TestCafe

Automation Framework

for Browsers and Mobile testing

Khai Pham | Staff Software Engineer in Test | khaipham@gmail.com

SELENIUM?

- Selenium is such a popular tool that it's almost a synonym for Test Automation. Or it was until recently. It's a tool with great community support, great language support, and a number of frameworks built on it with more features and flexibility (WebdriverIO, Protractor, CasperJS…)

- However, Selenium lacks some features especially with Single page applications and React components and with a lot of focus asynchronous operations and client-side performance. Single page application and asynchronous calls made the life of writing tests in Selenium tedious. Managing waits, waiting for page load, setting up expected conditions and writing tests; not so easy.

- Selenium also requires not so simple configurations to get what we need. Just to name a few:
  - Getting/Installing drivers for browser support.
  - Creating capabilities and profiles for browsers.
  - Matching of driver and browser versions to avoid compatibility issues.

Existing Javascript Frameworks

- **Jasmine, Mocha, WebdriverIO, Protractor** are all great Javascript test frameworks that can integrate with Selenium Webdriver, Karma, or Puppeteer to run tests on browsers, but still require not so simple configuration and maintain compatibility between drivers and browsers

New Kid on the block!

- TestCafe - open source https://devexpress.github.io/testcafe/
- An all-in-one solution that is easy to get up and running
- TestCafe is a mature Javascript framework built from scratch which is built on Node.js.
- TestCafe can run on Mac, Windows, Linux and detect all browsers (Chrome, FF, Safari, IE) installed on machine.
- The fundamental difference between Selenium and TestCafe is that Selenium runs the code in the browser process itself, whereas TestCafe uses a Proxy in between which performs URL rewriting, and injects the test scripts into the browser. Since these proxies manage all storage/cookies for the tests, you get a clean and isolated test environment which is difficult to achieve in Selenium.

TestCafe features https://devexpress.github.io/testcafe/

- One line install and boiler plate configurations (npm install -g TestCafe)
- Can detect all the browsers installed on your machine including legacy IE and run the tests on them without having to install/configure something.
- It can run tests in headless mode in both in Chrome and Firefox
- Can run tests in parallel and on different browsers without complex configurations (eg: TestCafe -c 4 chrome,firefox,safari test/test.js)
- Can run tests on remote desktops, mobiles and cloud-based browsers.
- It can run tests on Chrome device emulation (eg: TestCafe "chrome:emulation:device=iphone 6" tests/ios.js)
- Automatic Waiting Mechanisms and Smart Assertion Query Mechanism which helps to create more stable and faster running tests.

TestCafe features continue

- Functional style selectors (Selector(‘label’).withText(‘foo’), Selector(‘li’).filter(‘.someClass’))
- The ability to use framework specific selectors eg: React, Angular
- Ability to intercept HTTP responses. Ability to Mock HTTP Responses
- Observes JS errors (fails the test if there’s a script error if you would like)
- Easy integrations with CI tools like Jenkins, TeamCity.
- Easy debugging in VS Code, WebStorm, Chrome DevTools.
- Save screenshots on failure.
- Record test runs (requires ffmpeg).
- Excellent Documentation and Example codes.
- Companion with TestCafé Studio (in BETA) - A Cross-Platform IDE that can record test steps, convert to JS test scripts, run / edit / debug tests.

TestCafe Built-In Waiting Mechanisms

- TestCafe has built-in automatic waiting mechanisms, so that it does not need dedicated API to wait for page elements to appear or redirects to happen.
- Built-in Waiting mechanisms work with test actions, assertions, selectors and navigation.

- TestCafe automatically waits for the target element to become visible when an action is executed. TestCafe tries to evaluate the specified selector multiple times within the timeout. If the element does not appear, the test will fail.
- When evaluating a selector, TestCafe automatically waits for the element to appear in the DOM. TestCafe keeps trying to evaluate the selector until the element appears in the DOM or the timeout passes.
- TestCafe assertions feature the Smart Assertion Query Mechanism. This mechanism is activated when you pass a selector property or a client function as an actual value. In this instance, TestCafe keeps recalculating the actual value until it matches the expected value or the assertion timeout passes.

