Testing web applications using Selenium and Junit5 in Java

What you'll learn

- Define tests using Selenium and Junit5
- Run the test and push the test report to Xray
- Validate in Jira that the test results are available

Source-code for this tutorial

- code is available in GitHub

Overview

In this tutorial we will focus in taking advantage of the functionalities delivered by the new *JUnit 5* (Jupiter) , that is the next generation of JUnit.

This version is focusing on Java 8 and above and enables many different styles of testing.

JUnit 5 is the result of JUnit Lambda and its crowdfunding campaign on Indiegogo.

We will use an extension, developed in house, that will use the new functionalities provided by Junit5 to ingest richer reports in Xray.

The features available with the extension are:

- track started and finished date timestamps for each test
- link a test method to an existing Test issue or use auto-provisioning
- cover a "requirement" (i.e. an issue in Jira) from a test method
- specify additional fields for the auto-provisioned Test issues (e.g. summary, description, labels)
- attach screenshots or any other file as evidence to the Test Run, right from within the test method
- add comments to the Test Run, right from within the test method
- set the values for Test Run custom fields, right from within the test method

---

Prerequisites

In this tutorial, we will show how to use an extension to produce a report to be ingested in Xray using Selenium and Junit5 in Java.

We will need:

- Access to a demo site that we aim to test
- JDK8 and Maven installed (although you have the option to use a docker image, if you do this requirement is not mandatory)
- Configure the GitHub packages maven repo (to fetch the extension)
  - This will require authentication using GH username+token.
  - Use the *settings.yml.sample* where you need to set those credentials(once you have your credentials defined rename it to *settings.xml* and copy it to the *~/.m2* folder).

Code

The tests we have defined to demonstrate these new features consists in validating the login feature (with valid and invalid credentials) of the demo site, for that we have created a page object that will represent the loginPage

LoginPage.java

```
package com.xpandit.xray.tutorials;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.By;

public class LoginPage {

    private WebDriver driver;
    private RepositoryParser repo;
    private WebElement usernameElement;
    private WebElement passwordElement;
    private WebElement submitButtonElement;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
        repo = new RepositoryParser("./src/configs/object.properties");
        PageFactory.initElements(driver, this);
    }

    public LoginPage open()
    {
        driver.navigate().to(repo.getBy("url"));
        return this;
    }

    public void setUsername(String username) {
        usernameElement = driver.findElement(By.id(repo.getBy("username.
field.id")));
        usernameElement.sendKeys(username);
    }

    public void setPassword(String password) {
        passwordElement = driver.findElement(By.id(repo.getBy("password.
field.id")));
        passwordElement.sendKeys(password);
    }

    public WebElement getSubmitButton(){
        submitButtonElement = driver.findElement(By.id(repo.getBy("login.
button.id")));
        return submitButtonElement;
    }

    public LoginResultsPage submit()
    {
       getSubmitButton().submit();
       return new LoginResultsPage(driver);
    }

    public LoginResultsPage login(String username, String password)
    {
        setUsername(username);
        setPassword(password);
        return submit();
    }

    public Boolean contains(String text) {
        return driver.getPageSource().contains(text);
    }

    public String getTitle()
    {
        return driver.getTitle();
```

```
    }

    public Boolean isVisible()
    {
        WebDriverWait wait = new WebDriverWait(driver, 30000);
        return wait.until(ExpectedConditions.elementToBeClickable
(getSubmitButton())).isDisplayed();
    }

}
```

And another one to represent the Login Results Page, each one is a representation of the page we will interact in different times in the testing activities.

LoginResultsPage.java

```
package com.xpandit.xray.tutorials;

import org.openqa.selenium.WebDriver;

public class LoginResultsPage {
    private WebDriver driver;

    public LoginResultsPage(WebDriver driver) {
        this.driver = driver;
    }

    public Boolean contains(String text) {
        return driver.getPageSource().contains(text);
    }

    public String getTitle()
    {
        return driver.getTitle();
    }

}
```

As we can see in the above file we use an object repository (*RepositoryParser*) to enable an extra layer of abstraction, with it we can change the locators of the elements without the need to recompile or even change the endpoint of the application to be tested against several different deployments with no need to recompile.

