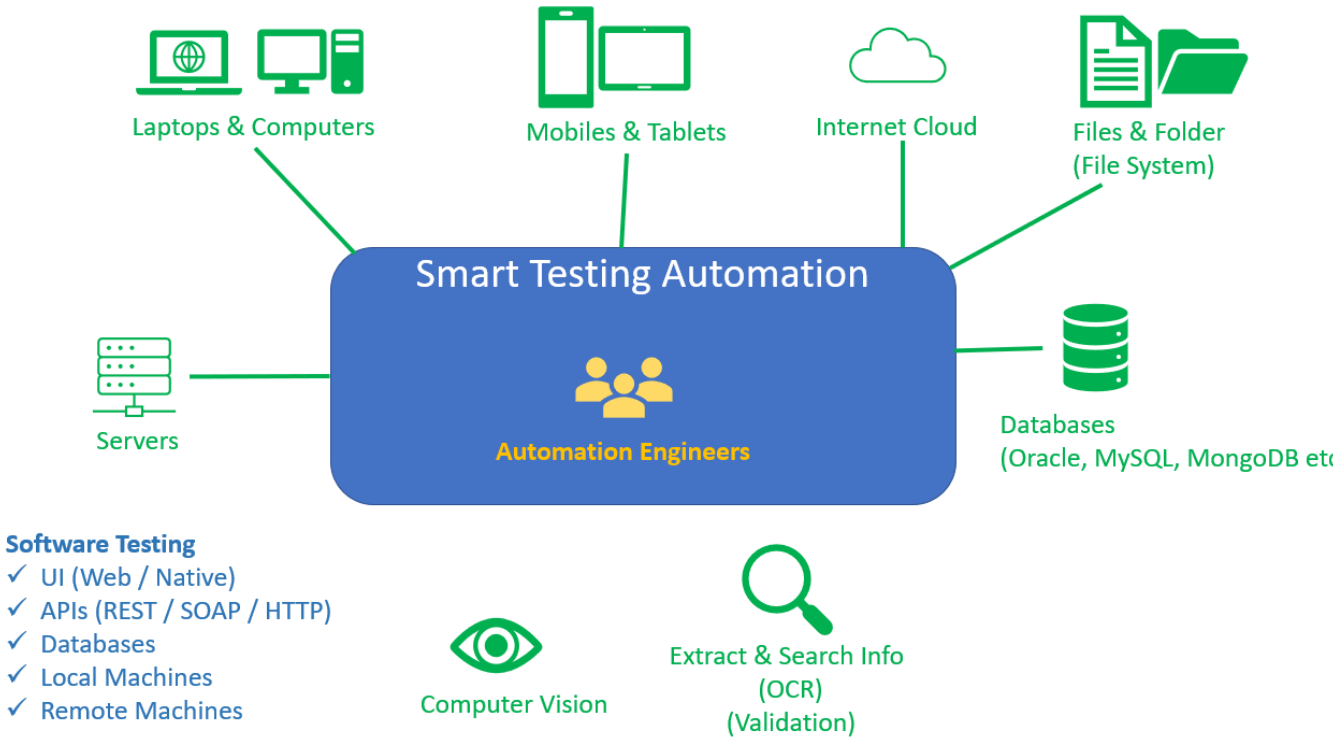


SMART TESTAUTO STUDIO (STAS)

Test Automation Developer's Guide

Abstract

This document describes the functionality of STAS tool and provides details of how to use STAS tool for software application automation.

Madhav Krishna
mkrishnacs20@gmail.com

Table of Contents

Table of Figures

1 Licensing and usage

This is developer's guide developed by the actual developer of Smart Test Automation Framework (STAF) / Smart TestAuto Studio tool (STAS). The purpose of this developer's guide is to help test automation engineers to setup their testing automation environment and develop the scripts for the testing automation to automate the following types of testing: Regression Testing, Sanity Testing, Smoke Testing, End-to-End Testing, Functionality Testing, User Acceptance Testing etc.

This is a free and open-source tool and licensed under **Apache License 2.0 (https://www.apache.org/licenses/LICENSE-2.0)**. This tool and the guide only used to help develop testing automation and there is no liability on the author if there is any defect found in the system. But users can raise the defects on the Github URL (https://github.com/mkrishna4u/smart-testauto-framework/issues) so the community team can work on that defect and close as per their convenience. Also, if you are looking for any change or enhancement you can raise that on the same Github URL.

Also, the licensing of 3rd party tools integrated in this tool belong to the individual owner of the 3rd party tool. If there is any defect on third party tool, the defect should be raised on the respective 3rd party tool website.

2 Introduction

Using STAS, Data driven End-to-End testing (UI, API, Database and Remote Machine Testing) is made easy. Develop testcases for one environment and run the same testcases on different environments.

NOTE: STAS is not a record and play based software testing automation tool. We have to manually write the scenario in Cucumber feature file, because STAS project main object is to validate the data and write the test scenarios even when the system is not developed / deployed. Test engineer can use the requirement document and mockup screens as input to develop the test scenarios.

STAS tool is a tool used for faster testcase development using the standard **Cucumber Gherkin** language. Using this tool, you can perform the different types of functional testing automation and group the testcases / test scenarios as we write. The following software testing automation can be performed:

A. Graphical User Interface (GUI) Testing
B. API Testing
C. Database Testing
D. Functional Testing
E. End-to-End Scenario Testing / Integration Testing (may include multiple applications and multiple sub systems)
F. Regression Testing
G. Smoke Testing
H. Cross applications testing
I. File contents verification testing i.e. PDF, MS Word, MS Excel, MS PowerPoint, Images etc.
J. File upload and download
K. OCR (Optical Character Recognition) Testing
L. Computer Vision / Image pattern matching

The following **approaches** are supported by STAS to write the test cases easily and manage them:

A. Data Driven Testing (DDT) Approach

Use different type of data file to feed the data to the system to verify the functionality. State of the art easy to use classes are provided to read the data from different types of files as given below:

1. ExcelFileReader
2. CSVFileReader
3. JsonDocumentReader
4. XmlDocumentReader
5. YamlDocumentReader
6. SmartDatabaseManager / SQLDatabaseActionHandler

B. Behavior Driven Testing (BDT) Approach

Use Cucumber Feature file to write the End-to-End scenarios and to provide the data to the system to verify the functionality.

C. **Configuration Driven Testing (CDT) Approach**
STAS provides standards configuration files in YAML format to configure your application related information (i.e. application config, user profile config, web driver config, database config, remote machine config etc.) that can be used to communicate with UI, REST Servers, Database Server and Remote Machines and also that can be used to validate the data on UI, REST Services, Database and Remote Machines.

STAS is a very powerful software testing automation tool. It is used to configure multiple software applications into it to perform testing automation across the applications to perform the End-to-End testing in a standard way. Standard way means it does not matter what platform (Windows, Linux, iOS, macOS etc.) or what application we use to perform testing automation. The way the testcases / test-scenarios are going to get written will be same and standardized, so that all test engineers can write the test-scenarios in similar fashion to reduce the complexity of the test-scenarios from the maintenance perspective. It uses standardized **Cucumber Gherkin** language to write the test scenarios. Standardized Cucumber Gherkin Step definitions are available on Github project:

- https://github.com/mkrishna4u/smart-testauto-cucumber-stepdefs-en

STAS provides **LOW CODE / NO CODE** model, it means test engineers have to write low code or no code based on the scenario description. But they have to write scenarios in Cucumber Gherkin language as per the standardized step definitions provided by this tool.

This tool supports the **real environment** for software testing (Similar way our manual tester performs software testing like data preparation, run test cases (data-driven), data verification and generate reports etc.). Here using this tool, we can automate data preparation, test cases execution, data verification and report generation easily.

Smart Testing Automation Framework / STAS tool connects with different types of applications (Web, Native, Mobile), servers (Database, file systems), remote machines (over SSH).

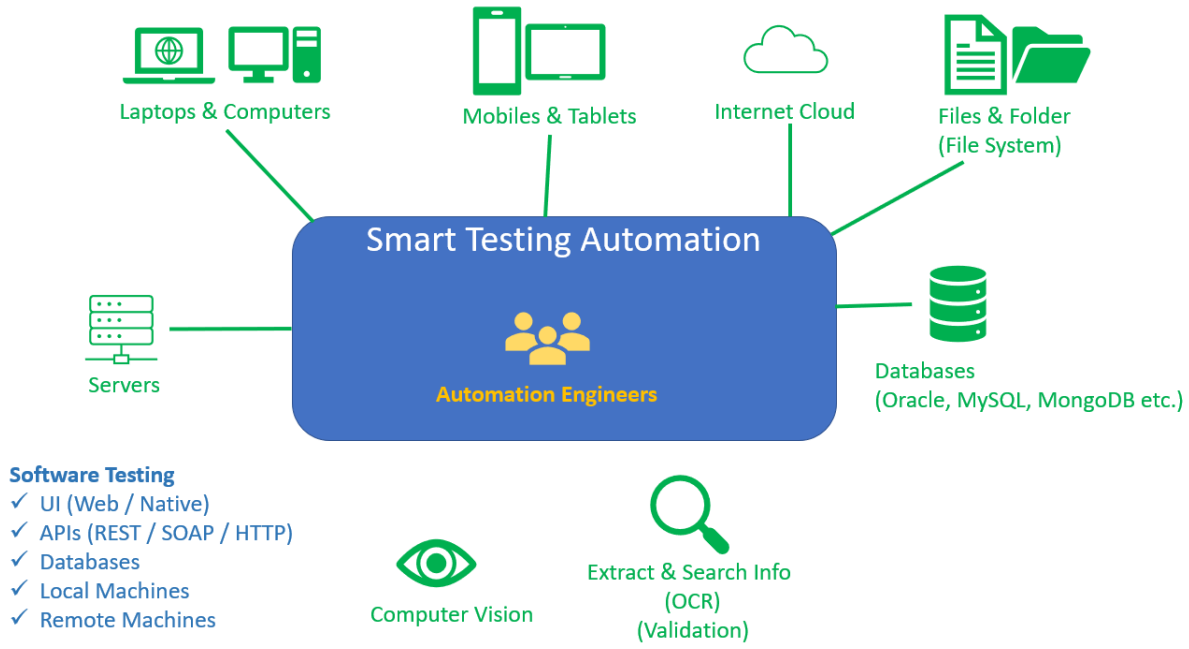


Figure 2-1 STAS Tool Connectivity

2.1 What can be automated using STAS?

Using this software tool, we can easily test the following functionality of the software applications:

1. **Web User Interface** running on different web browsers (like Chrome, Firefox etc.) on different platforms (like windows, mobile, mac, linux, android, iOS etc.)
2. **Native applications** like **Desktop applications** (like calculator), **Mobile applications** (like calculator)
3. **Visualization testing / Image recognition testing** (like image based testing using *SI object classes like ImageSI, ButtonSI, TextboxSI object classes etc.). NOTE: To perform visualization testing, application must be launched and visible on PC. Visualization testing cannot be performed on headless web browsers.
4. Perform **Data Driven Testing (DDT)** on user interface against your relational database i.e. Oracle, MySQL, Postgres, MariaDB etc. Use **SmartDatabaseManager** class to perform CRUD operations on database. **SqlDatabaseActionHandler** can be registered with database profiles to perform CRUD operations on relational/SQL databases.
5. Perform **Data Driven Testing (DDT)** using Excel, CSV data, JSON, YAML and XML data files. Reads excel (.xlsx and .xls) files data using **ExcelFileReader** class. Reads CSV file data using **CSVFileReader** class.
6. Perform **Behavior Driven Testing(BDT)** using Cucumber Gherkin feature files.
7. **API Testing (REST API)** using **SmartApiTestManager AbstractApiActionHandler** class. This class maintains sessions and supports customizable login(..) and logout() APIs. Create your own class that extends AbstractApiActionHandler class and use it to

perform HTTP operations like GET, POST, PUT, DELETE etc. These api action handler class can be registered with ApiConfig.yaml file.

8. Read JSON data using **JsonDocumentReader**. It uses the JSON Path information available on https://github.com/json-path/JsonPath
9. Validate JSON data using **JsonDocumentValidator** class. It uses the JSON Path information available on https://github.com/json-path/JsonPath
10. Read YAML data using **YamlDocumentReader**. It uses the JSON Path information available on https://github.com/json-path/JsonPath
11. Validate YAML data using **YamlDocumentValidator** class. It uses the JSON Path information available on https://github.com/json-path/JsonPath
12. Read XML data using **XmlDocumentReader**. It uses the XML Path information available on https://www.w3.org/TR/1999/REC-xpath-19991116/
13. Validate XML data using **XmlDocumentValidator** class. It uses the XML Path information available on https://www.w3.org/TR/1999/REC-xpath-19991116/
14. **Configure multiple applications** in a single project and write test scenarios that involves communication among multiple applications and automate test steps in a simple way by adding less code.
15. Support standard way of configuration for applications, application user profiles and application relational database profiles.
16. Write scenarios and its definitions once for any platform and run the same scenarios and definitions on any other platforms for any of the software application without changing the scenario and the step definition (Note the behavior of the application on different platforms should be same but locator may change). In this case you can attach multiple locators (platform specific) to a single UI control (like textbox, buttons etc.).
17. Use **SmartCucumberScenarioContext** cucumber CDI (Constructor Dependency Injection) class for UI, API and Database test step definition.
18. Platform specific sample driver configuration files (AppDriver.yaml) are present in the following directory **sample-config/apps-drivers**. Copy specific file in your application config directory like **test-config/apps-config/[app-name]/driver-configs** and also copy your application file like ?.app, ?.apk ?.exe, or ?.api etc.
19. Use **TestDataBuilder** class to build randomized test data of any length that may include alphabets, numbers, special characters, newline, whitespaces, leading characters etc.
20. Use **FieldValidator** class to validate the field value as per the expected value or criteria. Also **StringUtil** class is very handy to check the textual / string data.
21. Use **HttResponseValidator** as part of API **HttpResponse** to validate the HTTP response. You can validate the contents of the files like any file .docx, .xlsx, .pptx, .pdf, .gif, .jpeg, .jpg, .png etc.. Many file formats are supported. Also you can validate the HTTP response payload using different validators like **JsonDocumentValidator**, **YamlDocumentValidator**, **XmlDocumentValidator** etc..
22. Use **DownloadedFileValidator** class to validate the downloaded file (downloaded by web browser). If you want to validate the contents of any file (like .docx, .xlsx, .pptx, .pdf, .gif, .jpeg, .jpg, .png etc..) use **FileContentsValidator** class.
23. Use **SmartRemoteMachineManager** class is used to talk to remote machines / servers using SSH to perform uploaded file verifications, execute remote command or download remote file etc.

