Testing front end with Mocha and Playwright

I've told myself for a while that I would learn how to write and use tests for front end code.

Most of the code and tutorials I've seen are written for backend code. But using tools like Playwright together with Mocha and the Chai assertion library we can build verry robust testing for front end code that works in multiple browsers.

Background: Playwright

Playwright is Microsoft's browser testing and automation library. It allows you to control a browser: Chrome, Firefox or Safari (WebKit) and perform operations as if you were working on the browser directly.

Because we support multiple browsers we can configure wich one we will run for a specific task.

Background: Mocha and Chai

Mocha is one of the oldest testing libraries available (the first alpha version of the library was released in 2011). It allows you to create tests for the frontend and the backend using the same vocabulary.

Chai is the assertion library I use with Mocha. It's Chai's `should` syntax that allows me to do this:

```
articles.length.should.be.greaterThan(1);
```

Yes, it is more wordy but it makes it easier to understand and reason through what we want to say.

Putting them together

I've broken the code in sections to make it easier to write about the different

sections.

The first section requires the necessary modules.

Chai's `should` assertion style requires that we actually use a function as part of the require statement.

We destructure the elements we require from Playwright, we want access to Chromium, Firefox and WebKit.

```
/* eslint-disable no-unused-vars */
const should = require('chai').should();
const {chromium, webkit, firefox} = require('playwright');
```

The next block defines variables and constants.

`browser`, `page`, and `context` are Playwright specific variables that we define outside the tests and functions to make them global.

The `BASE_URL` is the URL that we'll run the tests against.

The `browserName` is either the value of the `BROWSER` environmental variable or the string `chromium`. We can leverage this to run the tests against the different browsers that Playwright supports.

```
// Playwright variables
let browser, page, context;

const BASE_URL = 'https://publishing-project.rivendellweb.net/';
const browserName = process.env.BROWSER || 'chromium';
```

`beforeEach` and `afterEach` will execute before and after each test. We reset the Playwright configuration before every test and we close the browser we tested with after each test is completed.

```
// Runs this before each test
beforeEach(async () => {
  browser = await {chromium, webkit, firefox}[browserName].launch({
```

```
    headless: true,
  });
  context = await browser.newContext();
  page = await context.newPage(BASE_URL);
  await page.goto(BASE_URL);
});

// Runs this after each test
afterEach(async () => {
  await browser.close();
});
```

I've taken a small set of tests to illustrate some things I found particularly useful.

The first test checks the content of the page's heading using Playwright's page.$eval to query the document for one instace.

We then use Chai to test if the content of the header matches the title for the site.

```
describe('[PLAYWRIGHT]: Blog Structure Tests', () => {
  it('[PLAYWRIGHT]: should render TPP homepage', async () => {
    const headerText = await page.$eval(
        '.site-title',
        (header) => header.innerText,
    );
    headerText.should.equal('The Publishing Project');
  })
```

This test will query the document for all articles that have a class with the word post in the attribute value.

The we test if the number of returned matches is five. The blog is coded to have five posts on the home page so anything else will fail.

```
  it('[PLAYWRIGHT]: should render one or more posts', async () => {
    const availableArticles = 'article[class*=post]';
```

```
    const articles = await page.$$(availableArticles);

    articles.length.should.be.greaterThan('0');
  });
```

This test will check if the images on the page have an alt attribute by creating an array of all the images with `alt` attribute.

```
describe('[PLAYWRIGHT]: Basic Accessibility Tests', () => {
  it('[PLAYWRIGHT]: images have alt attributes', async () => {
    const availableImages = 'img[alt]';
    const images = await page.$$(availableImages);

    images.length.should.be.greaterThan(0);
  });
```

The next task is to make sure that the attribute is not empty. We have do some weird contortions to make sure that it works as intended.

`page.$$()` is equivalent to `querySelectorAll()` where it will return all elements in the page that match our condition.

We then test if the condition is bigger than 0, meaning there are images and they don't have an empty `alt` attribute.

```
  it('[PLAYWRIGHT]: has no images with empty alt attribute', async () => {
    const nonEmptyImages = '[alt]:not([alt=""])';
    const result = await page.$$(nonEmptyImages);

    result.length.should.be.greaterThan(0);
  });
});
```

Running the tests

We choose to run the test with any of the Playwright supported browsers will the

following command:

```
BROWSER=firefox mocha --timeout 10000
```

and replace firefox with the browser you want to use (the options are: **Firefox**, **Chromium** and **WebKit**)

The reason why I use Playwright is that it also allows me to emulate a browser to run more detailed tests than you can do with Mocha alone.

For example, I can use Playwright to test the navigation of the site.

This example will load the homepage and then click on the title of the first post on the page.

The locator class allows us to query the document to match specific criteria, the first argument will retrieve the first value (regardless of how many values the locator returns) and the click method will click on the element.

```
  it('[Playwright]: Should render the first post', async() => {
    const pageClick = await page.locator('article :first-child h2 a').firs
  })
```

We could use a similar technique to fill out forms with test data to see if they work or to test that the site's navigation works as expected.

One final detail. The script we've written is setup to run in headless mode, meaning we don't get to see what the browser does. We can change that by changing the value of the headless attribute in the function attached to `beforeEach`. This way we get to see the browser in action.

The new `beforeEach` declaration looks like this:

```
// Runs this before each test
beforeEach(async () => {
  browser = await {chromium, webkit, firefox}[browserName].launch({
    headless: false,
  });
```

```
  context = await browser.newContext();
  page = await context.newPage(BASE_URL);
  await page.goto(BASE_URL);
});
```