FE Enterprise Python automation framework guide

shell india

functional excellence team

## **REVISION HISTORY**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Reviewed By** | **Change Description** |
| 12/27/2021 | 1.0 | Praveen Kumar/ Sakthivel | Srish | Details on Enterprise Python automation framework structure, working and functionalities release for Phase I |
| 02/04/2022 | 1.1 | Sakthivel | Praveen Kumar/ Srish | Included details on iOS automation and iOS pre-requisites |
| 03/30/2022 | 1.2 | Praveen Kumar/ Sakthivel/ Chinmaya/ Kallappa | Srish | Included details on Phase II functionalities for Web and Mobile, API automation details |

Contents

[**REVISION HISTORY** 1](#_Toc99574138)

[1. INTRODUCTION 3](#_Toc99574139)

[2. FRAMEWORK ARCHITECTURE 3](#_Toc99574140)

[3. PRE-REQUISITES 4](#_Toc99574141)

[a. APPIUM PRE-REQUISITES 6](#_Toc99574142)

[b. IOS PRE-REQUSITES: 8](#_Toc99574143)

[4. FRAMEWORK FOLDER STRUCTURE 10](#_Toc99574144)

[5. TEST EXECUTION FLOW 17](#_Toc99574145)

[6. CORE FEATURES 18](#_Toc99574146)

[a. SELENIUM CORE 18](#_Toc99574147)

[b. APPIUM CORE 19](#_Toc99574148)

[c. REQUESTS CORE 19](#_Toc99574149)

[7. REPORTING 20](#_Toc99574150)

[8. ACCESSIBILITY TESTING 24](#_Toc99574151)

[9. CLOUD TOOLS INTEGRATION 25](#_Toc99574152)

[a. BROWSERSTACK INTEGRATION 25](#_Toc99574153)

[10. DB INTEGRATION 26](#_Toc99574154)

[11. FILE COMPARISON 27](#_Toc99574155)

[a. EXCEL COMPARISON 27](#_Toc99574156)

[b. IMAGE COMPARISON 28](#_Toc99574157)

[c. PDF PARSING 28](#_Toc99574158)

[12. VISUAL TESTING 28](#_Toc99574159)

[13. RANDOM TEST DATA GENERATION 28](#_Toc99574160)

[14. ENCRYPTION & DECRYPTION 28](#_Toc99574161)

[15. AUTO RERUN 29](#_Toc99574162)

[16. FEATURES AVAILABLE 29](#_Toc99574163)

## INTRODUCTION

The Shell FE Python Automation framework is a robust framework developed by the Functional Excellence team with the aim to provide UI, Mobile and API automation solution for the project automation needs as a single entity.

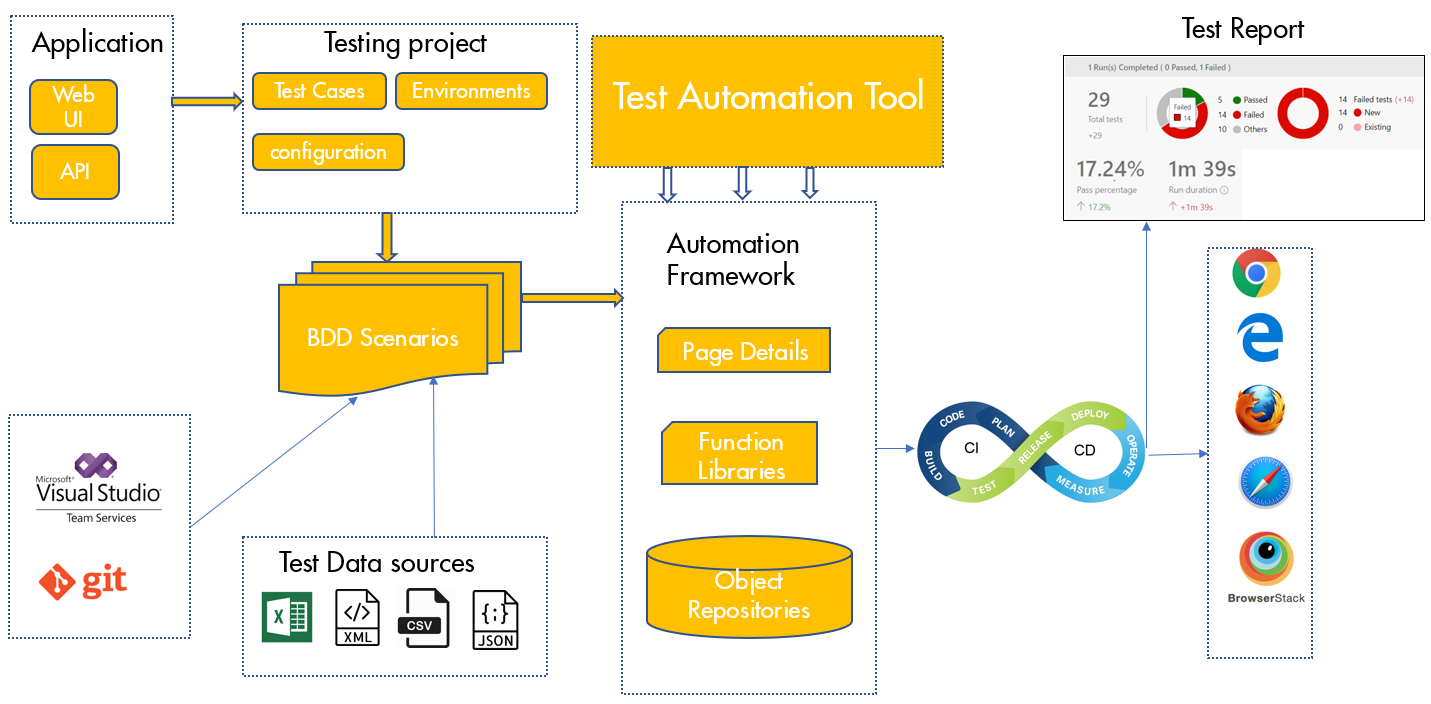
This framework is written using Python and uses Selenium for UI automation, Appium for mobile automation, Requests for API automation.

This framework has features for most of the use cases that users would perform including DB validation, accessibility testing, visual testing, multi browser testing, android and iOS device automation, parallel execution, CI/CD integration of the automated tests and has been developed in a BDD approach providing visually appealing reports for the test results.

NOTE: THE CODE AVAILABLE IN CORE PACKAGES (I.E.) SHELL-FE-SELENIUM-CORE & SHELL-FE-APPIUM-CORE SHOULD NOT BE EDITED BY THE USERS. THE FRAMEWORK MAY STOP WORKING AS INTENDED IF ANY CHANGES ARE MADE BY THE USER TO THE CORE PACKAGES AND ALSO WILL IMPACT THE USAGE OF FUTURE VERSIONS OF CORE PACKAGES RELEASED BY FE TEAM.

## FRAMEWORK ARCHITECTURE

This framework is written in a BDD approach using Behave framework in Python.



Listed below are the salient features of this framework.

* Follows a POM design pattern where element locators and functions specific to a page are written as individual classes
* BDD approach has been implemented using Behave. Tests are written as feature files with the implementation for features available in Step definitions
* Supports UI automation using Selenium
* Supports API automation using Requests library in Python
* Supports both Android and iOS mobile automation using Appium
* Test data parameterization is supported via Excel, Json, CSV, XML and also data from DB
* Supports browser initiation using Webdriver Manager as well as browser driver binaries
* Custom utilities for common actions
* The core features for Selenium, Appium and Requests would be provided as installable python packages
* Parallel execution supported using behave parallel
* DB validation for SQL is provided
* Supports generation of proper logs and screenshots for failed cases
* Rerun of failed test cases
* Supports integration with cloud-based tools such as Browser stack. This can be extended to other tools based on user needs
* Supports automation of accessibility tests
* Supports automation of Visual testing
* Multi browser support (Chrome, Edge, Firefox and Safari)
* Supports integration of CI/CD pipelines with Azure Devops
* Provided visually appealing reports of test execution results using Allure

## PRE-REQUISITES

To set up the framework in the user’s machine the follow the below steps provided in the attached Excel sheet.

