Build a Node.js Tool to Record and Compare Google Lighthouse Reports

Luke Harrison on Mar 16, 2020

In this tutorial, I'll show you step by step how to create a simple tool in Node.js to run Google Lighthouse audits via the command line, save the reports they generate in JSON format and then compare them so web performance can be monitored as the website grows and develops.

I'm hopeful this can serve as a good introduction for any developer interested in learning about how to work with Google Lighthouse programmatically.

But first, for the uninitiated...

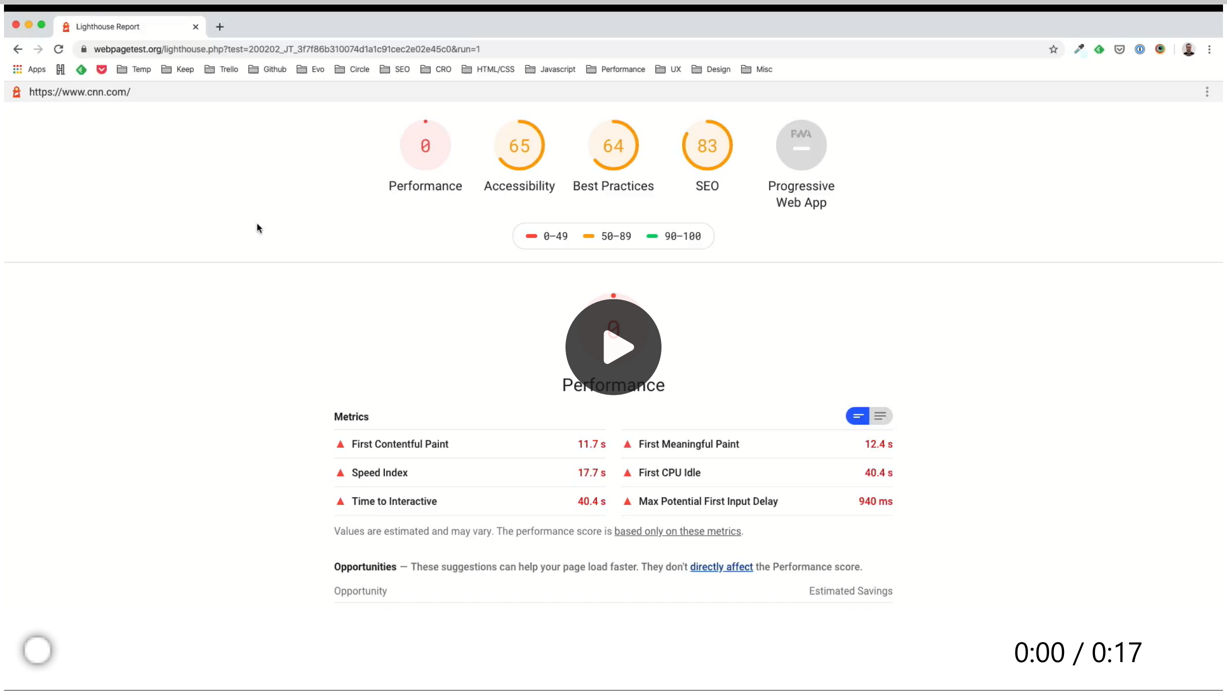
(#aa-what-is-google-lighthouse) What is Google Lighthouse?

Google Lighthouse is one of the best-automated tools available on a web developer's utility belt. It allows you to quickly audit a website in a number of key areas which together can form a measure of its overall quality. These are:

- Performance
- Accessibility
- Best Practices
- SEO
- Progressive Web App

Once the audit is complete, a report is then generated on what your website does well... and not so well, with the latter intending to serve as an indicator for what your next steps should be to improve the page.

Here's what a full report looks like.



Metrics

▲ First Contentful Paint	11.7 s	▲ First Meaningful Paint	12.4 s
▲ Speed Index	17.7 s	▲ First CPU Idle	40.4 s
▲ Time to Interactive	40.4 s	▲ Max Potential First Input Delay	940 ms

Values are estimated and may vary. The performance score is based only on these metrics.