24. **Multiple environment support**, means same testcases can be executed on different environments like **development, acceptance, pre-production and different web browsers etc.** without changing the code. To do so create environment files in **test-config/apps-config/[app-name]/environments/** directory and also create the different **AppDriver-[env-name].yaml** (if using different app driver), different database profiles in **database-profiles/** (if different database connection required) and different **ApiConfig-[env-name].yaml** in **api-configs/** (if different parameters are used for different environment). During testcase execution time, specify the following system property in **mvn** command **"-Dapps.active.environment=[app-name1]:[environment-name],[app-name2]:[environment-name]"** to activate the environment for the applications configured in project.

This is also a **Configuration Driven Testing (CDT)** framework. Sample configurations are present here that can be used under **./test-config** directory to configuration your project environment: https://github.com/mkrishna4u/smart-testauto-framework/tree/main/src/main/resources/sample-config**.**

NOTE: STAS also provides the default standardized Cucumber Step Definitions that can be used out of the box to automate any scenario or prepare scenario outline.

For document on STAS standardized **Cucumber Step Definitions** please refer this link: https://github.com/mkrishna4u/smart-testauto-cucumber-defaults-en/tree/main/docs/STAS-StepDefinitions-En-TestDevelopersGuide.pdf

2.2 Integrated software testing tools (High Level)?

STAS has integration with the following 3rd party and open-source tools to perform the software testing automation:

1. Cucumber
2. Selenium
3. SikuliX
4. Appium (For mobile testing)
5. TestNG
6. Maven
7. Relational Database ORM tools
8. Excel / CSV / JSON / YAML file reader tools
9. Java / JDK: Minimum Version Required = 11
10. Secured Shell (SSH/SFTP)
11. Tesseract-OCR (Ref Link: https://tesseract-ocr.github.io/tessdoc/) - Must be installed separately and environment PATH variable must have the path of this installed directory to perform image, audio and video file content matching. **FileContentsValidator** class will only work along with Tesseract-OCR.

This framework removes the complexity of all other software tools and provides a **Single Homogeneous Platform** for testing automation. Using that you can automate any app on any platform.

2.3 Supported Platform Types

The following platforms are supported to perform the software testing automation:

1. windows
2. linux
3. mac
4. android-mobile
5. ios-mobile

2.4 Supported Application Types

The following application types are supported to perform the software testing automation:

1. **native-app**: Like Desktop applications (like calculator) on any platform like windows, mac, android, iOS etc.
2. **web-app**: Like applications running on web browsers (like Github UI) on any platform like windows, mac, android, iOS etc.

2.5 Supported Web Browsers

The following web browsers are supported to perform the software testing automation:

1. Chrome
2. Firefox
3. Edge
4. Opera
5. Safari
6. Internet Explorer
7. Custom Webdriver (Remote Web driver)
8. Not Applicable: This is set for native applications.

2.6 Data Validator classes

There are many data validator classes that can be used to validate the data. These classes are given below:

FieldValidator	DownloadedFileValidator
HTTPResponseValidator	FileContentsValidator
JsonDocumentValidator	UI Field Validators
XmlDocumentValidator	StringUtil
YamlDocumentValidator	

2.7 What you need to know to work on this tool?

There are many tools integrated in this tool to make the test engineers life easy. The most import things that every test engineer must know are:

1. **Cucumber Gherkin Language:** Study about it on the following URL: https://cucumber.io/docs/gherkin/reference/
2. **JSON and JSON Path:** JSON is a very powerful data format that is used while writing cucumber scenario using STAS tool. For JSON path, please refer https://github.com/json-path/JsonPath link. JSON Path is used to modify the JSON data or retrieve field data from JSON data.
3. **Basic Knowledge of Java:** Basic knowledge of Java programming is needed to create the page object classes. It only require, how to create class object using parameterized constructor or how to call class methods. Other basic knowledge of Java programming will be useful if you are planning to write your customized cucumber step definitions.
4. **Different type of data files:** If you are writing data driven test scenarios that requires to read data from Excel, CSV, TXT, XML, JSON, YAML etc. file then you should know the format of these data files. This tool provide ready to use step definition to read the data from these types of files.
5. **XML and XPATH:** Knowledge of XML file contents are mandatory to work on the user interface and web services (API testing) that uses XML data format. Using XPATH mechanism you can access any element in XML document or modify the contents of XML document. XPATH references: https://www.w3.org/TR/1999/REC-xpath-19991116/
6. **HTML and XPATH:** For user interface testing automation, STAS tool uses Selenium / Appium internally to perform operation on page object elements like Textbox, Button etc. You can use different type of locators to locate element on the user interface like ID, AutomationID, AccessibilityID, Name, CSS Selector, LinkText, XPATH etc. By default STAS tool uses XPATH mechanism to locate element on user interface as XPATH mechanism is much solid and we can identify any element on user interface. XPATH references: https://www.w3.org/TR/1999/REC-xpath-19991116/
7. **Basic knowledge of Shell Scripting:** Since STAS uses command line to run the test scenarios. So, knowledge of how to run shell script (windows or linux) is required to run the maven commands or STAS provided scripts using command line.
8. **YAML file:** All the configuration in STAS tool is given in YAML format. It is very simple and standard format to specify the configuration. To know the YAML file format, you can refer any online document.

3 STAS Architecture

STAS architecture is generic and it can be integrated with any CI/CD tools like Jenkins, Gitlab etc.

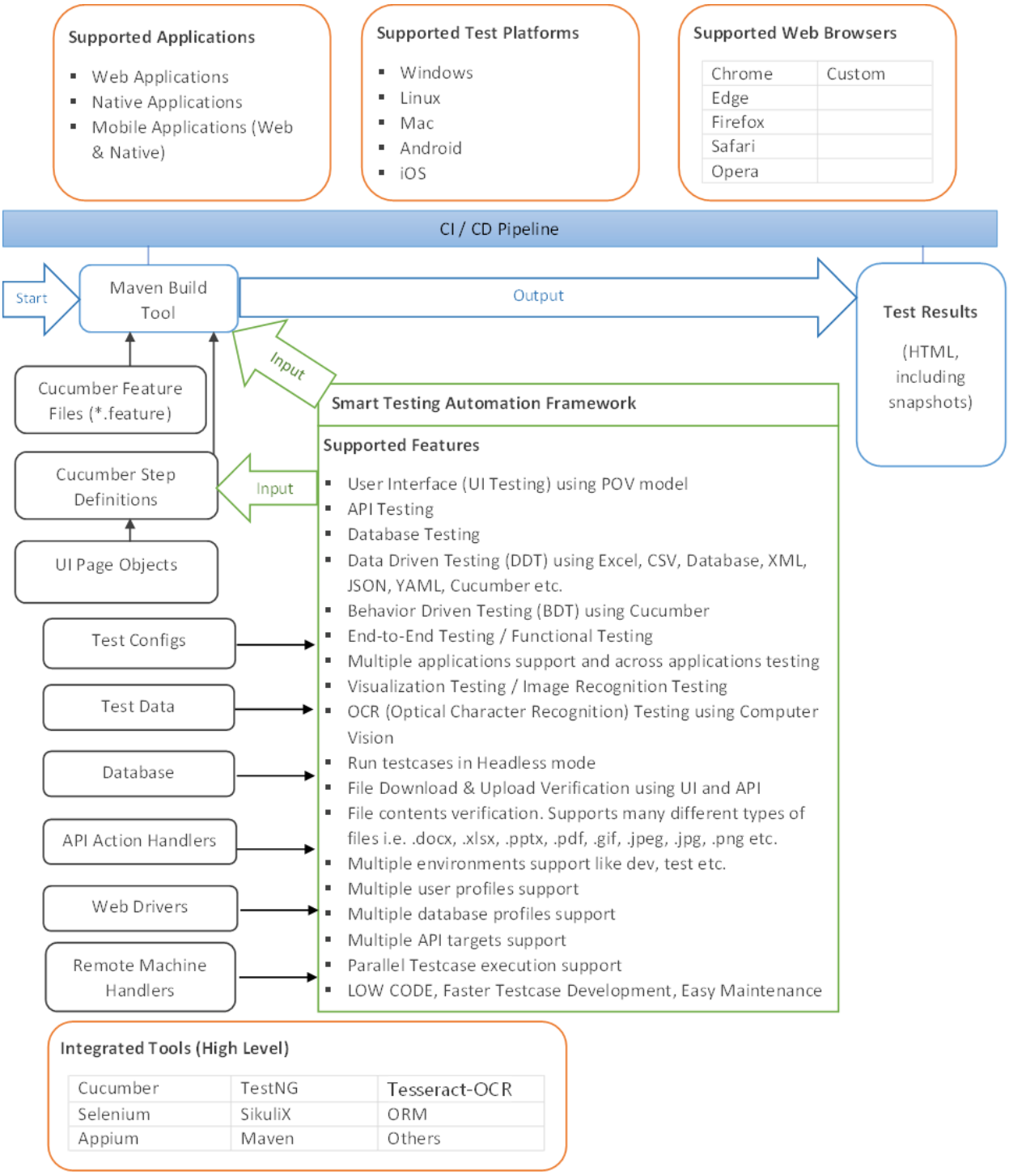


Figure 3-1 STAS Architecture

This tool provides state of the art generic configuration for testing automation. So that test engineers can organize their test configuration, test data, test scenarios and step definitions better way. This tool uses standard YAML based configuration.

You can run test scenarios in parallel using command line to expedite the test execution. Since this tool support command line, so it can be integrated with any CI/CD tool like Jenkins, Gitlab etc.

4 Setup Project Environment

STAS provides the command line interface to setup project environment on your laptop or desktop or any server machine.

4.1 Configure first application as part of project initialization

To configure the first application as part of project initialization, use the following steps:

1. Create a project directory. Let's say "**Smart-TestAuto-Studio**" directory is created at some location of your choice.
2. Download the following files from the Github path (for the latest available version): https://github.com/mkrishna4u/smart-testauto-framework/tree/main/smart-testauto-studio/latest and put these files into **./Smart-TestAuto-Studio** directory:
    - **pom.xml**
    - **smart-studio.sh**
    - **smart-studio.cmd**
    - **README.md**
    - **LICENSE**
3. Open command prompt / Console / Terminal and change directory to **<your-base-path>/Smart-TestAuto-Studio** (your project directory)
    - Make sure Maven tool and JDK 11 or later is installed on your machine and system **M2_HOME**, **PATH, JAVA_HOME** variables are configured properly to run **javac** and **mvn** commands from console window.
    - Run the following command to create the project setup for your first application:
        - **smart-studio --init <your-app-name>**
          > This command will create the full maven project environment using first app. This will generate many directories to manage the full project.
4. Now you are all set with the maven project. Import this maven project in your favorite Java Code Editor like Eclipse or IntellijIDEA etc. to work on this project. Make sure **Cucumber plugin** is installed on your code editor to write test scenarios using Gherkin language. Also you can convert your project as Cucumber project on code editor for Step Auto Completion in Cucumber feature file.
5. Update the configuration file as per your application need that are present in **test-config/** directory.

4.2 Smart Studio command line options

Once your first application is configured then run the following command to see what are the command line options are present to configure STAS project:

> **smart-studio**

After typing command hit enter key to get the results. It will show all the options like:

Options are:

--init <app-name>

Description: Used to initialize the project with new application name. If application is not configured then it will configure the application also.

--add-app <app-name>

Description: Used to add new app when it does not exist.

--add-env <environment-name>

Description: Used to add new environment for the already configured applications. Environment file is created under test-config/apps-config/<app-name>/environments directory.

--prepare-app-driver-config <app-name>

Description: Used to prepare driver config for the application based on the application AppConfig.yaml file details.

--prepare-app-driver-config-for-env <app-name> <env-name>

Description: Used to prepare driver config for the application environment based on the application AppConfig-<env-name>.yaml file details.

--update-web-driver <platform-type> <web-browser-type> <version>

Description: Used to download the web browser for the specified platform and web browser. Please refer application AppConfig.yaml file for more details on <platform-type> and <web-browser-type>. Or see the directory structure under test-config/app-drivers/ directory to get details on <platform-type> and <web-browser-type>.

--install-appium-server

Description: Used to install appium server for mobile testing. Note installed nodejs directory path should be configured in PATH system variable so that it can use npm or node command.

--start-appium-server

Description: Used to start appium server on default host and port. Note installed nodejs directory path should be configured in PATH system variable so that it can use npm or node command.

4.3 Add / Configure new application in STAS project

To add / configure a new application in the existing STAS project, use the following steps:

- Open command prompt / Console / Terminal and change directory to **<your-base-path>/Smart-TestAuto-Studio** (your project directory)
  - Run the following command to add / configure new application:
    - **smart-studio --add-app "new-app-name"**
      This command will add / configure new application into the existing STAS project.

      Let's say we are adding **myapp2** as a new application then the following directories will get created for myapp2:

      1. src/main/java/page_objects/**myapp2**: This directory will contain the sample PO (page object) classes that we can modify as per application need.
      2. src/main/java/validators/**myapp2:** This directory will contain the sample validator file that we can modify as per application need.
      3. src/test/java/stepdefs/api/**myapp2:** This directory can be used to add the customized cucumber step definition files for API automation.
      4. src/test/java/stepdefs/ui/**myapp2:** This directory can be used to add the customized cucumber step definition java files for UI automation.
      5. test-config/apps-config/**myapp2**/: This directory will contain all the configuration files for **myapp2** application that we can modify as per the application need like AppConfig.yaml, AppDriver.yaml, remoteMachinesConfig.yaml, database profiles, user profiles, environment files etc.

