Inventi Automation Framework Docs

Table of Contents

Introduction

Welcome to the official documentation for the **Inventi Automation Framework** - Java boilerplate for easier multi-platform test automation!

Basic Information

Inventi Automation Framework is an ecosystem of libraries and design patterns that allows you to test various kinds and parts of applications with ease. The framework's main features are following:

- Testing of front-end of web applications
  - Using **XPATH** locators to locate elements
  - Easy to understand web page object structure following the **Page-Object pattern**
- Testing of back-end implementations (**REST**)
- Easy to understand API object structure
- **Multi-language** support
- Simultaneous testing of different applications
  - Easily implement custom solution for your application type
- Better reporting and structure with **Allure reports**
- **Open-sourced**

Download

- Latest release
- Inventi Automation Framework on GitLab

Used Libraries

The framework's project is a Maven configuration using multiple third-party libraries for compiling, testing and in other phases.For more details about particular software used please visit appropriate POM files in the project.Below is provided basic information about software used along with developer's webpage and license type under which given software is supplied.

- Allure Framework (Apache License 2.0)
- AssertJ (Apache License 2.0)
- Apache Commons (Apache License 2.0)
- Apache Log4J (Apache License 2.0)
- AspectJ Weaver (Eclipse Public License 1.0)
- Google Guava (Apache License 2.0)
- Hibernate Validator (Apache License 2.0)
- Jackson (Apache License 2.0)
- Lombok (MIT License)
- Maven SureFire Plugin (Apache License 2.0)
- REST Assured (Apache License 2.0)
- Selenium WebDriver (Apache License 2.0)
- TestNG (Apache License 2.0)
- WebDriverManager (Apache License 2.0)

Licensing

The **Inventi Automation Framework** solution (`core-automation-framework` Maven module artifact) is distributed under **Apache License 2.0**. See attached NOTICE and LICENSE files located at the `core-automation-framework` folder for more information.

Other artifacts supplied along this project are meant for framework's usage demonstration purposes.

Support

In case of need for providing a closer look at the implementation, bug resolving or any other information, please contact developers at Inventi GitLab Repository.

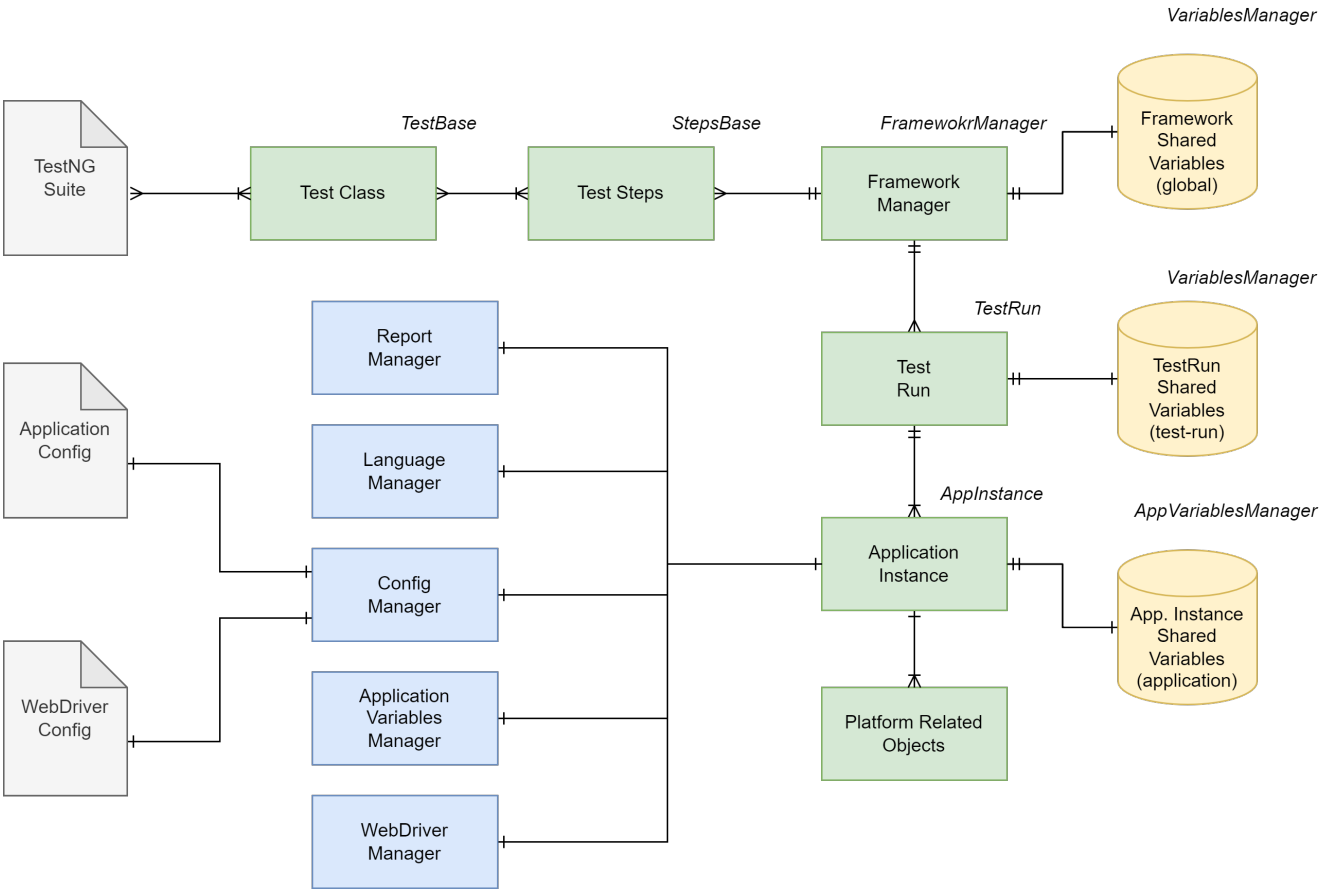
Structure

For easier development of automated tests, the framework follows possibilities of Java's object-oriented programming to abstract different levels of application "objects" in order to provide easier access and use of these application parts.

