Testing using WebDriverIO and Cucumber in JavaScript

Overview

In this tutorial, we will create UI tests as Cucumber Scenario(s)/Scenario Outline(s) and use WebDriverIO to implement the tests in JavaScript.

- code is available in GitHub

Requirements

- nodejs
- WebDriverIO

Description

For the purpose of this tutorial, we'll use a dummy website provided by Heroku. In our case, it contains just a few pages to support login features which we will be testing.

To start using WebDriverIO please follow the Get Started documentation.

WebDriverIO provides a client that after being installed will guide you through bootstrapping a *Hello World* test suite into your project, for this tutorial we will use the code generated by this tool for simplicity (with page objects).

The test consists in validating the login feature (with valid and invalid credentials) of the demo site, for that we have created a feature file that will have the description of the test supported by a base page that contains all methods and functionality that is shared across all page objects, a login page, that will extend the base page, that will have all the methods for interacting with the login page and a result page that will have the methods to interact in the page that is loaded after the login operation.

We have followed the documentation and executed the command to install the WebDriverIO test runner:

```
npm install @wdio/cli
```

Then we answered a series of questions that will define the code to be generated using:

```
npx wdio config
```

The output of the questionnaire will look like this:

```
=======================
WDIO Configuration Helper
=======================

? Where is your automation backend located? On my local machine
? Which framework do you want to use? cucumber
? Do you want to use a compiler? No!
? Where are your feature files located? ./features/**/*.feature
? Where are your step definitions located? ./features/step-definitions/steps.js
? Do you want WebdriverIO to autogenerate some test files? Yes
? Do you want to use page objects (https://martinfowler.com/bliki/PageObject.html)? Yes
? Where are your page objects located? ./features/pageobjects/**/*.js
? Which reporter do you want to use? spec
? Do you want to add a service to your test setup? chromedriver
? What is the base url? http://localhost
```

This will automatically generate the following files:

./pageobjects/page.js

```
const Page = require('./page');

/**
 * sub page containing specific selectors and methods for a specific page
 */
class LoginPage extends Page {
    /**
     * define selectors using getter methods
     */
    get inputUsername () { return $('#username') }
    get inputPassword () { return $('#password') }
    get btnSubmit () { return $('button[type="submit"]') }

    /**
     * a method to encapsule automation code to interact with the page
     * e.g. to login using username and password
     */
    async login (username, password) {
        await (await this.inputUsername).setValue(username);
        await (await this.inputPassword).setValue(password);
        await (await this.btnSubmit).click();
    }

    /**
     * overwrite specifc options to adapt it to page object
     */
    open () {
        return super.open('login');
    }
}

module.exports = new LoginPage();
```

./pageobjects/login.page.js

```
/**
* main page object containing all methods, selectors and functionality
* that is shared across all page objects
*/
module.exports = class Page {
    /**
    * Opens a sub page of the page
    * @param path path of the sub page (e.g. /path/to/page.html)
    */
    open (path) {
        return browser.url(`https://the-internet.herokuapp.com/${path}`)
    }
}
```

./pageobjects/secure.page.js

```
const Page = require('./page');

/**
 * sub page containing specific selectors and methods for a specific page
 */
class SecurePage extends Page {
    /**
     * define selectors using getter methods
     */
    get flashAlert () { return $('#flash') }
}

module.exports = new SecurePage();
```

And a feature file where we describe the tests:

login.feature

```
Feature: As a user, I can log into the secure area

        Scenario Outline: As a user, I can log into the secure area
                Given I am on the login page
                When I login with <username> and <password>
                Then I should see a flash message saying <message>

                        Examples:
                                | username | password             | message                         |
                                | tomsmith | SuperSecretPassword! | You logged into a secure area! |
                                | foobar   | barfoo               | Your username is invalid!      |
```

With the respective code behind

./step-definitions/steps.js

```
const { Given, When, Then } = require('@cucumber/cucumber');

const LoginPage = require('../pageobjects/login.page');
const SecurePage = require('../pageobjects/secure.page');

const pages = {
    login: LoginPage
}

Given(/^I am on the (\w+) page$/, async (page) => {
    await pages[page].open()
});

When(/^I login with (\w+) and (.+)$/, async (username, password) => {
    await LoginPage.login(username, password)
});

Then(/^I should see a flash message saying (.*)$/, async (message) => {
    await expect(SecurePage.flashAlert).toBeExisting();
    await expect(SecurePage.flashAlert).toHaveTextContaining(message);
});
```

The last two steps to have everything configured is to define that we will use the CucumberJS framework, for that we execute the following command:

```
npm install wdio-cucumberjs-json-reporter --save-dev
```

And in the wdio.conf.js we have added, in the reporters area, the following CucumberJS definition:

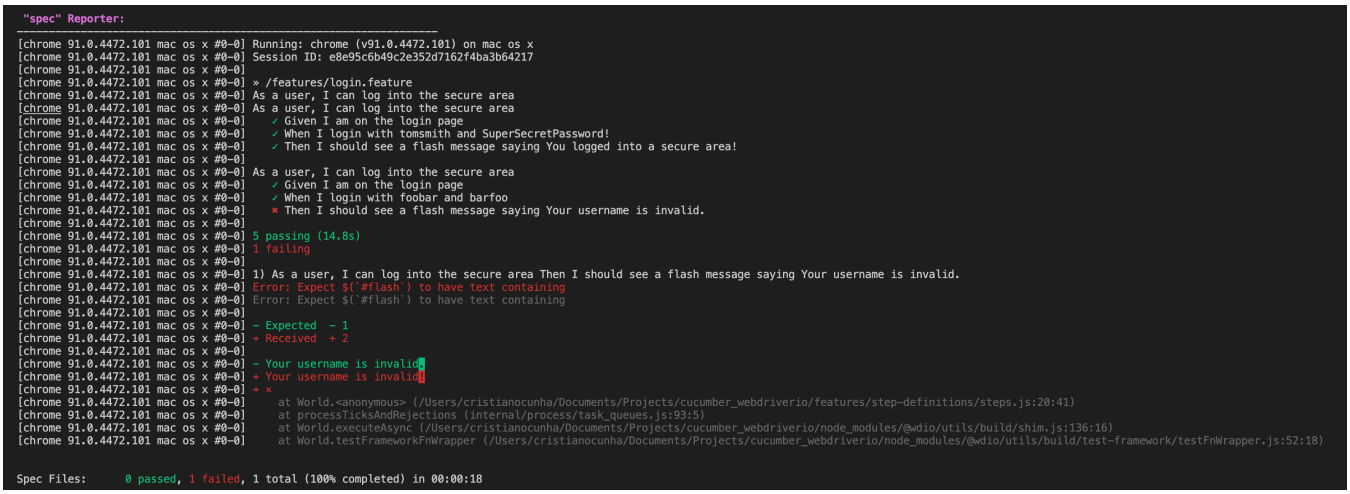
./wdio.conf.js

```
...
   reporters: ['spec',
            ['cucumberjs-json', {
                    jsonFolder: '.tmp/json/',
                    language: 'en',
                },
                   ],
...
```

Once the code is implemented (and we will make it fail on purpose on one test, to show the failure reports), it can be executed with the following command:

```
npx wdio run ./wdio.conf.js
```

The results are immediately available in the terminal



In case you need to interact with the Xray REST API at low-level using scripts (e.g. Bash/shell scripts), this tutorial uses auxiliary files that will handle those interactions.

Example of cloud_auth.json used in this tutorial

```
- export_features.sh
- import_features.sh
- import_results.sh
- run_all_git_workflow.sh
- run_all_standard_workflow.sh
```

Now we need to decide which workflow to use: do you want to use Xray/Jira as the master for writing the declarative specification (i.e. the Gherkin based Scenarios), or do you want to manage those outside using some editor and store them in Git, for example?

Learn more

Please see Testing in BDD with Gherkin based frameworks (e.g. Cucumber) for an overview of the possible workflows.

The place that you'll use to edit the Cucumber Scenarios will affect your workflow. There are teams that prefer to edit Cucumber Scenarios in Jira using Xray, while there are others that prefer to edit them by writing the .feature files by hand using some IDE.

Using Jira and Xray as master

This section assumes using Xray as master, i.e. the place that you'll be using to edit the specifications (e.g. the scenarios that are part of .feature files).

