of the item',
          description: 'This is a description of an item.',
          footer: 'Footer of the item'
          openUrlAction: {
            url: 'https://example.com/page',
            urlTypeHint: 'DEFAULT' // Optional
          }
        },
        {
          title: 'Title of the item',
          description: 'This is a description of an item.',
          footer: 'Footer of the item'
          openUrlAction: {
            url: 'https://example.com/page',
            urlTypeHint: 'DEFAULT' // Optional
          }
        },
      ],
    }
  }
});
```

10.15.4 Suggestions

Actions on Google Documentation

Use suggestion chips to hint at responses to continue or pivot the conversation. If during the conversation there is a primary call for action, consider listing that as the first suggestion chip.

Whenever possible, you should incorporate one key suggestion as part of the chat bubble, but do so only if the response or chat conversation feels natural.

```
app.onState('someState', () => {
  return {
    dialogflowSuggestions: ['Exit', 'Continue']
  }
});
```

```
app.onState('someState', () => {
  return {
    dialogflowLinkOutSuggestion: {
      name: "Suggestion Link",
      url: 'https://assistant.google.com/',
    }
  }
});
```

10.15.5 BasicCard

Actions on Google Documentation

A basic card displays information that can include the following:

- Image
- Title
- Sub-title
- Text body
- Link button
- Border

Use basic cards mainly for display purposes. They are designed to be concise, to present key (or summary) information to users, and to allow users to learn more if you choose (using a weblink).

In most situations, you should add suggestion chips below the cards to continue or pivot the conversation.

Avoid repeating the information presented in the card in the chat bubble at all costs.

```
app.onState('someState', () => {
  return {
    dialogflowBasicCard: {
      text: `This is a basic card.  Text in a basic card can include "quotes" and
      most other unicode characters including emoji.  Basic cards also support
      some markdown formatting like *emphasis* or _italics_, **strong** or
      __bold__, and ***bold itallic*** or ___strong emphasis___ `,
      subtitle: 'This is a subtitle',
      title: 'Title: this is a title',
      buttons: new Button({
        title: 'This is a button',
        url: 'https://assistant.google.com/',
      }),
      image: new Image({
```

(continues on next page)

(continued from previous page)

```
        url: 'https://example.com/image.png',
        alt: 'Image alternate text',
      }),
    }
  }
});
```

10.15.6 AccountLinkingCard

Actions on Google Documentation

Account linking is a great way to lets users connect their Google accounts to existing accounts on your service. This allows you to build richer experiences for your users that take advantage of the data they already have in their account on your service. Whether it's food preferences, existing payment accounts, music preferences, your users should be able to have better experiences in the Google Assistant by linking their accounts.

```
app.onState('someState', () => {
  return {
    dialogflowAccountLinkingCard: "To track your exercise"
  }
});
```

10.15.7 MediaResponse

Actions on Google Documentation

Media responses let your app play audio content with a playback duration longer than the 120-second limit of SSML. The primary component of a media response is the single-track card. The card allows the user to perform these operations:

- Replay the last 10 seconds.
- Skip forward for 30 seconds.
- View the total length of the media content.
- View a progress indicator for audio playback.
- View the elapsed playback time.

```
const { MediaObject } = require('actions-on-google');

app.onState('someState', () => {

  const mediaObject = new MediaObject({
    name,
    url,
  });

  return {
    dialogflowMediaResponse: mediaObject
  };
});
```

10.15.8 User Information

Actions on Google Documentation

User information You can obtain the following user information with this helper:

- Display name
- Given name
- Family name
- Coarse device location (zip code and city)
- Precise device location (coordinates and street address)

```
app.onState('someState', () => {
  return {
    dialogflowPermission: {
      context: 'To read your mind',
      permissions: 'NAME',
    }
  };
});
```

10.15.9 Date and Time

Actions on Google Documentation <https://developers.google.com/actions/assistant/helpers#date_and_time>

You can obtain a date and time from users by requesting fulfillment of the actions.intent.DATETIME intent.

```
app.onState('someState', () => {
  return {
    dialogflowDateTime: {
      prompts: {
        initial: 'When do you want to come in?',
        date: 'Which date works best for you?',
        time: 'What time of day works best for you?',
      }
    }
  };
});
```

10.15.10 Confirmation

Actions on Google Documentation <https://developers.google.com/actions/assistant/helpers#confirmation>

You can ask a generic confirmation from the user (yes/no question) and get the resulting answer. The grammar for "yes" and "no" naturally expands to things like "Yea" or "Nope", making it usable in many situations.

```
app.onState('someState', () => {
  return {
    dialogflowConfirmation: 'Can you confirm?',
  };
});
```

10.15.11 Android Link

Actions on Google Documentation

You can ask the user to continue an interaction via your Android app. This helper allows you to prompt the user as part of the conversation. You'll first need to associate your Android app with your Actions Console project via the Brand Verification page.

```
app.onState('someState', () => {
  const options = {
    destination: 'Google',
    url: 'example://gizmos',
    package: 'com.example.gizmos',
    reason: 'handle this for you',
  };

  return {
    dialogflowDeepLink: options
  };
});
```

10.15.12 Place and Location

Actions on Google Documentation

You can obtain a location from users by requesting fulfillment of the actions.intent.PLACE intent. This helper is used to prompt the user for addresses and other locations, including any home/work/contact locations that they've saved with Google.

Saved locations will only return the address, not the associated mapping (e.g. "123 Main St" as opposed to "HOME = 123 Main St").

```
app.onState('someState', () => {
  return {
    dialogflowPlace: {
      context: 'To find a place to pick you up',
      prompt: 'Where would you like to be picked up?',
    }
  };
});
```

10.15.13 Digital Goods

Actions on Google Documentation

You can add dialog to your Action that sells your in-app products in the Google Play store, using the digital purchases API.

You can use the *google.digitalGoods* object to get the subscriptions and InAppEntitlements filtered by the skuIds you pass to the function. Voxa handles all operations in background to get access to your digital goods in the Play Store. To do that, you need to pass to the GoogleAssistantPlatform object, the packageName of your Android application along with the keyFile with the credentials you created in your Google Cloud project.

10.15.14 TransactionDecision

10.15.15 TransactionRequirements

10.15.16 Routine Suggestions

Actions on Google Documentation

To consistently re-engage with users, you need to become a part of their daily habits. Google Assistant users can already use Routines to execute multiple Actions with a single command, perfect for those times when users wake up in the morning, head out of the house, get ready for bed or many of the other tasks we perform throughout the day. Now, with Routine Suggestions, after someone engages with your Action, you can prompt them to add your Action to their Routines with just a couple of taps.

```
app.onState('someState', () => {
  return {
    dialogflowRegisterUpdate: {
      intent: 'Show Image',
      frequency: 'ROUTINES'
    }
  };
});
```

10.15.17 Push notifications

Actions on Google Documentation

Your app can send push notifications to users whenever relevant, such as sending a reminder when the due date for a task is near.

```
app.onState('someState', () => {
  return {
    dialogflowUpdatePermission: {
      intent: 'tell_latest_tip'
    }
  };
});
```

10.15.18 Multi-surface conversations

Actions on Google Documentation

At any point during your app's flow, you can check if the user has any other surfaces with a specific capability. If another surface with the requested capability is available, you can then transfer the current conversation over to that new surface.

```
app.onIntent('someState', async (voxaEvent) => {
  const screen = 'actions.capability.SCREEN_OUTPUT';
  if (!_.includes(voxaEvent.supportedInterfaces, screen)) {
    const screenAvailable = voxaEvent.conv.available.surfaces.capabilities.
↪has(screen);

    const context = 'Sure, I have some sample images for you.';
    const notification = 'Sample Images';
```

(continues on next page)

(continued from previous page)

```
    const capabilities = ['actions.capability.SCREEN_OUTPUT'];

    if (screenAvailable) {
      return {
        sayp: 'Hello',
        to: 'entry',
        flow: 'yield',
        dialogflowNewSurface: {
          context, notification, capabilities,
        },
      };
    }

    return {
      sayp: 'Does not have a screen',
      flow: 'terminate',
    };
  }

  return {
    sayp: 'Already has a screen',
    flow: 'terminate',
  };
});
```

10.15.19 Output Contexts

Actions on Google Documentation

If you need to add output contexts to the dialog flow webhook you can use the *dialogflowContext* directive

```
app.onIntent("LaunchIntent", {
  dialogflowContext: {
    lifespan: 5,
    name: "DONE_YES_NO_CONTEXT",
  },
  sayp: "Hello!",
  to: "entry",
  flow: "yield",
});
```

10.15.20 Session Entities

Google Documentation

A session represents a conversation between a Dialogflow agent and an end-user. You can create special entities, called session entities during a session. Session entities can extend or replace custom entity types and only exist during the session that they were created for. All session data, including session entities, is stored by Dialogflow for 20 minutes.

For example, if your agent has a @fruit entity type that includes “pear” and “grape”, that entity type could be updated to include “apple” or “orange”, depending on the information your agent collects from the end-user. The updated entity type would have the “apple” or “orange” entity entry for the rest of the session.

Create an entity in your dialogflow agent and make sure that define synonyms option is checked. Add some values and synonyms if needed according agent instructions. Notice that the name of the entity is the value to be used by the

directive (i.e., list-of-fruits).

```
// variables.js

  export function mySessionEntity(voxaEvent: VoxaEvent) {
    // Do something with the voxaEvent, or not...

    const sessionEntityType = [
      {
        "entities": [
          {
            "synonyms": ["apple", "green apple", "crabapple"],
            "value": "APPLE_KEY"
          },
          {
            "synonyms": ["orange"],
            "value": "ORANGE_KEY"
          }
        ],
        "entityOverrideMode": "ENTITY_OVERRIDE_MODE_OVERRIDE",
        "name": "list-of-fruits"
      }
    ];

    return sessionEntityType;
  }

// views.js

  const views = {
    "en-US": {
      translation: {
        mySessionEntity: "{mySessionEntity}"
      },
    };
  };

// state.js

  app.onState('someState', {
    dialogflowSessionEntity: "mySessionEntity",
    flow: "yield",
    sayp: "Hello!",
    to: "entry",
  });

// Or you can do it directly...

  app.onState('someState', {
    dialogflowSessionEntity:  [
      {
        "entities": [
          {
            "synonyms": ["apple", "green apple", "crabapple"],
            "value": "APPLE_KEY"
          },
          {
            "synonyms": ["orange"],
```

(continues on next page)

(continued from previous page)

```
            "value": "ORANGE_KEY"
          }
        ],
        "entityOverrideMode": "ENTITY_OVERRIDE_MODE_OVERRIDE",
        "name": "list-of-fruits"
      }
    ],
    flow: "yield",
    sayp: "Hello!",
    to: "entry",
  });
```

10.15.21 Entity Directive

The Entity Directive allows to create Session Entities based on a generic structure.

Entities are sent with the *entities* key in your controller. You need to pass a view name with the types array or object.

If the entityOverrideMode is ommited, the ENTITY_OVERRIDE_MODE_OVERRIDE value is used by default.

The value in googleEntityName should be the same as the one used for the Entities in the Dialogflow agent.