1. Framework Specific Structure

2. Platform-Specific Structure
3. Fluent Interface Implementation

Framework Structure



Framework Configuration Objects

Bellow you can find description related to the objects from the

TestNG Suite

Basic TestNG XML test suite file. The structure and use can be found at the TestNG official documentation.

Sample TextNG XML test suite:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
<suite name="Sample Test Set" parallel="classes">
    <parameter name="parameter_name" value="parameter_value">
    <test name="Some Test" thread-count="1">
        <classes>
            <class name="cz.inventi.qa.sampleapplication.tests.SampleTest"/>
        </classes>
    </test>
```

```
</suite> <!-- Suite -->
```

Note: when defining the test suite, please mind the specifics of parallelization.

Application Config

Application config file is supposed to provide application-bound specifics, mainly URL, environments, and other necessary information. This config file should have YAML structure.

Application Configuration File Structure

Now let's get on to the creation of the file. The required structure of the file differs according to the application platform. Create a `appConfig.yml` file in the configuration folder mentioned above with following structure:

appConfig.yml file structure

```
application:
  <application type>:
    environments:
      <environment name>: "<value/url>"
```

The application name should correspond to the name defined for the application resources folder.

appConfig.yml file structure for web application

```
application:
  web:
    environments:
      PROD: "https://sampleapplication.com"
```

appConfig.yml file structure for API application

```
application:
  api:
    environments:
      PROD: "https://sampleapplication.com"
    relaxedHttpsValidation: true
```

WebDriver Config

Specifically for web-based applications, WebDriver approach is supported to allow front-end testing. Similarly to the Application Config file, the WebDriver config file provides configuration values for the WebDriver, like wait timeouts, browser capabilities, waiting mode, and other specifics. More information about this file can be found in the Project Setup chapter - WebDriver Config File.

WebDriver Configuration File Structure

webDriverConfig.yml file structure

```
generalSettings:
  windowSize:
    sizeType: maximized
  customTargetPath: "<custom path to the WebDriver executables folder>"
  takeScreenshots: <true/false>
  wait:
    waitsAutomatically: <true/false>
    timeouts:
      min: <amount of milliseconds to wait>
      mid: <amount of milliseconds to wait>
      max: <amount of milliseconds to wait>

chrome:
  arguments: [ "<ChromeDriver arguments>" ]
```

Framework Abstraction Objects

Test Class

The **Test Class** is designed to contain all the "tests" - primarily *test methods* annotated with the TestNG's `@Test` annotation.It should always extend the `TestBase` class which provides it some core functionalities related to the framework in order to properly run all the tests.In the *test methods*, **Test Steps** class and its methods should be addressed in order to follow a real-world test scenario.

Test Class can contain *test setup steps* (`testSetup()`), *test steps* and *test teardown steps* (`tearDown()`) in separate methods. *Assertions* can be used in test methods as well - however, it is recommended to use primarily methods defined in the **Test Steps** class in order to maintain simple and clear test methods.

To use the *Test Steps* class, you should initialize it within a separate dedicated `initSteps()` method which is executed before all the **Test Class** is instantiated.

Sample of a Test class:

```
@Epic("Provide Sample Structure for Custom Module")
public class SampleTest extends TestBase {
    private SampleApiSteps sampleApiSteps;
    private SampleWebSteps sampleWebSteps;

    @BeforeClass
    @Override
    public void initSteps() {
        sampleApiSteps = new SampleApiSteps();
        sampleWebSteps = new SampleWebSteps();
    }
```

```
    @BeforeClass
    @Override
    public void setUp() {
        // Setup steps belong here
    }

    @Test(description = "This Is a Sample Test Method")
    public void someTestMethod() {
        // Write some test here
        // API test example:
        ArticlesResultsListDto articles = sampleApiSteps
                .retrieveAllArticles()
                .as(ArticlesResultsListDto.class);
        Assert.assertEquals(
            articles.size(),
            4,
            "There are 4 articles in the response"
        );
        // Web test example:
        sampleWebSteps
                .checkContactFormIsDisplayed()
                .fillContactForm("joedoe@email.com", "Hello, I really like your
products!")
                .sendContactForm()
                .leaveComponent();
    }

    @AfterClass
    @Override
    public void tearDown() {
        // TearDown steps go here
    }
}
```

Test Steps

The **Test Steps** class serves as a bridge to your application-related objects and is supposed to provide simpler look to the **Test Class** with test methods, as well as follows the DRY principle by allowing you to define *step methods* with repeatable steps connected to given application anywhere in your tests.

Step methods are defined by using the `@Step` annotation from the Allure Framework. All the Test Steps classes should extend `StepsBase` class to gain access to application-related methods.

Sample of a Test Steps class:

```
public class ContactFormSteps extends StepsBase {
    // Create application instance and obtain access to it
    private final HomePage homePage = getWebAppInstanceOf(HomePage.class);
```

