Skill Testing Markup Language

Draft Design and Specification

Prepared by:

Bespoken

Version	*Author*	*Date*	*Comments*
1.0	John Kelvie	4/10/2018	Initial Draft

Overview

This document describe a complete YAML syntax for testing Alexa skills. It is based in large part upon Kay Lerch's Test Automation project and Bespoken's Virtual Device Script - additional references are below. This scripting format allows for comprehensive testing of Alexa skills, across features, development stages and audiences (dev, QA, ops, etc.).

The goal is to make it easy for Alexa skill builders to build and test their skills.

Objectives

Key objectives for the syntax:

- It should be easy to read and understand, for programmers and non-programmers alike
- It should rely on existing standards as much as possible, such as YAML and JSONPath
- It should be independent of any particular implementation
- It should scale up or down in terms of development stage and depth of the test desired

The last point requires a bit of explanation - it means the syntax should allow for:

- Local-only testing (no infrastructure required)
- Testing via Invocation API (requires Lambda deployment - no AVS interaction)
- Testing via Simulation API (requires Lambda deployment - textual AVS interaction)
- Testing via Bespoke Virtual Devices (which interact via audio and the AVS endpoints)

This is important because, for example, local-only testing is extremely useful for creating skills using a test-driven development (TDD) approach, where no code has been deployed and the skill has not been configured yet. At the same time, comprehensive testing using the Simulation API and Bespoken's Virtual Devices allows for deeper end-to-end and regression testing.

YAML is used as the syntax because it is simultaneously readable and accessible to novices while being expressive and powerful for experts.

References

Alexa Skills Kit Test Java - From Kay Lerch, Amazon
Automating Alexa Skill Dialog Testing - From Macadamian
BotTalk - YAML Syntax for building Alexa skills
JSONPath - Syntax used for interacting with JSON objects
Skill Sample Nodejs Automation - From Alexa Team
Virtual Device Script - From Bespoken
YAML Syntax

Syntax Description

Conversations are constructed as YAML elements. A basic example, built against the Node.js Fact Sample skill is below:

```yaml
# A simple example of skill test suite
--- # Configuration YAML document
configuration:
  locale: en-US

--- # The --- indicates the start of a new test, which is a self-contained YAML document
- test: "Launches successfully" # Optional info about the test
- LaunchRequest: # LaunchRequest is "reserved" - it is not an utterance but a request type
  - response.outputSpeech.ssml == "Here's your fact: *" # == means an exact match
  - response.reprompt == undefined
  - response.card.content =~ /.*/ # =~ means a regular expression match

---
- test: "Gets a new fact intent"
- "Get New Facts":
  - response.outputSpeech.ssml == "Here's your fact: *" # Wildcard matches anything
  - response.card.title == "Space Facts" # == means exact match
  - response.card.content != undefined # Checks that content is specified

---
- test: "Gets help with multi-turn interaction"
- "Help":
  - response.outputSpeech.ssml == "You can say tell me a space fact, or, you can say exit...*"
  - response.outputSpeech.ssml == “*What can I help you with?" # Multiple tests for one field
  - response.reprompt.outputSpeech.ssml =~ /.*can I help you with.*/i
- "Get a fact":
  - response.outputSpeech.ssml == "Here's your fact: *"

---
- "Cancel":
  - response.outputSpeech.ssml == "Goodbye!"
  - response.shouldEndSession == true

---
- "Stop":
  - response.outputSpeech.ssml == "Goodbye!"
```

Skill Testing File Structure

Skill tests are contained within a YAML file, which is referred to as a Skill test suite. Each suite can contain one or many tests (each a YAML document, denoted by “---”), as well as an optional “configuration” YAML document at the top.

Tests are delimited with the “---” which denotes a new YAML document. A tests essentially represents an Alexa session, though in special cases, such as interactions with the AudioPlayer or VideoApp, it can span the standard Alexa session. The mechanisms for continuing to interact with the skill being tested after the session are closed are left to the particular testing implementation.

Each test can contain one-or-many interactions with Alexa. The test can be prefaced with a “test” element which contains information about the test itself.

Each element in the collection after the “test” represents an interaction. With each interaction, any test setup is done and assertions are defined. An interaction typically corresponds to a user saying something to Alexa, but it can also represent non-speech interactions, such as a button press or receiving an event from the Event API.

The interactions are structured as named collections. The name is either an utterance (what is being said to Alexa) or a request type (such as LaunchRequest, SessionEndedRequest, etc.).