Example

login_spec.js

login_page.js

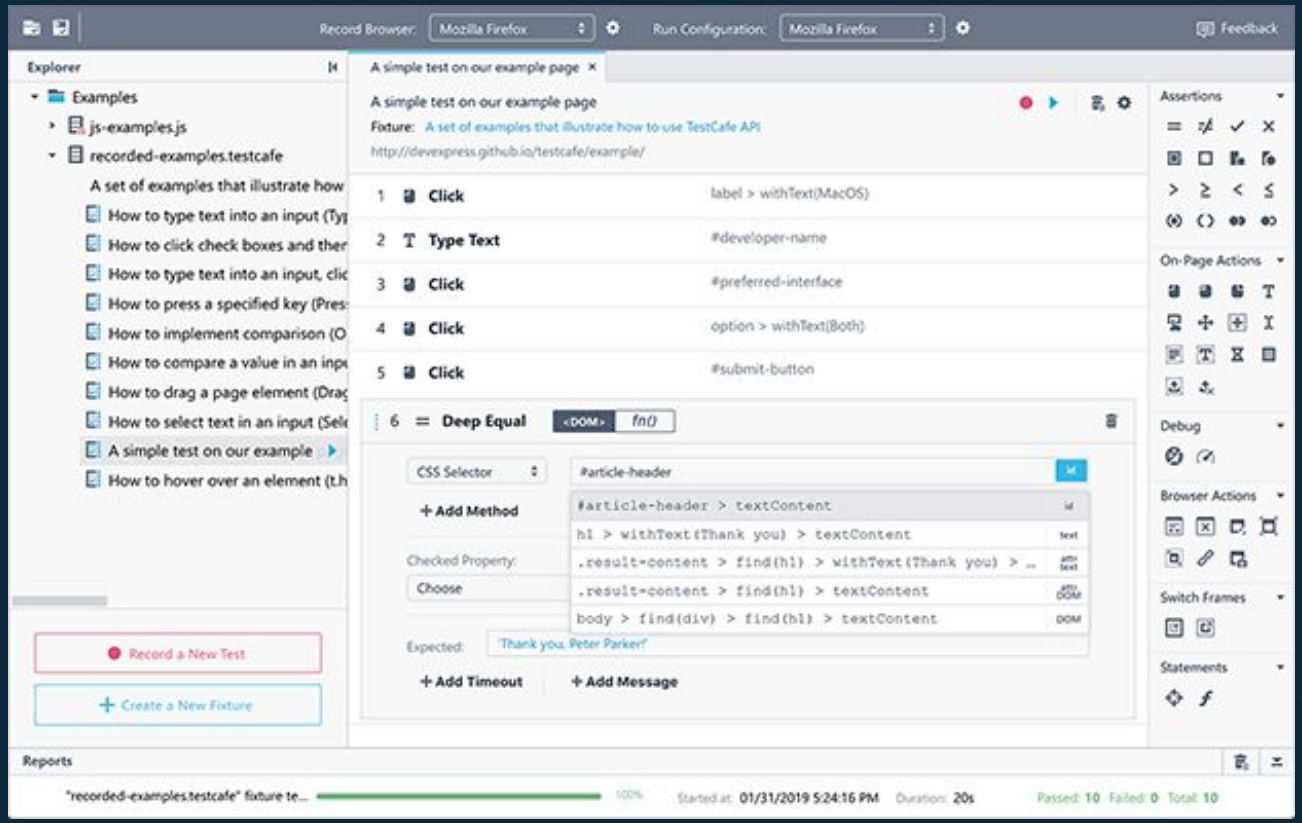
```
fixture `Signin into ${env} studio module`
 .page `${baseUrl}`;
test('login', async t => {
 await loginPage.signin(config.name,
config.password[env]);
 console.info(' current URL:', await getURL());
 await t.expect(getURL()).contains(baseUrl);
});


export default class LoginPage {
 constructor () {
   this.email = Selector('#email');
   this.password = Selector('#password');
   this.signin = Selector('#signinButton');
  }
 async signin (name, password) {
   console.info(' signing in with:', name);
   await t
     .typeText(this.email, name)
     .typeText(this.password, password)
     .click(this.signin);
  }
```

Demos

- Run TestCafe with all local browsers
- testcafe all webdriver/suites/portal/accessibility
- Run test on Jenkins against a remote Browser
  testcafe remote webdriver/suites/portal/accessibility
- Run test on chrome mobile emulator
  testcafe "chrome:emulation:device=iphone 6" webdriver/suites/portal/accessibility
- Run test on remote Mobile with QR code
  testcafe remote webdriver/suites/portal/accessibility --qr-code
- Run/Debug TestCafe from IDE with breakpoints