```
    @Step("Fill Contact Form ({email})")
    public ContactFormSteps fillContactForm(String email, String message) {
        ContactForm contactForm = homePage.getContactForm();
        contactForm
            .fillEmail(email)
            .fillMessage(message);
        Assert.assertEquals(
                contactForm.getEmailInput().getText(),
                email,
                "Email was filled correctly"
        );
        Assert.assertEquals(
                contactForm.getMessageInput().getText(),
                message,
                "Message was filled correctly"
        );
        return this; // Use the fluent interface
    }

    @Step("Send Contact Form")
    public ContactFormSteps sendContactForm() {
        ...
    }
}
```

Framework Manager

One object rules them all - that is the **Framework Manager**.With a singleton pattern, this object is predefined to manage all the **Test Runs** and global test variables, and therefore it serves as a gate to all the related objects for the **Test Run**.You can access its instance anywhere in the code by calling the `FrameworkManager` class instance.Its instance is first created with **Test Classes**.

Test Run

In order to preserve possibilities of parallelization, the **Test Run** object was created (`TestRun` class).The **Test Run** instance represents a running TestNG "test class" (above-mentioned **Test Class**) with all related applications that were started in order to perform given test.

The **Test Run** collects and manages all the application instances (`AppInstance`) for various platforms.Along with that, it also saves all the failed soft-assertions via `SoftAssertCollector` class instance, that is bound to the test run.

Application Instance

Application Instance contains all the information about current platform-dependent application.At one moment, **only one instance of the same application can be run**.However, it is allowed to run **multiple different applications at the same time**.

This instance is represented by the `AppInstance` class.

Platform Related Objects

This layer of abstraction covers objects related to the platform itself. Please follow to the next subchapter dedicated to each of the platform-specific objects.

Framework Variables Containers

If you are in a need of sharing data between **Test Runs** or for given **Application Instance** (i.e. to reuse data created in the first setup script without the need to run it again), there is a possibility to do so by using the "variable containers". These containers are currently on two levels:

Shared Variables

The automation framework provides possibilities to store data using Java's Object representation and String as an identifier key. Currently, maintaining the variables is taken care of by the `VariablesManager` (or `AppVariablesManager` respectively) class instance, that can be found on three levels - "global", "test-run" and "application".

Global Variables

Representation for the "global variables" can be found in the class of `FrameworkManager`, where you can access `VariablesManager` instance using the method call `getGlobalVariables()`.

Variables manager located at this level allows you to share data between TestNG class runs, as there is always only one instance of `FrameworkManager` running.

TestRun Variables

TestRun-specific variables are located in the `TestRun` class instance, available through `getTestRunVariables()` method call. The manager for these variables is the `VariablesManager`.

These variables can be used to share data between multiple applications in the current `TestRun` instance.

Application Variables

The application-specific variables can be saved in the `AppVariablesManager` class instance, which can be accessed from the AppInstance of given test run by `getTestVariablesManager()` method call. This variables manager is supposed to hold mainly app-specific variables and parameters, or custom variables, eventually. Therefore, depending on the type of current application, you can access API or web specific variables by retrieving `ApiAppVariables` or `WebAppVaribales` class from the `AppVariablesManager`.

Platform-Specific Structure

At the moment, the framework supports testing of API and web (frontend) applications.Please find description to platform-specific structure bellow:

1. API Component Structure
2. Web Component Structure

ApiObject

`ApiObject` is a central piece for all API related objects. It is used to hold information using mainly the `AOProps`. Similarly to the Page Object pattern in the web front-end testing with their Xpath elements, API testing abstracts endpoints in a similar manner.

AOProps

Properties object contains information regrading i.e. parent `ApiObject`, current `ApiAppInstance`, endpoint authentication methods, authentication parameters and other aspects, that can be used while making requests calling an endpoint of the API. `AOProps` are present in every `ApiObject` and `ApiObject` is a parent for every object related to API.

WebComponent

`WebComponent` stands for an element that can be more complex, like a table, menu, sidebar, etc. It can not only contain another `WebComponent`, but also a `WebElement`. It provides you with possibilities of adding additional functions and methods to work with these objects.

Sample of WebComponent declaration in a WebPage class

```
SidePanel<T> sidePanel;
```

Sample of WebComponent class implementation

```
@FindElement(xpath = "//aside")
public class SidePanel<T extends WebObject> extends WebComponent<T> {

    @FindElement(xpath = "/h2")
    WebElement<SidePanel<T>> panelTitle;

    public SidePanel(WOProps props) {
        super(props);
    }
}
```

WebComponent Annotations

- @FindElement
- @NoParent

Fluent Interface Implementation

Mind that the framework uses the Fluent Interface technique to navigate through the test.This stands behind the use of Java's generic types that can be seen in the code samples bellow.In this case, Fluent Interface should allow you to see and use options that are available in the current state of tested application.

Generic types are used in order to preserve possibility to return from various elements, like

`WebComponent` or `WebElement` that are described further.

Setup

Starting test automation with the framework takes two main steps:

1. Installation
2. Project Setup

Installation

All of these parts need to be installed and **correctly set in the system/user environment variables list**:

- JDK SE (15+)
- Maven (3.5+)
- Allure Framework (2.14+)

After downloading the framework, unzip downloaded archive to a separate folder and run following command at the root of this folder:

```
mvn clean install -DskipTests
```

All the necessary Maven packages will be downloaded (beware if you use custom Maven settings). Installation will run unit tests for the framework by default. If you want to skip these unit tests, add `-DskipTests` parameter to the above-mentioned command.

That's it, the framework is now ready to use!

Recommended Development Tool

You can use any of your favourite IDEs to work with the framework. As a recommended IDE for the development process we encourage use of the **IntelliJ IDEA** - either a Community or Ultimate version.

Recommended IntelliJ Plugins

- Lombok

Setting-up a Project

To start creating automated tests for your application, only two steps are necessary:

1. Create a Maven Module for Your Application
2. Create Application Configuration Files

1. Create a Maven Module for Your Application

Create a new Maven module in your favourite IDE - adjust the package naming to your needs.Recommended name of the module is in a lower-case format with dash instead of spaces - **this name will be used later in the code and in folder names for application resources!**

Child module should contain a Maven POM configuration file that will be linked to the `core-automation-framework` module and have a parent project defined.

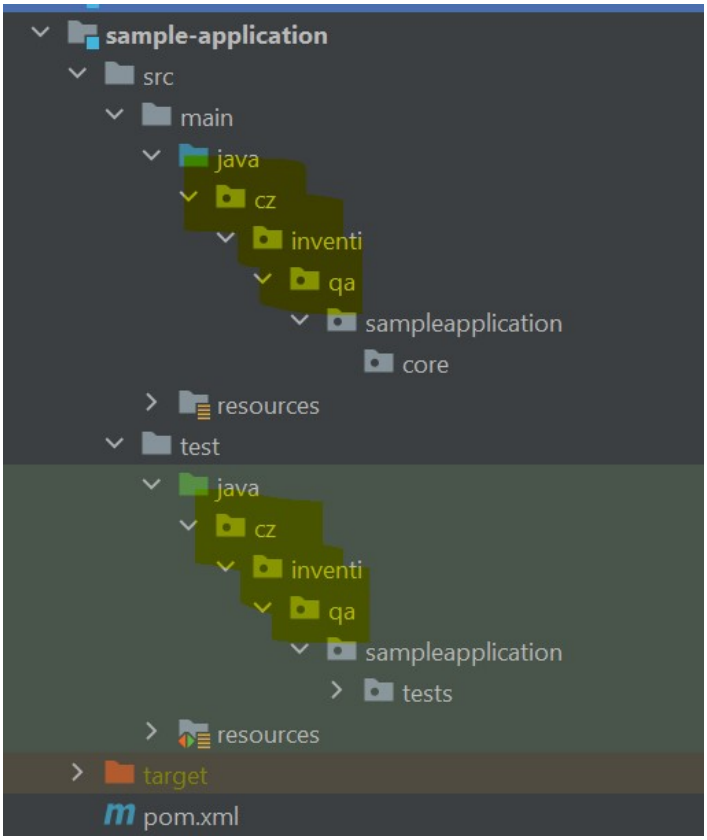
Example of Maven's child module POM file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <artifactId>sample-application</artifactId> <!-- Your child module name. -->

    <parent>
        <artifactId>automation-framework</artifactId> <!-- Here define your project
name. -->
        <groupId>cz.inventi.qa</groupId> <!-- Define group ID if changed. -->
        <version>1.0-SNAPSHOT</version> <!-- Don't forget to change the version if
necessary. -->
    </parent>

    <dependencies>
        <!-- Linking the automation framework. -->
        <dependency>
            <groupId>cz.inventi.qa</groupId>
            <artifactId>core-automation-framework</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

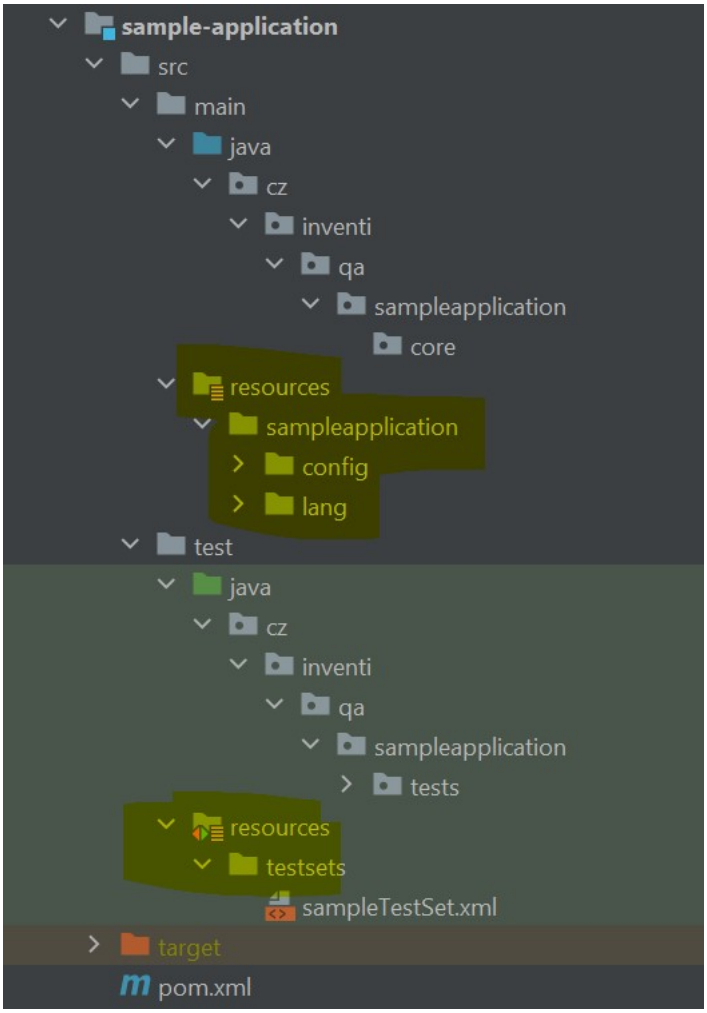
Depending on your needs, it is recommended to create Java packages according to your group ID that was defined in the root project POM and is then derived in the children's POMs.For the POM file specified above, the structure of the child module could look like this:



2. Create Application-related Configuration Files

Typically, applications under the test can have their own specifics, like a URL (for cloud applications with backend and frontend implementations), paths to executables (desktop applications) or APK files (Android mobile application).These platform-dependent specifics can be defined in configuration files that are mandatory to be provided at the input to the framework.

All the configuration files should be saved in the application resources folder: `<your module name>/src/java/main/resources/<your application name>/config`.



Application Configuration File

Such specifics can be saved in framework configuration files. By default, framework requires at least one main configuration file - the application settings file `appConfig.yml`.

If you want to supply your custom YAML configuration file or want to just simply rename the file, you can supply it as an input parameter:

Using command line