To achieve it we have created an *object.properties* file that will hold the key/values to be loaded at execution time (in fact we have two ways to achieve this: using an XML file or using a properties file), in our case we have chosen to use a properties file.

This object repository file have information that can change but that does not required changes in code and as such does not need to trigger a compilation if changed, so instead of including those in the code we are loading them at execution time, removing the need to compile again after the change. In our case we have the locators that will be used to find the page elements and the expected messages returned by each operation.

object.properties

```
url=http://robotwebdemo.herokuapp.com/
password.field.id=password_field
username.field.id=username_field
login.button.id=login_button
expected.login.title=Welcome Page
expected.login.success=Login succeeded
expected.error.title=Error Page
expected.login.failed=Login failed
```

In order to demonstrate this functionality we have defined two tests: a valid login test and an invalid login test, as you can see in the below file:

LoginTests.java

```
package com.xpandit.xray.tutorials;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.junit.jupiter.api.Assertions.*;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.io.File;

import com.xpandit.xray.junit.customjunitxml.XrayTestReporter;
import com.xpandit.xray.junit.customjunitxml.
XrayTestReporterParameterResolver;
import com.xpandit.xray.junit.customjunitxml.annotations.Requirement;
import com.xpandit.xray.junit.customjunitxml.annotations.XrayTest;

@ExtendWith(XrayTestReporterParameterResolver.class)
public class LoginTests {
    WebDriver driver;
    RepositoryParser repo;

    @BeforeEach
    public void setUp() throws Exception {
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--no-sandbox"); // Bypass OS security model,
to run in Docker
        options.addArguments("--headless");
        driver = new ChromeDriver(options);
        repo = new RepositoryParser("./src/configs/object.properties");
    }

    @AfterEach
    public void tearDown() throws Exception {
        driver.quit();
        driver = null;
        repo = null;
    }

    @Test
```

```
    @XrayTest(key = "XT-12")
    @Requirement("XT-10")
    public void validLogin()
    {
        LoginPage loginPage = new LoginPage(driver).open();
        assertTrue(loginPage.isVisible());
        LoginResultsPage loginResultsPage = loginPage.login("demo",
"mode");
        assertEquals(loginResultsPage.getTitle(), repo.getBy("expected.
login.title"));
        assertTrue(loginResultsPage.contains(repo.getBy("expected.login.
success")));
    }

    @Test
    @XrayTest(summary = "invalid login test", description = "login attempt
with invalid credentials")
    public void invalidLogin(XrayTestReporter xrayReporter)
    {
        LoginPage loginPage = new LoginPage(driver).open();
        assertTrue(loginPage.isVisible());
        LoginResultsPage loginResultsPage = loginPage.login("demo",
"invalid");
        TakesScreenshot screenshotTaker =((TakesScreenshot)driver);
        File screenshot = screenshotTaker.getScreenshotAs(OutputType.FILE);
        xrayReporter.addTestRunEvidence(screenshot.getAbsolutePath());
        xrayReporter.addComment("auth should have failed");
        assertEquals(loginResultsPage.getTitle(), repo.getBy("expected.
error.title"));
        assertTrue(loginResultsPage.contains(repo.getBy("expected.login.
failed")));
    }

}
```

Let's look into the above code in more detail, the first highlight is regarding the ExtendWith annotation at the top of the class:

LoginTests.java

```
...
@ExtendWith(XrayTestReporterParameterResolver.class)
...
```

This annotation that is used to register extensions for the annotated test class or test method. In our case the extension: XrayTestReporterParameterResolver (as we have referred in the Prerequisites section).

Next we are initialising the driver with the following options:

LoginTests.java

```
...
        @BeforeEach
    public void setUp() throws Exception {
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--no-sandbox"); // Bypass OS security model,
to run in Docker
        options.addArguments("--headless");
        driver = new ChromeDriver(options);
        repo = new RepositoryParser("./src/configs/object.properties");
    }
...
```

Adding two arguments to the driver options:

- --no-sandbox, to bypass the OS security model and be able to run in Docker
- --headless, to execute the browser instance in headless mode

Also notice that we are doing these operations before each test (behaviour added with the annotation *BeforeEach)* and we are also initialising the object repository *RepositoryParser* (that only needs the file path to be loaded, this will enable us to change the file content without the need to change the code).