|  |  |
| --- | --- |
| **TASK** | **STEPS** |
| **Install Python** | 1. Install Python by following instructions provided in Devkit 2. During installation **unselect** the install launcher for all users option and select the adding Python path option |
| **Install Pycharm** | Download Pycharm by following instructions in Devkit. |
| **Make Pypi.org as trusted** | Execute the following command to make the Python site trusted (if facing any untrusted site issue when installing the required packages): WINDOWS: pip install --trusted-host pypi.org --trusted-host files.pythonhosted.org --trusted-host pypi.python.org python-certifi-win32 MAC: python -m pip install <package name> --only-binary :all: --trusted-host pypi.org --trusted-host files.pythonhosted.org --trusted-host pypi.org |
| **Download the Abstract Shell FE Python automation framework** | Download the Shell FE Python Automation framework from the provided location and open the framework in IDE |
| **Set up Virtual environment** | Do not select Virtual environment from the window that is displayed initially. Follow the below steps to set up virtual environment in Pycharm.  - Open the framework in Pycharm - Go to File->Settings->PythonInterpreter - Click on 'wheel' icon and select ‘Add..’ - If no virtual environment has been configured previously, then select Virtualenv Environment -> New environment (A venv folder would be mapped to the current project path in the 'Location' field) - Select a valid location where python.exe is present in your machine.  - Click on OK  - Go to Terminal in Pycharm and type the below command to activate Virtual environment: venv/scripts/activate  - Once the virtual environment has been activated, the folder path in terminal would be prefixed with (venv) |
| **Install the packages required for framework** | Execute the below command: pip install -r requirements.txt |
| **Set up authentication for downloading the Selenium core packages** | 1. Create a folder 'pip' in the below location: C:\ProgramData (The ProgramData would be a hidden folder; to view it select the 'Hidden items' in the View tab of Windows explorer) 2. Paste the 'pip.ini' file provided by the FE team inside the pip folder |
| **Install the Selenium core package** (if using UI automation) | 1. Install the Selenium core packages using the below command: pip install Shell-FE-Selenium-Core 2. If user is updating the version of Shell-FE-Selenium-Core package, then it is advised to remove the existing package using the below command.   pip uninstall Shell-FE-Selenium-Core |
| **Install the Appium core package** (if using Mobile automation) | 1. Install the Appium core packages using the below command: pip install Shell-FE-Appium-Core 2. If user is updating the version of Shell-FE-Appium-Core package, then it is advised to remove the existing package using the below command.   pip uninstall Shell-FE-Appium-Core |
| **Install the Requests core package** (if using API automation) | 1. Install the Requests core packages using the below command: pip install Shell-FE-Requests-Core 2. If user is updating the version of Shell-FE-Requests-Core package, then it is advised to remove the existing package using the below command.   pip uninstall Shell-FE-Requests-Core |
| **If using Selenium automation, download browser drivers** | 1. Download the latest browser drivers based on the browser version and place it in the WebDriverExecutables folder (Download browser drivers if user is not going to webdrivermanager) |
| **Appium setup** | For mobile automation follow the steps in Appium Pre-requisites |
| **Java setup** | Follow the instructions in Devkit portal and download JDK (>=1.8), set the Java related environment variables. |
| **Allure report** | 1. Download Allure binaries from the below link and save it in a folder. Add the bin folder path of Allure to the 'Path' variable in user environment variables.   [Releases · allure-framework/allure2 · GitHub](https://github.com/allure-framework/allure2/releases)  E.g. C:\Users\User1\Downloads\PythonAutomationBinaries\allure-2.16.1\bin. |

### APPIUM PRE-REQUISITES

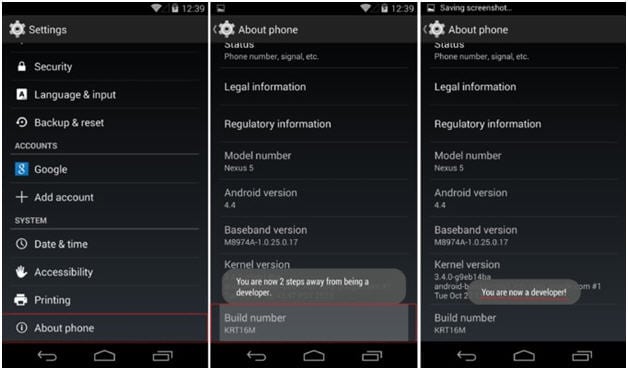
|  |  |
| --- | --- |
| **TASK** | **STEPS** |
| **Download Java** | 1.Download and install java from devkit 2. Add Path as mentioned int the devkit portal |
| **Download NodeJs** | 1.Install latest node js(>=version14.0) from the devkit portal 2.Download the binary , unzip and extract it to say c:\Apps\Nodejs. 3.Now open command prompt to check installation of node. Type node –version. It should display the version |
| **Download Appium** | Install the Appium using the below command: 1.Npm install -g Appium 2. If you get an error while installing Appium then run following 2 commands to add environment variables for HTTP\_PROXY and HTTPS\_PROXY and run aove command again from command prompt: SETX HTTPS\_PROXY "http://nbproxy-hon.asia-pac.shell.com:8080/ SETX HTTP\_PROXY "http://nbproxy-hon.asia-pac.shell.com:8080/ |
| **Add Path for Appium** | Add Appium to path variable. Copy folder path till build\lib and add it to PATH environment variables to your path: C:\Apps\nodejs\node-v14.16.1-win-x64\npm\node\_modules\appium\build\lib (or) C:\Apps\node-v14.16.1-win-x64\npm\node\_modules\appium\build\lib |
| **Download Android Studio** | 1.Download the Android studio from the below link to get the android sdk: https://devkit.shell.com/content/tools/androidstudio 2.Once it is installed go to "More options>SDK manager> get the location of the AndroidSDK" NOTE: Need dev rights to install or the user can get help from IT team for installation |
| **Add ANDROID\_HOME path** | Add ANDROID\_HOME to your environment variable. Value is location of android sdk. Add following new values to PATH variable: %ANDROID\_HOME% %ANDROID\_HOME%\platform-tools %ANDROID\_HOME%\tools %ANDROID\_HOME%\tools\bin %ANDROID\_HOME%\android-sdk\tools\bin |
| **Check Appium and Android Home** | To Check if Appium and Android Sdk variables are working fine, open command prompt and check following commands: appium : This command should start Appium server on your local machine echo "%Android\_Home%" : This command should display location of android sdk. |
| **Download Appium GUI** | If want to use AppiumDesktop GUI for inspecting locators, download from the official Appium site https://appium.io/ |
| **Add Path in appium GUI** | Once Appium is installed, Click Edit Configuration and add the  1.Android sdk path  2. Java Home |

**Preparing mobile device for automation with Appium:**

Connect your mobile testing device via USB to PC. Enable developer mode on mobile phone before automating the app.

**Steps to enable developer mode / options on Android phone / tablet:**

* Open Settings, Tap on ‘About’ Option, Tap on ‘Software Information’, Tap On ‘More’.
* Tap on “Build number” 7 times to enable Developer options.
* Go back to Settings and make sure that “Developer options” is there.
* Tap on Developer options and turn on USB Debugging option from the menu on the next screen.



To check if the mobile device has been attached follow the below steps:

* In adb terminal type ‘adb devices’ [location of adb.exe is found under /android-sdk/platform-tools  
  C:\Users\{UserName}>adb devices  
  List of devices attached  
  MyDevice device

### IOS PRE-REQUSITES:

**\*\*DEVELOPER RIGHTS IS REQUIRED TO PERFORM THE iOS AUTOMATION**

|  |  |
| --- | --- |
| **TASK** | **STEPS** |
| **Download Java** | 1.Download and install java (>=1.8) from devkit portal |
| **Download XCode** | 1.Download the XCode from Shell app store or Appstore  2. After installation move the XCode to application folder Then we need to execute this below query  **sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer**  3**.**To check the XCode pat, execute the query below  **xcode-select -p** |
| **Download NodeJs** | 1.Install latest nodejs (>=version14.0) from the devkit portal 2.Now open command prompt to check installation of node.  3.Type node –version. It should display the version |
| **Install HomeBrew** | 1. Open terminal then run this command and use the below command:  **usr/bin/ruby -e “$(curl -fsSL** [**https://raw.githubusercontent.com/Homebrew/install/master/install**](https://raw.githubusercontent.com/Homebrew/install/master/install)**)”**  2.Once Installation is completed, execute the query below to check the brew version:  **brew  — version** |
| **Install Carthage** | 1.Install Carthage using the command:  **brew install carthage** |
| **Install Appium and Webdriver components** | Install the Appium using the command below: **1.npm install -g Appium**  **2.npm install wd** |
| **Install Appium-doctor** | 1.Appium Doctor is used to check all the pre-requisites have been installed properly. Run the command below to install appium-doctor:  **npm install –g appium-doctor**  2.Once installation is done, run the command below to check all the pre-requisites are installed properly for iOS:  **appium-doctor –ios**  3.If you find some of the pre-requisites are failed, kindly use the commands to install the failed requirements |
| **Download Appium GUI** | 1.If you want to use Appium Desktop GUI for appium sever, download from the official Appium site <https://appium.io/>  2.If you are facing any error while installing Appium like “Appium is blocked by the System” perform the command below and open the Appium GUI:  **xattr -cr <path to appium app>** |
| **Download Appium-Inspector GUI** | 1.if you want to use Appium-Inspectore Desktop GUI for inspecting locators, download from the official git hub site:  [**https://github.com/appium/appium-inspector**](https://github.com/appium/appium-inspector)  2.If you are facing any error while installing Appium like “Appium is blocked by the System” perform the command below and open the Appium GUI:  **xattr -cr <path to appium-inspector app>** |

**Links for installation:**

Devkit portal: <https://devkit.shell.com/content/tools/devrights>

Appium official site: <https://appium.io/>

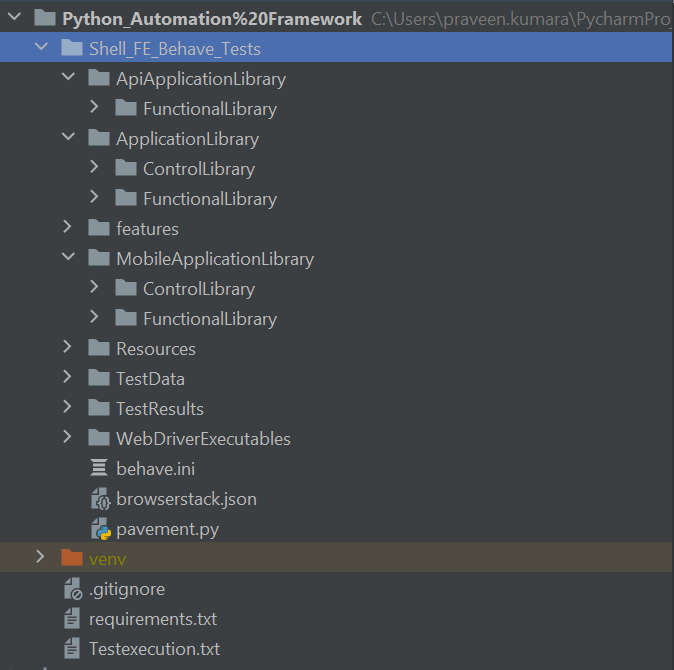
Appium-Inspector: <https://github.com/appium/appium-inspector/releases/tag/v2021.12.2>

Android studio download link: <https://devkit.shell.com/content/tools/androidstudio>

XCode Link:

<https://prodds.mdm.shell.com/Catalog/ViewCatalog/Nwbdpokgx8LD6QKSrwLS2pHHDiH3QRXq6lDjFd2hodGA3vuC7JctLlMNeEaMHNAD/AppleOsX?legacyRedirect=true>

## FRAMEWORK FOLDER STRUCTURE



The above is the folder structure of the Shell FE Python automation framework. Please find the detailed explanation of the folder structure below.