4.4 Project directory structure

When you import your maven project, you will see the following directory structure:

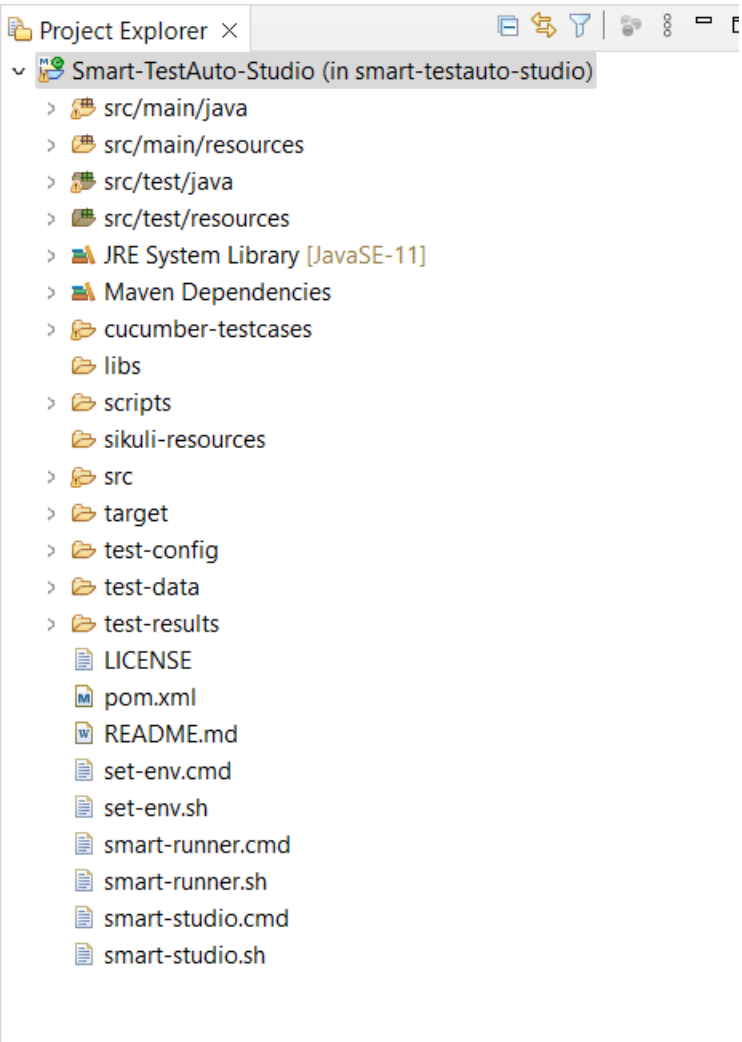


Figure 4-1 STAS Project Directory Structure

Directory and the content type information is given below in successive sections.

4.4.1 Directory Details: cucumber-testcases/

The following directory structure is created under **cucumber-testcases/** directory:

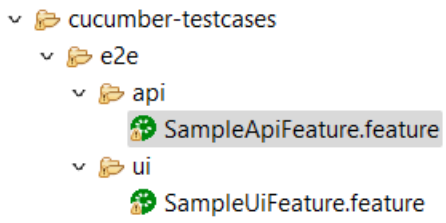


Figure 4-2 Directory Details: cucumber-testcases/

Note:

End-to-End testcases can be defined under **cucumber-testcases/e2e** directory. This directory contains by default two directory api/ and ui/ where you can add your Cucumber feature file and

write your scenario / scenario outline inside these feature files. Also, you can tag (Note: Scenario Tag information is defined in ***Tags supported in STAS Cucumber Feature File to group scenarios*** section) these scenarios or scenario outlines as per your need. You can add directories as per your needs to organize your feature files.

4.4.2 Directory Details: src/main/java

The following directory structure is created under **src/main/java** directory:

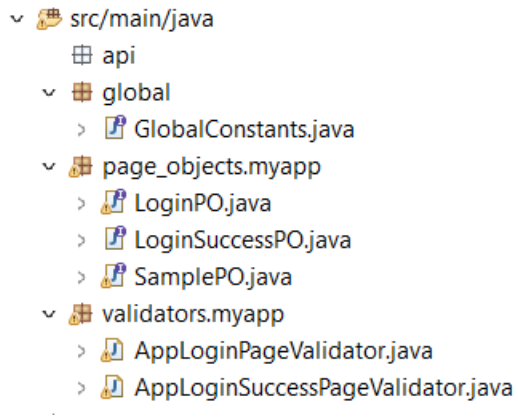


Figure 4-3 Directory Details: src/main/java

Note:

1. **global/** package: It contains the java files that can store the information globally like you can store global constants inside default **GlobalConstants.java** file. That you can use in step definitions files present in **src/test/java/stepdefs/** directory or any other java / class files.
2. **page_objects.myapp** package: In this package you can create page object (PO) classes similar to what the sample classes present in this package. You can use *SD or *SI classes of smart-testauto-framework to create UI page elements and that can be defined in the PO classes. Also update the existing page elements as per your application needs.
3. **validators.myapp** package: This contains default validator classes for the app to validate Login page and login success page (i.e. Home page to Dashboard page). You can add the implementation of the login and logout functionality here in these Validator classes. Some sample code is already there that you can use to create your own code as per your application requirement. If required, you can also create your own validator classes here that you can use in your step definition.

4.4.3 Directory Details: src/test/java

The following directory structure is created under **src/test/java** directory:

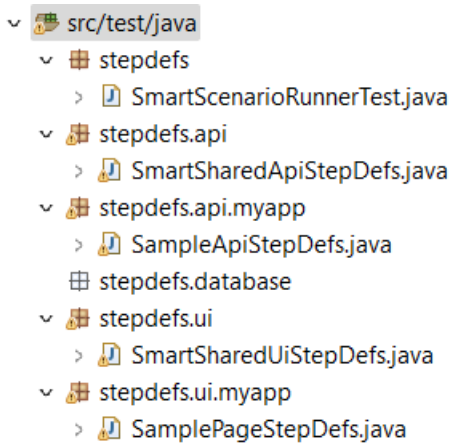


Figure 4-4 Directory Details: src/test/java

Note:

1. **stepdefs** package: This is the main package where you can add the customized cucumber step definitions. By default system creates the following java file that contains the following information:

Java File Name	Description
SmartScenarioRunnerTest.java	This is the main Test Runner class that defines the CucumberOptions to map the Cucumber feature files to step definitions (glue code) files. Generally, it is not required to modify this file until you want to have a very specific need.
SmartSharedApiStepDefs.java	If you would like to add any common API automation related step definition, you can add it here.
SampleApiStepDefs.java	This is a sample API Step Definition file / Template file. If you would like to add your customized API step definitions for your application automation. You can add in this package or the file. You can modify the SampleApiStepDefs.java file name to the meaningful name for that you would like to add step definitions.
SmartSharedUiStepDefs.java	If you would like to add any common UI automation related step definition, you can add it here.
SamplePageStepDefs.java	This is a sample file / template file. If you are willing to add new step definitions then create your PageStepDefs.java file in this package and add the step definitions here.

4.4.4 Directory Details: test-config/

test-config/ is the main config directory where all the STAS project configuration files are stored. The configurations are divided multiple parts like

1. **Test Configuration**: **TestConfig.yaml** is the main Test Configuration file in STAS project. That defines the global configuration parameters.
   Sample **TestConfig.yaml** file:

```
# Comma separated app names that will be used for running test scenarios.
appNames: myapp

# preferDriverScreenshots valid values: true, false
preferDriverScreenshots: true

# embedScreenshotsInTestReport: valid values: true, false
embedScreenshotsInTestReport: true

# useDefaultStepDefsHooks valid values: true. false
useDefaultStepDefsHooks: true

# Additional properties (Define under additionalProps as child property)
additionalProps:
```

2. **Application Configuration**: Application configuration files are defined under **test-config/apps-config/<app-name>/** directory. It contains many configuration files
   a. **AppConfig.yaml**: This is the main file for application configuration. That contains application specific configuration parameters:
      Sample **AppConfig.yaml** file:

```
# Application name
applicationName: myapp

# Application type
# Supported application types: native-app, web-app
applicationType: web-app

# Test Platform Type on which testing is going to be performed
# Supported Test platforms: windows, linux, mac, android-mobile, ios-mobile
testPlatformType: windows

# Application Launch URL. For NativeApp it will contain the path of .app, .api, .apk, .exe etc. application file.
# And for web-app it will contain the web URL that will be used to launch application on web browser.
appLaunchURL:

# Login Validator
appLoginPageValidatorClass: validators.myapp.AppLoginPageValidator

# Login Success Page Validator
appLoginSuccessPageValidatorClass: validators.myapp.AppLoginSuccessPageValidator

# Remote Web Driver Provider class (Applicable only if appWebBrowser = remoteWebDriverProvider)
remoteWebDriverProviderClass:
```

```
# Application Web Browser where all the test cases need to be executed. Applicable only for
'applicationType = web-app'
# Supported Web Browsers: firefox, chrome, edge, opera, safari, internet-explorer, remote-web-
driver-provider, not-applicable
appWebBrowser: chrome
enableBrowserExtensions: false
# Browser window size (width x height)
browserWindowSize: 1024 x 768

# APP Driver fonfig file name
appDriverConfigFileName: AppDriver.yaml

# API Config file name
apiConfigFileName: ApiConfig.yaml

# RemoteMachinesConfig file name
remoteMachinesConfigFileName: RemoteMachinesConfig.yaml

# User profile configuration
userProfileNames: [SampleUserProfile]

# Application database connection setting (Using Hibernate)
dbProfileNames: []

proxySettings:
  # Proxy configuration
  #PROXY Configuration Options:NO_PROXY, AUTO_DETECT, USE_SYSTEM_PROXY,
MANUAL_PROXY
  proxyConfigType: NO_PROXY

  # Parameters for MANUAL_PROXY
  httpProxyHostname: <proxy-host>
  httpProxyPort: <proxy-port>

  sslProxyHostname: <proxy-host>
  sslProxyPort: <proxy-port>

  ftpProxyHostname: <proxy-host>
  ftpProxyPort: <proxy-port>

  socksHostname: <proxy-host>
  socksPort: <proxy-port>

  # SOCKS_VERSION options: V4, V5
  socksVersion: V5
  socksUsername:
  socksPassword:

  noProxyFor: localhost,127.0.0.1

# Additional properties (Define under additionalProps as child property)
additionalProps:
```

b. **api-configs/ApiConfig.yaml**: This is the configuration file to configure target servers to make HTTP connections to perform HTTP operations like GET, POST, PUT etc.

Sample **ApiConfig.yaml** file**:**

```
# configure target servers. these servers will get registered using its unique name.
targetServers:
 - name:
   baseURL:
   # Class name that extends AbstractApiActionHandler class
   actionHandlerClass:
   sessionExpiryDurationInSeconds: 1800
   # Additional properties for target server (Define under additionalProps as child property)
   additionalProps:

# Additional properties (Define under additionalProps as child property)
additionalProps:
```

c. **user-profiles/**: This is the directory where we can add user profiles that can be used to authenticate users on UI Application and API Services. These profiles are configured in **AppConfig.yaml** file. User profiles are saved with the same name as configured in AppConfig.yaml file.

SampleUserProfile.yaml file:

```
profileName: SampleUserProfile

# Application User Credentials
appLoginUserId:
appLoginUserPassword:

# User account information
userAccountType:
userAccountTypeCode:

# Comma separated user roles
userRoles: []

# Additional properties (Define under additionalProps as child property)
additionalProps:
```

d. **remote-machines-configs/**: This is the directory where we can configure remote machines that can be used in Cucumber steps and step definitions to access remote machine information over SSH to validate the information like uploaded file information, remote machine CPU and memory statistics etc. Each remote machine is configured by unique name that can be used in scenario steps.

Sample **RemoteMachinesConfig.yaml** file**:**

```
remoteMachines:
 - name: Sample-Machine
   hostNameOrIpAddress: 194.168.0.100
   port: 22
   platformType: linux
```

```
    # Class name that extends AbstractApiActionHandler class
    actionHandlerClass:
org.uitnet.testing.smartfwk.remote_machine.SmartRemoteMachineActionHandler
    sessionExpiryDurationInSeconds: 3600
    userName: sample-user
    password: sample-user-password
    privateKeyPath: test-config/apps-config/<app-name>/remote-machines-configs/<name-of-file>
    privateKeyPassphrase:
    publicKeyPath: test-config/apps-config/<app-name>/remote-machines-configs/<name-of-file>
    sshConfigs:
      StrictHostKeyChecking: "no"
      PreferredAuthentications: "publickey,password"
    proxyConfig:
      # Proxy configuration
      #PROXY Configuration Options:NO_PROXY, AUTO_DETECT, USE_SYSTEM_PROXY,
MANUAL_PROXY
      proxyConfigType: NO_PROXY
      # Parameters for MANUAL_PROXY
      httpProxyHostname: <proxy-host>
      httpProxyPort: <proxy-port>
      sslProxyHostname: <proxy-host>
      sslProxyPort: <proxy-port>
      ftpProxyHostname: <proxy-host>
      ftpProxyPort: <proxy-port>
      socksHostname: <proxy-host>
      socksPort: <proxy-port>
      # SOCKS_VERSION options: V4, V5
      socksVersion: V5
      socksUsername:
      socksPassword:
      noProxyFor: localhost,127.0.0.1
    # Additional properties for remote machine (Define under additionalProps as child property)
    additionalProps:

# Additional properties (Define under additionalProps as child property)
additionalProps:
```

e. **driver-configs/**: This is the directory where we keep the AppDriver.yaml config file. This driver config file is configured as per the information present in AppConfig.yaml file (Based on the parameters like applicationType, testPlatformType, appWebBrowser). We can use the following command from the command prompt to add new AppDriver.yaml file based on AppConfig.yaml configuration:

> smart-studio --prepare-app-driver-config <app-name>

f. **environments/**: In this directory we can add any environment file based on the information given in AppConfig.yaml file. We can override any parameter information present in AppConfig.yaml file and create our new environment file.

Sample **TestEnv.yaml** file**:**

```
# Environment name
environmentName: TestEnv
```

```
# Application Launch URL. For NativeApp it will contain the path of .app, .api, .apk, .exe etc. application file.
# And for web-app it will contain the web URL that will be used to launch application on web browser.
appLaunchURL:

# API Config file name
apiConfigFilename: ApiConfig-TestEnv.yaml

# Comma separated user profile names
userProfileNames: SampleUserProfile-TestEnv

# Comma separated database profile names
dbProfileNames: [sample-db-TestEnv]

# RemoteMachinesConfig file name
remoteMachinesConfigFileName: RemoteMachinesConfig-TestEnv.yaml
```

NOTE: All these parameters are defined in AppConfig.yaml file but here we are overriding the information present in AppConfig.yaml file to make another environment. Using the command line interface, we can execute the testcases for this environment using the command given below:

```
> mvn clean verify -Dcucumber.filter.tags="@RegressionTest and not @Pending" -Dparallel.threads=1 -Dapps.active.environment=[app-name1]:[environment-name],[app-name2]:[environment-name]
```

3. **App Web Drivers**
   We can store the app web drivers' executable files in the directory **test-config/app-drivers/<platform-name>/<app-type>/<web-browser-type>/** directory. Like for windows platform if you would like to use chrome web browser then the chromedriver.exe file will be stored at the following location:

   test-config/app-drivers/windows/web-app/chrome/

   We can also use the following command on command prompt to get the instruction how to install the app web driver.

