Botium

Botium

May 03, 2021

TABLE OF CONTENTS:

1. **Selenium** is the de-facto-standard for testing web applications.
2. **Appium** is the de-facto-standard for testing smartphone applications.
3. **Botium** is for testing conversational AI.

Just as Selenium and Appium, Botium is free and Open Source, and available on Github.

CHAPTER

ONE

QUICKSTART

If you cannot wait to start, get a quick glimpse of Botium by starting your command line of choice and start typing (**Node.js installation is required**).

1.1 ... with Mocha

The following commands will install Botium Bindings, extend your Mocha specs with the Botium test case runner and run a sample Botium test:

```
npm init -y
npm install --save-dev mocha botium-bindings
npx botium-bindings init mocha
npm install && npm run mocha
```

What's happening here:

- A fresh Node.js project is created with mocha and botium-bindings
- The *package.json* file is extended with a "botium"-Section and some devDependencies
- A *botium.json* file is created in the root directory of your project
- A *botium.spec.js* file is created in the *spec* folder to dynamically create test cases out of your Botium scripts
- A sample convo file is created in the *spec/convo* folder
- A first test run is started

1.2 ... with Botium CLI

```
npm install -g botium-cli
botium-cli init
botium-cli run
```

What's happening here:

- The first command installs the Botium CLI on your workstation
- The second command initializes a Botium project in the current directory (a *botium.json* file and a *convo file*)
- Finally, the Botium project runs and the test report is shown

CHAPTER

TWO

INTRODUCTION

2.1 What is Botium

1. **Selenium** is the de-facto-standard for testing web applications.
2. **Appium** is the de-facto-standard for testing smartphone applications.
3. **Botium** is for testing conversational AI.

Just as Selenium and Appium, Botium is free and Open Source, and available on Github.

2.2 What is Botium good for ?

Botium supports chatbot makers in training and quality assurance:

- **Chatbot makers** define what the chatbot is supposed to do
- **Botium** ensures that the chatbot does what it is supposed to do

Here is the "Hello, World!" of Botium:

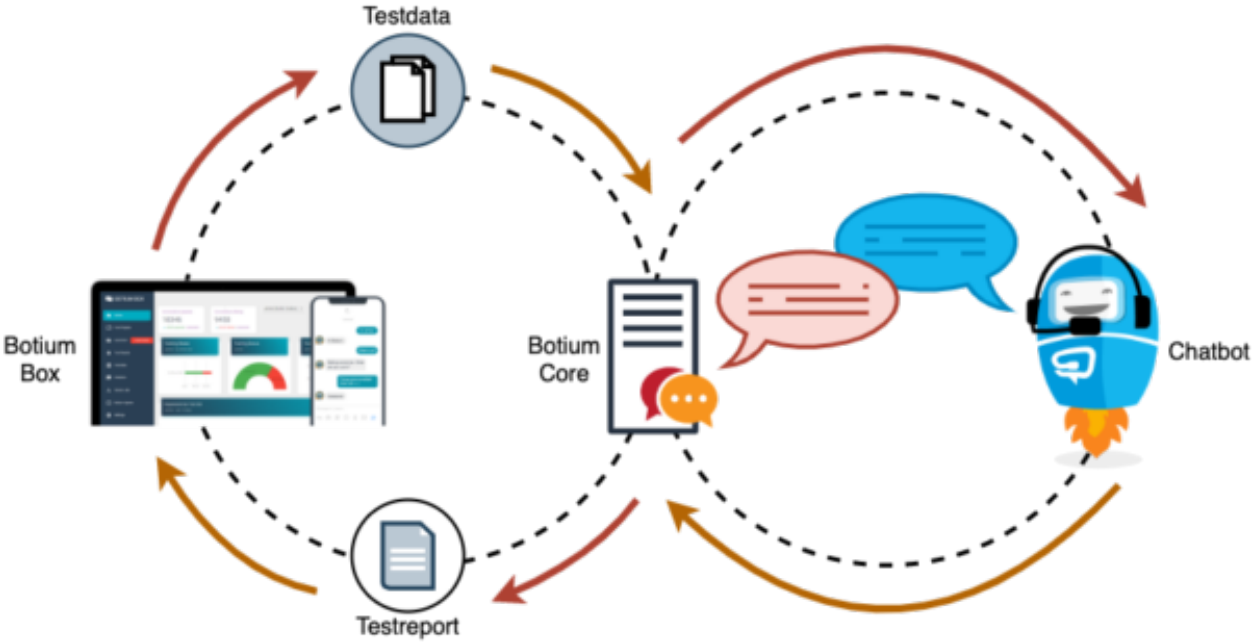
```
TC01_HELLO

#me
hello bot!

#bot
Hello, meat bag! How can I help you ?
```

The test case is named *TC01_HELLO*, and the chatbot is supposed to respond to a user greeting.

To name just a few features of Botium:

- Testing conversation flow of a chatbot
  - Capture and Replay
  - Integrated speech processing for testing voice apps
- Testing NLP model of a chatbot
  - Domain specific and generic datasets included
  - Paraphrasing to enhance test coverage
- E2E testing of a chatbot based on Selenium and Appium
- Non-functional testing of a chatbot

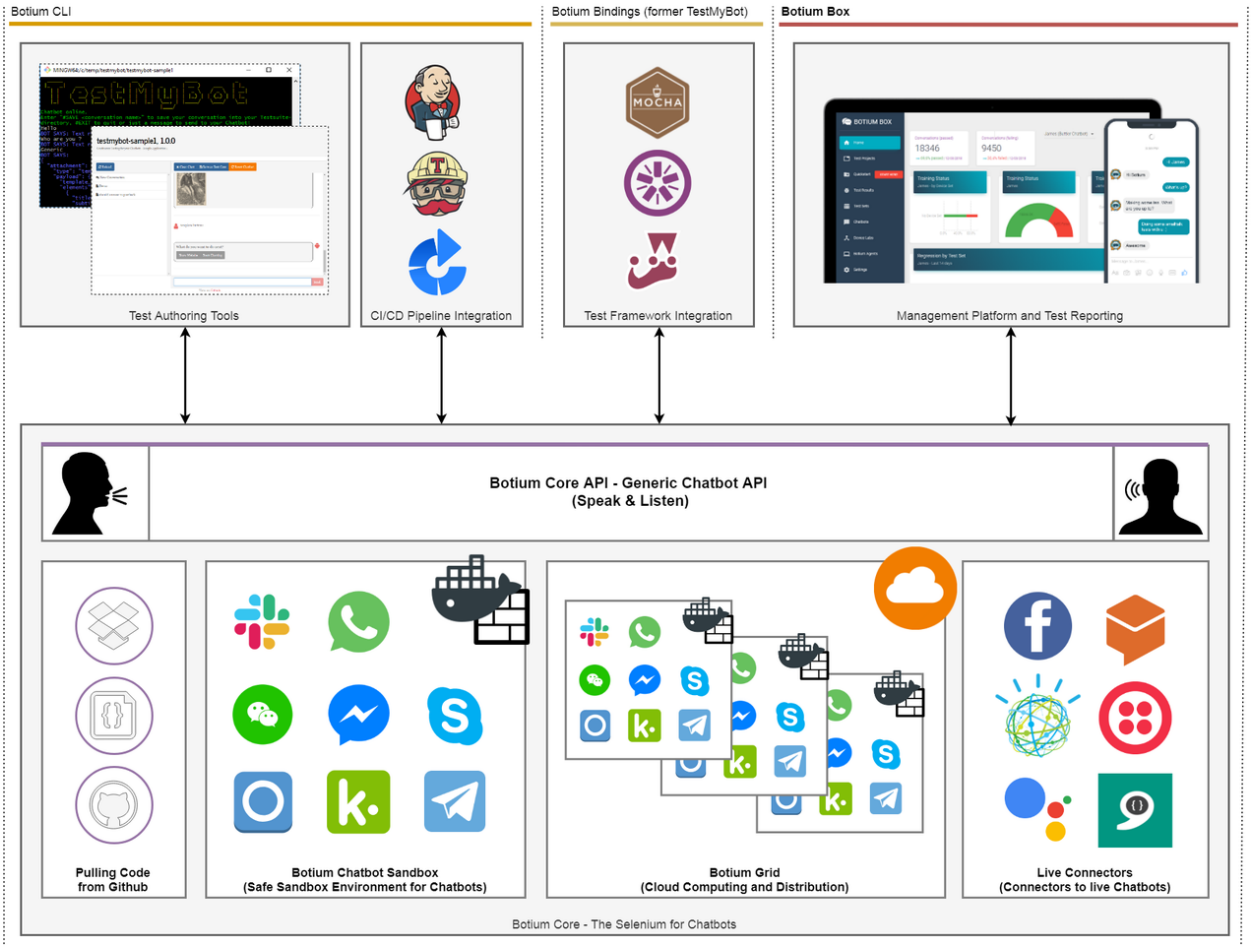
  - Load- and Stress testing
  - Security testing
  - GDPR testing
- CI/CD integration with all common products in that space (Jenkins, Bamboo, Azure DevOps Pipelines, IBM Toolchain, . . .)
- and many more



2.3 Understanding the Botium Stack

When we talk about Botium, we usually mean the whole Botium Stack of components. It is built on several components:

- Botium Core SDK to automate conversations with a chatbot or virtual assistant
- *Botium CLI*, the swiss army knife to use all functionality of Botium Core in the command line
- *Botium Bindings*, the glue to use Botium Core with test runners like Mocha, Jasmine or Jest
- *Botium Crawler*, like a web crawler, just for chatbots
- Botium Box, the management and reporting platform for making chatbot test automation fast and easy - Get your free instance here
- Botium Coach for continuous visualization of NLP performance metrics - See Botium Coach Wiki



2.3.1 Botium Core, the heart and brain of Botium

All of the components in the Botium Stack build on top of Botium Core (except Botium Core itself, naturally), the heart and brain of Botium. Just as the heart and brain in your body are vital parts with very low chances to ever see or touch them, you most likely won't ever get in touch with Botium Core directly: it's the fuel, the nuts and bolts, the core technology, the heart and brain of Botium.

2.3.2 Botium CLI, the swiss army knife of Botium

The Botium CLI is a command line tool to actually use everything Botium Core is capable to do. If Botium Core is the heart and brain of Botium, then the Botium CLI stands for the extremities, the hands and feets of Botium (this was the last anatomic analogy in this article). It is a command line tool, which means it doesn't provide a graphical user interface with buttons, pictures and hyperlinks. While graphical user interfaces are nice for first time usage, they are a big impediment in process and test automation. That's why we built the Botium CLI.

2.3.3 Botium Bindings, the glue to bind Botium to test runners

A "test runner" is a piece of software which automatically runs thousands of test cases and outputs a nicely formatted summary about successful and failed test cases at the end. There are several test runners available you can choose from (Mocha, Jasmine, Jest, ...) and Botium Bindings make them run the Botium test cases.

2.3.4 Botium Crawler, like a website crawler

The Botium Crawler is doing the work of detecting the conversation flows supported by your chatbot by itself. It does so by analyzing the quick responses offered by your chatbot and simulating clicks on all of the options in parallel, following all pathes down until it reaches the end of the conversation.

All detected conversation flows along all pathes are saved as Botium test cases and utterance lists and can be used as base for a regression test set.

2.3.5 Botium Box, the management and reporting platform of Botium

Botium Box is the pretty face of Botium : A modern, responsive, easy-as-hell web-based graphical user interface to configure, control and monitor every aspect of Botium Core.

2.3.6 Botium Platform: Everything you need to run Botium in the Enterprise

Botium Platform extends the open source Botium Stack libraries with enterprise features (relational database support, multiple deployment options, monitoring, build pipeline integration, load balancing and more) and includes enterprise-grade support. While the Botium Stack libraries are open source and free ("free" as in "freedom" as well as in "free beer"), the Botium Platform is an SaaS offer.

2.4 Installation

Here you can find installation instructions:

- *Botium CLI* - available as Node.js module and Docker image
- *Botium Bindings* - available as Node.js module
- *Botium Crawler* - available as Node.js module
- Botium Box - available on-promise and as SaaS - Get your free instance here

2.5 How do I get help ?

- Read the Botium in a Nutshell series
- If you think you found a bug in Botium, please use the Github issue tracker
- Consult the Botium Wiki
- For asking questions please use Stackoverflow-we are monitoring and answering questions there
- Enter our Discord channel

2.5.1 Enterprise Support

For enterprise agreements, please contact us on our website.

CHAPTER

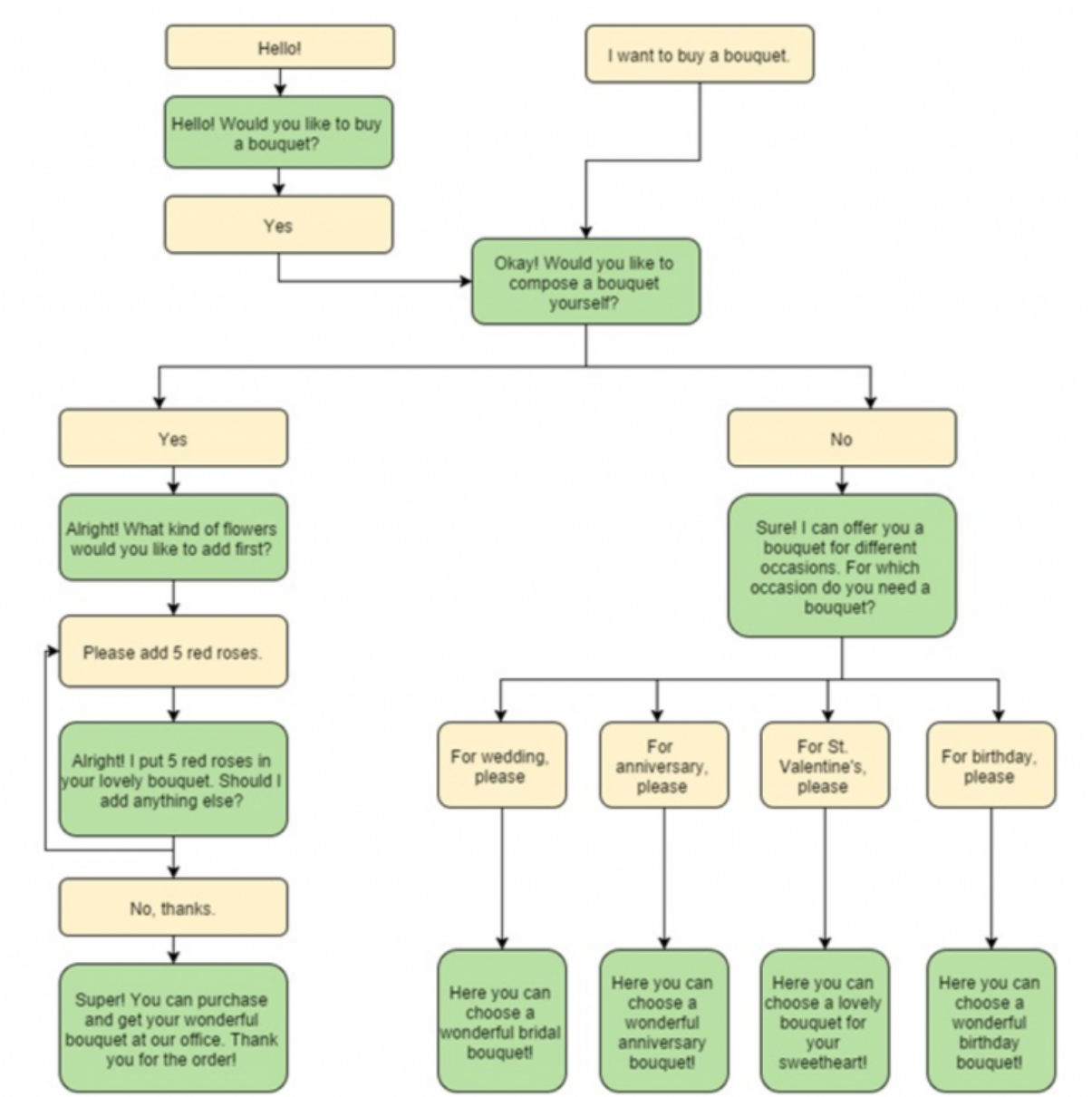
THREE

BOTIUM TUTORIAL

Parts of this guide have been published in the book ACCELERATING SOFTWARE QUALITY - Machine Learning & Artificial Intelligence in the Age of DevOps by Eran Kinsbruner.

3.1 Testing Conversational Flow

This section starts with some technical background on Botium and then demonstrates a methodology to identify and formalize test cases for the conversational flow of a chatbot. The conversational flow, often called “user stories”, can be visualized in a flow chart.



3.1.1 Hello, World! The Botium Basics

The most basic test case in Botium consists of

- submitting a phrase possibly entered by a real user to the chatbot
- checking the response of the chatbot with the expected outcome

BotiumScript

In Botium, the test cases are described by conversational flows the chatbot is supposed to follow. For a sample "greeting" scenario, the Botium test case looks like this — also known as "BotiumScript":

```
#me
hello bot!

#bot
Hello, meat bag! How can I help you ?
```

You can write BotiumScript in several file formats

- plain text file with Notepad or any other text editor
- Excel file
- CSV file (comma separated values)
- JSON
- YAML

Convos and Utterances

So, let's elaborate the "Hello, World!"-example from above. While some users will say "hello", others maybe prefer "hi":

```
#me
hi bot!

#bot
Hello, meat bag! How can I help you ?
```

Another user may enter the conversation with "hey dude!":

```
#me
hey dude

#bot
Hello, meat bag! How can I help you ?
```

And there are plenty of other phrases we can think of. For this most simple use case, there are now at least three or more BotiumScripts to write. So let's rewrite it. We name this file hello.convo.txt:

```
TC01 - Greeting

#me
HELLO_UTT

#bot
Hello, meat bag! How can I help you ?
```

You may have noticed the additional lines at the beginning of the BotiumScript. The first line contains a reference name for the test case to make it easier for you to locate the failing conversation within your test case library. And we add another file hello_utt.utterances.txt:

```
HELLO_UTT
hello bot!
hi bot!
hey dude
good evening
hey are you here
anyone at home ?
```

- **The first BotiumScript is a convo file — it holds the structure** of the conversation you expect the chatbot to follow.
- **The second BotiumScript is an utterances file — it holds several** phrases for greeting someone, and you expect your chatbot to be able to recognize every single one of them as a nice greeting from the user.