```
// variables.js

  export function myEntity(voxaEvent: VoxaEvent) {
    // Do something with the voxaEvent, or not...

    const entityType = [
      {
        "entities": [
          {
            "synonyms": ["apple", "green apple", "crabapple"],
            "value": "APPLE_KEY"
          },
          {
            "synonyms": ["orange"],
            "value": "ORANGE_KEY"
          }
        ],
        "googleEntityName": "list-of-animals"
      }
    ];

    return entityType;
  }

// views.js

  const views = {
    "en-US": {
      translation: {
        myEntity: "{myEntity}"
      },
    };
  };
```

(continues on next page)

(continued from previous page)

```
// state.js

  app.onState('someState', {
    entities: "myEntity",
    flow: "yield",
    sayp: "Hello!",
    to: "entry",
  });
```

// Or you can do it directly. . .

app.onState('someState', {

entities: [

{

"entities": [

{ "synonyms": ["apple", "green apple", "crabapple"], "value": "APPLE_KEY"

}, {

"synonyms": ["orange"], "value": "ORANGE_KEY"

}

], "googleEntityName": "list-of-animals"

}

], flow: "yield", sayp: "Hello!", to: "entry",

});

10.16 Botframework Directives

10.16.1 Sign In Card

A sign in card is used to account link your user. On Cortana the parameters are ignored and the system will use the parameters configured in the cortana channel

```
app.onIntent("LaunchIntent", {
  botframeworkSigninCard: {
    buttonTitle: "Sign In",
    cardText: "Sign In Card",
    url: "https://example.com",
  },
  to: "die",
});
```

10.16.2 Hero Card

```
import { HeroCard } from "botbuilder";

const card = new HeroCard()
```

(continues on next page)

(continued from previous page)

```
  .title("Card Title")
  .subtitle("Card Subtitle")
  .text("Some Text");

app.onIntent("LaunchIntent", {
  botframeworkHeroCard: card,
  to: "die",
});
```

10.16.3 Suggested Actions

```
import { SuggestedActions } from "botbuilder";
const suggestedActions = new SuggestedActions().addAction({
  title: "Green",
  type: "imBack",
  value: "productId=1&color=green",
});

app.onIntent("LaunchIntent", {
  botframeworkSuggestedActions: suggestedActions,
  to: "die",
});
```

10.16.4 Audio Card

```
import { AudioCard } from "botbuilder";

const audioCard = new AudioCard().title("Sample audio card");
audioCard.media([
  {
    profile: "audio.mp3",
    url: "http://example.com/audio.mp3",
  },
]);

app.onIntent("LaunchIntent", {
  botframeworkAudioCard: audioCard,
  to: "die",
});
```

10.16.5 Text

The `Text` directive renders a view and adds it to the response in plain text, this response is then shown to the user in devices with a screen

```
app.onIntent("LaunchIntent", {
  say: "SomeView",
  text: "SomeView",
  to: "die",
});
```

10.16.6 Text P

```
app.onIntent("LaunchIntent", {
  sayp: "Some Text",
  textp: "Some Text",
  to: "die",
});
```

10.16.7 Attachments and Attachment Layouts

```
const cards = _.map([1, 2, 3], (index: number) => {
  return new HeroCard().title(`Event ${index}`).toAttachment();
});

app.onIntent("LaunchIntent", {
  botframeworkAttachmentLayout: AttachmentLayout.carousel,
  botframeworkAttachments: cards,
  to: "die",
});
```

10.17 Dialogflow Platform Integrations

Dialogflow offers a variety of integrations so you share your base code across several platforms like Google Assistant, Facebook Messenger and more. For more information about these platforms, visit their Integration docs.

More integrations comming soon to Voxa

10.17.1 Google Assistant

The most common Dialogflow integration is the GoogleAssistantPlatform.

```
const { GoogleAssistantPlatform } = require('voxa');

const app = new VoxaApp({ views, variables });
const googleAction = new GoogleAssistantPlatform(app);
```

Facebook Messenger

The `DialogflowPlatform` for voxa has available some of the core functionalities to send to your chatbot in responses. When you initialize the Facebook Platform object, you can optionally pass a configuration object with the Facebook Page Access Token:

```
const { FacebookPlatform } = require('voxa');

const config = {
  pageAccessToken: 'EAAaKuJF183EBAApxv.........',
};
const app = new VoxaApp({ views, variables });
const facebookBot = new FacebookPlatform(app, config);
```

Voxa will use this token to perform some authentication operations like sending actions to the user's chat window. Check the Facebook Event object for more details.

Voxa also offers a variety of built-in rich components for you to send along with your response. For now, you can integrate the following:

10.17.2 Account Linking button

You need to include in your controller the following field: `facebookAccountLink`, which takes a URL to go into the account linking flow. For more information about the account linking flow, check how to add a Log In Button, and Account Linking.

```
app.onState('someState', () => {
  return {
    facebookAccountLink: "https://www.messenger.com",
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookAccountLink"
  };
});
.....
views
.....
{
  "FacebookAccountLink": {
    "facebookAccountLink": "https://www.messenger.com"
  }
}
```

10.17.3 Account Unlink button

You need to include in your controller the following field: `facebookAccountUnlink`, which can take any value like a boolean, just to indicate to Voxa we're adding this button to the response. For more information about the account linking flow, check how to add a Log Out Button, and Account Linking.

```
app.onState('someState', () => {
  return {
    facebookAccountUnlink: true,
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookAccountUnlink"
  };
});
.....
views
```

(continues on next page)

(continued from previous page)

```
.....
{
  "FacebookAccountLink": {
    "facebookAccountUnlink": true
  }
}
```

10.17.4 Location Quick Reply

You need to include in your controller the following field: `facebookQuickReplyLocation`, which takes a string with the title of the message that goes along with the button requesting user's location. For more information about the account linking flow, check how to add a Location Quick Reply.

```
app.onState('someState', () => {
  return {
    facebookQuickReplyLocation: "Send me your location",
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookQuickReplyLocation"
  };
});
.....
views
.....
{
  "FacebookQuickReplyLocation": {
    "facebookQuickReplyLocation": "Send me your location"
  }
}
```

10.17.5 Phone Number Quick Reply

You need to include in your controller the following field: `facebookQuickReplyPhoneNumber`, which takes a string with the title of the message that goes along with the button requesting user's phone number. For more information about the account linking flow, check how to add a User Phone Number Quick Reply.

```
app.onState('someState', () => {
  return {
    facebookQuickReplyPhoneNumber: "Send me your phone number",
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookQuickReplyPhoneNumber"
  };
```

(continues on next page)

(continued from previous page)

```
});
.....
views
.....
{
  "FacebookQuickReplyPhoneNumber": {
    "facebookQuickReplyPhoneNumber": "Send me your phone number"
  }
}
```

10.17.6 Text Quick Reply

You need to include in your controller the following field: `directives`, which takes an array of directives, and the one you're going to send is a FacebookQuickReplyText directive, that takes 2 parameters: - message: string with the title of the message that goes along with the button requesting user's email. - replyArray: a IFacebookQuickReply object or array of objets with the options to render in the chat.

For more information about the account linking flow, check how to add a User Text Quick Reply.

```
const { FacebookQuickReplyText, IFacebookQuickReply } = require('voxa');

app.onState('someState', () => {
  const quickReplyTextArray: IFacebookQuickReply[] = [
    {
      imageUrl: "https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/
↪16777216colors.png/220px-16777216colors.png",
      payload: "square",
      title: "Square Multicolor",
    },
    {
      imageUrl: "https://www.w3schools.com/colors/img_colormap.gif",
      payload: "hexagonal",
      title: "Hexagonal multicolor",
    },
  ];

  const facebookQuickReplyText = new FacebookQuickReplyText("What's your favorite␣
↪shape?", quickReplyTextArray);

  return {
    directives: [facebookQuickReplyText],
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookQuickReplyText"
  };
});
.....
views
.....
{
```

(continues on next page)

(continued from previous page)

```
  "FacebookQuickReplyText": {
    "facebookQuickReplyText": "{quickReplyText}"
  }
}
.........
variables
.........
const { FacebookQuickReplyText } = require('voxa');

export function quickReplyText(request) {
  const quickReplyTextArray = [
    {
      imageUrl: "https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/
↪16777216colors.png/220px-16777216colors.png",
      payload: "square",
      title: "Square Multicolor",
    },
    {
      imageUrl: "https://www.w3schools.com/colors/img_colormap.gif",
      payload: "hexagonal",
      title: "Hexagonal multicolor",
    },
  ];

  const facebookQuickReplyText = new FacebookQuickReplyText("What's your favorite␣
↪shape?", quickReplyTextArray);

  return {
    directives: [facebookQuickReplyText],
  };
},
```

10.17.7 Email Quick Reply

You need to include in your controller the following field: `facebookQuickReplyUserEmail`, which takes a string with the title of the message that goes along with the button requesting user's email. For more information about the account linking flow, check how to add a User Email Quick Reply.

```
app.onState('someState', () => {
  return {
    facebookQuickReplyUserEmail: "Send me your email",
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookQuickReplyUserEmail"
  };
});
.....
views
.....
{
```

(continues on next page)

(continued from previous page)

```
  "FacebookQuickReplyUserEmail": {
    "facebookQuickReplyUserEmail": "Send me your email"
  }
}
```

10.17.8 Postbacks buttons (Suggestion chips)

You need to include in your controller the following field: `facebookSuggestionChips`, which could be a simple string that the Voxa renderer will get from your views file with an array of strings, or directly an array of strings. For more information about this, check how to add Postback Buttons.

```
app.onState('someState', () => {
  return {
    facebookSuggestionChips: ["YES", "NO"],
    textp: "Select YES or NO",
    to: "entry",
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookSuggestionChips"
  };
});
.....
views
.....
{
  "FacebookSuggestionChips": {
    "facebookSuggestionChips": ["YES", "NO"]
  }
}
```

10.17.9 Carousel

You need to include in your controller the following field: `facebookCarousel`, which takes an object with an array of elements to be taken as items in a generic list of buttons. For more information about the carousel, check how to add a Generic Template.