ApiApplicationLibrary:

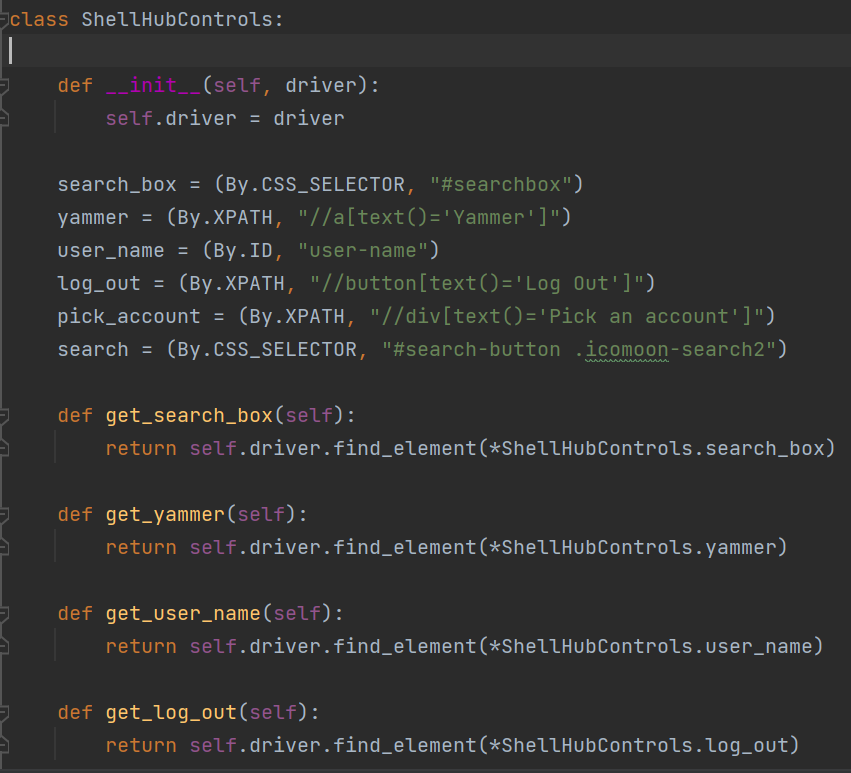
* This folder implements singleton pattern for API tests by storing the methods related to a specific API in an individual Python class

ApplicationLibrary:

* This folder implements POM pattern for the UI automation tests by storing the relevant Page related locators and functions. It consists of the below two sub folders.

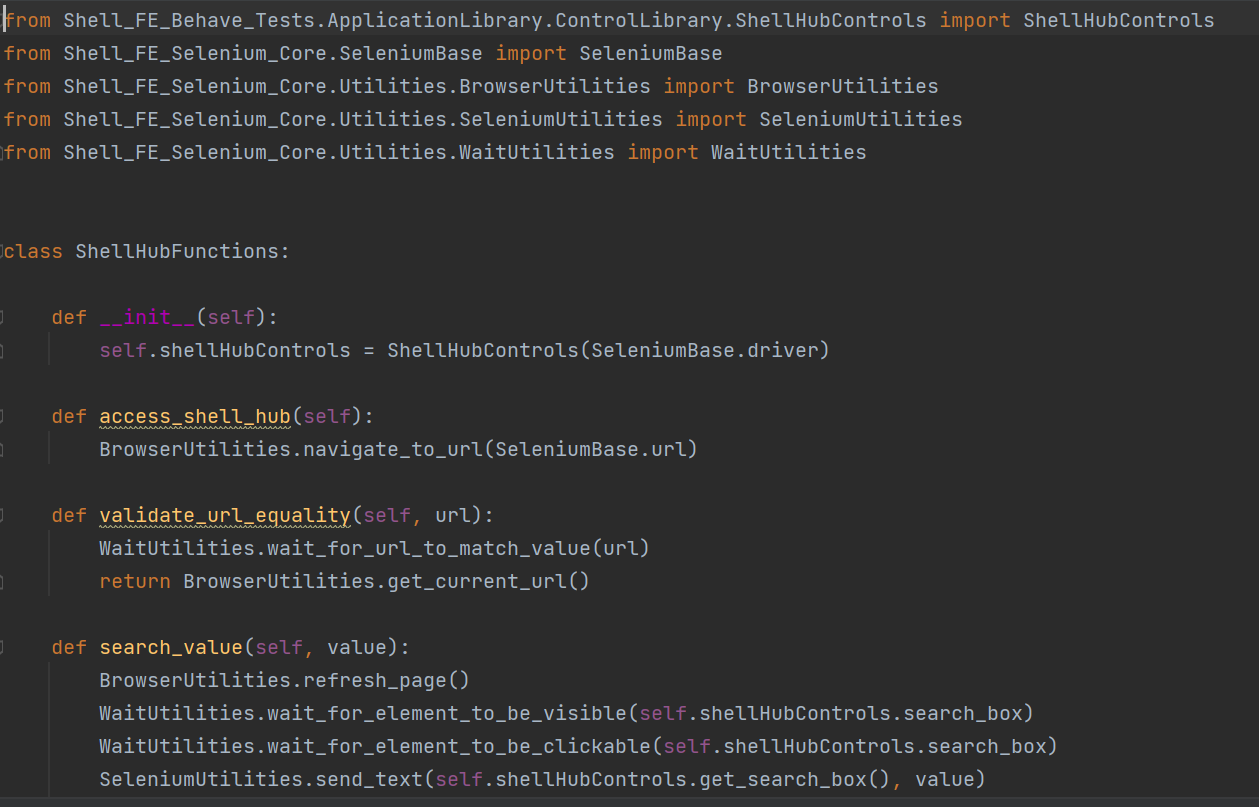
ControlLibrary:

* This folder contains the individual Python classes for each web pages to store locators for the elements present in the relevant page. The driver variable is stored as an instance variable and would be initialized via the constructor.
* Two implementations need to be done for each locator (i.e.) as first step each locator needs to be declared as a tuple containing the “By“ locator type & relevant value and the second step involves creating a method which returns a Web Element by unpacking the relevant tuple
* The class can be named by suffixing the corresponding web page with ‘Controls’ keyword (i.e.) ShellHubControls (where ShellHub is the page name)



FunctionalLibrary:

* This folder contains the individual Python classes for each web pages to store method for business actions relevant to the corresponding page. In the constructor, an object for the relevant page locator class should be created passing the SeleniumBase.driver (which contains the browser related webdriver) as an argument for the instance.
* All business actions specific to a web page needs to be created as methods by calling the corresponding Utilities from Selenium Core excluding assertions. Assertions should be performed only in the Step definition files
* The class can be named by suffixing the corresponding web page with ‘Functions’ keyword (i.e.) ShellHubFunctions (where ShellHub is the page name)



Features:

* The features folder contains the feature files
* The hooks file should be created with the name ‘environment.py’. The code that needs to be executed before or after test run / feature / scenario / step can be written here
* The features folder would contain a sub folder named ‘steps’ which contains step definition files with implementations for feature files
* In the Step definition files, the user needs to call the methods written in the corresponding page function libraries. In addition to that Assertions can be done in the Step definition files by calling the common assertion methods available via AssertionUtilities present in Selenium core
* To ensure sharing of data in Step definition files between different steps in a scenario or feature file it is advised to use the context keyword available in Behave (i.e.) to create an instance for a Page functions class which needs to be shared between the steps in a scenario / feature the user needs to create the instance with the context keyword. If the object needs to be shared within feature file then user needs to use the context.feature keyword.
* It’s advisable to create all reusable variables against the context keyword in the first step implementation of the feature or create a dedicated step for the initializations.



