Guide to using Jest/Puppeteer testing system for Sogive

Run command syntax

Basic call syntax: ./jest.sh $ENDPOINT $JESTOPTIONS

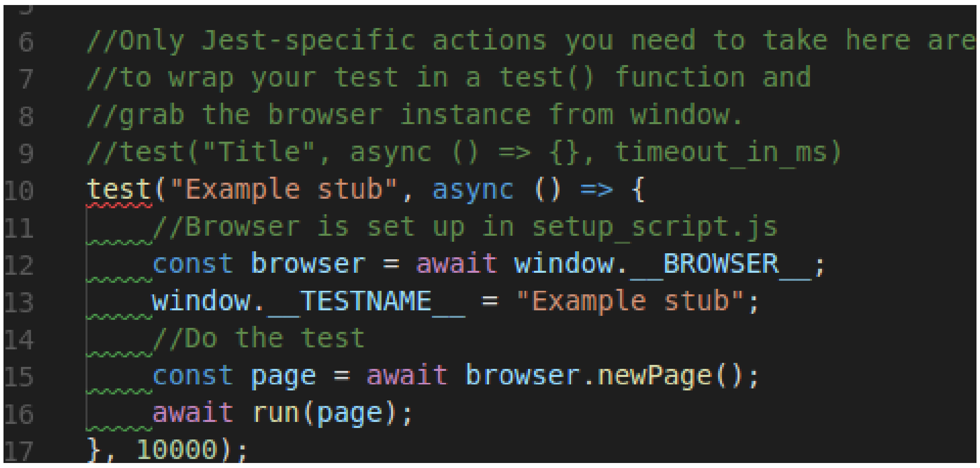
$ENDPOINT valid values: local, test, production

$JESTOPTIONS: a list of Jest CLI commands can be found here. These do not need to be modified in order to be picked up by jest.sh.

Running the puppeteer tests against the test server could then be achieved by calling ./jest.sh test. As an example of adding a Jest CLI command, you could force Jest to only run tests that contain the word “donation” by calling ./jest.sh test -t=”donation” (-t=”<regex>”).

Writing a test

Jest will attempt to run all js files placed in the __tests__ folder. For a block of code to be picked up as a test, it must be wrapped in a function call to test(“test_name”, async callback(), timeout_in_ms).

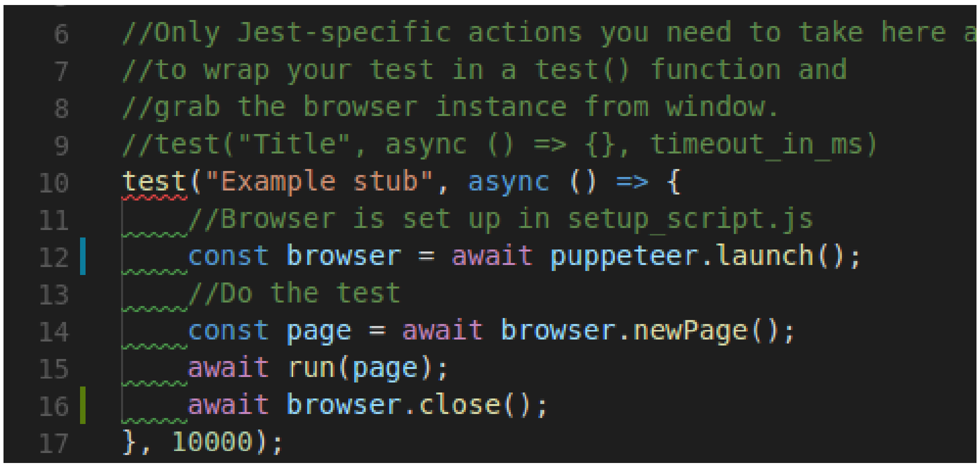


```
//Only Jest-specific actions you need to take here are
//to wrap your test in a test() function and
//grab the browser instance from window.
//test("Title", async () => {}, timeout_in_ms)
test("Example stub", async () => {
    //Browser is set up in setup_script.js
    const browser = await window.__BROWSER__;
    window.__TESTNAME__ = "Example stub";
    //Do the test
    const page = await browser.newPage();
    await run(page);
}, 10000);
```

Note that it isn’t necessary to import the function test into the js file: Jest handles this at run-time.