```
... -DappConfig=myOwnAppConfig.yml ---
```

Using TestNG suite

```
...
    <parameter name="appConfig" value="myOwnAppConfig.yml"/>
...
```

Note that this file should be located as well inside the application's resources folder.

Now let's get on to the creation of the file.The required structure of the file differs according to the application platform.Create a `appConfig.yml` file in the configuration folder mentioned above.

Application Configuration File Structure

Now let's get on to the creation of the file. The required structure of the file differs according to the application platform. Create a `appConfig.yml` file in the configuration folder mentioned above with following structure:

appConfig.yml file structure

```
application:
  <application type>:
    environments:
      <environment name>: "<value/url>"
```

The application name should correspond to the name defined for the application resources folder.

appConfig.yml file structure for web application

```
application:
  web:
    environments:
      PROD: "https://sampleapplication.com"
```

appConfig.yml file structure for API application

```
application:
  api:
    environments:
      PROD: "https://sampleapplication.com"
    relaxedHttpsValidation: true
```

WebDriver Configuration File

If you plan to test front-end of a web application, you need to define also a WebDriver configuration file along with the application config file.This file goes to the same folder and it's default name is `webDriverConfig.yml`.

Currently, only the ChromeDriver implementation is supported, however an extension for other browsers can be easily provided using the WebDriverManager.

WebDriver Configuration File Structure

webDriverConfig.yml file structure

```
generalSettings:
  windowSize:
    sizeType: maximized
  customTargetPath: "<custom path to the WebDriver executables folder>"
  takeScreenshots: <true/false>
  wait:
    waitsAutomatically: <true/false>
```

```
    timeouts:
      min: <amount of milliseconds to wait>
      mid: <amount of milliseconds to wait>
      max: <amount of milliseconds to wait>

chrome:
  arguments: [ "<ChromeDriver arguments>" ]
```

Similarly to the application config file, if you want to supply your custom YAML configuration file or want to just simply rename the file, you can supply it as an input parameter:

Using command line

```
... -DwebDriverConfig=myOwnWebDriverConfig.yml ---
```

Using TestNG suite

```
...
<parameter name="webDriverConfig" value="myOwnWebDriverConfig.yml"/>
...
```

That's it, now you can start developing your first test!

After you finished setting up the project, you can start developing your first tests.

Test Development

Creating a Test

In order to create a test and have it set up for the next test run, you should create (or update) three of the following items:

1. Create Test Class
2. Create Test Steps
3. Create Test Suite

Create Your First Test Class

1. Create Class

Browse into your application Maven module and navigate to the test packages. Here, you can define your own test classes according to your test structure defined by packages. Name your class according to your needs … preferred naming convention is to provide a *Test* (or *Tests*) keyword in each of the class names (i.e. *BasicComponentTests.java*).

Let us create a sample test class. Remember that each of the test classes creates a `TestRun` instance

in the automation framework while being run.

2. Setup Test Class Methods

After you created the Java class itself, it is now time to write methods that will define the test-run structure - the test setup, test methods, and test teardown.

Please bellow provided test class structure sample. The description of each of the parts will be provided in following subchapters.

Note that basic structure methods all use some kind of TestNG's annotations in order to preserve the correct test run behaviour. You can alter these annotations according to your needs (i.e. in case of test class inheritance), however, it is important that the steps classes are initialized before any of the test methods is run (as they provide reach/connection to the application under test).

Sample structure of a test class

```
@Epic("Epic description")
@Story("Story description")
public class SomeTest extends TestBase {
    private final SomeStepsClass someStepsClass;

    // Steps Class Initialization
    @BeforeClass
    @Override
    public void initSteps() {
        someStepsClass = new SomeStepsClass();
    }

    // Class Setup
    @BeforeClass
    @Override
    public void setUp() {
        // Prepare test data ...
    }

    // Test Methods in the Test Class
    @Test(description = "First Test Method")
    public void firstTestMethod() {
        someStepsClass
            .step1()
            .step2()
            .assertSomething();
    }

    @Test(
            description = "Second Test Method",
            dependsOnTestMethods = "firstTestMethod"
    )
    public void secondTestMethod() {
        someStepsClass
```

```
            .anotherStep1()
            .anotherStep2()
            .assertSomethingElse();
    }