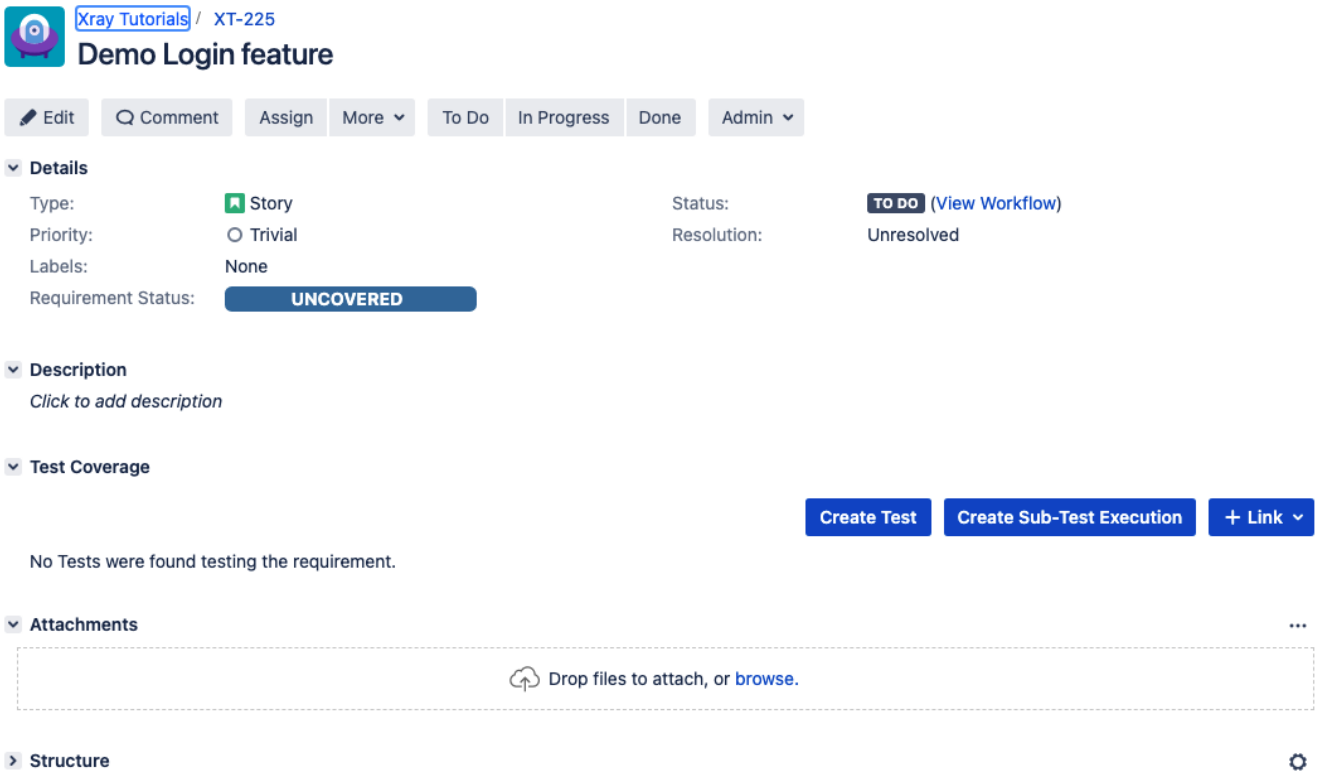
The overall flow would be something like this:

1. create Scenario/Scenario Outline as a Test in Jira; usually, it would be linked to an existing "requirement"/Story (i.e. created from the respective issue screen)
2. implement the code related to Gherkin statements/steps and store it in Git, for example
3. generate .feature files based on the specification made in Jira
4. checkout the code from Git
5. run the tests in the CI
6. import the results back to Jira

Usually, you would start by having a Story, or similar (e.g. "requirement"), to describe the behavior of a certain feature and use that to drive your testing.

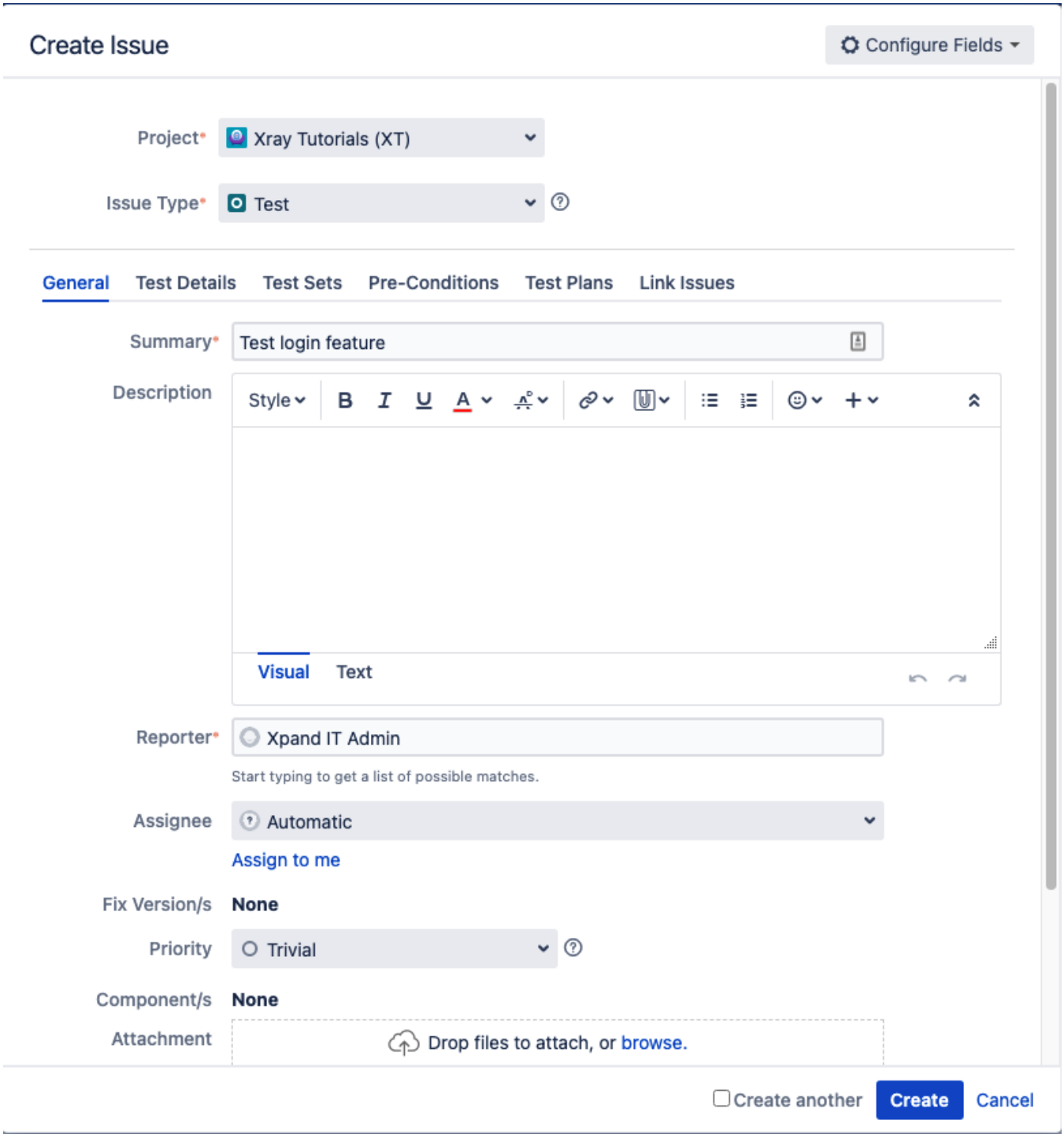
If you have it, then you can just use the "Create Test" on that issue to create the Scenario/Scenario Outline and have it automatically linked back to the Story/"requirement."

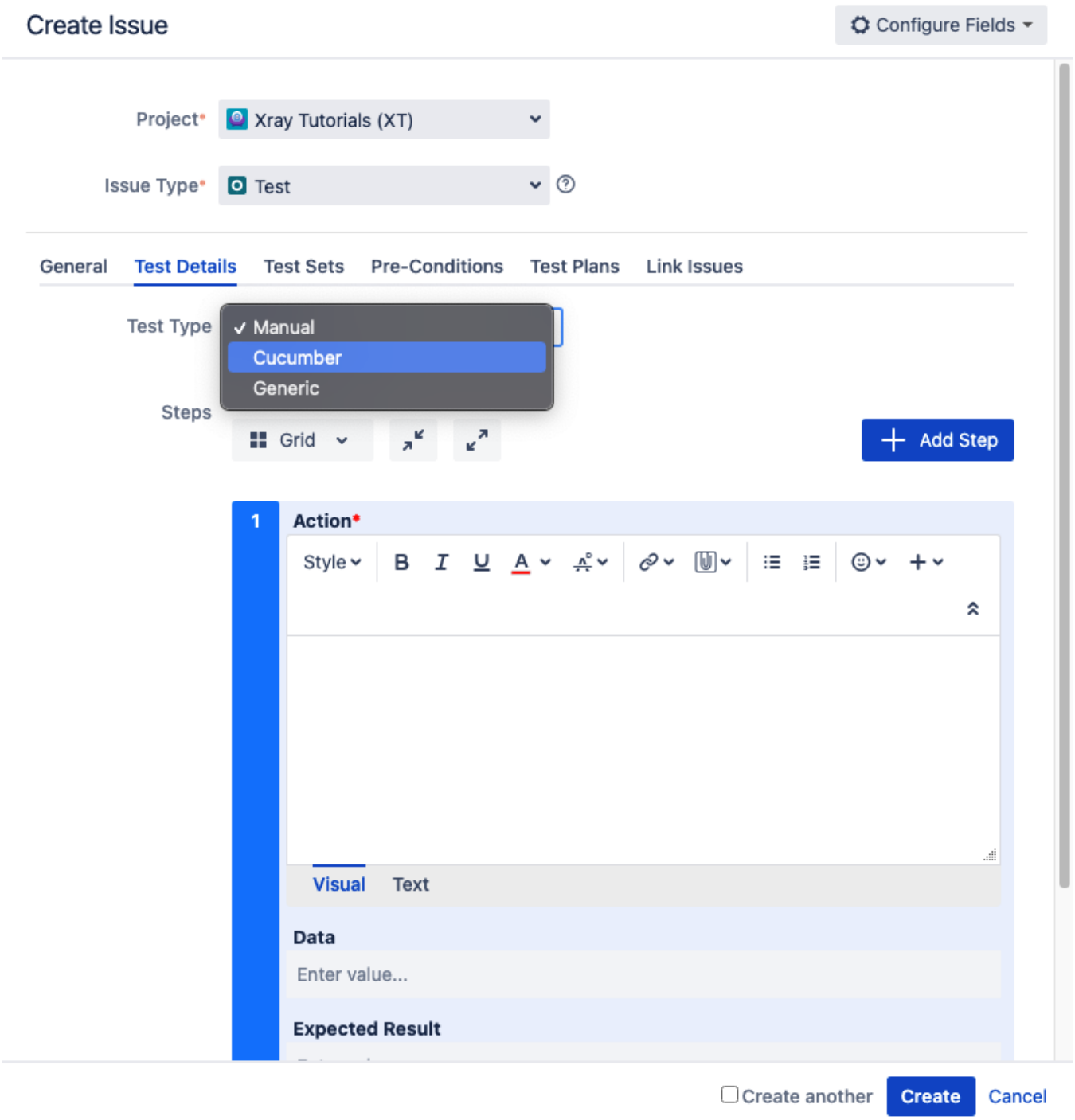
Otherwise, you can create the Test using the standard (issue) Create action from Jira's top menu.



