E2E web testing locally, remote or in the cloud (BrowserStack, SauceLabs) using TestCafé in JavaScript

Overview

In this tutorial, we will create some E2E tests in JavaScript using TestCafé.

TestCafé is a Node.js utility for E2E testing that does not use Selenium WebDriver; it uses a proxy mechanism instead. Some of its features include the ability to handle automatically the wait times of page loads/XHR requests for stabler tests.

Besides being easy to setup, it runs in multiple OSes and supports almost, if not all, browsers out there. It supports desktop, mobile, remote, cloud and parallel testing.

You may also use an IDE (TestCafé Studio) to easily record, implement and playback tests.

TestCafé is not just an automation library; it's a complete testing tool that provides assertions and a runner, to implement and run the tests.

Requirements

- TestCafé
- testcafe-reporter-xunit module
- testcafe-browser-provider-saucelabs module (if using Sauce Labs)
- testcafe-browser-provider-browserstack module (if using BrowserStack)

Description

The following examples are taken from TestCafé's documentation; the first one implements a test in a very simple way although hard to maintain in the long run.

In the second example, two tests are built using the page object pattern.

In TestCafé terminology, a fixture is essentialy a group of tests. A Javascript/TypeScript file may contain one or more fixtures and associated to each one there can be multiple tests.

Simple message validation after submission of input

In this test, an input box is filled out and submitted. Afterwards, the submission message is evaluated.

Hardcoded selectors, directly in the test code, are used to implement the test.

test1.js

```
import { Selector } from 'testcafe'; // first import testcafe selectors

fixture `Getting Started`// declare the fixture
    .page `https://devexpress.github.io/testcafe/example`;  // specify the start page


//then create a test and place your code there
test('My first test', async t => {
    await t
        .typeText('#developer-name', 'John Smith')
        .click('#submit-button')

        // Use the assertion to check if the actual header text is equal to the expected one
        .expect(Selector('#article-header').innerText).eql('Thank you, John Smith!');
});
```

Implementing tests using page object pattern

In this scenario, taken from TestCafé's documentation, we start by defining a page object.

page-model.js

```
import { Selector } from 'testcafe';

const label = Selector('label');

class Feature {
    constructor (text) {
        this.label    = label.withText(text);
        this.checkbox = this.label.find('input[type=checkbox]');
    }
}

class OperatingSystem {
    constructor (text) {
        this.label       = label.withText(text);
        this.radioButton = this.label.find('input[type=radio]');
    }
}

export default class Page {
    constructor () {
        this.nameInput             = Selector('#developer-name');
        this.triedTestCafeCheckbox = Selector('#tried-test-cafe');
        this.populateButton        = Selector('#populate');
        this.submitButton          = Selector('#submit-button');
        this.results               = Selector('.result-content');
        this.commentsTextArea      = Selector('#comments');

        this.featureList = [
            new Feature('Support for testing on remote devices'),
            new Feature('Re-using existing JavaScript code for testing'),
            new Feature('Running tests in background and/or in parallel in multiple browsers'),
            new Feature('Easy embedding into a Continuous integration system'),
            new Feature('Advanced traffic and markup analysis')
        ];

        this.osList = [
            new OperatingSystem('Windows'),
            new OperatingSystem('MacOS'),
            new OperatingSystem('Linux')
        ];

        this.slider = {
            handle: Selector('.ui-slider-handle'),
            tick:   Selector('.slider-value')
        };

        this.interfaceSelect       = Selector('#preferred-interface');
        this.interfaceSelectOption = this.interfaceSelect.find('option');
    }
}
```

Two tests can easily be implemented using the previous page object, one of them iterating over a bunch of checkboxes.

test2.js

```
import Page from './page-model';

fixture `My fixture`
    .page `https://devexpress.github.io/testcafe/example/`;

const page = new Page();

test('Text typing basics', async t => {
    await t
        .typeText(page.nameInput, 'Peter')
        .typeText(page.nameInput, 'Paker', { replace: true })
        .typeText(page.nameInput, 'r', { caretPos: 2 })
        .expect(page.nameInput.value).eql('Parker');
});

test('Click check boxes and then verify their state', async t => {
    for (const feature of page.featureList) {
        await t
            .click(feature.label)
            .expect(feature.checkbox.checked).ok();
    }
});
```

Running the tests

In this example, we'll run the tests locally using Chrome browser.

```
testcafe chrome test1.js test2.js --reporter xunit > results.xml
```