```
const {
  FACEBOOK_BUTTONS,
  FACEBOOK_WEBVIEW_HEIGHT_RATIO,
  FacebookButtonTemplateBuilder,
  FacebookElementTemplateBuilder,
  FacebookTemplateBuilder,
} = require('voxa');

app.onState('someState', () => {
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const elementBuilder1 = new FacebookElementTemplateBuilder();
```

(continues on next page)

(continued from previous page)

```
  const elementBuilder2 = new FacebookElementTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setTitle("Go to see this URL")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("https://www.example.com/imgs/imageExample.png");

  buttonBuilder2
    .setPayload("value")
    .setTitle("Send this to chat")
    .setType(FACEBOOK_BUTTONS.POSTBACK);

  elementBuilder1
    .addButton(buttonBuilder1.build())
    .addButton(buttonBuilder2.build())
    .setDefaultActionUrl("https://www.example.com/imgs/imageExample.png")
    .setDefaultMessengerExtensions(false)
    .setDefaultWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.COMPACT)
    .setImageUrl("https://www.w3schools.com/colors/img_colormap.gif")
    .setSubtitle("subtitle")
    .setTitle("title");

  elementBuilder2
    .addButton(buttonBuilder1.build())
    .addButton(buttonBuilder2.build())
    .setDefaultActionUrl("https://www.example.com/imgs/imageExample.png")
    .setDefaultMessengerExtensions(false)
    .setDefaultWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.TALL)
    .setImageUrl("https://www.w3schools.com/colors/img_colormap.gif")
    .setSubtitle("subtitle")
    .setTitle("title");

  facebookTemplateBuilder
    .addElement(elementBuilder1.build())
    .addElement(elementBuilder2.build());

  return {
    facebookCarousel: facebookTemplateBuilder.build(),
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookCarousel"
  };
});
.....
views
.....
{
  "FacebookCarousel": {
    "facebookCarousel": "{carousel}"
  }
}
```

(continues on next page)

(continued from previous page)

```
.........
variables
.........
carousel: function carousel(request) {
  const buttons = [
    {
      title: "Go to see this URL",
      type: FACEBOOK_BUTTONS.WEB_URL,
      url: "https://www.example.com/imgs/imageExample.png",
    },
    {
      payload: "value",
      title: "Send this to chat",
      type: FACEBOOK_BUTTONS.POSTBACK,
    },
  ];

  const carousel = {
    elements: [
      {
        buttons,
        defaultActionUrl: "https://www.example.com/imgs/imageExample.png",
        defaultMessengerExtensions: false,
        defaultWebviewHeightRatio: FACEBOOK_WEBVIEW_HEIGHT_RATIO.COMPACT,
        imageUrl: "https://www.w3schools.com/colors/img_colormap.gif",
        subtitle: "subtitle",
        title: "title",
      },
      {
        buttons,
        defaultActionUrl: "https://www.example.com/imgs/imageExample.png",
        defaultMessengerExtensions: false,
        defaultWebviewHeightRatio: FACEBOOK_WEBVIEW_HEIGHT_RATIO.TALL,
        imageUrl: "https://www.w3schools.com/colors/img_colormap.gif",
        subtitle: "subtitle",
        title: "title",
      },
    ],
  };

  return carousel;
},
```

10.17.10 List

You need to include in your controller the following field: `facebookList`, which takes an object with an array of elements to be taken as items in a list of buttons. For more information about the carousel, check how to add a List Template.

```
const {
  FACEBOOK_BUTTONS,
  FACEBOOK_WEBVIEW_HEIGHT_RATIO,
  FACEBOOK_TOP_ELEMENT_STYLE,
  FacebookButtonTemplateBuilder,
  FacebookElementTemplateBuilder,
```

(continues on next page)

(continued from previous page)

```
  FacebookTemplateBuilder,
} = require('voxa');

app.onState('someState', () => {
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const elementBuilder1 = new FacebookElementTemplateBuilder();
  const elementBuilder2 = new FacebookElementTemplateBuilder();
  const elementBuilder3 = new FacebookElementTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setPayload("payload")
    .setTitle("View More")
    .setType(FACEBOOK_BUTTONS.POSTBACK);

  buttonBuilder2
    .setTitle("View")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/IMG_
↪6541-960x720_0.jpg")
    .setWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.FULL);

  elementBuilder1
    .addButton(buttonBuilder2.build())
    .setImageUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/
↪IMG_6541-960x720_0.jpg")
    .setSubtitle("See all our colors")
    .setTitle("Classic T-Shirt Collection");

  elementBuilder2
    .setDefaultActionUrl("https://www.w3schools.com")
    .setDefaultWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.TALL)
    .setImageUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/
↪IMG_6541-960x720_0.jpg")
    .setSubtitle("See all our colors")
    .setTitle("Classic T-Shirt Collection");

  elementBuilder3
    .addButton(buttonBuilder2.build())
    .setDefaultActionUrl("https://www.w3schools.com")
    .setDefaultWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.TALL)
    .setImageUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/
↪IMG_6541-960x720_0.jpg")
    .setSubtitle("100% Cotton, 200% Comfortable")
    .setTitle("Classic T-Shirt Collection");

  facebookTemplateBuilder
    .addButton(buttonBuilder1.build())
    .addElement(elementBuilder1.build())
    .addElement(elementBuilder2.build())
    .addElement(elementBuilder3.build())
    .setSharable(true)
    .setTopElementStyle(FACEBOOK_TOP_ELEMENT_STYLE.LARGE);

  return {
    facebookList: facebookTemplateBuilder.build(),
```

(continues on next page)

(continued from previous page)

```
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookList"
  };
});
.....
views
.....
{
  "FacebookList": {
    "facebookList": "{list}"
  }
}
.........
variables
.........
list: function list(request) {
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const elementBuilder1 = new FacebookElementTemplateBuilder();
  const elementBuilder2 = new FacebookElementTemplateBuilder();
  const elementBuilder3 = new FacebookElementTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setPayload("payload")
    .setTitle("View More")
    .setType(FACEBOOK_BUTTONS.POSTBACK);

  buttonBuilder2
    .setTitle("View")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/IMG_
↪6541-960x720_0.jpg")
    .setWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.FULL);

  elementBuilder1
    .addButton(buttonBuilder2.build())
    .setImageUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/
↪IMG_6541-960x720_0.jpg")
    .setSubtitle("See all our colors")
    .setTitle("Classic T-Shirt Collection");

  elementBuilder2
    .setDefaultActionUrl("https://www.w3schools.com")
    .setDefaultWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.TALL)
    .setImageUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/
↪IMG_6541-960x720_0.jpg")
    .setSubtitle("See all our colors")
    .setTitle("Classic T-Shirt Collection");

  elementBuilder3
```

(continues on next page)

(continued from previous page)

```
    .addButton(buttonBuilder2.build())
    .setDefaultActionUrl("https://www.w3schools.com")
    .setDefaultWebviewHeightRatio(FACEBOOK_WEBVIEW_HEIGHT_RATIO.TALL)
    .setImageUrl("https://www.scottcountyiowa.com/sites/default/files/images/pages/
↪IMG_6541-960x720_0.jpg")
    .setSubtitle("100% Cotton, 200% Comfortable")
    .setTitle("Classic T-Shirt Collection");

  facebookTemplateBuilder
    .addButton(buttonBuilder1.build())
    .addElement(elementBuilder1.build())
    .addElement(elementBuilder2.build())
    .addElement(elementBuilder3.build())
    .setSharable(true)
    .setTopElementStyle(FACEBOOK_TOP_ELEMENT_STYLE.LARGE);

  return facebookTemplateBuilder.build();
},
```

10.17.11 Button Template

You need to include in your controller the following field: `facebookButtonTemplate`, which takes an object with an array of buttons to be taken as items in a list of buttons. For more information about the button template, check how to add a Button Template.

```
const {
  FACEBOOK_BUTTONS,
  FacebookButtonTemplateBuilder,
  FacebookTemplateBuilder,
} = require('voxa');

app.onState('someState', () => {
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const buttonBuilder3 = new FacebookButtonTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setPayload("payload")
    .setTitle("View More")
    .setType(FACEBOOK_BUTTONS.POSTBACK);

  buttonBuilder2
    .setPayload("1234567890")
    .setTitle("Call John")
    .setType(FACEBOOK_BUTTONS.PHONE_NUMBER);

  buttonBuilder3
    .setTitle("Go to Twitter")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("http://www.twitter.com");

  facebookTemplateBuilder
    .addButton(buttonBuilder1.build())
    .addButton(buttonBuilder2.build())
```

(continues on next page)

(continued from previous page)

```
    .addButton(buttonBuilder3.build())
    .setText("What do you want to do?");

  return {
    facebookButtonTemplate: facebookTemplateBuilder.build(),
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookButtonTemplate"
  };
});
.....
views
.....
{
  "FacebookButtonTemplate": {
    "facebookButtonTemplate": "{buttonTemplate}"
  }
}
.........
variables
.........
buttonTemplate: function buttonTemplate(request) {
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const buttonBuilder3 = new FacebookButtonTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setPayload("payload")
    .setTitle("View More")
    .setType(FACEBOOK_BUTTONS.POSTBACK);

  buttonBuilder2
    .setPayload("1234567890")
    .setTitle("Call John")
    .setType(FACEBOOK_BUTTONS.PHONE_NUMBER);

  buttonBuilder3
    .setTitle("Go to Twitter")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("http://www.twitter.com");

  facebookTemplateBuilder
    .addButton(buttonBuilder1.build())
    .addButton(buttonBuilder2.build())
    .addButton(buttonBuilder3.build())
    .setText("What do you want to do?");

  return facebookTemplateBuilder.build();
},
```

10.17.12 Open Graph Template

You need to include in your controller the following field: `facebookOpenGraphTemplate`, which takes an object with an array of buttons to be taken as items in a list of buttons and a url for the open graph link. For more information about the button template, check how to add a Open Graph Template.

