Building a performance test suite

Ever so often you see questions in Quora about Brave, a Chromium-based browser that claims to be faster than stock Chromium and other derivative browsers like Edge, or Opera on Desktop.

The last time I saw the performance testing results and methodology was when they first released version 1.0, documented in Brave 1.0 Performance: Methodology and Results with the code used to do the analysis available on Github but I haven't seen more recent data.

The tests were done in older versions of the browsers tested and the Brave Browser Comparison Kit hasn't been updated for newer versions (not that I would expect tthem to), so the best solution is to create my own tests using Playwright to see if the performance gap has been reduced or the 3x claim speed improvement over other browsers is still valid.

From this original idea, the notion evolved into using Playwright to do cross-platform performance testing. What can we do everywhere, what's specific to browsers or performance libraries. Does Playwright work better than WebDriver (even though WebDriver is a W3C spec)? Does the tool used to test affect the results of the test?

Getting started: Implementing the playwright script

I've broken the initial Playwright script into smaller sections that are easier to explain.

Configuring the browser

One of the things Playwright does that allow us to configure which Chromium browser we want to run the tests with.

The following configuration will run the test with an instance of Chrome Canary installed on my macOS laptop. It will run in headless mode.

```
// Set the executable to Chrome canary
const chrome = await playwright.chromium.launch({
  headless: true,
  executablePath: `/Applications/Google\ Chrome\ Canary.app/Contents/MacOS
})
```

We can configure other Chromium-based browsers. I chose to work Edge Canary and Brave Nightly. I believe other Chromium browsers would also work. I haven't tested them.

We can also use the default Playwright configurations to run Firefox and WebKit. We configure both browsers to run in headless mode

```
// TESTING WITH FIREFOX
const firefox = await playwright.firefox.launch({
  headless: true,
})

// TESTING WITH WEBKIT
const webkit = await playwright.webkit.launch({
  headless: true,
})
```

Basic performance configuration

Below are some basic examples of how to test for performance items using the deprecated `window.performance.timing` object.

`performanceTimingJson` uses Playwright's evaluate to get the values of the `window.performance.timing` object.

We then create another variable with the parsed result of `performanceTimingJson`.

We then run some calculations to get values for some performance metrics that I'm interested in.

```
let performanceTimingJson = await page.evaluate(() => {
  JSON.stringify(window.performance.timing)
})
let pt = JSON.parse(performanceTimingJson)

let startToInteractive = pt.domInteractive - pt.navigationStart
let domContentLoadedComplete = pt.domContentLoadedEventEnd - pt.navigation
let startToComplete = pt.domComplete - pt.navigationStart
```

The other way to get performance data is to use `performance.getEntriesByName` to query different types of performance entries. In this case we create variables for `firstPaint` and `firstContentfulPaint`.

These tests produce different results than the other tests we configured. The values for paint-related events are the high resolution time when the event happened and have a duration of 0.

```
let firstPaint = JSON.parse(
  await page.evaluate(() =>
    JSON.stringify(performance.getEntriesByName('first-paint'))
  )
);

let firstContentfulPaint = JSON.parse(
  await page.evaluate(() =>
    JSON.stringify(performance.getEntriesByName('first-contentful-paint'))
  )
);
```

Adding tracing collection

One thing I hadn't realized until I saw it in Playwright's documentation is that you can create a DevTools trace that we can then feed back into DevTools or the trace viewer available as a web application or directly in the browser via `chrome://tracing/`

```
const chrome = await playwright.chromium.launch({
  headless: false,
  executablePath: `/Applications/Google\ Chrome\ Canary.app/Contents/MacOS
})
// create new chrome instance
const page = await chrome.newPage();
await chrome.startTracing(page, {path: 'chrome-trace.json'});
'chrome-trace.json'// go to page
await page.goto('https://example.com')
await chrome.stopTracing();

// Add performance tests here

await chrome.close();
```

If needed, we could also add CSS and Javascript coverage, equivalent to the coverage menu in Chromium-based browsers.

Adding lighthouse data to Chromium browser runs

The playwright-lighthouse provides an interface between Playwright and Lighthouse to gather results from Lighthouse runs as part of a Playwright automation script.

We customize the Chrome launcher by adding a `remote-debugging-port` argument.

After we go to the page we want to review, we configure the `playwright-lighthouse` script by adding a configuration to `playAudit`.

We can generate three formats for the report: CSV, JSON and HTML (the same format that you see when you run lighthouse in Chrome DevTools).

We can also choose a location and a file name for the report. Each version will get the appropriate extension.