   > smart-studio --prepare-app-driver-config <app-name>

   Also if you have prepared the environment file then you can use the following command to get the instruction that how to install the app web driver for the new environment:

   > smart-studio --prepare-app-driver-config-for-env <app-name> <env-name>

4. **Sikuli Configuration**

Sikuli is a tool that is used for OCR (Optical Character Recognition) testing and the testing of the image contents based on the pattern matching. We can configure the Sikuli tool using the **SikuliSettings.yaml** file that is present under **test-config/sikuli-config** directory. Sample contents of **SikuliSettings.yaml** file is given below:

```
#OCR Settings
ocr-tessdata-location: "C:/Program Files/Tesseract-OCR/tessdata"
settings:
  OcrLanguage: "string:eng"
  OcrTextSearch: "boolean:true"
  OcrTextRead: "boolean:true"
  OverwriteImages: "boolean:true"
  ImageCache: "integer:0"

# Additional properties should be added as child under additionalProps element
additionalProps:
```

To use Sikuli tool we should install Tesseract-OCR tool on your machine and the path of **tessdata** directory should be configured in Sikuli Configuration file.

4.4.5 Directory Details: test-data/

This is the directory where we can store test input data in organized way that you can use in Cucumber steps to validate the actual results against this data. You can create sub-directories to store your test data in organized way.

4.4.6 Directory Details: test-results/

This is the directory where system produces the test results in JSON and HTML format. Also it stores all the snapshots taken by the automation program and the downloaded files. The directory structure is given below that is created during testcases execution time:

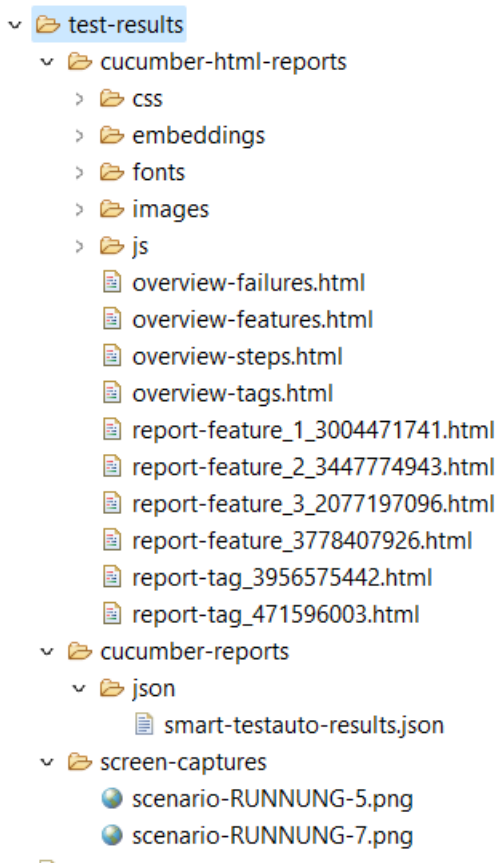


Figure 4-5 Directory Details: test-results/

You can open the following files to see the test results.

- **overview-features.html**: This file contains the overall feature wise test report.
- **overview-failures.html:** This file contains all the failed test scenario information.
- **overview-steps.html:** This file contains the test report step wise.
- **overview-tags.html:** This file contains the organized view of the test scenarios by cucumber tags.

4.5 Execution of testcases / test-scenarios using STAS

STAS uses Maven, TestNG and Cucumber tools to run the automated testcases / test scenarios written in Cucumber Gherkin language. The following command is used to run the test-cases based on the applied tags on Cucumber feature files:

> smart-runner [command line options]

Example:

> smart-runner run-regression-tests 5

The above command will run the regression tests with degree of parallelism = 5.

Supported command line options:

A. **run-regression-tests <num-parallel-threads>**
Description: This is used to run the Cucumber test scenarios on which @**RegressionTest** tag is applied and **@Pending** tag is not applied on the same scenario.

B. **run-regression-but-not-sequential-tests <num-parallel-threads>**
Description: This is used to run the Cucumber test scenarios on which **@RegressionTest** tag is applied and **@Pending @Sequential** tags are not applied on the same scenario.

C. **run-failed-tests <num-parallel-threads>**
Description: This is used to run the Cucumber test scenarios on which **@Failed** tag is applied and **@Pending** tag is not applied on the same scenario.

D. **run-smoke-tests <num-parallel-threads>**
Description: This is used to run the Cucumber test scenarios on which **@SmokeTest** tag is applied and **@Pending** tag is not applied on the same scenario.

E. **run-sanity-tests <num-parallel-threads>**
Description: This is used to run the Cucumber test scenarios on which **@SanityTest** tag is applied and **@Pending** tag is not applied on the same scenario.

F. **run-sequential-tests**
Description: This is used to run the Cucumber test scenarios on which **@Sequential** tag is applied and **@Pending** tag is not applied on the same scenario. These testcases cannot be run in parallel mode.

G. **run-temp-scenarios <num-parallel-threads>**
Description: This is used to run the Cucumber test scenarios on which **@TempScenario** tag is applied and **@Pending** tag is not applied on the same scenario.

H. **start-debug-server <num-parallel-threads>**
Description: This is used to start the debug server at port 5005 so that test script developer can debug the step definition code line by line to identify the problem in the code. This option is used to run the Cucumber test scenarios on which **@TempScenario** tag is applied and **@Pending** tag is not applied on the same scenario. Using your Java code editor tool like Eclipse or IntellijIDEA, we can configure the remote application at remote port 5005 to start the debugger in remote mode to debug the program.

I. **find-all-missing-stepdefs**
Description: This option is used to find all missing step definitions in the Cucumber step definition files for that the user defined step is not present. This is used to run the Cucumber test scenarios on which **@Pending** tag is not applied.

J. **find-missing-stepdefs-for-temp-scenarios**
Description: This is used to find all missing step definitions from the Cucumber test scenarios on which **@TempScenario** tag is applied and **@Pending** tag is not applied on the same scenario.

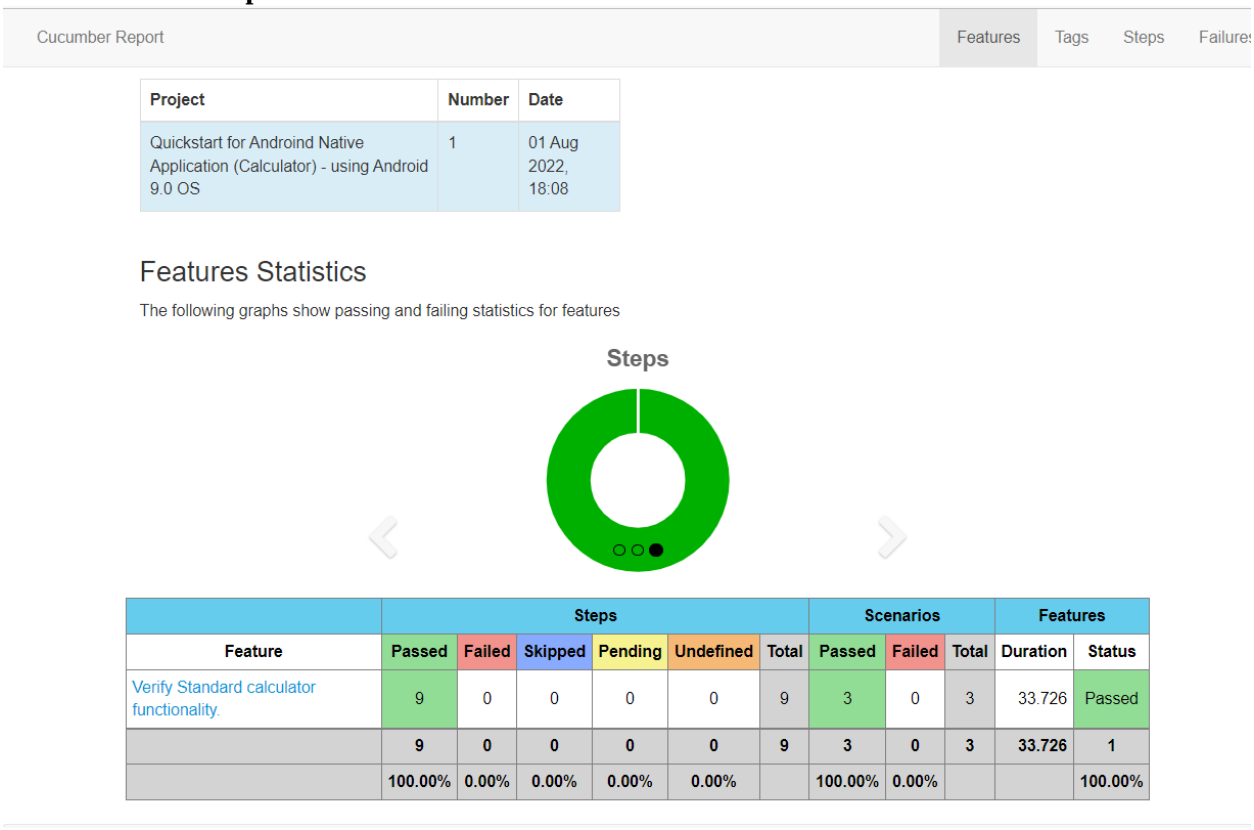
K. **generate-report**

Description: This is used to manually generate the HTML reports in the case when system may fail to generate report after the testcase execution.

4.5.1 Sample test execution reports

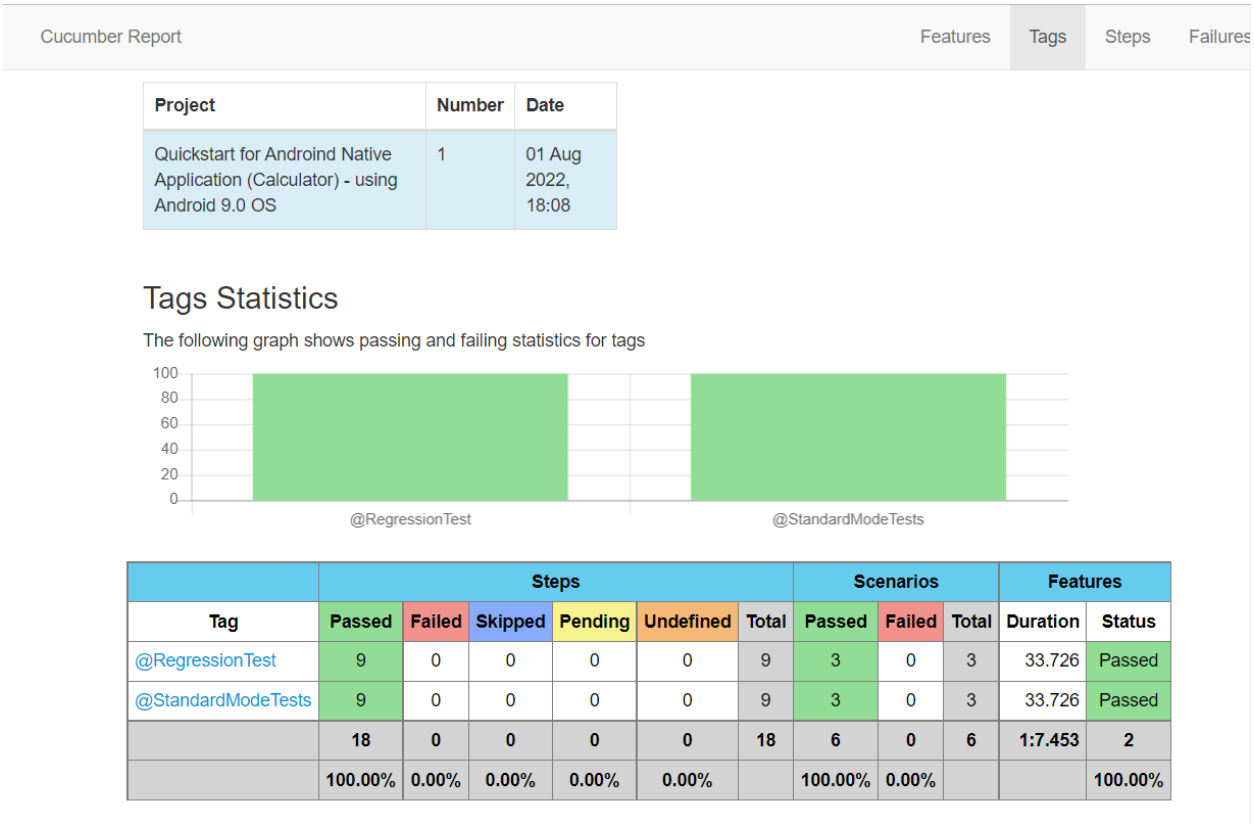
These are just the sample reports that this tool generates after the execution of test cases. Reports are generated by 3rd party plugin that is configured in Maven pom.xml file. These reports contain the screenshot embedded in each scenario. STAS tool generates the reports in **test-results/** directory.

1. Feature Overview report



	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
Verify Standard calculator functionality.	9	0	0	0	0	9	3	0	3	33.726	Passed
	9	**0**	**0**	**0**	**0**	**9**	**3**	**0**	**3**	**33.726**	**1**
	100.00%	**0.00%**	**0.00%**	**0.00%**	**0.00%**		**100.00%**	**0.00%**			**100.00%**

2. Tags Overview Report



Cucumber Report Features Tags Steps Failures

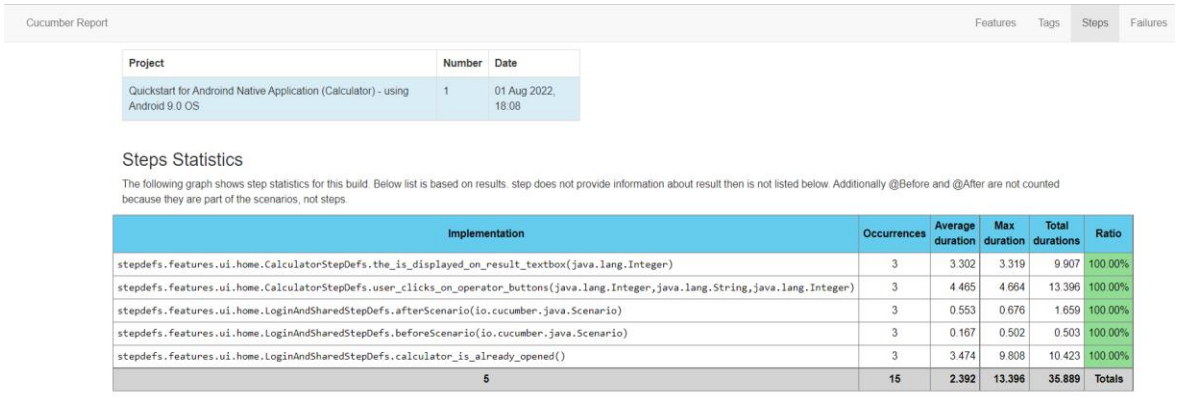
Project	Number	Date
Quickstart for Androind Native Application (Calculator) - using Android 9.0 OS	1	01 Aug 2022, 18:08

Tags Statistics

The following graph shows passing and failing statistics for tags

	Steps						Scenarios			Features	
Tag	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
@RegressionTest	9	0	0	0	0	9	3	0	3	33.726	Passed
@StandardModeTests	9	0	0	0	0	9	3	0	3	33.726	Passed
	18	0	0	0	0	18	6	0	6	1:7.453	2
	100.00%	0.00%	0.00%	0.00%	0.00%		100.00%	0.00%			100.00%

3. Steps Overview Report



Cucumber Report Features Tags Steps Failures

Project	Number	Date
Quickstart for Androind Native Application (Calculator) - using Android 9.0 OS	1	01 Aug 2022, 18:08

Steps Statistics

The following graph shows step statistics for this build. Below list is based on results. step does not provide information about result then is not listed below. Additionally @Before and @After are not counted because they are part of the scenarios, not steps.