Botium will take care that the convo and utterances files are combined to verify every response of your chatbot to every greeting phrase. So now let's assume that your chatbot uses several phrases for greeting the user back. In the morning it is:

```
#me
HELLO_UTT

#bot
Good morning, meat bag! How can I help you this early ?
```

And in the evening it is:

```
#me
HELLO_UTT

#bot
Good evening, meat bag! How can I help you at this late hour ?
```

Let's extract the bot responses to another utterances file:

```
BOT_GREETING_UTT
Good evening
Good morning
Hello
Hi
```

And now comes the magic, we change the convo file to:

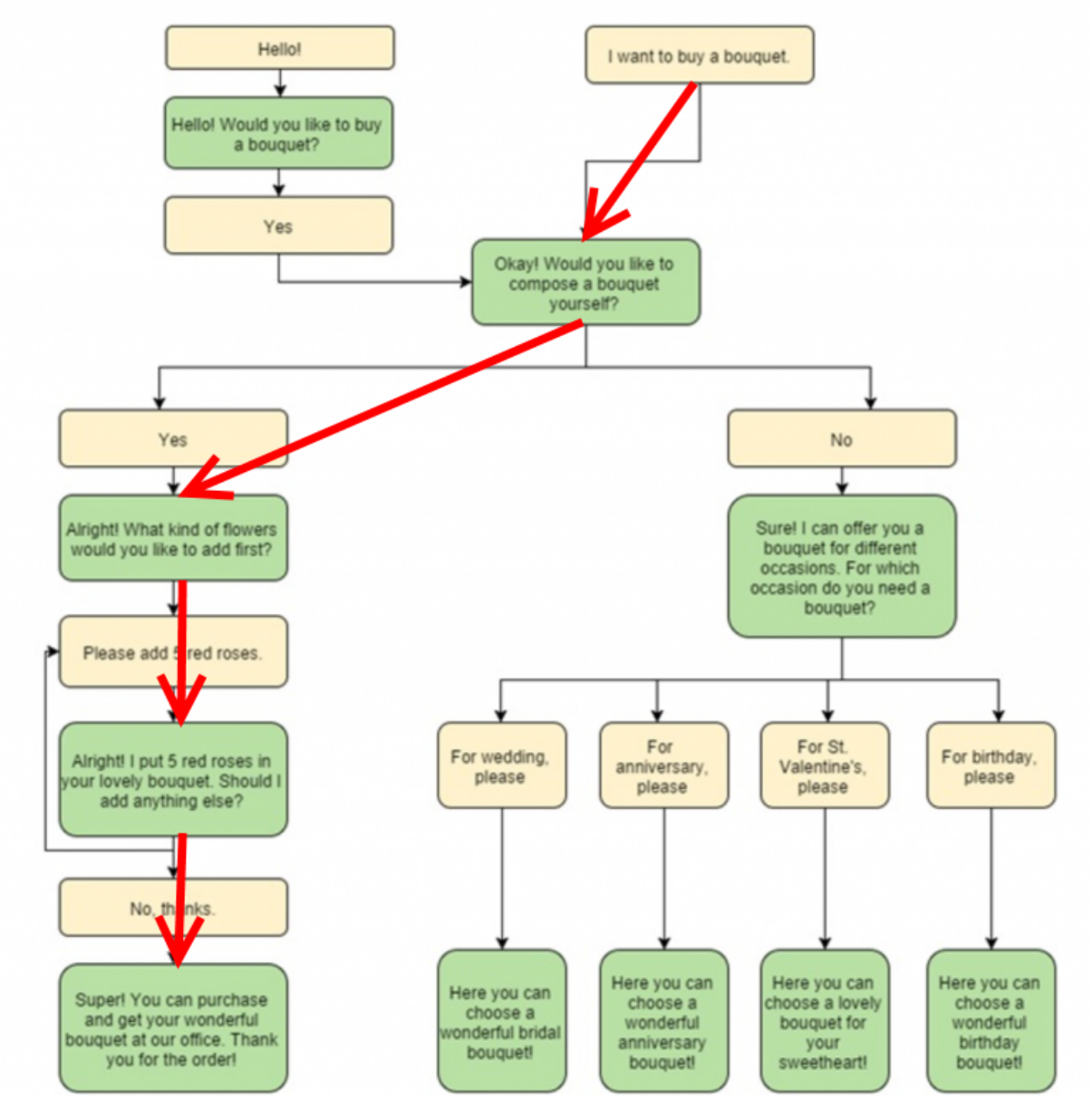
```
#me
HELLO_UTT

#bot
BOT_GREETING_UTT
```

Utterances files can be used to verify chatbot responses as well. To summarize:

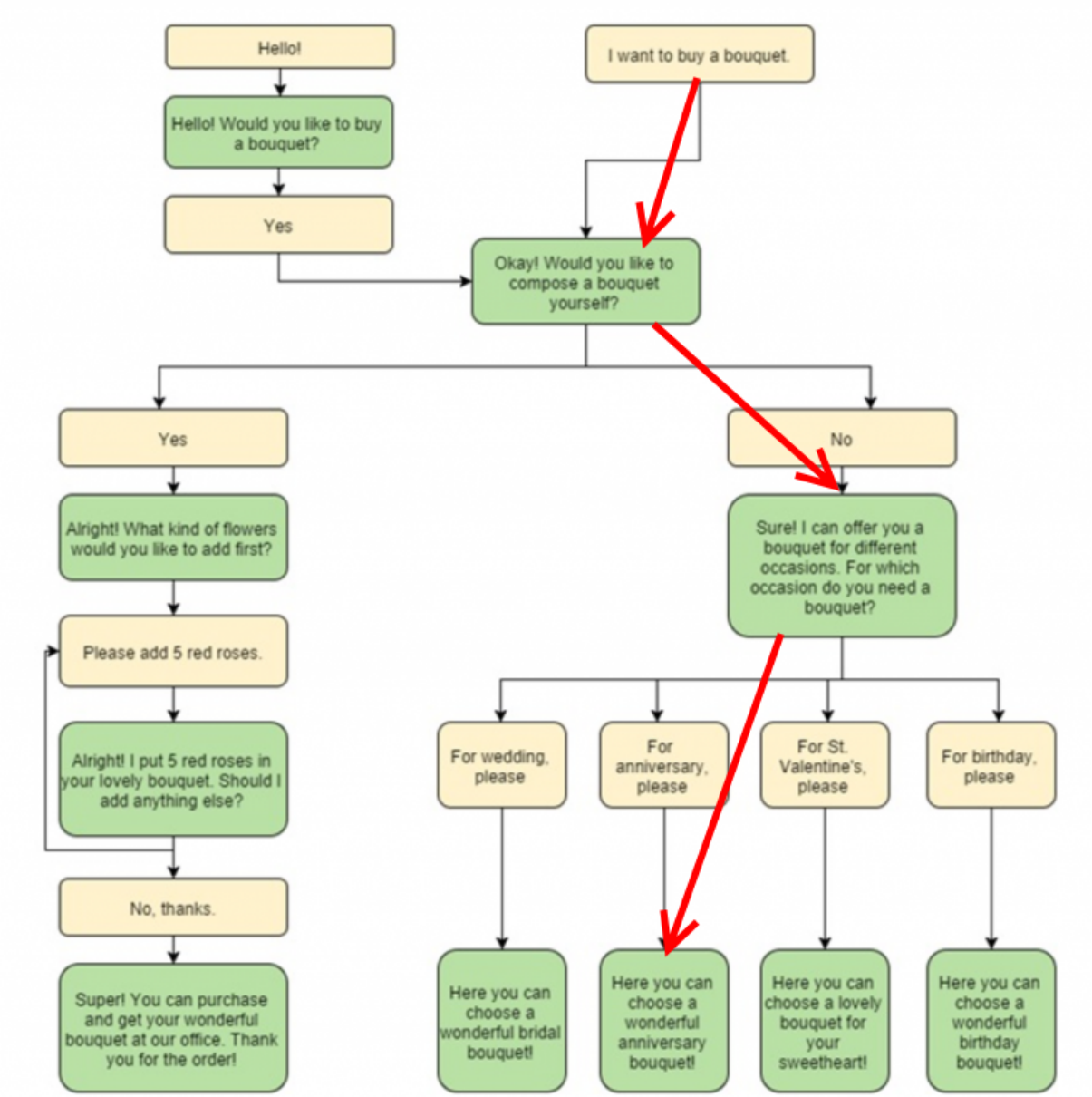
- **An utterance referenced in a #me-section means: Botium, send every** single phrase to the chatbot and check the response
- **An utterance referenced in a #bot-section means: Botium, my chatbot** may use any of these answers, all of them are fine

3.1.2 Identification of Test Cases

If the flow chart is available, identification of the test cases is actually straightforward: Each path through the flow chart from top to bottom is a test case. Here is the path for the user story "User composes customized bouquet".



And here is the path for "User selects anniversary bouquet".



3.1.3 Scripting Test Cases for a conversational flow

In BotiumScript, the conversational flow for user story "User composes customized bouquet" can be expressed like this:

```
#me
I want to buy a bouquet

#bot
OK, do you want to compose a bouquet yourself ?

#me
Yes
```

(continues on next page)

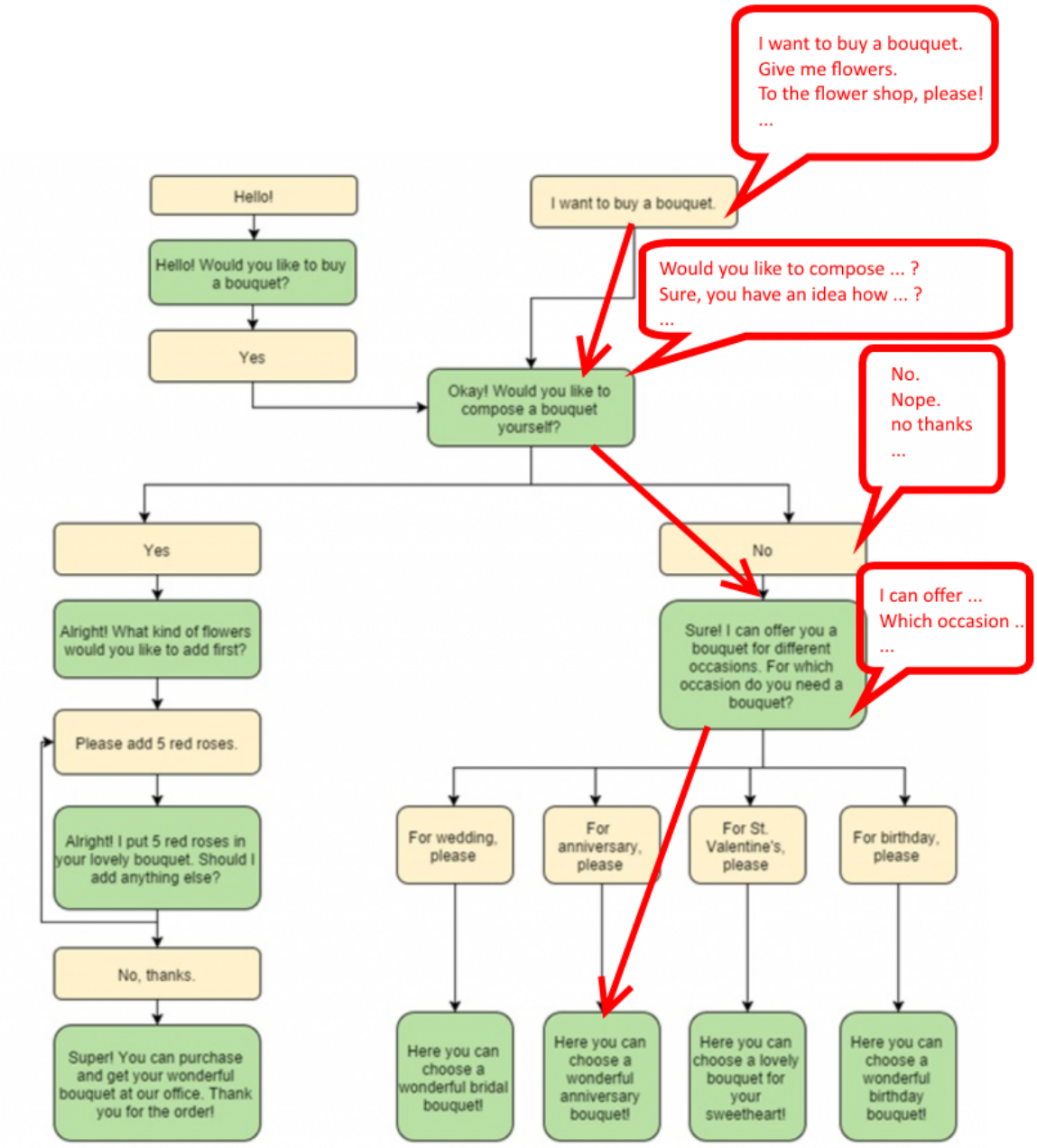
(continued from previous page)

```
#bot
OK, what kind of flowers would you like to add first ?

#me
Please add 5 red roses

#bot
Alright, I put 5 red roses in your lovely bouquet. Should I add anything else ?

#me
No, thanks.

#bot
Super!
```

As soon as a chatbot doesn't respond as expected, the test case is considered as failed and reported.

3.1.4 Scripting Utterance Lists

What the flow charts don't show are the endless possibilities for a user to express an intent. For each node in the flow chart, there are various input and output utterances to consider. The flow chart typically pictures a "happy path" in the conversation, in a real-world scenario the same conversation path and test case should be satisfied with most usual utterances and utterance combinations.



For the "I want to buy a bouquet", there are plenty of other ways for a user to express this intent:

- "Give me some flowers"
- "To the flower shop, please"
- "purchase a bouquet"
- ...

All of these user examples are valid input for the same test case, and in Botium these user examples are collected within an utterance list in a text file:

```
UTT_USER_ORDER_FLOWERS
I want to buy a bouquet
```

(continues on next page)

(continued from previous page)

```
Give me some flowers
To the flower shop, please purchase a bouquet
```

What the flow charts don't show as well are the utterances used on the other side, by the chatbot itself: a well-designed chatbot provides some variance in conversation responses.

For example instead of "Okay! Would you like to compose a bouquet yourself" the chatbot might as well respond with:

- "Do you want me to suggest a composition?"
- "Is it for a special occasion" ?
- ...

These utterances can be collected in another utterance list and used in the test cases to allow the chatbot all responses matching one in this list. The first part of the user story "User composes customized bouquet" would then look like this:

```
#me
UTT_USER_ORDER_FLOWERS

#bot
UTT_BOT_COMPOSE_YN
```

The conversational flow remains the same, but there are many user examples and chatbot responses allowed now.

3.1.5 Dealing with Uncertainty

When using Botium, there are many options for asserting the chatbots behaviour - the most simple one, assertion of the text response, is shown above.

- **Asserting the presence of user interface elements, such as quick** response buttons, media attachments, form input elements
- Asserting with regular expressions and utterance lists
- Asserting tone with a tone analyzer
  - **Validation that the chatbot tone matches the intended brand** communication style
- Asserting availability of hyperlinks presented to the user
- Asserting custom message payload with JSONPath queries
- Asserting business logic with API and data storage queries

3.1.6 Generating a Test Report

There are several frontends available for generating a test report with Botium.

Option 1: Botium CLI

Run Botium CLI like this:

```
botium-cli run
```

Botium CLI will build up a communication channel with your chatbot and run all of your test cases. Status information and a summary are displayed in the command line window.

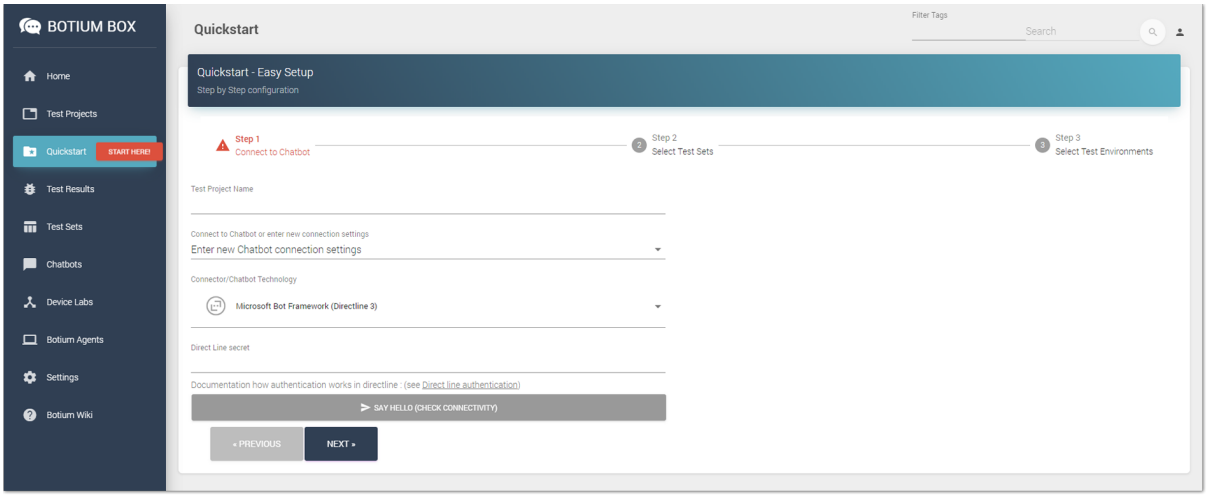
Option 2: Botium Bindings

With Botium Bindings an established test runner like Mocha, Jest or Jasmine can be used for running Botium test cases.:

```
mocha ./spec
```

Option 3: Botium Box

Use the Quickstart Wizard to connect your chatbot to your test sets and run them.



3.2 E2E Testing with Selenium or Appium

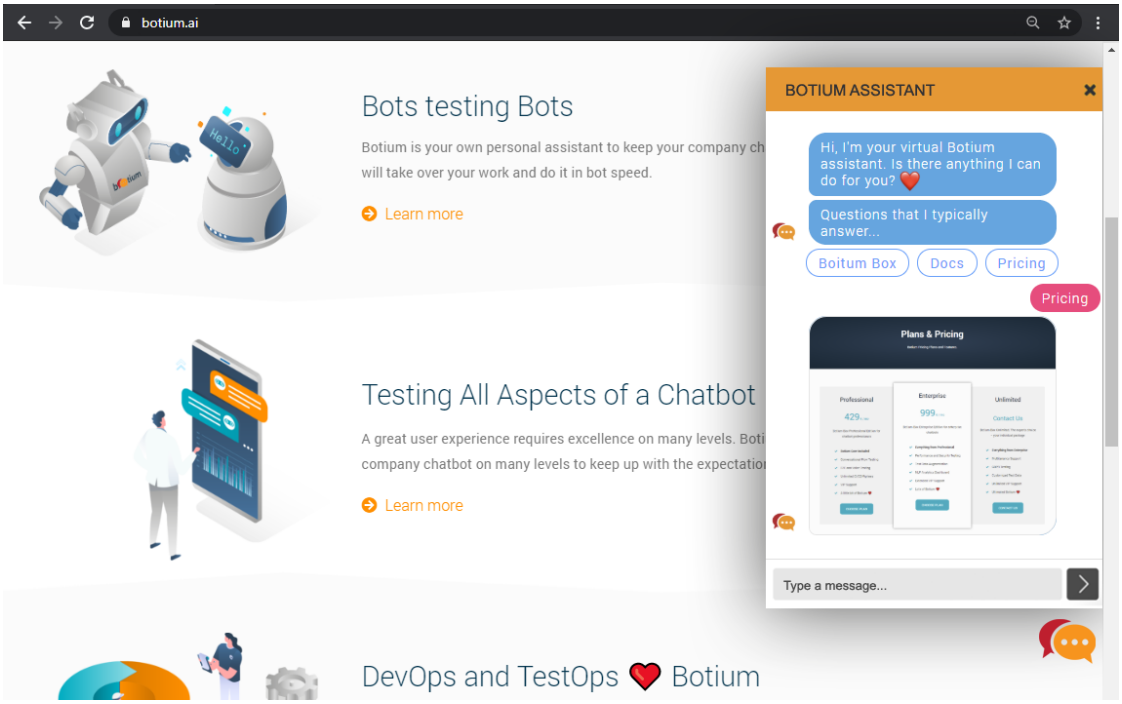
Testing the user experience end-to-end has to be part of every test strategy. Apart from the conversation flow, which is best tested on API level, it has to be verified that a chatbot published on a company website works on most used end user devices.

The special challenges when doing E2E tests for a chatbot are the high amount of test data needed (> 100.000 utterances for a medium sizes chatbot) and the slow execution time - in an E2E scenario tests are running in real time. The good news are that for testing device compatibility, a small subset of test cases is sufficient.

