Testing using Cypress and Cucumber in JavaScript

Overview

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

Source-code for this tutorial

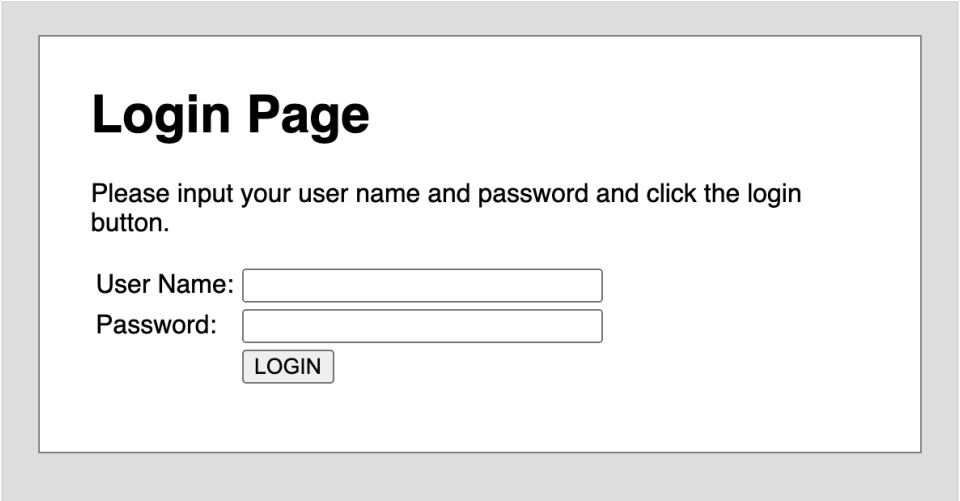
Code is available in GiHub; the repo contains some auxiliary scripts.

Requirements

- nodejs
- npm packages
  - cypress
  - cypress-cucumber-preprocessor
  - cucumber-json-merge

Description

For the purpose of this tutorial, we will use a dummy website (source-code here) containing just a few pages to support login/logout kind of features; we aim to test precisely those features.



We need to configure Cypress to use the cypress-cucumber-preprocessor, which provides the ability to understand .feature files and also to produce Cucumber JSON reports.

cypress/plugins/index.js

```
const cucumber = require('cypress-cucumber-preprocessor').default

/**
 * @type {Cypress.PluginConfig}
 */
module.exports = (on, config) => {
  // `on` is used to hook into various events Cypress emits
  // `config` is the resolved Cypress config
  on('file:preprocessor', cucumber())
}
```

In Cypress' main configuration file, define the base URL of the website under test, the regex of the files that contain the test scenarios (i.e. <...>.feature files). Other options may be defined e.g for bypassing chromeWebSecurity, additional reporters, the ability to upload results to Cypress infrastructure in the cloud, etc).

/cypress.json

```
{
  "baseUrl": "https://robotwebdemo.herokuapp.com/",
  "testFiles": "**/*.feature",
  "ignoreTestFiles": [
    "*.js",
    "*.md"
  ],
  "reporter": "junit",
  "reporterOptions": {
    "mochaFile": "test-results/test-output-[hash].xml"
  },
  "chromeWebSecurity": false,
  "projectId": "bfi83g"
}
```

Next, here is an example of the contents of package.json.

package.json

```
{
  "name": "cypress-cucumber-robotdemo",
  "version": "1.0.0",
  "description": "An example for Cypress and Cucumber usage using Robot login demo website",
  "main": "index.js",
  "scripts": {
    "cypress:open:local": "CYPRESS_ENV=localhost npm run cypress:open",
    "cypress:open:prod": "CYPRESS_ENV=production npm run cypress:open",
    "cypress:open": "cypress open",
    "test:local": "CYPRESS_ENV=localhost npm run test  --spec 'cypress/integration/**/*.feature",
    "test:prod": "CYPRESS_ENV=production npm run test",
    "test": "cypress run --spec 'features/**/*.feature' --config integrationFolder=.",
    "test:debug:local": "CYPRESS_ENV=localhost npm run test:debug",
    "test:debug:prod": "CYPRESS_ENV=production npm run test:debug",
    "test:debug": "cypress run --headed --browser chrome --env TAGS='@e2e-test' --spec 'cypress/integration/**
/*.feature'",
    "test:pull-features": "git submodule update --remote gherkin-features && cp -rf gherkin-features/* cypress
/integration && node ./scripts/remove-old-features.js",
    "attach_screenshots": "node attach_screenshots.js"
  },
  "author": "",
  "license": "Private",
  "dependencies": {
    "axios": "^0.18.0",
    "cucumber-json-merge": "0.0.4",
    "fs-extra": "^7.0.1",
    "glob": "^7.1.3"
  },
  "devDependencies": {
    "cypress": "^5.5.0",
    "cypress-cucumber-preprocessor": "^4.0.0",
    "eslint": "^5.13.0",
    "eslint-config-airbnb-base": "^12.1.0",
    "eslint-config-prettier": "^2.9.0",
    "eslint-plugin-import": "^2.11.0",
    "eslint-plugin-prettier": "^2.6.0",
    "husky": "^1.3.1",
    "lint-staged": "^8.1.3"
  },
  "cypress-cucumber-preprocessor": {
    "nonGlobalStepDefinitions": true,
    "cucumberJson": {
      "generate": true,
      "outputFolder": "cypress/cucumber-json",
      "filePrefix": "",
      "fileSuffix": ".cucumber"
    }
  },
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.js": [
      "eslint",
      "git add"
    ]
  }
}
```

Before moving into the actual implementation, we need to decide which workflow we'll use: do we want to use Xray/Jira as the master for writing the declarative specification (i.e. the Gherkin based Scenarios), or do we 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 others prefers to edit them by writing the .feature files by hand using some IDE.

Using Jira and Xray as master

This section assumes you will use 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).

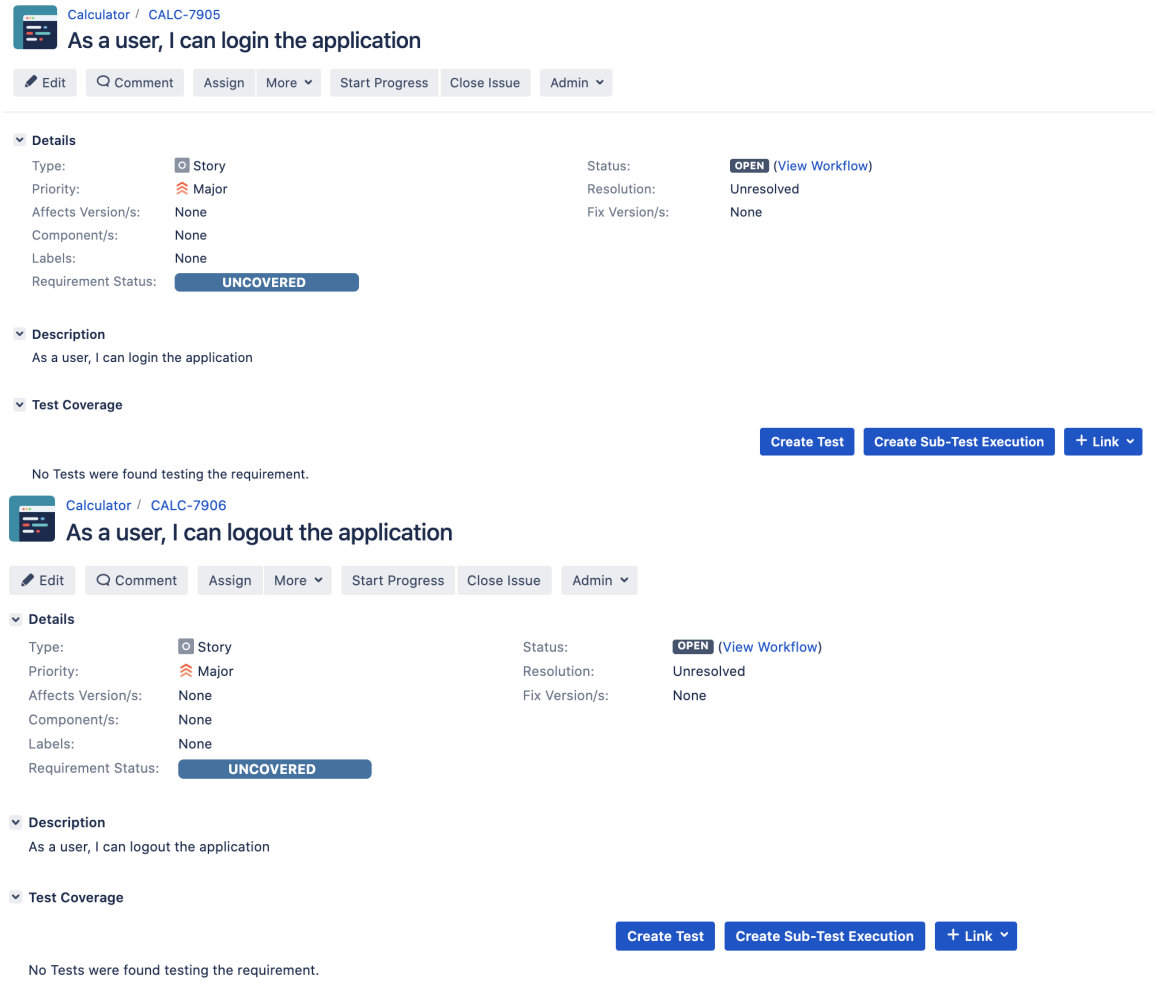
The overall flow would be something like this:

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.

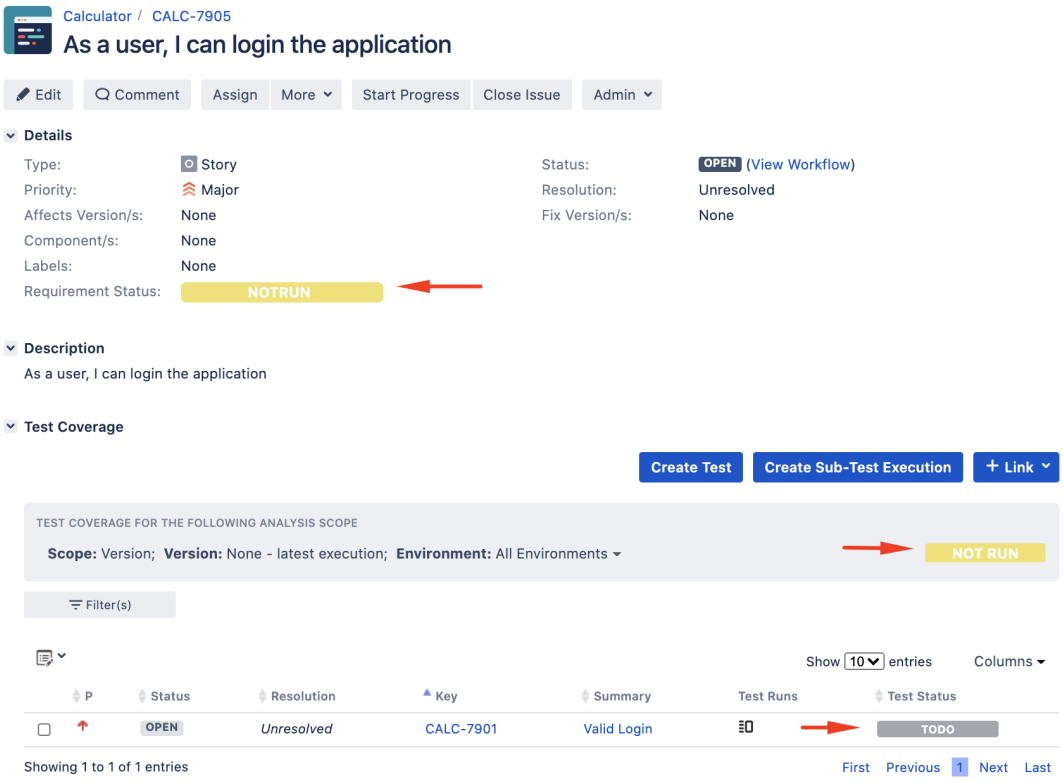


In this case, we'll create a Cucumber Test, of Cucumber Type "Scenario."

We can fill out the Gherkin statements immediately on the Jira issue "create dialog" or we can create the Test issue first and fill out the details on the next screen, from within the Test issue. In the latter case, we can take advantage of the built-in Gherkin editor which provides auto-complete for Gherkin steps.

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

The coverage and the test results can be tracked in the "requirement" 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, 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 Cypress, the test code is stored under `cypress/integration` directory, which itself contains several other directories. In this case, we've organized them as follows:

- `cypress/integration/common`: step implementation files, in JavaScript.

- **cypress/integration/common/login.js**

```
import { Given, When } from 'cypress-cucumber-preprocessor/steps';
import LoginPage from '../../pages/login-page';
import LoginResultsPage from '../../pages/login-results-page';

Given(/^browser is opened to login page$/, () => {
  LoginPage.visit();
});

When('user {string} logs in with password {string}', (username, password) => {
  LoginPage.enter_username(username);
  LoginPage.enter_password(password);
  LoginPage.pressLogin();
});

Then(/^welcome page should be open$/, () => {
    LoginResultsPage.expect().toBeSuccessful();
  });

Then(/^error page should be open$/, () => {
    LoginResultsPage.expect().toBeUnsuccessful();
  });
```

- **cypress/integration/common/logout.js**

```
import { Given, When } from 'cypress-cucumber-preprocessor/steps';
import LoginPage from '../../pages/login-page';
import LoginResultsPage from '../../pages/login-results-page';

Given(/^browser is opened to login page$/, () => {
  LoginPage.visit();
});

When('user {string} logs in with password {string}', (username, password) => {
  LoginPage.enter_username(username);
  LoginPage.enter_password(password);
  LoginPage.pressLogin();
});

Then(/^welcome page should be open$/, () => {
    LoginResultsPage.expect().toBeSuccessful();
  });

Then(/^error page should be open$/, () => {
    LoginResultsPage.expect().toBeUnsuccessful();
  });
```

- `cypress/integration/pages`: abstraction of different pages, somehow based on the page-objects model

- **cypress/integration/pages/login.js**

```
import LoginResultsPage from './login-results-page';

const USERNAME_FIELD = 'input[id=username_field]';
const PASSWORD_FIELD = 'input[id=password_field]';
const LOGIN_BUTTON = 'input[type=submit]';
const LOGIN_TEXT = 'LOGIN';


class LoginPage {
  static visit() {
    cy.visit('/');
  }


  static enter_username(username) {
    cy.get(USERNAME_FIELD)
      .type(username);
  }

  static enter_password(password) {
    cy.get(PASSWORD_FIELD)
      .type(password);
  }

  static pressLogin() {
    cy.get(LOGIN_BUTTON).contains(LOGIN_TEXT)
      .click();
    return new LoginResultsPage();
  }

}

export default LoginPage;
```

- **cypress/integration/pages/login-results-page.js**

```
const RESULT_HEADER = 'h1';

class LoginResultsPage {
    static expect() {
      return {
        toBeSuccessful: () => {
          cy.get(RESULT_HEADER).should('have.text', 'Welcome Page')
        },

        toBeUnsuccessful: () => {
          cy.get(RESULT_HEADER).should('have.text', 'Error Page')
        },
      };
    }
  }

export default LoginResultsPage;
```

- **cypress/integration/pages/logout-results-page.js**

```
const RESULT_HEADER = 'h1';

class LogoutResultsPage {
    static expect() {
      return {
        toBeSuccessful: () => {
          cy.get(RESULT_HEADER).should('have.text', 'Login Page')
        },
      };
    }
  }

export default LogoutResultsPage;
```