In this case, we'll create a Cucumber Scenario.

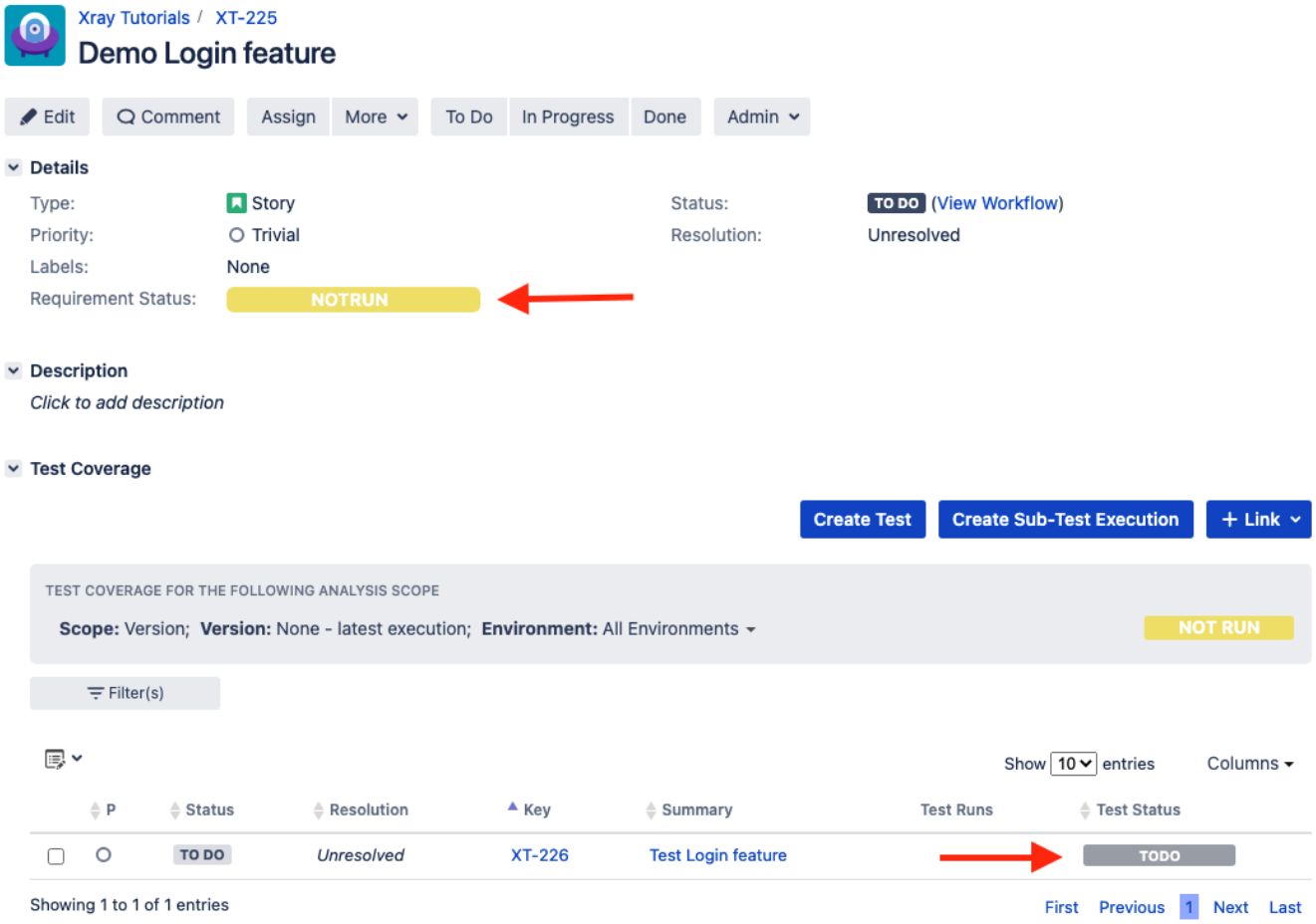
We need to create the Test issue first and fill out the Gherkin statements later on in the Test issue screen.


Create Issue
Configure Fields
Project*
Xray Tutorials (XT)
Issue Type*
Test
General
Test Details
Test Sets
Pre-Conditions
Test Plans
Link Issues
Summary*
Test login feature
Description
Style
Visual
Text
Reporter*
Xpand IT Admin
Start typing to get a list of possible matches.
Assignee
Automatic
Assign to me
Fix Version/s
None
Priority
Trivial
Component/s
None
Attachment
Drop files to attach, or browse.
Create another
Create
Cancel



After the Test is created it will impact the coverage of related "requirement," if any.

The coverage and the test results can be tracked on the "requirements" side (e.g. user story). In this case, you may see that coverage changed from being UNCOVERED to NOTRUN (i.e. covered and with at least one test not run).



Additional tests could be created and eventually linked to the same Story or linked to another one (e.g. logout).

The related statement's code is managed outside of Jira and stored in Git, for example.

In our source code, test code is stored under `steps-definitions` directory, which itself can contain several other directories or files. In this case, we've only one referring to the login feature:

./step-definitions/steps.js

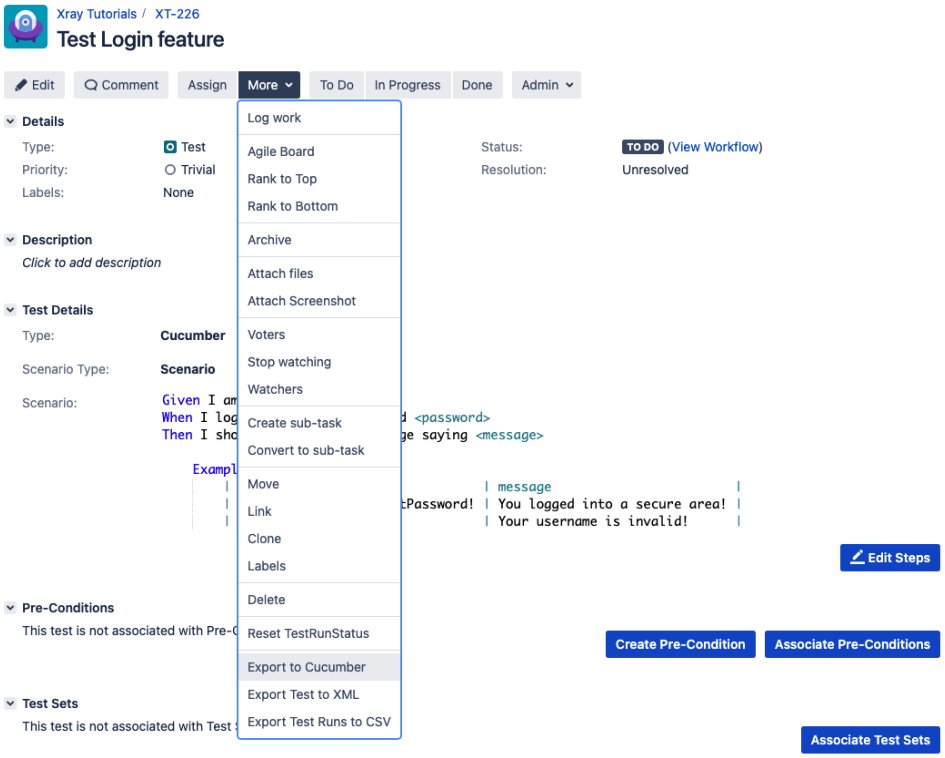
```
const { Given, When, Then } = require('@cucumber/cucumber');

const LoginPage = require('../pageobjects/login.page');
const SecurePage = require('../pageobjects/secure.page');

const pages = {
    login: LoginPage
}

Given(/^I am on the (\w+) page$/, async (page) => {
    await pages[page].open()
});

When(/^I login with (\w+) and (.+)$/, async (username, password) => {
    await LoginPage.login(username, password)
});

Then(/^I should see a flash message saying (.*)$/, async (message) => {
    await expect(SecurePage.flashAlert).toBeExisting();
    await expect(SecurePage.flashAlert).toHaveTextContaining(message);
});

After((scenario) => {
    const path =  '.tmp/screenshots/Error.png';
    if(scenario.result.status == 6){
        browser.saveScreenshot(path);
        const cucumberJson = require('wdio-cucumberjs-json-reporter').default;
        const data = fs.readFileSync(path);
        if (data) {
            const base64Image = Buffer.from(data, 'binary').toString('base64')
            cucumberJson.attach(base64Image, 'image/png');
        }

    }
});
```

Notice that we have added an After scenario that will be executed after each scenario. After validating that an error occurred it will take a screenshot and attach it to the report using the `wdio-cucumberjs-json-reporter` library.