    // Test Class Teardown Methods
    @AfterClass(alwaysRun = true)
    @Override
    public void tearDown() {
        // Delete prepared test data ...
    }
}
```

Test class should always extend the `TestBase` class in order to preserve methods provided by the test framework.

Class Setup

The class setup method is supposed to help with actions that need to be done before the test - this can be test data preparation, infrastructure setup, etc. In order to run this method before all the test methods, a `@BeforeClass` annotation has to be provided.

Steps Class Initialization

As the test steps class is declared at the beginning of test class, it is necessary to also instantiate it - that is the purpose of the `initSteps()` method.

3. Test Methods in the Test Class

Test classes are annotated with the the TestNG's `@Test` annotation. The `description` value is then used to stand for the test name in Allure Reports.

4. Test Class Teardown Methods

Last piece of the test class is a `tearDown()` method, which can provide actions after all the test methods were finished. It can be test data deletion or any other necessary action. It is important to provide the `@AfterClass` annotation in order for TestNG to execute this method after all the test methods.

Create Your First Test Steps Class

1. Create Test Steps Class

Before we create a test steps class, it is recommended to decide, where to keep such classes and how to organize them. It is recommended to locate them either in the test folder in their own package (i.e. *src/test/java/steps/homepage/HomePageSteps.java*), or to store them directly in the main folder beside corresponding objects (i.e. *src/main/webobjects/homepage/HomePageSteps.java*). For the naming convention, it is good to keep the *Steps* keyword in the class name.

Sample structure of a test steps class

```
public class SomeSteps extends StepsBase {
    // Define Application Object Instance
    private final SomeApi someApi = getApiAppInstanceOf(SomeApi.class);
    private final SomeWeb someWeb = getApiAppInstanceOf(SomeWeb.class);

    // Define Steps
    @Step("Step Description ({someInput})")
    public Post someStep(String someInput) {
        Post post = someApi
                .endpoint1
                .endpoint2
                .createSomePost(someInput);
        Assert.assertNotNull(post.getId(), "Post ID is not null.");
        return post;
    }
}
```

All the steps classes should extend the `StepsBase` class in order to provide functions of the automation framework.

2. Define Application Object Instance

Application object instance stands as a connection between the test and the application under test. It allows you to create an application instance which you can later use to perform calls on it.

You can retrieve an application instance by calling the `getApiInstanceOf(ApiApplicationRootClass.class)` or `getWebInstanceOf(WebApplicationRootClass.class)`.

Mind that only one application instance of the same application can exist in a test run, but you can have multiple different applications instantiated at the same time and access them using the `TestRun` class.

3. Define Steps

Steps are defined using methods with `@Steps` TestNG annotation. They provide you opportunity to group i.e. API calls or web actions like filling a form into one method call, so that you do not have to repeat the calls every time and duplicate the code.

Steps are also a great place where you can put your assertions.

Create Your First Test Suite

1. Create TestNG Test Suite

Test suites follow the primary TestNG structure with XML files. It is necessary to put these XML test suites into test's resources *testsets* folder or subfolder (*src/test/resources/testsets*).

Link Test Class to the Test Suite

Defining the test suite follows all general rules for TestNG test suite definition. You can find more information in TestNG documentation.

TestNG Test Suite Sample

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">
<suite name="Sample Test Suite" parallel="classes">
    <parameter name="browser" value="chrome"/>
    <parameter name="environment" value="test"/>
    <parameter name="language" value="en"/>
    <test name="Test Name" thread-count="1">
        <classes>
            <class name="cz.inventi.qa.framework.tests.sampletests.SampleTest"/>
        </classes>
    </test>
</suite> <!-- Suite -->
```

Test Parallelization

You can also run tests in parallel.

After finishing your first test, you can continue to the chapter about test execution.

Features

- Assertions
- Multi-Language Support
- Test Execution Parallelization

Assertions

Assertions allow you to compare and validate values. Framework's assertion classes provide you with predefined methods to easily compare various data types in many ways.

If you need to extend provided assertion types even further, you can easily do so by extending bellow mentioned classes.

Soft Assertions

Soft assertions allows you to continue the test even when some of these assertion did not pass. The result of assertion is still kept in the test report - for that purpose, the `SoftAssertCollector` was implemented.

Soft-Assert Collector

`SoftAssertCollector` class instance collects and holds all the failed soft assertions, so that they can

be reported later, when the test ends.

The `SoftAssertCollector` class instance is bound to the `TestRun` instance, that stands for the current test run and can be accessed from there by `getSoftAssertCollector()` method call.

Use of the Soft Assertion

You can use soft assertions by calling the static `SoftAssert` class in your *test* and *steps* classes. The class is a child of the `Assert` class, about which you can find more information bellow - `SoftAssert` class provides most of its methods coverage.

Example use of SoftAssert class call

```
SoftAssert.assertTrue(value1, value2, "Description of the assertion");
```

Please check the source code to find what assertions you can use with the `SoftAssert` class.

Hard Assertions

Use of the Hard Assertion

Similarly to the soft assertions call, you can use hard assertions by calling the static `Assert` class. This class overrides the TestNG's default `Assert` class.

Example use of Assert class call

```
Assert.assertTrue(value1, value2, "Description of the assertion");
```

Please check the source code to find what assertions you can use with the `Assert` class.

Multi-language Support

The framework also supports testing of multi-language applications! In the following steps you will get to know how to set up the project for such use case.