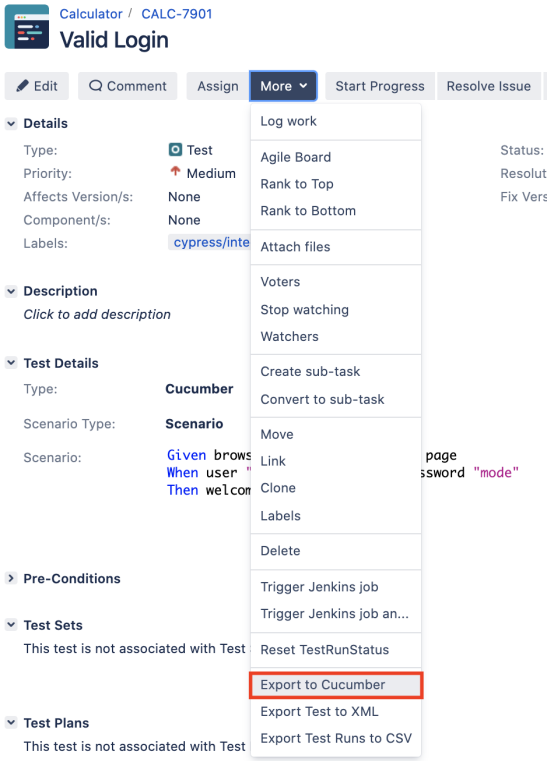
- **cypress/integration/pages/welcome-page.js**

```
import LoginPage from './login-page';

const LOGOUT_LINK = 'a';
const LOGOUT_TEXT = 'logout';


class WelcomePage {
  static visit() {
    cy.visit('/welcome.html');
  }


  static pressLogout() {
    cy.get(LOGOUT_LINK).contains(LOGOUT_TEXT)
      .click();
    return new LoginPage();
  }

}

export default WelcomePage;
```

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)

```
#!/bin/bash

rm -f features/*.feature
curl -u admin:admin  "http://jiraserver.example.com/rest/raven/1.0/export/test?keys=CALC-7905;CALC-7906&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 Cypress project (we'll need to tell Cypress to use this folder).

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.

features/1_CALC-7905.feature

```
@REQ_CALC-7905
Feature: As a user, I can login the application
        #As a user, I can login the application


        @TEST_CALC-7903
        Scenario Outline: Login With Invalid Credentials Should Fail
                Given browser is opened to login page
                When user "<username>" logs in with password "<password>"
                Then error page should be open

                        Examples:
                                | username | password |
                                | invalid  | mode     |
                                | demo     | invalid  |
                                | invalid  | invalid  |
                                | demo     | mode     |


        @TEST_CALC-7902
        Scenario: Invalid Login
                Given browser is opened to login page
                When user "dummy" logs in with password "password"
                Then error page should be open


        @TEST_CALC-7901
        Scenario: Valid Login
                Given browser is opened to login page
                When user "demo" logs in with password "mode"
                Then welcome page should be open
```

features/1_CALC-7906.feature

```
@REQ_CALC-7906
Feature: As a user, I can logout the application
        #As a user, I can logout the application


        @TEST_CALC-7904
        Scenario: Valid Logout
                Given user is on the welcome page
                When user chooses to logout
                Then login page should be open
```

To run the tests and produce Cucumber JSON reports(s), we can either use `npm` or `cypress` command directly.

```
npm run test

# or instead...

node_modules/cypress/bin/cypress run --spec 'features/**/*.feature' --config integrationFolder=.
```

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

The cypress-cucumber-preprocessor package, as of v4.0.0, does not produce reports containing the screenshots embedded.

However, the following script (credits to the user that provided it on GitHub) can be used to update the previous JSON reports so that they contain the screenshots of the failed tests.

attach_screenshots.js

```
const fs = require('fs-extra')
const path = require('path')
const chalk = require('chalk')

const cucumberJsonDir = './cypress/cucumber-json'
const cucumberReportFileMap = {}
const cucumberReportMap = {}
const jsonIndentLevel = 2
const ReportDir = './cypress/reports/cucumber-report'
const screenshotsDir = './cypress/screenshots'

getCucumberReportMaps()
addScreenshots()

//Mapping cucumber json files from the cucumber-json directory to the features
function getCucumberReportMaps() {
    const files = fs.readdirSync(cucumberJsonDir).filter(file => {
        return file.indexOf('.json') > -1
    })
    files.forEach(file => {
        const json = JSON.parse(
            fs.readFileSync(path.join(cucumberJsonDir, file))
        )
        if (!json[0]) { return }
        const [feature] = json[0].uri.split('/').reverse()
        cucumberReportFileMap[feature] = file
        cucumberReportMap[feature] = json
    })
}

//Adding screenshots to the respective failed test steps in the feature files
function addScreenshots() {

    const prependPathSegment = pathSegment => location => path.join(pathSegment, location)

    const readdirPreserveRelativePath = location => fs.readdirSync(location).map(prependPathSegment(location))

    const readdirRecursive = location => readdirPreserveRelativePath(location)
        .reduce((result, currentValue) => fs.statSync(currentValue).isDirectory()
            ? result.concat(readdirRecursive(currentValue))
            : result.concat(currentValue), [])
    const screenshots = readdirRecursive(path.resolve(screenshotsDir)).filter(file => {
        return file.indexOf('.png') > -1
    })

    const featuresList = Array.from(new Set(screenshots.map(x => x.match(/[\w-_.]+\.feature/g)[0])))
    featuresList.forEach(feature => {
        screenshots.forEach(screenshot => {

            const regex = /(?<=\ --\ ).*?((?=\ \(example\ \#\d+\))|(?=\ \(failed\)))/g
            const [scenarioName] = screenshot.match(regex)
            console.info(chalk.blue('\n Adding screenshot to cucumber-json report for'))
            console.info(chalk.blue(scenarioName))

            console.log(featuresList)
            console.log(feature)
            console.log(cucumberReportMap)
            const myScenarios = cucumberReportMap[feature][0].elements.filter(
                e => scenarioName.includes(e.name)
            )
            if (!myScenarios) { return }
```

```
            let foundFailedStep = false
            myScenarios.forEach(myScenario => {
                if (foundFailedStep) {
                    return
                }
                let myStep
                if (screenshot.includes('(failed)')) {
                    myStep = myScenario.steps.find(
                        step => step.result.status === 'failed'
                    )
                } else {
                    myStep = myScenario.steps.find(
                        step => step.name.includes('screenshot')
                    )
                }
                if (!myStep) {
                    return
                }
                const data = fs.readFileSync(
                    path.resolve(screenshot)
                )
                if (data) {
                    const base64Image = Buffer.from(data, 'binary').toString('base64')
                    if (!myStep.embeddings) {
                        myStep.embeddings = []
                        myStep.embeddings.push({ data: base64Image, mime_type: 'image/png' })
                        foundFailedStep = true
                    }
                }
            })
            //Write JSON with screenshot back to report file.
            fs.writeFileSync(
                path.join(cucumberJsonDir, cucumberReportFileMap[feature]),
                JSON.stringify(cucumberReportMap[feature], null, jsonIndentLevel)
            )
        })
    })
}
```

The cucumber-json-merge utility may be handy to merge the results of each feature, so they can be then submitted to Xray as one single file.

Next, is an example of a shell script with all these steps.

example of a Bash script to run the tests and produce a unified Cucumber JSON report

```
#!/bin/bash

rm -f cypress/cucumber-json/*
npm run test
npm run attach_screenshots
cucumber-json-merge  -d cypress/cucumber-json/
```

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).

```
curl -H "Content-Type: application/json" -X POST -u admin:admin --data @"report.json" http://jiraserver.example.
com/rest/raven/1.0/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):

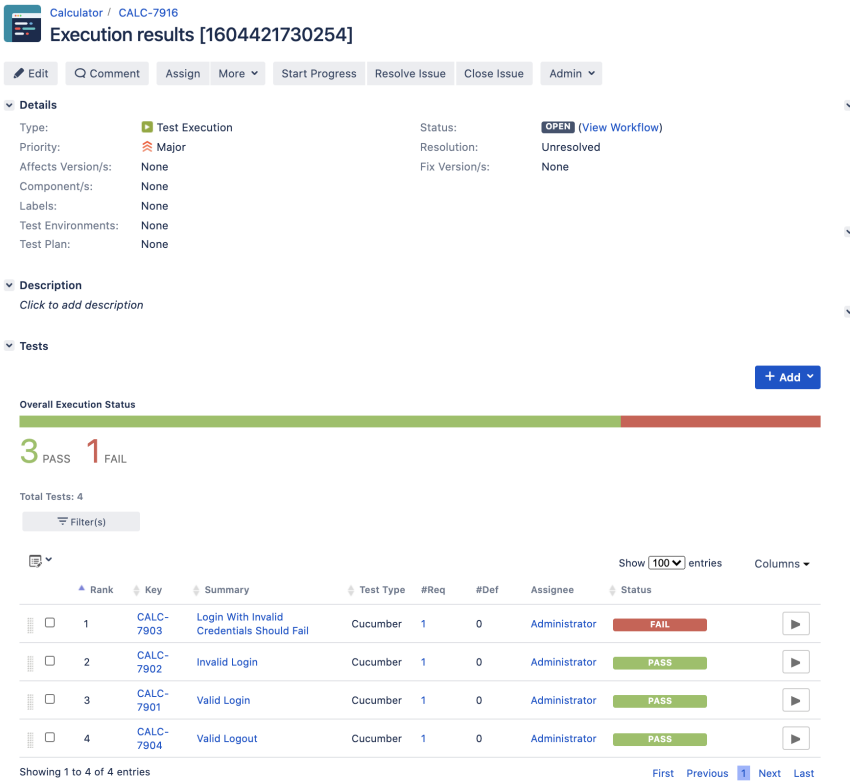
1. the "standard cucumber" endpoint
2. the "multipart cucumber" endpoint

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).