The collection contains a combination of the following:

- Setting up of intents and slots [Optional - for use when the utterance can not be interpreted correctly by the test executor]
- Setting up of request object - to exactly represent particular request states
- Assertions on the response object as well as control flow based on the response

Built-in Objects

Test Built-in Objects

Tests have the following built-in objects:

- test - meta information about the test itself
    - description - description of the test
    - timeout - the maximum duration of the test

Interaction Built-in Objects/Statements

Each interaction supports the following built-on objects:

- intent - the name of the intent to be invoked - the utterance is used if not defined.
- slots - the slots, defined as key-value pairs, to be passed in the invocation.
- request - the request object itself - allows for arbitrary objects and properties to be set on the request. See the Gadget API example below.
- exit - Indicates the test is complete and to stop processing - part of flow control described here.

Any object **NOT** in the list above is assumed to be part of the JSON response, and will be interpreted an assertion/control flow statement. This future proofs against new elements being added to JSON response structure, as well as allows for easily writing complex JSONPath expressions against the response structure (for example, ones that start with * or .. to avoid writing out long paths in their entirety - see the AudioPlayer or DisplayInterface sections for examples).

Meta Information

Meta information about the test suite as a whole is specified at the top - it can include data about the locale in which the test is meant to run, the underlying API it should use (e.g., such as Invocation versus Simulation), or any other data useful for executing the test.

Test Execution

Tests within the test suite file are executed in sequence.

Within a particular test, the interactions are executed in the order they appear in the collection.

Within a particular interaction, assertions will be executed in the order they appear. The test executor will stop processing if any assertion fails. It will also stop processing if it encounters the statement “stop” as part of an interaction. When it stops processing a test, it will still go on and process the next test, if there is one.

There is a special case with assertions - if they end with “goto <TARGET>”, they are treated as a goto statement. In that case, if the expression part of the assertion matches, the test will stop processing and jump to the named interaction in the test - more details here. An assertion that contains a goto statement does NOT cause the test to fail if it is not matched.

Assertions

Assertions are defined using a combination of JSONPath and a simple expression syntax.

The expression syntax follows the format:
<JSON_PATH_EXPRESSION> <OPERATOR> <VALUE> [“goto” <TARGET>]

Supported Operators

The list of supported operators are based upon the JSONPath operators described here:

Expression	***Definition***
==	*Exact match (excluding opening and closing tags for SSML)*
!=	*Does not match*
=~	*Regular expression match*
>	*Greater than - must be a numeric value*
>=	*Greater than or equal to*
<	*Less than*
<=	*Less than or equal to*

Supported Value Syntax

The value syntax can be any of the following supported types:

- string - A text value - can include the wildcard *, which will match anything
- regex - A regular literal, as described here
- boolean - **true** or **false**
- numeric - An integer or decimal value
- null - The JSON value of null
- undefined - The property does not exist

Flow Control

An example of a test that uses flow control is show here:

```
---
- Test: "Guesses a number - exits when guessed correctly"
- LaunchRequest:
  - response.outputSpeech.ssml == "Guess a number between 1 and 3"
  - response.shouldEndSession == false
- "2":
  - response.outputSpeech.ssml == "Too high. Guess again" goto “1”
  - response.outputSpeech.ssml == "Too low. Guess again" goto “3”
  - response.outputSpeech.ssml == “You got it - thanks for playing!”
  - response.outputSpeech.shouldEndSession == true
  - exit
- "1":
  - response.outputSpeech.ssml == "You got it - thanks for playing!"
  - response.outputSpeech.shouldEndSession == true
  - exit
- "3":
  - response.outputSpeech.ssml == "You got it - thanks for playing!"
  - response.outputSpeech.shouldEndSession == true
```

In this example, the test executes until the number is matched. It leverages the explicit “exit” command to force the end of test execution when the number is guessed correctly.

It uses the “goto” statement within the assertions to direct the test to correct next step when the number is not guessed correctly.