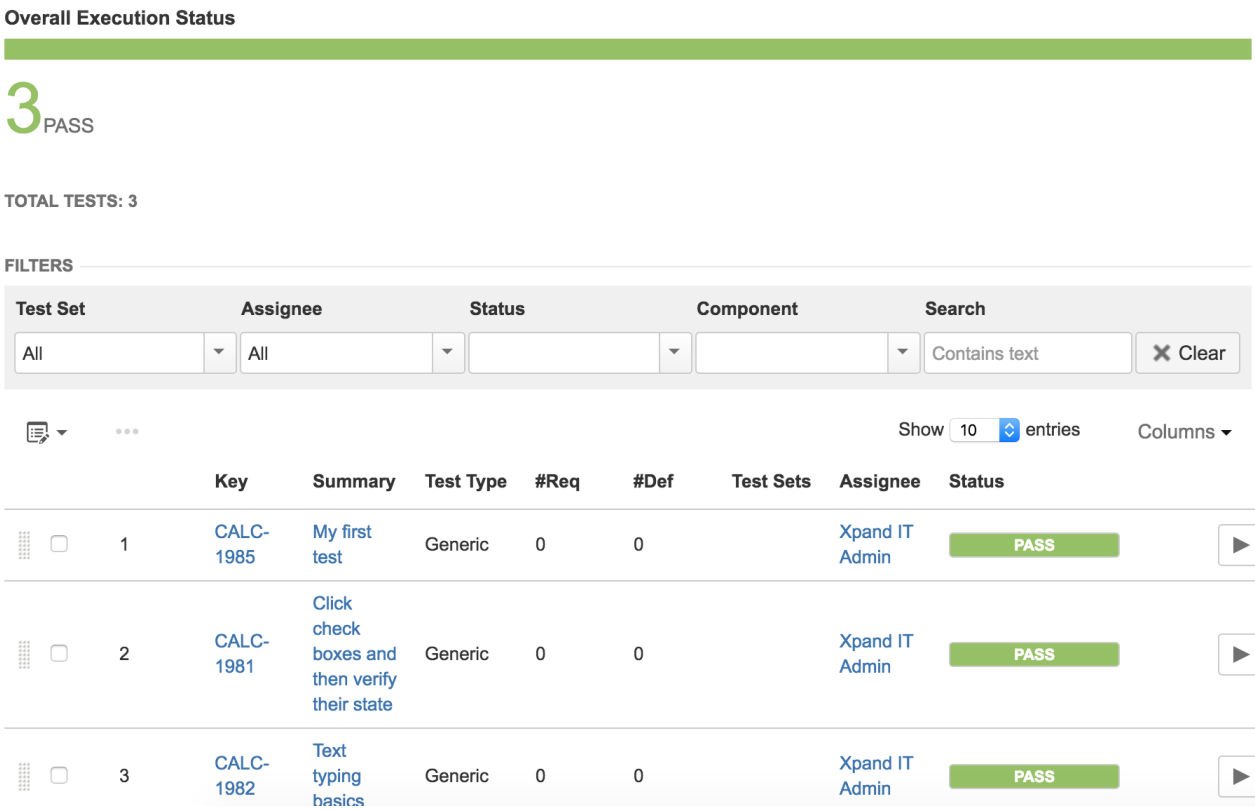
In case you want ro run your tests concurrently, you can pass an argument to identify the number of sessions and the browsers to use.

```
testcafe -c 10 chrome,firefox test1.js test2.js --reporter xunit > results.xml
```

It is also possible to perform headless testing (e.g. use "chrome:headless" as the browser name, for example).