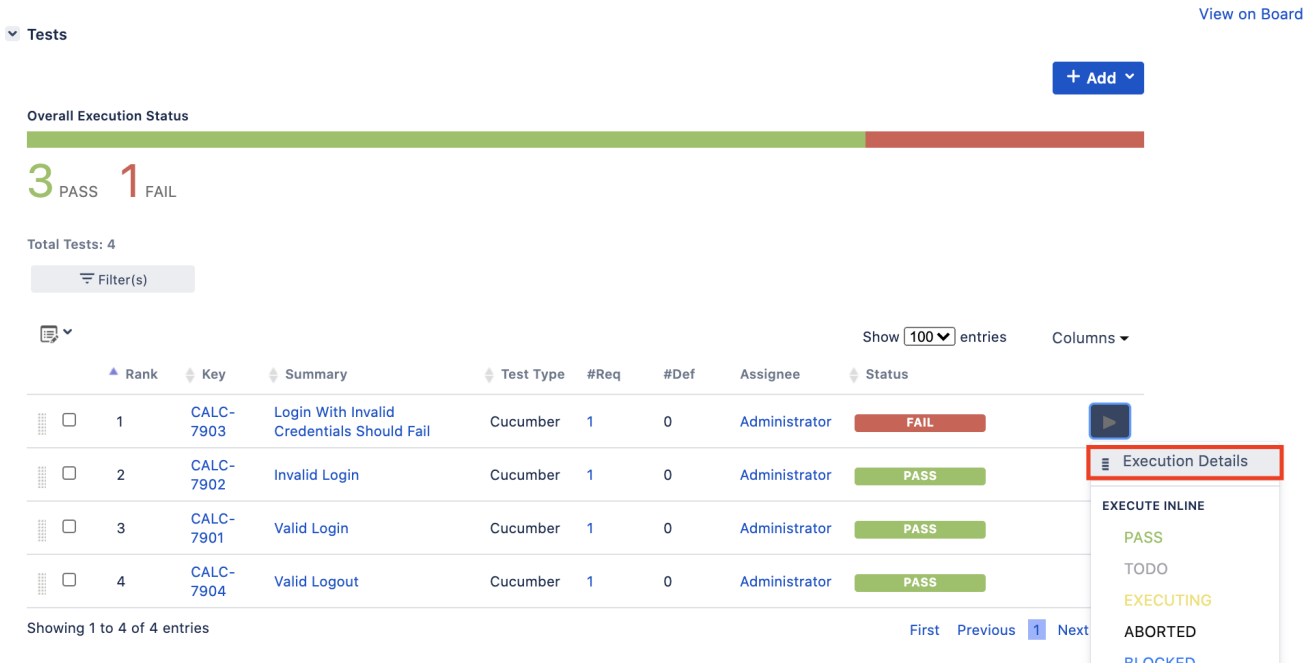
In sum, if you want to customize the Fix Version, Test Plan and/or Test Environment of the Test Execution issue that will be created, you'll have to use the "multipart cucumber" endpoint.

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

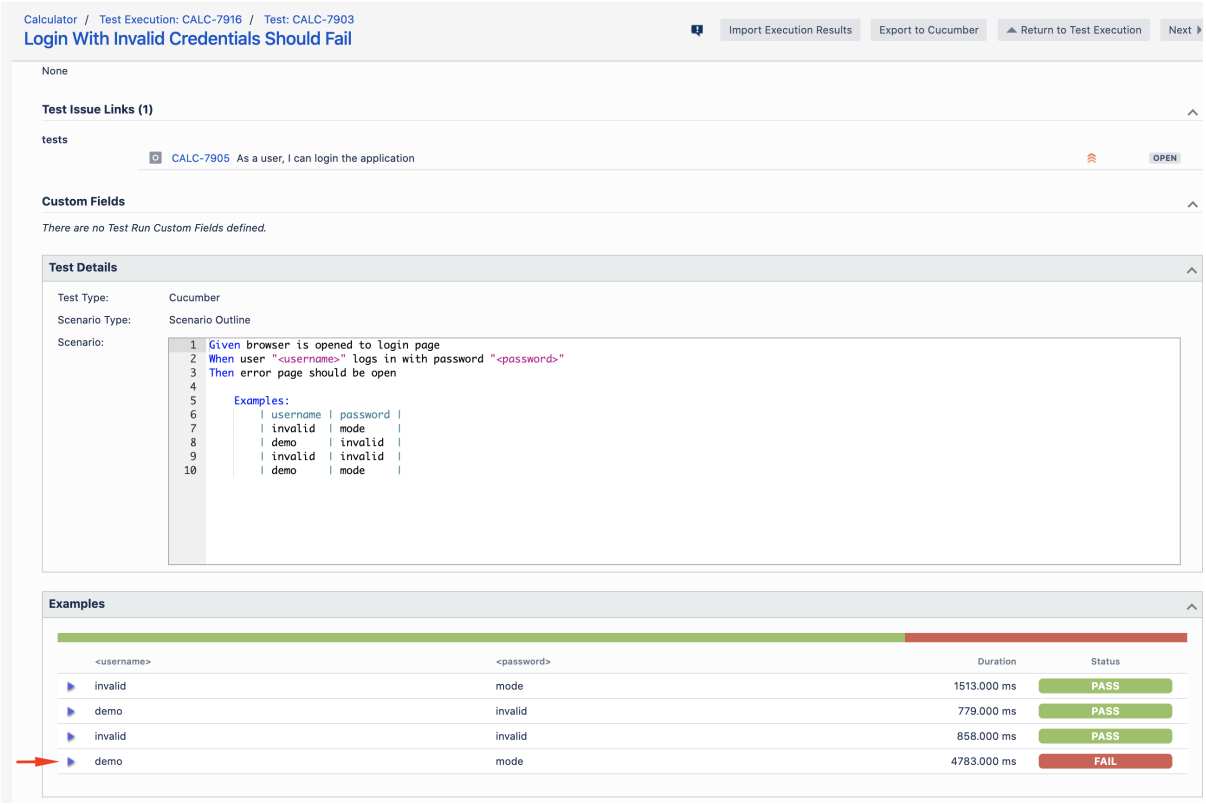


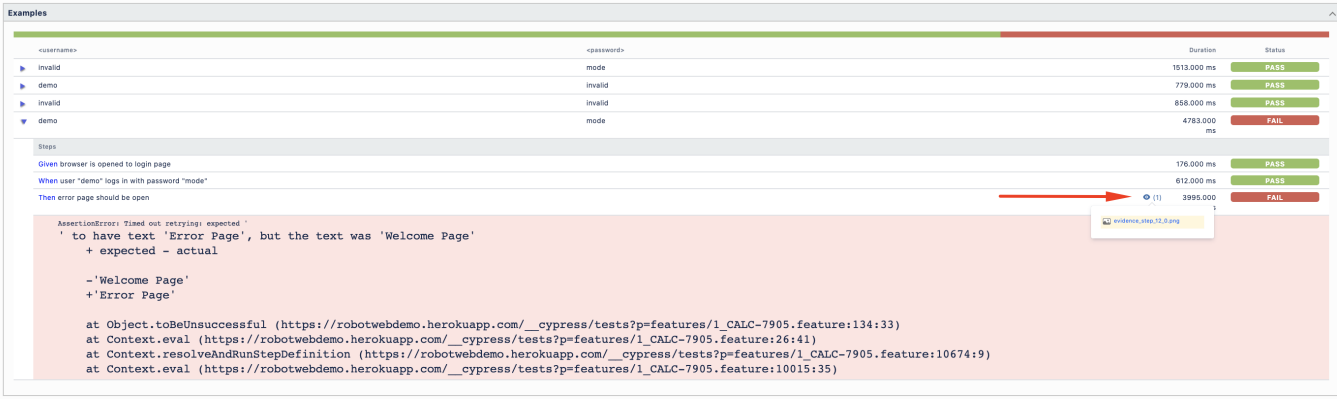
One of the tests fails (on purpose).