MobileApplicationLibrary:

* This folder implements POM pattern for the Mobile automation tests by storing the relevant Page related locators and functions. It consists of the below two sub folders. Its function and behavior is similar to the Application library except that Application library is used for UI automation and Mobile Application library is used for Mobile automation

ControlLibrary:

* This folder contains the individual Python classes for each app pages to store locators for the elements present in the relevant page. The driver variable is stored as an instance variable and would be initialized via the constructor.
* Two implementations need to be done for each locator (i.e.) as first step each locator needs to be declared as a tuple containing the “By” locator type & relevant value and the second step involves creating a method which returns a Web Element by unpacking the relevant tuple

The class can be named by suffixing the corresponding web page with ‘Controls’ keyword (i.e.) ShellHubControls (where ShellHub is the page name)

FunctionalLibrary:

* This folder contains the individual Python classes for each app pages to store method for business actions relevant to the corresponding page. In the constructor, an object for the relevant page locator class should be created passing the SeleniumBase.driver (which contains the browser related webdriver) as an argument for the instance.
* All business actions specific to a web page needs to be created as methods by calling the corresponding Utilities from Selenium Core excluding assertions. Assertions should be performed only in the Step definition files
* The class can be named by suffixing the corresponding web page with ‘Functions’ keyword (i.e.) ShellHubFunctions (where ShellHub is the page name)

Resources:

* The Resources folder should contain the mobile app APK which is to be tested

TestData:

* This folder is used to store the test data that needs to be parameterized into the tests
* This framework supports features to read and write data from Excel, CSV, XML and Json files
* All the test data files from which data needs to be read or wrote should be present only in the TestData folder
* Also when new files are created when writing data then those files would also be created under the TestData folder

TestResults:

* This folder contains the outcomes / artifacts of the test automation execution such as Logs, Screenshots, AllureJson data, Allure reports, Json reports, XML reports. It contains the below folders:

Logs:

* Log files would be saved into the Logs folder. The log files would be suffixed with timestamp in the format of DDYYHHSS(DateYearHoursSeconds)

Screenshots:

* All the screenshots taken using the available methods in Selenium core would be stored in the Screenshots folder in PNG format

JSONReports:

* When the user executes the command for generating JSON reports, a folder named JSONReports would be created, and the reports would be saved in this folder. The command would be found in the TestExecution.txt file

XMLReports:

* When the user executes the command for generating XML reports, a folder named XMLReports would be created and the reports would be saved in this folder. The command would be found in the TestExecution.txt file

AllureJson:

* The AllureJson folder contains the data required for generating Allure report in the form of JSON. It also contains the failed test scenario screenshots which would be embedded into xthe Allure HTML report

Reports:

* The Reports folder contains the metadata needed to generate Allure HTML report

TestExecution.txt:

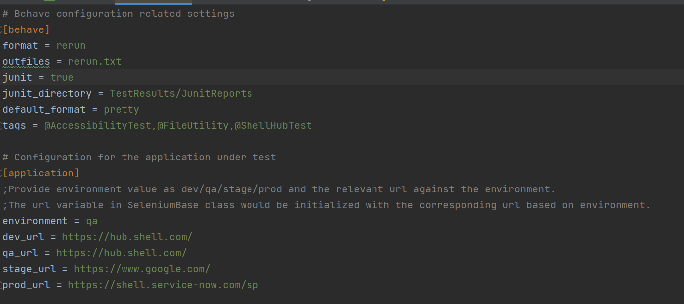
* This file contains details regarding the commands for executing the tests and generating different reports

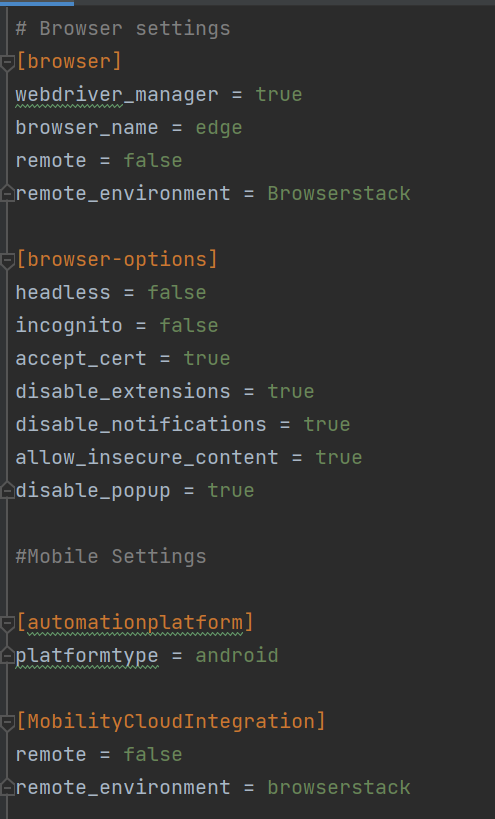
Requirements.txt:

* This contains the PIP package dependencies that is required for this framework

Behave.INI:

* This file contains the configuration values for the framework for UI, Mobile and API.
* The values would be separated based on sections provided within []
* Relevant variables are present in the SeleniumBase class (apart from the values in ‘behave’ section) to which the values in INI file are initialized are used for setting up the needed configuration
* If in case the user needs to add new values to INI file then the user may do so; for reading the values available in INI file the user may make use of the read\_config() method from SeleniumBase which returns an instance of ConfigParser
* Using this instance the user may read values from INI file using the section name and key name
* The ‘behave’ section in INI files contain configuration for Behave framework via which the user may specify the tags to run (if multiple tags are present then they should be separated by comma), tags to exclude( by mentioning the tags to exclude in a new line using a ~ symbol), can generate Junit reports and rerun the failed test cases (by setting the format to ‘rerun’. The failed feature file details would be generated in the path provided in ‘outfiles’. The user may then execute this file containing failed features)







## TEST EXECUTION FLOW

In this section, the flow of test execution would be explained.

1. User needs to initiate execution using the below command (the below command includes code for generating Allure json data as well. User can tweak the command for executing specific feature files / tag names):

behave --no-capture -f allure\_behave.formatter:AllureFormatter -o TestResults/AllureJson/ features

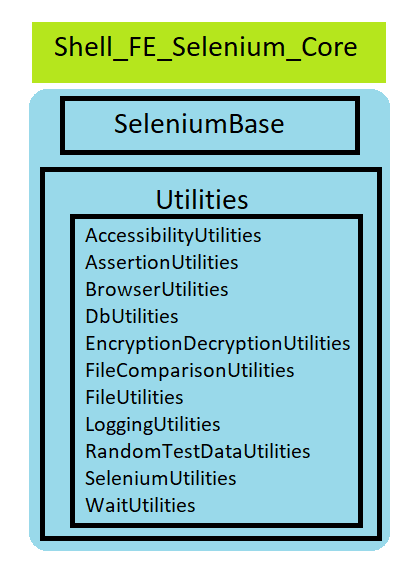
1. All the required values for execution would be provided by the user and stored in INI file (i.e.) the test environment, url, browser, browser options, Db value, mobile device details etc.
2. The test execution starts from the hooks file (i.e.) environment.py. In the before\_all() hook, the initialize\_values() method from SeleniumBase (for UI automation) and the start\_appium\_server() & read\_values() methods from AppiumBase (for Mobile automation), initialize\_values() method from RequestsBase (for API automation) would be called. This will initialize values from INI file to relevant variables in SeleniumBase, AppiumBase, RequestsBase and also start the Appium server.
3. Based on the tags or feature files marked for execution, the before\_feature() hook would be executed. Here the user can specify different steps based on UI / Mobile (i.e.) for features with "web" tag, the browser\_initialization() from SeleniumBase would be called whereas for the features with "mobile" tag launch\_application() method would be called. These methods would initialze the driver variable from SeleniumBase with the relevant browser drivers based on the values in INI file (for UI) and initialize the driver variable from AppiumBase with relevant configurations based on values in INI file (for Mobile).
4. After this the corresponding features would be picked up for execution where the framework would search for relevant step implementations in the steps folder and the corresponding methods from FunctionalLibraries would be called and so on.
5. If in case any failures occur in test execution a screenshot of would be taken and saved under the TestResults/Screenshots folder with the image being named using the scenario name and timestamp. This is achieved using the after\_step() hook in environment.py. The take\_screenshot() method from SeleniumBase / AppiumBase would be called depending on UI / Mobile execution.
6. Also, in case of test execution failure, then the failed step screenshot would be attached to the Allure report. This is handled by the allure.attach() code written in the after\_scenario() hook in environment.py.
7. Once all the test execution has been done, the webdriver and the Appium service can be terminated. This is done by calling the SeleniumBase.dispose() / AppiumBase.stop\_appium\_server() methods in the after\_all() hook present in environment.py file.
8. Finally the user can generate the Allure HTML report by executing the below command:

allure generate TestResults/AllureJson --clean -o TestResults/Reports

## CORE FEATURES

## SELENIUM CORE

The structure of the Selenium Core package is detailed below.



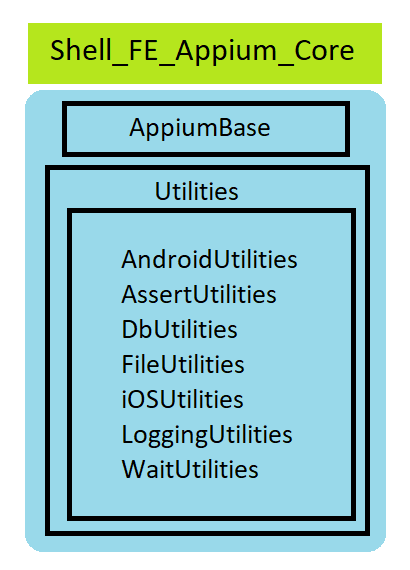
The SeleniumBase and all other utilities are modules of the Shell\_FE\_Selenium\_Core package which can be reused across the framework.

The details of the method available in SeleniumBase and the other utilities can be found in the below embedded excel file.



## APPIUM CORE

The structure of the Appium Core package is detailed below.



