Puppeteer: Headless Chrome to the rescue

Puppeteer is a project from Chrome's Devtools team to provide a high level way to automate running Chrome in Headless mode (Chrome running without a graphical user interface. Headless browsers provide automated control of a web page in an environment similar to regular Chrome, but executed via a command-line interface or using network communication).

The idea behind headless browsers like PhantomJS, Headless Chrome or He adless Firefox is to automate tasks like testing and doing screen shots of the page visited.

As we go through some of these examples we'll explore the Puppeteer API in some (but not all) details. For a deep look at the API check the API docs.

Capturing page screenshots

The first, and simplest, way we'll use Puppeteer is to generate screenshots of a web page or app. We'll take advantage of Puppeteer's predefined device descriptions to ease the workload and generate a png screenshot of my persoal blog. The code to do so look like this:

```
const puppeteer = require('puppeteer');
const devices = require('puppeteer/DeviceDescriptors');

(async () => {

  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.emulate(devices['iPad Pro']);
  await page.goto('https://rivendellweb.net',);
  await page.screenshot({ path: 'full.png'});
  await browser.close();

})();
```

From top to bottom, the script:

1.- Loads the required scripts.

2.- Sets up a variable to hold our puppeteer.launch declaration. This declaration has an optional parameter of an object used to configure the Chromium instance

3.- Create sa new page object using browser.newPage()

4.- Configures the browser to run our commands with page.emulate.

You can replate the viewport object with one of the predefined values from `puppeteer/DeviceDescriptors`. The descriptors will pre-populate all the values for the viewport items.

I normally use the raw viewport items and values when I need to create a custom viewport and the Device Descriptors otherwise.

5.- Tells Puppeteer where to go and when to consider the page loaded using page.goto. It returns a Promise which resolves to the main resource response. In case of multiple redirects, the navigation will resolve with the response of the last redirect.

6.- Configures the screenshot we want to take with page.screenshot.

7.- Runs `browser.close()` to close the connection.

Capturing full screen images

There are times when I would like to see all content for the page, even if it goes beyond the default screen size for the device I've chosen to test with (in this case an iPad Pro in portrait mode). We can add a second parameter to `page.screenshot` to indicate this. `fullPage` is a boolean value that, when true, takes a screenshot of the full scrollable page. It defaults to false

```
const puppeteer = require('puppeteer');
const devices = require('puppeteer/DeviceDescriptors');
```

```
(async () => {

  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.emulate(devices['iPad Pro']);
  await page.goto('https://rivendellweb.net', { waitUntil: 'networkidle2'
  await page.screenshot({ path: 'full.png', fullPage: true});
  await browser.close();

})();
```

Disabling headless mode

There are time when we need to see what the headless browser is doing to troubleshoot, or just because we're curious. Puppeteer provides two tools to accomplish this as part of the options for `puppeteer.launch`:

- `headless` is a boolean that controls if the browser is launched in headless mode. Using false as the value will disable headless mode and let you see what the browser is doing
- `slowMo` will slow the browser by the specified number of miliseconds. This may let you actually see what the browser is doing since the actual process may be too fast to catch

The revised code looks like this:

```
const puppeteer = require('puppeteer');
const devices = require('puppeteer/DeviceDescriptors');

(async () => {

  const browser = await puppeteer.launch({
    headless: false,
    slowMo: 250,
  });
  const page = await browser.newPage();
  await page.emulate(devices['iPad Pro']);
  await page.goto('https://rivendellweb.net');
  await page.screenshot({ path: 'full.png'});
  await browser.close();

})();
```

Device emulation

`DeviceDescriptors` contains information about a set of predefined device descriptions to make it easier to use Puppeteer without having to manuall tweak the configuration.

It provides the following preconfigured information:

- userAgent string
- viewport
    - width
    - height
    - deviceScaleFactor
    - isMobile
    - hasTouch
    - isLandscape

For the list of supported devices check DeviceDescriptors.js in the Puppeteer Github repository.

When to consider the page loaded?

Particularly when working with lazy loaded resources, interacting with the page doesn't necessarily mean that we're done loading it. There may be videos that are still loading or images where intersection observers haven't triggered. It's important to be able to tell Puppeteer when we're done.

`waitUntil` is an optional parameter for `page.goto` that, given an array of one or more event strings, considers navigation to be successful after all events have been fired. Events can be:

- **load** - consider navigation to be finished when the load event is fired
- **domcontentloaded** - consider navigation to be finished when the

DOMContentLoaded event is fired
- **networkidle0** - consider navigation to be finished when there are no network connections for at least 500 ms
- **networkidle2** - consider navigation to be finished when there are no more than 2 network connections for at least 500 ms.

```
const puppeteer = require('puppeteer');
const devices = require('puppeteer/DeviceDescriptors');

(async () => {

  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.emulate(devices['iPad Pro']);
  await page.goto('https://rivendellweb.net', { waitUntil: 'networkidle2'
  await page.screenshot({ path: 'full.png'});
  await browser.close();
```

Changes to package.json

In order to save myself from typing all the commands to generate screenshots and to make sure Jest works as intended (and will be described in the next section) I've added the following blocks to my `package.json` file.

The first block specifies commands to run when using `npm test` and is a simpler way of running Jest in verbose mode.

The other commands run the screenshot scripts using `npm run` and the name of the script.