3.2.1 Safe Assumptions when testing a chatbot with Selenium

When testing a chatbot with Selenium, there are some safe assumptions you can rely on to reduce effort when coding test cases:

1. **The chatbot is accessible on a website and there maybe is some kind** of click-through to actually open the chatbot window. The procedure to navigate and open the chatbot window is always the same for all test cases.
2. **Somewhere in the chatbot window there is an input field for text** messages. When hitting "Enter" or clicking on a button besides the input field the text will be sent to the chatbot.
3. **Somewhere in the window the chatbot responds in some kind of list** view. The text sent from the user is mirrored there as well.
    a. **The chatbot response contains text, pictures, hyperlinks and maybe** quick response buttons to click



Based on these assumptions an experienced Selenium developer will build a page object model to reuse for all of the chatbot test cases.

3.2.2 Botium Webdriver Connector

If you ever worked with Selenium, you are aware that writing an automation script usually is a time-consuming task. Botium helps you in writing automation scripts for a chatbot widget embedded on a website and speeds up the development process by providing a parameterizable, default configuration for adapting it to your actual chatbot website with Selenium selectors and pluggable code snippets:

- Website address to launch for accessing the chatbot
- Selenium selector for identification of the input text field
- **Selenium selector for identification of the "Send"-Button (if** present, otherwise message to the chatbot is sent with "Enter" key)
- Selenium selector for identification of the chatbot output elements

- **Selenium capabilities for device or browser selection or any other** Selenium specific settings

Note: Botium can work with any Selenium or Appium endpoint available - either with a virtual browser like PhantomJS, an integrated standalone Selenium service, your own custom Selenium grid, or with cloud providers like Perfecto Labs.

If there are additional steps (mouse clicks) to do on the website before the chatbot is accessible, you will have to extend the pre-defined Selenium scripts with custom behaviour as Javascript code.:

```
module.exports = async (container, browser) => {
  const ccBtn = await browser.$('#onetrust-accept-btn-handler')
  await ccBtn.waitForClickable({ timeout: 20000 })
  await ccBtn.click()

  const startChat = await browser.$('#StartChat')
  await startChat.waitForClickable({ timeout: 20000 })
  await startChat.click()
}
```

This code snippet does the following:

1. Waiting for a "Cookie Consent" button to appear on the website
2. Clicking this button to make the website usable
3. **Waiting for a "Start Chat" button to appear and clicking it when** available
4. Waiting until the basic chatbot interaction elements are visible

The full Botium configuration for this scenario looks like this:

```
{
  "botium": {
    "Capabilities": {
      "PROJECTNAME": "WebdriverIO Plugin Sample",
      "CONTAINERMODE": "webdriverio",
      "WEBDRIVERIO_OPTIONS": {
        "capabilities": {
          "browserName": "chrome"
        }
      },
      "WEBDRIVERIO_URL": "https://www.my-company.com",
      "WEBDRIVERIO_OPENBOT": "./snippets/openbot",
      "WEBDRIVERIO_INPUT_ELEMENT": "//input[@id='textInput']",
      "WEBDRIVERIO_INPUT_ELEMENT_SENDBUTTON": "//button[contains(@class,'bot__send')]
↪",
      "WEBDRIVERIO_OUTPUT_ELEMENT": "//div[contains(@class,'from-watson')]"
    }
  }
}
```

With this configuration, all of your convo and utterances files can be used to run test cases with Botium and Selenium.

3.3 Testing Voice-Enabled Chatbots

When testing voice apps, all of the principles from the previous sections apply as well. Some of the available voice-enabled chatbot technologies natively support both text and voice input and output, such as Google Dialogflow or Amazon Lex. Others are working exclusively with voice input and output, such as Alexa Voice Service. And all the other technologies can be extended with voice capabilities by inserting speech-to-text and text-to-speech engines in the processing pipeline.

For doing serious tests at least the chatbot response has to be available as text to use text assertions. Botium supports several text-to-speech and speech-to-text engines for doing the translations.

In addition to the well-known cloud services from Google and Amazon, Botium also has its own free and open source speech service included - Botium Speech Processing.

There is one good reason for using voice instead of text as input to your test cases, if there are historic recordings available when transitioning from a legacy IVR system. Such libraries often are a valuable resource for test data.

CHAPTER

FOUR

BOTIUM USAGE

4.1 Writing Chatbot Test Cases

The *previous section* demonstrated how to write Botium Test Cases in *BotiumScript*. Botium test projects are organized in a directory structure which is

1. extendable
2. overviewable
3. and git-friendly

4.1.1 Anatomy of a Botium Project

The **recommended** directory structure is like this:

- botium.json
- package.json (*Botium Bindings only*)
- spec/
  - botium.spec.js (*Botium Bindings only*)
  - convo/
    - some.convo.txt
    - some.utterances.txt
    - subfolder1/
    - another.convo.txt
    - another.utterances.txt
    - ...
    - subfolder2/
    - onemore.convo.txt
    - ...
    - ...

Starting from the base directory (in the example above *spec/convo*), Botium will recursivly traverse the directory tree. You can choose any directory structure you think is useful in your project.

4.1.2 Skipping/Ignoring Files and Test Cases

Test Cases, files and folders are ignored if:

- the file or folder name starts with *skip**, *skip-* or *skip_* (case is ignored)
- there is a *.gitignore* and the file or folder is matched by one of the rules

4.1.3 Utterance Expansion

Utterance and scripting memory expansion is done dynamically when running test cases.

4.2 Running Chatbot Test Cases

4.2.1 Botium CLI vs Botium Bindings

You should use **Botium Bindings** if:

- you are familiar with Node.js and test runners like Mocha, Jasmine or Jest
- you already have some unit tests in your Node.js project and want to add Botium tests to it

You should use **Botium CLI** if:

- you want to use more Botium Core functionality than just test automation
- you do not want to deploy a new technology (Node.js) to your workstations (Botium CLI is available as Docker)

4.2.2 Using Botium CLI

```
botium-cli run
```

See *Botium CLI Documentation*

4.2.3 Using Botium Bindings

```
npm test
```

See *Botium Bindings Documentation*

4.3 Configuration with Capabilities

This section describes how to configure Botium. **Capabilities** are similar to the "DesiredCapabilities" as used in Selenium and Appium: they describe in what context a Chatbot runs (or should run) and how Botium can connect to it.

4.3.1 Configuration Source

Botium reads configuration from several configuration sources:

- *botium.json* in the current directory
- In case *NODE_ENV* environment variable is set *botium.<node-env>.json* in the current directory
- *botium.local.json* in the current directory - can be used to extract sensitive information out of the botium.json file and should be added to .gitignore
- In case *NODE_ENV* environment variable is set *botium.<node-env>.local.json* in the current directory
- In case the environment variable "BOTIUM_CONFIG" points to a file, same files read as above (botium.json, botium.<node-env>.json, botium.local.json, botium.<node-env>.local.json)
- Environment variables "BOTIUM_capability name" are read and considered

Every step overwrites the configuration capabilities from the previous step.

When using Botium in continuous build / testing / deployment environment, it is generally advised to not include passwords or other secrets in the configuration files, but handing it over with environment variables

The configuration files are JSON files with this anatomy:

```
{
  "botium": {
    "Capabilities": {
      "PROJECTNAME": "My Botium Project",
      "CONTAINERMODE": "echo",
      "SOME_OTHER_CAPABILITIY": "..."
    }
  }
}
```

4.3.2 Connector/Chatbot Technology Selection

The capability **CONTAINERMODE** is one of the most important settings, as it defines the Botium Connector to use. Botium will try to load the connector by several means, in this order:

- If it refers to a relative filename of a custom connector (see Howto develop your own Botium Connector) in the current working directory, this file is loaded and used as connector
- Botium tries to load an NPM module with this name
- Botium tries to load an NPM module named with a "botium-connector-" prefix

A list of well-known Botium Connectors is available here: Botium Connectors

Examples::

```
"CONTAINERMODE": "src/myconnector.js"

"CONTAINERMODE": "echo"

"CONTAINERMODE": "botium-connector-echo"
```

4.3.3 Generic Capabilities

Those capabilities apply to all Chatbot types for all Botium Connectors.

PROJECTNAME

Default: "defaultproject"

The name of the chatbot project. This will be shown in logfiles and reports.

TEMPDIR

Default: "botiumwork"

The working directory for Botium (relative or absolute). For each session there will be a separate unique working directory created in this directory. It will be created if it doesn't exist.

CLEANUPTEMPDIR

Default: "true"

Botium will remove the unique working directory after each session, including all created logfiles and docker containers. In case Botium or a Chatbot doesn't work as expected, this value should be changed to "false" to keep the logfiles, making it able to investigate.

WAITFORBOTTIMEOUT

Default: "10000" (10 seconds)

When waiting for a Chatbot response, this is the default timeout (in milliseconds). An error will be thrown if Chatbot doesn't answer in time. This can be overruled when using the Botium API (in case long waiting period is expected).

SIMULATE_WRITING_SPEED

Default: "false"

Simulates human typing speed. Falsy, or ms/keystroke. (Average typing speed is about 290 ms/keystroke)

SECURITY_ALLOW_UNSAFE

Default: "true"

If turned off then some features which may damage the run environment, are deactivated.

So is not possible

- to execute own JavaScript code
- change environment variable

4.3.4 Specific Capabilities

Capabilities which are specific to a Botium Connector are documented *for each Botium Connector separately*.

4.3.5 Scripting Capabilities

These capabilities are for fine-tuning the *Botium Scripting behaviour*.

SCRIPTING_MATCHING_MODE

Default: "wildcardIgnoreCase"

Logic to use for comparing the bot response to the utterances:

- **wildcard** to use the asterisk * as wildcard (case sensitive)
- **wildcardIgnoreCase** to use the asterisk * as wildcard (case insensitive)
- **regexp** to use regular expressions (case sensitive)
- **regexpIgnoreCase** to use regular expressions (case insensitive)
- **include** to do a substring matching (case sensitive)
- **includeIgnoreCase** (or includeLowerCase - legacy value) to do a substring matching (case insensitive)

SCRIPTING_ENABLE_MEMORY

Default: false

Enable the *scripting memory*.

SCRIPTING_NORMALIZE_TEXT

Default: true

All texts can be "normalized" (cleaned by HTML tags, multiple spaces, line breaks etc)

SCRIPTING_ENABLE_MULTIPLE_ASSERT_ERRORS

Default for Botium Core: false Default for Botium Box: true

Collect all asserter errors for a conversation step and return all with one test failure (instead of failing on first failure)

SCRIPTING_TXT_EOL

Default: \n

Line ending character for text files.

SCRIPTING_UTTEXPANSION_MODE

Default: all

Logic to use for utterances expansion:

- *all*: using all utterances (number of scripts grows exponential)
- *first*: only take first utterance
- *random*: select random utterances (count: see below)

SCRIPTING_UTTEXPANSION_RANDOM_COUNT

Default: 1

Number of utterances to select by random

SCRIPTING_UTTEXPANSION_INCOMPREHENSION

Default: empty

When expanding utterances, Botium can be instructed to add an INCOMPREHENSION asserter to make sure the chatbot answers with something meaningful. One of the utterances is noted as INCOMPREHENSION.

For example, the INCOMPREHENSION utterance looks like this:

```
INCOMPREHENSION
sorry i don't understand
i didn't get that
can you please repeat
```

Expanded convos will look like this:

test case 1

#me

sending some text

#bot

!INCOMPREHENSION

SCRIPTING_UTTEXPANSION_USENAMEASINTENT

Default: false

In many data collections, the utterance name is the same as the intent the NLU engine should predict. For these cases, this flag can be used to add an INTENT asserter when expanding the utterances to convos.

For example, an utterance looks like this:

MY_INTENT_NAME

user example 1

user example 2

user example 3

Expanded convos will look like this:

MY_INTENT_NAME.L

#me

MY_INTENT_NAME

#bot

INTENT MY_INTENT_NAME

SCRIPTING_MEMORYEXPANSION_KEEP_ORIG

Default: "false"

Used while reading scripting memory from file. If it is set to true then the original convo will be kept

SCRIPTING_MEMORY_MATCHING_MODE

Determines how the variables are extracted from text.

Default: "non_whitespace"

non_whitespace: captures every non whitespace characters:

botsays	capturing text	captured
Your name is Joe.	Your name is $name	Joe.
Your name is John Doe.	Your name is $name	John
Today is 02/15/2019	Today is $today	02/15/2019

word: only take captures word characters:

botsays	capturing text	captured
Your name is Joe.	Your name is $name	Joe
Your name is John Doe.	Your name is $name	John
Today is 02/15/2019	Today is $today	02

joker: capture everything (result is not trimmed!)

botsays	capturing text	captured
Your name is Joe.	Your name is $name	Joe.
Your name is John Doe.	Your name is $name	John Doe.
Today is 02/15/2019	Today is $today	02/15/2019

4.3.6 Excel Parsing Capabilities

See *Composing in Excel files*

4.3.7 CSV Parsing Capabilities

See *Composing in CSV files*

4.3.8 Rate Limiting

Some cloud-based APIs are subject to rate limiting and only allow a fixed number of requests in a defined time period. Botium Core can limit the number of requests sent to the Botium connector.

When running in Botium Box on multiple agents in parallel, these settings are applied to each agent separately.

See Bottleneck project page for details.

RATELIMIT_USERSAYS_MINTIME

The minimum number of milliseconds between two UserSays calls.

Example: use 333 to limit rate to at most 3 calls per second.

RATELIMIT_USERSAYS_MAXCONCURRENT

The maximum number of concurrent calls.

4.3.9 Configuring Generic Retry Behaviour

Botium can be configured to retry test cases on certain error conditions. This is an optional behaviour, but it can help you to avoid flaky tests. Some examples where it makes sense:

- **Connection to the chatbot engine is somehow unstable, leading to** failing test cases, where not the chatbot engine itself is the source of the problem, but the infrastructure
- **Maybe chatbot engine itself occasionally fails on high load, but only** in test environment. Using the retry mechanism it can be avoided to fail in these cases.

The following capabilities are available for various connector operations:

- BUILD
- START
- USERSAYS
- STOP
- CLEAN
- ASSERTER
- LOGICHOOK
- USERINPUT

RETRY_<operation>_ONERROR_REGEXP

Default: nothing

Configure a regular expression or a list of regular expressions (JSON array) to trigger the retry behaviour. Often, a simple substring matching is enough.

RETRY_<operation>_NUMRETRIES

Default: 1

Number of retries in case a retry-able error has been identified

RETRY_<operation>_FACTOR

Default: 1

If more than one retry, you can decide to increase wait times between retries by applying a factor higher than 1 for calculating the time to wait for the next retry

RETRY_<operation>_MINTIMEOUT

Default: 1000 (1 sec)

Given in milliseconds. The minimum timeout to wait for the next retry.

4.4 Botium CLI Documentation

Botium CLI is the command line tool to access Botium Core functionality (and more).

4.4.1 Installation

Botium CLI is available as Node.js module and as Docker image.

Installing as global Node.js module:

```
npm install -g botium-cli
```

Installing into an existing NPM project (package.json exists):

```
npm install --save-dev botium-cli
```

Using the Botium CLI docker image

Instead of installing the NPM package, you can use the Botium CLI docker image instead:

```
docker run --rm -v $(pwd):/app/workdir botium/botium-cli
```

Special considerations:

- You cannot use absolute pathes, but all pathes should be given relative to the current working directory. The current working directory is mapped to the docker container with the *-v* switch (above this is mapped to the current working directory)

- For running the console emulator, you will have to add the *-it* flag to the docker command to enable terminal interactions

```
docker run --rm -v \$(pwd):/app/workdir -it botium/botium-cli emulator console
```

Docker Usage under Windows

When using the above command under Windows, especially with *git bash*, you may receive an error like this:

```
C:\Program Files\Docker\Docker\Resources\bin\docker.exe: Error response from daemon:␣
↪Mount denied:
The source path "C:/dev/xxxxx;C"
doesn't exist and is not known to Docker.
```

In this case you have to disable the bash path conversion:

```
export MSYS_NO_PATHCONV=1
```

4.4.2 Usage

Prepare and run a simple Botium test case:

```
botium-cli init
botium-cli run
```

Get help on the command line options:

```
botium-cli help
```

4.4.3 Botium Capabilities configuration

The chatbot capabilities are described in a configuration file. By default, the file named "botium.json" in the current directory is used, but it can be specified with the "–config" command line parameter. The configuration file holds capabilities, envs and sources. Configuration via environment variables is supported as well.

```
{
  "botium": {
    "Capabilities": {
      "PROJECTNAME": "botium-sample1",
      ....
    },
    "Sources: {
      ....
    },
    "Envs": {
      "NODE_TLS_REJECT_UNAUTHORIZED": 0,
      ....
    }
  }
}
```

4.4.4 Commands

botium-cli init

Prepare a directory for Botium usage:

- Adds a simple botium.json
- Adds a sample convo file

botium-cli init-dev [connector|asserter|logichook]

Setup a boilerplate development project for Botium connectors, asserters or logic hooks in the current directory: * Adds a Javascript source file with the skeleton code * Adds a botium.json with connector/asserter/logic hook registration * Adds a sample convo file

botium-cli run

Automatically run all your scripted conversations against your chatbot and output a test report

botium-cli hello

Runs a connectivity check against your chatbot by sending a message (by default 'hello') and waiting for an answer from bot.

botium-cli nlpanalytics <algorithm>

Runs NLP analytics with the selected algorithm.

- **validate** - run one-shot training and testing of NLP engine
- **k-fold** - run k-fold training and testing of NLP engine

See this article for further information.

botium-cli nlpextract

Extract utterances from selected Botium connector and write to Botium Utterances files. Supported not by all connectors, please check connector documentation. Supported at least by:

- Dialogflow
- IBM Watson
- Amazon Lex
- Wit.ai
- NLP.js

and more to come.

botium-cli *import

Import conversation scripts or utterances from some source (for example, from IBM Watson workspace)

botium-cli inbound-proxy

Launch an HTTP/JSON endpoint for inbound messages, forwarding them to Redis to make them consumable by Botium Core.

See Botium Wiki how to use.

botium-cli emulator

The Botium Console Emulator is a basic command line interface to your chatbot running within Botium. You can record and save your conversation files.:

```
botium-cli emulator console
```

botium-cli crawler-run / botium-cli crawler-feedbacks

The Botium Crawler is command line interface to generate conversations along buttons.

The simplest way you can use it from the same folder where you a *botium.json* file placed. In this case the crawler is going to start with *hello* and *help* entry points, and by default try to make the all possible conversation 5 depth along buttons. By default the result is stored in the *./crawler-result* folder:

```
botium-cli crawler-run
```

The Botium Crawler is able to ask user for feedbacks in case of there are no buttons in the bot answer, so the conversation is stucked before the depth is reached. By default the user feedbacks are stored in *./crawler-result/userFeedback.json* file, and these feedbacks are reused in the next runs. With the following command you can edit (*add*, *remove*, *overwrite*) your stored feedbacks:

```
botium-cli crawler-feedbacks
```

There are many other configuration parameters. For more information see *Botium Crawler*.

4.5 Botium Bindings Documentation

4.5.1 Installation

Botium Bindings is available as Node.js module:

```
npm install botium-cli
```

4.5.2 Usage

You should already have a Node.js project set up with the test runner of your choice (Mocha, Jasmine, Jest supported out of the box). For mocha, you can do it like this:

```
cd my-project-dir
npm init -y
npm install --save-dev mocha
```

The following commands will install Botium Bindings, extend your Mocha specs with the Botium test case runner and run a sample Botium test:

```
cd my-project-dir
npm install --save-dev botium-bindings
npx botium-bindings init mocha
npm install && npm run mocha
```

Here is what's happening:

- Your package.json file is extended with a "botium"-Section and some devDependencies
- A botium.json file is created in the root directory of your project
- A botium.spec.js file is created in the "spec" folder to dynamically create test cases out of your Botium scripts
- A sample convo file is created in the "spec/convo" folder

Place your own Botium scripts in the "spec/convo" folder and Mocha will find them on the next run.

4.5.3 Botium Capabilities configuration

The Botium Capabilities are read from *botium.json*, see *Botium Capabilities*.