Advanced Example

Here is a more advanced example, that shows testing the Pet Match sample skill. It

```
---
configuration:
  locale: en-US

---
- Test: "Launch and elicit slots"
- LaunchRequest:
  - response.outputSpeech.ssml == "Welcome to pet match. I can help you find the best dog for
you"
  - response.shouldEndSession == false
- "I want a tiny dog":
  - intent: PetMatchIntent # Intent name is optional - for cases when utterance cannot be easily
resolved, or to be explicitly
  - slots:
     size: "tiny"
  - response.outputSpeech.ssml == "Are you looking for more of a family dog or a guard dog?"
  - response.outputSpeech.ssml == "Would you prefer a dog to hang out with kids or to protect
you?"
- "family dog":
  - intent: PetMatchIntent
  - slots:
      temperament: "family"
  - response.outputSpeech.ssml == "Do you prefer high energy or low energy dogs?"
- "high energy":
  - response.outputSpeech.ssml == "So a tiny family high energy dog sounds good for you. Consider
a Chihuahua"

---
- Test: "Launch and one-shot dialog"
- LaunchRequest:
  - response.outputSpeech.ssml == "Welcome to pet match. I can help you find the best dog for
you"
  - response.shouldEndSession == false
- "I want a tiny family high energy dog":
  - response.outputSpeech.ssml == "So a tiny family high energy dog sounds good for you. Consider
a Chihuahua"
```

This example demonstrates setting explicit slots and intents, as well as an extended interaction with the skill.

Feature-Specific Examples

The following section covers how to handle various Alexa features using the proposed syntax.

AudioPlayer Interface

An example of testing the AudioPlayer interface is contained below:

```
---
- “test”: “AudioPlayer interaction works correctly”
- LaunchRequest: # Tests the contents of the AudioPlayer.Play directive
  - response.outputSpeech.ssml == “Here is some great music for you”
  - response.shouldEndSession == true
  - response.directives[0].type == “AudioPlayer.Play”
  - ..audioItem.stream.url == “https://streaming.com/GreatSong.mp3”
  - ..audioItem.stream.token == “StreamToken”
- AMAZON.NextIntent
  - ..audioItem.stream.url == “https://streaming.com/AnotherGreatSong.mp3”
  - ..audioItem.stream.token == “StreamToken2”
```

The interaction actually spans the Alexa session - the session ends after the LaunchRequest, though the tests continues, sending a NextIntent that will be received by the skill.

Dialog Management

From a testing perspective, dialog interactions are scripted in the same way as any other Alexa interaction. For handling this within an emulator, there is a difference underneath the covers, as it requires mimicking the behavior of the Alexa dialog manager. Addressing how this will be done is outside the scope of this document.

Display Interface

The display interface can be tested via use of the regular syntax as well as the ***Display.ElementSelected*** request type. Here is an example:

```
---
- “test”: “Echo show interaction works correctly”
- LaunchRequest: # Tests the contents of the Display.RenderTemplate directive
  - ..listItems[0].textContent.primaryText.text == “Choice Number One”
  - ..listItems[0].textContent.primaryText.token == “Item1”
  - ..listItems[1].textContent.primaryText.text == “Choice Number Two”
  - ..listItems[1].textContent.primaryText.token == “Item2”
  - ..backgroundImage.sources[0].url == “https://images.com/my-image.png”
- Display.ElementSelected:
  - request.token: “Item2”
  - response.outputSpeech.ssml == “Great choice!”
```

The more advanced syntax of JSONPath is leveraged to structure assertions on the Display directive.

Entity Resolution

See above with regard to Dialog Management - for a tester, the syntax is the same as a normal invocation.

Gadget API Support

Gadgets are supported with the request type ***GameEngine.InputHandlerEvent.*** An example test that sents an InputHandlerEvent:

```
---
- LaunchRequest
- GameEngine.InputHandlerEvent:
  - request.request.events[0].name: “ButtonPress”
  - request.request.events[0].inputEvents[0].action: “down”
  - request.request.events[0].inputEvents[0].color: “FF0000”
  - request.request.events[0].inputEvents[0].gadgetId: “GadgetID1”
  - request.request.events[0].inputEvents[0].timestamp: “2017-08-18T01:32:40.027Z”
  - response.outputSpeech.ssml == “You pressed a button!”
```

We are simply leveraging the ability to set arbitrary fields on the request via JSON path.

Progressive Responses

Testing of progressive responses is outside the scope of this document.

VideoApp Interface

Testing of the VideoApp is simlar to the AudioPlayer - example:

```
---
- “test”: “VideoApp interaction works correctly”
- LaunchRequest: # Tests the contents of the VideoApp.Launch directive
  - response.outputSpeech.ssml == “Here is a video you will enjoy!”
  - response.shouldEndSession == true
  - response.directives[0].type == “VideoApp.Launch”
  - ..videoItem.source == “https://streaming.com/CoolVideo.mp3”
  - ..videoItem.metadata.title == “A cool video”
```