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



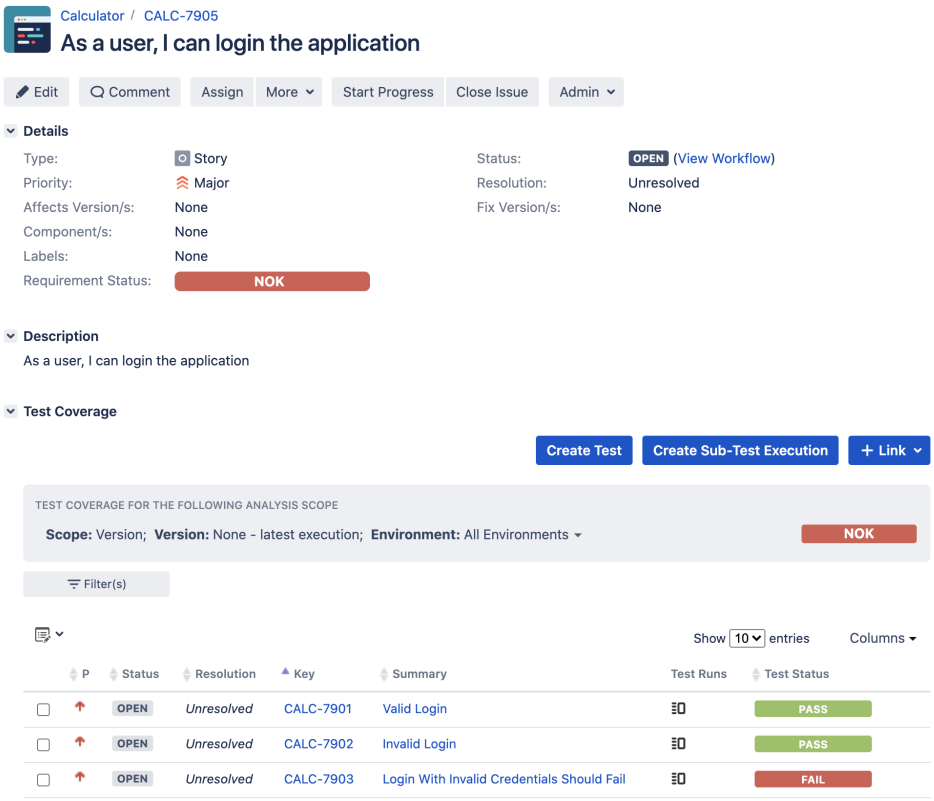
A given example can be expanded to see all Gherkin statements and, if available, it is possible to see also the attached screenshot(s).





Note: in this case, the bug was on the Scenario Outline example which was using a valid username/password combination.

Results are reflected on the covered item (e.g. Story). On the issue screen, coverage now shows that the item is OK based on the latest testing results which 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.

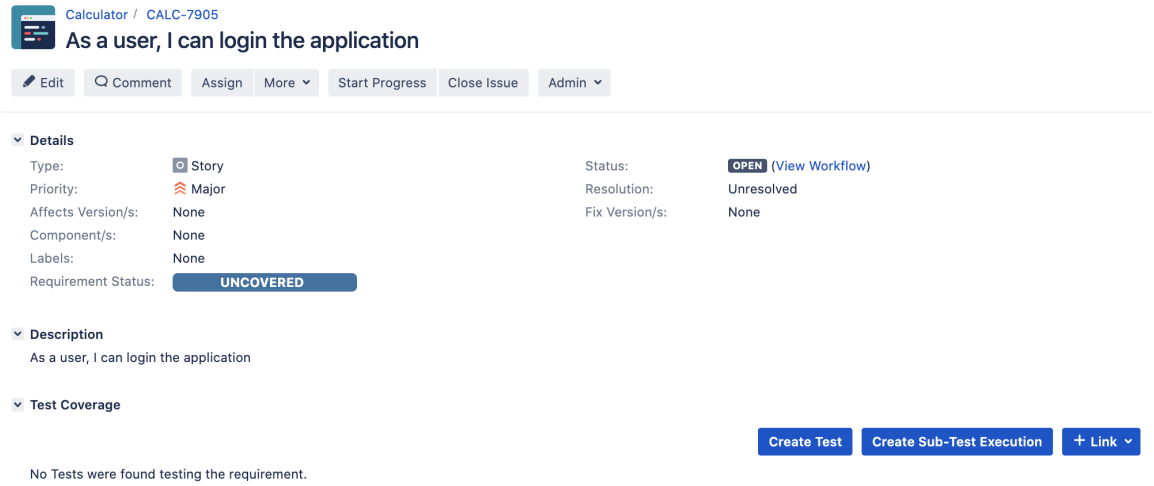
In any case, you'll need to synchronize your .feature files to Jira so that you can have visibility of them and report results against them.

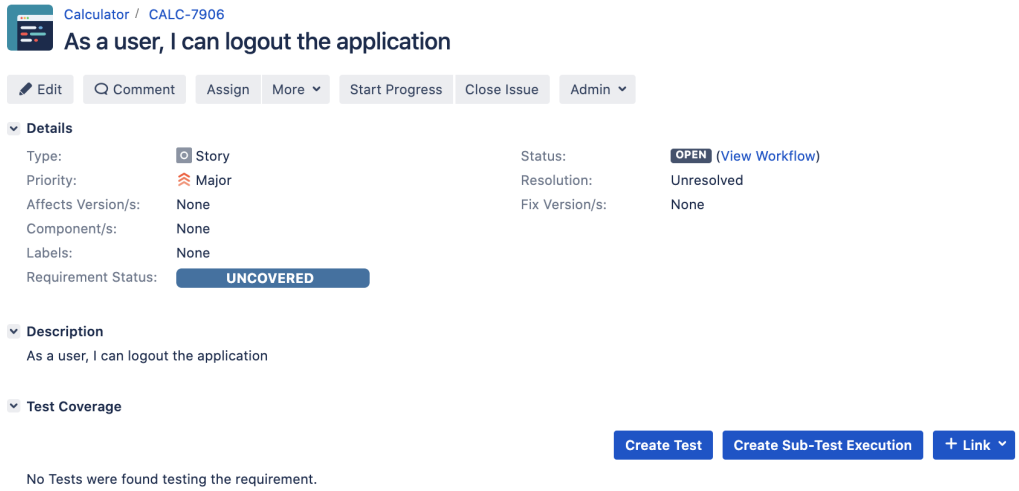
The overall flow would be something like this:

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 supporting Cypress 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/synchronize 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 Cypress 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 behavior of a certain feature and use that to drive your testing.





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

In Cypress, test related code is stored inside the `cypress/integration` directory, which itself contains several other directories. In this case, we've organized them as follows:

- `cypress/integration/common`: step implementation files, in JavaScript.