On the Tests definition we have added special annotations that will trigger special behaviour when ingested by Xray, for this first test we are using: *XrayTest* and *Requirement*.

LoginTests.java

```
...
        @Test
    @XrayTest(key = "XT-12")
    @Requirement("XT-10")
    public void validLogin()
    {
...
```

This will allow the Test to be linked to the Test in Xray with the id XT-12 and link this Test to the Requirement in the Xray side: XT-10, we will see the informations added to the report that will be generated further ahead.

On the second Test, *invalidLogin,* we have other examples of annotations, this time within the *XrayTest* we are adding a specific summary and description:

LoginTests.java

```
...
        @Test
    @XrayTest(summary = "invalid login test", description = "login attempt
with invalid credentials")
    public void invalidLogin(XrayTestReporter xrayReporter)
    {
        LoginPage loginPage = new LoginPage(driver).open();
        assertTrue(loginPage.isVisible());
        LoginResultsPage loginResultsPage = loginPage.login("demo",
"invalid");
        TakesScreenshot screenshotTaker =((TakesScreenshot)driver);
        File screenshot = screenshotTaker.getScreenshotAs(OutputType.FILE);
        xrayReporter.addTestRunEvidence(screenshot.getAbsolutePath());
        xrayReporter.addComment("auth should have failed");
        assertEquals(loginResultsPage.getTitle(), repo.getBy("expected.
error.title"));
        assertTrue(loginResultsPage.contains(repo.getBy("expected.login.
failed")));
    }
...
```

Lastly we are using the *xrayReporter* to add an evidence to the report, in this case it's a screenshot and a comment, that would appear in the report and also be ingested in Xray.

For more informations about the features available with this new extension please check xray-junit-extensions.

Execution

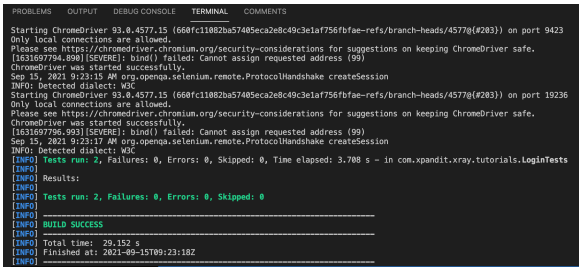
To execute the code use the following command:

```
mvn test
```

We also made available the possibility to execute the code inside a Docker container (note that a local directory should be mounted so that JUnit XML results are stored locally).

```
docker build . -t tutorial_java_junit5_selenium
docker run --rm -v $(pwd)/reports:/source/reports -t
tutorial_java_junit5_selenium
```

Once the execution as ended the results are immediately available in the terminal



Report

The execution will also produce a JUnit report that will look like this one:

Junit Report


```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="JUnit Jupiter" tests="2" skipped="0" failures="0" errors="
0" time="3" hostname="e4ed87727c63" timestamp="2021-09-15T09:23:18">
<properties>
<property name="awt.toolkit" value="sun.awt.X11.XToolkit"/>
<property name="basedir" value="/source"/>
<property name="file.encoding" value="UTF-8"/>
<property name="file.encoding.pkg" value="sun.io"/>
<property name="file.separator" value="/"/>
<property name="java.awt.graphicsenv" value="sun.awt.
X11GraphicsEnvironment"/>
<property name="java.awt.printerjob" value="sun.print.PSPrinterJob"/>
<property name="java.class.path" value="/source/target/test-classes:/source
/target/surefire.test.class.pathclasses:/...
<property name="java.class.version" value="52.0"/>
<property name="java.endorsed.dirs" value="/usr/local/openjdk-8/jre/lib
/endorsed"/>
<property name="java.ext.dirs" value="/usr/local/openjdk-8/jre/lib/ext:/usr
/java/packages/lib/ext"/>
<property name="java.home" value="/usr/local/openjdk-8/jre"/>
<property name="java.io.tmpdir" value="/tmp"/>
<property name="java.library.path" value="/usr/java/packages/lib/amd64:/usr
/lib64:/lib64:/lib:/usr/lib"/>
<property name="java.runtime.name" value="OpenJDK Runtime Environment"/>
<property name="java.runtime.version" value="1.8.0_282-b08"/>
<property name="java.specification.name" value="Java Platform API
Specification"/>
<property name="java.specification.vendor" value="Oracle Corporation"/>
<property name="java.specification.version" value="1.8"/>
<property name="java.vendor" value="Oracle Corporation"/>
<property name="java.vendor.url" value="http://java.oracle.com/"/>
<property name="java.vendor.url.bug" value="http://bugreport.sun.com
/bugreport/"/>
<property name="java.version" value="1.8.0_282"/>
<property name="java.vm.info" value="mixed mode"/>
<property name="java.vm.name" value="OpenJDK 64-Bit Server VM"/>
<property name="java.vm.specification.name" value="Java Virtual Machine
Specification"/>
<property name="java.vm.specification.vendor" value="Oracle Corporation"/>
<property name="java.vm.specification.version" value="1.8"/>
<property name="java.vm.vendor" value="Oracle Corporation"/>
<property name="java.vm.version" value="25.282-b08"/>
<property name="line.separator" value="
"/>
<property name="localRepository" value="/home/automation/.m2/repository"/>
<property name="os.arch" value="amd64"/>
<property name="os.name" value="Linux"/>
<property name="os.version" value="5.10.47-linuxkit"/>
<property name="path.separator" value=":"/>
<property name="sun.arch.data.model" value="64"/>
<property name="sun.boot.class.path" value="/usr/local/openjdk-8/jre/lib
/resources.jar:/usr/local/openjdk-8/jre/lib/rt.jar:/usr/local/openjdk-8/jre
/lib/sunrsasign.jar:/usr/local/openjdk-8/jre/lib/jsse.jar:/usr/local
/openjdk-8/jre/lib/jce.jar:/usr/local/openjdk-8/jre/lib/charsets.jar:/usr
/local/openjdk-8/jre/lib/jfr.jar:/usr/local/openjdk-8/jre/classes"/>
<property name="sun.boot.library.path" value="/usr/local/openjdk-8/jre/lib
/amd64"/>
<property name="sun.cpu.endian" value="little"/>
<property name="sun.cpu.isalist" value=""/>
<property name="sun.io.unicode.encoding" value="UnicodeLittle"/>
<property name="sun.java.command" value="/source/target/surefire
/surefirebooter3181939442444687220.jar /source/target/surefire 2021-09-
15T09-23-12_536-jvmRun1 surefire3986671724248479717tmp
surefire_06063017965226830567tmp"/>
<property name="sun.java.launcher" value="SUN_STANDARD"/>
```

```
<property name="sun.jnu.encoding" value="UTF-8"/>
<property name="sun.management.compiler" value="HotSpot 64-Bit Tiered
Compilers"/>
<property name="sun.os.patch.level" value="unknown"/>
<property name="surefire.real.class.path" value="/source/target/surefire
/surefirebooter3181939442444687220.jar"/>
<property name="surefire.test.class.path" value="/source/target/test-
classes:/source/target/classes:...
<property name="user.dir" value="/source"/>
<property name="user.home" value="/home/automation"/>
<property name="user.language" value="en"/>
<property name="user.name" value="automation"/>
<property name="user.timezone" value="Etc/UTC"/>
</properties>
<testcase name="invalidLogin" classname="com.xpandit.xray.tutorials.
LoginTests" time="1" started-at="2021-09-15T09:23:16.982" finished-at="
2021-09-15T09:23:18.105">
<system-out><![CDATA[
unique-id: [engine:junit-jupiter]/[class:com.xpandit.xray.tutorials.
LoginTests]/[method:invalidLogin(com.xpandit.xray.junit.customjunitxml.
XrayTestReporter)]
display-name: invalidLogin(XrayTestReporter)
]]></system-out>
<system-out><![CDATA[

]]></system-out>
<properties>
<property name="testrun_comment"><![CDATA[auth should have failed]]><
/property>

<property name="test_description"><![CDATA[login attempt with invalid
credentials]]></property>
<property name="test_summary" value="invalid login test"/>
<property name="testrun_evidence">
<item name="screenshot1244912439270873928.png"
>iVBORw0KGgoAAAANSUhEUgAAAyAAAAJYCAYAAACadoJwAAAAAXNSR0IArs4c6QAAIABJREFUeJ
...=</item>
</property>
</properties>
</testcase>
<testcase name="validLogin" classname="com.xpandit.xray.tutorials.
LoginTests" time="2" started-at="2021-09-15T09:23:14.399" finished-at="
2021-09-15T09:23:16.981">
<system-out><![CDATA[
unique-id: [engine:junit-jupiter]/[class:com.xpandit.xray.tutorials.
LoginTests]/[method:validLogin()]
display-name: validLogin()
]]></system-out>
<properties>
<property name="requirements" value="XT-10"/>
<property name="test_key" value="XT-12"/>
</properties>
</testcase>
<system-out><![CDATA[
unique-id: [engine:junit-jupiter]
display-name: JUnit Jupiter
]]></system-out>
</testsuite>
```