```
const {
  FACEBOOK_BUTTONS,
  FacebookButtonTemplateBuilder,
  FacebookTemplateBuilder,
} = require('voxa');

app.onState('someState', () => {
  const elementBuilder1 = new FacebookElementTemplateBuilder();
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setTitle("Go to Wikipedia")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("https://en.wikipedia.org/wiki/Rickrolling");

  buttonBuilder2
    .setTitle("Go to Twitter")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("http://www.twitter.com");

  elementBuilder1
    .addButton(buttonBuilder1.build())
    .addButton(buttonBuilder2.build())
    .setUrl("https://open.spotify.com/track/7GhIk7Il098yCjg4BQjzvb");

  facebookTemplateBuilder
    .addElement(elementBuilder1.build());

  return {
    facebookOpenGraphTemplate: facebookTemplateBuilder.build(),
  };
});
```

Or you can also handle these values from your views file

```
app.onState('someState', () => {
  return {
    reply: "FacebookOpenGraphTemplate"
  };
});
.....
views
.....
{
  "FacebookOpenGraphTemplate": {
    "facebookOpenGraphTemplate": "{openGraphTemplate}"
  }
}
.........
variables
```

(continues on next page)

(continued from previous page)

```
.........
openGraphTemplate: function openGraphTemplate(request) {
  const elementBuilder1 = new FacebookElementTemplateBuilder();
  const buttonBuilder1 = new FacebookButtonTemplateBuilder();
  const buttonBuilder2 = new FacebookButtonTemplateBuilder();
  const facebookTemplateBuilder = new FacebookTemplateBuilder();

  buttonBuilder1
    .setTitle("Go to Wikipedia")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("https://en.wikipedia.org/wiki/Rickrolling");

  buttonBuilder2
    .setTitle("Go to Twitter")
    .setType(FACEBOOK_BUTTONS.WEB_URL)
    .setUrl("http://www.twitter.com");

  elementBuilder1
    .addButton(buttonBuilder1.build())
    .addButton(buttonBuilder2.build())
    .setUrl("https://open.spotify.com/track/7GhIk7Il098yCjg4BQjzvb");

  facebookTemplateBuilder
    .addElement(elementBuilder1.build());

  return facebookTemplateBuilder.build();
},
```

For more information check the Dialogflow documentation for Facebook Messenger

Telegram

The `DialogflowPlatform` for voxa can be easily integrated with telegram, just make sure to use `text` responses in your controllers and everything should work as usual.

For more information check the Dialogflow documentation for telegram

10.18 Alexa APIs

Amazon has integrated several APIs so users can leverage the Alexa's configurations, device's and user's information.

10.18.1 Customer Contact Information Reference

When a customer enables your Alexa skill, your skill can request the customer's permission to the their contact information, which includes name, email address and phone number, if the customer has consented. You can then use this data to support personalized intents to enhance the customer experience without account linking. For example, your skill may use customer contact information to make a reservation at a nearby restaurant and send a confirmation to the customer.

class CustomerContact(*alexaEvent*)

Arguments

- **alexaEvent** (`VoxaEvent.rawEvent`) – Alexa Event object.

`CustomerContact.getEmail()`
Gets user's email

Returns string A string with user's email address

`CustomerContact.getGivenName()`
Gets user's given name

Returns string A string with user's given name

`CustomerContact.getName()`
Gets user's full name

Returns string A string with user's full name

`CustomerContact.getPhoneNumber()`
Gets user's phone number

Returns object A JSON object with user's phone number and country code

`CustomerContact.getFullUserInformation()`
Gets name or given name, phone number, and email address

Returns object A JSON object with user's info with the following structure

```
{
  "countryCode": "string",
  "email": "string",
  "givenName": "string",
  "name": "string",
  "phoneNumber": "string"
}
```

With Voxa, you can ask for the user's full name like this:

```
app.onIntent('FullAddressIntent', async (voxaEvent) => {
  const name = await voxaEvent.alexa.customerContact.getName();

  voxaEvent.model.name = name;
  return { ask: 'CustomerContact.Name' };
});
```

Voxa also has a method to request all parameters at once:

```
app.onIntent('FullAddressIntent', async (voxaEvent) => {
  const info = await voxaEvent.alexa.customerContact.getFullUserInformation();
  const { countryCode, email, name, phoneNumber } = info;

  voxaEvent.model.countryCode = countryCode;
  voxaEvent.model.email = email;
  voxaEvent.model.name = name;
  voxaEvent.model.phoneNumber = phoneNumber;

  return { ask: 'CustomerContact.FullInfo' };
});
```

To send a card requesting user the permission to access their information, you can simply add the card object to the view in your *views.js* file with the following format:

```
ContactPermission: {
  tell: 'Before accessing your information, you need to give me permission. Go to␣
↪your Alexa app, I just sent a link.',
  card: {
    type: 'AskForPermissionsConsent',
    permissions: [
      'alexa::profile:name:read',
      'alexa::profile:email:read',
      'alexa::profile:mobile_number:read'
    ],
  },
},
```

10.18.2 Device Address Information Reference

When a customer enables your Alexa skill, your skill can obtain the customer's permission to use address data associated with the customer's Alexa device. You can then use this address data to provide key functionality for the skill, or to enhance the customer experience. For example, your skill could provide a list of nearby store locations or provide restaurant recommendations using this address information. This document describes how to enable this capability and query the Device Address API for address data.

Note that the address entered in the Alexa device may not represent the current physical address of the device. This API uses the address that the customer has entered manually in the Alexa app, and does not have any capability of testing for GPS or other location-based data.

class DeviceAddress(*alexaEvent*)

Arguments

- **alexaEvent** (VoxaEvent.rawEvent) – Alexa Event object.

DeviceAddress.**getAddress**()
Gets full address info

Returns object A JSON object with the full address info

DeviceAddress.**getCountryRegionPostalCode**()
Gets country/region and postal code

Returns object A JSON object with country/region info

With Voxa, you can ask for the full device's address like this:

```
app.onIntent('FullAddressIntent', async (voxaEvent) => {
  const info = await voxaEvent.alexa.deviceAddress.getAddress();

  voxaEvent.model.deviceInfo = `${info.addressLine1}, ${info.city}, ${info.
↪countryCode}`;
  return { ask: 'DeviceAddress.FullAddress' };
});
```

You can decide to only get the country/region and postal code. You can do it this way:

```
app.onIntent('PostalCodeIntent', async (voxaEvent) => {
  const info = await voxaEvent.alexa.deviceAddress.getCountryRegionPostalCode();

  voxaEvent.model.deviceInfo = `${info.postalCode}, ${info.countryCode}`;
  return { ask: 'DeviceAddress.PostalCode' };
});
```

To send a card requesting user the permission to access the device address info, you can simply add the card object to the view in your *views.js* file with the following format:

```
FullAddressPermision: {
  tell: 'Before accessing your full address, you need to give me permission. Go to␣
↪your Alexa app, I just sent a link.',
  card: {
    type: 'AskForPermissionsConsent',
    permissions: [
      'read::alexa:device:all:address',
    ],
  },
},

PostalCodePermission: {
  tell: 'Before accessing your postal code, you need to give me permission. Go to␣
↪your Alexa app, I just sent a link.',
  card: {
    type: 'AskForPermissionsConsent',
    permissions: [
      'read::alexa:device:all:address:country_and_postal_code',
    ],
  },
},
```

10.18.3 Device Settings Reference

Alexa customers can set their timezone, distance measuring unit, and temperature measurement unit in the Alexa app. The Alexa Settings APIs allow developers to retrieve customer preferences for these settings in a unified view.

class DeviceSettings(*voxaEvent*)

Arguments

- **alexaEvent** (VoxaEvent.rawEvent) – Alexa Event object.

DeviceSettings.**getDistanceUnits**()
Gets distance units

Returns string A string with the distance units

DeviceSettings.**getTemperatureUnits**()
Gets temperature units

Returns string A string with the temperature units

DeviceSettings.**getTimezone**()
Gets timezone

Returns string A string with the timezone value

DeviceSettings.**getSettings**()
Gets all settings details

Returns object A JSON object with device's info with the following structure

```
{
  "distanceUnits": "string",
  "temperatureUnits": "string",
```

(continues on next page)

(continued from previous page)

```
  "timezone": "string"
}
```

With Voxa, you can ask for the full device's address like this:

```
app.onIntent('FullSettingsIntent', async (voxaEvent) => {
  const info = await voxaEvent.alexa.deviceSettings.getSettings();

  voxaEvent.model.settingsInfo = `${info.distanceUnits}, ${info.temperatureUnits}, $
↪{info.timezone}`;
  return { ask: 'DeviceSettings.FullSettings' };
});
```

You don't need to request to the user the permission to access the device settings info.

10.18.4 In-Skill Purchases Reference

The in-skill purchasing feature enables you to sell premium content such as game features and interactive stories for use in skills with a custom interaction model.

Buying these products in a skill is seamless to a user. They may ask to shop products, buy products by name, or agree to purchase suggestions you make while they interact with a skill. Customers pay for products using the payment options associated with their Amazon account.

For more information about setting up ISP with the ASK CLI follow this link. And to understand what's the process behind the ISP requests and responses to the Alexa Service click here.

With Voxa, you can implement all ISP features like buying, refunding and upselling an item:

```
app.onIntent('BuyIntent', async (voxaEvent) => {
  const { productName } = voxaEvent.intent.params;
  const token = 'startState';
  const buyDirective = await voxaEvent.alexa.isp.buyByReferenceName(productName,␣
↪token);

  return { alexaConnectionsSendRequest: buyDirective };
});

app.onIntent('RefundIntent', async (voxaEvent) => {
  const { productName } = voxaEvent.intent.params;
  const token = 'startState';
  const buyDirective = await voxaEvent.alexa.isp.cancelByReferenceName(productName,␣
↪token);

  return { alexaConnectionsSendRequest: buyDirective };
});
```

You can also check if the ISP feature is allowed in a locale or the account is correctly setup in the markets ISP works just by checking with the *isAllowed()* function.

```
app.onIntent('UpsellIntent', async (voxaEvent) => {
  if (!voxaEvent.alexa.isp.isAllowed()) {
    return { ask: 'ISP.Invalid', to: 'entry' };
  }

  const { productName } = voxaEvent.intent.params;
```

(continues on next page)

(continued from previous page)

```
  const token = 'startState';
  const buyDirective = await voxaEvent.alexa.isp.upsellByReferenceName(productName,␣
↪upsellMessage, token);

  return { alexaConnectionsSendRequest: buyDirective };
});
```

To get the full list of products and know which ones have been purchased, you can do it like this:

```
app.onIntent('ProductsIntent', async (voxaEvent) => {
  const list = await voxaEvent.alexa.isp.getProductList();

  voxaEvent.model.productArray = list.inSkillProducts.map(x => x.referenceName);

  return { ask: 'Products.List', to: 'entry' };
});
```

When users accept or refuse to buy/cancel an item, Alexa sends a Connections.Response directive. A very simple example on how the Connections.Response JSON request from Alexa looks like is:

```
{
  "type": "Connections.Response",
  "requestId": "string",
  "timestamp": "string",
  "name": "Upsell",
  "status": {
    "code": "string",
    "message": "string"
  },
  "payload": {
    "purchaseResult": "ACCEPTED",
    "productId": "string",
    "message": "optional additional message"
  },
  "token": "string"
}
```

10.18.5 Alexa Shopping and To-Do Lists Reference

Alexa customers have access to two default lists: Alexa to-do and Alexa shopping. In addition, Alexa customer can create and manage custom lists in a skill that supports that.

Customers can review and modify their Alexa lists using voice through a device with Alexa or via the Alexa app. For example, a customer can tell Alexa to add items to the Alexa Shopping List at home, and then while at the store, view the items via the Alexa app, and check them off.

class Lists(*alexaEvent*)

Arguments

- **alexaEvent** (VoxaEvent.rawEvent) – Alexa Raw Event object.

Lists.**getDefaultShoppingList**()

Gets info for the Alexa default Shopping list

Returns Object A JSON object with the Shopping list info

Lists.**getDefaultToDoList**()
Gets info for the Alexa default To-Do list

Returns Object A JSON object with the To-Do list info

Lists.**getListMetadata**()
Gets list metadata for all user's lists including the default list

Returns Array An object array

Lists.**getListById**(*listId*, *status = 'active'*)
Gets specific list by id and status

Arguments

- **listId** – List ID.
- **status** – list status, defaults to active (only value accepted for now)

Returns Object A JSON object with the specific list info.

Lists.**getOrCreateList**(*name*)
Looks for a list by name and returns it, if it is not found, it creates a new list with that name and returns it.

Arguments

- **name** – List name.

Returns Object A JSON object with the specific list info.

Lists.**createList**(*name*, *state = 'active'*)
Creates a new list with the name and state.

Arguments

- **name** – List name.
- **active** – list status, defaults to active (only value accepted for now)

Returns Object A JSON object with the specific list info.

Lists.**updateList**(*listId*, *name*, *state = 'active'*, *version*)
Updates list with the name, state, and version.

Arguments

- **listId** – List ID.
- **state** – list status, defaults to active (only value accepted for now)
- **version** – List version.

Returns Object A JSON object with the specific list info.

Lists.**deleteList**(*listId*)
Deletes a list by ID.

Arguments

- **listId** – List ID.

Returns undefined. HTTP response with 200 or error if any.

Lists.**getListItem**(*listId*, *itemId*)
Creates a new list with the name and state.

Arguments

- **listId** – List ID.

- **itemId** – Item ID.

Returns Object A JSON object with the specific list info.

`Lists.`**`createItem`**(*listId*, *value*, *status = 'active'*)
Creates a new list with the name and state.

Arguments

- **listId** – List ID.
- **value** – Item name.
- **status** – item status, defaults to active. Other values accepted: 'completed'

Returns Object A JSON object with the specific item info.

`Lists.`**`updateItem`**(*listId*, *itemId*, *value*, *status*, *version*)
Creates a new list with the name and state.

Arguments

- **listId** – List ID.
- **itemId** – Item ID.
- **value** – Item name.
- **status** – Item status. Values accepted: 'active | completed'

Returns Object A JSON object with the specific item info.

`Lists.`**`deleteItem`**(*listId*, *itemId*)
Creates a new list with the name and state.

Arguments

- **listId** – List ID.
- **itemId** – Item ID.

Returns undefined. HTTP response with 200 or error if any.

With Voxa, you can implement all lists features. In this code snippet you will see how to check if a list exists, if not, it creates one. If it does exist, it will check if an item is already in the list and updates the list with a new version, if no, it adds it:

```
app.onIntent('AddItemToListIntent', async (voxaEvent) => {
  const { productName } = voxaEvent.intent.params;
  const listsMetadata = await voxaEvent.alexa.lists.getListMetadata();
  const listName = 'MY_CUSTOM_LIST';

  const listMeta = _.find(listsMetadata.lists, { name: listName });
  let itemInfo;
  let listInfo;

  if (listMeta) {
    listInfo = await voxaEvent.alexa.lists.getListById(listMeta.listId);
    itemInfo = _.find(listInfo.items, { value: productName });

    await voxaEvent.alexa.lists.updateList(listMeta.name, 'active', 2);
  } else {
    listInfo = await voxaEvent.alexa.lists.createList(listName);
  }
```

(continues on next page)

(continued from previous page)

```
  if (itemInfo) {
    return { ask: 'List.ProductAlreadyInList' };
  }

  await voxaEvent.alexa.lists.createItem(listInfo.listId, productName);

  return { ask: 'List.ProductCreated' };
});
```

There's also a faster way to consult and/or create a list. Follow this example:

```
app.onIntent('AddItemToListIntent', async (voxaEvent) => {
  const { productName } = voxaEvent.intent.params;
  const listName = 'MY_CUSTOM_LIST';

  const listInfo = await voxaEvent.alexa.lists.getOrCreateList(listName);
  const itemInfo = _.find(listInfo.items, { value: productName });

  if (itemInfo) {
    return { ask: 'List.ProductAlreadyInList' };
  }

  await voxaEvent.alexa.lists.createItem(listInfo.listId, productName);

  return { ask: 'List.ProductCreated' };
});
```

Let's review another example. Let's say we have an activity in the default To-Do list and we want to mark it as completed. For that, we need to pull down the items from the default To-Do list, find our item and modify it:

```
app.onIntent('CompleteActivityIntent', async (voxaEvent) => {
  const { activity } = voxaEvent.intent.params;

  const listInfo = await voxaEvent.alexa.lists.getDefaultToDoList();
  const itemInfo = _.find(listInfo.items, { value: activity });

  await voxaEvent.alexa.lists.updateItem(
    listInfo.listId,
    itemInfo.id,
    activity,
    'completed',
    2);

  return { ask: 'List.ActivityCompleted' };
});
```

Let's check another example. Let's say users want to remove an item in their default shopping list that they had already marked as completed. We're going to first fetch the default shopping list's info, then look for the product to remove, we're going to first check if the product is marked as completed to then delete it:

```
app.onIntent('RemoveProductIntent', async (voxaEvent) => {
  const { productId } = voxaEvent.model;

  const listInfo = await voxaEvent.alexa.lists.getDefaultShoppingList();
  const itemInfo = await voxaEvent.alexa.lists.getListItem(listInfo.listId,␣
↪productId);
```

(continues on next page)

(continued from previous page)

```
  if (itemInfo.status === 'active') {
    return { ask: 'List.ConfirmProductDeletion', to: 'wantToDeleteActiveProduct?' };
  }

  await voxaEvent.alexa.lists.deleteItem(listInfo.listId, productId);

  return { ask: 'List.ProductRemoved' };
});
```

Finally, if you want to remove the list you had created:

```
app.onIntent('DeleteListIntent', async (voxaEvent) => {
  const listName = 'MY_CUSTOM_LIST';

  const listInfo = await voxaEvent.alexa.lists.getOrCreateList(listName);
  await voxaEvent.alexa.lists.deleteList(listInfo.listId);

  return { ask: 'List.ListRemoved' };
});
```

To send a card requesting user the permission to read/write Alexa lists, you can simply add the card object to the view in your *views.js* file with the following format:

```
NeedShoppingListPermission: {
  tell: 'Before adding an item to your list, you need to give me permission. Go to␣
↪your Alexa app, I just sent a link.',
  card: {
    type: 'AskForPermissionsConsent',
    permissions: [
      'read::alexa:household:list',
      'write::alexa:household:list',
    ],
  },
},
```

10.18.6 Alexa Reminders API Reference

Use the Alexa Reminders API to create and manage reminders from your skill. This reference describes the available operations for the Alexa Reminders API.

Note that you need to modify your skill manifest by adding the reminder permission:

```
{
  "permissions": [
    {
      "name": "alexa::alerts:reminders:skill:readwrite"
    }
  ],
}
```

class Reminders(*alexaEvent*)

Arguments

- **alexaEvent** (VoxaEvent.rawEvent) – Alexa Event object.

Reminders.**getReminder**()
Gets a reminder

Arguments

- **alertToken** – Reminder's ID.

Returns object A JSON object with the reminder's details

Reminders.**getAllReminders**()
Gets all reminders

Returns object A JSON object with an array of the reminder's details

Reminders.**createReminder**(*reminder*)
Creates a reminder

Arguments

- **reminder** – Reminder Builder Object.

Returns object A JSON object with the details of reminder's creation

Reminders.**updateReminder**(*alertToken*, *reminder*)
Updates a reminder

Arguments

- **alertToken** – Reminder's ID.
- **reminder** – Reminder Builder Object.

Returns object A JSON object with the details of reminder's update

Reminders.**deleteReminder**(*alertToken*)
Deletes a reminder

Arguments

- **alertToken** – Reminder's ID.

Returns object A JSON object with the details of reminder's deletion

class ReminderBuilder()

ReminderBuilder.**setCreatedTime**(*createdTime*)
Sets created time

Arguments

- **createdTime** – Reminder's creation time.

Returns object A ReminderBuilder object

ReminderBuilder.**setRequestTime**(*requestTime*)
Sets request time

Arguments

- **requestTime** – Reminder's request time.

Returns object A ReminderBuilder object

ReminderBuilder.**setTriggerAbsolute**(*scheduledTime*)
Sets the reminder trigger as absolute

Arguments

- **scheduledTime** – Reminder's scheduled time.

Returns object A ReminderBuilder object

ReminderBuilder.**setTriggerRelative**(*offsetInSeconds*)
Sets the reminder trigger as relative

Arguments

- **offsetInSeconds** – Reminder's offset in seconds.

Returns object A ReminderBuilder object

ReminderBuilder.**setTimeZoneId**(*timeZoneId*)
Sets time zone Id

Arguments

- **timeZoneId** – Reminder's time zone.

Returns object A ReminderBuilder object

ReminderBuilder.**setRecurrenceFreqDaily**()
Sets reminder's recurrence frequence to "DAILY"

Returns object A ReminderBuilder object

ReminderBuilder.**setRecurrenceFreqWeekly**()
Sets reminder's recurrence frequence to "WEEKLY"

Returns object A ReminderBuilder object

ReminderBuilder.**setRecurrenceByDay**(*recurrenceByDay*)
Sets frequency by day

Arguments

- **recurrenceByDay** – Array of frequency by day.

Returns object A ReminderBuilder object

ReminderBuilder.**setRecurrenceInterval**(*interval*)
Sets reminder's interval

Arguments

- **interval** – Reminder's interval

Returns object A ReminderBuilder object

ReminderBuilder.**addContent**(*locale*, *text*)
Sets reminder's content

Arguments

- **locale** – Reminder's locale
- **text** – Reminder's text

Returns object A ReminderBuilder object

ReminderBuilder.**enablePushNotification**()
Sets reminder's push notification status to "ENABLED"

Returns object A ReminderBuilder object

ReminderBuilder.**disablePushNotification**()
Sets reminder's push notification status to "DISABLED"

Returns object A ReminderBuilder object

With Voxa, you can create, update, delete and get reminders like this:

```
const { ReminderBuilder } = require("voxa");

app.onIntent('CreateReminderIntent', async (voxaEvent) => {
  const reminder = new ReminderBuilder()
    .setCreatedTime("2018-12-11T14:05:38.811")
    .setTriggerAbsolute("2018-12-12T12:00:00.000")
    .setTimeZoneId("America/Denver")
    .setRecurrenceFreqDaily()
    .addContent("en-US", "CREATION REMINDER TEST")
    .enablePushNotification();

  const reminderResponse = await voxaEvent.alexa.reminders.createReminder(reminder);

  voxaEvent.model.reminder = reminderResponse;
  return { tell: "Reminder.Created" };
});

app.onIntent('UpdateReminderIntent', async (voxaEvent) => {
  const alertToken = '1234-5678-9012-3456';
  const reminder = new ReminderBuilder()
    .setRequestTime("2018-12-11T14:05:38.811")
    .setTriggerAbsolute("2018-12-12T12:00:00.000")
    .setTimeZoneId("America/Denver")
    .setRecurrenceFreqDaily()
    .addContent("en-US", "CREATION REMINDER TEST")
    .enablePushNotification();

  const reminderResponse = await voxaEvent.alexa.reminders.updateReminder(alertToken,␣
↪reminder);

  voxaEvent.model.reminder = reminderResponse;
  return { tell: "Reminder.Updated" };
});

app.onIntent('UpdateReminderIntent', async (voxaEvent) => {
  const alertToken = '1234-5678-9012-3456';
  const reminderResponse = await voxaEvent.alexa.reminders.deleteReminder(alertToken);

  return { tell: "Reminder.Deleted" };
});

app.onIntent('GetReminderIntent', async (voxaEvent) => {
  const alertToken = '1234-5678-9012-3456';
  const reminderResponse = await voxaEvent.alexa.reminders.getReminder(alertToken);

  voxaEvent.model.reminder = reminderResponse.alerts[0];
  return { tell: "Reminder.Get" };
});

app.onIntent('GetAllRemindersIntent', async (voxaEvent) => {
  const reminderResponse = await voxaEvent.alexa.reminders.getAllReminders();

  voxaEvent.model.reminders = reminderResponse.alerts;
  return { tell: "Reminder.Get" };
});
```

10.18.7 Skill Messaging API Reference

The Skill Messaging API is used to send message requests to skills. These methods are meant to work for out-of-session operations, so you will not likely use it in the skill code. However, you might have a separate file inside your Voxa project to work with some automated triggers like CloudWatch Events or SQS functions. In that case, your file has access to the voxa package, thus, you can take advantage of these methods.

class Messaging(*clientId*, *clientSecret*)

Arguments

- **clientId** – Client ID to call Messaging API.
- **clientSecret** – Client Secret to call Messaging API.

Messaging.**sendMessage**()
Sends message to a skill

Arguments

- **request** – Message request params with the following structure:

```
{
  endpoint: string, // User's endpoint.
  userId: string, // User's userId.
  data: any, // Object with key-value pairs to send to the skill.
  expiresAfterSeconds: number, // Expiration time in milliseconds, defaults to␣
↪3600 milliseconds.
}

:returns: undefined
```

In the following example, you'll see a simple code of a lambda function which calls your database to fetch users to whom you'll send a reminder with the Reminder API, the message is sent via Messaging API:

```
'use strict';

const Promise = require('bluebird');
const { Messaging } = require('voxa');

const Storage = require('./Storage');

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

exports.handler = async (event, context, callback) => {
  const usersOptedIn = await Storage.getUsers();
  const messaging = new Messaging(CLIENT_ID, CLIENT_SECRET);

  await Promise.map(usersOptedIn, (user) => {
    const data = {
      timezone: user.timezone,
      title: user.reminderTitle,
      when: user.reminderTime,
    };

    const request = {
      endpoint: user.endpoint,
      userId: user.userId,
      data,
```

(continues on next page)

(continued from previous page)

```
    };

    return messaging.sendMessage(request)
      .catch((err) => {
        console.log('ERROR SENDING MESSAGE', err);

        return null;
      });
  });

  callback(undefined, "OK");
};
```

This will dispatch a 'Messaging.MessageReceived' request to every user and you can handle the code in Voxa like this:

```
const { ReminderBuilder } = require("voxa");

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const reminderData = voxaEvent.rawEvent.request.message;

  const reminder = new ReminderBuilder()
    .setCreatedTime("2018-12-11T14:05:38.811")
    .setTriggerAbsolute(reminderData.when)
    .setTimeZoneId(reminderData.timezone)
    .setRecurrenceFreqDaily()
    .addContent("en-US", reminderData.title)
    .enablePushNotification();

  await voxaEvent.alexa.reminders.createReminder(reminder);

  return reply;
});
```

The main advantage of sending a message with the Messaging API is that it generates a new access token valid for 1 hour. This is important for out-of-session operations where you don't have access to a valid access token. The event sent to your skill now has a new access token valid for 1 hour. So now, you can use it to call any Alexa API that requires an access token in the authorization headers. The request object of the event looks like this:

```
{
  "request": {
    "type": "Messaging.MessageReceived",
    "requestId": "amzn1.echo-api.request.VOID",
    "timestamp": "2018-12-17T22:06:28Z",
    "message": {
      "name": "John"
    }
  }
}
```

10.18.8 Proactive Events API Reference

The ProactiveEvents API enables Alexa Skill Developers to send events to Alexa, which represent factual data that may interest a customer. Upon receiving an event, Alexa proactively delivers the information to customers subscribed to receive these events. This API currently supports one proactive channel, Alexa Notifications. As more proactive

channels are added in the future, developers will be able to take advantage of them without requiring integration with a new API.

class ProactiveEvents(*clientId*, *clientSecret*)

Arguments

- **clientId** – Client ID to call Messaging API.
- **clientSecret** – Client Secret to call Messaging API.

ProactiveEvents.**createEvent**()

Creates proactive event

Arguments

- **endpoint** – User's default endpoint
- **body** – Event's body
- **isDevelopment** – Flag to define if the event is sent to development stage. If false, then it goes to live skill

Returns undefined

This API is meant to work as an out-of-session task, so you'd need to use the Messaging API if you want to send a notification triggered by your server. The following examples show how you can use the different schemas to send proactive events (formerly Notifications):

- WeatherAlertEventsBuilder

```
const { ProactiveEvents, WeatherAlertEventsBuilder } = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: WeatherAlertEventsBuilder = new WeatherAlertEventsBuilder();
  event
    .setHurricane()
    .addContent("en-US", "source", eventData.localizedValue)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- SportsEventBuilder

```
const { ProactiveEvents, SportsEventBuilder } = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
```

(continues on next page)

(continued from previous page)

```
  const eventData = voxaEvent.rawEvent.request.message;

  const event: SportsEventBuilder = new SportsEventBuilder();
  event
    .setAwayTeamStatistic("Boston Red Sox", 5)
    .setHomeTeamStatistic("New York Yankees", 2)
    .setUpdate("Boston Red Sox", 5)
    .addContent("en-US", "eventLeagueName", eventData.localizedValue)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- MessageAlertEventBuilder

```
const {
  MESSAGE_ALERT_FRESHNESS,
  MESSAGE_ALERT_STATUS,
  MESSAGE_ALERT_URGENCY,
  MessageAlertEventBuilder,
  ProactiveEvents,
} = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: MessageAlertEventBuilder = new MessageAlertEventBuilder();
  event
    .setMessageGroup(eventData.creatorName, eventData.count, MESSAGE_ALERT_URGENCY.
↪URGENT)
    .setState(MESSAGE_ALERT_STATUS.UNREAD, MESSAGE_ALERT_FRESHNESS.NEW)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- OrderStatusEventBuilder

```
const {
  ORDER_STATUS,
  OrderStatusEventBuilder,
  ProactiveEvents,
```

(continues on next page)

(continued from previous page)

```
} = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: OrderStatusEventBuilder = new OrderStatusEventBuilder();
  event
    .setStatus(ORDER_STATUS.ORDER_DELIVERED, eventData.expectedArrival, eventData.
↪enterTimestamp)
    .addContent("en-US", "sellerName", eventData.localizedValue)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- OccasionEventBuilder

```
const {
  OCCASION_CONFIRMATION_STATUS,
  OCCASION_TYPE,
  OccasionEventBuilder,
  ProactiveEvents,
} = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: OccasionEventBuilder = new OccasionEventBuilder();
  event
    .setOccasion(eventData.bookingType, OCCASION_TYPE.APPOINTMENT)
    .setStatus(OCCASION_CONFIRMATION_STATUS.CONFIRMED)
    .addContent("en-US", "brokerName", eventData.brokerName)
    .addContent("en-US", "providerName", eventData.providerName)
    .addContent("en-US", "subject", eventData.subject)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- TrashCollectionAlertEventBuilder

```
const {
  GARBAGE_COLLECTION_DAY,
  GARBAGE_TYPE,
  TrashCollectionAlertEventBuilder,
  ProactiveEvents,
} = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: TrashCollectionAlertEventBuilder = new␣
↪TrashCollectionAlertEventBuilder();
  event
    .setAlert(GARBAGE_COLLECTION_DAY.MONDAY,
      GARBAGE_TYPE.BOTTLES,
      GARBAGE_TYPE.BULKY,
      GARBAGE_TYPE.CANS,
      GARBAGE_TYPE.CLOTHING)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- MediaContentEventBuilder

```
const {
  MEDIA_CONTENT_METHOD,
  MEDIA_CONTENT_TYPE,
  MediaContentEventBuilder,
  ProactiveEvents,
} = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: MediaContentEventBuilder = new MediaContentEventBuilder();
  event
    .setAvailability(MEDIA_CONTENT_METHOD.AIR)
    .setContentType(MEDIA_CONTENT_TYPE.ALBUM)
    .addContent("en-US", "providerName", eventData.providerName)
    .addContent("en-US", "contentName", eventData.contentName)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
```

(continues on next page)

(continued from previous page)

```
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

- SocialGameInviteEventBuilder

```
const {
  SOCIAL_GAME_INVITE_TYPE,
  SOCIAL_GAME_RELATIONSHIP_TO_INVITEE,
  SocialGameInviteEventBuilder,
  ProactiveEvents,
} = require("voxa");

const CLIENT_ID = 'CLIENT_ID';
const CLIENT_SECRET = 'CLIENT_SECRET';

app["onMessaging.MessageReceived"](async (voxaEvent, reply) => {
  const eventData = voxaEvent.rawEvent.request.message;

  const event: SocialGameInviteEventBuilder = new SocialGameInviteEventBuilder();
  event
    .setGame(SOCIAL_GAME_OFFER.GAME)
    .setInvite(eventData.name, SOCIAL_GAME_INVITE_TYPE.CHALLENGE, SOCIAL_GAME_
↪RELATIONSHIP_TO_INVITEE.CONTACT)
    .addContent("en-US", "gameName", eventData.localizedValue)
    .setReferenceId(eventData.referenceId)
    .setTimestamp(eventData.timestamp)
    .setExpiryTime(eventData.expiryTime)
    .setUnicast(eventData.userId);

  const proactiveEvents = new ProactiveEvents(CLIENT_ID, CLIENT_SECRET);
  await proactiveEvents.createEvent(endpoint, event, true);

  return reply;
});
```

10.19 CanFulfillIntentRequest

Name-free interaction enables customers to interact with Alexa without invoking a specific skill by name, which helps facilitate greater interaction with Alexa because customers do not always know which skill is appropriate.

When Alexa receives a request from a customer without a skill name, such as "Alexa, play relaxing sounds with crickets," Alexa looks for skills that might fulfill the request. Alexa determines the best choice among eligible skills and hands the request to the skill.

To make your skill more discoverable for name-free interaction, you can implement the the CanFulfillIntentRequest interface in your skill.

In Voxa, you can take advantage of this feature by following this example:

```
app.onCanFulfillIntentRequest((alexaEvent, reply) => {
  if (alexaEvent.intent.name === 'InfluencerIntent') {
    reply.fulfillIntent('YES');

    _.each(alexaEvent.intent.params, (value, slotName) => {
      reply.fulfillSlot(slotName, 'YES', 'YES');
    });
  }

  return reply;
});
```

Voxa offers the function **'onCanFulfillIntentRequest'_** so you can implement it in your code to validate wether you're going to fulfill the request or not.

