**Automation Framework**

**Version 0.2**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason for Change** | **Version** |
| Aaron Briel | 05/02/2018 | Initial Draft | 0.1 |
| Nathan Abfalter | 05/04/2018 | Added email setup | 0.2 |

[Setup 2](#_Toc513196155)

[Installation/Configuration 2](#_Toc513196156)

[Email Configuration 2](#_Toc513196157)

[IE11 Configuration 3](#_Toc513196158)

[Android Configuration 4](#_Toc513196159)

[iPhone Configuration 5](#_Toc513196160)

[IDE Configuration 5](#_Toc513196161)

[Running Tests 6](#_Toc513196162)

[Viewing Results 7](#_Toc513196163)

[Documentation 7](#_Toc513196164)

[Architecture 7](#_Toc513196165)

[Directory Structure 8](#_Toc513196166)

[Dependencies 9](#_Toc513196167)

[Scripts 9](#_Toc513196168)

[Driver Configurations 9](#_Toc513196169)

[Component-Based, Keyword-Driven Approach 10](#_Toc513196170)

[Adding Tests 10](#_Toc513196171)

# **Setup**

## **Installation/Configuration**

1. Install and configure the latest Java JDK (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>)

2. Install the latest nodejs (<https://nodejs.org/en/download>)

3. If you are running on Windows, install all the required tools and configurations using Microsoft's windows-build-tools with the following command from an elevated PowerShell or CMD.exe run as administrator:

npm install --global production windows-build-tools

4. Go to the main directory of the framework and install all dependencies by running:

npm install

This will take several minutes. This installs the underlying WebDriverIO framework, the browser drivers, and all software dependencies used by the framework.

5. Install the latest versions of all browsers you wish to test on.

6. Install allure reporter command line utility for generating reports:

npm install -g allure-commandline

NOTE:

* If you are running on IE11, follow the steps in the “[IE11 Configuration](#IE11 Configuration)” section.
* If you are running on Android, refer to the section “[Android Configuration](#Android Configuration)”.
* If you are running on iPhone Simulator, refer to the section “[iPhone Configuration](#iPhone Configuration)”.

## **Email Configuration**

1. Turn on the Gmail API. Make sure it’s logged in to the account being used.

a. Use <https://console.developers.google.com/start/api?id=gmail> to create or select a project in the Google Developers Console and automatically turn on the API. Click **Continue**, then **Go to credentials**.

b. On the **Add credentials to your project** page, click the **Cancel** button.

c. At the top of the page, select the **OAuth consent screen** tab. Select an **Email address**, enter a **Product name** if not already set, and click the **Save** button.

d. Select the **Credentials** tab, click the **Create credentials** button and select **OAuth client ID**.

e. Select the application type **Other**, enter the name "Gmail API Quickstart", and click the **Create** button.

f. Click **OK** to dismiss the resulting dialog.

g. Click the  (Download JSON) button to the right of the client ID.

h. Move this file to framework directory in the *src/utility* folder and rename it *client\_secret.json*. If the file already exists, replace it.

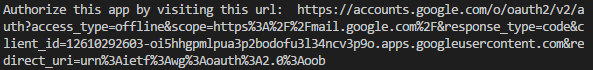
2. Within the project main directory, run the following command:

npm run setupEmail

3. The terminal will prompt you to enter generate or delete a token.



4. Respond with **c** or **create** to start generating a token. The delete functionality is there if there is ever an email change or the token expires. The terminal will again prompt to go to the URL. **Note: image is an example, URL is uniquely generated each time.**



5. After going to the URL the, log in with the Gmail account being used for the tests and it will provide a code to be copied back into the terminal. **Note: image is an example, code is uniquely generated each time.**



6. On success, the terminal will respond with the location the token is stored and say Authorized. If there was already a valid token, the terminal will respond with Authorized if successful. If the token is invalid, an error message will appear, deleting and generating a new token will be needed.

## **IE11 Configuration**

The 32 bit driver is the commonly used driver for this platform as there are issues with the 64 bit version. Currently, WDIO pulls the 64 bit version so we have a config.ie.js file that forces WDIO to pull the 32 bit version from googleapis.

Follow these steps for local configuration of IE11:

1. In IE11, you must set the Protected Mode settings for each zone to be the same value. The value can be on or off, as long as it is the same for every zone. To set the Protected Mode settings, choose "Internet Options..." from the Tools menu, and click on the Security tab. For each zone, there will be a check box at the bottom of the tab labeled "Enable Protected Mode".

2. Additionally, "Enhanced Protected Mode" must be disabled. This option is found in the Advanced tab of the Internet Options dialog.