Opportunities — These suggestions can help your page load faster. They don't directly affect the Performance score.

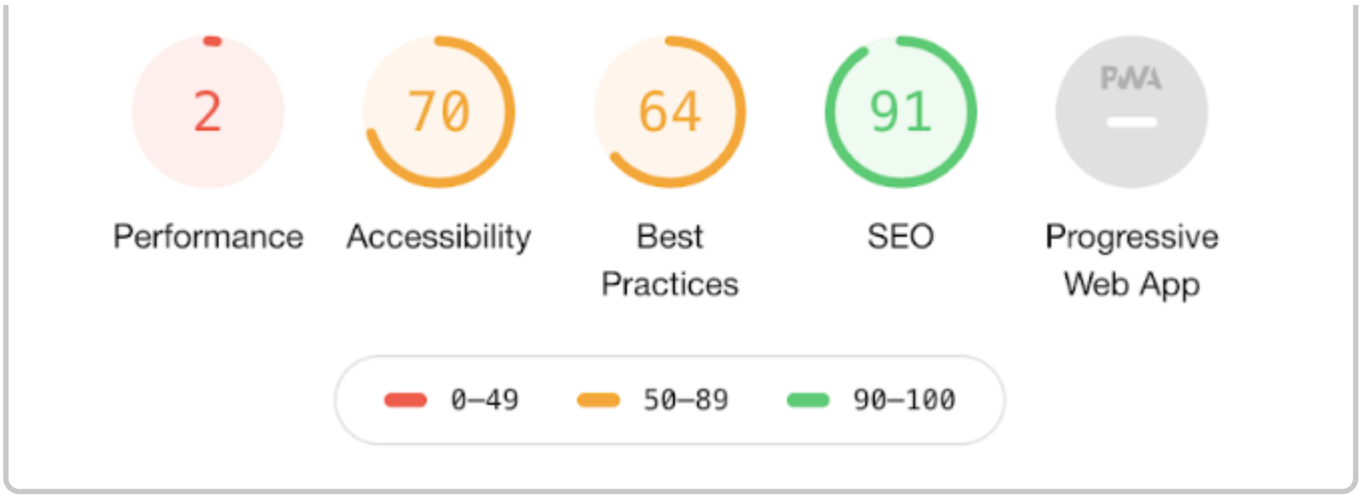
Opportunity Estimated Savings

Along with other general diagnostics and web performance metrics, a really useful feature of the report is that each of the key areas is aggregated into color-coded scores between 0-100.