Implementation	Occurrences	Average duration	Max duration	Total durations	Ratio
stepdefs.features.ui.home.CalculatorStepDefs.the_is_displayed_on_result_textbox(java.lang.Integer)	3	3.302	3.319	9.907	100.00%
stepdefs.features.ui.home.CalculatorStepDefs.user_clicks_on_operator_buttons(java.lang.Integer,java.lang.String,java.lang.Integer)	3	4.465	4.664	13.396	100.00%
stepdefs.features.ui.home.LoginAndSharedStepDefs.afterScenario(io.cucumber.java.Scenario)	3	0.553	0.676	1.659	100.00%
stepdefs.features.ui.home.LoginAndSharedStepDefs.beforeScenario(io.cucumber.java.Scenario)	3	0.167	0.502	0.503	100.00%
stepdefs.features.ui.home.LoginAndSharedStepDefs.calculator_is_already_opened()	3	3.474	9.808	10.423	100.00%
5	15	2.392	13.396	35.889	Totals

4. Failure Overview Report

5 Page Object Model in STAS?

STAS system has its own page object model to create UI page elements / page objects. It uses java object initialization mechanism.

5.1 Page Object Locators

We can create page elements using **different type of locators** (**Locators** are the mechanism using that system can identify the page element on the web page or native applications so that different kind of operations can be performed on the page element like extract value, type text, mouse operations, touch screen operations etc.) as given below:

1. Id
2. Name
3. ClassName
4. TagName
5. CssSelector
6. AccessibilityId: Mostly used with mobile testing automation.
7. LinkText
8. PartialLinkText
9. Xpath

Also, we can be able specify platform specific and web browser specific locators to the page element. That helps when we are running the same scenario on different platforms and different web browsers then it can run successfully without changing the test scenario steps.

In STAS project, we store all page object classes under **src/main/java/page_objects/<app-name>/** directory for the specific application.

Sample **LoginPO.java** file:

```
public interface LoginPO {
        TextBoxSD Textbox_Username = new TextBoxSD("Username", "//input[@id='username']");
        TextBoxSD Textbox_Password = new TextBoxSD("Password", "//input[@id='password']");
        ButtonSD Button_SignIn = new ButtonSD("Sign in", "//button[@id='signin']");
}
```

Sample **LoginSuccessPO.java** file:

```
public interface LoginSuccessPO {
        // TODO: Change these elements according to your application
        // Add unique element that is always visible after login (even page is switched)
        // You can change these elements according to your application and accordingly update the
AppLoginSuccessPageValidator.java
        LabelSD Label_UniqueElement = new LabelSD("<Unique-element-name-here>", "//span[normalize-
        space()='<page-tiltle-here>']");
        ButtonSD Button_Logout = new ButtonSD("Logout", "//button[@id='logout']");
}
```

To create a new page object (PO) class within any application just go to the respective application directory (**src/main/java/page_objects/<app-name>/**) on your code editor and create a Java interface with the Naming convention **<PageName>PO.java**. In this interface you can declare / create all the page elements that are required for that particular page automation. We can use the **Standard DOM classes** like TextBoxSD, LabelSD, ButtonSD etc. or **Standard Image classes** like TextBoxSI, LabelSI, ButtonSI, ImageSI etc. to create the page elements in the PO classes. The page elements defined in these PO classes can be used directly on Cucumber feature file to perform operations on web browser or native applications.

5.2 How to use page elements / page objects in Cucumber feature file?

The page element usage is very simple in Cucumber feature file. Since STAS provides the Standard Cucumber steps and step definitions. Using these step definitions, we can frame any scenario. In step definitions wherever it mention like page element or page object in that place we have to pass these page elements as an argument.

Example:

Then verify "**myapp.LoginSuccessPO.Label_UniqueElement**" page element is visible on "Home page".

Then verify "**myapp.LoginSuccessPO.Label_UniqueElement**" page object is visible on "Home page".

In the steps above, the highlighted text is showing the page element specified as a **relative path** to **page_objects** package. Terms "page object" and "page element" are used interchangeably to represent the same thing.

NOTE: Page Object (PO) class is a term used to represent the web page / screen name that contains the set of page elements.

Refer https://github.com/mkrishna4u/smart-testauto-cucumber-defaults-en/tree/main/docs/STAS-StepDefinitions-En-TestDevelopersGuide.pdf **user guide for more details on Standard Cucumber Step Definitions supported by STAS tool.**

5.3 Standard DOM classes to create Page Element / Page Object

There are many standard DOM (Document Object Model) classes are supported by STAS tool. These classes can be used to create the page element for Web Page or The Native Application screens. The supported SD Classes and its description is given below:

All these SD classes are present in org.uitnet.testing.smartfwk.ui.standard.domobj package of smart-testauto-framework project.

S. No.	SD Class Name / Page Element Class	Description
1	TextBoxSD	On web page it is the DOM **input** element where user can type the text. On native application, it is the textbox element that may be identified by any locator where user can type text.

2	TextAreaSD	On web page it is the DOM **textarea** element where user can type the multiline text. On native application, it is the textarea element that may be identified by any locator where user can type multiline text.
3	InputFileSD	On web page it is the DOM **input** element (with type=file property) where user can select the file from the local system. On native application, it is the file element that may be identified by any locator that can be used to select a file from local system.
4	CheckBoxSD	On web page it is the DOM **checkbox** element where user can check / uncheck the box. On native application, it is the checkbox element that may be identified by any locator where user can check / uncheck box.
5	RadioButtonSD	On web page it is the DOM **radio** button element where user can select the box. On native application, it is the radio button element that may be identified by any locator where user can select the box.
6	RadioButtonGroupSD	Generally radio buttons are used in a group so user can select one of the option from the group. In that case we can use this class.
7	ComboBoxSD	On web page it is the DOM **select** element with dropdown values where user can select specific value. On native application, it is the combobox element with dropdown options that may be identified by any locator where user can select specific value. It supports single and multi selection.
8	ListBoxSD	On web page it is the DOM **select** element (with multiple=true property) with option values where user can select single or multiple options. On native application, it is the Listbox element that allow single or multiple selection that may be identified by any locator.
9	HyperlinkSD	On web page it is the DOM **anchor (a)** element that is also called hyperlink.

		On native application, it is the hyperlink element that may be identified by any locator.
10	ImageSD	On web page it is the DOM **img** element that is also called image element. On native application, it is the image element that may be identified by any locator.
11	ButtonSD	On web page it is the DOM **input** element (with type=button property) or **button** element that is used to perform operation. On native application, it is the button element that may be identified by any locator.
12	LabelSD	Label is nothing but any textual information visible on the screen like Heading, Section, Control Label etc.
13	WebPageTitleSD	This is also a label that is the title of the web page or any screen or modal window etc.

Example:

Default declaration is given below that uses the xpath mechanism.

```
TextBoxSD Textbox_Username = new TextBoxSD("Username", "//input[@id='username']");
```

OR

If we would like to configure different locators based on the different application types and web browser types, we can use the chaining methods as given below with any of the SD class. STAS system will determine automatically which locator to use based on the application configuration and its environment configuration (present in test-config/apps-config directory)

```
TextBoxSD Textbox_Username = new TextBoxSD("Username", "//input[@id='password']") // this is default locator set as xpath
        .addPlatformLocatorForWebApp(PlatformType.linux, WebBrowserType.firefox, LocateBy.Id, "username")
        .addPlatformLocatorForWebApp(PlatformType.linux, WebBrowserType.chrome, LocateBy.Id, "username")
        .addPlatformLocatorForNativeApp(PlatformType.windows, LocateBy.AccessibilityId, "username")
        .addPlatformLocatorForNativeApp(PlatformType.android_mobile, LocateBy.AccessibilityId, "username")
        .addPlatformLocatorForNativeApp(PlatformType.ios_mobile, LocateBy.AccessibilityId, "username")
```

```
.addPlatformLocatorForNativeApp(PlatformType.mac, LocateBy.CssSelector, "#username");
```

In the example above, different type of locators are configured based on the platform type and web browser type.

5.3.1 Parameterized locators supported in SD classes

This tool support parameterized locators. Most of the time these kind of locators are used when we process the data table on user interface. Parameters are specified in locators using the following syntax:

:paramName

Also, we can specify multiple parameters in a single locator as per your need. This feature reduces the creation of lot of locators that are required to by system to automate the UI functionality.

Example:

Let's say we have **Employees table** on **Employee Search screen**, we wanted to perform search based on department name like to know what employees exist in let's say **Admin** department. Then after search employee information we would like to check that all the rows contains department name as Admin in the search results. To do so we can create page element of department name column as parameterized.

```
LabelSD EmployeeTable_LabelColumn = new LabelSD("Column data for column ':columnName'", "//div[contains(@class, 'table')]//div[@col-id=':colId']/span[@role='label']");

HyperlinkSD EmployeeTable_HyperlinkColumn = new HyperlinkSD("Column data for column ':columnName'", "//div[contains(@class, 'table')]//div[@col-id=':colId']/a");
```

In the example above, we created two page elements, one for label type of columns (which contain just text data) and another for hyperlink type of columns (which contains hyperlinks on the values). In both the page elements **columnName** and **colId** are parameters that we can pass using cucumber step definition to get the column information like:

```
Then verify that the text part of '{name: "myapp.EmployeeSearchPO. EmployeeTable_LabelColumn", maxTimeToWaitInSeconds: 10, params: {columnName: "Department Name", colId: "2"}}' page element matches "Admin" text where TextMatchMechanism="exact-match-with-expected-value".

And verify that the text part of '{name: "myapp.EmployeeSearchPO. EmployeeTable_HyperlinkColumn", maxTimeToWaitInSeconds: 10, params: {columnName: "State", colId: "3"}}' page element matches "Virginia" text where TextMatchMechanism="exact-match-with-expected-value".
```

The steps given above are used to verify the column value of the table. First step is verifying the **Department Name** column value that is present as a second column in table, each row contains Admin as department name. Then second step is verifying the 3rd column (State) value that all

rows present in State column should have Virginia link. Here **maxTimeToWaitInSeconds** parameter value is used to locate the element on user interface until time limit is expired.

5.4 Standard Images classes to create Page Element / Page Object

These are the classes that are used to create page element based on the images. It uses image pattern mechanism to detect the page elements on the screen. The following operations can be performed on the detected page elements: click, double click, right click, type text etc. To create image classes we should use the SikuliX tool to get the image snapshot from the screen that can be configured in SI classes. The following SI classes are supported:

Download URL for SikuliX IDE to take small clips from screen for SI classes:
https://raiman.github.io/SikuliX1/downloads.html

S. No.	SI Class Name / Page Element Class	Description
1	TextBoxSI	Used to perform operations on textbox control on UI.
2	TextAreaSI	Used to perform operations on textarea control on UI.
3	CheckBoxSI	Used to perform operations on checkbox control on UI.
4	RadioButtonSI	Used to perform operations on Radio Button on UI.
5	ComboBoxSI	This is a combobox control that has dropdown. Functionality is limited.
6	ListBoxSI	It is a listbox control.
7	HyperlinkSI	This is used for hyperlink control on UI.
8	ImageSI	This is used for any image object on UI.
9	LabelSI	This is used for any text label like headings, section, control label etc.
10	ButtonSI	Used to perform operations on button control on UI.

Example:

```
ButtonSI Button_Login = new ButtonSI("Login", "login-button.jpg", new ObjectLocation());
```

NOTE: login-button.jpg image should be present in **sikuli-resources/** directory. We can organize the folders under sikuli-resources/ directory and accordingly we have to specify the relative image path with respect to sikuli-resources/ directory like

```
ButtonSI Button_Login = new ButtonSI("Login", "login/login-button.jpg", new ObjectLocation());
```

In the above example, the login-button.jpg image is present under sikuli-resources/login/ directory.

OR

If we would like to configure different images based on the different application types and web browser types, we can use the chaining methods as given below with any of the SD class. STAS system will determine automatically which image to use based on the application configuration and its environment configuration (present in test-config/apps-config directory)

```
ButtonSI Button_Login = new ButtonSI("Login", "sikuli-resources/login-button.jpg", new ObjectLocation())
        .addPlatformImageForNativeApp(PlatformType.windows, "login-button-windows.jpg")
        .addPlatformImageForNativeApp(PlatformType.linux, "login-button-linux.jpg")
        .addPlatformImageForWebApp(PlatformType.windows, WebBrowserType.chrome, "login-button-windows-chrome.jpg")
        .addPlatformImageForWebApp(PlatformType.windows, WebBrowserType.firefox, "login-button-windows-firefox.jpg");
```

In the example above, different type of images is configured based on the platform type and web browser type.

Object Location: ObjectLocation class is used to inform SI validator classes where to find page element. There are different ways we can specify the object location on UI.

1. **new ObjectLocation()**: We are not specifying the specific region so the system will detect image on the whole screen.
2. **new ObjectLocation(x, y, width, height)**: We can specify the region on the screen in which this object can be found. If the object not found within this region then system will throw error.
3. **new ObjectLocation(refObject, refObjectPosition, refObjectDistanceInPx)**: We can specify any page element as refObject. In this case we have to tell system where the ref object is present like left, top, right, bottom and what is the approximate distance of the refObject from the the object that we are trying to detect on UI. Example:

```
LabelSI Label_Username = new LabelSI("Username", "username-label.jpg", new ObjectLocation());
```

```
TextBoxSI TextBox_Username = new TextBoxSI("Username", "textbox/textbox-left-part.jpg",
"textbox/textbox-right-part.jpg",
new ObjectLocation(Label_Username, ReferenceObjectPosition.LEFT, 30));
```

In the example above, we created a username label first (that will be detected anywhere on the screen), then created username textbox control by specifying the textbox images and then we added the object location with respect to reference object (here username label control is used as a reference object), here we are telling system that find username textbox that is present right to the reference object as a maximum distance of 30 pixels from the textbox. If system does not find textbox element on screen as per the configuration specified then it will give assertion error.