In the above example, you will note that a pre-instantiated browser object has been retrieved from window. This is always preferable to creating a new object within the test as it allows for the use of custom setup and teardown methods (defined in test-base/res/setup_script.js) across the entire test-suite.

The ability to define teardown methods is especially important, as it is the only way to guarantee that code is run after any given test. Imagine that the test had been written to create and then, after testing, close its own browser object.



```
 6  //Only Jest-specific actions you need to take here a
 7  //to wrap your test in a test() function and
 8  //grab the browser instance from window.
 9  //test("Title", async () => {}, timeout_in_ms)
10  test("Example stub", async () => {
11      //Browser is set up in setup_script.js
12      const browser = await puppeteer.launch();
13      //Do the test
14      const page = await browser.newPage();
15      await run(page);
16      await browser.close();
17  }, 10000);
```

As tests abort when an exception is encountered, any issue encountered in “await run(page)” (line 15) will mean that the browser instance is never closed.

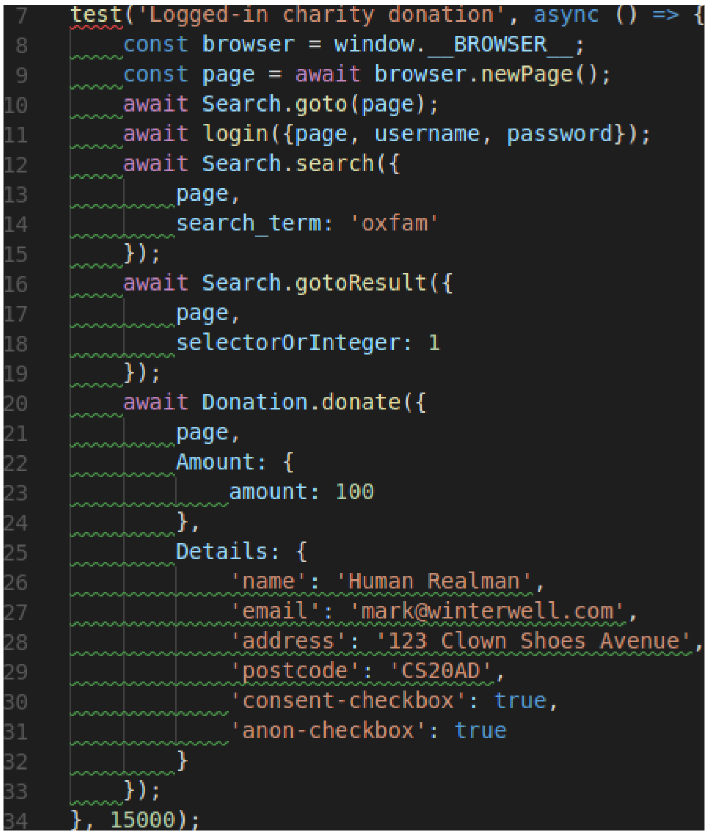
Anything defined in Jest’s beforeEach and afterEach methods is guaranteed to run before or after each test respectively – this makes them a much safer option for writing set-up and clean-up code.

The puppeteer code

The aim of this section is to describe and explain how the existing puppeteer code-base has been managed so far.

Files for use by puppeteer can be found in two places: sogive-scripts and test-base/res/UtilityFunctions. In an ideal world, all sogive-specific scripts would go in sogive-scripts, and all general purpose functions (logging in, filling in forms, etc.) would go in test-base/res/UtilityFunctions.