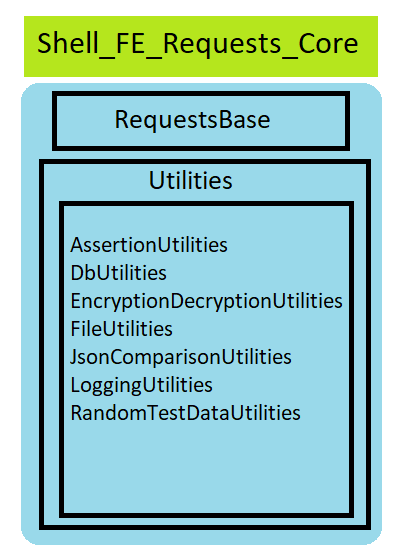
The AppiumBase and all other utilities are modules of the Shell\_FE\_Appium\_Core package which can be reused across the framework.

The details of the method available in AppiumBase and the other utilities can be found in the below embedded excel file.



## REQUESTS CORE

The structure of the Requests Core package is detailed below.



The RequestsBase and all other utilities are modules of the Shell\_FE\_Requests\_Core package which can be reused across the framework.

The details of the method available in RequestsBase and the other utilities\ can be found in the below embedded excel file.



## REPORTING

The framework uses Allure reporting for representation of Test results. Allure report provides the test results in a neat format segregating tests based on status such as Passed / Failed / Broken / Skipped and represents data in form of graphs. It also segregates test results based on Severity status in the form of a graph, user can view the history of the test case execution *(copy the contents of TestResults/Reports/history into TestResults/AllureJson/Reports after report has been generated to get the accurate historical test execution results)*, failed step error message and the screenshot for the failed test.

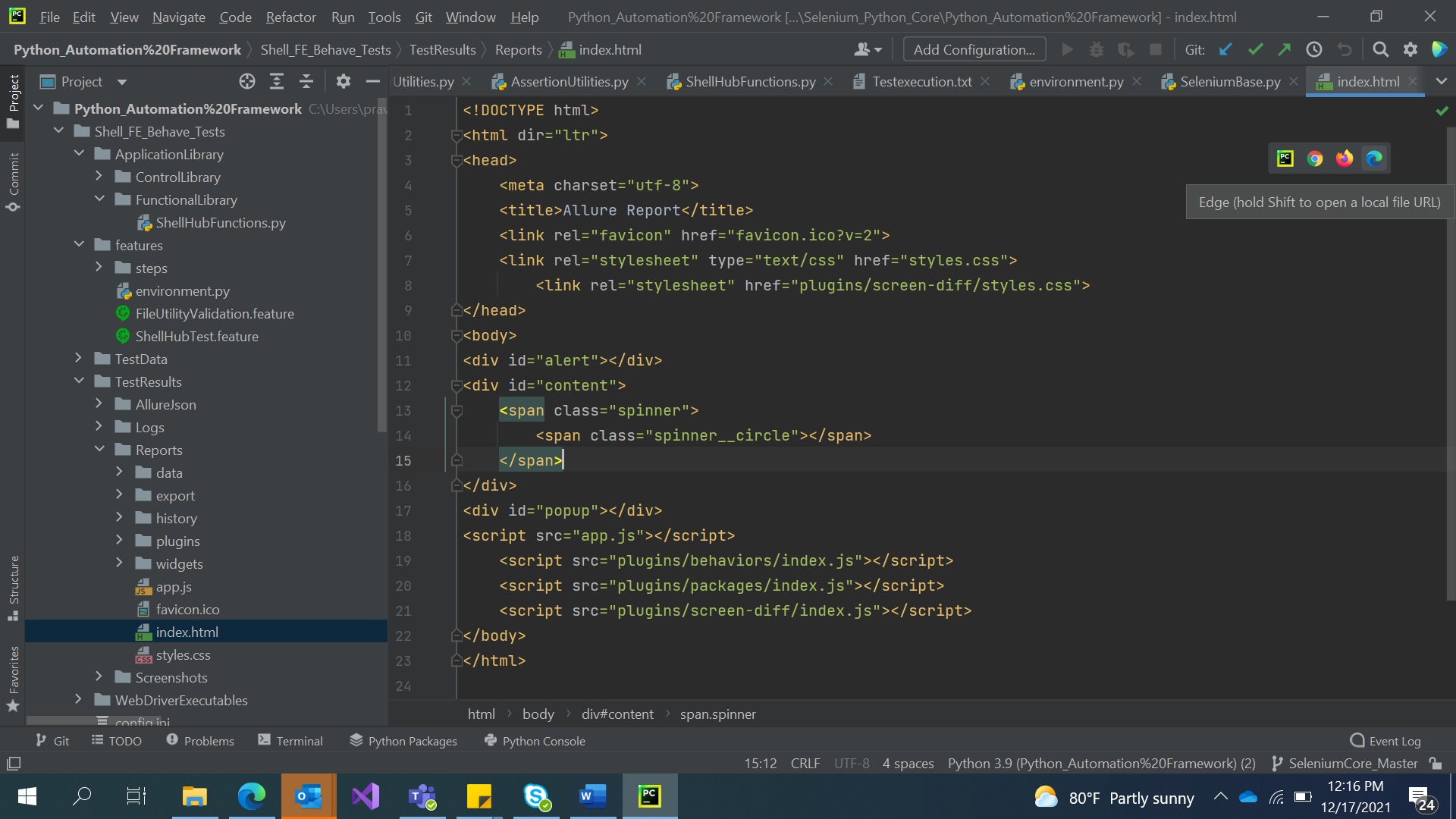
The steps for installing Allure has been detailed in the Pre-requisites section and the folders where Allure report and data are stored has been detailed in the Framework folder structure section.

The Allure HTML report would be saved as index.html file. User can view the report by opening the html file via IDE and clicking on any suitable browser icon.

An alternate way for opening the report is to execute the following command: allure open TestResults/Reports.

User can update the data in the following files located under AllureJson based on their project needs