6 How to write test scenarios using STAS?

As mentioned above STAS uses Cucumber Gherkin language to design scenarios or scenario outlines using out of the box step definitions provided by STAS tool. Refer ***Standard Step Definition Details (provided by STAS tool)*** section for standard cucumber step definitions information. Each subsection given below describes what information can be recorded while designing your scenario or scenario outline.

6.1 Tags supported in STAS Cucumber Feature File to group scenarios

The following Tags are supported in Cucumber Feature File to group the test scenarios. You can create your own tags and run the test scenarios. But STAS tool provides the following tags support out of the box:

A. **@TempScenario:** This tag is generally used by test engineer during development of the test scenario. When the test scenario is in development phase, apply this tag on the cucumber scenario / scenario outline. And this scenario / scenario outline can be run using the following command:

```
> smart-runner run-temp-scenarios
```

NOTE: Remove this tag once the scenario is successfully developed.

B. **@RegressionTest:** Apply this tag on cucumber scenario / scenario outline to mark that scenario as regression test scenario. When we use the following below command, this will run all the scenarios that are marked @RegressionTest.

```
> smart-runner run-regression-tests
```

C. **@SanityTest:** Apply this tag on cucumber scenario / scenario outline to mark that scenario as sanity test scenario. When we use the following below command, this will run all the scenarios that are marked @SanityTest.

```
> smart-runner run-sanity-tests
```

D. **@SmokeTest:** Apply this tag on cucumber scenario / scenario outline to mark that scenario as smoke test scenario. When we use the following below command, this will run all the scenarios that are marked @SmokeTest.

```
> smart-runner run-smoke-tests
```

E. **@Failed:** Apply this tag on cucumber scenario / scenario outline to mark that scenario as Failed scenario. When we use the following below command, this will run all the scenarios that are marked @Failed.

```
> smart-runner run-failed-tests
```

F. **@Pending:** Apply this tag on cucumber scenario / scenario outline to mark that scenario as Pending scenario. Pending scenario means that it is not implemented and do not run. When we use any command given above, it will not include scenarios that are marked @Pending.

6.2 Find missing step definitions for customized cucumber steps

To find the missing step definitions for the scenarios that are marked as **@TempScenario** in cucumber feature file. Use the following command on console window:

```
> smart-runner find-missing-stepdefs-for-temp-scenarios
```

To find the missing step definitions for all the scenarios, use the following command on console window:

```
> smart-runner find-all-missing-stepdefs
```

6.3 Standard Step Definition Details (provided by STAS tool)

To find the standard step definitions supported by the STAS tool, please visit https://github.com/mkrishna4u/smart-testauto-cucumber-stepdefs-en link. If you have configured your project in Eclipse or IntellijIDEA code editor then during scenario writing, inside feature file you can see the code completion for the steps. Only thing is that you have to configure Cucumber plugin and make your project as Cucumber project. Also specify the following paths as glue code.

```
stepdefs,org.uitnet.testing.smartfwk.core.stepdefs.en
```

We have different type of standard step definitions files using that you can automate any scenario. For more information, please refer **TBD** document. High level overview is given below:

1. **SmartStepDefs:** This is a main hook file that defines the @Before and @After methods definition that used to run before each scenario and after each scenario completion respectively.
2. **SmartUiBasicAppOperationsStepDefs:** This defines basic UI operations to perform operations on UI like connect to UI application using specified user profile. User profiles are configured in test-config/apps-config/ directory to its specific app. Also it contains step definitions related to opening URL, take screenshot etc.
3. **SmartUiFormElementOperationsStepDefs:** This defines the UI operations like operations on the form elements / page objects like enter the information on page elements, type text on textbox or textarea, extract information from the element, verify information of the elements (i.e. visibility of the element, default information of the element) etc.
4. **SmartUiMouseOperationsStepDefs:** This defines the mouse operations related step definitions that user can perform on user interface. It includes operations like click, double click, click hold and release, right click, hoverover etc.

5. **SmartUiTouchScreenOperationsStepDefs:** This defines touch pad / touch screen operations on user interface like tap, swipe, zoom in, zoom out etc operations.
6. **SmartUiWindowAndFrameOperationsStepDefs:** This defines the step definitions to handle multiple windows, iframes etc. like switching to windows and switching to frame etc.
7. **SmartUiKeyboardOperationsStepDefs:** This defines the step definitions to handle keyboard related operations like keydown, keyup, keypress, combo keys operations like (CTRL + a, CTRL + v, CTRL + click) etc.
8. **SmartFileUploadStepDefs:** This defines file upload related operations on UI.
9. **SmartApiStepDefs:** This defines API Testing related operations and response data verifications. Like HTTP GET, POST, PUT, DELETE etc.
10. **SmartDatabaseManagementStepDefs:** This defines Database Management related step definition like to access the database information using SQL, update database information using SQL query etc.
11. **SmartLocalFileManagementStepDefs:** This defines step definitions to verify the downloaded files from UI or API.
12. **SmartRemoteFileManagementStepDefs:** This defines step definitions to verify the uploaded files on remote remote server using SSH (Secured Shell) protocol.
13. **SmartExcelDataManagementStepDefs:** This defines step definitions to read and verify the contents of Excel File.
14. **SmartCsvDataManagementStepDefs:** This defines step definitions to read and verify the contents of CSV (Comma Separated Value) File.
15. **SmartJsonDataManagementStepDefs:** This defines step definitions to read and verify the contents of JSON File or data. It also defines the steps to update the JSON data.
16. **SmartYamlDataManagementStepDefs:** This defines step definitions to read and verify the contents of YAML File or data. It also defines the steps to update the YAML data.
17. **SmartXmlDataManagementStepDefs:** This defines step definitions to read and verify the contents of XML File or data. It also defines the steps to update the XML data.
18. **SmartTextualDataManagementStepDefs:** This defines step definitions to read and verify the contents of text File or data.
19. **SmartVariableManagementStepDefs:** This defines step definitions to read and verify the contents of the variables.

For more details on these default cucumber step definitions, please refer the following link:

https://github.com/mkrishna4u/smart-testauto-cucumber-stepdefs-en/tree/main/docs

Using the smart step definition (specified above) you can automate

1. Any user interface like **web or native user interface**.
2. Automate **API Testing** like REST APIs.
3. Automate **database testing**.
4. Automate **file download or upload testing**.
5. Manage multiple windows and frames for web based applications.
6. Automate mobile native and web applications (Android and iOS both).

6.4 Variables in Cucumber feature files

Variables are the name in which we can store any value like String, Integer, Long, Float, Boolean, Double, List, Map etc. value. These variables are very useful when we have a requirement to carry information from one step to another step. Like in one step, we can read the value from UI control and store into a variable and in another step you can validate the variable value as expected value. For example:

```
Then get input text value of "myapp.UserRegistrationPO.Textbox_GeneratedID" page element and store into "GENERATED_ID_VAR" variable.

And verify valueOf("GENERATED_ID_VAR") variable "=" "TEST-ID-1".
```

Here highlighted GENERATED_ID_VAR is a variable. You can give any name that you can use in successive steps. First step store the input text value into this variable. And in second step we can verify the value of this variable is "TEST-ID-1".

Similarly, we can read value from any configuration or test data (i.e. Excel, CSV, JSON, YAML etc.) files and store into the variable. After that we can verify the stored variable value or use the variable value to verify the API response information and UI screen information.

NOTE: System does not carry the variable value across the scenarios. The scope of the variable is within the single scenario in which it is defined.

6.5 Cucumber feature file: Steps and its arguments information

The standardized steps definitions provided by STAS has standardized information that you can pass into step arguments. This information related to the following topics:

A. Arguments related to - UI steps / step definitions

1. **UI Page Element / Page Object**

   UI Page Element or Page Object are the same terms used interchangeably to represent the page element like Textbox, TextArea, Button, Hyperlink, Label etc. on the UI page. In Cucumber scenario steps you can specify page element in the formal given below:

   a. **Direct dotted notation**

   Example:

   ```
   Then get input text value of "myapp.UserRegistrationPO.TextBox_GeneratedID" page element and store into "GENERATED_ID_VAR" variable.
   ```

   Where: highlighted page element has 3 parts:
   <AppName>.<PO-ClassName>.<PageElementName>

   b. **In JSON format**

   Using JSON format, additional information can also be supplied to the page element.
   Syntax:

   ```
   { name: "myapp.UserRegistrationPO.TextBox_GeneratedID",
     maxTimeToWaitInSeconds: 5, params:  {key1: "value1", key2: "value2"} }
   ```

   Where:

name is the name of page element in the format given below (Mandatory param): <AppName>.<PO-ClassName>.<PageElementName>

maxTimeToWaitInSeconds is the max time to locate element on the web page. It polls element every 2 seconds until it finds or timeout. It is mandatory param.

params information is used in parameterized xpath or locators that are specified page element declaration in PO classes using **:paramName1** syntax. Here **paramName1** is parameter name in page element. This is optional param.

Example: Sample Step information

Then get input text value of '**{name: "myapp.UserRegistrationPO.TextBox_GeneratedID", maxTimeToWaitInSeconds: 10, params: {paramName1: "pvalue"} }**' page element and store into "GENERATED_ID_VAR" variable.

Sample for **TextBox_GeneratedID** page element with **paramName1**:

*TextBoxSD **TextBox_GeneratedID** = new TextBoxSD("Generated ID - :paramName1", "//form[@Id='userReg']/input[@name='genId' and contains(@class, '**:paramName1**')]");*

2. **Browser Types**
   The following supported web browser type values can be pass as an argument to steps: chrome, firefox, edge, opera, safari, internet-explorer, remote-web-driver-provider, not-applicable

3. **Application Name**
   We should pass the correct application name that are configured in the STAS. All configured application names are present under **./test-config/apps-config/** directory in STAS project.

4. **User Profile Name**
   User profiles contains the user and role related information. We can also add additional parameters information also into these user profiles as per application need.

   We should pass the correct user profile name for the specified application that are configured in the STAS. All configured user profiles for any application are configured under **./test-config/apps-config/<app-name>/AppConfig.yaml** file in STAS project. The corresponding user profiles details are present at **./test-config/apps-config/<app-name>/user-profiles/** directory.

5. **Validator method of UI Page Element / Page Object**
   Validator methods are the methods that are defined in **<>ValidatorSD.java** and **<>ValidatorSI.java** files for the respective page element. To know the method name, open the respective **<>ValidatorSD.java** file and get the method name and argument information and according pass on that information cucumber step as JSON info in the format given below:

{name: "methodName", argsType: [Integer.TYPE, String.class], argsValue: [15, "test string"]}

Example:
When call '**{name: "methodName", argsType: [Integer.TYPE, String.class], argsValue: [15, "test string"]}**' validator method of "<page-element-info-from-PO>" page element to "perform some operation" and store output to "VAR_NAME" variable.

This step will call the method of the validator class associated with specified page element and store the value into variable VAR_NAME.

6. **Keyboard keys**
   We can also be able to perform operations using multiple key sequences like "ctrl+c" keys combination is used to copy the selected text on UI page. How we can use these keys in Cucumber steps? These keys can be specified in cucumber steps as JSON array like given below:

   When use '**["CONTROL", "c"]**' key(s) on "<page-element-info-from-PO>" page element to "copy the selected text".
   Then "selected text" will be "copied to the clipboard".

7. **Mouse events like click, rightClick, doubleClick**
   We can also be able to perform mouse event on the UI page elements like click, doubleClick, rightClick, clickAndHold, release, hoverover etc. Examples are given below:

   When use '**["CONTROL"]**' key(s) and "**click**" mouse event together on "<page-element-info-from-PO>" page element to "open link in a new window/tab".

   Or

   When **click** on "<page-element-info-from-PO>" page element to "submit form data on server".
   Then "form data" will be "submitted on server".

8. **Location where to type new text**
   We can type the text in editable textbox or textarea page element 3 ways: append in the beginning, append at the end or replace with new text. By default system types the text in field by replacing the existing text (if exists) with new text. So based on this concept there are 3 locations value supported by system:
   start: Appends text in the beginning.
   end: Appends text at the end.
   replace: Replace existing text with new text.

   Example:
   When type "sample text here" text in "<page-element-info-from-PO>" page element at "start" location.

   This step will add the text in the beginning of the text present in the page element.

   When type "sample text here" text in "<page-element-info-from-PO>" page element.

This step will add the text by replacing the existing text with new text in the specified page element.

B. Arguments related to – General steps / step definitions / any definition where applicable

9. Expected Information / Expected Value

Expected information is the information that can be validated on UI page element(s). The information can be specified in String form or in JSON form in Cucumber steps.

JSON Syntax:

{ev: "<value here>", valueType: "value type here", textMatchMechanism: "Text match mechanism here", n: "", inOrder: "yes/no", ignoreCase: "yes/no"}

Where:

ev: This is expected value where we can specify any type of value like String, Integer, Long, Float, Double, List etc in JSON format. This is mandatory.

valueType: This indicates what type of information is specified in **ev** parameter. The following value types are supported (This is mandatory. Default is String):

string, integer, decimal, boolean, string-list, integer-list, decimal-list, boolean-list.

textMatchMechanism: This is the mechanism to match with actual value. The following types of text match mechanism supported as specified in ***11-Text Match Mechanism*** section. This is optional. Default value is **exact-match-with-expected-value**.

n: This is a number value that is generally used in conjunction to "**contains-atleast-n**" or "**contains-atmax-n**" operator. Where we generally verify the count of the actual results. This is optional.

inOrder: This is generally used to check the actual results in list are in order to the expected value. Valid values are: **yes**, **no**. This is optional. Default value is "no".

ignoreCase: This is generally used to verify the contents of two lists. System checks whether it should do case insensitive matching or not based on the ignoreCase value specified. Valid values are: **yes**, **no**. This is optional.

NOTE: User can also be able to specify the expected information in pure string form like.

"This is sample text"

10. Variables / Variable Name

These are named variable. Variable name is a name in which any value can be stored. Variables are used to carry the value from one Cucumber step to another step within the same scenario. Example:

```
Then get input text value of "myapp.UserRegistrationPO.TextBox_GeneratedID" page element and store into "GENERATED_ID_VAR" variable.

And verify valueOf("GENERATED_ID_VAR") variable "=" "TEST-ID-1".
```

Where **GENERATED_ID_VAR** is the name of the variable in which the input text value of page element will get stored. And the value stored in this variable can be verified in another step as given above.