Rather than writing scripts to carry out one specific test (donating £1 to Puppeteer fundraiser) I have tended to write reusable blocks of code (donate £n to any given fundraiser). Having discrete blocks of puppeteer code is nice as it allows us to chop-and-change actions that touch upon different areas of SoGive. An example – where we donate to a charity by first searching for it in SoGive – is shown below.



```
test('Logged-in charity donation', async () => {
    const browser = window.__BROWSER__;
    const page = await browser.newPage();
    await Search.goto(page);
    await login({page, username, password});
    await Search.search({
        page,
        search_term: 'oxfam'
    });
    await Search.gotoResult({
        page,
        selectorOrInteger: 1
    });
    await Donation.donate({
        page,
        Amount: {
            amount: 100
        },
        Details: {
            'name': 'Human Realman',
            'email': 'mark@winterwell.com',
            'address': '123 Clown Shoes Avenue',
            'postcode': 'CS20AD',
            'consent-checkbox': true,
            'anon-checkbox': true
        }
    });
}, 15000);
```

The blocks of puppeteer code are organised primarily by the area of the site that they cover (e.g fundraiser for sogive.org/#fundraiser).

Issues with pages loading

Throughout the puppeteer code, you will notice "page.waitForSelector()" statements scattered hither and thither. This is to solve a particular issue where puppeteer considers SoGive pages to be loaded before there are any actual elements on the page. If you find that your tests are failing to click on an element that you know should be there, this is probably what's wrong.

Forcing puppeteer to wait for a specific element to appear before carrying on with its commands is an effective work-around.

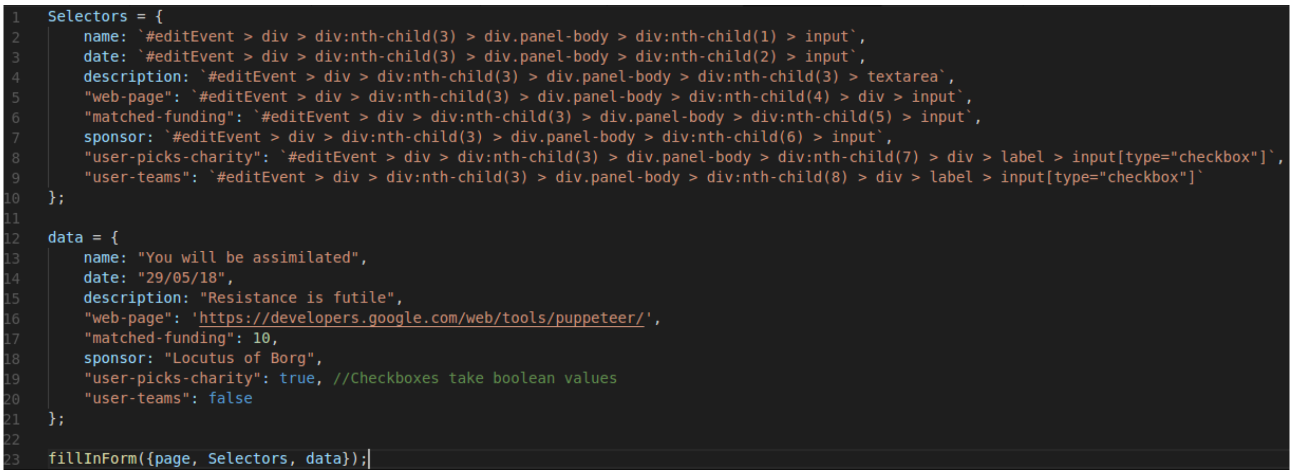
Selectors.js

Puppeteer can be directed to click on a page element by either of two methods: absolute (x,y) page coordinates and CSS selectors. The latter have been used throughout as I believe them to be much more reliable.

All of these selectors have been packaged into Selectors.js. The primary purpose of this was to remove clutter, but it is also a useful way of managing selectors that appear in many different places. The donation widget is a good example of this.

fillInForm({page, data, Selectors})

Selectors relating to form elements (Fundraiser.EditFundraiser, General.DonationForm, etc.) have been written to work with fillInForm. Example data structures to fill in an #editEvent page form are shown below.