- **cypress/integration/common/login.js**

```
import { Given, When } from 'cypress-cucumber-preprocessor/steps';
import LoginPage from '../../pages/login-page';
import LoginResultsPage from '../../pages/login-results-page';

Given(/^browser is opened to login page$/, () => {
  LoginPage.visit();
});

When('user {string} logs in with password {string}', (username, password) => {
  LoginPage.enter_username(username);
  LoginPage.enter_password(password);
  LoginPage.pressLogin();
});

Then(/^welcome page should be open$/, () => {
    LoginResultsPage.expect().toBeSuccessful();
  });

Then(/^error page should be open$/, () => {
    LoginResultsPage.expect().toBeUnsuccessful();
  });
```

- **cypress/integration/common/logout.js**

```
import { Given, When } from 'cypress-cucumber-preprocessor/steps';
import LoginPage from '../../pages/login-page';
import LoginResultsPage from '../../pages/login-results-page';

Given(/^browser is opened to login page$/, () => {
  LoginPage.visit();
});

When('user {string} logs in with password {string}', (username, password) => {
  LoginPage.enter_username(username);
  LoginPage.enter_password(password);
  LoginPage.pressLogin();
});

Then(/^welcome page should be open$/, () => {
    LoginResultsPage.expect().toBeSuccessful();
  });

Then(/^error page should be open$/, () => {
    LoginResultsPage.expect().toBeUnsuccessful();
  });
```

- `cypress/integration/pages`: abstraction of different pages, somehow based on the page-objects model

- **cypress/integration/pages/login.js**

```
import LoginResultsPage from './login-results-page';

const USERNAME_FIELD = 'input[id=username_field]';
const PASSWORD_FIELD = 'input[id=password_field]';
const LOGIN_BUTTON = 'input[type=submit]';
const LOGIN_TEXT = 'LOGIN';


class LoginPage {
  static visit() {
    cy.visit('/');
  }


  static enter_username(username) {
    cy.get(USERNAME_FIELD)
      .type(username);
  }

  static enter_password(password) {
    cy.get(PASSWORD_FIELD)
      .type(password);
  }

  static pressLogin() {
    cy.get(LOGIN_BUTTON).contains(LOGIN_TEXT)
      .click();
    return new LoginResultsPage();
  }

}

export default LoginPage;
```

- **cypress/integration/pages/login-results-page.js**

```
const RESULT_HEADER = 'h1';

class LoginResultsPage {
    static expect() {
      return {
        toBeSuccessful: () => {
          cy.get(RESULT_HEADER).should('have.text', 'Welcome Page')
        },

        toBeUnsuccessful: () => {
          cy.get(RESULT_HEADER).should('have.text', 'Error Page')
        },
      };
    }
  }

export default LoginResultsPage;
```

- **cypress/integration/pages/logout-results-page.js**

```
const RESULT_HEADER = 'h1';

class LogoutResultsPage {
    static expect() {
      return {
        toBeSuccessful: () => {
          cy.get(RESULT_HEADER).should('have.text', 'Login Page')
        },
      };
    }
  }

export default LogoutResultsPage;
```

- **cypress/integration/pages/welcome-page.js**

```
import LoginPage from './login-page';

const LOGOUT_LINK = 'a';
const LOGOUT_TEXT = 'logout';


class WelcomePage {
  static visit() {
    cy.visit('/welcome.html');
  }


  static pressLogout() {
    cy.get(LOGOUT_LINK).contains(LOGOUT_TEXT)
      .click();
    return new LoginPage();
  }

}

export default WelcomePage;
```

- `cypress/integration/login`: 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 a global Xray setting.

- **cypress/integration/login/login.feature**

```
@REQ_CALC-7905
Feature: As a user, I can login the applicaiton

Scenario: Valid Login
    Given browser is opened to login page
    When user "demo" logs in with password "mode"
    Then welcome page should be open

Scenario: Invalid Login
    Given browser is opened to login page
    When user "dummy" logs in with password "password"
    Then error page should be open

Scenario Outline: Login With Invalid Credentials Should Fail
    Given browser is opened to login page
    When user "<username>" logs in with password "<password>"
    Then error page should be open

    Examples:
        | username  | password |
        | invalid   |   mode   |
        | demo      | invalid  |
        | invalid   | invalid  |
```

- **cypress/integration/login/logout.feature**

```
@REQ_CALC-7906
Feature: As a user, I can logout the application

Scenario: Valid Logout
    Given user is on the welcome page
    When user chooses to logout
    Then login page should be open
```

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.

```
zip -r features.zip cypress/integration/ -i \*.feature
curl -H "Content-Type: multipart/form-data" -u admin:admin -F "file=@features.zip" "http://jiraserver.example.
com/rest/raven/1.0/import/feature?projectKey=CALC"
```

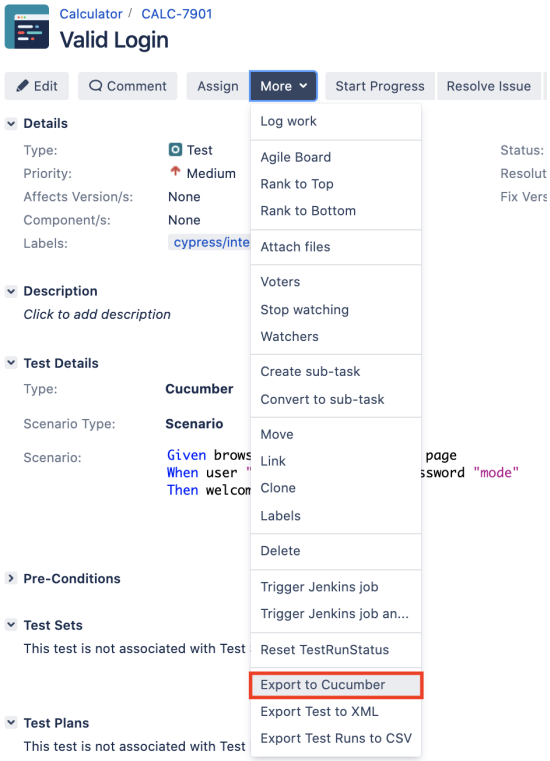
> **Please note**
>
> Each Scenario of each .feature will be created as a Test issue that contains unique identifiers, so that if you import once again then Xray can update the existent Test and don't create any duplicated tests.

Afterwards, 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)

```
#!/bin/bash

rm -f features/*.feature
curl -u admin:admin  "http://jiraserver.example.com/rest/raven/1.0/export/test?keys=CALC-7905;CALC-7906&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 purpose, we will export the features to a new temporary directory named `features/` on the root folder of your Cypress project (we'll need to tell Cypress to use this folder). Please note that while implementing the tests, .feature files should be edited inside the cypress/integration/login folder, in this case;

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.

features/1_CALC-7905.feature

```
@REQ_CALC-7905
Feature: As a user, I can login the application
        #As a user, I can login the application


        @TEST_CALC-7903 @cypress/integration/login/login.feature
        Scenario Outline: Login With Invalid Credentials Should Fail
                Given browser is opened to login page
                When user "<username>" logs in with password "<password>"
                Then error page should be open

                        Examples:
                                | username | password |
                                | invalid  | mode     |
                                | demo     | invalid  |
                                | invalid  | invalid  |
                                | demo     | mode     |


        @TEST_CALC-7902 @cypress/integration/login/login.feature
        Scenario: Invalid Login
                Given browser is opened to login page
                When user "dummy" logs in with password "password"
                Then error page should be open


        @TEST_CALC-7901 @cypress/integration/login/login.feature
        Scenario: Valid Login
                Given browser is opened to login page
                When user "demo" logs in with password "mode"
                Then welcome page should be open(base)
```

To run the tests and produce Cucumber JSON reports(s), we can either use `npm` or `cypress` command directly.