You can then export the specification of the test to a Cucumber .feature file via the REST API, or the **Export to Cucumber** UI action from within the Test /Test Execution issue or even based on an existing saved filter. A plugin for your CI tool of choice can be used to ease this task.

So, you can either:

- use the UI



- use the REST API (more info here)
  - **example of a shell script to export/generate .features from Xray**

```
#!/bin/bash

JIRA_BASEURL=https://192.168.2.168
JIRA_USERNAME=admin
JIRA_PASSWORD=admin
KEYS="XT-142"

rm -f features.zip
curl -u $JIRA_USERNAME:$JIRA_PASSWORD  "$JIRA_BASEURL/rest/raven/2.0/export/test?
keys=$KEYS&fz=true" -o features.zip
unzip -o features.zip  -d features
```

- use one of the available CI/CD plugins (e.g. see an example of Integration with Jenkins)

We will export the features to a new directory named `features/` on the root folder of your project.

After being exported, the created .feature(s) will contain references to the Test issue key, eventually prefixed (e.g. "TEST_") depending on an Xray global setting, and the covered "requirement" issue key, if that's the case. The naming of these files is detailed in Export Cucumber Features backup.

features/1_XT-225.feature

```
@REQ_XT-225
Feature: Login feature

        @TEST_XT-226
        Scenario: Test Login feature
                Scenario Outline: As a user, I can log into the secure area
                                Given I am on the login page
                                When I login with <username> and <password>
                                Then I should see a flash message saying <message>

                                        Examples:
                                                | username | password             |
message                         |
                                                | tomsmith | SuperSecretPassword! | You logged into a secure
area! |
                                                | foobar   | barfoo               | Your username is
invalid.      |
```

To run the tests and produce Cucumber JSON reports(s), we can either use `the same command as before`.

```
npx wdio run ./wdio.conf.js
```

This will produce one results file that will hold the test results.

After running the tests, results can be imported to Xray via the REST API, or the **Import Execution Results** action within the Test Execution, or by using one of the available CI/CD plugins (e.g. see an example of Integration with Jenkins).

import_results.sh

```
#!/bin/bash
JIRA_BASEURL=https://192.168.0.168
JIRA_USERNAME=admin
JIRA_PASSWORD=admin

curl -H "Content-Type: application/json" -X POST -u $JIRA_USERNAME:$JIRA_PASSWORD --data @".tmp/json/login-
feature.json" $JIRA_BASEURL/rest/raven/2.0/import/execution/cucumber
```

Which Cucumber endpoint/"format" to use?

To import results, you can use two different endpoint/"formats" (endpoints described in Import Execution Results - REST):

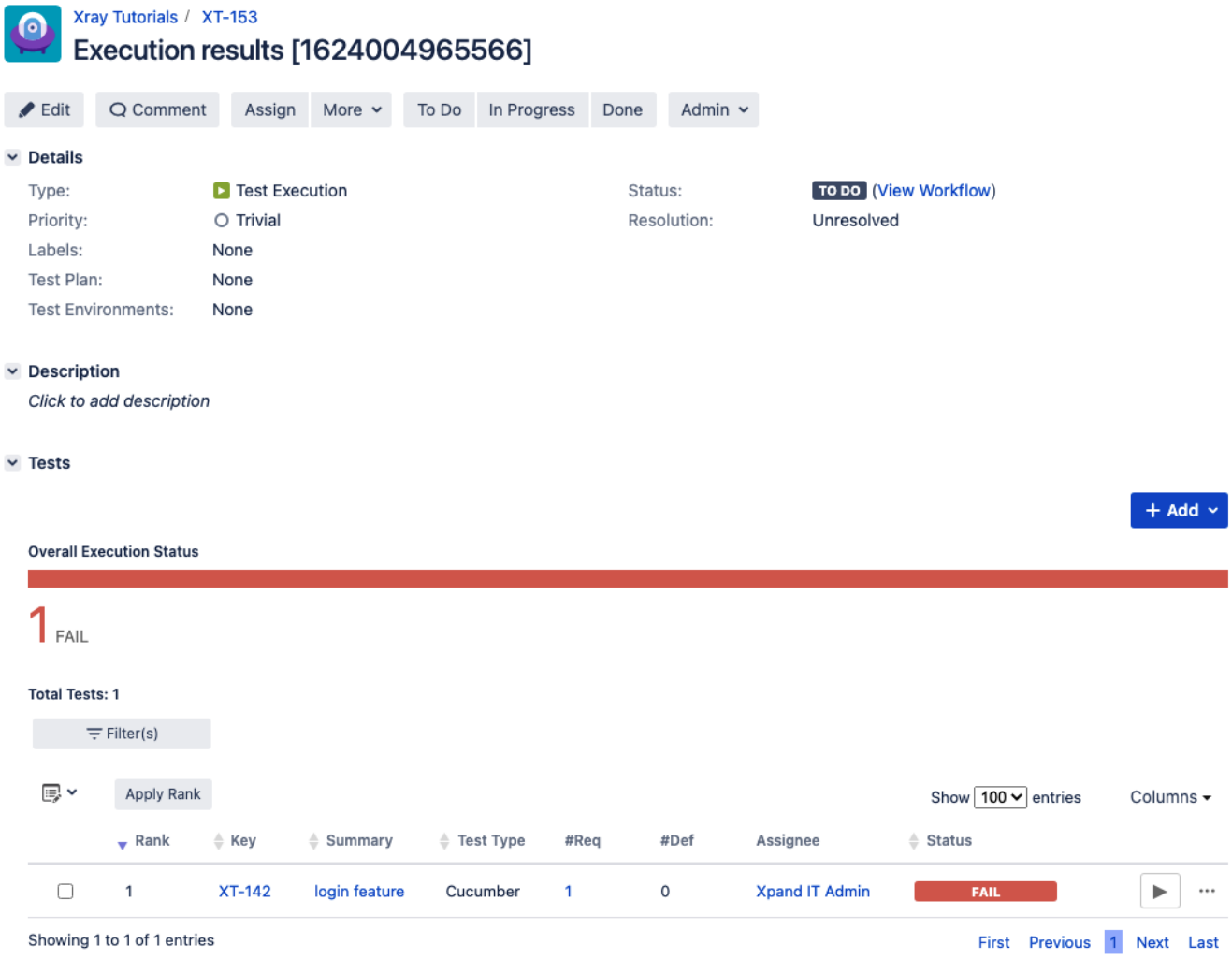
1. the "standard cucumber" endpoint
2. the "multipart cucumber" endpoint

The standard cucumber endpoint (i.e. */import/execution/cucumber)* is simpler but more restrictive: you cannot specify values for custom fields on the Test Execution that will be created. This endpoint creates new Test Execution issues unless the Feature contains a tag having an issue key of an existing Test Execution.

The multipart cucumber endpoint will allow you to customize fields (e.g. Fix Version, Test Plan) if you wish to do so, on the Test Execution that will be created. Note that this endpoint always creates new Test Executions (as of Xray v4.2).