* Configuration.json
* Executor.json
* Environment.properties

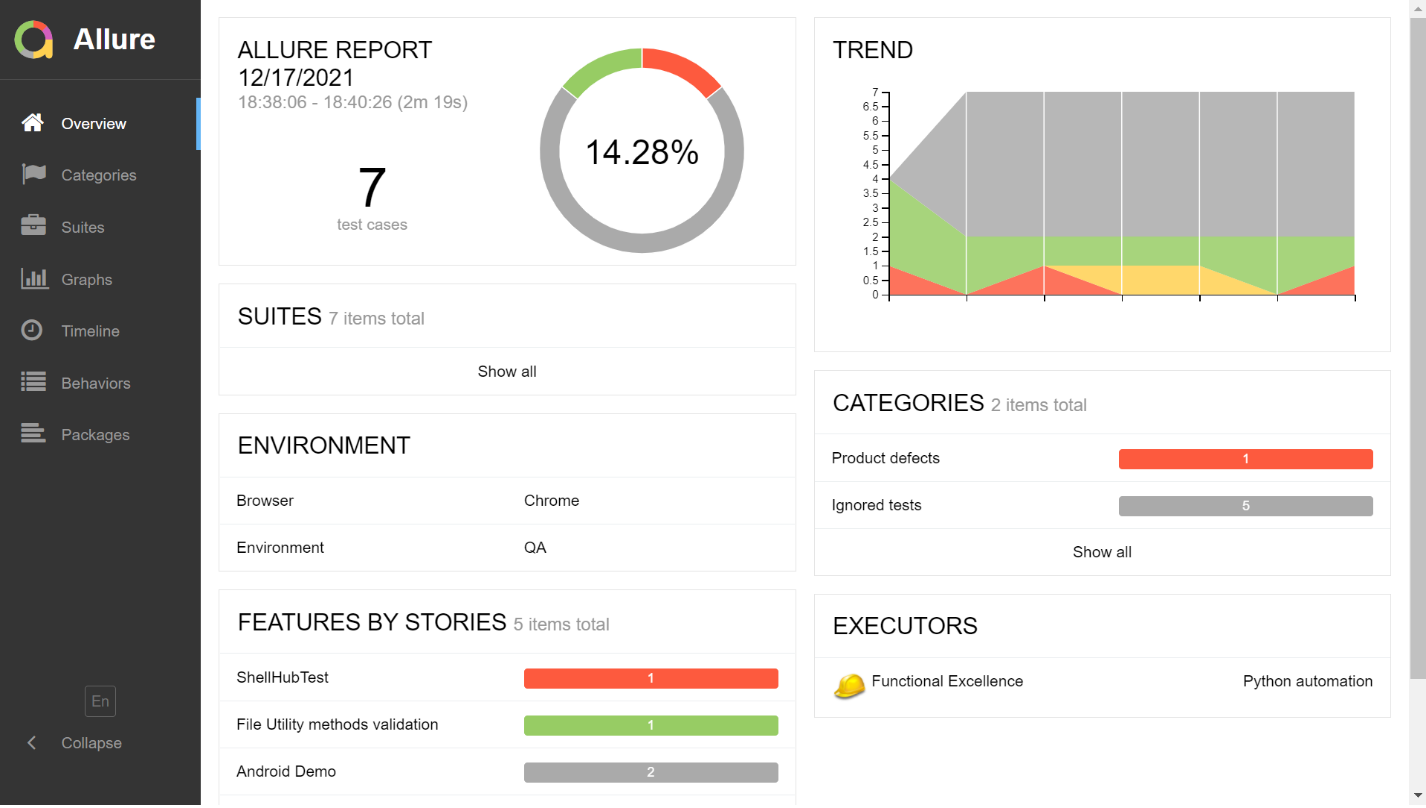


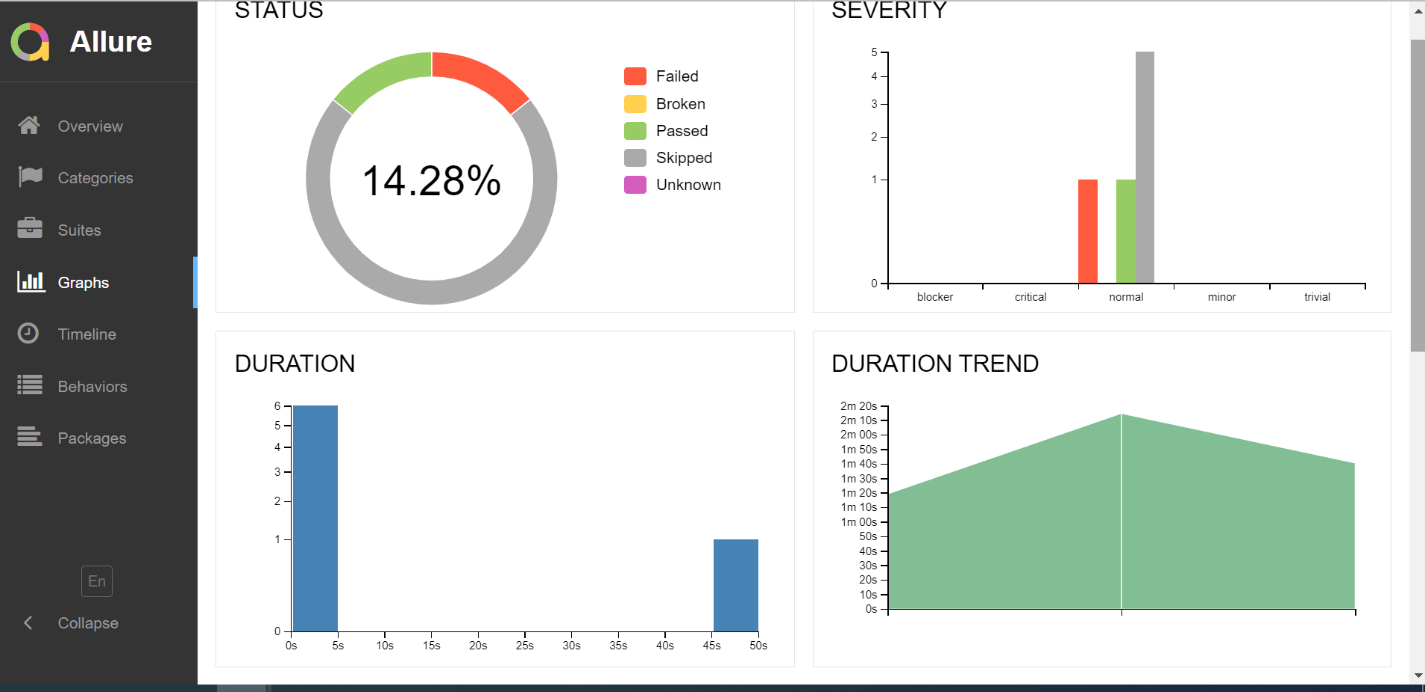
For those who wish to explore more on Allure reporting, check out the below link.

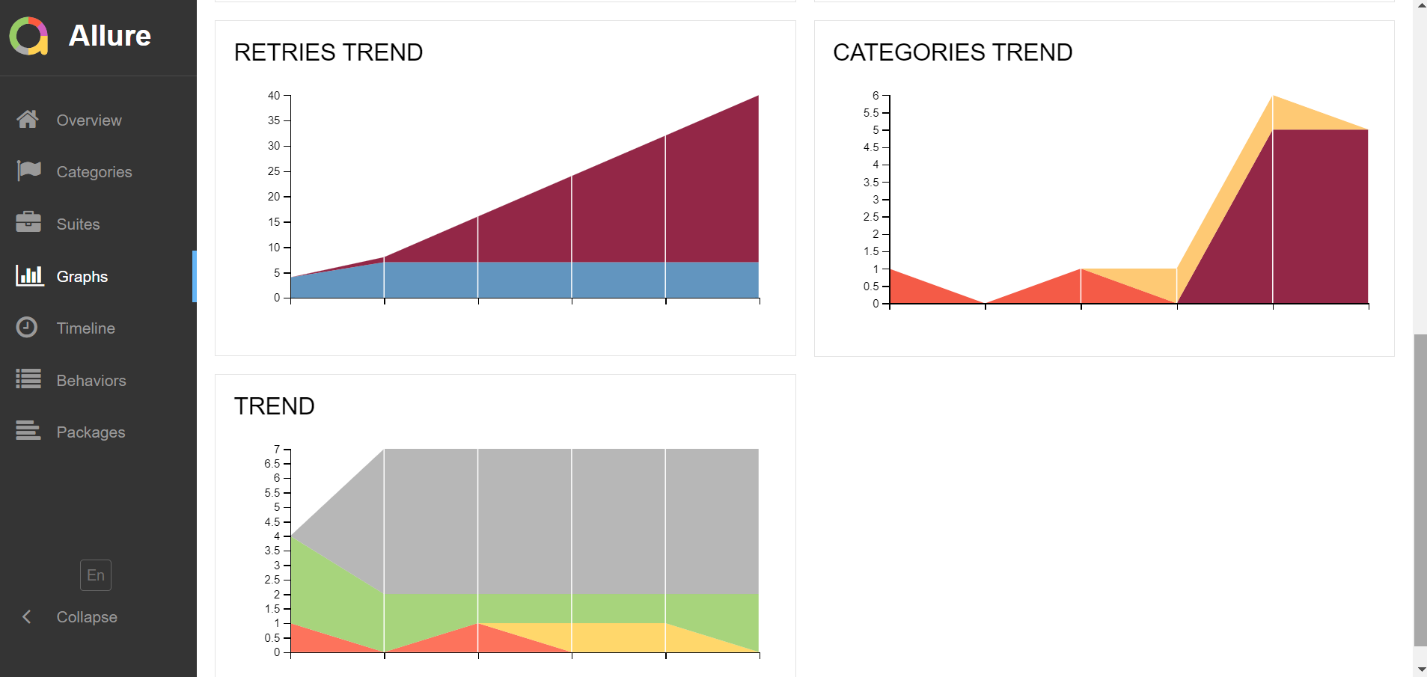
[Allure Framework (qameta.io)](https://docs.qameta.io/allure/#_categories)

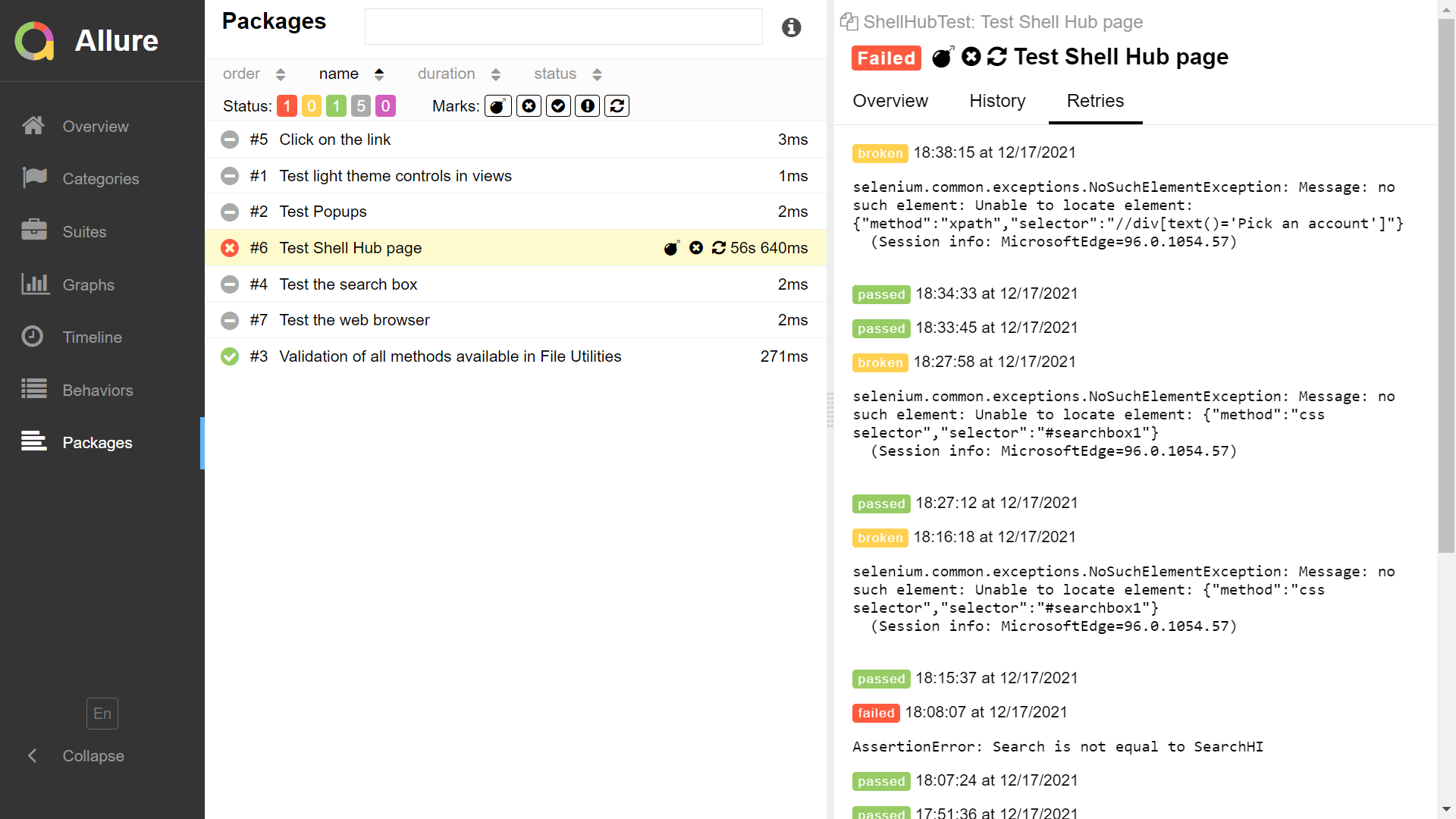
The framework also provides options to generate JSON and HTML reports. The reports would be saved in respective folders created under TestResults.