```
npm run test

# or instead...

node_modules/cypress/bin/cypress run --spec 'features/**/*.feature' --config integrationFolder=.
```

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

The cypress-cucumber-preprocessor package, as of v4.0.0, does not produce reports containing the screenshots embedded.

However, the following script (credits to the user that provided it on GitHub) can be used to update the previous JSON reports so that they contain the screenshots of the failed tests.

attach_screenshots.js

```
const fs = require('fs-extra')
const path = require('path')
const chalk = require('chalk')

const cucumberJsonDir = './cypress/cucumber-json'
const cucumberReportFileMap = {}
const cucumberReportMap = {}
const jsonIndentLevel = 2
```

```
const ReportDir = './cypress/reports/cucumber-report'
const screenshotsDir = './cypress/screenshots'

getCucumberReportMaps()
addScreenshots()

//Mapping cucumber json files from the cucumber-json directory to the features
function getCucumberReportMaps() {
    const files = fs.readdirSync(cucumberJsonDir).filter(file => {
        return file.indexOf('.json') > -1
    })
    files.forEach(file => {
        const json = JSON.parse(
            fs.readFileSync(path.join(cucumberJsonDir, file))
        )
        if (!json[0]) { return }
        const [feature] = json[0].uri.split('/').reverse()
        cucumberReportFileMap[feature] = file
        cucumberReportMap[feature] = json
    })
}

//Adding screenshots to the respective failed test steps in the feature files
function addScreenshots() {

    const prependPathSegment = pathSegment => location => path.join(pathSegment, location)

    const readdirPreserveRelativePath = location => fs.readdirSync(location).map(prependPathSegment(location))

    const readdirRecursive = location => readdirPreserveRelativePath(location)
        .reduce((result, currentValue) => fs.statSync(currentValue).isDirectory()
            ? result.concat(readdirRecursive(currentValue))
            : result.concat(currentValue), [])
    const screenshots = readdirRecursive(path.resolve(screenshotsDir)).filter(file => {
        return file.indexOf('.png') > -1
    })

    const featuresList = Array.from(new Set(screenshots.map(x => x.match(/[\w-_.]+\.feature/g)[0])))
    featuresList.forEach(feature => {
        screenshots.forEach(screenshot => {

            const regex = /(?<=\ --\ ).*?((?=\ \(example\ \#\d+\))|(?=\ \(failed\)))/g
            const [scenarioName] = screenshot.match(regex)
            console.info(chalk.blue('\n Adding screenshot to cucumber-json report for'))
            console.info(chalk.blue(scenarioName))

            console.log(featuresList)
            console.log(feature)
            console.log(cucumberReportMap)
            const myScenarios = cucumberReportMap[feature][0].elements.filter(
                e => scenarioName.includes(e.name)
            )
            if (!myScenarios) { return }
            let foundFailedStep = false
            myScenarios.forEach(myScenario => {
                if (foundFailedStep) {
                    return
                }
                let myStep
                if (screenshot.includes('(failed)')) {
                    myStep = myScenario.steps.find(
                        step => step.result.status === 'failed'
                    )
                } else {
                    myStep = myScenario.steps.find(
                        step => step.name.includes('screenshot')
                    )
                }
                if (!myStep) {
                    return
                }
```

```
                const data = fs.readFileSync(
                    path.resolve(screenshot)
                )
                if (data) {
                    const base64Image = Buffer.from(data, 'binary').toString('base64')
                    if (!myStep.embeddings) {
                        myStep.embeddings = []
                        myStep.embeddings.push({ data: base64Image, mime_type: 'image/png' })
                        foundFailedStep = true
                    }
                }
            })
            //Write JSON with screenshot back to report file.
            fs.writeFileSync(
                path.join(cucumberJsonDir, cucumberReportFileMap[feature]),
                JSON.stringify(cucumberReportMap[feature], null, jsonIndentLevel)
            )
        })
    })
}
```

The cucumber-json-merge utility may be handy to merge the results of each feature, so they can be then submitted to Xray as one single file.

Next, is an example of a shell script with all these steps.

example of a Bash script to run the tests and produce a unified Cucumber JSON report

```
#!/bin/bash

rm -f cypress/cucumber-json/*
npm run test
npm run attach_screenshots
cucumber-json-merge  -d cypress/cucumber-json/
```

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 a Bash script to import results using the standard Cucumber endpoint

```
#!/bin/bash

curl -H "Content-Type: application/json" -X POST -u admin:admin --data @"report.json" http://jiraserver.example.
com/rest/raven/1.0/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):

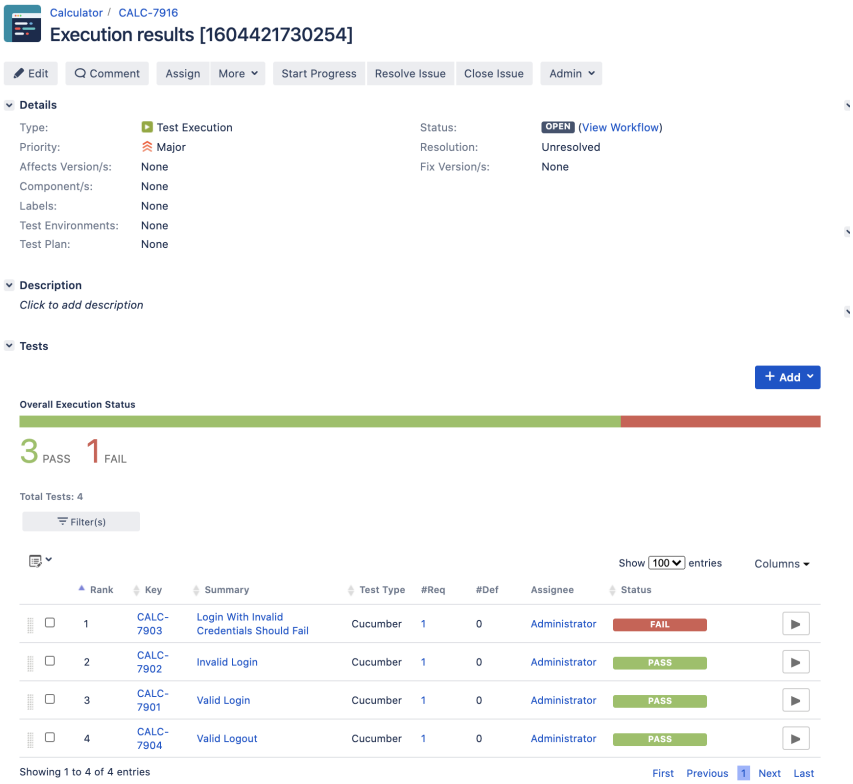
1. the "standard cucumber" endpoint
2. the "multipart cucumber" endpoint

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).

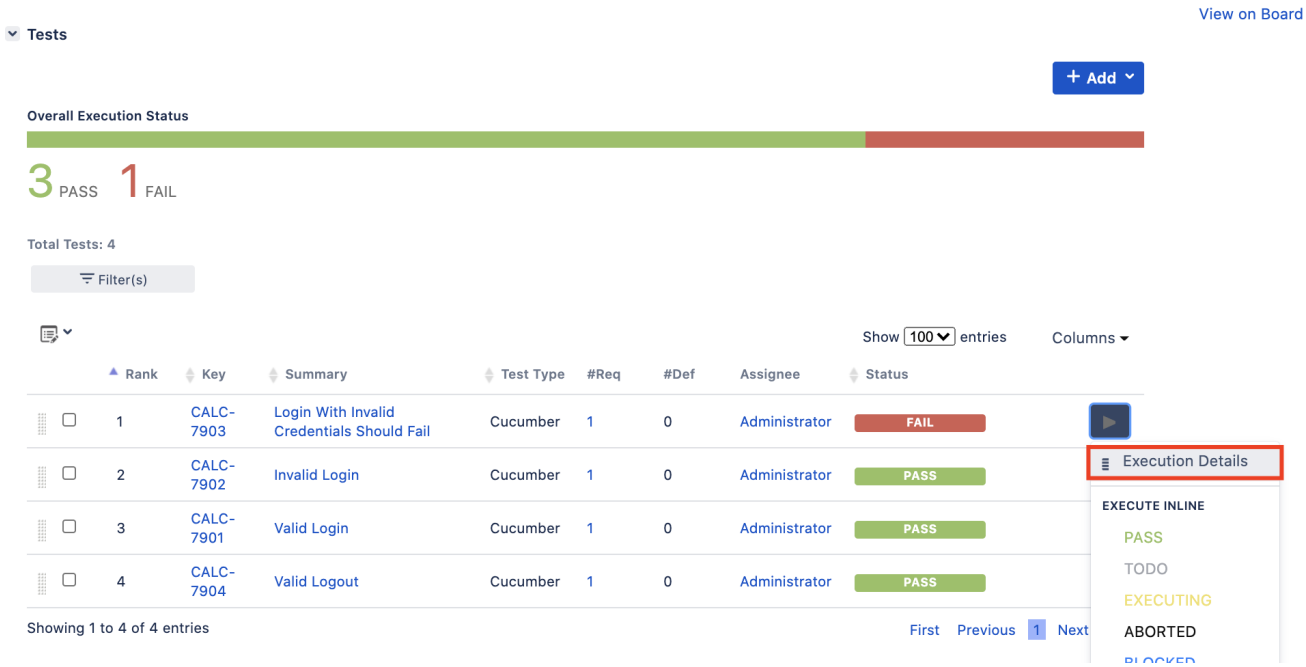
In sum, if you want to customize the Fix Version, Test Plan and/or Test Environment of the Test Execution issue that will be created, you'll have to use the "multipart cucumber" endpoint.