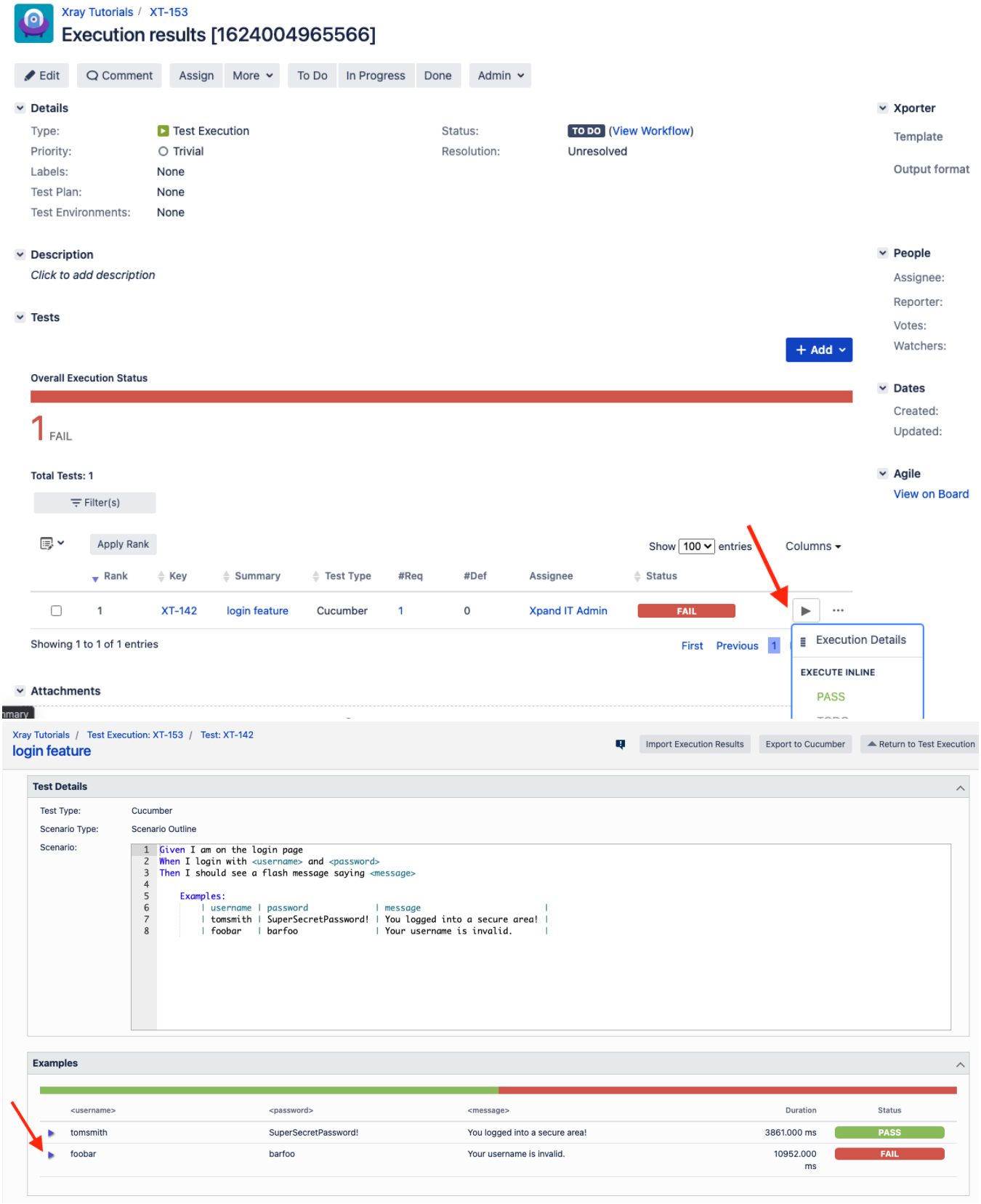
In sum, if you want to customise the Fix Version, Test Plan and/or Test Environment of the Test Execution issue that will be created, you'll have to use the "multipart cucumber" endpoint.

A new Test Execution will be created (unless you originally exported the Scenarios/Scenario Outlines from a Test Execution).

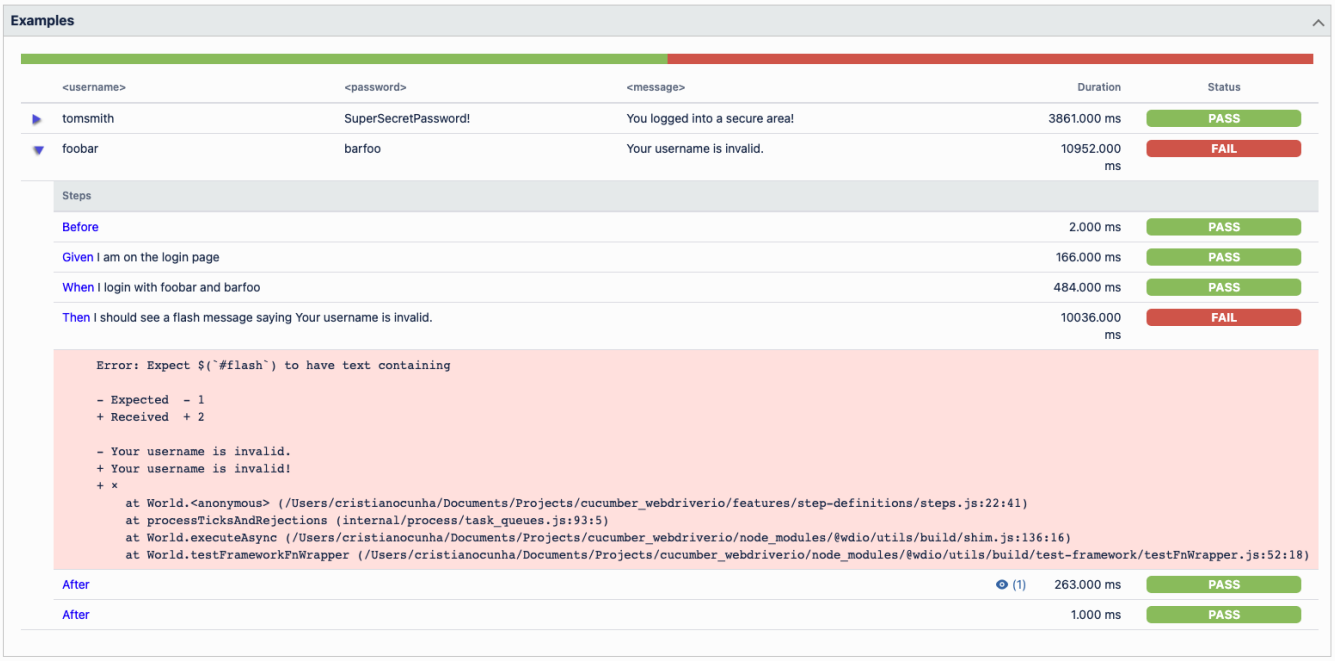


The tests have failed (on purpose).

The execution screen details of the Test Run will provide overall status information and Gherkin statement-level results, therefore we can use it to analyze the failing test.

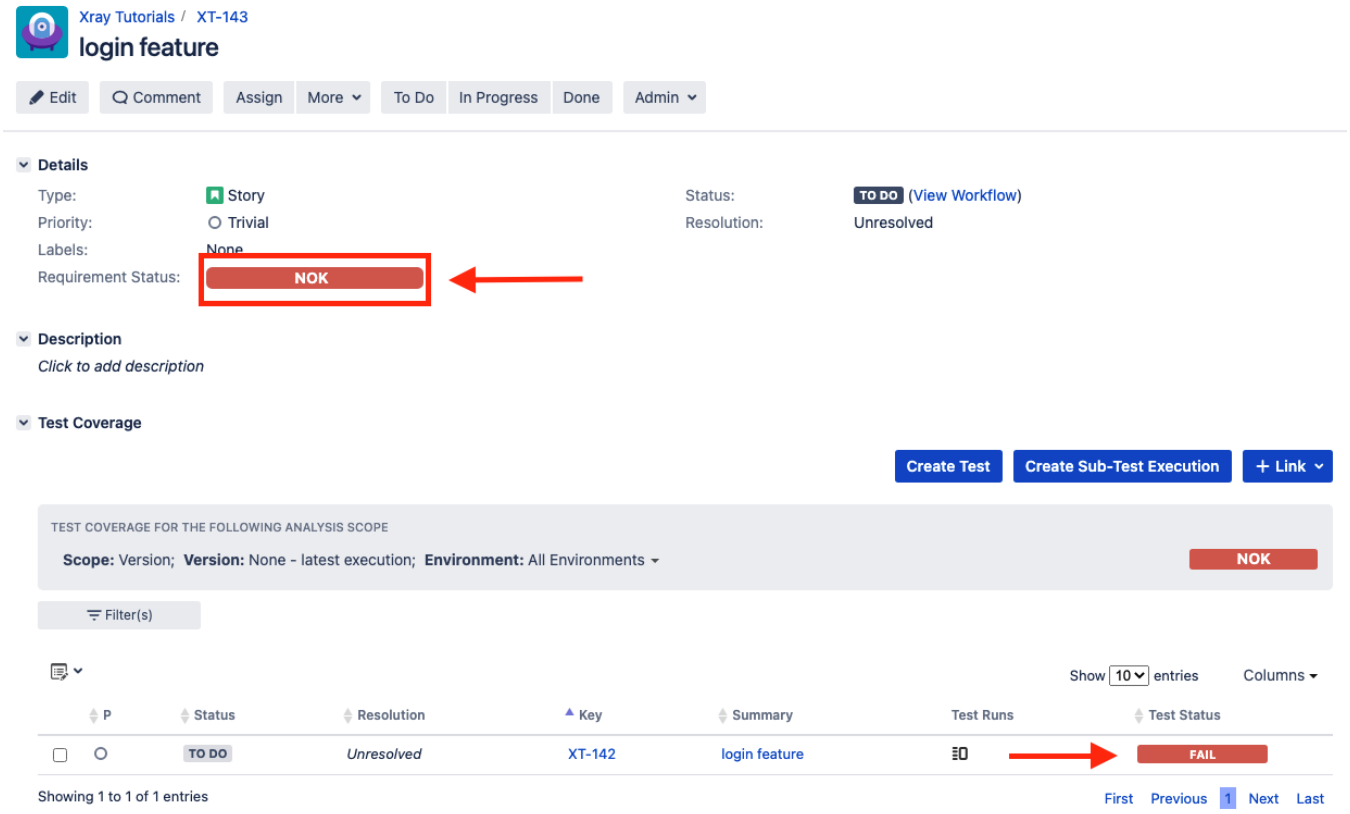


A given example can be expanded to see all Gherkin statements and, if available, it is possible to see also the attached stack trace.



Note: in this case, the bug was on the Scenario Outline example which was expecting an invalid message.

Results are reflected on the covered item (e.g. Story). On its issue screen, coverage now shows that the item is NOK based on the latest testing results, this can also be tracked within the Test Coverage panel bellow.