TestCafe Studio: A Cross-Platform IDE for End-to-End Web Testing

Visually Record, Edit, and Run tests on browsers



Run Tests on Remote Computer / Mobile

Run Tests on a Remote Computer

Use remote as a browser alias to specify that tests should run on a remote machine.TestCafe prepares a URL for the remote browser to connect to server.

```
$ testcafe remote tests/test.js -L
Using locally installed version of TestCafe.
Connecting 1 remote browser(s)...
Navigate to the following URL from each remote browser.
Connect URL: http://10.1.10.10:55568/browser/connect
```

Run Tests on a Mobile Device

Add the --qr-code flag to generate a QR-code for the mobile device.

```
$ testcafe remote tests/test.js -L --qr-code
```

TestCafe will output URL and the QR-code to the console for the remote device to connect. Tests start as soon as browser is connected to the server.

Run Tests on BrowserStack

Install browserstack plugin

$ npm install testcafe-browser-provider-browserstack --save-dev

Establishing a Local Testing connection

$ npm install browserstack-local --save-dev

Before using this plugin, set environment variables with export

```
export BROWSERSTACK_USERNAME=UserName
export BROWSERSTACK_ACCESS_KEY=AccessKey
export BROWSERSTACK_DISPLAY_RESOLUTION="1280x800"
export BROWSERSTACK_CHROME_ARGS="--autoplay-policy=no-user-gesture-required"
```

Determine the available browser aliases:

$ testcafe -b browserstack

Run tests from the command line, use the alias when specifying browsers:

$ testcafe "browserstack:ie@11.0:Windows 10" "path/to/test/file.js"

TestCafe Live Mode

Use the -L (--live) flag to enable live mode from the command line interface.

```
$ testcafe chrome tests/test.js -L
```

When you run tests with live mode enabled, TestCafe opens the browsers, runs tests there, shows the reports and waits for your further actions.

Then TestCafe starts watching for changes in the test files and all files referenced in them (like page objects or helper modules). Once you make changes in any of those files and save them, TestCafe immediately reruns the tests.

When the tests are done, browsers stay on the last opened page so you can work with it and explore it with the browser's developer tools.

TestCafe quarantineMode

Use the -q (--quarantine-mode) flag to enable quarantine mode from the command line interface.

```
$ testcafe chrome tests/test.js -q
```

The quarantine mode is designed to isolate non-deterministic tests (that is, tests that pass and fail without any apparent reason) from the other tests. When the quarantine mode is enabled, tests run according to the following logic:

1. A test runs at the first time. If it passes, TestCafe proceeds to the next test.
2. If the test fails, it runs again until it passes or fails three times.

Server Debugging

Starting with version v6.3.0, Node.js allows for debugging applications in Chrome Developer Tools. If you have Chrome and an appropriate version of Node.js installed on your machine, you can easily debug test code.
To do this, add special flags --inspect and --debug-brk to test run command.

```
testcafe --inspect-brk chrome ./tests
```

Also, put the debugger; keyword in test code where you want to stop.

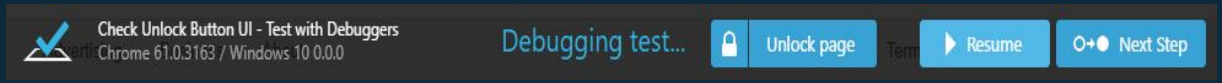
Client-Side Debugging

To debug client test code, use a special action t.debug().
When test execution reaches this action, it pauses so that you can open browser's developer tools and check the expected web page state, DOM elements location, their CSS styles. In the footer, you'll find buttons that allow you to continue test execution or step to the next test action.

You can also use the --debug-mode command line option to pause the test before the first action or assertion

The debugger does not stop at creating and resolving the selectors and client functions

TestCafe logs points in code where the debugger stopped.