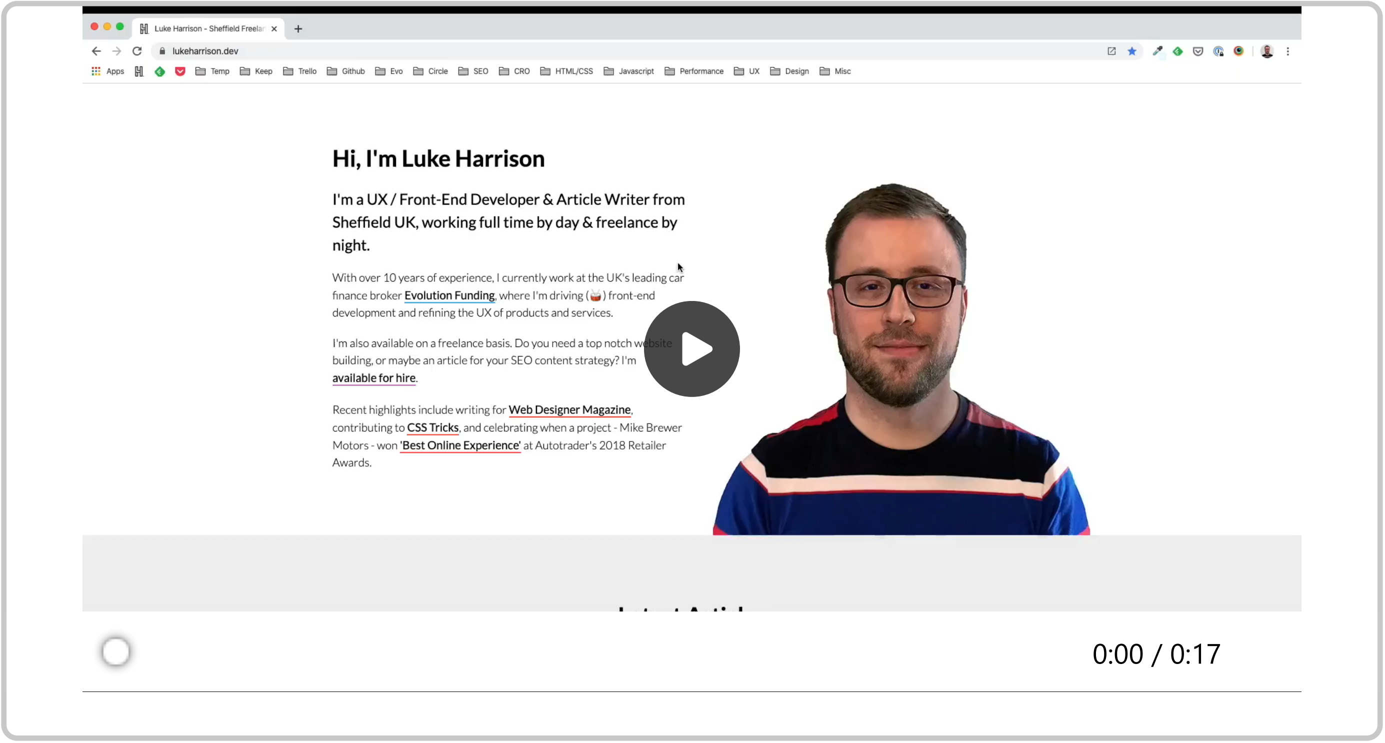
Not only does this allow developers to quickly gauge the quality of a website without further analysis, but it also allows non-technical folk such as stakeholders or clients to understand as well.

For example, this means, it's much easier to share the win with Heather from marketing after spending time improving website accessibility as she's more able to appreciate the effort after seeing the Lighthouse accessibility score go up 50 points into the green.

But equally, Simon the project manager may not understand what Speed Index or First Contentful Paint means, but when he sees the Lighthouse report showing website performance score knee deep in the red, he knows you still have work to do.



If you're in Chrome or the latest version of Edge, you can run a Lighthouse audit for yourself right now using DevTools. Here's how:



Hi, I'm Luke Harrison

I'm a UX / Front-End Developer & Article Writer from Sheffield UK, working full time by day & freelance by night.

With over 10 years of experience, I currently work at the UK's leading car finance broker Evolution Funding, where I'm driving ( ) front-end development and refining the UX of products and services.

I'm also available on a freelance basis. Do you need a top notch website building, or maybe an article for your SEO content strategy? I'm available for hire.

Recent highlights include writing for Web Designer Magazine, contributing to CSS Tricks, and celebrating when a project - Mike Brewer Motors - won 'Best Online Experience' at Autotrader's 2018 Retailer Awards.

You can also run a Lighthouse audit online via PageSpeed Insights (https://developers.google.com/speed/pagespeed/insights/) or through popular performance tools, such as WebPageTest (https://www.webpagetest.org/) .

However, today, we're only interested in Lighthouse as a Node module, as this allows us to use the tool programmatically to audit, record and compare web performance metrics.

Let's find out how.

(#aa-setup) Setup

First off, if you don't already have it, you're going to need Node.js. There are a million different ways to install it. I use the Homebrew package manager, but you can also download an installer straight from the Node.js website (https://nodejs.org/en/) if you prefer. This tutorial was written with Node.js v10.17.0 in mind, but will very likely work just fine on the most versions released in the last few years.

You're also going to need Chrome installed, as that's how we'll be running the Lighthouse audits.

Next, create a new directory for the project and then `cd` into it in the console. Then run `npm init` to begin to create a `package.json` file. At this point, I'd recommend just bashing the Enter key over and over to skip as much of this as possible until the file is created.

Now, let's create a new file in the project directory. I called mine `lh.js`, but feel free to call it whatever you want. This will contain all of JavaScript for the tool. Open it in your text editor of choice, and for now, write a `console.log` statement.