Notice that in the above report some properties were added to support the annotations we talked about previously, namely:

TEST-junit-report.xml

```
...
<property name="testrun_comment"><![CDATA[auth should have failed]]><
/property>

<property name="test_description"><![CDATA[login attempt with invalid
credentials]]></property>
<property name="test_summary" value="invalid login test"/>
<property name="testrun_evidence">
...
<property name="requirements" value="XT-10"/>
<property name="test_key" value="XT-12"/>
...
```

We will not go into details as the names are self-explanatory (as they directly link to the annotations that we described previously).

Integrating with Xray

As we saw in the above example, where we are producing Junit reports with the result of the tests, it is now a matter of importing those results to your Jira instance, this can be done by simply submitting automation results to Xray through the REST API, by using one of the available CI/CD plugins (e.g. for Jenkins) or using the Jira interface to do so.

API

API

Once you have the report file available you can upload it to Xray through a request to the REST API endpoint for JUnit, and for that the first step is to follow the instructions in v1 or v2 (depending on your usage) and use login/password or a personal token to authenticate.

Authentication

Jira 8.14 introduced the concept of Personal Access Tokens, which Xray takes advantage of. These tokens can be created in the user's profile section in Jira and have an expiration date; they can also be revoked at any moment (more information here).

To use them in Jira's and in Xray REST API calls, we need to use the HTTP header "Authorization" with the "Bearer <token>" value.

For the purpose of this tutorial we will be using the username/password approach in the requests.

JUnit XML results

In order to upload the test results we will use the API request with the definition of the project key (where the execution will be created) and the test plan key (to identify the TestPlan that we want to associate this run to).

```
curl -H "Content-Type: multipart/form-data" -u
$JIRA_USERNAME:$JIRA_PASSWORD -F "file=@./reports/TEST-junit-jupiter.xml"
"$JIRA_BASEURL/rest/raven/1.0/import/execution/junit?
projectKey=XT&testPlanKey=XT-304"
```

With this command we are ingesting the results back to Xray in project *XT* associated to the TestPlan *XT-304*.

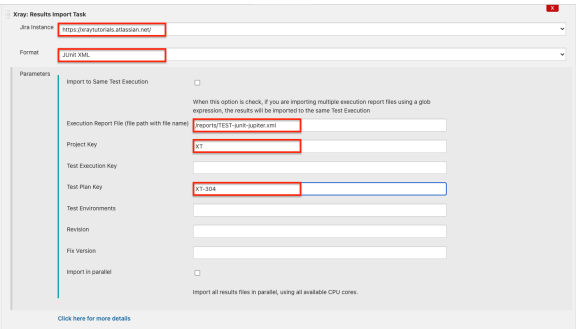
Jenkins

Jenkins

As you can see below we are adding a post-build action using the "*Xray: Results Import Task*" (from the Xray plugin available), where we have some options, we will focus on the one called "*Junit XML*".

Junit XML

- the Jira instance (where you have your Xray instance installed)
- the format as "*JUnit XML*"
- the test results file we want to import
- the Project key corresponding of the project, in Jira, where the results will be imported
- the Test Plan key where the execution will be created



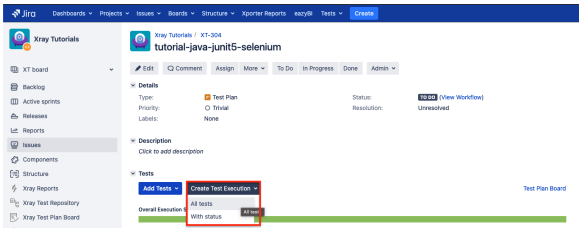
Once the step is saved and you execute your pipeline the Test results will be ingested in Xray, to have more detail about it check the section Xray imported results.