The last thing is to close the browser.

```
const playwright = require('playwright-core');
const { playAudit } = require('playwright-lighthouse');

(async () => {
  'playwright-lighthouse'aywright.chromium.launch({
    headless: true,
    args: ['--remote-debugging-port=9222'],
  })
  const page = await chrome.newPage()
  await page.goto('https://publishing-project.rivendellweb.net')

'--remote-debugging-port=9222'page,
    port: 9222,
    thresholds: {
      performance: 50,
      accessibility: 50,
      'best-practices': 50,
      seo: 50,
      pwa: 50,
    },
    reports: {
      formats: {
        json: false, 'best-practices'//defaults to false
        html: true, //defaults to false
        csv: false, //defaults to false
      },
      name: `lighthouse-${new Date().getTime()}`,
      directory:  `${process.cwd()}/lighthouse-${new Date().getDate()}`,
    },
  });

  await chrome.close();
})()
`lighthouse-${new Date().getTime()}`
```

Writing the results to a file

The final part of this basic performance setup is to write the unpublished results to

a file. Since were mixing stringified JSON and plain text content the attempt at making it all readable is not as straighforward as I thought it would.

A working firrst step uses Node's fs module to append an array and several lines of text to a file. If the file doesn't exist the module will create it for us. I make the file unique by appending a new data object to the name.

```
dataToAppend = (data.join('\n'));

fs.appendFile(`chrome-performance-${new Date()}.txt`, dataToAppend, funct
  if (err) {
    throw err;
  } else {
    console.log('Performance data saved to file');
  }
});
'Performance data saved to file'
```

The problem with the code is that the first array object is stored as a single line, making it very difficult to read. Ideally we'd be able to pretty print it to make it easier for humans to read it.

Fortunately, JSON.stringify() can do it for us.

The third argument in the stringify method provides a String or Number used to insert white space into the output JSON string for readability purposes.

...

If this is a String, the string (or the first 10 characters of the string, if it's longer than that) is used as white space. If this parameter is not provided (or is null), no white space is used.

so we can use a tab character escape sequence as the third parameter to the stringify method:

```
JSON.stringify(pt, null, '\t'),
```

It produces the following result that we can do further processing later in our build

process.


```
{
  "connectStart": 1614063600439,
  "navigationStart": 1614063600381,
  "loadEventEnd": 1614063603698,
  "domLoading": 1614063602269,
  "secureConnectionStart": 1614063600451,
  "fetchStart": 1614063600381,
  "domContentLoadedEventStart": 1614063603387,
  "responseStart": 1614063602263,
  "responseEnd": 1614063602320,
  "domInteractive": 1614063603353,
  "domainLookupEnd": 1614063600439,
  "redirectStart": 0,
  "requestStart": 1614063600538,
  "unloadEventEnd": 0,
  "unloadEventStart": 0,
  "domComplete": 1614063603696,
  "domainLookupStart": 1614063600383,
  "loadEventStart": 1614063603696,
  "domContentLoadedEventEnd": 1614063603389,
  "redirectEnd": 0,
  "connectEnd": 1614063600538
}
```


To make the remaining lines legible, we can insert return characters `\n` before and after each line.

Customization: Entering the URL from the command line

One final customization I'd like to do is to have the ability to enter the URL to test from the command line rather than hard-code it on the script. This will give us the ability to run it multiple times from a shell script.

We capture the URL from the command line using Node's process.argv()

method.

We know that the name of the script is the second value (zero-index) because the values are:

- 0: Node (the command to run)
- 1: Name of the script that Node will run
- 2 and later: additional parameters

In the async anonymous IFFE we use to run the code we add the URL as a parameter with a default value; this default value prevents errors when we don't add a URL in the command line.

The changes to the code are simple:

```
const url = process.argv[2];

(async (url="https://publishing-project.rivendellweb.net") => {

// code goes here

})()
```

Using shell scripts to enchance the playwright script

I built a list of 85 URLs that are possible candidates for testing by combining the URLs in the Brave testing list with the Alexa Top 50 websites taken from the larger top 500 sites report on 2/23/21.

This will only work for Linux, Mac and Windows using WSL (Window Subsystem for Linux). I currently don't have an equivalent for Power Shell.

The script does the following:

1. Sets up the path to the Bash shell executable
2. Seeds the random number generator that we'll use to select the random URL

3. Loads the data from fullsites.txt into the `urls` variable
4. Creates a variable with a randomly selected URL from the urls variable
5. Echos the URL we're testing
6. Runs the `performance-test` Node script with the random URL as the parameter

```
#!/usr/local/bin/bash #1

# 2
RANDOM=$$$(date +%s)

$(date +%s)#3
mapfile urls <fullset.txt

# 4
URL_TO_TESTNDOM % ${#urls[@]}]}

# 5
echo Te${#urls[@]}# 5
echo Testing $URL_TO_TEST

# 6
node performance-test.js URL_TO_TEST
```

Conclusions and final touches

As presented the scripts do a lot:

- Generate traces that developers can review later
- Generate Lighthouse reports in machine readable (JSON) and human readable (HTML) formats
- Generate some Core Vital metrics
- Produce other performance metrics

We can refine the script in different ways:

- There are more metrics we can incorporate to the scripts
- We can choose to put all the browser tests in one browser or put them in separate files and call them from the main shell script

- Find a better way to randomize the URLs to choose. The built-in randomizing function in Bash is not as random as I'd like

If you have any specific idea for improvevment, please open an issue in the project's Github repo or contact me on Twitter @elrond25