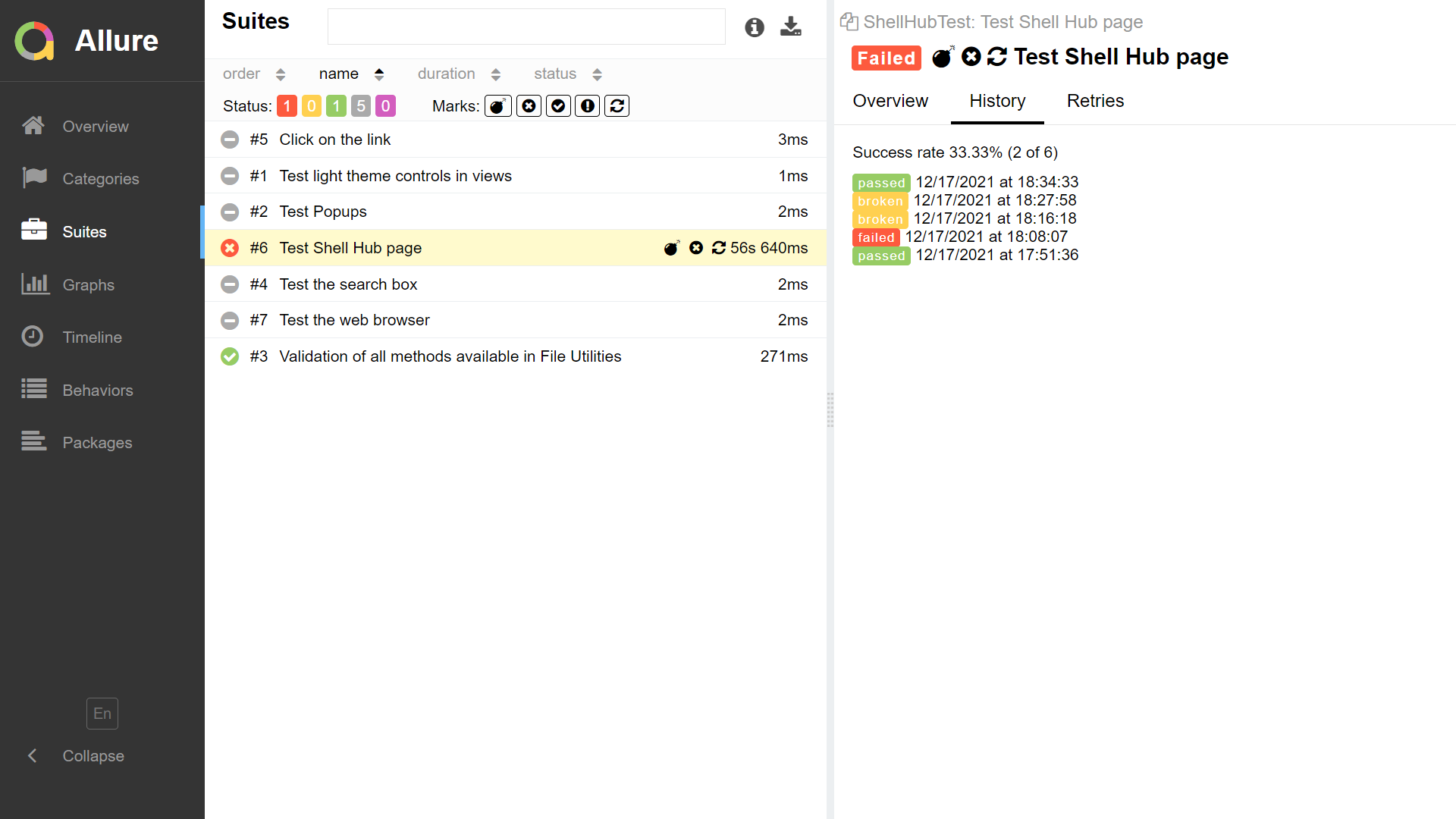
Sample Report:

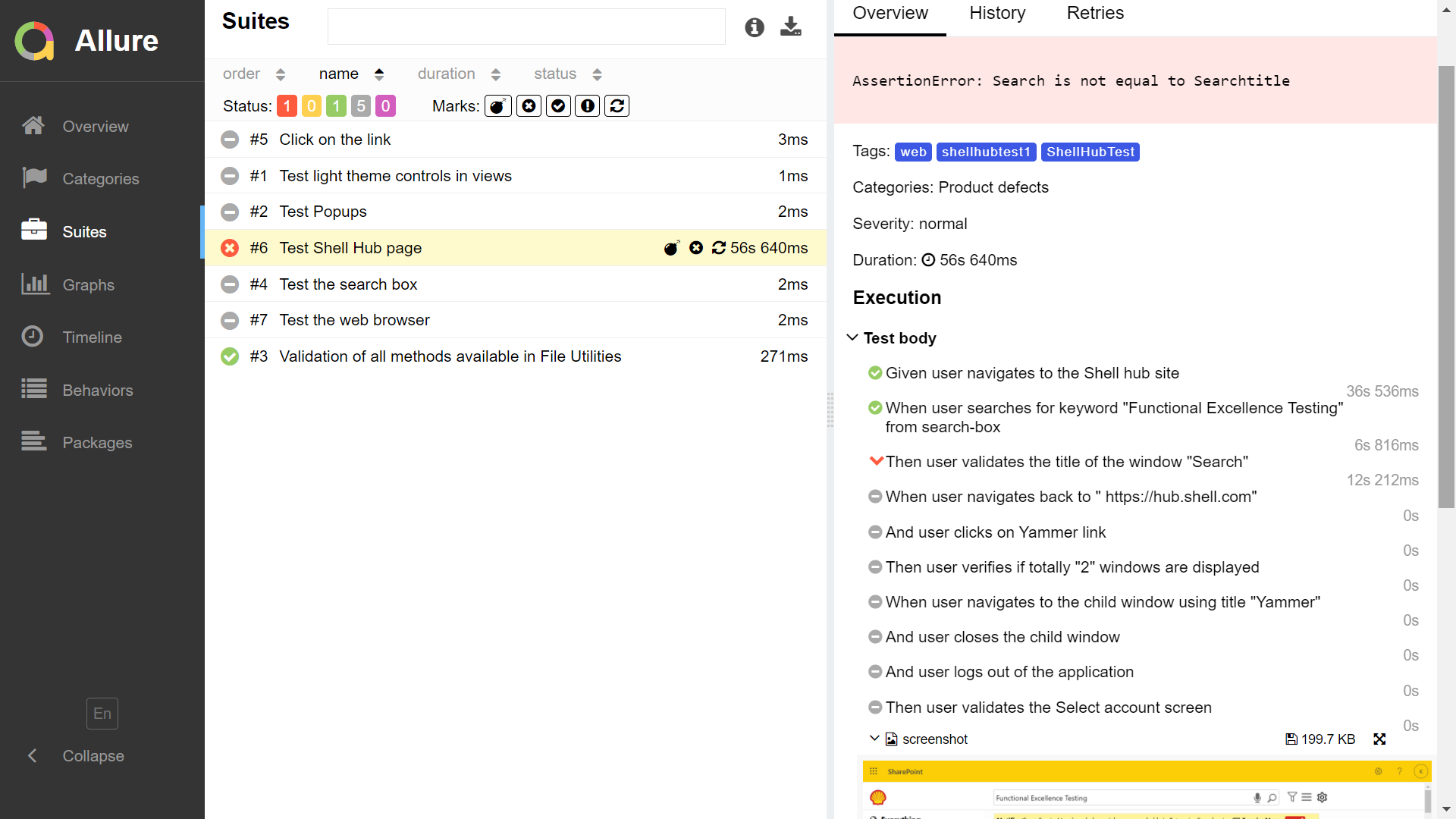












## ACCESSIBILITY TESTING

The framework provides a feature for performing accessibility testing on the web page. Accessibility testing has been implemented only in UI automation. Accessibility testing in Selenium has been implemented using the axe-selenium-python library.

The method is available in the AccessibilityUtilities under Selenium Core package named as ‘analyze\_page()’ and is created as a static method. This method will generate a json report for the web accessibility test containing the accessibility rules that has been passed and also the violations that do not match the accessibility rules spelled out by WCAG in that current web page.

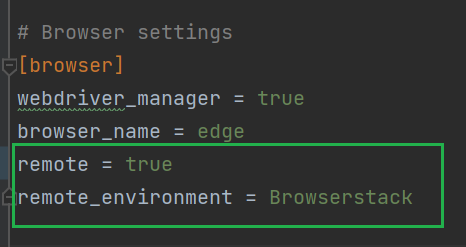
The analyze\_page() method will return the number of violations present in accessibility report and the user can assert the case based on the need.7

## CLOUD TOOLS INTEGRATION

## BROWSERSTACK INTEGRATION

The framework allows integration with BrowserStack for both Web and Mobile automation. There are two files which is mandatory for BrowserStack integration viz. browserstack.json (the BrowserStack account related details, environment / device details shall be mentioned here) and pavement.py (this is used for parallel multi device testing for both web and mobile automation)

Before starting the execution, the user needs to change the corresponding values in INI file. For Web automation, the user needs to set the below values in ’browser’ section of the behave.ini file.