JavaScript
```
console.log('Hello world');
```

Then in the console, make sure your CWD (current working directory) is your project directory and run `node lh.js`, substituting my file name for whatever you've used.

You should see:

Terminal
```
$ node lh.js
Hello world
```

If not, then check your Node installation is working and you're definitely in the correct project directory.

Now that's out of the way, we can move on to developing the tool itself.

(#aa-opening-chrome-with-node-js) Opening Chrome with Node.js

Let's install our project's first dependency: Lighthouse itself.

Terminal

```
npm install lighthouse --save-dev
```

This creates a `node_modules` directory that contains all of the package's files. If you're using Git, the only thing you'll want to do with this is add it to your `.gitignore` file.

In `lh.js`, you'll next want to delete the test `console.log()` and import the Lighthouse module so you can use it in your code. Like so:

JavaScript

```
const lighthouse = require('lighthouse');
```

Below it, you'll also need to import a module called chrome-launcher (https://github.com/GoogleChrome/chrome-launcher) , which is one of Lighthouse's dependencies and allows Node to launch Chrome by itself so the audit can be run.

JavaScript

```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
```

Now that we have access to these two modules, let's create a simple script which just opens Chrome, runs a Lighthouse audit, and then prints the report to the console.

Create a new function that accepts a URL as a parameter. Because we'll be running this using

Node.js, we're able to safely use ES6 syntax as we don't have to worry about those pesky Internet Explorer users.

JavaScript

```javascript
const launchChrome = (url) => {

}
```

Within the function, the first thing we need to do is open Chrome using the chrome-launcher module we imported and send it to whatever argument is passed through the `url` parameter.

We can do this using its `launch()` method and its `startingUrl` option.

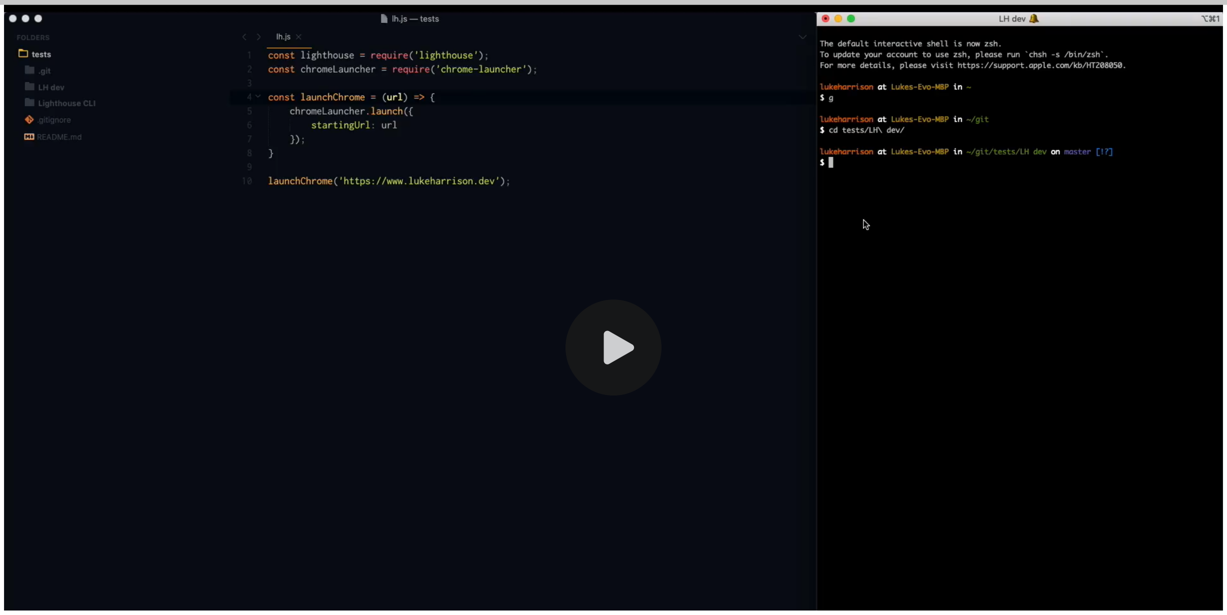
JavaScript

```javascript
const launchChrome = url => {
  chromeLauncher.launch({
    startingUrl: url
  });
};
```

Calling the function below and passing a URL of your choice results in Chrome being opened at the URL when the Node script is run.