Using Git or other VCS as master

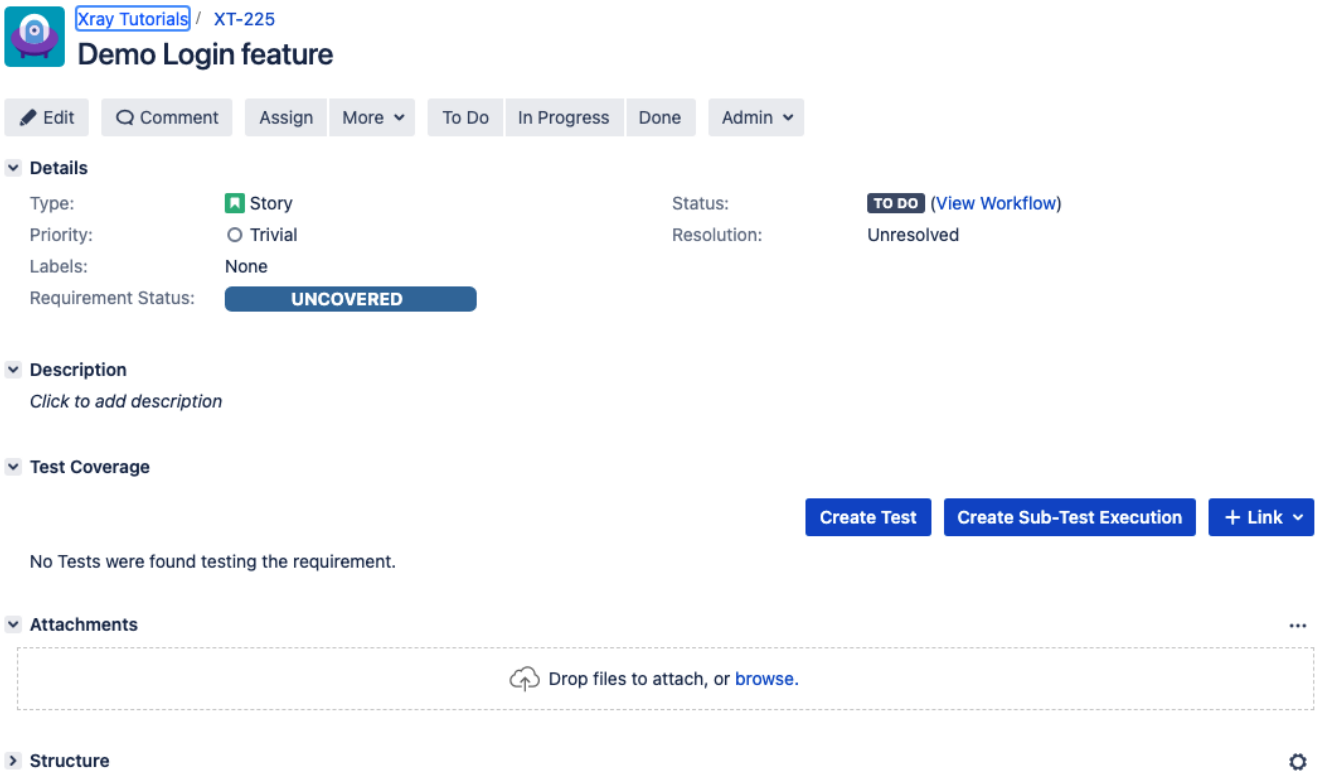
You can edit your .feature files using your IDE outside of Jira (eventually storing them in your VCS using Git, for example) alongside the remaining test code.

In any case, you'll need to synchronize your .feature files to Jira so that you can have visibility of them and report results against them.

The overall flow would be something like this:

1. look at the existing "requirement"/Story issue keys to guide your testing; keep their issue keys
2. specify Cucumber/Gherkin .feature files in your IDE and store it in Git, for example
3. implement the code related to Gherkin statements/steps and store it in Git, for example
4. import/synchronise the .feature files to Xray to provision or update corresponding Test entities
5. export/generate .feature files from Jira, so that they contain references to Tests and requirements in Jira
6. checkout the WebDriverIO related code from Git
7. run the tests in the CI
8. import the results back to Jira

Usually, you would start by having a Story, or similar (e.g. "requirement"), to describe the behaviour of a certain feature and use that to drive your testing.



Having those to guide testing, we could then move to our code to describe and implement the Cucumber test scenarios.

Test code is stored inside the `step-definitions` directory. We also have other directories present, to hold for instance the page object definitions in the `pageobjects` directory.

In this case, we've organized them as follows:

- `step-definitions`/steps.js: step implementation files, in JavaScript.

- **step-definitions/steps.js**

```
const { Given, When, Then } = require('@cucumber/cucumber');

const LoginPage = require('../pageobjects/login.page');
const SecurePage = require('../pageobjects/secure.page');

const pages = {
    login: LoginPage
}

Given(/^I am on the (\w+) page$/, async (page) => {
    await pages[page].open()
});

When(/^I login with (\w+) and (.+)$/, async (username, password) => {
    await LoginPage.login(username, password)
});

Then(/^I should see a flash message saying (.*)$/, async (message) => {
    await expect(SecurePage.flashAlert).toBeExisting();
    await expect(SecurePage.flashAlert).toHaveTextContaining(message);
});
```

- `pageobjects`: abstraction of different pages, somehow based on the page-objects model
  - **pageobjects/page.js**

```
/**
* main page object containing all methods, selectors and functionality
* that is shared across all page objects
*/
module.exports = class Page {
    /**
    * Opens a sub page of the page
    * @param path path of the sub page (e.g. /path/to/page.html)
    */
    open (path) {
        return browser.url(`https://the-internet.herokuapp.com/${path}`)
    }
}
```

- **pageobjects/login-page.js**

```
const Page = require('./page');

/**
 * sub page containing specific selectors and methods for a specific page
 */
class LoginPage extends Page {
    /**
     * define selectors using getter methods
     */
    get inputUsername () { return $('#username') }
    get inputPassword () { return $('#password') }
    get btnSubmit () { return $('button[type="submit"]') }

    /**
     * a method to encapsule automation code to interact with the page
     * e.g. to login using username and password
     */
    async login (username, password) {
        await (await this.inputUsername).setValue(username);
        await (await this.inputPassword).setValue(password);
        await (await this.btnSubmit).click();
    }

    /**
     * overwrite specifc options to adapt it to page object
     */
    open () {
        return super.open('login');
    }
}

module.exports = new LoginPage();
```

- **pageobjects/secure-page.js**

```
const Page = require('./page');

/**
 * sub page containing specific selectors and methods for a specific page
 */
class SecurePage extends Page {
    /**
     * define selectors using getter methods
     */
    get flashAlert () { return $('#flash') }
}

module.exports = new SecurePage();
```

- `features/login`.feature: Cucumber .feature files, containing the tests as Gherkin Scenario(s)/Scenario Outline(s). Please note that each "Feature: <..>" section should be tagged with the issue key of the corresponding "requirement"/story in Jira. You may need to add a prefix (e.g. "REQ_") before the issue key, depending on an Xray global setting.

features/login.feature

```
@REQ_XT-225
Feature: Login feature

        Scenario: Test Login feature
                Scenario Outline: As a user, I can log into the secure area
                                Given I am on the login page
                                When I login with <username> and <password>
                                Then I should see a flash message saying <message>

                                        Examples:
                                                | username | password             |
message                         |
                                                | tomsmith | SuperSecretPassword! | You logged
into a secure area! |
                                                | foobar   | barfoo               | Your username
is invalid.      |
```

Before running the tests in the CI environment, you need to import your .feature files to Xray/Jira; you can invoke the REST API directly or use one of the available plugins/tutorials for CI tools.

example of a shell script to import/synchronize Scenario(s)/Background(s) from .features to Jira and Xray

```
#!/bin/bash

JIRA_BASEURL=https://192.168.0.168
JIRA_USERNAME=admin
JIRA_PASSWORD=admin
PROJECT=XT
FILE=features.zip

zip -r $FILE features/ -i \*.feature
curl -H "Content-Type: multipart/form-data" -u $JIRA_USERNAME:$JIRA_PASSWORD -F "file=@$FILE" "$JIRA_BASEURL
/rest/raven/1.0/import/feature?projectKey=$PROJECT"
```

Please note

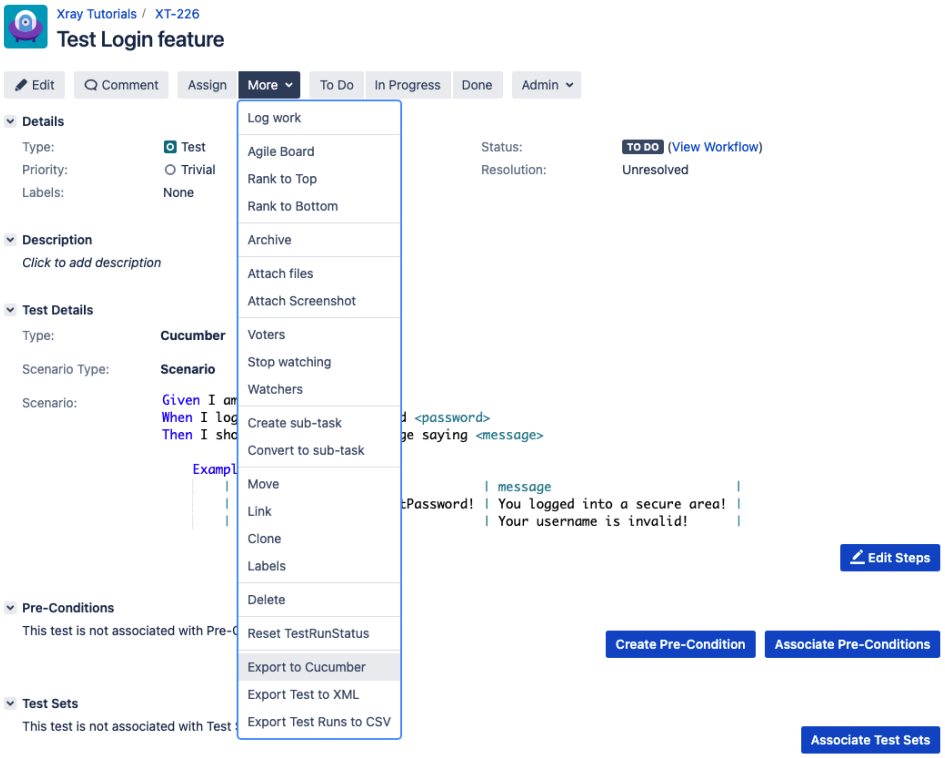
Each Scenario of each .feature will be created as a Test issue that contains unique identifiers, so that if you import once again then Xray can update the existent Test and don't create any duplicated tests.