Additionally, if you have several intents that you want to automatically fulfill, regardless of the slot values in the request, you can simply add an array of intents to the property: **'defaultFulfillIntents'_** of the Voxa config file:

```
const defaulFulfillIntents = [
  'NameIntent',
  'PhoneIntent',
];

const voxaApp = new VoxaApp({ views, variables });
const alexaSkill = new AlexaPlatform(voxaApp, { defaultFulfillIntents });
```

If Alexa sends an intent that you didn't register with this function, then you should implement the **'onCanFulfillIntentRequest'_** method to handle it. Important: If you implement this method in your skill, you should always return the **'reply'_** object.

If a skill has implemented canFulfillIntent according to the interface specification, the skill should be aware that the skill is not yet being asked to take action on behalf of the customer, and should not modify any state outside its scope, or have any observable interaction with its own calling functions or the outside world besides returning a value. Thus, the skill should not, at this point, perform any actions such as playing sound, turning lights on or off, providing feedback to the customer, committing a transaction, or making a state change.

10.20 Testing using ASK-CLI

There are 2 options to test this feature manually. The first one is using the *Manual JSON* section of the *Test* tab in the developer portal. And the other one, is to use the ASK CLI from Amazon.

You can just trigger this command in the console, and you'll get the result in your terminal:

```
$ ask api invoke-skill --skill-id amzn1.ask.skill.[unique-value-here] --file /path/to/
↪input/json --endpoint-region [endpoint-region-here]
```

10.21 LoginWithAmazon

With LoginWithAmazon, you can request a customer profile that contains the data that Login with Amazon applications can access regarding a particular customer. This includes: a unique ID for the user; the user's name, the user's email address, and their postal code. This data is divided into three scopes: profile, profile:user_id and postal_code.

LoginWithAmazon works a seamless solution to get user's information using account linking via web browser. To see more information about it, follow this link. To implement LoginWithAmazon in your Alexa skill, follow this step-by-step tutorial. You can also do account linking via voice. Go *here* to check it out!

With Voxa, you can ask for the user's full name like this:

```
app.onIntent('ProfileIntent', async (alexaEvent: AlexaEvent) => {
  const userInfo = await alexaEvent.getUserInformation();

  alexaEvent.model.email = userInfo.email;
  alexaEvent.model.name = userInfo.name;
  alexaEvent.model.zipCode = userInfo.zipCode;

  return { ask: 'CustomerContact.FullInfo' };
});
```

In this case, Voxa will detect you're running an Alexa Skill, so, it will call the *getUserInformationWithLWA()* method. But you can also call it directly. Even if you create voice experiences in other platforms like Google or Cortana, you can take advantage of the methods for authenticating with other platforms.

10.22 Google Sign-In

Google Sign-In for the Assistant provides the simplest and easiest user experience to users and developers both for account linking and account creation. Your Action can request access to your user's Google profile during a conversation, including the user's name, email address, and profile picture.

The profile information can be used to create a personalized user experience in your Action. If you have apps on other platforms and they use Google Sign-In, you can also find and link to an existing user's account, create a new account, and establish a direct channel of communication to the user.

To perform account linking with Google Sign-In, you ask the user to give consent to access their Google profile. You then use the information in their profile, for example their email address, to identify the user in your system. Check out this link for more information.

With Voxa, you can ask for the user's full name like this:

```
app.onIntent('ProfileIntent', async (googleAssistantEvent: GoogleAssistantEvent) => {
  const userInfo = await googleAssistantEvent.getUserInformation();

  googleAssistantEvent.model.email = userInfo.email;
  googleAssistantEvent.model.familyName = userInfo.familyName;
  googleAssistantEvent.model.givenName = userInfo.givenName;
  googleAssistantEvent.model.name = userInfo.name;
  googleAssistantEvent.model.locale = userInfo.locale;

  return { ask: 'CustomerContact.FullInfo' };
});
```

In this case, Voxa will detect you're running a Google Action, so, it will call the *getUserInformationWithGoogle()* method. Since this is a Google-only API, you can't use this method on other platforms for the moment.

10.23 The `reply` Object

class IVoxaReply()

The `reply` object is used by the framework to render voxa responses, it takes all of your `statements`,

`cards` and `directives` and generates a proper json response for each platform.

`IVoxaReply.IVoxaReply.`**`clear`**`()`
Resets the response object

`IVoxaReply.IVoxaReply.`**`terminate`**`()`
Sends a flag to indicate the session will be closed.

`IVoxaReply.IVoxaReply.`**`addStatement`**`(`*statement*, *isPlain*`)`
Adds statements to the `Reply`

Arguments

- **`statement`** – The string to be spoken by the voice assistant
- **`isPlain`** – Indicates if the statement is plain text, if null, it means is SSML

`IVoxaReply.IVoxaReply.`**`addReprompt`**`(`*statement*, *isPlain*`)`
Adds the reprompt text to the `Reply`

Arguments

- **`statement`** – The string to be spoken by the voice assistant as a reprompt
- **`isPlain`** – Indicates if the statement is plain text, if null, it means is SSML

`IVoxaReply.IVoxaReply.`**`hasDirective`**`()`
Verifies if the reply has directives

Returns A boolean flag indicating if the reply object has any kind of directives

`IVoxaReply.IVoxaReply.`**`saveSession`**`(`*event*`)`
Converts the model object into session attributes

Arguments

- **`event`** – A Voxa event with session attributes

For the speceific classes used in every platform you can check:

10.23.1 The `AlexaReply` Object

`class AlexaReply``()`
`AlexaReply` 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

`AlexaReply.Reply.`**`fulfillIntent`**`(`*canFulfill*`)`
Fulfills the request in the response object

Arguments

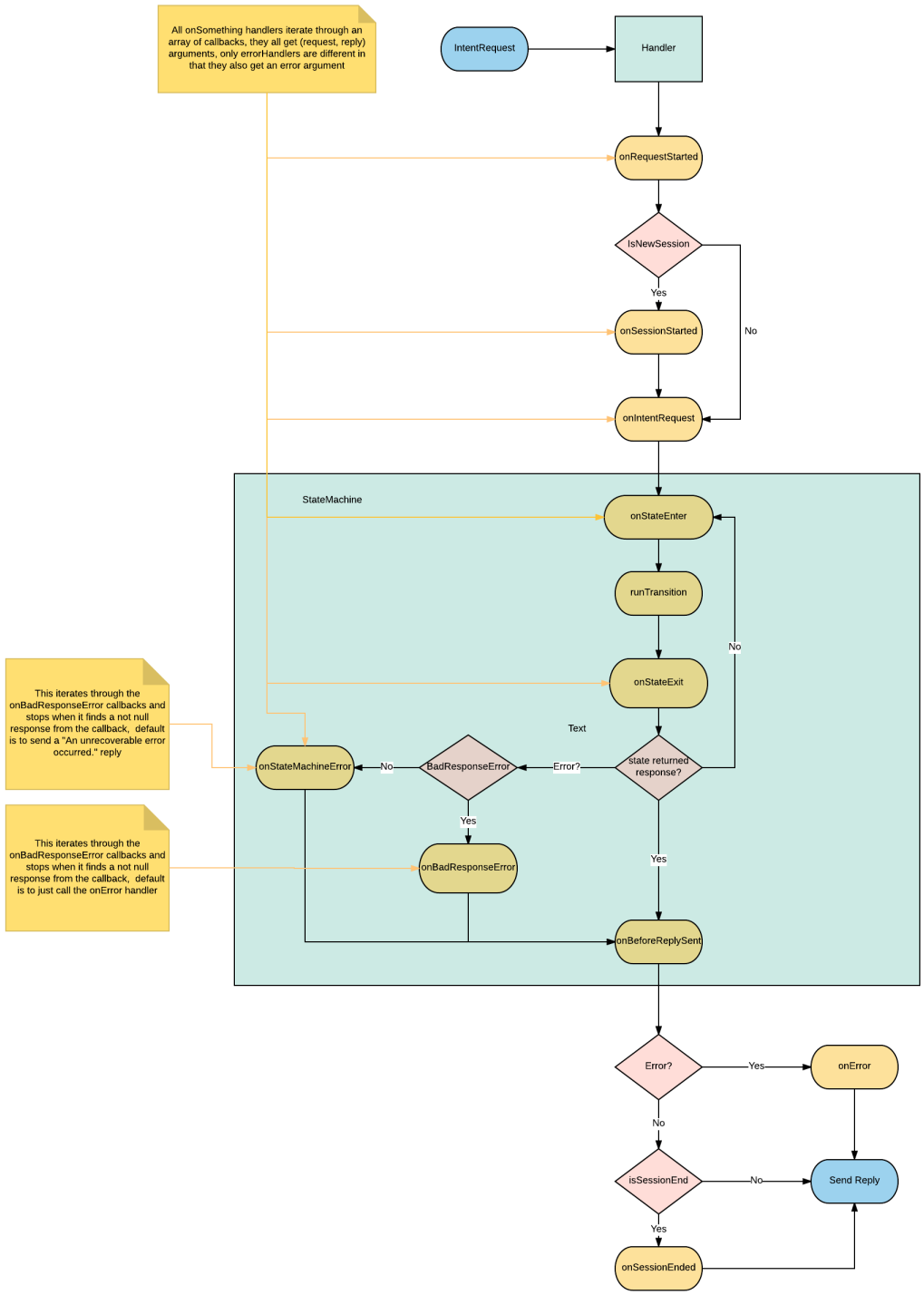
- **`canFulfill`** – A string with possible values: YES | NO | MAYBE to fulfill request

`AlexaReply.Reply.`**`fulfillSlot`**`(`*slotName*, *canUnderstand*, *canFulfill*`)`
Fulfills the slot with fulfill and understand values

Arguments

- **`slotName`** – A string with the slot to fulfill
- **`canUnderstand`** – A string with possible values: YES | NO | MAYBE that indicates slot understanding
- **`canFulfill`** – A string with possible values: YES | NO that indicates slot fulfillment

10.24 Request Flow



10.25 Gadget Controller Interface Reference

The Gadget Controller interface enables your skill to control Echo Buttons. This interface works with compatible Amazon Echo devices only. With the Gadget Controller interface, you can send animations to illuminate the buttons with different colors in a specific order.