For mobile automation, the user needs to set the below values in ‘’ section of the behave.ini file.

Once the above step is done, the user needs to update the browserstack.json file with the relevant details.

The user name, key and the server details can be found in the BrowserStack account and those details should be entered against the corresponding keys.

If using web application automation integration with BrowserStack then values for the following keys should be entered.

* webServer: The server details which can be found in BrowserStack
* webEnvironments: This is an array of json objects where the user can specify the environment details (desktop or mobile devices) where the tests need to be executed. The capabilities for each system can be mentioned as part of the device details.

If using mobile app automation integration with BrowserStack then values for the following keys should be entered.

* appServer: The server details which can be found in BrowserStack
* appCapabilities: The build name and options / capabilities for the test to be run
* appEnvironments: This is an array of json objects where the user can specify the environment details (mobile devices) where the tests need to be executed

**BrowserStack single device execution:**

* After providing relevant values in the INI file and browserstack.json file the user can execute his tests using the usual commands found in TestExecution.txt.
* The first device mentioned in the ‘webEnvironments’ / ‘appEnvironments’ will be taken for execution. The tests being executed can be viewed in the dashboard section of the browserstack site after logging in as the relevant user.

**BrowserStack multi device parallel execution:**

* After providing relevant values in the INI file and browserstack.json file, the user can execute his tests using the below command

**For Web Application automation using Selenium:** paver browserstack\_parallel\_web

**For Mobile app automation using Appium:** paver browserstack\_parallel\_app

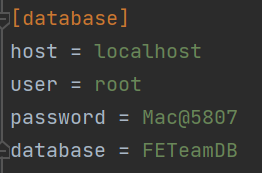
The above commands would execute the methods available in pavement.py. The users are required not to edit / change any lines of code from pavement.py file.

For the test results to be updated in BrowserStack based on the assertions in Tests, add the below code in after\_feature() of environment.py

if context.failed is True:  
 SeleniumBase.driver.execute\_script(  
 'browserstack\_executor: {"action": "setSessionStatus", "arguments": {"status":"failed", "reason": "At '  
 'least 1 assertion failed"}}')  
if context.failed is not True:  
 SeleniumBase.driver.execute\_script(  
 'browserstack\_executor: {"action": "setSessionStatus", "arguments": {"status":"passed", "reason": "All '  
 'assertions passed"}}')

## DB INTEGRATION

The framework allows integration of test scripts with mysql database. For integration with database, server should be downloaded in the machine. Before execution the user has to define the host, server, username and password of the database in the behave.ini file. The methods in the DBUtilities will help to establish the connection between the server and script. The framework is designed in such a way that if user wants to perform any query they can establish the connection with mysql server and then they can perform the query.



## FILE COMPARISON

The framework provides options to compare files (Excel, Image) and provide the results. These file comparison functionalities are available in Web application automation alone.

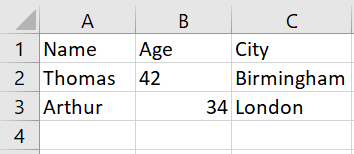
## EXCEL COMPARISON

- The framework supports comparison of two excel files. This is achieved using the method compare\_excel available under FileComparisonUtilities. Using this method the user can compare two excel files and a Boolean value (True/False) is returned based on the comparison. The method returns True if the files are equal and returns False if the files are not equal. Also, if the files are not equal a new excel file would be created containing the difference at the cell level.

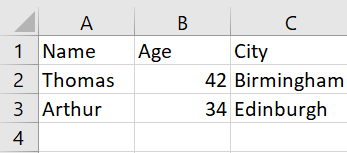
- The files to be compared should be present in the folder TestData/ExcelComparisons

- The comparison excel file would be generated into the TestData/ExcelComparisons with the name “ExcelDifference” suffixed with the current timestamp

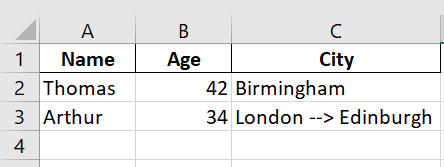
- The files to be compared should have the same number of cells



**File1.xlsx**

****

**File2.xlsx**



**ExcelComparison24031247.xlsx**

## IMAGE COMPARISON

The framework supports comparison of image files. The type of images supported are ‘.svg’ which is generally used for logo and ‘.png’ which is generally used while taking screenshot. The image comparison is done using the ‘compare\_image’ method which is available under FileComparisonUtilities. The image difference is calculated using the numpy so the difference will be a float value. The tolerance is taken as ‘1’ and any difference greater than ‘1’ will not be considered as same image and the return result will be in Boolean(True/False). “compare\_screenshot” will download the screenshot of the current page and then call the compare method to compare with local image.

## c. PDF PARSING

The framework provides support to search for words in the pdf . This is achieved by the method “search\_word\_from\_pdf” which is available under “FileUtilities.py”. The method will search for the string /substring (defined by the user) and returns the result (Boolean value) to confirm whether the string/substring is present in the PDF file. If the user is not aware of the page number in which the text is present then user can use “get\_number\_of\_pages” method to return the number of pages from pdf and then this function can also be used in test cases and used to run the loop for all the pages after getting the page count.

## VISUAL TESTING

The framework supports Visual testing by taking a screenshot of the current webpage and comparing it against an existing screenshot. This is achieved using the ‘compare\_screenshot’ method available in FileComparisonUtilities. This method will download the screenshot of the current page and then call the compare method to compare the screenshot with an existing image. The image difference is calculated using numpy so the difference will be a float value. The tolerance is taken as ‘1’ and any difference greater than ‘1’ will not be considered as same image and the return result will be in Boolean(True/False).

Visual testing is available only for Web application automation.

## RANDOM TEST DATA GENERATION

The framework provides a functionality to generate random test data which can be used as test data in the test scripts. This feature to generate random test data is available under the RandomTestDataUtilities. It contains methods which can generate random data viz. name, address, date, time, email, phone number, id, string, password etc.

This utility is available in Web and API automation core.

## ENCRYPTION & DECRYPTION

The framework supports Encryption and Decryption functionality via the methods available under EncryptionDecryptionUtilities. Initially the user needs to call the “encrypt\_message” method available in EncryptionDecryptionUtilities to encrypt the user message which will create a ‘secret.key’ which should be used whenever the user needs to encrypt or decrypt the message. The method ‘encrypt\_user\_creds’ will create a file ‘creds.json’ and then the user can call the ‘encrypt\_message’ method to encrypt user password. To decrypt user credentials the ‘decrypt\_user\_creds’ method should be used which will call “decrypt\_message” method to decrypt user password using the same ‘secret.key’ which was generated the first time while encrypting the user creds. This utility is available in Web and API automation core.

## AUTO RERUN

The framework supports auto rerun of failed scenarios. This is achieved by adding the below code in before\_feature() block in environment.py file. The number of rerun attempts for failed scenarios can be customized by changing the value in ‘max\_attempts’.

for scenario in feature.scenarios:  
 patch\_scenario\_with\_autoretry(scenario, max\_attempts=2)

## FEATURES AVAILABLE

|  |
| --- |
| **WEB APPLICATION AUTOMATION - SELENIUM** |
| BDD Integration using Behave |
| Custom utilities for common user actions, wait, assertions |
| Multi browser support (Chrome, Edge, Firefox, Safari) |
| Allure reporting along with separate logs and screenshots |
| Data parameterization (supports file read and write options from Excel, Json and read options for CSV, XML) |
| Core features provided as installable python packages |
| Build and Release pipeline implementation via Azure Devops |
| XML and Json report creation supported which can be used for CI/CD |
| Rerun failed test cases |
| Accessibility Testing |
| Browser initiation via Webdriver manager as well as browser binaries |
| Parallel execution |
| Cloud tools integration (Browser stack) |
| Random test data generation utility |
| DB integration |
| Visual testing automation |
| Image comparison |
| Excel comparison |
| PDF parsing |

|  |
| --- |
| **MOBILE APP AUTOMATION - APPIUM** |
| BDD Integration using Behave |
| Custom utilities for common user actions, wait, assertions |
| Supports Android and iOS automation |
| Supports Native app, Hybrid app, mobile browser execution (Chrome and Safari) |
| Allure reporting along with separate logs and screenshots |
| Data parameterization (supports file read and write options from Excel, Json and read options for CSV, XML) |
| Core features provided as installable python packages |
| Build pipeline implementation via Azure Devops |
| XML and Json report creation supported which can be used for CI/CD |
| Rerun failed test cases |
| Cloud tools integration (Browser stack) |
| DB integration |

|  |
| --- |
| **API AUTOMATION - REQUESTS** |
| BDD Integration using Behave |
| Reusable methods for common HTTP requests viz. GET, POST, PUT, DELETE, PATCH |
| Reusable methods for parsing the response from request (i.e.) getting response status code, body, headers etc. |
|  |
| Supports basic HTTP and Oauth2 authentication |
| Custom utilities for Assertions, Json Comparisons |
| Data parameterization (supports file read and write options from Excel, Json and read options for CSV, XML) |
| Core features provided as installable python packages |
| Build and Release pipeline implementation via Azure Devops |
| XML and Json report creation supported which can be used for CI/CD |
| Rerun failed test cases |
| Allure reporting along with separate logs and screenshots |
| Random test data generation utility |
| DB integration |