Afterward, you can export those features out of Jira based on some criteria, so they are properly tagged with corresponding issue keys; this is important because results need to contain these references.

You can then export the specification of the test to a Cucumber .feature file via the REST API, or the **Export to Cucumber** UI action from within the Test /Test Execution issue or even based on an existing saved filter. A plugin for your CI tool of choice can be used to ease this task.

So, you can either:

- use the UI



- use the REST API (more info here)
  - **example of a shell script to export/generate .features from Xray**

```
#!/bin/bash

JIRA_BASEURL=https://192.168.2.168
JIRA_USERNAME=admin
JIRA_PASSWORD=admin
KEYS="XT-142"

rm -f features.zip
curl -u $JIRA_USERNAME:$JIRA_PASSWORD  "$JIRA_BASEURL/rest/raven/2.0/export/test?
keys=$KEYS&fz=true" -o features.zip
unzip -o features.zip  -d features
```

- use one of the available CI/CD plugins (e.g. see an example of Integration with Jenkins)

For CI only purposes, we will export the features to a new temporary directory named `features/` on the root folder of your project. Please note that while implementing the tests, .feature files should be edited inside their respective folder.

After being exported, the created .feature(s) will contain references to the Test issue keys, eventually prefixed (e.g. "TEST_") depending on an Xray global setting, and the covered "requirement" issue key, if that's the case. The naming of these files is detailed in Export Cucumber Features backup.

features/1_COM_19.feature

```
@REQ_XT-225
Feature: Login feature

        @TEST_XT-226
        Scenario Outline: As a user, I can log into the secure area
                Given I am on the login page
                When I login with <username> and <password>
                Then I should see a flash message saying <message>

                        Examples:
                                | username | password             | message                         |
                                | tomsmith | SuperSecretPassword! | You logged into a secure area! |
                                | foobar   | barfoo               | Your username is invalid.      |
```

To run the tests and produce Cucumber JSON reports(s), we will use the following command:

```
npx wdio run ./wdio.conf.js
```

This will produce one Cucumber JSON report in `.tmp/json` directory per each .feature file.

After running the tests, results can be imported to Xray via the REST API, or the **Import Execution Results** action within the Test Execution, or by using one of the available CI/CD plugins (e.g. see an example of Integration with Jenkins).

Example of shell script to import results (e.g. import_results_cloud.sh)

```
#!/bin/bash

JIRA_BASEURL=https://192.168.0.168
JIRA_USERNAME=admin
JIRA_PASSWORD=admin

curl -H "Content-Type: application/json" -X POST -u $JIRA_USERNAME:$JIRA_PASSWORD --data @".tmp/json/login-feature.json" $JIRA_BASEURL/rest/raven/2.0/import/execution/cucumber
```

Which Cucumber endpoint/"format" to use?

To import results, you can use two different endpoints/"formats" (endpoints described in Import Execution Results - REST):

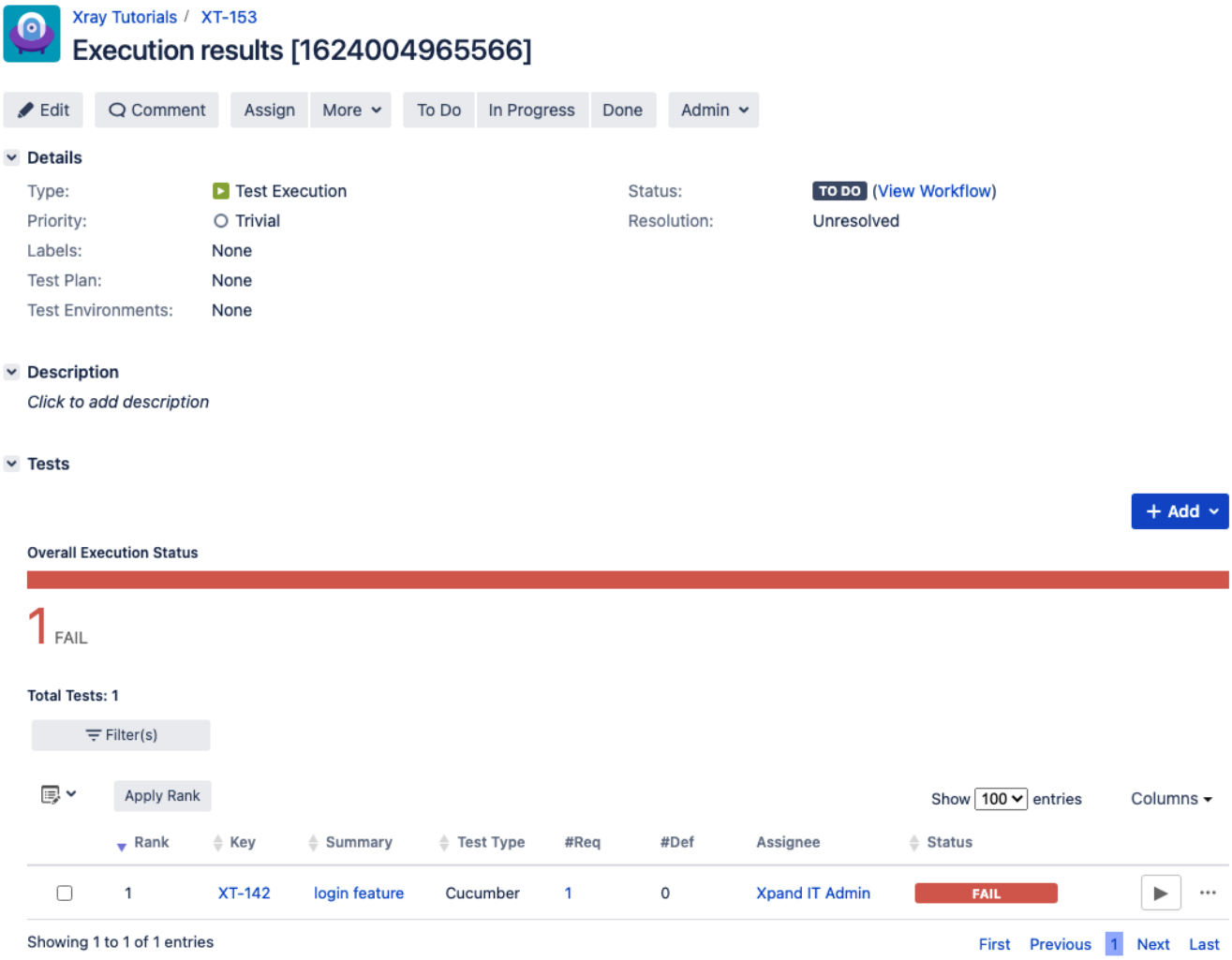
1. the "standard cucumber" endpoint
2. the "multipart cucumber" endpoint

The standard cucumber endpoint (i.e. */import/execution/cucumber)* is simpler but more restrictive: you cannot specify values for custom fields on the Test Execution that will be created. This endpoint creates new Test Execution issues unless the Feature contains a tag having an issue key of an existing Test Execution.

The multipart cucumber endpoint will allow you to customize fields (e.g. Fix Version, Test Plan) if you wish to do so, on the Test Execution that will be created. Note that this endpoint always creates new Test Executions (as of Xray v4.2).