The second block, jest, configures Jest by disabling automock and configuring the test file names (all files that end with `_test.js`).

```
  "scripts": {
    "test": "jest --verbose",
    "jest --verbose""screenshot": "node screenshot/screenshot.js",
    "screenshot-full"ot/screenshot-full.js",
    "headfull": "nod": ""headfull": "node screenshot/screenshot-headfull.
  },
  "jest": {
    "automock"gex": "\\_tes": ""testRegex": "\\_test\\.js$"
  }
```

Page Testing

Expanding on the article at UI testing with Jest and Puppeteer: an introduction we'll look at how to test a form and the UI of the page. The form is available in the repository for this article at https://github.com/caraya/jest-puppeteer/blob/master/testing/form.html

We'll use the following libraries:

- Jest: a testing framework by Facebook. Jest provides a platform for automated testing along with a basic assertion library (Expect)
- Puppeteer: a Node.js library for controlling headless Chrome
- Faker: a Node.js library for generating random data like names, phones and addresses

In addition we'll set up Babel, preset-env and Babel libraries related to Jest.

The command to install the required Node modules is:

```
npm i -D jest puppeteer faker \
babel-core babel-jest babel-preset-env
```

Installing the required Node packages may take a long time. This is because Puppeteer installs a local version of Chromium (the open source project Chrome is based on) and ties its functionality to the specific version it installs.

You can force Puppeteer to use your locally installed version of Chrome or the Chromium open source browser but it's not guaranteed to work.

Once the modules are installed, we can start working through our testing script.

First we import all our module dependencies. We're using ES6 syntax, that's why we imported Babel and babel-jest.

```
import faker from "faker";
import puppeteer from "puppeteer";
import devices from 'puppeteer/DeviceDescriptors';
```

Next we setup variable and constants we'll use throughout the script. These variables are:

- APP points to the URL for the page we want to test
- `lead` is an array of randomly generated data created using Faker
- page and `browser` are Puppeteer variables we'll use later

```
const APP = "https://caraya.github.io/jest-puppeteer/testing/form.html";

const lead = {
  name: faker.name.firstName(),
  email: faker.internet.email(),
  phone: faker.phone.phoneNumber(),
  message: faker.random.words()
};

let page;
let browser;
```

The next two functions are part of Jest. They will be executed before and after each test repsectively. `beforeAll` sets up Puppeteer and works by launching it, starting the new page, emulate an iPad Pro and going to the page we want to test and waiting until all connections are finished.

I've chosen to use an iPad Pro as my emulated testing device rather than use the options for `puppeteer.launch()` to generate custom dimensions for the browser. The actual testing device is not important for this test. It may be for yours.

`afterAll` will close the browser connection.

```
beforeAll(async () => {
  browser = await puppeteer.launch();
  page = await browser.newPage();
  await page.emulate(devices['iPad Pro'])
  await page.goto(`${APP}`, { waitUntil: 'networkidle0' });
});

afterAll(() => {
  browser.close();
});
'networkidle0'
```

The first test uses Puppeteer to navigate and fill out a form.

`page.waitForSelector` waits for the selector to appear in page. If at the moment of calling the method the selector already exists, the method will return immediately.

`page.click` fetches an element with selector, scrolls it into view if needed, and then uses page.mouse to click in the center of the element. If there's no element matching selector, the method throws an error.

`page.type` sends a keydown, keypress/input, and keyup event for each character in the text. In this example, it will fill the field with the corresponding value from our lead array generated with Faker.

```
describe("Contact form", () => {
  test("lead can submit a cont"lead can submit a contact request"it page.
    await page.click("input[name=name]");
    await page.type("input[name=name]", lead.na"form"    await page.click("
    await page.type("input[name=email]", lead.email);
    await page.click("input[name=tel]");
    await page.type("input[name=tel]", lead.phone);
    await page.click("textarea[name=message]");
    await"input[name=email]"a[name=message]", lead.message);
    await page.click("input[type=checkbox]");
    "input[type=checkbox]"// await page.click("button[type=submit]");
  }, 16000);
});
```

The second test suite is more traditional and uses a combination of Puppeteer and Jest to perform assertion tests. Each test has a constant that sets the value we want to test and an expect-style test that test the condition against the value we want.

The first test checks that the title of the page is correct.

The second tests checks that there is an element with class navbar in the page. `page.$$eval` is the Puppeteer equivalent to `querySelectorAll`.

The final test checks that there are 6 elements with the field class. It uses `page.$$eval` to check for elements with class `field` and then tests that there are 6 of them.

```
describe("Testing the frontend", () => {
  test("assert that <title> is correct"assert that <title> is correct" = 
    expect(title).toBe("Demo form");
  });
  test("assert that a div named navbar exists", async () => {
    const navbar = await page"Demo form"vbar", el => (el ? true : false))
    expect(navbar).toBe(true);
  });
  test("assert that there are 6 fields", async () => {
    const fieldCount = await page.$$eval(".field", fields => fields.lengt
    expect(fieldCount).toBe(6)
  });
  "assert that there are 6 fields"// Insert more tests starting from here
});
```

There is more

If you look at the API docs for Puppeteer you'll see that there's plenty more you can do and more elaborate tests you can write.

We could turn the testing section into a full Test-Driven Development environment by writing the tests first and the code to match it.

Although this is a Chrome-only tool, I’m excited to see what else we can do with it.

Links and Resources

- Puppeteer API docs
- Puppeteer examples
- Making your UI Tests Resilient To Change
- UI testing with Jest and Puppeteer: an introduction
- Getting started with Jest