More Client-Side debugging

Use browser console

- Selector with document.queryselector(“selector”)
- Actions with document.queryselector(“selector”).click()
- Select one a group of elements
  document.querySelectorAll("li")[0].click();

Use Selector with partial matching

- Selector('[class^="Input-1-"]')

Helpful options for debugging

TestCafe includes a few features helpful to find the cause of issues:

> Screenshots - you can explicitly specify places in your test where screenshots should be taken. Use the t.takeScreenshot([path]) action for this. You can also turn on the --screenshots-on-fails CLI option

```
testcafe chrome ./tests --screenshots ./screenshots --screenshots-on-fails
```

This option enables TestCafe to take a screenshot when a test fails

> Test speed - use this option to change testing speed. By default, tests are executed at full speed with minimum delays between actions and assertions. This makes it hard to identify problems visually when running the test. To slow down the test, use the --speed CLI flag. Its value changes from 1 to 0.01.

```
testcafe chrome ./tests --speed 0.1
```

TestCafe Configuration file

TestCafe uses the .testcaferc.json configuration file to store its settings.

Settings you specify when you run TestCafe from the command line and programming interfaces override settings from .testcaferc.json.
TestCafe prints information about every overridden property in the console.
Keep .testcaferc.json in the directory from which you run TestCafe. This is usually the project's root directory.

```
{
    "screenshotPath" : "webdriver/screenshots/" ,
    "screenshotPathPattern" :
"${DATE}_${TIME}/test-${TEST_INDEX}/${USERAGENT}/${FILE_INDEX}.png"  ,
    "takeScreenshotsOnFails" : true,
    "stopOnFirstFail" : false,
    "skipJsErrors" : true,
    "skipUncaughtErrors" : true,
    "selectorTimeout" : 30000,
    "quarantineMode" : true
}
```

Accessibility Node.js module

Axe-testcafe

`axeCheck(t)`

```
// Generate and run separate tests in a loop
for (let i = 0; i < templates.length; i++) {
 const siteUrl = `https://${templates[i]}.sample.com`;
 // Navigate to the sample live site
 test.page(siteUrl) (`Check accessibility against ${siteUrl}`, async t => {
   await axeCheck(t);
```

```
✓ Check accessibility against https://catalogue.sample.com
X Check accessibility against https://discovery.sample.com
  1) AxeError:
     Buttons must have discernible text
        nodes:
           "li:nth-child(2) > button.site-search__button"
     Headings must not be empty
        nodes:
           ".mod-categories > h2.mod-categories__head"
     Form elements must have labels
        nodes:
           "#site-search-input"
     Page must contain one main landmark.
```

ReactSelector Node.js module

The React selectors module provides the ReactSelector class that allows you to select DOM elements by the component name. You can get a root element or search through the nested components or elements. In addition, you can obtain the component props and state.

```
$ npm install testcafe-react-selectors
```

```
import ReactSelector from 'testcafe-react-selectors';
```

```
const TodoList = ReactSelector('TodoApp TodoList');
```

User Profiles

By default, TestCafe launches browsers (Google Chrome and Mozilla Firefox so far) with a clean profile, i.e. without extensions, bookmarks and other profile settings. This was done to minimize the influence of profile parameters on test runs.

However, if you need to start a browser with the current user profile, you can do this by specifying the :userProfile flag after the browser alias.

```
$ testcafe firefox:userProfile tests/test.js
```

:UserProfile

@khaidpham/testcafe-common

Use common routines for VideoCloud Studio's TestCafe automation.

Instead of having to implement the same routines for each project.

Install:

```
$ npm install @khaidpham/testcafe-common --save-dev
```

Usage example:

```
import * as common from '@khaidpham/testcafe-common';
test('login', async t => { await common.signin(url, email, password); });
```

Ready to Get Started with TestCafe ?

- https://devexpress.github.io/testcafe/documentation/getting-started/
- Tutorial to create a test script test1.js for the http://devexpress.github.io/testcafe/example sample page.

Specify this page as a start page for the fixture using the page function.
Then, create the test function where you can enter test code.

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

fixture `Getting Started`

 .page `http://devexpress.github.io/testcafe/example`;


test('My first test', async t => {

 // Test code

 console.log('Navigated to TestCafe sample page')

});
```

Running the Test
You can run the test from a command you specify the target browser and file path.
$ testcafe chrome test1.js

Run Concurrent Threads with TestCafe ?

- Use CLI parameter -c, --concurrency <number> to run tests concurrently
  $ testcafe -c 4 “chrome --autoplay-policy=no-user-gesture-required” --env=qa webdriver/suites/app1/end2end

- Use a list of browsers to run tests concurrently on different browsers
  $ testcafe all --env=qa webdriver/suites/smoke
  $ testcafe chrome,firefox,safari --env=qa webdriver/suites/smoke

Thanks!

Any questions/comments?

◇ Khai Pham | khaipham@gmail.com