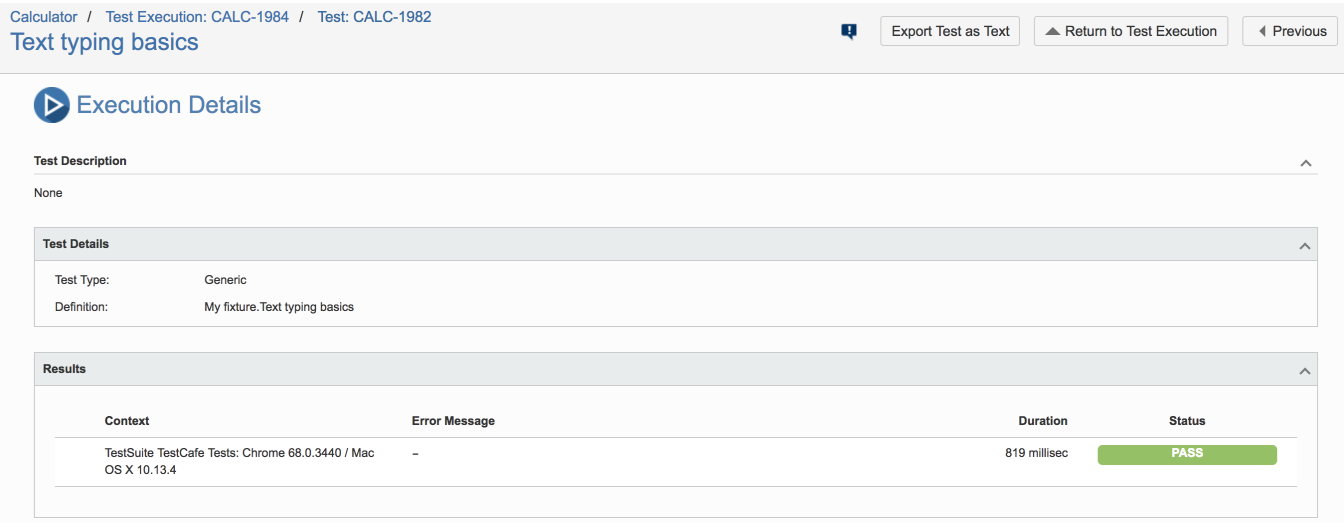
```
testcafe "chrome:headless"  test1.js test2.js --reporter xunit > results.xml
```

After running the tests and generating the JUnit XML reports (e.g., results.xml), they can be imported to Xray (either by the REST API or through the **Import Execution Results** action within the Test Execution).



JUnit's Test Case is mapped to a Generic Test in Jira, and the **Generic Test Definition** field contains the name of of the *fixture* concatenated with the name of the test .

The Execution Details of the Generic Test contains information about the Test Suite, which in this case contains the identification of the target environment (i.e. browser + OS).



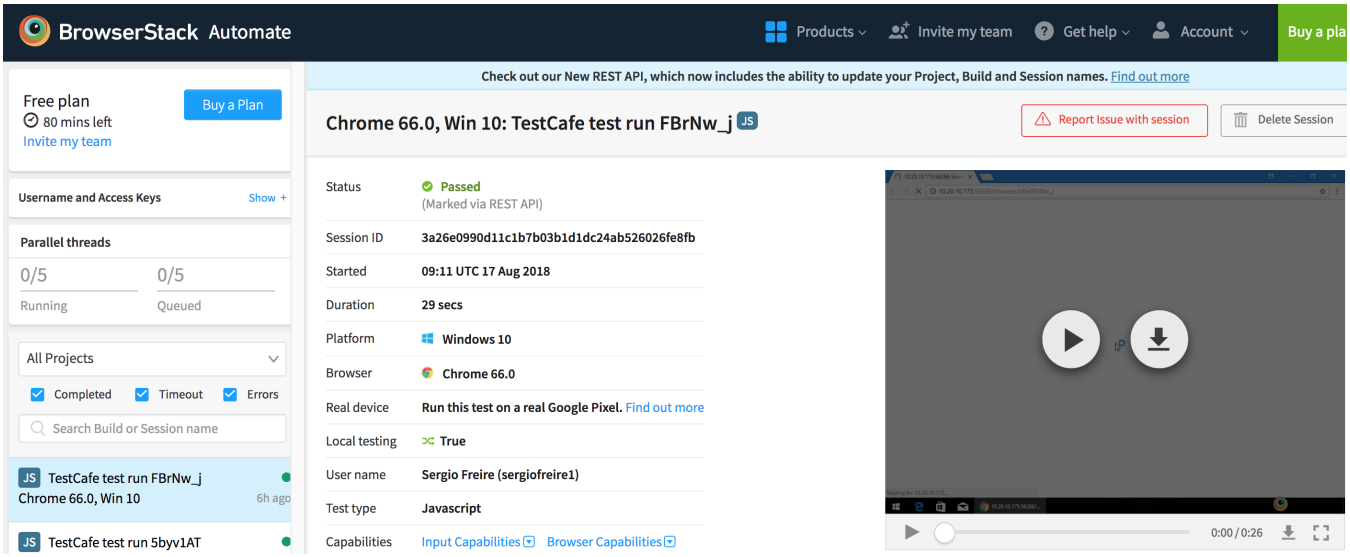
Running tests using cloud providers

Tests can easily be run using different cloud providers and it's possible to generate a JUnit XML report with the results on a per test basis.