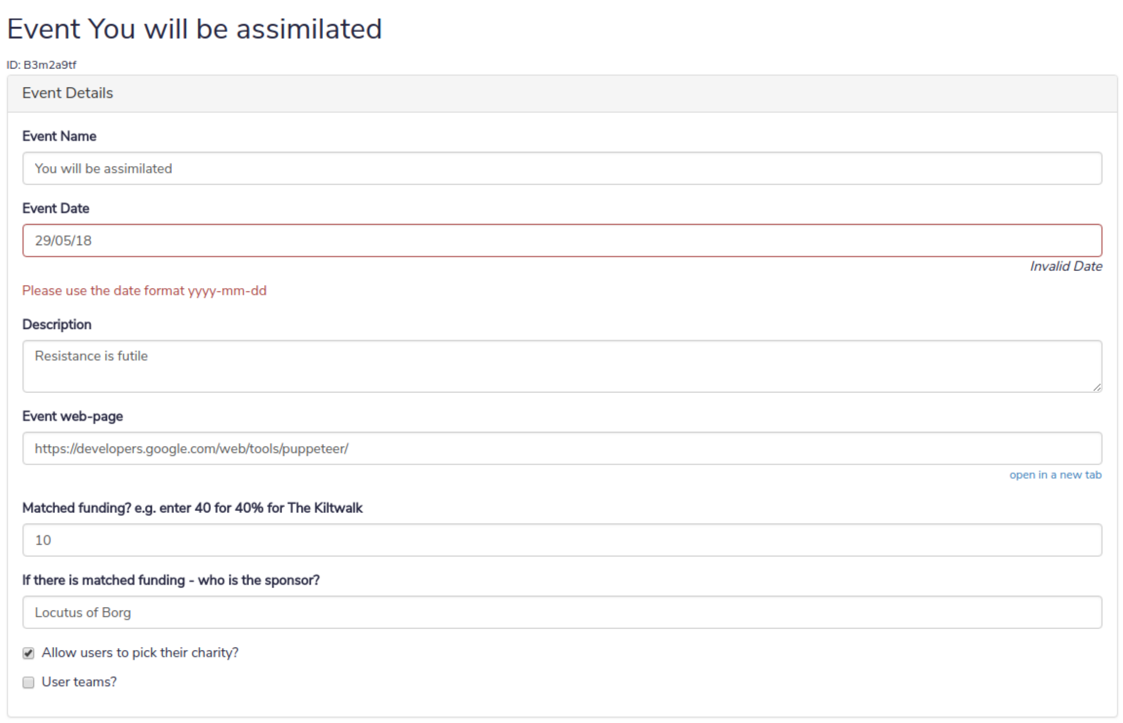
```
Selectors = {
    name: `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(1) > input`,
    date: `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(2) > input`,
    description: `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(3) > textarea`,
    "web-page": `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(4) > div > input`,
    "matched-funding": `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(5) > input`,
    sponsor: `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(6) > input`,
    "user-picks-charity": `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(7) > div > label > input[type="checkbox"]`,
    "user-teams": `#editEvent > div > div:nth-child(3) > div.panel-body > div:nth-child(8) > div > label > input[type="checkbox"]`
};

data = {
    name: "You will be assimilated",
    date: "29/05/18",
    description: "Resistance is futile",
    "web-page": 'https://developers.google.com/web/tools/puppeteer/',
    "matched-funding": 10,
    sponsor: "Locutus of Borg",
    "user-picks-charity": true, //Checkboxes take boolean values
    "user-teams": false
};

fillInForm({page, Selectors, data});
```

As an aside, data values corresponding to a checkbox should be given a boolean value; fields with default values will be automatically cleared before anything is entered.

Running the above command while on a valid #editEvent page, puppeteer will fill out the form as shown below.


Event You will be assimilated

ID: B3m2a9tf

Event Details

Event Name

You will be assimilated

Event Date

29/05/18

Invalid Date

Please use the date format yyyy-mm-dd

Description

Resistance is futile

Event web-page

https://developers.google.com/web/tools/puppeteer/

open in a new tab

Matched funding? e.g. enter 40 for 40% for The Kiltwalk

10

If there is matched funding - who is the sponsor?

Locutus of Borg

- [x] Allow users to pick their charity?
- [ ] User teams?

It is important to note that fillInForm is not magic: it can only fill in fields currently present on the page. An example of how to handle multi-stage forms (such as the donation widget) can be found in sogive-scripts/donation-form.