Jira UI

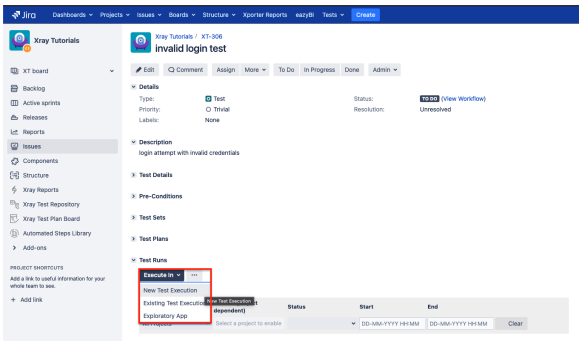
Jira UI

1

Create a Test Execution for the Test Plan that represents the Tests and results you had executed

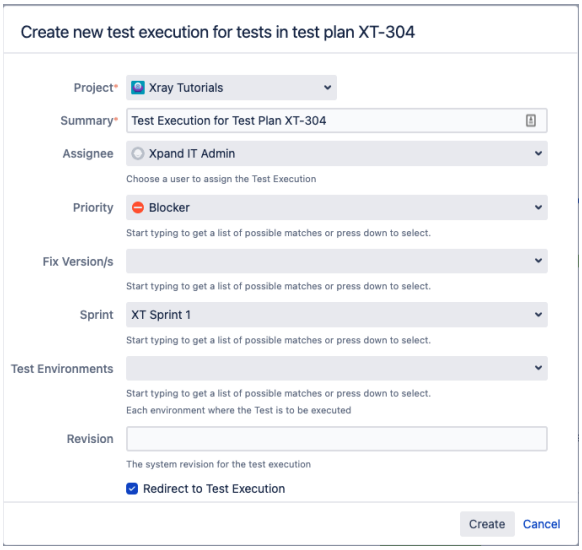


Or within a Test



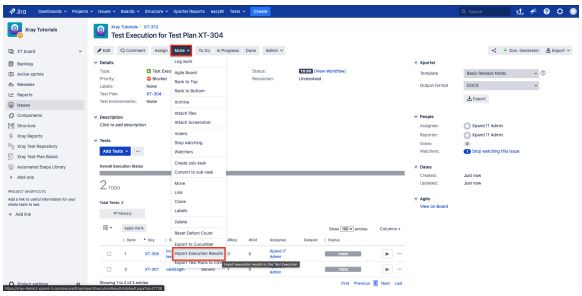
2

Fill in the necessary fields and press "*Create*"



3

Open the Test Execution and import the JUnit report

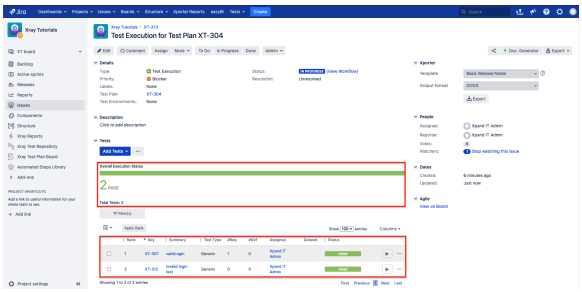


4

Choose the results file and press *"Import"*

5

The Test Execution is now updated with the test results imported



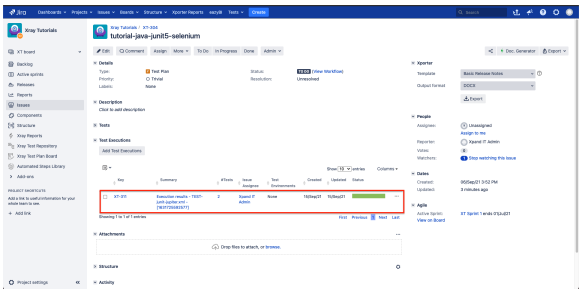
Details about the import details can be found in the next section.