With Voxa, you can implement this interface like this:

```
const voxa = require('voxa');
const { GadgetController, TRIGGER_EVENT_ENUM } = voxa.alexa;

app.onIntent('GameEngine.InputHandlerEvent', (voxaEvent) => {
  // REMEMBER TO SAVE THE VALUE originatingRequestId in your model
  voxaEvent.model.originatingRequestId = voxaEvent.request.originatingRequestId;

  const gameEvents = voxaEvent.request.events[0] || [];
  const inputEvents = _(gameEvents.inputEvents)
    .groupBy('gadgetId')
    .map(value => value[0])
    .value();

  const directives = [];
  let customId = 0;

  _.forEach(inputEvents, (gadgetEvent) => {
    customId += 1;
    const id = `g${customId}`;

    if (!_.includes(voxaEvent.model.buttons, id)) {
      const buttonIndex = _.size(voxaEvent.model.buttons);
      const targetGadgets = [gadgetEvent.gadgetId];

      _.set(voxaEvent.model, `buttonIds.${id}`, gadgetEvent.gadgetId);

      voxaEvent.model.buttons = [];
      voxaEvent.model.buttons.push(id);

      const triggerEventTimeMs = 0;
      const gadgetController = new GadgetController();
      const animationBuilder = GadgetController.getAnimationsBuilder();
      const sequenceBuilder = GadgetController.getSequenceBuilder();

      sequenceBuilder
        .duration(1000)
        .blend(false)
        .color(COLORS[buttonIndex].dark);

      animationBuilder
        .repeat(100)
        .targetLights(['1'])
        .sequence([sequenceBuilder]);

      directives.push(gadgetController
        .setAnimations(animationBuilder)
        .setTriggerEvent(TRIGGER_EVENT_ENUM.NONE)
        .setLight(targetGadgets, triggerEventTimeMs));
```

(continues on next page)

(continued from previous page)

```
      const otherAnimationBuilder = GadgetController.getAnimationsBuilder();
      const otherSequenceBuilder = GadgetController.getSequenceBuilder();

      otherSequenceBuilder
        .duration(500)
        .blend(false)
        .color(COLORS[buttonIndex].hex);

      otherAnimationBuilder
        .repeat(1)
        .targetLights(['1'])
        .sequence([otherSequenceBuilder.build()]);

      directives.push(gadgetController
        .setAnimations(otherAnimationBuilder.build())
        .setTriggerEvent(TRIGGER_EVENT_ENUM.BUTTON_DOWN)
        .setLight(targetGadgets, triggerEventTimeMs));
    }
  });

  return {
    alexaGadgetControllerLightDirective: directives,
    tell: 'Buttons.Next',
    to: 'entry',
  };
});
```

If there's an error when you send this directive, Alexa will return a System ExceptionEncountered request.

A very simple example on how the GadgetController.SetLight JSON response looks like is:

```
{
  "version": "1.0",
  "sessionAttributes": {},
  "shouldEndSession": true,
  "response": {
    "outputSpeech": "outputSpeech",
    "reprompt": "reprompt",
    "directives": [
      {
        "type": "GadgetController.SetLight",
        "version": 1,
        "targetGadgets": [ "gadgetId1", "gadgetId2" ],
        "parameters": {
          "triggerEvent": "none",
          "triggerEventTimeMs": 0,
          "animations": [
            {
              "repeat": 1,
              "targetLights": ["1"],
              "sequence": [
                {
                  "durationMs": 10000,
                  "blend": false,
                  "color": "0000FF"
                }
              ]
```

(continues on next page)

(continued from previous page)

```
            }
          ]
        }
      }
    ]
  }
}
```

10.26 Game Engine Interface Reference

The Game Engine interface enables your skill to receive input from Echo Buttons. This interface works with compatible Amazon Echo devices only.

Your skill uses the Game Engine Interface by sending directives that start and stop the Input Handler, which is the component within Alexa that sends your skill Echo Button events when conditions that you define are met (for example, the user pressed a certain sequence of buttons).

With Voxa, you can implement this interface like this:

```
const voxa = require('voxa');
const { ANCHOR_ENUM, EVENT_REPORT_ENUM, GameEngine } = voxa.alexa;

app.onIntent('LaunchIntent', (voxaEvent) => {
  const alexaGameEngineStartInputHandler = rollCall(voxaEvent);

  return {
    alexaGameEngineStartInputHandler,
    tell: 'Buttons.Discover',
  };
});

function rollCall(voxaEvent) {
  const gameEngineTimeout = 15000;
  const eventBuilder = GameEngine.getEventsBuilder('sample_event');
  const timeoutEventBuilder = GameEngine.getEventsBuilder('timeout');
  const recognizerBuilder = GameEngine.getPatternRecognizerBuilder('sample_event');

  eventBuilder
    .fails(['fails'])
    .meets(['sample_event'])
    .maximumInvocations(1)
    .reports(EVENT_REPORT_ENUM.MATCHES)
    .shouldEndInputHandler(true)
    .triggerTimeMilliseconds(1000);

  timeoutEventBuilder
    .meets(['timed out'])
    .reports(EVENT_REPORT_ENUM.HISTORY)
    .shouldEndInputHandler(true);

  recognizerBuilder
    .actions('actions')
    .fuzzy(true)
    .gadgetIds(['gadgetIds'])
```

(continues on next page)

(continued from previous page)

```
    .anchor(ANCHOR_ENUM.ANYWHERE)
    .pattern(rollCallPattern);

  return gameEngine
    .setEvents(eventBuilder, timeoutEventBuilder.build())
    .setRecognizers(recognizerBuilder.build())
    .startInputHandler(gameEngineTimeout, proxies);
}
```

The recognizers object contains one or more objects that represent different types of recognizers: the patternRecognizer, deviationRecognizer, or progressRecognizer. In addition to these recognizers, there is a predefined timed out recognizer. All of these recognizers are described next.

The events object is where you define the conditions that must be met for your skill to be notified of Echo Button input. You must define at least one event.

If there's an error when you send these directives, Alexa will return a System ExceptionEncountered request.

A very simple example on how the GameEngine.InputHandlerEvent JSON request from Alexa looks like is:

```
{
  "version": "1.0",
  "session": {
    "application": {},
    "user": {},
    "request": {
      "type": "GameEngine.InputHandlerEvent",
      "requestId": "amzn1.echo-api.request.406fbc75-8bf8-4077-a73d-519f53d172a4",
      "timestamp": "2017-08-18T01:29:40.027Z",
      "locale": "en-US",
      "originatingRequestId": "amzn1.echo-api.request.406fbc75-8bf8-4077-a73d-
↪519f53d172d6",
      "events": [
        {
          "name": "myEventName",
          "inputEvents": [
            {
              "gadgetId": "someGadgetId1",
              "timestamp": "2017-08-18T01:32:40.027Z",
              "action": "down",
              "color": "FF0000"
            }
          ]
        }
      ]
    }
  }
}
```

The field `originatingRequestId` provides the requestId of the request to which you responded with a StartInputHandler directive. You need to save this value in your session attributes to send the StopInputHandler directive. You can send this directive with Voxa as follows:

```
const voxa = require('voxa');

app.onIntent('ExitIntent', (voxaEvent) => {
  const { originatingRequestId } = voxaEvent.model;
```

(continues on next page)

(continued from previous page)

```
  return {
    alexaGameEngineStopInputHandler: originatingRequestId,
    tell: 'Buttons.Bye',
  };
});
```

This will stop Echo Button events to be sent to your skill.

10.27 Plugins

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

10.27.1 Using a plugin

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

```
'use strict';
const { VoxaApp, plugins } = require('voxa');
const Model = require('./model');
const views = require('./views'):
const variables = require('./variables');

const app = new VoxaApp({ Model, variables, views });

plugins.replaceIntent(app);
```

10.27.2 State Flow plugin

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

stateFlow(*app*)

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

Arguments

- **app** (`VoxaApp`) – The app object

Usage

```
const { plugins, VoxaApp } = require('voxa');
const voxaApp = new VoxaApp();
plugins.stateFlow(voxaApp);

voxaApp.onBeforeReplySent((voxaEvent) => {
```

(continues on next page)

(continued from previous page)

```
  console.log(voxaEvent.session.outputAttributes.flow.join(' > ')); // entry >␣
↪firstState > secondState > die
});
```

10.27.3 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(*app*[, *config*])

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

Arguments

- **app** (VoxaApp) – The stateMachineSkill
- **config** – An object with the `regex` to look for and the `replace` value.

Usage

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

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

Why OnlyIntents?

A good practice is to isolate an utterance into another intent if it contains a single slot. By creating the OnlyIntent, Alexa will prioritize this intent if the user says only a value from that slot.

Let's explain with the following scenario. You need the user to provide a zipcode. You would have an *intent* called `ZipCodeIntent`. But you still have to manage if the user only says a zipcode without any other words. So that's when we create an OnlyIntent. Let's call it `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`.

10.27.4 CloudWatch plugin

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

Params

cloudwatch (*app*, *cloudwatch*[, *eventMetric*])
CloudWatch plugin uses *VoxaApp.onError()* and *VoxaApp.onBeforeReplySent()* to log metrics

Arguments

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

Usage

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

const cloudWatch = new AWS.CloudWatch({});
const eventMetric = {
  MetricName: 'Caught Error', // Name of your metric
  Namespace: 'SkillName' // Name of your skill
};

Voxa.plugins.cloudwatch(app, cloudWatch, eventMetric);
```

10.27.5 Autoload plugin

It accepts an adapter to autoload info into the model object coming in every alexa request.

Params

autoLoad (*app*[, *config*])
Autoload plugin uses `app.onSessionStarted` to load data the first time the user opens a skill

Arguments

- **app** (VoxaApp) – The stateMachineSkill.
- **config** – An object with an `adapter` key with a *get* Promise method in which you can handle your database access to fetch information from any resource.

Usage

```
const app = new VoxaApp({ Model, variables, views });

plugins.autoLoad(app, { adapter });
```

10.27.6 S3Persistence plugin

It stores the user's session attributes in a file in an S3 bucket. You can use this plugin when you host your Node.js code with the Alexa-Hosted skills feature. For more details about how this work, check the official documentation.

If you host your code in your own AWS account and plan to use S3 as an storage alternative, keep in mind that you cannot do any Scan or Query operations from S3 and the time to storage and get info is a little longer than DynamoDB.

Params

s3Persistence (*app*[, *config*])

S3Persistence plugin uses `app.onRequestStarted` to load data every time the user sends a request to the skill S3Persistence plugin uses `app.onBeforeReplySent` to store the user's session data before sending a response back to the skill

Arguments

- **app** (VoxaApp) – The stateMachineSkill.
- **config** – An object with a `bucketName` key for the S3 bucket to store the info. A `pathPrefix` key in case you want to store this info in a folder. An `aws` key if you want to initialize the S3 object with specific values, and an `s3Client` key, in case you want to provide an S3 object already initialized.

Usage

```
const app = new VoxaApp({ Model, variables, views });

const s3PersistenceConfig = {
  bucketName: 'MY_S3_BUCKET',
  pathPrefix: 'userSessions',
};

plugins.s3Persistence(app, s3PersistenceConfig);
```

10.28 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
```

You can also get more per platform debugging information with

```
DEBUG=voxa:alexa
```

```
DEBUG=voxa:botframework
```

```
DEBUG=voxa:dialogflow
```

Index