4.5.4 Botium Bindings configuration

Configuration settings for Botium Bindings are read from the *botium* section of the *package.json* file. A typical package.json looks like this:

```
{
  "name": "custom",
  "version": "1.0.0",
  "scripts": {
    "test": "mocha spec"
  },
  "devDependencies": {
    "botium-bindings": "latest",
    "botium-connector-echo": "latest",
    "mocha": "latest"
  },
  "botium": {
    "convodirs": [
      "spec/convo"
    ],
    "expandConvos": true,
    "expandUtterancesToConvos": false
  }
}
```

convodirs

The folders to look for the *BotiumScript* test cases.

expandConvos

If you are using BotiumScript with utterances files, enable this to expand the convo files with all possible user examples.

expandUtterancesToConvos

If you are using BotiumScript with utterance files only, enable this to expand all user examples to simple question/response test cases.

expandScriptingMemoryToConvos

If you are using BotiumScript with *Scripting Memory Files*, enable this to expand the scripting memory.

See this repl.it as an example.

4.5.5 Test Runner Configuration

Configuration the test runners (Mocha, Jasmine or Jest) are done in the package.json command line calls to the test runner CLI. For example, to choose another Mocha reporter than the default one, change it in package.json:

```
{
  ...
  "scripts": {
    "test": "mocha --reporter json spec"
  },
  ...
}
```

4.5.6 Test Runner Timeouts

Botium tests can take a rather long time, whereas test runners like Mocha and Jasmine expect the tests to complete within a short period of time. It is possible to extend this period of default 60000ms (60 seconds) by setting the environment variables *BOTIUM_MOCHA_TIMEOUT* / *BOTIUM_JASMINE_TIMEOUT* (milliseconds).

4.6 Botium Crawler Documentation

The **Botium Crawler** is doing the work of detecting the conversation flows supported by your chatbot by itself. It does so by analyzing the **quick responses** offered by your chatbot and **simulating clicks on all of the options** in parallel, following all pathes down **until it reaches the end of the conversation**.

All detected conversation flows along all pathes are saved as Botium test cases and utterance lists and can be used as base for a **regression test set**.

4.6.1 Installation

Install as CLI tool

Install *Botium CLI* (botium crawler is included):

```
npm install -g botium-cli
```

Or you can install directly Botium Crawler:

```
npm install -g botium-crawler
```

Install as Node.js module

You can install botium crawler as library in your own project:

```
npm install botium-crawler
```

4.6.2 Using as CLI tool with Botium CLI

Botium CLI using Botium Crawler is able crawl your chatbot various way according to the parameters, and it is able to generate and store all possible conversations.

Basically there are two command in Botium CLI to use Botium Crawler *crawler-run* and *crawler-feedbacks*.

4.6.3 crawler-run command

Get help on the available parameters:

```
botium-cli crawler-run --help
```

The parameters can be applied in two different ways:

- One is the classic way to add these as command line parameters after the *crawler-run* command.
- The other way is to store these parameters into *botium-crawler.json* file into the root of you working directory and reuse them for the next run. In this case the parameters are read from the *botium-crawler.json* as default.

You are able to generate `botium-crawler.json` file with `–storeParams` flag.

–config

You can set the path of a json configuration file (e.g.: *botium.json*):

```
botium-cli crawler-run --config ./custom-path/botium.json
```

–output

You can set the output folder of the crawler result. By default the path is *./crawler-result*. A *scripts* folder is going to be created under the output path, and the generated convos and utterances are going to be stored here:

```
botium-cli crawler-run --config ./botium.json --output ../custom-output
```

–entryPoints

In the entry points array you can define one or more 'user message' from where the crawler is going to start the conversations. * By default the crawler is going to start with *['hello', 'help']* entry points, if the chatbot has no auto welcome message(s). * If the chatbot has auto welcome messages, than these welcome messages are going to be taken as entry points, if the user do not specify others in this parameter. (see *–numberOfWelcomeMessages* parameter)

```
botium-cli crawler-run --config ./botium.json --entryPoints 'Good Morning' 'Next␣
↪conversation'
```

–numberOfWelcomeMessages

You have to specify the number of auto welcome messages exactly, because the crawler has to wait for these welcome messages before each conversation. By default this is 0. If the bot has auto welcome messages, each generated conversation will start with the auto welcome messages.:

```
botium-cli crawler-run --config ./botium.json --numberOfWelcomeMessages 2
```

–depth

You can specify the depth of the crawling, by default it is 5.:

```
botium-cli crawler-run --config ./botium.json --depth 3
```

–ignoreSteps

You can specify here the array of messages has to be ignore during the crawling process.:

```
botium-cli crawler-run --config ./botium.json --ignoreSteps 'this message is ignored'
```

–incomprehension

You can specify here the array of messages, which has to be considered during incomprehension validation. The result of the validation is going to be stored in *error.log* file in the *output* folder.:

```
botium-cli crawler-run --config ./botium.json --incomprehension 'Unkown command'
```

–mergeUtterances

Setting this flag *true* the same bot answers are going to be merged in one utterance file. By default the flag is *true* to avoid high number of utterance files.:

```
botium-cli crawler-run --config ./botium.json --mergeUtterances false
```

–recycleUserFeedback

When the crawler stuck at a point in the conversation, before *depth* is reached, then the crawler is able to ask the user for answers. If this flag is true, then these feedbacks are going to be stored in *userFeedback.json* file in the *output* folder, and these answers are automatically used during the next run of the crawler. By default the flag is *true*.:

```
botium-cli crawler-run --config ./botium.json --recycleUserFeedback false
```

–waitForPrompt

Milliseconds to wait for the bot to present the prompt ore response. Useful if the bot sends multiple responses at once.:

```
botium-cli crawler-run --waitForPrompt 1000
```

–storeParams

If you would like to generate/overwrite the *./botium-crawler.json* file with you currect parameters, you can turn this flag on. This way the parameter are going to be read from this file for the next run. By default the flag is *false*.:

```
botium-cli crawler-run --config ./botium.json --storeParams true
```

Content of ./botium-crawler.json:

```
{
  "recycleUserFeedback": true,
  "output": "./crawler-result",
  "incomprehension": [],
  "config": "./botium.json",
  "entryPoints": [],
  "numberOfWelcomeMessages": 0,
  "depth": 5,
  "ignoreSteps": [],
  "mergeUtterances": true,
  "waitForPrompt": 100
}
```

Example of crawler-run usage

In this example the botium echo connector will be used, which basically just echoing back what you say. The *botium.json* configuration file looks like this:

```
{
  "botium": {
    "Capabilities": {
      "SCRIPTING_MATCHING_MODE": "wildcardIgnoreCase",
      "CONTAINERMODE": "echo"
    },
    "Envs": {}
  }
}
```

Keeping it simple I set just 'hi' as entry points. The commandline will look like this:

```
$ botium-cli crawler-run --config ./botium.json --entryPoints 'hi'
Crawler started...

----------------------------------------

    hi

#me
hi

#bot
You said: hi


----------------------------------------

This path is stucked before reaching depth.
Would you like to continue with your own answers?  [yes, no, no all]: yes
Enter your 1. answer: I said hi
```

(continues on next page)

(continued from previous page)

```
Do you want to add additional answers? [y/n]: n

---------------------------------------

    hi_I said hi

#me
hi

#bot
You said: hi

#me
I said hi

#bot
You said: I said hi


---------------------------------------

This path is stucked before reaching depth.
Would you like to continue with your own answers?  [yes, no, no all]: no
Saving testcases...
The 'crawler-result/scripts/1.1_HI_I-SAID-HI.convo.txt' file is persisted
Crawler finished successfully
```

The *crawler-result* folder will look like this:

```
crawler-result
    ├── scripts
    │   ├── 1.1_HI_I-SAID-HI.convo.txt
    │   ├── UTT_1.1_HI_I-SAID-HI_BOT_1.utterances.txt
    │   └── UTT_1.1_HI_I-SAID-HI_BOT_2.utterances.txt
    └── userFeedback.json
```

In the next run nothing is asked from the user, because the previous feedbacks are stored in *userFeedback.json*. (Before next run the *crawler-result/scripts* folder has to be emptied.) So now the commandline much simpler than at the previous run:

```
$ botium-cli crawler-run --config ./botium.json --entryPoints 'hi'
Crawler started...
Saving testcases...
The 'crawler-result/scripts/1.1_HI_I-SAID-HI.convo.txt' file is persisted
Crawler finished successfully
```

- The convo file is going to be created, despite something goes wrong with any conversation, but it will be differentiated by a *FAILED* postfix in convo name and filename (e.g.: *1.1_HI_I-SAID-HI_FAILED.convo.txt*).*

4.6.4 crawler-feedback command

With crawler-feedback command you can edit (*add*, *remove*, *overwrite*) your stored feedbacks in *userFeedback.json*:

```
botium-cli crawler-feedback --help
```

–input

You can specify the path of the json file, where the user feedbacks are stored. By default it reads the *./crawler-result/userFeedback.json* if it exits.

–output

You can specify the output path, where the edited feedback has to be stored. By default it is the same as input, so basically the input file is going to be overwritten.

Example of crawler-feedback usage

In this example you have to edit in the previous example stored *userFeedback.json* file. You will overwrite the previously set *I said hi* answer with *I said hello* and then skip the rest:

```
$ botium-cli crawler-feedbacks

----------------------------------------
hi

#me
hi

#bot
You said: hi


----------------------------------------

User answers:
1: I said hi


What would you like to do with these answers? [add, remove, overwrite, skip, skip␣
↪all]: overwrite
Enter your 1. answer: I said hello
Do you want to add additional answers? [y/n]: n

----------------------------------------
hi_I said hi

#me
hi

#bot
You said: hi

#me
I said hi

#bot
```

(continues on next page)

(continued from previous page)

```
You said: I said hi


----------------------------------------

User answers:


What would you like to do with these answers? [add, remove, overwrite, skip, skip␣
↪all]: skip
Edit finished, exiting... Do you want to save your modifications? [y/n]: y
```

Now if I run again the crawler from the previous *crawler-run* example, then the *crawler-result* folder will look like this:

```
botium-cli crawler-run --config ./botium.json --entryPoints 'hi'
```

```
crawler-result
    ├── scripts
    │   ├── 1.1_HI_I-SAID-HELLO.convo.txt
    │   ├── UTT_1.1_HI_I-SAID-HELLO_BOT_1.utterances.txt
    │   └── UTT_1.1_HI_I-SAID-HELLO_BOT_2.utterances.txt
    └── userFeedback.json
```

You can use Botium Crawler as individual CLI tool pretty similar as with Botium CLI

4.6.5 Using as library - API Docs

The Botium Crawler is publishing a *Crawler* and a *ConvoHandler*. See Github Repository for an example.

Crawler Object

The *Crawler* need an initialized *BotiumDriver* from Botium Core or a *config* parameter, which is a json object with the corresponding *Capabilities*. Two callback function can be passed as well. The first for ask user to give feedback for the stucked conversations. The second for validating bot answers. You can find example for these callback functions in the sample code as well.

The *Crawler* has a *crawl* function, with that the crawling process can be triggered. This function parameters are identical with the CLI parameters:

```
crawl ({ entryPoints = [], numberOfWelcomeMessages = 0, depth = 5, ignoreSteps = [] })
```

ConvoHandler Object

The *ConvoHandler* can decompile the result of the *crawl* function with *decompileConvos* function. The *decompileConvos* function result is an object with a *scriptObjects* array and a *generalUtterances* array property.

4.7 Botium Grid Documentation

The Botium Grid allows you to distribute Botium tests over several machines. Just start the Botium Agent on a remote machine and connect your Botium scripts to the remote agent.

4.7.1 Starting the Botium Grid Agent

... with Botium CLI (recommended)

Install the *Botium CLI* and run this command for showing the command line options:

```
botium-cli agent help
```

And this one to start the Botium Agent:

```
botium-cli agent
```

... from Botium Core Source

Clone the *Botium Core Github repository <https://github.com/codeforequity-at/botium-core>* and run:

```
npm install
npm run agent
```

This is meant or developers only, as it requires you to setup npm links to botium-core as well.

4.7.2 Using the Botium Agent

... from Botium Bindings or Botium CLI

Set these capabilities:

- BOTIUMGRIDURL to http://remoteservername:46100 (default Botium Agent port)
- BOTIUMAPITOKEN for setting the Botium API Token

And run your script as usual.

... from another client

Botium Agent exports a very simple HTTP/JSON API (see Swagger definition). This can be used from any programming language capable of doing HTTP communication, or from tools like Tricentis Tosca and Postman/Newman.

4.7.3 Security

Botium Agent should be started with security enabled. The environment variable “BOTIUM_API_TOKEN” is expected to contain the API Token (“password”) the clients should send in all HTTP requests (in HTTP-Header “BOTIUM_API_TOKEN”).

CHAPTER

FIVE

BOTIUMSCRIPT API DOCS

5.1 The Basics

Botium supports running scripted, pre-defined or pre-recorded conversations written as **BotiumScript**. A conversation consists of a collection of

- User inputs which are sent to the chatbot
- Bot responses which are expected to be received from the chatbot
- Asserters and Logic Hooks to add advanced assertion or conversation flow logic
- Scripting Memory to set an get variables

In case there are any differences in the bot responses from the pre-recorded scripts, the script returns a failure notification to the caller. (There are multiple checking methods. You can choose with *SCRIPTING_MATCHING_MODE capability*)

You can compose your scripted conversations using Text, Excel, CSV, YAML, JSON, Markdown files. You can even mix them in a single test. And with precompilers you can use your own custom JSON or Markdown format.

5.1.1 Convos

Convos are the skeleton of the Botium Scripting, they are describing the flow of a conversation.:

```
#me
hello

#bot
Hi!

#me
bye

#bot
Goodbye!
```

5.1.2 Partial Convos

With *partial convos* it is possible to reuse parts of a convo.

For instance, in order to always start and end with a greeting, you can define a partial convo **PCONVO_GREETING** (in a file *greeting.pconvo.txt*):

```
PCONVO_GREETING

#me
hello

#bot
Hi!
```

And another one **PCONVO_BYE** (file *bye.pconvo.txt*):

```
PCONVO_BYE

#me
bye

#bot
Goodbye!
```

Those partial convos can now be included in other convo files:

```
TC_01

#include PCONVO_GREETING

#me
how are you ?

#bot
I am fine

#include PCONVO_BYE
```

Another possible syntax would be:

```
TC_01

#include
PCONVO_GREETING

#me
how are you ?

#bot
I am fine

#include
PCONVO_BYE
```

Or:

```
TC_01

#begin
INCLUDE PCONVO_GREETING

#me
how are you ?

#bot
I am fine

#end
INCLUDE PCONVO_BYE
```

5.1.3 Utterances

With Botium you can separate conversation structure from conversation content using Utterances. They can help you to create multilingual conversations, or alternative messages (like 'bye', and 'goodbye').

For the sample convo script above, the first text sent to the bot is hello - you surly want your chatbot to react on other greetings like hi, good afternoon, ... write an additional utterances file:

```
USER_HELLO_UTT
hi
hello
nice day
```

To use this utterance named USER_HELLO:

```
#me
USER_HELLO_UTT

#bot
Hi!

#me
bye

#bot
Goodbye!
```

To make Botium use the utterances files in your convos:

- *When using Botium CLI*, use the –expandutterances command line switch
- *When using Botium Bindings*, use the expandConvos flag in the package.json configuration

5.1.4 Scripting Memory

You can use Scripting Memory to make your test more dynamic. Within a single Botium conversation, it is possible to push variables to a memory and reuse it later. For example:

- an eCommerce chatbot tells some kind of "order number" ("Your order number is X-1235123")
- BotiumScript asks the bot later for the order status ("pls tell me the status for X-1235123")

You can use the predefined functions of Scripting Memory:

```
#me
My ID is $random10
```

And you can multiply your convo using Scripting Memory File. You can create two convos from your buy-beer convo to check that 2 beers costs 4$, and 3 beers costs 6$.

The scripting memory is enabled by setting the :ref:`SCRIPTING_ENABLE_MEMORY capability <cap-scripting-enable-memory>`.

5.1.5 Asserters and Logic Hooks

Asserters and Logic Hooks are used to inject advanced assertion or conversation logic into the conversation flow. They can be added at any position inside the convo file.:

```
#me
hello
PAUSE 5000
```

PAUSE is one of the integrated logic hooks, which will just wait for a defined amount of time. The text following the asserter/logic hook reference name are the arguments, separated by a pipe sign ("|").

Some asserters and logic hooks are integrated into Botium, others are available as additional NPM packages (like Hyperlink Asserter), and you can develop them on your own using Sample Code.

Logic Hooks and User Input Methods always have to be placed below all text in the convo files, as they are always executed at the last possible point in the processing pipeline.

5.1.6 User Input Methods

Main communication channel between a user and chatbot is text. Some chatbots provide simple user interface elements such as buttons:

```
#me
show me some buttons

#bot
BUTTONS Button1|Button2|Button3

#me
BUTTON Button1
```

BUTTON will make Botium simulate a click on a button. The implementation depends on the connector in use - for example, the Webdriver connector will look for a HTML button and simulate a user click.

You can use Integrated User Inputs, or develop your own.

5.2 Supported File Formats

5.2.1 Composing in Text files

It should be so simple that everyone could compose the conversation files manually. Here is an example for a simple test conversation:

```
Call Me Captain

#me
hello

#bot
Try: `what is my name` or `structured` or `call me captain`

#me
call me captain

#bot
Got it. I will call you captain from now on.

#me
who am i

#bot
Your name is captain
```

Conversation and Partial Conversation Syntax

The rules are simple and concise:

- The first line is the name of the conversation or test case
- The second line up to the first line starting with one of the special tags below (#begin, #me, #bot, #include, #end) is an optional description text
- A line starting with **#me** will send the text following on the next line(s) to your chatbot
  - Anything following the #me in the same line will be the channel to send to - for example: #me #private will send the message to the private channel (Slack only)
  - In case there is a registered utterance detected with matching reference code (see below), the utterance samples are expanded (one conversation for each utterance) and sent to the chatbot
  - If the message to send is not specified, then an empty message will be sent to bot
- A line starting with **#bot** will expect your chatbot to answer accordingly
  - Anything following the #bot in the same line will be the channel to listen to - for example: #bot #general will wait for a message on the #general-channel (Slack only)
  - In case there is a registered utterance detected with matching reference code (see below), your chatbot is expected to answer with one of the sample utterances
  - In case the utterance starts with a "?", the answer is OPTIONAL. Except if it starts with at least two "?". In this case first "?" will be removed, and the remaining is checked normally (without optional).

  - In case the utterance starts with a "!", the answer is checked to NOT match the text or one of the utterances samples. Except if it starts with at least two "!". In this case first "!" will be removed, and the remaining is checked normally (without negation).
  - The OPTIONAL and NOT can be combined. The correct order is first optional then negation: "?!".
  - If the message to receive is not specified, then the answer wont be checked.

- A line starting with **#include** will insert a named partial convo at this place
- A line starting with **#begin** will be used on conversation begin only (mainly for asserters and logic hooks, see next section)
- A line starting with **#end** will be used on conversation end only (mainly for asserters and logic hooks, see next section)
- For partial convos, #begin and #end is ignored

That's it.