Xray imported results

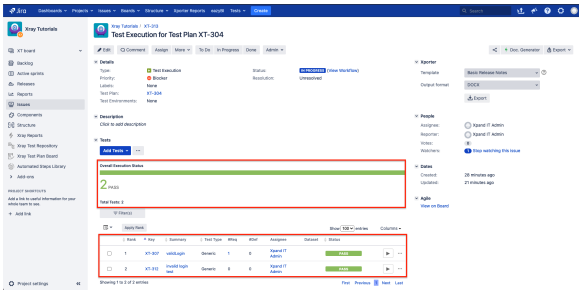
Giving that we are using a in house extension to add extra details to the results of the execution we will take a better look on what it means in Xray side, starting to look at the parameters we added in the request to import the execution results:

- TestPlanKey=XT-304
- ProjectKey=XT

With these parameters we are ingesting the results back to Xray in project *XT* associated to the TestPlan *XT-304*.



These executions are linked to Tests, so it has automatically added the Tests to the TestPlan as we can see:



Two Tests were added:

- XT-307 - validLogin
- XT-312 - invalid Login test

Let's look closer to each Test and the properties we added in the code, starting with the *"successLogin"* T est, in code we have:

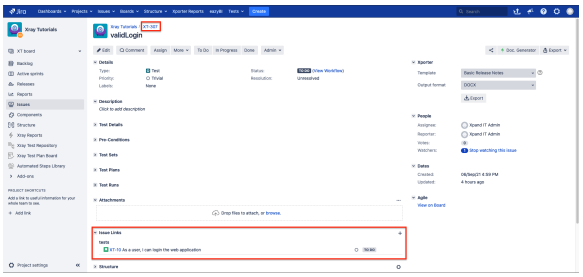
LoginTests.java

```
        @Test
    @XrayTest(key = "XT-307")
    @Requirement("XT-10")
    public void successLogin()
    {
        LoginPage loginPage = new LoginPage(driver).open();
        assertTrue(loginPage.isVisible());
        LoginResultsPage loginResultsPage = loginPage.login("demo",
"mode");
        assertEquals(loginResultsPage.getTitle(), repo.getBy("expected.
login.title"));
        assertTrue(loginResultsPage.contains(repo.getBy("expected.login.
success")));
    }
```

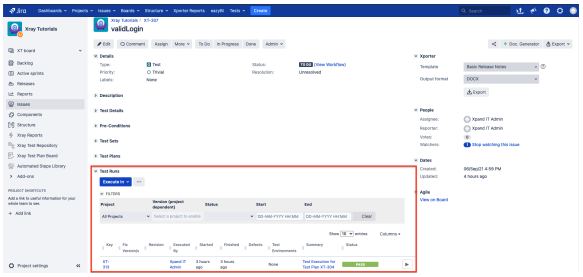
In this Test we are using two new annotations that will allow us to set information on the Xray side, namely:

- *@XrayTest(key = "XT-307")*, this line will associate this Test (*successLogin()*) to the Test in Xray with identifier XT-307
- *@Requirement("XT-10")*, with this one we are defining what is the requirement that this Test will cover (creating the relation between them)

We can check that the above information is present in Xray by opening the Test *XT-307*:



We also have a Test Execution associated to the above Test (that was added as we have uploaded the results):



On the second Test we have a different usage of the annotations and the usage of the reporter to add extra information like we can see:

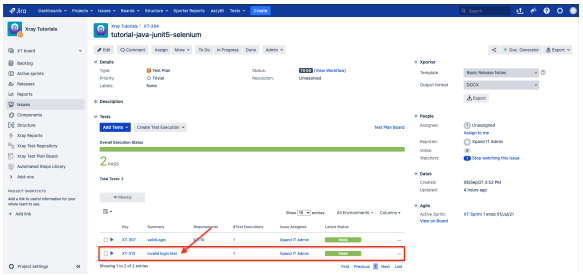
LoginTests.java

```
    @Test
    @XrayTest(summary = "invalid login test", description = "login attempt
with invalid credentials")
    public void nosuccessLogin(XrayTestReporter xrayReporter)
    {
        LoginPage loginPage = new LoginPage(driver).open();
        assertTrue(loginPage.isVisible());
        LoginResultsPage loginResultsPage = loginPage.login("demo",
"invalid");
        TakesScreenshot screenshotTaker =((TakesScreenshot)driver);
        File screenshot = screenshotTaker.getScreenshotAs(OutputType.FILE);
        xrayReporter.addTestRunEvidence(screenshot.getAbsolutePath());
        xrayReporter.addComment("auth should have failed");
        assertEquals(loginResultsPage.getTitle(), repo.getBy("expected.
error.title"));
        assertTrue(loginResultsPage.contains(repo.getBy("expected.login.
failed")));
    }
```