If you run into issues with IE11, please refer to the following documentation: <https://github.com/SeleniumHQ/selenium/wiki/InternetExplorerDriver> > Required Configuration

## **Android Configuration**

If you wish to run Android emulators, install Android Studio from this link: <https://developer.android.com/studio/> . Create an Android Emulator by following the steps listed here: <https://developer.android.com/studio/run/managing-avds>

If the version of Chrome in the Android emulator is out of date, follow these steps to update it:

1. Download the latest SPK update from: <https://www.apkmirror.com/apk/google-inc/chrome/> (For example, <https://www.apkmirror.com/apk/google-inc/chrome/chrome-65-0-3325-109-release/chrome-browser-65-0-3325-109-8-android-apk-download/>)

2. In the command prompt, navigate to the directory that contains the emulator (Ie, C:\Users\Admin\AppData\Local\Android\Sdk).

3. Run the following command to install the new version of Chrome you downloaded to the emulator:

adb install -r C:\Users\Admin\Desktop\com.android.chrome\_65.0.3325.109-332510962\_minAPI24(x86,x86\_64)(nodpi)\_apkmirror.com.apk

Once the emulator configuration is complete, the process is as follows:

1. Open Android Studio.

2. Start a cmd instance and change the directory to the framework root.

3. Enter appium & to start Appium.

4. Start a new cmd instance and change the directory to C:\Users\Admin\AppData\Local\Android\Sdk\emulator\ (Mac: /Users/USERNAME/Library/Android/sdk/tools)

5. Enter the following command:

emulator -avd EmulatorName -writable-system (where ‘EmulatorName’ is the name of the android emulator you want to run).

## **iPhone Configuration**

Install XCode from the Apple App Store. Once XCode is installed, running tests on iPhone Simulator simply requires you to start Appium in a separate console window. The framework will automatically initialize the iPhone Simulator. Running on an actual iPhone device requires additional configuration steps (COMING SOON!).

## **IDE Configuration**

We highly recommend Microsoft’s free, open source IDE Visual Studio Code (VSC). Steps to configure VSC are here:

1. Download and install Visual Studio Code (<https://code.visualstudio.com/download>).

2. Open VSC and access the framework or drag the framework folder into the workspace area of the explorer bar on the left.

3. Open VSC and click the 'Extensions' icon on the far left menu bar (<https://code.visualstudio.com/docs/editor/extension-gallery>).

4. Install the following extensions from the 'Search Extensions in Marketplace' bar:

-Babel ES6/ES7

-Cucumber (Gherkin) Full Support

-Gherkin step autocomplete

6. Go to File > Preferences > Settings and place the following code in the 'USER SETTINGS' window on the right. This will allow you to drill into steps of feature files:

{

"window.zoomLevel": 0,

"cucumberautocomplete.steps": [

"test/\*/steps/\*.js",

"node\_modules/qa-lib/src/step\_definitions/\*.js"

],

"cucumberautocomplete.syncfeatures": "test/\*/features/\*feature",

"cucumberautocomplete.strictGherkinCompletion": true,

"cucumberautocomplete.smartSnippets": true,

"cucumberautocomplete.stepsInvariants": true,

"cucumberautocomplete.customParameters": [

{

"parameter":"{ab}",

"value":"(a|b)"

}

],

"cucumberautocomplete.pages": {

"users": "test/features/page\_objects/users.storage.js",

"pathes": "test/features/page\_objects/pathes.storage.js",

"main": "test/features/support/page\_objects/main.page.js"

},

"editor.quickSuggestions": {

"comments": false,

"strings": true,

"other": true

},

"files.autoSave": "afterDelay"

}

# **Running Tests**

With set up complete, you can now run test cases. You can run them either in a DOS or PowerShell window (shift + right-click in main framework directory), or directly within VSC by going to View > Integrated Terminal to open the terminal panel.

You can run tests with the following simple command:

npm run <BROWSER>

<BROWSER> can be chrome, firefox, edge, ie11, safari, androidEmulator, iphoneSimulator and others (See the full list in capabilities.json). For example, to run tests on chrome you would execute:

npm run chrome

If you wish to run specific scenarios (tests), simply execute the following command to set the “TAG” environmental parameter. This will set it for the current test session. Here is an example for scenario @login:

Mac: export TAG=@login

Windows: env:TAG="@login"

To reset TAG, set it to the empty string (ie, env:TAG=""). This will run all tests for “Product”. To run specific features, edit the “specs” setting in wdio.conf.js.

To run on a specific environment (currently DEV or PROD, defaulted to DEV), follow the same steps as above to set the TEST\_ENV environmental parameter. Similarly, the same can be done for PRODUCT (defaults to expect “Product”).

## **Viewing Results**

Results are displayed in the console during run-time. Alternatively, upon completion the full report can be viewed under ../results/allure/allure-report/index.html (Please view in Firefox). These are also viewable in the Jenkins runs.