Utterances Syntax

- First line contains a "reference code" for the utterances
- Following lines contain sample utterances

In order to have a clear distinction between literal text and reference code, the recommendation is to use a naming scheme with a special prefix, for example UTT_utterancename

Example file:

```
UTT_HELLO
hi
hello
nice day
```

An example for a convo - saying "hello" to the bot should make the bot anwer "hi" or "hello" or any other of the above utterance samples.:

```
Reply to hello

#me
Hello, Bot!

#bot
UTT_HELLO
```

Utterances Args

If an utterance name is followed by additional text, those are used to apply formatting with util.format:

```
UTT_HELLO
hi, %s
hello, %s
nice day
```

When using this utterance list in the *#me*-side of a convo files, you have to add a parameter:

```
Reply to hello

#me
UTT_HELLO bot

#bot
hello
```

The texts sent to the bot are:

- hi, bot
- hello, bot
- nice day bot

In case there is no format specifier given, the extra arguments are concatenated to the utterance, separated by spaces - that's why the third example above is missing the comma

When using this utterance list in the *#bot*-side of a convo file:

```
Reply to hello

#me
Hello, Bot!

#bot
UTT_HELLO user
```

So the texts matched are

- hi, user
- hello, user
- nice day user

Scripting Memory Syntax

It's a visual table format, columns are separated with the ||-character:

```
        |$productName    |$customer
product1|Wiener Schnitzel|Joe
product2|Frankfurter     |Joe
```

File naming convention

- a file named "*.convo.txt" will be considered as conversation file
- a file named "*.pconvo.txt" will be considered as partial conversation file
- a file named "*.utterances.txt" will be considered to contain utterances
- while a file named "*.scriptingmemory.txt" will be considered to contain scripting memory data

5.2.2 Composing in Excel files

The structure is simple and visually appealing.

Conversation and Partial Conversation Syntax

- First column holds the test case name (optional)
- Left column corresponds to the *#me* tag
- Right column corresponds to the *#bot* tag
- An empty row means the convo is over, and the next will start below

Download an example file `with explicit test case names` and another one `without explicit test case names`

If you put the #me and #bot message in the same row, then it is recognized as a simple one question one answer conversation. (You cannot mix this two mode on a single sheet) - download an example file `here`.

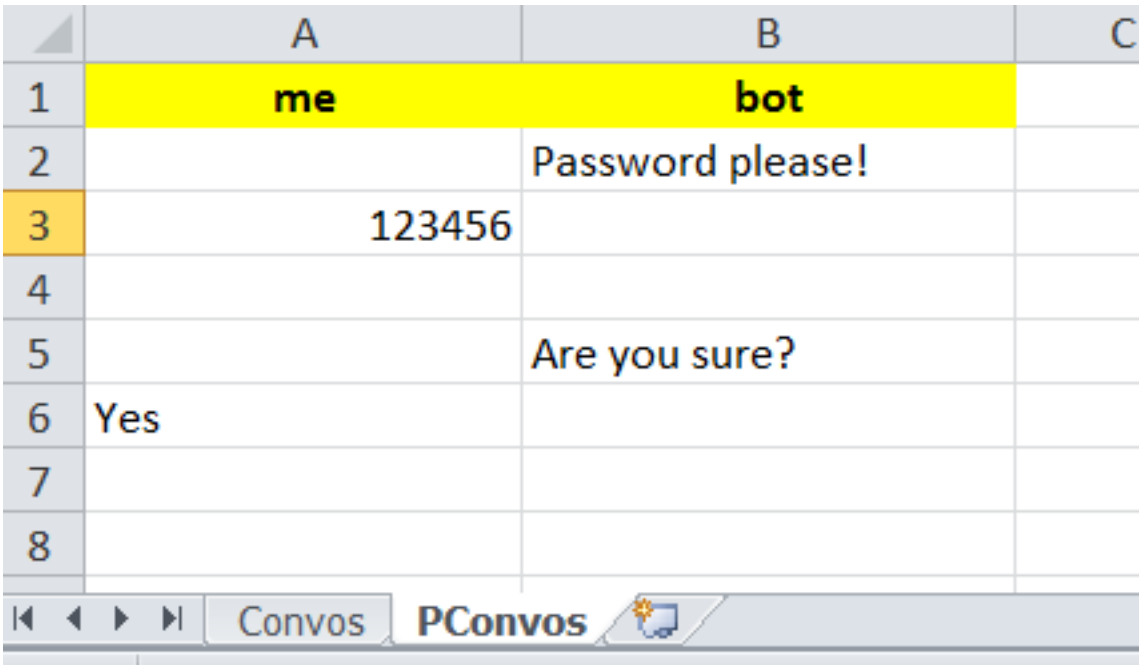
	A	B
1	**User**	**Car**
2	GREETING_NAME Bot	
3		GREETING
4	WHERE_IS_RESTAURANT	
5		Of course. Do you have a specific cuisine in mind?
6	pizza	
7		Super! I've found 5 locations for you. Which one would you like to drive to?
8	first	
9		Sure thing! The first restaurant gets great reviews.
10		What day/time did you want to go to the restaurant?
11	10th of january	
12		OK
13		

Test Case Naming

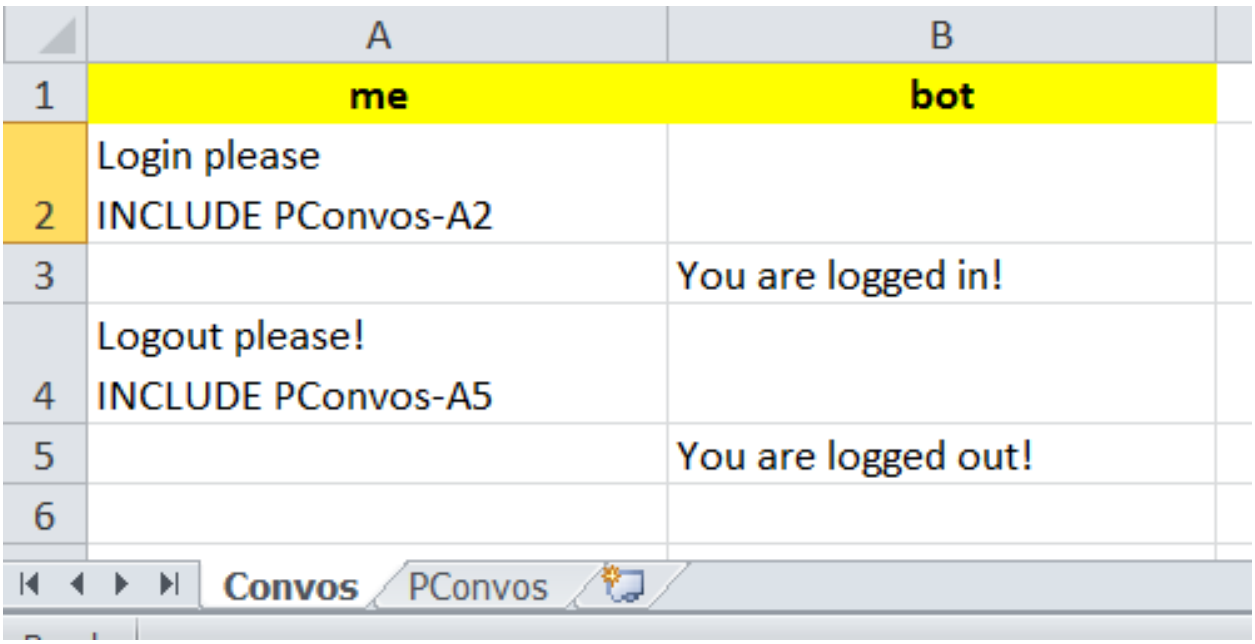
- If the first column contains the test case name, it is used as-is
- Otherwise the test cases are named after the worksheet and the starting cell of the convo in the Excel file - in the above example, the test case is named *Dialogs-A2* (worksheet name + "-" + Excel cell reference)

Partial convos

Partial convos are written same way as test case convos:



They are included by convo name with the *INCLUDE* statement:



Download an example file `here`

Utterances Syntax

- Left column has the utterance name
- Right column holds the list of utterance texts

	A	B
1	**REFCODE**	**UTTERANCE**
2	GREETING	hi
3		hello
4	OK	ok
5		fine
6		super
7	GREETING_NAME	hi, %s
8		hello, %s
9	WHERE_IS_RESTAURANT	where is the next restaurant
10		where is a restaurant
11		

Download an example file `here`

Scripting Memory Syntax

- First column contains the test case name
- Second column contains the variable name as header and the variable value

	A	B
1		
2		$customerName
3	customer1	GoodCustomer
4	customer2	BadCustomer
5		

Download example files `Products` / `Customers` / `Convo`

Specify Excel Worksheets and Regions

You can tell Botium the sheets and the regions to look for convos and utterances using additional capabilities - see below. By default, Botium will identify the content areas in the worksheets automatically by searching for the first filled cell (row by row).

When feeding Botium with **Excel files**, the worksheet names point to either conversations, partial conversations utterances, or scripting memory entries. By default, Botium assumes:

- that all Excel worksheets with name containing "convo" or "dialog" and not "partial" are for convos
- that all Excel worksheets with name containing "utter" are for utterances
- that all Excel worksheets with name containing "partial" are for partial convos
- that all Excel worksheets with name containing "scripting" or "memory" are for scripting memory

You can use these capabilities to tell Botium what worksheets to select for convos, utterances, partial convos and scripting memory:

- SCRIPTING_XLSX_SHEETNAMES
- SCRIPTING_XLSX_SHEETNAMES_UTTERANCES
- SCRIPTING_XLSX_SHEETNAMES_PCONVOS
- SCRIPTING_XLSX_SHEETNAMES_SCRIPTING_MEMORY

Excel Parsing Capabilities

SCRIPTING_XLSX_MODE

Default: ROW_PER_MESSAGE

Set it to QUESTION_ANSWER to force simple question-answer conversations. Botium makes a guess, so usually you dont have to use this cap.

SCRIPTING_XLSX_HASHEADERS

Default: true

When identifying content areas in the excel sheet, the first row usually is a header row and skipped.

SCRIPTING_XLSX_STARTROW

Disable the automatic identification of content areas and use this starting row in the excel sheets to look for convos and utterances. Counting from 1.

SCRIPTING_XLSX_STARTCOL

Disable the automatic identification of content areas and use this starting column in the excel sheets to look for convos and utterances. Counting from 1. You can use column letters here as well ("A", "B", . . .).

SCRIPTING_XLSX_SHEETNAMES

Comma separated list for sheetnames to look for convos. By default, all sheets containing the name "convo" (and not "partial") are used.

SCRIPTING_XLSX_SHEETNAMES_UTTERANCES

Comma separated list for sheetnames to look for utterances. By default, all sheets containing the name "utter" are used.

SCRIPTING_XLSX_SHEETNAMES_PCONVOS

Comma separated list for sheetnames to look for partial convos. By default, all sheets containing the name "partial" are used.

SCRIPTING_XLSX_SHEETNAMES_SCRIPTING_MEMORY

Comma separated list for sheetnames to look for scripting memory. By default, all sheets containing the name "scripting" or "memory" are used.

SCRIPTING_XLSX_EOL_SPLIT

Default: \r

Line ending character in Excel. You shouldn't change this.

SCRIPTING_XLSX_EOL_WRITE

Default: \r\n

Line ending character for Botium assertions. You shouldn't change this.

5.2.3 Composing in CSV files

You can read convos (*.convo.csv), partial convos (*.pconvo.csv) and utterances from CSV file.

CSV File Structure

There are several structures possible. The suggestion is to stick with the default structures, but you can tune them with capabilities, see below.

- First row is the header row (will be skipped)
- Column delimiter is auto-dected (comma, tab, . . .) (can be fixed)
- structure is recognized by number of columns

3 Columns: Multi-Turn Conversations

For multi-turn conversations, there are 3 columns required:

- the "conversationId"-column for grouping conversations together (something unique, no restrictions on format - can be something like the test case name)
- The "sender"-column for Botium to know if to send to the bot or listen for bot responses ("me" or "bot")
- The "text" column for Botium to send to the bot or listen as response

A simple conversation looks like this:

```
conversationId,sender,text
first,me,hello
first,bot,Hi!
```

2 Columns: 1-Turn Conversations (Question/Answer)

There are 2 columns required for question/answer:

- first column contains the question ("#me")
- second column contains the expected answer ("#bot")

A simple conversation looks like this:

```
question,answer
hello,Hi!
```

1 Column: Utterances list

Same format as text utterances file

- first line (header) is the utterance name (header won't be skipped here)
- other lines are the user examples

```
UTT_NAME
hello
Hi!
```

CSV Parsing Capabilities

SCRIPTING_CSV_DELIMITER

Default: auto-detected

Column separator used for CSV format

SCRIPTING_CSV_QUOTE

Default: "

SCRIPTING_CSV_ESCAPE

Default: "

SCRIPTING_CSV_SKIP_HEADER

By default, a header line is expected.

Column Selectors

By default, the column order is according to the structure (see above). If you have a different column order, you can select other columns by specifying the header name (if present), or the column index (starting with 0):

- SCRIPTING_CSV_MULTIROW_COLUMN_CONVERSATION_ID
- SCRIPTING_CSV_MULTIROW_COLUMN_SENDER
- SCRIPTING_CSV_MULTIROW_COLUMN_TEXT
- SCRIPTING_CSV_QA_COLUMN_QUESTION
- SCRIPTING_CSV_QA_COLUMN_ANSWER

5.2.4 Composing in YAML files

```
convos:
  - name: goodbye
    description: desc of convo goodbye
    steps:
      - begin:
          - PAUSE 500
      - me:
          - bye
      - bot:
          - goodbye!
  - name: convo 1 name
    description: desc of convo
    steps:
      - me:
          - GEETING
          - PAUSE:
            - 500
      - bot:
          - NOT_TEXT:
            - hello
          - INTENT:
            - intent_greeting
      - bot:
          - what can i do for you?
      - me:
          - nothing
      - bot:
          - thanks
utterances:
  GREETING:
    - hi
    - hello!
scriptingMemory:
  - header:
      name: scenario1
    values:
      $var1: var1_1
      $var2: var2_1
  - header:
      name: scenario2
    values:
      $var1: var1_2
      $var2: var2_2
```

Starting ! is used to denote the YAML, so quote can help to negate assertions (if using flat strings for assertions).:

```
convos:
  - name: quote
    steps:
      - me:
          - Hello!
      - bot:
          - "!TEXT_CONTAINS_ANY goodbye, bye"
```

When using nested YAML objects for assertions (see example above), prefix the asserter name with NOT_ (! is not allowed to be used as tag names in YAML).

5.2.5 Composing in JSON files

```
{
  "convos": [
    {
      "name": "goodbye",
      "description": "desc of convo goodbye",
      "steps": [
        {
          "begin": [
            { "logichook": "PAUSE", "args": "500" }
          ]
        },
        {
          "me": [
            "bye"
          ]
        },
        {
          "bot": [
            "goodbye!"
          ]
        }
      ]
    },
    {
      "name": "convo 1 name",
      "description": "desc of convo",
      "steps": [
        {
          "me": [
            "hi",
            "PAUSE 500"
          ]
        },
        {
          "bot": [
            { "asserter": "TEXT", "args": "hello", "not": true },
            { "asserter": "INTENT", "args": "intent_greeting" }
          ]
        },
        {
          "bot": [
            "what can i do for you?"
          ]
        },
        {
          "me": [
            "nothing"
          ]
        },
        {
          "bot": [
            "thanks"
          ]
        }
      ]
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "utterances": {
    "GREETING": [
      "hi",
      "hello!"
    ]
  },
  "scriptingMemory": [
    {
      "header": {
        "name": "scenario1"
      },
      "values": {
        "$var1": "var1_1",
        "$var2": "var2_1"
      }
    },
    {
      "header": {
        "name": "scenario2"
      },
      "values": {
        "$var1": "var1_2",
        "$var2": "var2_2"
      }
    }
  ]
}
```

5.2.6 Composing in Markdown files

```
# Convos
## Test Case 1
- me
  - hello bot
- bot
  - hello meat bag
  - BUTTONS checkbutton|checkbutton2
## Test Case 2
- me
  - hello bot
- bot
  - TEXT
    - hello meat bag
  - BUTTONS
    - checkbutton
    - checkbutton2
## Test Case with utterances
- me
  - UTT_HELLO
# Utterances
## UTT_HELLO
- hi
- hello
```

(continues on next page)

(continued from previous page)

```
- greeting
```

5.3 Using the Scripting Memory

Convos, and utterances are the static. You can't send with them a random number to the bot. Assert that the bot answers the current year, or uses your name if you told it before. But you can do it with scripting memory.

You can think of Scripting Memory as collection of system functions like current year, and variables like your name.

The scripting memory is enabled by setting the :ref:`SCRIPTING_ENABLE_MEMORY capability <cap-scripting-enable-memory>`.

5.3.1 Scripting Memory Variables

Within a single Botium conversation, it is possible to push some information items to a memory and reuse it later. For example:

- an eCommerce chatbot tells some kind of "order number" ("Your order number is X-1235123")
- BotiumScript asks the bot later for the order status ("pls tell me the status for X-1235123")

For a conversation step originating from the chatbot, use $varname as a placeholder. The scripting memory is filled from this part of the chatbot output text:

```
#bot
Your order number is $orderNum
```

For a conversation step originating from the user, again use $varname as a placeholder. The scripting memory will be used to complete the text sent to the chatbot.:

```
#me
pls tell me the status for $orderNum
```

There are some restrictions when choosing a variable name:

- They are all starting with a $, followed by a character (lowercase or uppercase)
- Followed by an arbitrary number of lowercase/uppercase characters and numbers

Using variables

They have two functions, depending on the usage. Lets say you already set a variable $username to Joe. Then you can send my name is Joe to bot in the #me section:

```
#me
my name is $username
```

Or you can use it to assert the response of the bot in #bot section. Did bot answered your name is Joe?:

```
#bot
your name is $username
```

As you see variables are starting with '$' to distinguish them from normal text.

Set variables

But how gets scripting memory data? You have to chose depending on your use case. Most basic case if you set them in convo:

```
#me
what is your name?

#bot
my name is $botname.

#me
Hello $botname!
```

You can use this way if the variable is coming from the bot.

It is not always obious how long is the variable. For example botium will extract &@#? as password from sentence my password is &@#? but there is a *capability* to tune this behavior.

Other case is, when you want to multiply your conversations.

Set the variable from file:

```
      |$toEat           |$toDrink |$costs
Case1 |Two salami pizza |Two cola |30
Case2 |Cheeseburger     |nothing  |8
```

And use it from convo:

```
#bot
Do you want to eat something?

#me
yes, $toEat please!

#bot
And some drink?

#me
$toDrink

#bot
It's $costs dollar.
```

This way we defined two conversations.

Third way is to use logic hooks.:

```
#begin
SET_SCRIPTING_MEMORY name|joe

#bot
what is your name?

#me
$name

#bot
hello $name!
```

As you see this conversation is still static. But can help you to create better managable conversations.

And if you want to clear a variable, you can use CLEAR_SCRIPTING_MEMORY logichook.

5.3.2 Scripting Memory Functions

They are the pretty functions provided by botium, like current year (*$year*), or uniqid (*$uniqid*). Can be send to bot in #me sections, and can be used as asserters in #bot sections same way as variables.

Some of them can even used with parameters - for example *$number(5)* generates 5 digit long random number.