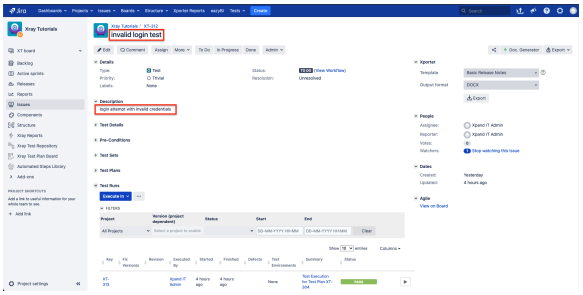
In more detail we have:

- *@XrayTest(summary = "invalid login test", description = "login attempt with invalid credentials")*, that is adding a summary and description to the Test that will be created when the results will be uploaded (if the Test already exists it will add this information)
- *xrayReporter.addTestRunEvidence(screenshot.getAbsolutePath());*, this line uses the *xrayReporter* to add an evidence to the report linked to this Test Execution (it can be any file), in our case we are adding a screenshot to the results of this Test.
- *xrayReporter.addComment("auth should have failed");*, this will allow us to add a comment to result

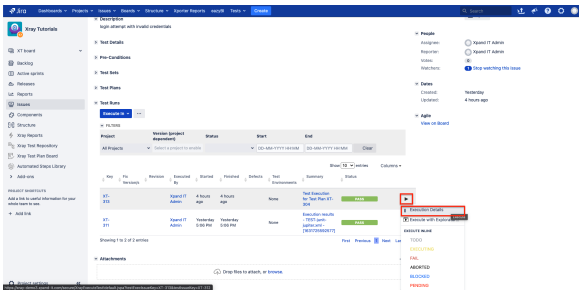
In Xray, if we open the Test Plan we can see that one Test (*XT-312*) was created and associated to it for this special case with a description and summary from the report:



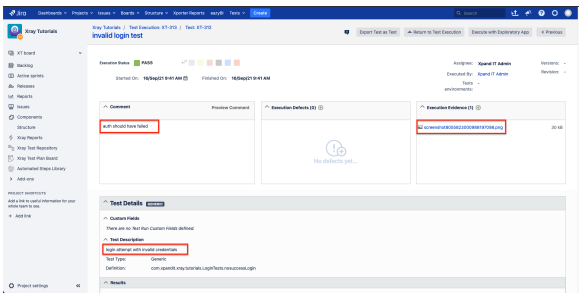
When we look to the details of that Test (by clicking in the link over the *XT-312*) we can see the summary and description match the ones we sent in the report:



Finally when we enter the Test Execution to check the details by clicking on the play icon and choosing the entry "Execution Details"



We are taken to the Execution Details screen where we can validate that the comment that we added with the code: *xrayReporter.addComment("auth should have failed");* is now present under the *comment* area. On the *Evidence* area we can see the screenshot we added in code also and the Description we added with *@XrayTest(summary = "invalid login test", description = "login attempt with invalid credentials")* is present under the Test Description area.



Tips

- after results are imported, in Jira, Tests can be linked to existing requirements/user stories (or in this case use the annotation to that from code), so you can track the impacts on their coverage.
- results from multiple builds can be linked to an existing Test Plan, to facilitate the analysis of test result trends across builds.
- results can be associated with a Test Environment, in case you want to analyze coverage and test results by environment later on. A Test Environment can be a testing stage (e.g. dev, staging, preprod, prod) or an identifier of the device/application used to interact with the system (e.g. browser, mobile OS).

References

- https://github.com/Xray-App/tutorial-java-junit5-selenium
- https://github.com/Xray-App/xray-junit-extensions
- https://junit.org/junit5/docs/current/user-guide/