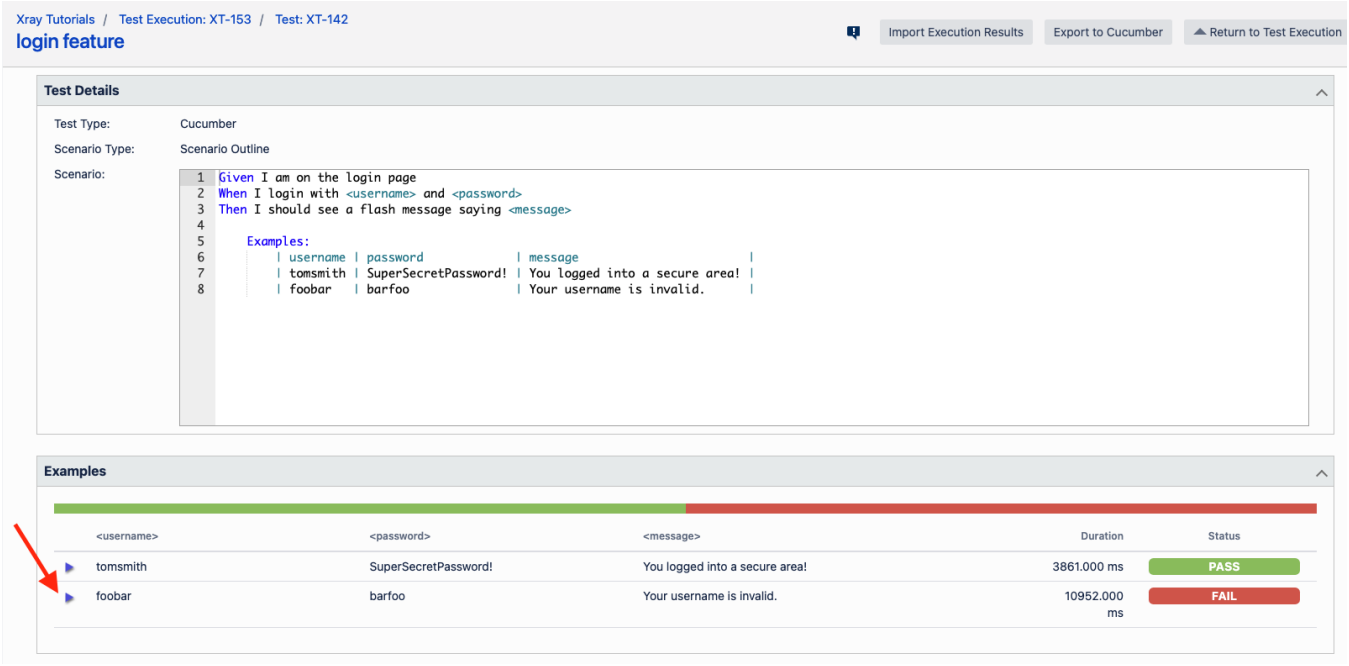
In sum, if you want to customize the Fix Version, Test Plan and/or Test Environment of the Test Execution issue that will be created, you'll have to use the "multipart cucumber" endpoint.

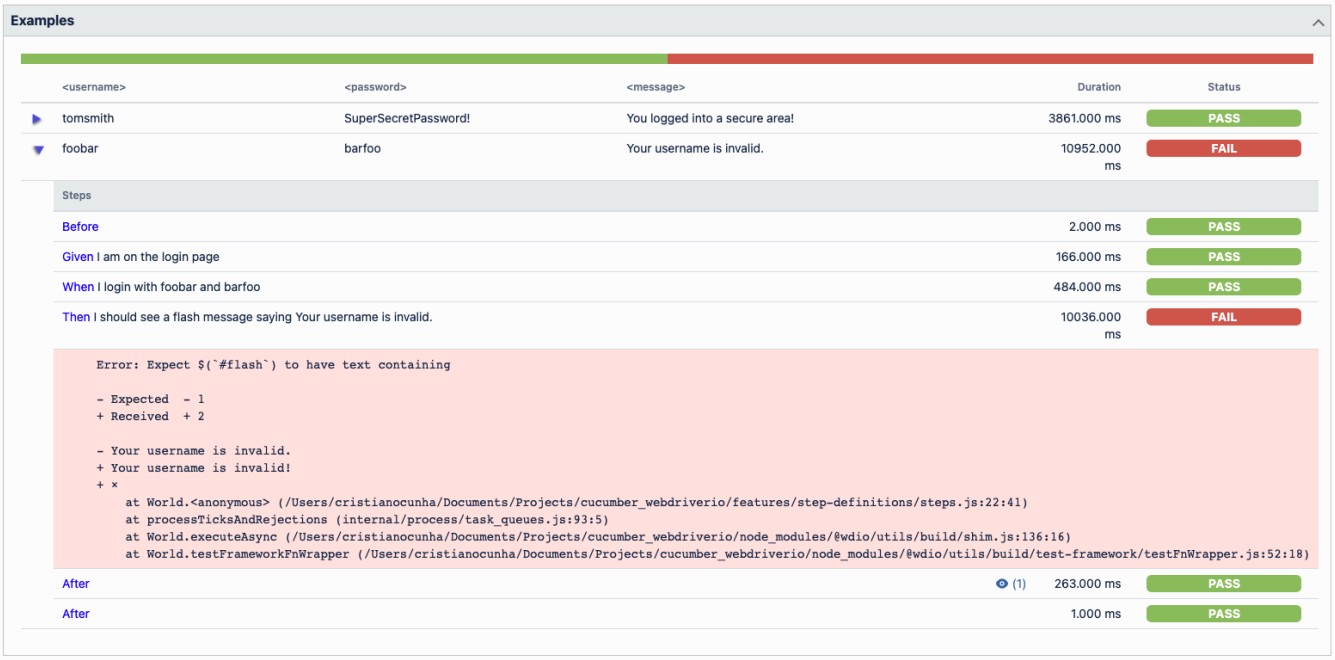
A new Test Execution will be created (unless you originally exported the Scenarios/Scenario Outlines from a Test Execution).



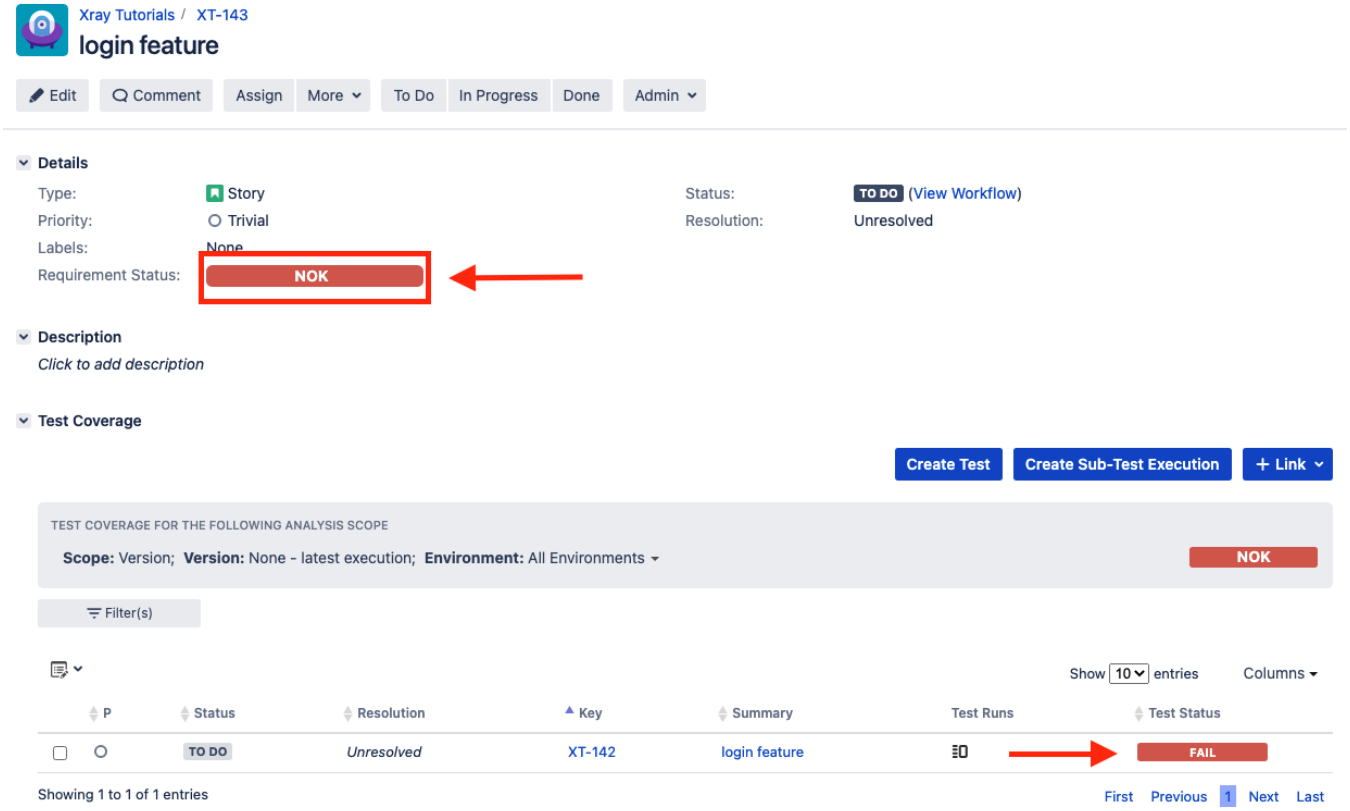
One of the tests fails (on purpose).

The execution screen details of the Test Run will provide overall status information and Gherkin statement-level results, therefore we can use it to analyse the failing test.





Results are reflected on the covered item (e.g. Story). On its issue screen, coverage now shows that the item is OK based on the latest testing results, that can also be tracked within the Test Coverage panel bellow.



If we change the specification (i.e. the Gherkin scenarios), we need to import the .feature(s) once again.

Therefore, in the CI we always need to start by importing the .feature file(s) to keep Jira/Xray on synch.

FAQ and Recommendations

Please see this page.

References

- WebDriverIO
- WebDriverIO documentation