You can assert the response of the bot with functions:

```
#me
What is the current year?

#bot
$year
```

Or you can send them to bot:

```
#me
Current year is $year.
```

You can use parameters:

```
#me
Please call me $random(5).
```

You can use system environment variables:

```
#me
Please authenticate my token $env(MY_PERSONAL_TOKEN)
```

And can execute javascript code with func:

```
#me
What costs 5 beer?

#bot
They costs $func(5*2) dollar
```

List of Functions

- $func(<some javascript code>): Executes JavaScript code. It can has multiple lines. The result of the last row will be injected.
- $env(MY_ENV_VAR): Reads sytem environment variables
- $cap(MY_CAP): Reads Botium capabilities
- $msg(JSONPATH): Reads something from the current Botium message with a JSONPath expression, for example: $msg($.messageText)
- $projectname: Test Project Name
- $testsessionname: Test Session Name
- $testcasename: Test Case Name (Convo Name)

- $date(<date pattern like hh:mm:ss or YYYY-MM-DD>): Pattern specific. You can use this to display date, and/or time.
- $now: date and time. Local specific.
- $now_ISO: date and time in ISO format. Example: “2019-04-13T19:27:31.882Z”
- $now_EN: Example: “4/13/2019, 7:24:48 PM”
- $now_DE: Example: “03.07.2019, 08:33:06”
- $date: Locale specific.
- $date_EN: Example: “4/13/2019”
- $date_DE: Example: “03.07.2019”
- $date_ISO: Example: “2019-4-13”
- $time: Local specific.
- $time_EN: Example: “7:44:11 PM”
- $time_DE: Example: “08:33:06”
- $time_ISO: Example: “19:45:12”
- $time_HH_MM: Example: “19:45” or “01:01“
- $time_HH: Example: “19” or “01“
- $time_H_A: Example: “7 PM”
- $timestamp: 13 digit long timestamp (in ms) like 1557386297267
- $day_of_month: day of month. Example: “26” if the date is 2019-3-26
- $day_of_week: day of week. Local specific. Example: “Monday”
- $month: current month. Local specific. Example: “March”
- $month_MM: current month. Local specific. Example: “03”
- $year: Example: “2019”
- $random10: 10 digit long random number. Example: “6084037818”
- $random(<length>): <length> digit long random number.
- $uniqid: V1. Example: “2e65c580-4fb4-11e9-b543-bf076857f1d1”

5.3.3 Scripting Memory Files

You can reuse the same convo more times with Scripting Memory. You have to enable this feature, depending on what Botium Flavour you are using:

- In Botium CLI, use the –expandscriptingmemory flag
- In Botium Bindings, add the expandScriptingMemoryToConvos setting to package.json
- In Botium Box, enable it in the Advanced Scripting Settings

If you don’t enable it explicitly, the scripting variables won’t get pre-filled from the scripting memory file.

Example 1, 4 convos expanded, dynamic variations

Scripting memory for product:

```
        |$productName
product1|Bread
product2|Beer
```

Scripting memory for order number:

```
            |$orderNumber
orderNumber1|1
orderNumber2|100
```

Convo:

```
#me
Hi Bot, i want to order $orderNumber $productName
```

Example 2, 3 convos expanded, scripted variations

Scripting memory for order:

```
        |$productName|$orderNumber
order1  |Bread       |1
order1  |Beer        |1
order2  |Beer        |100
```

Convo:

```
#me
Hi Bot, i want to order $orderNumber $productName
```

5.4 Using Asserters

Asserter are additional validators for conversations. For Example if you want to check if the links send by the bot are valid references you can use and asserter called HyperLinkAsserter, which is trying to reach the sent links.

5.4.1 Buttons Asserter

Some Chatbots are responding not only with text, but provide simple interaction elements such as buttons, to trigger special or predefined functionality. Botium can assert the existance of such buttons in the chatbot response.

It is possible to use this asserter without parameter. In this case asserter will fail if there are no buttons at all.

Processing button responses depends on the Botium connector to support it. For example, it is supported by the Directline and the Dialogflow connector. Please check the connector documentation.

Example

Imagine a chatbot taking orders for pizza delivery. It has a well-defined inventory of possible pizza sizes and toppings. The user interface should present those options to the user as simple buttons:

```
#me
please send me two salami pizza

#bot
Please select the size of the pizza
BUTTONS Kids|Normal|Family
```

The BUTTONS asserter (arguments: button texts to look out for), used in #bot section, will assert that buttons with text are present in response.

BUTTONS_COUNT and BUTTONS_COUNT_REC

Those asserters will validate the number of buttons (BUTTONS_COUNT_REC will also count nested buttons).

You can use number comparision there (or use a number for equality):

```
BUTTONS_COUNT 2
BUTTONS_COUNT =2
BUTTONS_COUNT >2
BUTTONS_COUNT <=3
```

5.4.2 Media Asserter

Some Chatbots are responding not only with text, but with pictures, videos or other media content. Botium can assert the existance of media attachments in the chatbot response.

It is possible to use this asserter without parameter. In this case asserter will fail if there is no media at all.

Processing media responses depends on the Botium connector to support it. For example, it is supported by the Directline and the Dialogflow connector. Please check the connector documentation.

Example

Imagine a chatbot taking orders for pizza delivery. It has a well-defined inventory of possible pizza sizes and toppings. The user interface should visualize the different sizes by using pictures:

```
#me
please send me two salami pizza

#bot
Please select the size of the pizza
MEDIA kids_pizza.png|normal_pizza.png|family_pizza.png
```

The MEDIA asserter (arguments: media uri to look out for), used in #bot section, will assert that media files are attached in the response.

MEDIA_COUNT and MEDIA_COUNT_REC

Those asserters will validate the number of media content (MEDIA_COUNT_REC will also count nested content).

You can use number comparision there (or use a number for equality):

```
MEDIA_COUNT 2
MEDIA_COUNT =2
MEDIA_COUNT >2
MEDIA_COUNT <=3
```

5.4.3 Forms Asserter

Some Chatbots are responding with form. You can assert the fields of the form using this asserter.

It is possible to use this asserter without parameter. In this case asserter will fail if there are no forms at all.

Processing forms depends on the Botium connector to support it. For example, it is supported by the Directline connector. Please check the connector documentation.

Example

Imagine a chatbot taking orders for pizza delivery using form. Form has two fields, type to set pizza type, and a count field for its count. You can check those fields:

```
#me
i want to order a pizza

#bot
FORMS type|count
```

It means that you excpect two fields, “type”, and “count”, and no more.

Fields have a name, or an ID, and most of them a label before. If you expect ‘type’, and there is a field with label ‘Type of the pizza’ then asserter will accept it, even if its name is not ‘type’.

5.4.4 JSONPath Asserter

This is a generic asserter to assert existance or the value of JSONPath expressions within the underlying chatbot response data. The structure of the data depends on the nature of the connector used - for example, with IBM Watson, the underlying response data is the API response from the Watson HTTP/JSON API.

You can use this asserter for adding your custom assertion logic to BotiumScript.

Example

Imagine an eCommerce chatbot - the response contains the shopping cart in session variables. The following BotiumScript asserts that the cart is available in the session, and the ordered item is in the cart:

```
#me
add to cart 5 bananas

#bot
```

(continues on next page)

(continued from previous page)

```
JSON_PATH $.session.cart
JSON_PATH $.session.cart.item[0].name | banana
```

The JSON_PATH asserter takes one or two arguments:

- First argument is the JSONPath expression to query
- If a second argument is given, the value is compared to the outcome of the JSONPath expression (if the expression results in multiple values, then it is compared to all of them). If not given, then only the existance of the element is asserted.

This asserter always works on the sourceData field of the botMsg, not on the botMsg as a whole.

JSON_PATH_COUNT

This asserter will validate the number of JSONPath results.

You can use number comparision there (or use a number for equality):

```
JSON_PATH_COUNT $.session.cart.item|2
JSON_PATH_COUNT $.session.cart.item|=2
JSON_PATH_COUNT $.session.cart.item|>2
JSON_PATH_COUNT $.session.cart.item|<=3
```

5.4.5 Extending JSONPath Asserter

JSONPath Asserter can optionally be configured with global args in botium.json. Arguments from convo file are handed over and used as specified.

- **argCount** - Number of arguments to expect in the convo file
- **path** - predefined JSONPath expression
- **pathTemplate** - Mustache template for predefined JSONPath expression (based on args)
- **assertTemplate** - Mustache template for assertion value (based on args)

Example 1 - WATSONV1_HAS_CONTEXT

```
{
  "botium": {
    "Capabilities": {
      ...
      "ASSERTERS": [
        {
          "ref": "WATSONV1_HAS_CONTEXT",
          "src": "JsonPathAsserter",
          "args": {
            "argCount": 1,
            "pathTemplate": "$.context['{{args.0}}']"
          }
        }
      ]
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Usage:

```
#bot
WATSONV1_HAS_CONTEXT my-context-variable
```

Example 1 - WATSONV1_CONTEXT

```
{
  "botium": {
    "Capabilities": {
      ...
      "ASSERTERS": [
        {
          "ref": "WATSONV1_CONTEXT",
          "src": "JsonPathAsserter",
          "args": {
            "argCount": 2,
            "pathTemplate": "$.context['{{args.0}}']",
            "assertTemplate": "{{args.1}}"
          }
        }
      ]
    }
  }
}
```

Usage:

```
#bot
WATSONV1_CONTEXT my-context-variable|expected-value
```

5.4.6 Response Length Asserter

This asserter checks the length of the response, and the count of the responses (if there are multiple delivered at once). Typically, a chatbot shouldn't deliver too much information at once.

Example

Imagine a user asking a chatbot for help:

```
#me
please help

#bot
RESPONSE_LENGTH 200|5
```

This asserter takes one or two arguments:

- First argument is maximum length of the bot response

- Second argument is the maximum count of the bot responses - some bots deliver multiple responses at once.

5.4.7 NLP Asserter (Intents, Entities, Confidence)

Natural language enabled chatbots are using some kind of NLP engine in the background to recognize intents and entities for the user input.

This information is not shown to the user directly. It may make sense to assert for the recognized intents and entities instead of the text response of the chatbot - or you can even use it in parallel (assert text and intent confidence for example).

Some NLP engines are pure stateless NLP engines without conversation flow (Like Microsoft Luis). They just returning this NLP information. For this engines you cant assert the responded message (text, buttons, etc), just this NLP information using NLP Asserters.

It is possible to extract statistics with the help of this asserters, comparing expectation with the responses from the NLP engine. You can do it on your own, or you can use the our Botium Coach to do it. (Botium Coach is not published yet. It wont be a standalone tool, will work just in the top of the Botium Box)

Not all Botium connectors support these asserters. It depends if the use chatbot technology exposes this information to Botium. For example, it is supported by the Dialogflow and IBM Watson connectors. Please check the connector documentation.

- **INTENT** (arguments: intent name to look out for), used in #bot section, will assert that bot answered with the specified intent.
- **INTENT_CONFIDENCE** (arguments: minimal accepted confidence, like "70" for 70%), used in #bot section, will assert that bot answered with at least the specified minimal confidence.
- **INTENT_UNIQUE** (no arguments), used in #bot section, will assert that the recognized intent is unique (not alternate intent with same confidence identified).
- **ENTITIES** (arguments: expected entities like "from|to", or minimal entities like "from|..."), used in #bot section, will assert that bot answered with the specified entities.
- **ENTITY_VALUES** (arguments: expected entity values like "2018|2019", or minimal entity values like "2018|..."), used in #bot section, will assert that bot answered with the specified entity values.
- **ENTITY_CONTENT** (arguments: entity and expected values like location|Budapest|Vienna)
  - One ENTITY_CONTENT asserter checks only one entity. Use more asserters to check more.
  - Does not fail if the response has more values as specified in arguments.

The INTENT_CONFIDENCE asserter can be used as global asserter to make sure the recognized confidence is always higher than a defined threshold.

Example

Imagine a chatbot taking orders for pizza delivery. It has a well-defined inventory of possible pizza sizes and toppings. The recognized intent, entities and the confidence should be asserter:

```
#me
please send me two salami pizza

#bot
INTENT I_ORDER_PIZZA
INTENT_CONFIDENCE 70
ENTITIES E_PIZZA_TYPE|E_FOOD
```

(continues on next page)

(continued from previous page)

```
ENTITY_VALUES salami|pizza
Please select the size of the pizza
```

Using ENTITY_VALUES asserter can be confusing sometimes. This assertation will be valid:

```
#me
I want to travel from Berlin to Vienna.

#bot
Im happy to hear it. And where are you now?
INTENT travel

#me
in Münich.

#bot
So you are in Münich, and want to travel from Berlin to Vienna?
You will travel to Berlin on your own?
INTENT travel
ENTITY_VALUES Berlin|Vienna|Münich
```

But maybe it is not what you want. You can be more specific using ENTITY_CONTENT asserter:

```
...
ENTITY_CONTENT FROM|Berlin
ENTITY_CONTENT TO|Vienna
ENTITY_CONTENT LOCATION|Münich
```

(This example works just on Dialogflow, it aggregates entities)

Using the Intent Confidence Asserter globally

A very common use case is to use the Intent Confidence Asserter as global asserter, to make sure to filter out the weakly resolved intents. To make all conversation steps fail where the intent falls below a confidence of 80, add this section to your botium.json:

```
{
  "botium": {
    "Capabilities": {
      ...
      "ASSERTERS": [
        {
          "ref": "INTENT_CONFIDENCE",
          "src": "IntentConfidenceAsserter",
          "global": true,
          "args": {
            "expectedMinimum": 80
          }
        }
      ]
    }
  }
}
```

5.4.8 Text Asserters

You can set globally how to assert response using *SCRIPTING_MATCHING_MODE capability*. You can extend/override this behavior using Text Asserters for each response.

Asserter names

There are more text asserters

- Asserter names are starting with TEXT
  - TEXT...
- The matching mode can be wildcard, regexp, include, and exact match
  - TEXT_WILDCARD...,
  - TEXT_REGEXP...,
  - TEXT_CONTAINS...,
  - TEXT_EQUALS... or simple TEXT...
- You can decide to use more args. With AND (..._ALL...) or OR (..._ANY...).
  - Exact match supports just OR, this postfix ist not allowed there
  - Example names:
    - TEXT..., (ALL or ANY is not allowed)
    - TEXT_CONTAINS_ALL...
    - TEXT_REGEXP_ANY...
- Each asserter can work case insensitive (optional _IC prefix)
  - Example names:
    - TEXT_IC,
    - TEXT_CONTAINS_ALL_IC

Features

Utterances as argument:

```
convos:
  - name: example
    steps:
      - me:
          - Hello!
      - bot:
          - "!TEXT_IC GOODBYE|bye bye"
utterances:
  GREETING:
    - Goodbye
    - Bye
```

This is conversation is in yaml format, because utterances. It will fail if bot says goodbye (bye bye, goodbye, or bye) for greeting. Check is case insensitive, but exact. Wont fail for byebye, or for bye Joe .

Starting ! is used to denote the YAML, so negation is quoted.

TEXT_IC is an alternative of TEXT_EQUALS_IC

Matching modes

Exact match works on the text part of the response. All other asserters on the whole response object (on response json as string).

Matching using joker

You can expect any text:

```
TEXT
```

or no text at all:

```
!TEXT
```

using exact match asserter.

Examples

```
TEXT_WILDCARD_ALL id2_*3|1*4
```

will not accept "Im Joe, my number is 12345, and my ID is id1_123", because noting found for regexp id2_*3

```
TEXT_REGEXP_ALL id1_\d\d\d|[0-9]+
```

will accept "Im Joe, my number is 12345, and my ID is id1_123", because booth regexps are found

```
TEXT_CONTAINS_ANY Joe|Jane|George
```

will accept "Im Joe, my number is 12345, and my ID is id1_123", because Joe is there

```
convos:
  - name: example
    steps:
      - me:
          - Hello!
      - bot:
          - "!TEXT_IC GOODBYE|bye bye"
utterances:
  GREETING:
    - Goodbye
    - Bye
```

This is conversation is in yaml format, because utterances. It will fail if bot says goodbye (bye bye, goodbye, or bye) for greeting. Check is case insensitive, but exact. Wont fail for byebye, or for bye Joe .

Starting ! is used to denote the YAML, so negation is quoted.

TEXT_IC is an alternative of TEXT_EQUALS_IC

5.4.9 Cards Asserter

Some Chatbots are responding not only with text, but with grouped UI elements. If the grouping is not just visual, but has some extra function like paging, or hiding, then it called Card. Botium can assert the existence of such Cards in the chatbot response.

It is possible to use this asserter without parameter. In this case asserter will fail if there are no cards at all.

Processing card responses depends on the Botium connector to support it. For example, it is supported by the Directline and the Dialogflow connector.

Example

Imagine a chatbot taking food orders. In the response there are cards for paging with titles Soup, Pizza, and Dessert. You can assert them:

```
#me
What can i order pls?

#bot
Please choose something from our Menu Card!
CARDS Soup|Pizza|Dessert
```

CARDS_COUNT and CARDS_COUNT_REC

Those asserters will validate the number of cards (CARDS_COUNT_REC will also count nested cards).

You can use number comparision there (or use a number for equality):

```
CARDS_COUNT 2
CARDS_COUNT =2
CARDS_COUNT >2
CARDS_COUNT <=3
```

5.4.10 Negation

It is possible to negate asserters. If you dont expect Button1 and Button2 in response:

```
#bot
!BUTTONS Button1|Button2
```

Some asserters are working without args (see asserter documentation):

```
#bot
BUTTONS
```

Which means, it must be at least one button. It is possible to negate those assertions:

```
#bot
!BUTTONS
```

It will throw error if bot responds with any button.

5.4.11 Register Asserter as Global Asserter

A Global Asserter is called at every convo step. This doesn't make sense for all asserters, but there are some where this makes sense. To use one of the integrated asserters as global asserter, you have to register it as global asserter in botium.json:

```
"ASSERTERS": [
  {
    "ref": "RESPONSE_LENGTH",
    "src": "ResponseLengthAsserter",
    "global": true,
    "args": {
        "globalArg1": 17
    }
  }
]
```

5.5 Using Logic Hooks

Logic Hooks are used to inject advanced conversation logic into the conversation flow. They can be added at any position inside the convo file.

5.5.1 PAUSE

- argument: pause time in milliseconds
- used in a #me section, will pause after text is sent to bot.

5.5.2 WAITFORBOT

- argument: wait timeout in milliseconds
- used in a #bot section, will wait for a bot response for given amount of milliseconds (or forever if nothing is given). See also WAITFORBOTTIMEOUT capability.

5.5.3 INCLUDE

- argument: name of a partial conversation
- will insert the referenced partial conversation in the current conversation

5.5.4 SET_SCRIPTING_MEMORY

- arguments: name of the variable, new value
- Sets/overwrites a variable
- Can be used in #begin, #me, and #bot sections, and in botium,json as global.
- You should start the variable name usually without "$" (Use "$" if you want to use logic hook argument replacement)
- It is executed in the end of the section.

So this wont work as expected:

```
#me
SET_SCRIPTING_MEMORY orderNum|111
pls tell me the status for $orderNum
```

But you can do it this way:

```
#begin
SET_SCRIPTING_MEMORY orderNum|111

#me
pls tell me the status for $orderNum
```

5.5.5 ASSIGN_SCRIPTING_MEMORY

- arguments: name of the variable, JSON-Path expression
- Sets/overwrites a variable from message content
- Can be used in #bot sections only, and in botium,json as global.
- You should start the variable name usually without "$" (Use "$" if you want to use logic hook argument replacement)
- It is executed in the end of the section.

Extract a variable from a card and use it in the next conversation step:

```
validate value
extract value from table and form further utterances based on that value

#me
get invoice details for customer number 435643

#bot
CARDS INVOICE NUMBER
ASSIGN_SCRIPTING_MEMORY invoiceNumber|$.cards[0].content

#me
get invoice due date for invoice number $invoiceNumber

#bot
invoice due date for invoice number $invoiceNumber is 02/12/2020
```

5.5.6 CLEAR_SCRIPTING_MEMORY

- arguments: name of the variable
- Deletes a variable.
- Can be used in #begin, #me, and #bot sections, but not in botium,json as global. (Global clear has no sense. There is nothing to clear there)
- You should start the variable name usually without "$" (Use "$" if you want to use logic hook argument replacement)
- It is executed in the end of the section as SET_SCRIPTING_MEMORY.

5.5.7 UPDATE_CUSTOM

Add custom data to an outgoing message to trigger custom behaviour in the connector (consult documentation of the Botium Connector).

- arguments: custom action, custom field, custom value
- This logic hook is used for triggering custom actions in a connector. You have to consult the connector documentation for the supported custom actions.
- When used in the #begin section, the custom action is called for all convo steps

Using UPDATE_CUSTOM globally

To attach custom data to each and every outgoing message, you can make the UPDATE_CUSTOM logic hook act globally:

```
{
  "botium": {
    "Capabilities": {
      ...
      "LOGIC_HOOKS": [
        {
          "ref": "UPDATE_CUSTOM",
          "src": "UpdateCustomLogicHook",
          "global": true,
          "args": {
            "name": "SET_DIALOGFLOW_QUERYPARAMS",
            "arg": "payload",
            "value": { "key":"value" }
          }
        }
      ]
    }
  }
}
```

5.6 Using User Inputs

Main communication channel between a user and chatbot is text. Some chatbots provide simple user interface elements such as buttons.

Not all user inputs are supported by all connectors. Some connectors only allow text input, others allow file uploads, and others allow filling out forms - it depends on the used technology.

5.6.1 BUTTON

- Will simulate a button click if the connector supports it.
- first argument: button payload
- second argument (optional): button text

Example:

```
sending button

#me
BUTTON Help|Displays Help
```

5.6.2 MEDIA

- Will simulate user sending a picture (url resolved relative to the baseUri or to the convo file) if the connector supports it.
- arguments: pathes to a media files
  - Can include wildcards to run same convo multiple times for multiple files (only loading from folders supported, not from HTTP servers)
- global argument **baseUri**: base media location as URI (for HTTP downloads)
- global argument **baseDir**: base media location directory (default: directory where convo file is located)
- global argument **downloadMedia**: flag if media should be downloaded and attached to message (see Connector documentation if this is required or not)

Example (one file):

```
sending picture file

#me
MEDIA send_this_file.png
```

Example (wildcards):

```
sending audio files

#me
MEDIA audiodirectory/*.wav
```

Most common use case is to send recorded audio files instead of text files for testing voice bots. This is supported by several Botium connectors.

- Dialogflow

- Lex
- Directline (just attachment)

5.6.3 FORM

- To simulate a user filling out a form, typically followed by a simulated button click.
- first argument: field name
- second argument: field value (If second argument is empty, form value will be set to "true")

Example:

```
sending form

#me
FORM text1|something entered
FORM text2|something else
BUTTON Submit
```

5.6.4 Global Arguments

Global arguments can be set in botium.json:

```
{
  "botium": {
    "Capabilities": {
      ...
      "USER_INPUTS": [
        {
          "ref": "MEDIA",
          "src": "MediaInput",
          "args": {
            "downloadMedia": true
          }
        }
      ]
    }
  }
}
```

CHAPTER

SIX

BOTIUM CONNECTORS

6.1 Supported technologies

For a list of all supported technologies see Botium Wiki

6.2 Generic HTTP(S)/JSON Connector

Lots of chatbots connect to a simple HTTP/JSON based API in the backend to handle user requests and prepare the responses. This Botium connector mode enables Botium to connect to such an API.

6.2.1 Features

- Sync API supported. (API is sync if the API messages are coming as HTTP response)
- **Async API is supported** (API is async if the API messages are coming via a webhook) - see below
- **Polling API is supported** (Botium will continue polling an endpoint for chatbot responses) - see below
- All HTTP methods are supported, JSON and Non-JSON content (only JSON content can be processed further)
- Session tracking is supported ("Context"). The context can be initialized and will be updated after each request/response and can be used for example to track a session-id (often called "conversation id") or additional conversation state information.
- Using Mustache templates to dynamically construct HTTP Headers, Body, URL, ConversationId, and StepId.
- You can use in Mustache the context ("context" scope in the mustache template) and message content ("msg" scope). You can reach scripting memory variables too ("msg.scripting") and *all the functions as in scripting memory*.
- The mustache templates are fed with a unique "conversationId" (by conversation) and a unique "stepId" (for each sent message) in the "botium" scope

Important: In case you don't want Mustache to do URL/HTML encoding with your texts (which you typically don't want for HTTP POST endpoints), use triple braces {{{ ... }}} as placeholders instead of two

6.2.2 Mustache Variables

For building up URLs, HTTP Headers, HTTP Body there are Mustache templates used. Several mustache variables are available, this is the Mustache view object constructed:

```
{
  container: <current-container>,
  context: <session-context>,
  msg: <botium-message>,
  botium: {
    conversationId: <uniq-id-for-conversation>,
    stepId: <uniq-id-for-conversation-step>
  },
  fnc: {
    <scripting-memory-functions>,
    jsonify: <json-escape string>
  }
}
```

context.*

This is the current session context, as initialized with the capability **SIMPLEREST_INIT_CONTEXT** and updated on each conversation step from the HTTP response with the capability **SIMPLEREST_CONTEXT_JSONPATH** (see below for details).

For example, if the session context holds something like a sessionId to be part of the message body, this can be done by initializing the session context with:

```
...
"SIMPLEREST_INIT_CONTEXT": { "sessionId": "" },
"SIMPLEREST_CONTEXT_JSONPATH": "$.session",
"SIMPLEREST_BODY_TEMPLATE": {
  "text": "{{msg.messageText}}",
  "sessionId": "{{context.sessionId}}"
},
...
```

msg.*

This is the Botium message in its internal representation. You can use the full structure of the message, see here for details.

- Message Text
- Test Project Name
- Test Session Name
- Convo Name
- ...

botium.*

In this scope for convenience there are unique ids for the conversation and for each conversation step available. A possible use case are chatbot endpoints where the session identifier is to be generated by the client, not by the server.

You can use them in this way:

```
...
"SIMPLEREST_BODY_TEMPLATE": {
  "text": "{{msg.messageText}}",
  "stepId": "{{botium.stepId}}",
  "conversationId": "{{botium.conversationId}}"
},
...
```

In case there are special formatting requirements for those, you can use the capabilities *SIMPLEREST_CONVERSATION_ID_TEMPLATE* and *SIMPLEREST_STEP_ID_TEMPLATE* (see below) as template for generating those values.

fnc.*

Scripting memory functions are available under this scope, see section **Using Scripting Memory within Mustache Templates**

6.2.3 Connecting Generic HTTP(S)/JSON chatbot to Botium

Create a botium.json with this URL in your project directory:

```
{
  "botium": {
    "Capabilities": {
      "PROJECTNAME": "<whatever>",
      "CONTAINERMODE": "simplerest",
      "SIMPLEREST_URL": "...",
      ...
    }
  }
}
```

6.2.4 Supported Capabilities

SIMPLEREST_URL *

This points to the URL of your endpoint. On each request, the URL can be adapted to current context and message by using Mustache rendering.

Example: Constructing the URL from context and message content:

```
https://my-api-website/api/{{context.conversation_id}}/{{msg.messageText}}
```

The URL is constructed by a base URL (https://my-api-website/api/) and extended by a context variable ("conversation_id") which has been retrieved previously and by the text of the current message to send to the chatbot.

SIMPLEREST_METHOD

Default: GET

Either GET or POST

SIMPLEREST_TIMEOUT

HTTP Timeout (default 10 seconds)

SIMPLEREST_HEADERS_TEMPLATE

If you require HTTP headers to be sent to the endpoint (for example for authorization), this is the capability to configure. The headers are constructed as Mustache template and can be extended with current context or message variables.

It is a JSON structure which contains key/value pairs for HTTP headers and values.

Example: Sending an API Token:

```
{ "apiToken": "testapitoken" }
```

SIMPLEREST_BODY_TEMPLATE

For POST endpoints, the HTTP body to be sent to the endpoint. Again, Mustache rendering is applied.

Example: Sending the message text and the current conversation id in the HTTP body:

```
{ "text": "{{msg.messageText}}", "conversation_id": {{context.conversation_id}}" }
```

SIMPLEREST_BODY_RAW

By default, a JSON structure is sent to the HTTP endpoint. If you want to send raw data (for example: x-www-form-urlencoded), set this capability to prevent JSON formating.

SIMPLEREST_BODY_JSONPATH

Available since Botium Core 1.6.2

If your endpoint is delivering multiple independent responses to be shown to the user, this JSONPath Expression can be used to split the HTTP response body into multiple junks to be handled by the other JSONPath Expressions below individually.

Example: this response contains multiple text messages:

```
{
  responses: [
    {
      text: 'text 1',
      media: 'http://botium.at/1.jpg'
    },
    {
      text: 'text 2',
```

(continues on next page)

(continued from previous page)

```
      media: 'http://botium.at/2.jpg'
    },
    {
      text: 'text 3',
      media: 'http://botium.at/3.jpg'
    }
  ]
}
```

Example: Set of capabilities to handle this response:

```
...
"SIMPLEREST_BODY_JSONPATH": "$.responses[*]",
"SIMPLEREST_RESPONSE_JSONPATH": "$.text",
"SIMPLEREST_MEDIA_JSONPATH": "$.media"
...
```

You can see in this example that the *SIMPLEREST_BODY_JSONPATH* capability splits the response into multiple chunks, and the other JSONPath expressions are evaluated **relative to them, 3 times**.

SIMPLEREST_RESPONSE_JSONPATH(*)

This capability is for extracting the actual response texts from the HTTP response body of the endpoint. This is given as a JSONPath Expression (JSONPath online evaluator) and in case your endpoint returns more than one response, there can be given additional capabilities starting with the SIMPLEREST_RESPONSE_JSONPATH-prefix. Every single capability is evaluated against the HTTP response body of the endpoint, yielding one chatbot response message per expression.

Example: the "text" attribute of the JSON response contains the message content:

```
...
"SIMPLEREST_RESPONSE_JSONPATH": "$.text.*"
...
```

Example: there are additional "quick response" elements to be extracted as text:

```
...
"SIMPLEREST_BUTTONS_JSONPATH": "$.quick_response.*"
...
```

Example: add multiple JSONPath expressions as array:

The two examples from above can be combined like this:

```
...
"SIMPLEREST_RESPONSE_JSONPATH": [ "$.text.*", "$.quick_response.*"]
...
```

Example: add multiple JSONPath expressions with separator:

Or like this:

```
...
"SIMPLEREST_RESPONSE_JSONPATH_SOMETHING_COMPLETELY_DIFFERENT": "$.text.*,$.quick_
↪response.*"
...
```

SIMPLEREST_IGNORE_EMPTY

Make Botium skip empty messages from processing (no text, no attachments, no buttons, no nlp . . .)

Default empty content is ignored.

SIMPLEREST_CONTEXT_JSONPATH(*)

The session variables you have to store in the current session context are extracted from the HTTP response body of the endpoint. This is a JSONPath Expression, just use "$" to use the full HTTP response body as session context (default: use full body). Can be specified multiple times, all found sections will be merged.

SIMPLEREST_MEDIA_JSONPATH(*)

JSONPath Expression(s) for retrieving media attachments from response body.

See SIMPLEREST_RESPONSE_JSONPATH how to use it with multiple JSONPath expressions.

SIMPLEREST_BUTTONS_JSONPATH(*)

JSONPath Expression(s) for retrieving buttons from response body.

See SIMPLEREST_RESPONSE_JSONPATH how to use it with multiple JSONPath expressions.

SIMPLEREST_INIT_TEXT

Some chatbots require an introductory "trigger" text from the user to start working (and maybe present a welcome message). To get the conversation rolling, the text in this capability is sent to the endpoint before actually starting the conversation. The context is evaluated (see SIMPLEREST_CONTEXT_JSONPATH), but the text response is ignored.

SIMPLEREST_INIT_CONTEXT

The initial value for the session context.

Example: init context variable "conversation_id":

```
{ "conversation_id": "none" }
```

SIMPLEREST_CONVERSATION_ID_TEMPLATE

Optional Mustache template. If it is not set, then an uuid will be generated.

Example: generating 13 digit long timestamp:

```
{{fnc.timestamp}}
```

SIMPLEREST_STEP_ID_TEMPLATE

Optional Mustache template. If it is not set, then an uuid will be generated.

Example: generating 8 digit long random number:

```
{{#fnc.random}}8{{/fnc.random}}
```

6.2.5 Plugging in Custom Functionality

UPDATE_CUSTOM ADD_QUERY_PARAM

The generic UPDATE_CUSTOM logic hook can be used to add additional query parameters to the URI from the convo file. You can use Mustache variables as well:

```
#me
Hello, World!
UPDATE_CUSTOM ADD_QUERY_PARAM|paramname1|paramvalue
UPDATE_CUSTOM ADD_QUERY_PARAM|paramname2|{{msg.location}}

#bot
Cool, I received additional query parameters from you!
```

The URI will now look something like this:

```
http://my-host/endpoint/msgText?paramname1=paramvalue&paramname2=mylocation
```

UPDATE_CUSTOM ADD_HEADER

The generic UPDATE_CUSTOM logic hook can be used to add additional headers to the HTTP request from the convo file. You can use Mustache variables as well:

```
#me
Hello, World!
UPDATE_CUSTOM ADD_HEADER|headername1|paramvalue
UPDATE_CUSTOM ADD_HEADER|headername2|{{msg.location}}

#bot
Cool, I received additional HTTP headers from you!
```

To add the additional headers for all requests in the current convo file, use the #begin section to set the additional headers:

```
#begin
UPDATE_CUSTOM ADD_HEADER|headername1|paramvalue
UPDATE_CUSTOM ADD_HEADER|headername2|{{msg.location}}

#me
Hello, World!

#bot
Cool, I received additional HTTP headers from you!
```

SIMPLEREST_REQUEST_HOOK

This is a dynamic way to change the request options right before it is sent out. You can use the same variables (**context**, **msg**, **botium**) as in Mustache. You can even change them, but usually you wont.

The format of the request options is described in request npm package

You can use this capability many ways:

- use **JavaScript code as string** in botium.json

```
"SIMPLEREST_REQUEST_HOOK": "requestOptions.body = { bodyField: 'val', bodyField2:␣
↪context.contextField }; context.contextField = 'new value'"
```

- reference a **JavaScript module** in botium.json

```
"SIMPLEREST_REQUEST_HOOK": "my-custom-module"
```

the module must export a single function:

```
module.exports = ({ requestOptions, context }) => {
  requestOptions.body = { bodyField: 'val', bodyField2: context.contextField }
  context.contextField = 'new value'
}
```

- reference a **Javascript file** in botium.json

```
"SIMPLEREST_REQUEST_HOOK": "connectors/simple/my-request-hook.js"
```

Again, the file has to export a single function (see above)

- add **direct Javascript function code**, only when using botium-core as API

```
SIMPLEREST_REQUEST_HOOK: ({ requestOptions, context }) => {
  requestOptions.body = { bodyField: 'val', bodyField2: context.contextField }
  context.contextField = 'new value'
}
```

Hook Function Arguments:

- requestOptions: full HTTP request including method, uri, body, headers (see here for details)
- container: the current container instance
- context: current session context
- msg: input message
- botium: conversationId, stepId (see Moustache variables)

SIMPLEREST_RESPONSE_HOOK

Connector extracts data from response like this if you use SIMPLEREST_RESPONSE_JSONPATH, and SIMPLEREST_BUTTONS_JSONPATH Capabilities:

```
{
  messageText: 'Choose please!',
  buttons: [
    'button1',
```

(continues on next page)

(continued from previous page)

```
    'button2'
  ]
}
```

(See all extractable fields here)

Response hook is a general purpose way to read the response, and update this message object, using JavaScript.

Use them if you want:

- Set a field which as no Capability.
- Set a field which cant be set by its Capability. (Response hook is javascript code, so very flexible)
- Process the response some way
- Update the Mustache contexts (**context**, **msg**, **botium**). What you put there, you can use in Mustache templates later.
- Overwrite a field is set already by other Response-Capability depending on a condition.

You can use this capability same way as SIMPLEREST_REQUEST_HOOK, just with **botMsg** instead of **requestOptions**

- all configuration options apply for this hook as well (reference your own custom module and your own custom Javascript code, . . .)

```
"SIMPLEREST_RESPONSE_HOOK": "botMsg.nlp = {intent: {name: botMsg.sourceData.intent}};"
```

Most likely you want to use it to extract some custom values from the HTTP/JSON response body. You can access this JSON data as part of botMsg - **botMsg.sourceData** contains the full JSON response body.

Hook Function Arguments:

- botMsg: add extracted values here
- botMsgRoot: part of the JSON response pointing to the extracted section for this bot message
  - in case there are multiple botMsg extracted from one JSON response
  - available since Botium Box 2.1
- messageTextIndex
  - available since Botium Box 2.1
- container: the current container instance
- context: current session context
- msg: input message
- botium: conversationId, stepId (see Moustache variables)

SIMPLEREST_START_HOOK/SIMPLEREST_STOP_HOOK

Those hooks work like the other hooks, they are called before starting any Botium conversation and after a conversation is finished and can be used to do some setup and teardown tasks.

Hook Function Arguments:

- container: the current container instance
- context: current session context
- msg: input message
- botium: conversationId, stepId (see Moustache variables)

6.2.6 HTTP Session Setup ("Ping" Request)

Botium will only start as soon as this URL is available (returns a non-error response) - for example, to wait until the bot service in the background is up and running. If there is JSON response returned, it will be added to the session context and can be used in the following Mustache templates.

Common scenario is that the ping request returns a JSON response containing a session id. This can be used in the following HTTP requests for session tracking. The payload of the HTTP response is otherwise ignored.

SIMPLEREST_PING_URL

HTTP-Url for the Ping call (can use Moustache template variables)

SIMPLEREST_PING_VERB

HTTP-Method for the Ping call - GET/POST/PUT/...

SIMPLEREST_PING_BODY

HTTP-Body for the Ping call (can use Moustache template variables)

SIMPLEREST_PING_BODY_RAW

Set to "false" to use plain text body instead of JSON for Ping call.

SIMPLEREST_PING_HEADERS

HTTP-Headers for the Ping call (can use Moustache template variables)

SIMPLEREST_PING_RETRIES

Number of times calling the Ping URL for a valid response (default 6)

SIMPLEREST_PING_TIMEOUT

HTTP Timeout and ping retry interval (default 10 seconds)

SIMPLEREST_PING_UPDATE_CONTEXT

Flag if ping response should be used to update the session context (default true)

6.2.7 HTTP Session Welcome (Start Request)

When starting a conversation, Botium will use this URL to send a first welcome message.

SIMPLEREST_START_URL

HTTP-Url for the Start call (can use Moustache template variables)

SIMPLEREST_START_VERB

HTTP-Method for the Start call - GET/POST/PUT/...

SIMPLEREST_START_BODY

HTTP-Body for the Start call (can use Moustache template variables)

SIMPLEREST_START_BODY_RAW

Set to “false” to use plain text body instead of JSON for Start call.

SIMPLEREST_START_HEADERS

HTTP-Headers for the Start call (can use Moustache template variables)

SIMPLEREST_START_RETRIES

Number of times calling the Start URL for a valid response (default 6)

SIMPLEREST_START_TIMEOUT

HTTP Timeout and retry interval (default 10 seconds)

6.2.8 HTTP Session Teardown ("Stop" Request)

When ending a conversation, Botium will use this URL to teardown the session.

Commen scenario is that the stop request is used for ending a server-side session.

SIMPLEREST_STOP_URL

HTTP-Url for the Stop call (can use Moustache template variables)

SIMPLEREST_STOP_VERB

HTTP-Method for the Stop call - GET/POST/PUT/...

SIMPLEREST_STOP_BODY

HTTP-Body for the Stop call (can use Moustache template variables)

SIMPLEREST_STOP_BODY_RAW

Set to "false" to use plain text body instead of JSON for Stop call.

SIMPLEREST_STOP_HEADERS

HTTP-Headers for the Stop call (can use Moustache template variables)

SIMPLEREST_STOP_RETRIES

Number of times calling the Stop URL for a valid response (default 6)

SIMPLEREST_STOP_TIMEOUT

HTTP Timeout and retry interval (default 10 seconds)

6.2.9 HTTP(S) Inbound Messages

For chatbots delivering messages asynchronously, that means, not as response to an HTTP call, but by doing outbound calls to another HTTP endpoint, it is possible to connect it to Botium as well. Botium has to launch an additional HTTP endpoint where the chatbot has to post it's responses.

There are capabilities to define what inbound messages are currently accepted by the Botium script running. Most likely, you will have some kind of session identifier included to let Botium know what messages belong together.

Response handling with JSONPaths for texts, media and buttons is the same as for the synchronous mode, see above.

When using Botium Box, the endpoint is integrated, you won't have to launch any external service. The endpoint is reachable at this url, and a valid API Key has to be appended as query parameter:

```
http(s)://your-botium-box-url/api/inbound?APIKEY=...
```

In all other cases, you will have to either make Botium core launch it's own internal API endpoint (using the *SIMPLEREST_INBOUND_ENDPOINT* and *SIMPLEREST_INBOUND_PORT* capabilities), or use Botium CLI to launch the endpoint (*botium-cli inbound-proxy*)

SIMPLEREST_INBOUND_REDISURL

Url of the Redis service to distribute

Not required with Botium Box, it always uses the default Redis service

SIMPLEREST_INBOUND_ENDPOINT and SIMPLEREST_INBOUND_PORT

Port and endpoint path to launch the endpoint

Not required with Botium Box, it launches it's own endpoint

SIMPLEREST_INBOUND_SELECTOR_JSONPATH

A valid JSONPath selector for extracting the session identifier from the received message. The received message will be handed in, including the full URL of the HTTP request (for identifying session identifiers included in the URL), the HTTP method used and the full message body:

```
{
  "originalUrl": "/api/inbound/xxxx",
  "originalMethod": "POST",
  "body": {
    ...
  }
}
```

The result of the JSONPath selector is compared with the SIMPLEREST_INBOUND_SELECTOR_VALUE to decide if the message belongs to the current Botium session.

SIMPLEREST_INBOUND_SELECTOR_VALUE

A Mustache template for comparing the inbound selector result.

Most likely you will have something like a unique session id or a unique user id, maybe generated by Botium in the Mustache element *botium.conversationId* - so this configuration capability will be something like *"{{botium.conversationId}}"*

SIMPLEREST_INBOUND_UPDATE_CONTEXT

Flag if inbound responses should be used to update the session context (default true)

SIMPLEREST_INBOUND_ORDER_UNSETTLED_EVENTS_JSONPATH

Filling this capability a debounce wrapped function is activated. All inbound requests are collecting in a certain timeout (by default 500 ms) and order these requests in the debounce function by using the JSONPath set in the capability. This has to be valid JSONPath selector for extracting the values for ordering from the received messages. The received messages will be handed in, including the full URL of the HTTP request, the HTTP method used and the full message body:

```
{
  "originalUrl": "/api/inbound/xxxx",
  "originalMethod": "POST",
  "body": {
    ...
  }
}
```

The result of the JSONPath selector is used to order the request ascending. Normally it should be e.g. a timestamp in the body:

```
$.body.timestamp
```

SIMPLEREST_INBOUND_DEBOUNCE_TIMEOUT

A timeout for the debounce wrapped function described in the previous SIMPLEREST_INBOUND_ORDER_UNSETTLED_EVENTS_JSONPATH capability section. By default it is 500 millisec.

6.2.10 HTTP(S) Polling

Chatbots can deliver messages asynchronously by expecting the client to continuously poll for new messages available for a specific user or channel. Botium can do this polling and processes the response messages the same way as if received synchronously.

Response handling with JSONPaths for texts, media and buttons is the same as for the synchronous mode, see above.

SIMPLEREST_POLL_URL

Botium will poll this URL. If there is JSON response returned, this will be processed the same way as synchronous responses.

SIMPLEREST_POLL_VERB

HTTP-Method for the Poll call - GET/POST/PUT/...

SIMPLEREST_POLL_BODY

HTTP-Body for the Poll call (can use Moustache template variables)

SIMPLEREST_POLL_BODY_RAW

Set to "false" to use plain text body instead of JSON for Poll call.

SIMPLEREST_POLL_HEADERS

HTTP-Headers for the Poll call (can use Moustache template variables)

SIMPLEREST_POLL_TIMEOUT

HTTP Timeout for Poll request (default 10 seconds)

SIMPLEREST_POLL_INTERVAL

Polling interval (default 1 second)

SIMPLEREST_POLL_UPDATE_CONTEXT

Flag if polling responses should be used to update the session context (default true)

6.2.11 User Authentication

HTTP Basic Authentication can be done by adding username/passwort to the URLs - this works for all sections (ping, polling, request/response):

```
...
"SIMPLEREST_URL": "http://my-username:my-password@myhost.com/endpoint"
"SIMPLEREST_POLL_URL": "http://my-username:my-password@myhost.com/polling"
...
```

Token-based authentication can be done by adding HTTP headers - for example, from an environment variable:

```
...
"SIMPLEREST_HEADERS_TEMPLATE": {
  "Authorization": "Bearer {{fnc.env}}MY_TOKEN{{/fnc.env}}
},
...
```

A common scheme is to first generate a session token with an initial request, and use this for subsequent calls:

```
...
"SIMPLEREST_PING_URL": "some url",
"SIMPLEREST_PING_VERB": "POST",
"SIMPLEREST_PING_HEADERS": {
  "token": "{{#fnc.env}}MY_TOKEN{{/fnc.env}}"
},
"SIMPLEREST_PING_BODY": { some json content for the body },
...
"SIMPLEREST_URL": "...",
"SIMPLEREST_HEADERS_TEMPLATE": {
  "sessionid":"{{context.sessionid}}"
},
...
```

6.2.12 HTTP(S) Proxy Support

Set a HTTP(S) proxy by setting the capability SIMPLEREST_PROXY_URL to the full url of the proxy:

```
...
"SIMPLEREST_PROXY_URL": "http://myproxy.com:3128"
...
```

You can use proxy authentication as well:

```
...
"SIMPLEREST_PROXY_URL": "http://my-username:my-password@myproxy.com:3128"
...
```

6.2.13 Dealing with SSL Certificates

Setting the capability *SIMPLEREST_STRICT_SSL* to false (default: true) will disable the SSL certificate validity check (accepting outdated certificates)

In order to accept self-signed certificates or certificates not signed by an accepted CA, set the **system environment variable** (not Botium capability) *NODE_TLS_REJECT_UNAUTHORIZED* to 0.

6.2.14 Using Scripting Memory within Mustache Templates

You can use all Scripting Memory features of Botium in the Mustache templates. *Scripting memory variables* are available in the *msg.scriptingMemory* namespace, *Scripting Memory functions* are available in the *fnc* namespace.

Some Mustache examples

- Using length scripting memory variable: {{msg.scriptingMemory.length}}
- Using year function: {{fnc.year}}
- Using random function with parameter: {{#fnc.random}}5{{/fnc.random}}
- Using random function with parameter from scripting memory: {{#fnc.random}}{{msg.scriptingMemory.length}}{{/fnc.random
- Using environment variable: {{#fnc.env}}MY_PERSONAL_TOKEN{{/fnc.env}}
  - Useful for handing over secrets like authentication headers

- Executing code: {{#fnc.func}} 1 + 2 {{/fnc.func}}
- Executing code from scripting memory: {{#fnc.func}}{{msg.scriptingMemory.javaScript}}{{/fnc.func}}

CHAPTER

SEVEN

EXTENDING BOTIUM / BOTIUM CHAMPIONS

7.1 Botium Champions

Botium champion is the award we set up to give out as our appreciation to those who are making an impact on our projects.

Community is the heart of Botium and our champions are driving the community! We would like to express our gratitude to those nerds by giving out some shiny stuffs and the privilege of being our champion!

There are several ways of contribution.

- Are you a restless developer? You can support us by writing code and catching bugs
- Do you have a big network? Share your Botium use cases and experience with your mates
- Are you an expert? Answer community questions on stackoverflow or github (+links)
- Do you have tester genes? Help us by pointing out flaws or suggesting new features
- Are you a researcher? Support us with the scientific proof of coolness

7.2 Contribution Guide

We love Open Source software. Open Source software helped us in our professional careers for the last 20 years, and we are giving back to the community.

All Botium Core code (Botium Core, Botium Bindings, Botium CLI) and most connectors are *hosted on Github <https://github.com/codeforequity-at>* and open for contributions.

7.2.1 Prerequisites

- Good knowledge of Node.js and Javascript
- Knowledge on Git and Github
- In case you have some troubles, please __ALWAYS__ attach the debug output from Botium
- Please don't post any secret information (like access keys for Dialogflow or IBM Watson)

7.2.2 Guidelines for code contributions

Of course, code contributions are welcome! There are many possible ways of extending Botium.

Contributions to Botium Core

- Please fork the repository
- Please create an issue and refer to it in the pull request
- Add unit tests for implemented behaviour
- **The NPM script "npm run build" has to succeed before posting a pull request**
  - it will run unit tests
  - it will enforce eslint and rollup build
- Someone from the core team will review and give feedback within 1-2 days

Contribute Botium Connectors

- There is a Developer Guide available below
- Tell us about your work! We are happy to include your connector in Botium Core and thank you in our release notes!

Contribute Botium Asserters

- There is a Developer Guide available below
- Tell us about your work! We are happy to include your asserter in Botium Core and thank you in our release notes!

We want to recognise everyone making positive influence in our community. We are doing our part of the job by tracking every possible Botium champion out there, but if you feel like that we are missing someone, let us know!

7.2.3 First Steps - Botium Core

Botium Core is the core library for Botium. Some core connectors and BotiumScript is implemented in Botium Core.

Clone and Run Tests

Here are the commands to clone the Botium Core repository, install the dependencies, running the build and running the test suite:

```
git clone https://github.com/codeforequity-at/botium-core.git
cd botium-core
npm install
npm run build
npm test
```

Pull Requests

As usual on Github, you can fork the repository and create a Pull Request with your changes. One of the core developers will get in contact.

We expect that any code changes are covered by unit tests.

Make sure that the commands npm run build and npm test are working with your changes, otherwise the Pull Request will be declined for sure!

Important Files

src/BotDriver.js - the main entry file

Reads the botium.json and other configuration sources

Initializes scripting and containers

src/Capabilities.js - capability names

Holds the list of supported capability names

src/Defaults.js - default capability values

Holds the list of default capability values

src/scripting/ScriptingProvider.js - entry point for BotiumScript

Initializes BotiumScript runtime

Holds context for scripting runtime

Reads BotiumScript files

src/scripting/Compiler*.js - BotiumScript parsers

Parsing the supported BotiumScript file formats

src/scripting/logichooks/asserters/*Asserter.js - integrated asserters

Media asserter, NLP asserters, ...

src/scripting/logichooks/logichooks/*LogicHook.js - integrated logic hooks

Pause, Wait, ...

src/scripting/logichooks/userinputs/*Input.js - integrated user input methods

Buttons, Forms, ...

7.3 Developing Botium Connectors

Howto develop your own Botium Connector - see Botium Wiki

Howto develop your own HTTP/JSON Botium Connector - see Botium Wiki

Howto deploy my own Botium Connector - see Botium Wiki

7.4 Developing Custom Asserters

Developing Custom Asserters - see Botium Wiki

7.5 Developing Botium Logic Hooks

Developing Custom Logic Hooks - see Botium Wiki

7.6 Developing Custom Hooks

There are capabilities for running your custom logic, developed in Javascript, during Botium script execution.

- CUSTOMHOOK_ONBUILD
- CUSTOMHOOK_ONSTART
- CUSTOMHOOK_ONUSERSAYS
- CUSTOMHOOK_ONBOTRESPONSE
- CUSTOMHOOK_ONSTOP
- CUSTOMHOOK_ONCLEAN

There are several options how to inject our Javascript code here. The functions are called with arguments from Botium as nested structure.

- **container**: the currently executing Botium container
- **meMsg** (only for CUSTOMHOOK_ONUSERSAYS): message sent to bot
- **botMsg** (only for CUSTOMHOOK_ONBOTRESPONSE): message received from bot
- **request**: for doing HTTP(S) requests

7.6.1 As NPM module

Your custom NPM module has to export exactly one function.:

```
"CUSTOMHOOK_ONUSERSAYS": "my-own-module",
```

7.6.2 As Javascript file

Your file has to export exactly one function.:

```
"CUSTOMHOOK_ONUSERSAYS": "path/to/your/javascript/file.js",
```

Example function in file.js:

```
module.exports = ({ container, meMsg }) => {
  console.log('in userSays hook');
  meMsg.CUSTOM_VALUE = 'something'
}
```

7.6.3 As Javascript code

Add javascript code snippets to the capability value:

```
"CUSTOMHOOK_ONUSERSAYS": "meMsg.CUSTOM_VALUE='something'",
```

7.7 Custom File Format Precompiler

If you have custom file format, then you have to use a precompiler to convert it to any standard file format.

Supported input and output file extensions are same as supported file extensions of Botium:

- .convo.txt,
- .utterances.txt,
- .pconvo.txt,
- .scriptingmemory.txt,
- .xlsx,
- .convo.csv,
- .pconvo.csv,
- .yaml,
- .yml,
- .json
- .md

You dont have to keep the file extension. (It is possible to convert .md to .json for example)

Output file format can be any standard Botium Script format, but we suggest to use JSON (or YAML) file format. They can contain all parts of a script in a single file, and it is easy to work with.

There is a limitation with precompilers. It is not possible to create more files from one. Using JSON output file format can help handle this limitation.

7.7.1 Configuration capabilities

The preformatters are dynamic. You can use more precompilers, even from the same type. The capabilities are dynamic too, there are many ways to structure them:

Just for one precompiler:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS.NAME": "<precompilername>",
  "PRECOMPILERS.VARIABLE1": "..."
}
```

or:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS": {
    "NAME": "<precompilername>",
    "VARIABLE1": "..."
  }
}
```

same as string:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS": "{\"NAME\": \"<precompilername>\",\"VARIABLE1\": \"...\"}"
}
```

For more precompilers:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS.0.NAME": "<precompilername>",
  "PRECOMPILERS.0.VARIABLE1": "...",
  "PRECOMPILERS.1.NAME": "<precompilername>",
  "PRECOMPILERS.1.VARIABLE1": "..."
}
```

or:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS.0": {
    "NAME": "<precompilername>",
    "VARIABLE1": "..."
  },
  "PRECOMPILERS.1": {
    "NAME": "<precompilername>",
    "VARIABLE1": "..."
  }
}
```

or:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS": [
    {
      "NAME": "<precompilername>",
      "VARIABLE1": "..."
    },
    {
      "NAME": "<precompilername>",
      "VARIABLE1": "..."
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

same as string:

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS": "[{\"NAME\": \"<precompilername>\",\"VARIABLE1\": \"...\"},{\"NAME\
↪": \"<precompilername>\",\"VARIABLE1\": \"...\"}]"
}
```

7.7.2 JSON_TO_JSON_JSONPATH Precompiler

- Compiles not-standard-json using JsonPath.
- This precompiler just supports extraction of utterances.

Capabilities

NAME

Set to JSON_TO_JSON_JSONPATH to use this compiler.

CHECKER_JSONPATH

Optional. If the precompiler does not found anything using this JsonPath, then ignores the source json file.

ROOT_JSONPATH

Optional. Maps the source JSON to utterance struct array. (One entry in map can be mapped to one utterance reference name)

UTTERANCE_REF_JSONPATH

JsonPath to the utterance reference name

UTTERANCES_JSONPATH

JsonPath to the utterances

Example

Source Json:

```
{
  "domains": [
    {
      "name": "Banking",
      "intents": [
        {
          "name": "Transfer",
          "sentences": [
            {
              "text": "Send 2 bucks to savings!"
            }
```

(continues on next page)

(continued from previous page)

```
            ]
          }
        ]
      }
    ]
}
```

Capabilities:

```
{
  PRECOMPILERS: {
    "NAME": "JSON_TO_JSON_JSONPATH",
    "CHECKER_JSONPATH": "$.domains[*].intents[*]",
    "ROOT_JSONPATH": "$.domains[*].intents[*]",
    "UTTERANCE_REF_JSONPATH": "$.name",
    "UTTERANCES_JSONPATH": "$.sentences[*].text"
  }
}
```

7.7.3 SCRIPTED Precompiler

Compiles not-standard Text, Excel, CSV, YAML, JSON, Markdown file toBotiumScript File Formats using JavaScript code.

Capabilities

NAME

Set to SCRIPT to use this precompiler.

SCRIPT

The JavaScript code - it is not a function, do not use *return* there, but set the module.exports variable.

Example

For the sake of simplicity we use JSON file with just utterances as output. But of course all features of all file types can be used.

Basic example, json

```
{
  "PROJECTNAME": "Precompiler",
  "CONTAINERMODE": "echo",
  "PRECOMPILERS": {
    "NAME": "SCRIPT",
    "SCRIPT": "const utterances = {};for (const entry of scriptData) {;
↪utterances[entry.intent] = entry.sentences;};module.exports = { scriptBuffer:
↪{utterances} };"
  }
}
```

Basic example

```
const utterances = {}
for (const entry of scriptData) {
  utterances[entry.intent] = entry.sentences
}
module.exports = { scriptBuffer: { utterances } }
```

scriptData is a predefined variable with the contents of the file. It is string, or JSON, depending on file format.

Set *module.exports* variable with the compiled contents. You can use two fields if you want to process the current file:

- *scriptBuffer* with the compiled content (text or json). Falsy value means, precompiler does not want to change anything.
- *filename* with this field you can change filename, and extension.

Or put the compiled contents direct into result field:

```
module.exports = { utterances }
```

Filtering by filename

```
if (filename.endsWith('.json')) {
  module.exports = ...
}
```

filename is a predefined variable. If you dont process the content, simply dont set *module.exports* field.

Filtering by content

```
if (scriptData.utterances) {
  const utterances = {}
  for (const entry of scriptData.utterances) {
    utterances[entry.intent] = entry.sentences
  }
  module.exports = ...
}
```

Change file extension

Lets suppose we have a json in a text file. And its format is different as Botium standard JSON format. So we have to change the content, and the extension too:

```
const utterances = {}
// creating utterances from scriptData
module.exports = { scriptBuffer:{utterances}, filename: filename + ".json" }
```

CHAPTER

EIGHT

TROUBLESHOOTING

Sometimes things just don't work. Here are some generic insights on how to start investigation.

8.1 Enable Logging

Botium uses the Debug library for logging and debugging purposes. Console output can be enabled by setting an environment variable before starting the Botium session:

```
export DEBUG=botium* # to enable Botium logging
export DEBUG= # to disable Botium logging
```

Please lookup Debug library documentation for details.

There should be pretty detailed output in the console now.

8.1.1 Additional logfiles

In the "botiumwork"-directory, for each run, there is a temporary work directory created, where some connectors place artifacts, logfiles etc. Usually, this directory is cleaned by Botium after finishing work, but sometimes it can be useful to keep the files for investigation. Setting the **CLEANUPTEMPDIR**-capability to "false" prevents the cleanup.

8.2 Problems with Installation

In case you have troubles with "npm install" in any Botium-related module, this is not a problem with Botium itself, but with your Node.js and npm installation. We won't be able to help you on this. But here are some generic steps you could take:

- remove the files "package-lock.json" and the directory "node_modules" and try again
- remove the global npm-cache with the command "npm cache clean –force" and try again
- sometimes you need to compile native modules for installation, please install the windows build tools first "npm install –global windows-build-tools"
- try the yarn package manager instead of npm (just replace "npm" with "yarn" in all npm-related commands)

8.3 Getting help

See *here*