# **Documentation**

Documentation for each class, module, and method is detailed in ../docs/index.html. All classes, etc follow the JSDoc standard (<http://usejsdoc.org/)>. New methods or modules added should thus follow this same standard, with a new JSDoc exported after each change. To export, use the following command from the main project directory:

jsdoc . -r -c /docs/config.json /docs

This command will use recursion to process all jsdoc in subdirectories following the configuration set in config.js to the /docs directory.

# **Architecture**

The Automation Framework is built in JavaScript (ES6). It use NodeJS/npm (<https://nodejs.org/en/>) for dependency management and scripting, and leverages WebDriverIO ([http://webdriver.io](http://webdriver.io/)) for underlying WebDriver calls and the test runner. wdio’s built in cucumber framework (wdio-cucumber-framework) is used, allowing us to write tests in the Gherkin style for ease of readability. This allows manual or even non-technical resources to be able to understand the generated reports and replicate any failed tests. Here is a glimpse of the directory structure of the framework:

## **Directory Structure**

**docs**: Contains JSDoc with descriptions of all methods (see index.html)

**logs**: Contain a full log of selenium (WebDriver) commands for the last test run

**node\_modules**: Contains all dependencies installed as specified in package.json

**results**: Contains allure report (under allure > allure-report), which shows last test run in easy to read HTML format with screenshots

**src**

**api**: Contains stubbed out classes for API calls

**database**: Contains stubbed out classes for database calls

**locators**: Contains subdirectories for each product with JSON files containing all locators for various pages

**utility**: Contains utility classes

**client\_secret.json**: Secret used for email connection

**constants.js**: Contains getter methods for standard settings such as default long wait, base url, or locator path

**driver.js**: Contains standard WebDriver and other methods shared across products

**email.js**: Class with methods for executing API calls to google’s email API

**utility.js**: Stubbed out for inclusion of non-WebDriver utility methods

**test** (with common, and any future products added as subdirectories)

**data**: Contains JSON files for data used with specific test suites

**features**: Contain Cucumber feature files specific to test suites such as login, which in turn contain all scenarios (tests) specific to that test suite

**steps**: Contain step definition methods which each scenario step in features are linked to. Step definition methods make calls to Driver, browser, or Email methods (or others as they may arise)

## **Dependencies**

Being a node project, all dependencies are contained within package.json in the main directory. After “npm install” is run, dependencies are pulled from this file and installed into a node\_modules directory. This file also contains what are called “npm scripts” under scripts. These npm scripts allow one to make calls such as “num run chrome”. What they do is call underlying betterScripts (better-npm-scripts) that match the name of the script called, hence allowing us to set a CAPABILITY environmental parameter to be used throughout execution.

## **Scripts**

The betterScripts make the call to the WebDriverIO (wdio) runner, whose configuration is contained in the file wdio.conf.js, again in the main directory. wdio.conf.js contains various setup options, such as cucumberOpts and reporters, as well as hooks to be executed prior to testruns or before each scenario. For example, beforeScenario contains logic to reload a new browser for each scenario or testcase, ensuring modularity and independence for each test executed. This file also sets the capability, which determines which browser and port we are going to execute on.

## **Driver Configurations**

The wdio runner pulls the capability from capabilities.json, which contains all of the DesiredCapabilities (or WebDriver configurations) for every supported browser or mobile device/emulator. Leveraging capabilities.json ensures a single point of maintenance for all browser-specific driver related configurations.

## **Component-Based, Keyword-Driven Approach**

The framework follows a component-based, keyword-driven approach. What this means is that it is designed to model the application being automated. This allows for maximum re-use of code. To illustrate, refer to ../test/common/steps/common.steps.js. Here we have step definition methods that execute various Driver or browser calls. The methods are page-agnostic. Namely, we can execute them on various pages regardless of indicator. The way this is achieved is through setting the “page context” in the “Given I go to the page” and “Given I am on the page” methods. Subsequent calls to steps definition methods then pull the required page specific locators. For example:

Given I go to the "login" page

The subsequent step to click a certain button is as follows, passing the keyword for the element:

And I click the "LOGIN\_BUTTON" button on the page

The step definition method in common.steps.js linked to this step pulls the specific locator for “login” with the following code, which is pulling the “LOGIN\_BUTTON” element from ../src/locators/Product/login.json:

const button = require(pagePath + `${global.pageContext}.json`).buttons[element]

# **Adding Tests**

1. Add a new feature file for the test suite. Add a feature-scoped tag to the top, for example, @login.

2. Create new scenarios for each test case to be automated. Setting TAG=[Tag\_Name] will allow you to run that specific test.

3. Add new step definition methods as needed. ../test/common/steps/common.steps.js contains most common step definition methods that will be needed. Any unique steps that need to be added can be placed within a new step definition class for the specific test suite. For example, ../test/Product/steps/login.steps.js contains steps that are unique to the “login” test suite.

4. Add new locators files under ../src/locators/Product (or common). These contain page-specific locator definitions.

5. Coding conventions such as file names, method case, naming, and documentation are self-evident throughout the framework.