11. **Text Match Mechanism**

Text match mechanisms are generally used in cucumber steps to match the actual value with expected value. The supported text match mechanisms are:

exact-match-with-expected-value: Matches actual value with expected value that are exactly same. This is case sensitive match.

starts-with-expected-value: Matches actual value starts with expected value. This is case sensitive match.

ends-with-expected-value: Matches actual value ends with expected value. This is case sensitive match.

contains-expected-value: Matches actual value contains expected value. This is case sensitive match.

match-with-regular-expression: Matches actual value satisfy the regular expression (specified in expected value). This is case sensitive match.

exact-match-with-expected-value-after-removing-spaces: System first removes all the spaces from actual value then it performs exact value match with expected value. This is case sensitive match.

ic-exact-match-with-expected-value: Matches actual value with expected value that are exactly same. This is case insensitive match.

ic-starts-with-expected-value: Matches actual value starts with expected value. This is case insensitive match.

ic-ends-with-expected-value: Matches actual value ends with expected value. This is case insensitive match.

ic-contains-expected-value: Matches actual value contains expected value. This is case insensitive match.

ic-match-with-regular-expression: Matches actual value satisfy the regular expression (specified in expected value). This is case insensitive match.

ic-exact-match-with-expected-value-after-removing-spaces: System first removes all the spaces from actual value then it performs exact value match with expected value. This is case insensitive match.

12. **File Name Match Mechanism**
    File name match mechanisms are same as **Text match mechanisms** as specified in previous section.

13. **Reading multiple records and store into variable**
    There are many Cucumber step definitions provided by STAS to read multiple values from UI Elements or from input data files (i.e. Excel, CSV, JSON, YAML, XML etc.) or from Database. These values can be stored in variables and can be validated or pass on as argument to another step to complete the step work.

14. **Operators**
    STAS supports many operators that can be used in Cucumber steps to validate the actual information with expected information. Supported operators are given below:

Operator	Description
=	Equal
!=	Not equals
>	Greater than
>=	Greater than and equal to
<	Less than
<=	Less than and equal to
in	Checks whether actual value in the provided list. List specified in JSON format like: [4, 5, 6]
!in	Checks whether actual value is not in the provided list. List specified in JSON format like: [4, 5, 6]
contains	Checks whether actual value contains the expected value. It supports all value types.
!contains	Checks whether actual value does not contain the expected value. It supports all value types.
contains-atleast-n	Generally this is used on list where we can check actual list contains at least N elements.
contains-atmost-n	Generally this is used on list where we can check actual list contains at most N elements.
starts-with	Checks actual value starts with expected value.
!starts-with	Checks actual value does not start with expected value.
ends-with	Checks actual value ends with expected value.

!ends-with	Checks actual value does not ends with expected value.
present	Checks value present and it is not null.
!present	Checks value does not present and it is null.

15. Input Value

Input value is used in UI based Cucumber steps where user enter the value on UI page elements or select any dropdown option or check/uncheck checkbox or select radio button etc. The JSON syntax is given below:

{value:"", valueType: "", action: "", location: "", autoValueInputs: "", selectingOptionMatchMechanism: "", toPo: ""}

Where

value: Any value can be specified like String, Integer, Long, Float, Double, Boolean, List etc. This is mandatory.

valueType: This indicates what type of information is specified in **value** parameter. The following value types are supported (This is mandatory. Default is String):

auto-gen, string, integer, decimal, boolean, string-list, integer-list, decimal-list, boolean-list.

action: Action indicates that what type of operation to be performed to enter the value on UI controls. Supported actions are:

Input Action Name	Description
type	Used to type value on UI input controls like TextBox, TextArea etc.
cmd-keys	Used to specify control key operations to enter the value like ["CONTROL", "v"]
mouse-click	Used to perform mouse click on the element to enter the value.
mouse-doubleclick	Used to perform mouse double click on the element to enter the value.
mouse-drag-drop	Used to perform drag and drag to move values from one UI control to another control.
check	Used on UI checkbox control to mark it checked.
check-all	Used on UI checkbox control to mark all the checkboxes checked specified by a specific xpath.

uncheck	Used on UI checkbox control to mark it unchecked.
uncheck-all	Used on UI checkbox control to mark all the checkboxes unchecked specified by a specific xpath.
select	Used on UI radio or combobox/select control to mark it selected or select the items respectively.
select-all	Used on combobox/select control to mark all items selected.
deselect	Used on combobox/select control to mark it deselected.
deselect-all	Used on combobox/select control to mark all items deselected.

location: location tells where to type new text. Supported options/values are: **start**, **end**, **replace**.

autoValueInputs: This is used to generate the **value** automatically based on the provided parameters (given below). This is applicable when **valueType: "auto-gen"** specified.

The following JSON object can be used to generate the value automatically:

{length: 80, maxWordLength: 20, includeAlphabetsLower: true, includeAlphabetsUpper: true, alphabetsLower: "abcdefghijklmnopqrstuvwxyz", alphabetsUpper: true, alphabetsUpper: "ABCDEFGHIJKLMNOPQRSTUVWXYZ", includeNumbers: true, numbers: "1234567890", includeSpecialCharacters: true, specialCharacters: "`~!@#$%^&*()_-+={[]}\\|;:'\",<.>/?", includeNewLine: true, includeWhiteSpaces: true, includeLeadingWhiteSpace: true, includeTrailingWhiteSpace: true}

We can use any combination of the parameter given above to generate the value automatically.

selectingOptionMatchMechanism: The supported option match mechanisms are same as Text Match Mechanisms. These are listed in ***11-Text Match Mechanism*** section.

toPo: This is generally used to select element by drag and drop mechanism using mouse where we can specify target UI page element into **toPO** parameter.

16. Operation name

This is a general argument to any Cucumber step to make the step information more useful from the perspective to understand it easily. Example:

When click on "<page-element-info-from-PO>" page element to "**submit form data on server**".

Then "**form data**" will be "**submitted on server**".

Here where the highlighted information is present is just the informative information to make the step information more meaningful and useful.

C. Arguments related to – Remote directory steps / step definitions

17. **Remote directory**
    This is the fully qualified name of the remote directory that is present on the Remote Machine / Remote Server.

18. **File name match mechanism**
    Filename match mechanism are same as Text Match Mechanisms as specified in ***11-Text Match Mechanism*** section. These are generally used to match the actual file name with expected file name.

19. **Remote Machine name**
    Remote machine name is the unique name of the configured machine present in **test-config/apps-config/<app-name>/remote-machines-config/RemoteMachineConfig.yaml** file. This is also called the target machine name on which user can perform operations like run remote machine commands or SFTP file etc.

    Remote machine is identified by the target machine IP Address and Port. In Cucumber step we do not have to specify the IP Address or Port information of the target server, just use the Remote Machine name to establish connection and perform operation.

20. **Local file download directory**
    This is directory that is generally present in **test-results/** directory. Generally download directory path is configured in **AppDriver.yaml** file for each application under **test-config/apps-conf/** directory.

21. **Keyword delimiter**
    This is generally used to prepare the text from the list elements.
    Example:
    When convert "MY_VAR1" list variable value into plain text using delimiter="," and store into "MY_VAR2" variable.

    Here we are using the comma (,) as delimiter to merge the information of list variable and creating the plain text and storing into new variable so that the new variable value can be used in future steps.

22. **Separators**
    Separators are used to split the text value into parts.
    Example:

    Then split valueOf("MY_VAR_1") variable using "," separator and store index(2) into "MY_VAR_2" variable [ShouldTrim="yes"].

This step will split the text value stored into MY_VAR_1 variable using comma(,) as a separator and the index 2 will get stored in another variable named MY_VAR_2. ShouldTrim=yes will remove the leading and trailing whitespaces of all the splitted parts.

23. **InOrder**
    InOrder value is used during validation of actual value (List type) with expected value (List type) where we would like to validate the actual value is in order to expected value. Like if expected value contains List [5, 9, 7] then when we are retrieving the actual value from UI component or HTTP response then it should be in the same order as expected value [5, 9, 7]. Valid values are: **yes, no.**
    Example:

```
Then verify "HTTP_RESP_VAR" HTTP response contains JSON data with the following expected params information:
| Parameter/JSON Path      | Operator          | Expected Information|
| $.jobTitles              | contains          | {ev: ["Cable operator", "Accountant"], valueType: "string-list", inOrder: "yes", ignoreCase: "no", textMatchMechanism: "exact-match-with-expected-value"} |
```

24. **Should Print**
    Should print is used in cucumber steps to print the contents of variables or file (based on the step). Valid values are: **yes, no**
    Example:

```
Then verify that the downloaded file "sample.docx" contains the following keywords [KeywordDelimiter='["section 1", "section 2", "section 3"]', InOrder="yes", ShouldPrint="yes"]:
```

    The above step will verify the downloaded file (**test-results/downloads/sample.docx**) contents that contains keywords "section 1", "section 2" and "section 3" in order in the downloaded file. Also, it will print the file contents on the console. During the test scenario development time it is useful to see the contents of the file on console window to check the logic is working fine or keywords are present in the file.

D. Arguments related to – Database steps / step definitions

25. **Database Profile Name**
    Database profile names are the names that are configured in application **test-config/apps-config/<app-name>/AppConfig.yaml** file under property **dbProfilesNames**. These profiles are present under directory **test-config/apps-config/<app-name>/database-profiles/.** Database profile contains the connection information to the target database, so that system can communicate with target database to access the database table information and perform CRUD (Create, Read, Update, Delete) operations. STAS uses hibernate to configure any relational database. Like:

```
profileName: sample-db
databaseHandlerClass: org.uitnet.testing.smartfwk.database.SqlDatabaseActionHandler
sessionExpiryDurationInSeconds: 1800

# Define additional properties here under additionalProps element as child element.
additionalProps:
```

```
        hibernate.dialect: org.hibernate.dialect.Oracle10gDialect
        hibernate.connection.driver_class: oracle.jdbc.driver.OracleDriver

        hibernate.connection.url: jdbc:oracle:thin:@db-host-ipaddress:1522:oracle-sid
        hibernate.connection.username: Test
        hibernate.connection.password: Test
hibernate.show_sql: false
```

Using the configured database profile, we can perform any operation using Cucumber step. Example:

```
When get all entries of "DEPT_NAME" column from "DEPARTMENT" table using query below and
store into "DEPT_NAMES_VAR" variable. Target DB Info [AppName="myapp",
DatabaseProfileName="sample-db"]:
  """
 select distinct DEPT_NAME from DEPARTMENT order by DEPT_NAME asc
 """
```

The above step will retrieve all entries of DEPT_NAME column from DEPARTMENT table and will store the retrieved information in DEPT_NAMES_VAR variable. This variable information can be used in further steps to complete the scenario. Note DEPT_NAMES_VAR will store the information as List<String>.

26. **Entity / Table Name**
    Entity name and Table name are the terms that are used interchangeably to represent the same thing. These are nothing but the database table name. Or if you are using the MongoDB then these are called the collection name or entity name.

27. **Param / Column Name**
    Parameter name or column name terms are using in database steps to represent the parameter with the file like (JSON, XML, YAML etc.) and column name in the database table name respectively.

28. **Database query**
    Database query is the query language used to perform CRUD operation on databases like relational databases (Oracle, MySQL, MariaDB, PostgreSQL etc.). In MongoDB terms we use JSON query language to perform CRUD operations.

E. Arguments related to – YAML file steps / step definitions

29. **YAML File Contents**
    YAML is a standard data format generally used to configure the system parameters. You can refer any online documentation for YAML file format. To read YAML file contents, we can use the following step, this step store the YAML contents as YAML object:

```
When read "test-data/sample.yaml" YAML file contents and store into "YAML_DATA_VAR"
variable.
```

30. **YAML Object**
    YAML object is a converted object that contains the parsed information of YAML contents in a JSON java object (DocumentContext class). Over YAML object we can use JSON Path to retrieve and modify the JSON file contents.

    ```
    When read "test-data/sample.yaml" YAML file contents and store into "YAML_DATA_VAR" variable.
    Then "the file contents" will be "stored in YAML_DATA_VAR variable".
    ```

    Here YAML_DATA_VAR contains the YAML object. From this YAML object we can read any parameter value using the step:

    ```
    When read "$.param1" parameter value from YAML object [YAMLObjRefVariable="YAML_DATA_VAR"] and store into "PARAM1_VALUE_VAR" variable.
    ```

    The step above can read the **param1** parameter value from YAML Object using JSON Path mechanism.

31. **YAML object reference variable**
    It is reference variable of YAML object that stored that YAML data in JSON format. So that we can read the parameter information using JSON Path mechanism.

32. **Parameter Path / YAML Path in YAML Object / YAML File Contents**
    Parameter Path or YAML Path are nothing but the JSON path using that we can access the value from YAML object / JSON object. For more details on JSON Path, please refer https://github.com/json-path/JsonPath link. In cucumber step it can be specified in the format given below:

    {path: "$.department[*].name", valueType: "string-list"}

    The above JSON / YAML path will return department names as the List<String> data object.

    OR we can directly specify the JSON path without JSON format like:

    $.department[*].name

33. **Assignment of new value to a parameter into YAML Object**
    We can also modify the parameter information in YAML object using JSON path mechanism. Example:

    ```
    When update the following parameters values into YAML object [YAMLObjRefVariable="YAML_DATA_VAR"]:

     | Parameter Path                          | New Value      |
     | $.name                                  | David          |
     | { path: "$.jobTitles", valueType: "string-list" } | ["Accountant", "Operator"] |
    ```

Using the above step, we can modify the YAML_DATA_VAR variable contents. Here we modified 2 parameters: name, jobTitles

F. Arguments related to – JSON file steps / step definitions

34. JSON File Contents

JSON is a standard data format generally used to configure the system parameters or used in REST API for data exchange. You can refer any online documentation for JSON file format. To read JSON file contents, we can use the following step, this step store the JSON contents as JSON object:

```
When read "test-data/sample.yaml" JSON file contents and store into "JSON_DATA_VAR" variable.
```

35. JSON Object

JSON object is a converted object that contains the parsed information of JSON contents in a JSON java object (DocumentContext class). Over JSON object we can use JSON Path to retrieve and modify the JSON file contents.

```
When read "test-data/sample.json" JSON file contents and store into "JSON_DATA_VAR" variable.
Then "the file contents" will be "stored in JSON_DATA_VAR variable".
```

Here JSON_DATA_VAR contains the JSON object. From this JSON object we can read any parameter value using the step:

```
When read "$.param1" parameter value from JSON object
[JSONObjRefVariable="JSON_DATA_VAR"] and store into "PARAM1_VALUE_VAR" variable.
```

The step above can read the **param1** parameter value from JSON Object using JSON Path mechanism.