JavaScript

```javascript
launchChrome('https://www.lukeharrison.dev');
```



```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');

const launchChrome = (url) => {
    chromeLauncher.launch({
        startingUrl: url
    });
}

launchChrome('https://www.lukeharrison.dev');
```

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.

lukeharrison at Lukes-Evo-MBP in ~
$ g

lukeharrison at Lukes-Evo-MBP in ~/git
$ cd tests/LH\ dev/

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev on master [!?]
$
```

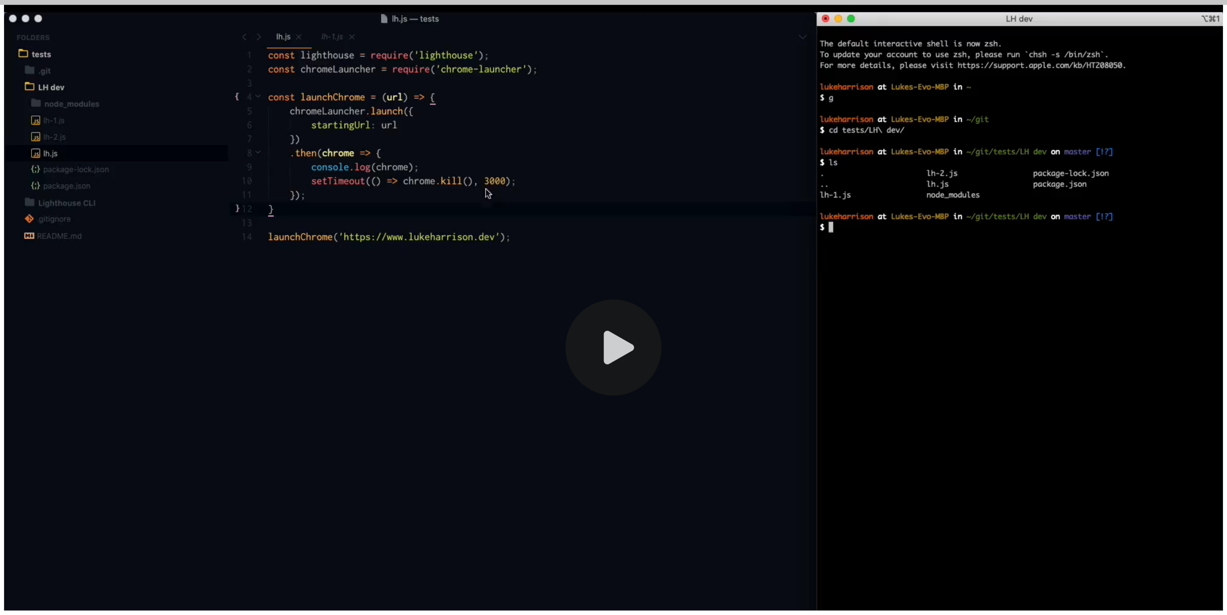
0:00 / 0:08

The launch function actually returns a promise, which allows us to access an object containing a few useful methods and properties.

For example, using the code below, we can open Chrome, print the object to the console, and then close Chrome three seconds later using its `kill()` method.

JavaScript

```javascript
const launchChrome = url => {
  chromeLauncher
    .launch({
      startingUrl: url
    })
    .then(chrome => {
      console.log(chrome);
      setTimeout(() => chrome.kill(), 3000);
    });
};

launchChrome("https://www.lukeharrison.dev");
```



```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');

const launchChrome = (url) => {
    chromeLauncher.launch({
        startingUrl: url
    })
    .then(chrome => {
        console.log(chrome);
        setTimeout(() => chrome.kill(), 3000);
    });
}

launchChrome('https://www.lukeharrison.dev');
```

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.

lukeharrison at Lukes-Evo-MBP in ~
$ g

lukeharrison at Lukes-Evo-MBP in ~/git
$ cd tests/LH\ dev/

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev on master [!?]
$ ls
.                       lh-2.js                 package-lock.json
..                      lh.js                   package.json
lh-1.js                 node_modules

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev on master [!?]
$
```

0:00 / 0:07

Now that we've got Chrome figured out, let's move on to Lighthouse.