1. Create a Language File Resources Folder
2. Create a Dictionary for Your Phrases
3. Validate Phrases in the Code Using Keywords and Index File
4. Specify Language Used in the Test

1. Create a Language File Resources Folder

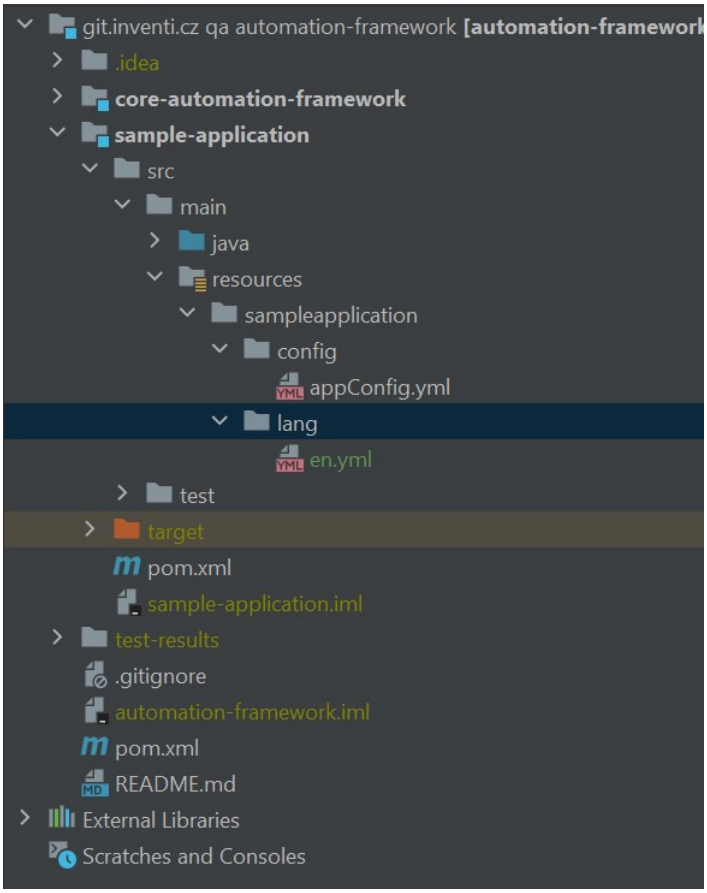
Navigate to your Maven project's module directory - typical path is `<your module name>/src/java/main/resources/<your application name>` and create a folder with name `lang`.Now you can supply dictionary files for any of the language you may need.

2. Create a Dictionary for Your Phrases

To be able to extend language support for the framework, we need to create a dictionary file that will contain all of the keywords (that will be used later in the code) and their corresponding phrases in given language.

Create a YAML file inside of the language file resources folder from the previous step, using following naming convention - the file's name should be an ISO 639-1 valid language identifier in lower-case format.For full list of supported identifiers, you can check `Language` class file.This Java enumeration contains all possible values in an upper-case format.

For example, to supply a dictionary file for English language, the filename would be following: `en.yml`.For Czech language, it would be either `cs_cz.yml` or `cs.yml`, and so on.



Inside this created file we now define our dictionary in a YAML file structure. The structure should be following:

File en.yaml

```
dictionary:
    HERE_DEFINE_YOUR_KEYWORD: "Keyword's phrase translation in given language"
    ANOTHER_KEYWORD: "Another translation"
```

The keywords should be defined in upper-case format as they will be later used as Java enumeration values.

For a real-world scenario, where you'd like to validate text of a "Continue" button in English and

Czech, the dictionary files would look like following:

File en.yaml

```
dictionary:
    BTN_CONTINUE: "Continue"
```

File cs_cz.yaml

```
dictionary:
    BTN_CONTINUE: "Pokračovat"
```

3. Validate Phrases in the Code Using Keywords and Index File

Now we are almost ready to validate text against our dictionary defined values using given keywords directly in the Java code.To access our keywords defined in the dictionary file an easier way, we should "transfer" the keywords from dictionary file to the Java world.For this purpose we will create an "index" Java enumeration that will contain all the possible keywords.Then, using this enumeration value in the code, we can validate against the specific text in a correct dictionary.

Create a Java enumeration class anywhere in your project you find comfortable.For our example, let's call it `Index.class`.Now provide enumeration values that will represent our keywords defined in the dictionary file (i.e. `BTN_CONTINUE` from the previous step).

Index.class example:

```
package cz.inventi.qa.framework.testapps.testweb.lang;

public enum Index {
        BTN_CONTINUE;
}
```

Now in the code of our test, we can validate text by te enumeration value using appropriate method accessible from the **LanguageManager** (static) object supplying the name of the keyword we want to verify and a valid application object:

```
LanguageManager.getTranslation(Index.BTN_CONTINUE, someApplicationObject);
```

Test use-case:

```
    public class SomeStepsExample extends StepsBase {
        private final HomePage homePage = getWebAppInstanceOf(HomePage.class);

        @Step("Check Continue Btn Text")
        public SomeStepsExample checkContinueBtnText() {
            String continueBtnTextTranslated = LanguageManager.getTranslation(Index
.BTN_CONTINUE, homePage);
            Assert.assertEquals(
```

```
                    homePage.getContinueBtn().getText(),
                    continueBtnTextTranslated,
                    "Check continue btn text is '" + continueBtnTextTranslated + "'"
            );
            return this;
        }
    }
```

Note that each of the applications has its own dictionary resources, therefore

4. Specify Language Used in the Test

In order for the framework to be able to use the right dictionary file, you need to supply a `language` parameter on the input.This can be done either via command line or TestNG suite.Again, the value of this parameter should correspond to the ISO 639-1 format in lower-case.

Example for command line:

```
... -Dlanguage=en ...
```

Example for TestNG suite:

```
...
    <parameter name="language" value="en"/>
...
```

If no `language` parameter is specified, the framework does not load any of the dictionary files.

Test Execution Parallelization

You can run tests in parallel by using the standard TestNG attributes. **Please mind that the framework is currently optimized only for running on the TestNG's "classes" level**.

- TestNG Parallelism Docs

How to Set up Parallel Test Execution

In your test suite XML file, you should add a `parallel="classes"` attribute to the `<suite>` tag. Moving onto the `<tests>` tag, please add a `thread-count=` attribute with a number of defined test classes (`<test>`) under this tag. Now when the test suite is run, defined test classes will be run in parallel.