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

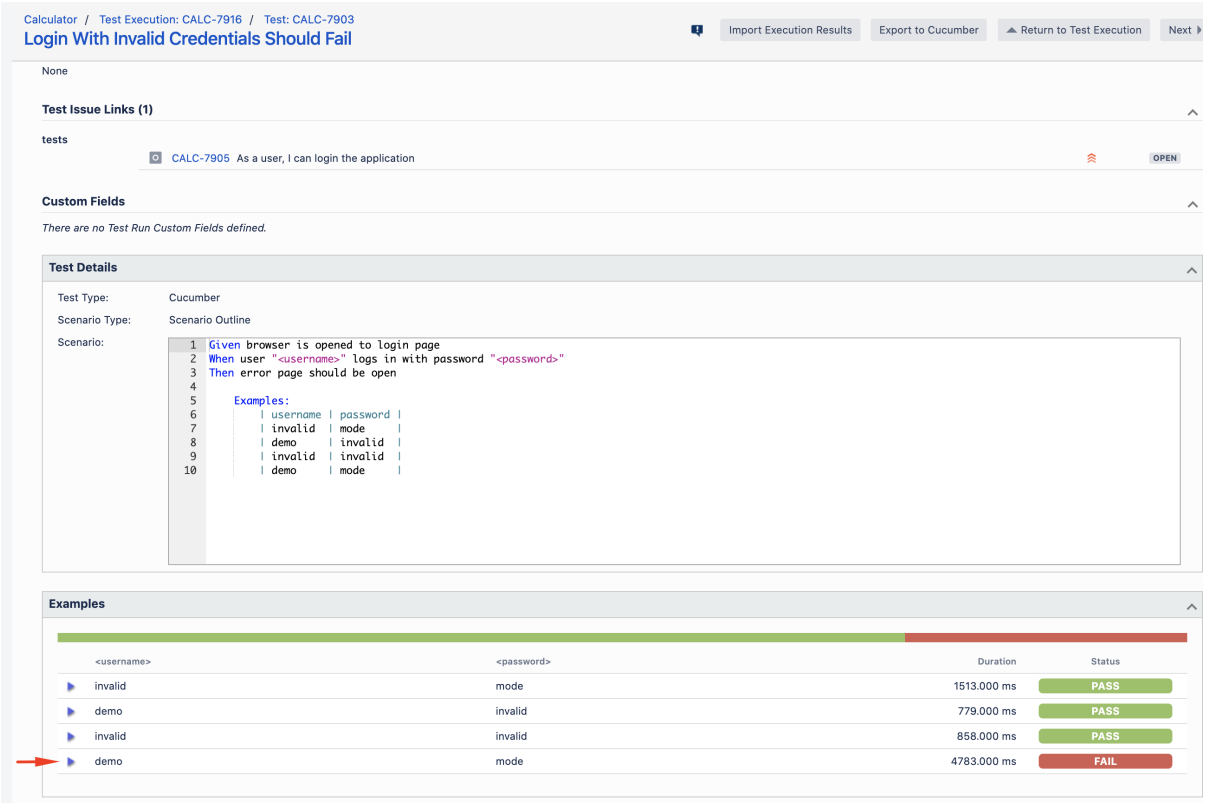


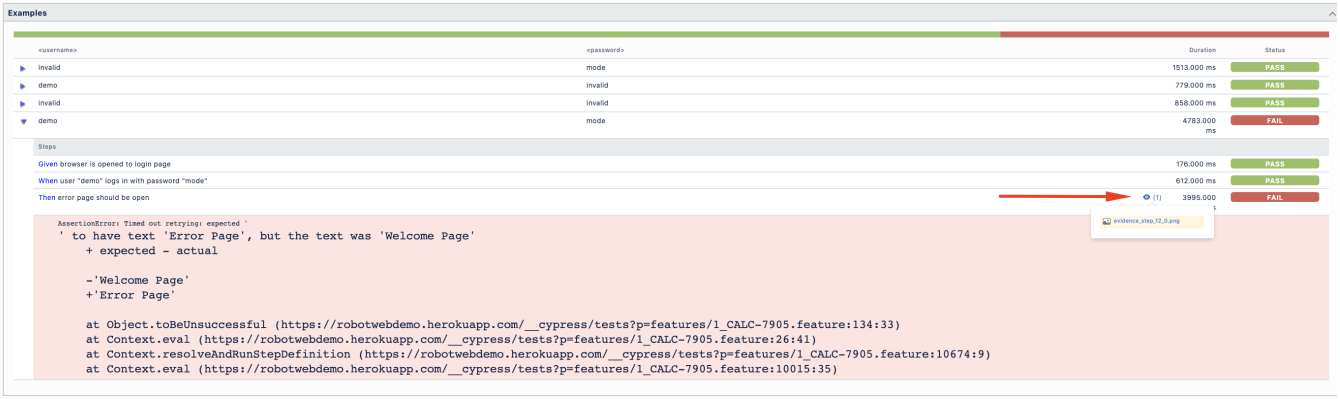
One of the tests fails (on purpose).

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



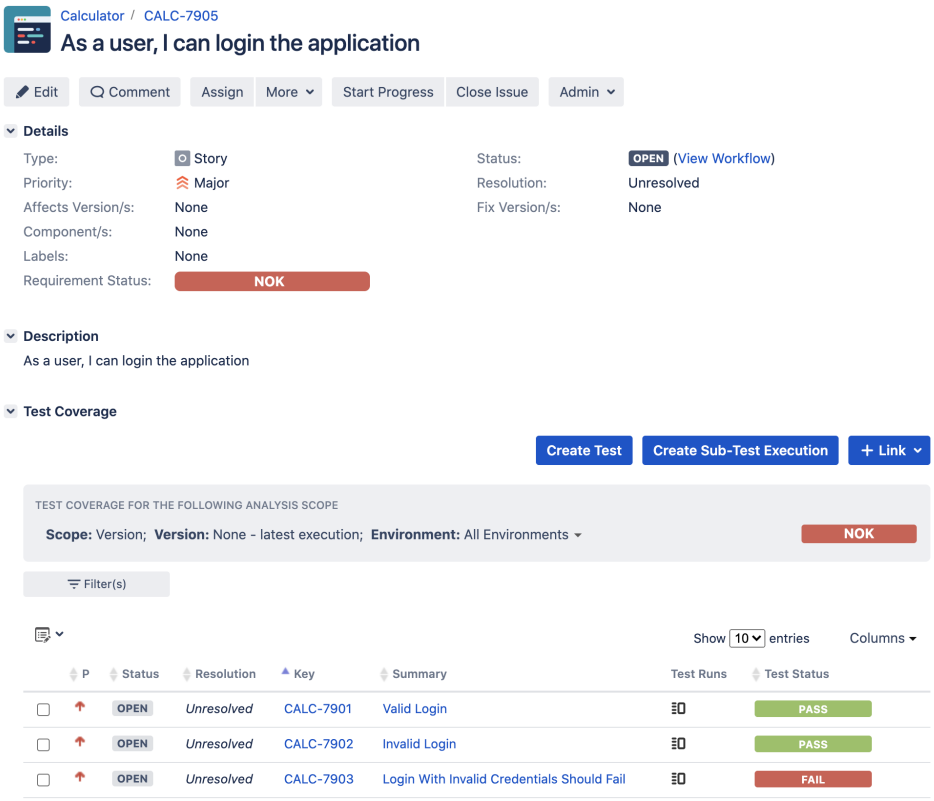
A given example can be expanded to see all Gherkin statements and, if available, it is possible to see also the attached screenshot(s).





Note: in this case, the bug was on the Scenario Outline example which was using a valid username/password combination.

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.



References

- Cypress
- Cypress documentation
- cypress-cucumber-example
- issue related to adding screenshots to the cucumber JSON report(s)