36. JSON object reference variable

It is reference variable of JSON object that stores data in JSON format. So that we can read the parameter information using JSON Path mechanism.

37. Parameter Path / JSON Path in JSON Object / JSON File Contents

Parameter Path or JSON Path is nothing but the JSON path using that we can access the value from JSON object. For more details on JSON Path, please refer https://github.com/json-path/JsonPath link. In cucumber step it can be specified in the format given below:

{path: "$.department[*].name", valueType: "string-list"}

The above JSON path will return department names as the List<String> data object.

OR we can directly specify the JSON path without JSON format like:

$.department[*].name

38. **Assignment of new value to a parameter into JSON Object**
    We can also modify the parameter information in JSON object using JSON path mechanism. Example:

```
When update the following parameters values into JSON object
[JSONObjRefVariable="JSON_DATA_VAR"]:

 | Parameter Path                          | New Value       |
 | $.name                                  | David           |
 | { path: "$.jobTitles", valueType: "string-list" } | ["Accountant", "Operator"] |
```

    Using the above step, we can modify the JSON_DATA_VAR variable contents. Here we modified 2 parameters: name, jobTitles

G. Arguments related to – XML file steps / step definitions

39. **XML File Contents**
    XML is a standard data format generally used to configure the system parameters or used in REST API for data exchange. You can refer any online documentation for XML file format. To read XML file contents, we can use the following step, this step store the XML contents as XML object:

```
When read "test-data/sample.xml" XML file contents and store into "XML_DATA_VAR" variable.
```

40. **XML Object**
    XML object is a converted object that contains the parsed information of XML contents in a XML java object (Document class). Over XML object we can use XML Path (XPATH) to retrieve and modify the XML file contents.

```
When read "test-data/sample.xml" XML file contents and store into "XML_DATA_VAR" variable.
Then "the file contents" will be "stored in XML_DATA_VAR variable".
```

    Here XML_DATA_VAR contains the XML object. From this XML object we can read any parameter value using the step:

```
When read "//department/@param1" parameter value from XML object
[XMLObjRefVariable="XML_DATA_VAR"] and store into "PARAM1_VALUE_VAR" variable.
```

    The step above can read the **param1** parameter value pf department element from XML Object using XML Path mechanism.

41. **XML object reference variable**
    It is reference variable of XML object that stores data in XML format. So that we can read the parameter information using XPATH / XML Path mechanism.

42. **Parameter Path / XML Path / XPATH in XML Object / XML File Contents**

Parameter Path or XML Path is nothing but the XPATH using that we can access the value from XML object. For more details on XML Path, please refer https://www.w3.org/TR/1999/REC-xpath-19991116/ link. In cucumber step it can be specified in the format given below:

{path: "//department/@name", valueType: "string-list"}

The above XML path will return department names as the List<String> data object.

OR we can directly specify the XML path without JSON format like:

//department/@name

43. **Assignment of new value to a parameter into XML Object**
    We can also modify the parameter information in XML object using XPATH mechanism. Currently this feature is not supported in default cucumber step definitions but you can write your specific custom step definition to update XML document content. Probably in future this may get supported.

H. Arguments related to – Excel file steps / step definitions

STAS by default support tabular data to be read from excel sheets.

44. **Read Excel Sheet data as Table**
    STAS supports reading of the excel sheet information and that information can be stored in variable so that It can be used in future steps of the scenario.
    Example:

    When read "Department" sheet data of "test-data/department.xlsx" excel file into tabular form and store into "DEPARTMENT_TABLE_VAR" excelsheet variable.

    The step above will read on **Department** sheet data that is present in **department.xlsx** file. Also it will store the read data into DEPARTMENT_TABLE_VAR variable. That can be used into future steps to complete the scenario.

45. **Read Excel Sheet Column data as List<String>**
    STAS support default step definitions that can be used to read any column data as List<String> from excel file. Example:
    When read "name" column data from "DEPARTMENT_TABLE_VAR" excelsheet variable and store into "DEPARTMENT_NAMES_VAR" variable.

    The step above can read the name column data from the DEPARTMENT_TABLE_VAR variable value. NOTE: DEPARTMENT_TABLE_VAR variable value is prepared by reading the excel file information. It also stores the column data information into DEPARTMENT_NAMES_VAR variable which can be used in future steps to complete the scenario.

46. Read particular row of Excel Sheet Column data as String

We can read any record of a particular column (From the Excel sheet data). Example:

When read **2** row of "**name**" column data from "DEPARTMENT_TABLE_VAR" excelsheet variable and store into "2ND_REC_OF_NAME_COL_VAR" variable.

Here, this step will read the 2nd record of **name** column from the information that is stored in excel sheet variable.

I. Arguments related to – CSV file steps / step definitions

STAS by default support tabular data to be read from CSV (Comma Separated Value) file.

47. Read CSV file data as Table

STAS supports reading of the CSV file information and that information can be stored in variable so that It can be used in future steps of the scenario.
Example:

When read "test-data/department.csv" CSV file into tabular form and store into "DEPARTMENT_TABLE_VAR" CSV variable.

The step above will read **department.csv** file and it will store the read data into DEPARTMENT_TABLE_VAR variable. That can be used into future steps to complete the scenario.

Also, we can use the following step to read customized CSV file.

When read "test-data/department.csv" CSV file into tabular form and store into "DEPARTMENT_TABLE_VAR" CSV variable [DataDelimiter="|", QuoteChar="'", TrimData="yes"].

48. Read CSV Column data as List<String>

STAS support default step definitions that can be used to read any column data as List<String> from CSV file. Example:

When read "name" column data from "DEPARTMENT_TABLE_VAR" CSV variable and store into "DEPARTMENT_NAMES_VAR" variable.

The step above can read the name column data from the DEPARTMENT_TABLE_VAR variable value. NOTE: DEPARTMENT_TABLE_VAR variable value is prepared by reading the CSV file information. It also stores the column data information into DEPARTMENT_NAMES_VAR variable which can be used in future steps to complete the scenario.

49. Read particular row of CSV file Column data as String

We can read any record of a particular column (From the CSV data). Example:

When read **2** row of "**name**" column data from "DEPARTMENT_TABLE_VAR" CSV variable and store into "2ND_REC_OF_NAME_COL_VAR" variable.

Here, this step will read the 2nd record of **name** column from the information that is stored in CSV variable.

J. Arguments related to – API testing steps / step definitions

API Testing automation is nothing but calling the HTTP service and verify the response data of the HTTP service. We also call REST Service Automation.

50. API Target Server

API Target Server is a server to which STAS system can make HTTP connection and perform the operations like HTTP GET, PUT, POST, DELETE etc.

API target server name is configured in **test-config/apps-config/<app-name>/api-configs/ApiConfig.yaml** file for the specific application. The following information can be configured in ApiConfig.yaml file for the target server:

```
- name:
  baseURL:
  # Class name that extends AbstractApiActionHandler class
  actionHandlerClass:
  sessionExpiryDurationInSeconds: 1800
  # Additional properties for target server (Define under additionalProps as child property)
  additionalProps:
```

Example:

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR", RespVar="RESP_VAR"]:
    """
    test-data/sample.json
    """
```

In the above step, target server is the server where system will connect to perform HTTP POST call.

51. API Target URL

Target URL is the relative URL that can be used to make HTTP call on target server.
Example:

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="department/search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR", RespVar="RESP_VAR"]:
    """
    test-data/sample.json
```

```
"""
```

In the example above, system will make HTTP POST call on Target URL **department/search-by-criteria**. System will automatically attach the base URL as prefix with target URL.

52. **API User Profile**

User profiles are the credentials that are used for authentication and authorization purpose so that the API call can be successful. Generally, it contains username, password and user roles or user-group related information.
Example:

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="department/search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR", RespVar="RESP_VAR"]:
    """
    test-data/sample.json
    """
```

Here, **SampleUserProfile** is used for authentication and authorization purpose. The corresponding actual file **SampleUserProfile.yaml** for this user profile is present under **test-config/apps-config/<app-name>/user-profile/** directory.

53. **API Content Type**

Content type in API is the header information that tells server what kind of contents are present in payload / body of HTTP call.
Example:

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="department/search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR", RespVar="RESP_VAR"]:
    """
    test-data/sample.json
    """
```

Here, Content type is application/json, means we have to send json data on HTTP POST API.

54. **API Accepts**

Accept in API is the header information that tells server what type of contents are acceptable in HTTP Response payload / body .
Example:

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="department/search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR",
RespVar="RESP_VAR"]:
    """
```

```
test-data/sample.json
"""
```

Here, application/json type of contents are acceptable in HTTP response payload / body.

55. **API Request Variable**
    Request variable is the variable that stores the whole request information that can be used in future steps.

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="department/search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR",
RespVar="RESP_VAR"]:
    """
    test-data/sample.json
    """
```

Here, REQUEST_VAR is the request variable name.

56. API Response Variable
    Response variable is the variable that stores the whole response information (received from server) that can be used in future steps.

```
When make HTTP POST request using the contents of the following template file on target server
[AppName="myapp",
TargetServer="test-server", TargetURL="department/search-by-criteria"] using
[UserProfile="SampleUserProfile"] with header info [ContentType="application/json",
Accept="application/json"] and variable info [ReqVar="REQUEST_VAR",
RespVar="RESP_VAR"]:
    """
    test-data/sample.json
    """
```

Here, RESP_VAR is the response variable name.

57. **API HTTP Response**
    HTTP Response is the response that is stored in a variable. This variable value can be used in future steps to complete the scenario like to verify the HTTP response information.
    Example:

```
When verify "HTTP_RESP_VAR" HTTP response contains JSON data with the following expected
params information:
 | Parameter/JSON Path  | Operator    | Expected Information      |
 | $.name               | =           | John Hopkins              |
 | $.jobTitles          | contains    | {ev: ["Cable operator", "Accountant"], valueType: "string-list",
inOrder: "yes", ignoreCase: "no", textMatchMechanism: "exactMatchWithExpectedValue"} |
```

In the step above, we can verify the HTTP response information that is stored in HTTP_RESP_VAR.

58. HTTP Response Code
    There are many HTTP Response code that server returns based on the error. Some of the most useful response codes information are given below:

HTTP Response Code	Description
2XX Success	
200	OK
201	Created
202	Accepted
204	No Content
3XX Redirection	
300	Multiple Choices
301	Moved Permanently
302	Moved Temporarily
4XX Client Errors	
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not found (URL not found)
5XX Server Errors	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

6.6 What if any step definition is not present to complete scenario implementation?

This situation may come, in that case you would have two options:

1. **Raise an enhancement** on https://github.com/mkrishna4u/smart-testauto-cucumber-defaults-en/issues link. The Smart TestAuto team will try to add new generic step definitions that can be reused. NOTE: Time for completion of adding new step definition may not be committed as this is a community project. If anyone would like to join to add new step definitions they would be more welcome to join our team.
2. You can create your step definition file in **src/test/java/stepdefs/** directory and implement your step definitions using smart testauto apis. NOTE: If you are writing your custom step or step definition then you should prefix each step or step definition with **[C]** text to make it independent. Else if framework will add new step and step definition then your custom step or step definitions may collide with framework step definition and may result in duplicate step definition error. For example custom step definition should like this:

```
Given [C} your step here.
When [C] your step here.
Then [C] your step here.
```

```
And [C] your step here.
```

7 Mobile Application Testing Automation

STAS supports the mobile native and web applications testing automation using Appium tool. STAS tool just provide the platform to write the testcases / test-scenario easily and then that can be executed on mobile devices / emulators using Appium Server. It supports both **Android** and **iOS** mobile testing automation. Scenario writing is same as specified in ***6-How to write test scenarios using STAS?*** section.

7.1 STAS configuration for running test cases for mobile applications

To run the testcases on mobile devices using STAS, we have to configure following things:

For more details on **Appium** tool, please refer https://appium.io/docs/en/writing-running-appium/caps/index.html website. This document covers only the high level information on Appium tool like how this tool can be configured in STAS for mobile testing automation.

NOTE: For mobile testing, we should install **nodejs** on your local machine and keep the nodejs directory path in system PATH variable. So that node and npm command will be available to command prompt.

1. **Install Appium Server**
   If you do not have any Appium server configured to perform mobile testing automation. Then you can install Appium server for quick configuration on your local machine using the following command under STAS project:

   > smart-studio --install-appium-server

   It will install Appium server in **appium-server/** directory.

   If you already have Appium server installed on any remote machine then you can configure the **test-config/apps-config/<app-name>/driver-configs/AppDriver.yaml** file. To see the basic default parameter's information for your testing platform please refer https://github.com/mkrishna4u/smart-testauto-framework/tree/main/src/main/resources/org/uitnet/testing/smartfwk/resources/sample-test-config/app-drivers URL where it contains platform specific, web browser specific and application specific AppDriver.yaml configuration file that can be copied into STAS project under **test-config/apps-config/<app-name>/driver-configs/** directory to integrate STAS tool with Appium. To know the mobile specific parameter information (like desired capabilities), please refer Appium documentation.

2. **Start Appium Server**
   To start the locally installed Appium server, use the following command:

   > smart-studio start-appium-server

   This will start Appium server at 4723 port. The following message will be displayed on successful start.

   Appium REST http interface listener started on 0.0.0.0:4723

3. **Install Appium Inspector**
   This tool is used to inspect the elements for the mobile application to create page elements / page objects that we would like to do the automation. Appium inspector is available for download from URL: https://github.com/appium/appium-inspector/releases

   Appium Inspector can be connected with Appium Server on http://127.0.0.1:4723/wd/hub URL. To connect to the specific mobile application, the proper desired capabilities should be configured in Appium Inspector.

4. **Attach Mobile Device With laptop, desktop or server machine**
   If you would like to do testing automation with real device, attach your mobile device using USB cable to your machine. Make sure in your mobile, developer mode is enabled else Appium server will not recognize your device.

   Configure the correct URL (Default: http://127.0.0.1:4723/wd/hub) and desired capabilities in **AppDriver.yaml** file of your configured application in STAS project, so that STAS tool can make connection to the Appium Server to perform operation on mobile application.

5. **Configure mobile emulator for testing automation**
   Mobile emulators are the software programs that simulates the whole mobile device on computer that can be configured with Appium and STAS to perform testing automation of mobile applications (web and native).

6. **Run testcases**

   Running the cucumber testcases for mobile automation using STAS is same as described in ***4.5 - Execution of testcases / test-scenarios using STAS*** section. Only important thing is that Appium Server should be up and running before starting the testing automation.