If the `thread-count` attribute is set up incorrectly, it can lead to an unexpected behaviour of the automation framework.

Execution

Running Tests

Framework uses **TestNG** and **Maven** configuration to run tests. You can run tests as follows:

From an XML suite:

```
mvn --projects <YOUR_PROJECT_MODULE_NAME> clean test -DsuiteXmlFile=<TESTNG-SUITE
-FILE> -D<PARAMETER-NAME>=<PARAMETER-VALUE>
```

Mind that testing suites should be primarily located in *src/test/resources/testsets* - you should enter path heading from this location to your test suite.

From a test class directly:

```
mvn --projects <YOUR_PROJECT_MODULE_NAME> clean test -Dtest=<TEST-CLASS-NAME>
-D<PARAMETER-NAME>=<PARAMETER-VALUE>
```

Launch maven test from core-automation-framework module:

```
mvn --projects core-automation-framework clean test -Dtest=WebElementTests
```

Launch core-automation-framework unit tests suite:

```
mvn --projects core-automation-framework clean test
-DsuiteXmlFile=framework/unitTests.xml
```

Passing Parameters to the Test

You can pass parameters either by using the TestNG suite xml file, or directly with Maven command (see command sample above).

Parameters for Running Multiple Applications at Once

If your test uses more applications at once, it is recommended to write input parameters in format of `parameterName:applicationname`. In this way, you can supply parameter with same name for multiple projects with different values for each of the projects at once - supplying application name helps to assign parameter value to a specific application. Given parameter is used for all the applications in case of using multiple applications in one test without supplying application name to the parameter.

Passing Application Dependent Mandatory Parameters Through TestNG Suite

To pass application dependent mandatory parameter (i.e. like **environment**) using the TestNG

suite, it is necessary to put this parameter directly under the `<suite>` or `<test>` tag. Other parameters can be passed through deeper levels of TestNG suite.

Framework Run Mode

You can set up framework run modes (**NORMAL**, **DEBUG**) adding `-DrunMode` Maven parameter to command with selected mode value. Run mode affects mainly log information that is being displayed in the console output. By default, **NORMAL** run mode is applied.

Test Results

Continue to the reporting section to learn how to generate test reports.

Reporting

The Inventi Automation Framework uses the Allure Framework for reporting. Such solution provides better options to customize reports and also use annotation supplied by Allure Framework to describe test steps, epics, stories, severity and many other features.

- Alure Java with TestNG docs

Allure Test Results

Test results files are saved by default to the directory `test-results/allure-results` located in the project's root folder. You can change this folder in the global POM configuration file (pom.xml) of the project by editing the value of `allure.results.directory` property in `systemPropertyVariables`.

Report Generation

All tests' results are being recorded for Allure report. To display results properly on a local web server, go to the `test-results` folder and type command:

Command to generate HTML Allure Report

```
allure serve
```

Masking Sensitive Data in Reports

There is a possibility to mask sensitive data in Allure's results files and Allure reports - by default all TestNG input parameters' values that have name containing *"password"* or *"secret"* are masked with `[**MASKED**]` string. Their values will be hidden only if `DEBUG` framework run mode is not used. This solution should be reworked in the future to use `@Secret` annotation and AspectJ or Allure native solution.

Plugins & Editing Report Templates

You can pass almost any kind of data to the Allure reports, create your own categories and even edit the report's styling. It is also possible to write custom plugins for the Allure Framework.

- Allure Framework Plugin system

Cheatsheets

Hard to remember all the annotations and function calls? Maybe you will find bellow described cheatsheets useful.

- Annotations

Annotations Cheatsheet

Framework Annotations

@Application

Name

Crucial annotation to describe object's assignment to a specific application.It is recommended to use this annotation type in the "root" or "starting point" object of given application.For web front-end application this can be the home page, for API application this could be the root URL endpoint (https://example.com).Because of the inheritance and abstraction layer, all the underlying object's' application assignment is also identified by their ancestors (regardless of the amount of ancestors).

API Objects Specific Annotations

@ApiAuth

AuthType

Defines the authentication type for given API endpoint and it's children `Endpoint` objects - to prevent this behaviour, simply define @ApiAuth annotation in given child endpoint.

@EndpointSpecs

Url

URL addition to all the URL strings defined in the ancestors' `@Url` annotation of given Java class.

@ResponseSpecs

Helps to save information about data that format that will be returned to the HTTP response.

PassedDto

A DTO object to be mapped from the response body when the request "passes" (HTTP 100, 200 is returned).

FailedDto

A DTO object to be mapped from the response body when the request does not "pass" (HTTP 300, 400, 500 is returned).

ResponseType

Helps to determine the additional content, that might be returned with the response, specifically in the response body (i.e. file, object, array of objects - helps with mapping the response body to DTOs using Jackson).

Web Objects Specific Annotations

@FindElement

Serves for locating given element on a web page using given type of locator.Currently, only XPath locators are supported.All the locators use by default the tree structure, so they will stack using the all the ancestors' locators.

XPath

Contains path using an XPath pattern to locate an element in the DOM of the browser.

Index

Designed specifically to select n-th element, if there are more than one element found using the same XPath.Edits current XPath in the following way:

```
(<currentXpathValue>)[<indexValue>]
```

@NoParent

Used either on a WebElement, WebComponent or WebPage, this annotation states that given object should not use the XPath locator defined in its ancestors (using the tree pattern), nor extend it.All the underlying elements and given object will then be located only by XPath defined with that object.