Both in BrowserStack and in Sauce Labs backoffice only a session/test will appear though.

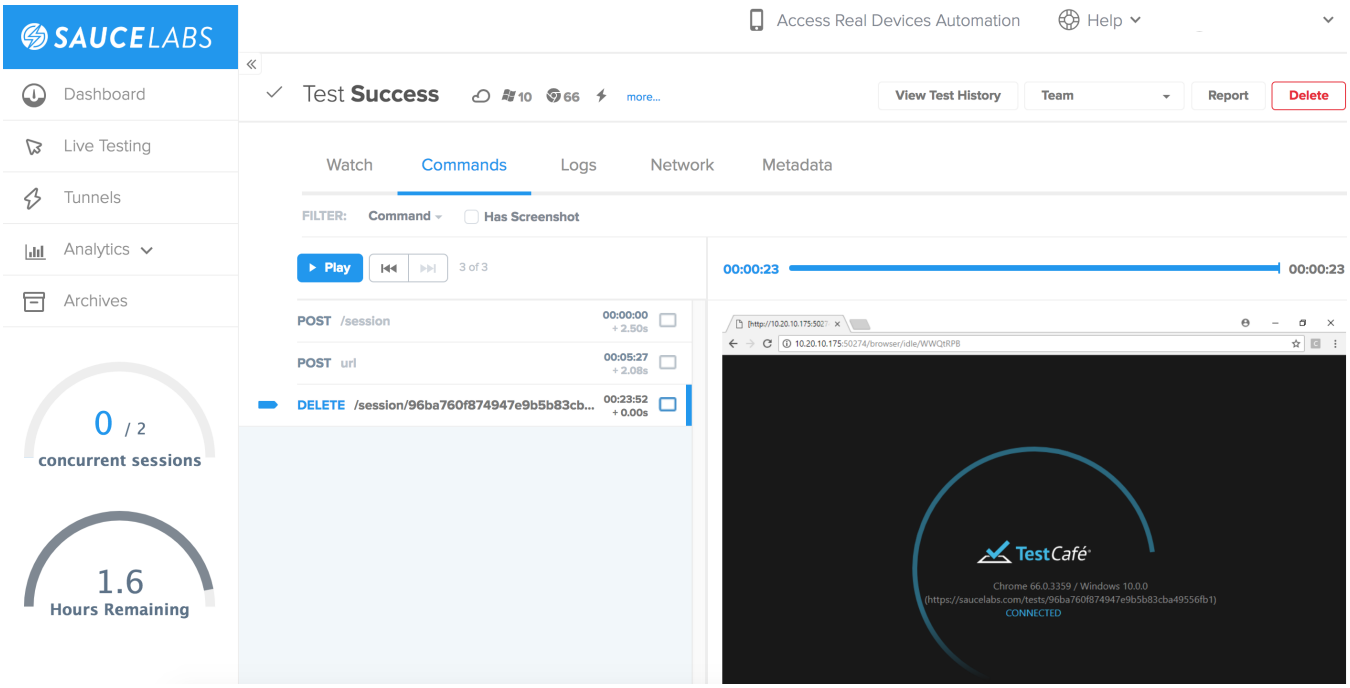
To run them in the cloud using BrowserStack, the command line would be something like this (BROWSERSTACK_USERNAME and BROWSERSTACK_ACCESS_KEY environment variables would need to be defined beforehand):

```
testcafe "browserstack:Chrome@66.0:Windows 10" test1.js test2.js --reporter xunit > results.xml
```



To run them in the cloud using Sauce Labs, the command line would be something like this (SAUCE_USERNAME and SAUCE_ACCESS_KEY environment variables would need to be defined beforehand):

```
testcafe "saucelabs:Chrome@66.0:Windows 10" test1.js test2.js --reporter xunit > results.xml
```



References

- https://devexpress.github.io/testcafe/
- https://devexpress.github.io/testcafe/documentation/getting-started/
- https://devexpress.github.io/testcafe/documentation/recipes/using-page-model.html
- https://github.com/DevExpress/testcafe-reporter-xunit
- https://dzone.com/articles/testcafe-e2e-testing-tool
- https://testcafe-studio.devexpress.com/documentation/getting-started/
- https://medium.com/yld-engineering-blog/evaluating-cypress-and-testcafe-for-end-to-end-testing-fcd0303d2103