(#aa-running-lighthouse-programmatically) Running Lighthouse programmatically

First off, let's rename our `launchChrome()` function to something more reflective of its final functionality: `launchChromeAndRunLighthouse()`. With the hard part out of the way, we can now use the Lighthouse module we imported earlier in the tutorial.

In the Chrome launcher's then function, which only executes once the browser is open, we'll pass Lighthouse the function's `url` argument and trigger an audit of this website.

JavaScript

```
const launchChromeAndRunLighthouse = url => {
  chromeLauncher
    .launch({
      startingUrl: url
    })
    .then(chrome => {
      const opts = {
```

```
      port: chrome.port
    };
    lighthouse(url, opts);
  });
};

launchChromeAndRunLighthouse("https://www.lukeharrison.dev");
```

To link the lighthouse instance to our Chrome browser window, we have to pass its port along with the URL.

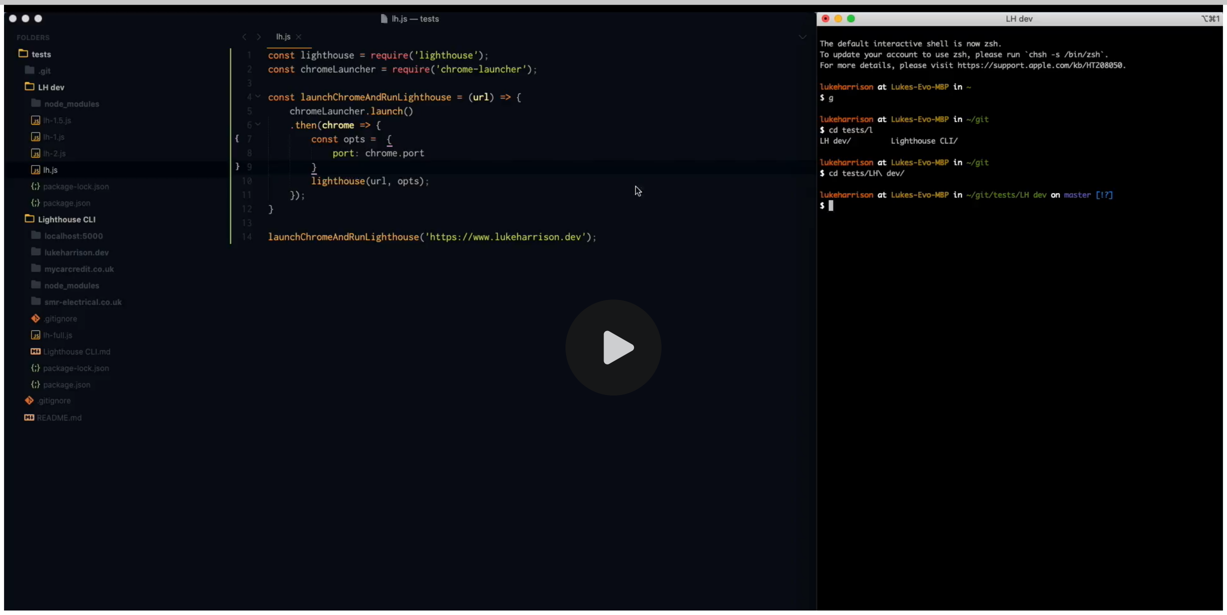
If you were to run this script now, you will hit an error in the console:

```
Console
(node:47714) UnhandledPromiseRejectionWarning: Error: You probably have multiple tabs open to the same origin.
```

To fix this, we just need to remove the `startingUrl` option from Chrome Launcher and let Lighthouse handle URL navigation from here on out.

```
JavaScript
const launchChromeAndRunLighthouse = url => {
  chromeLauncher.launch().then(chrome => {
    const opts = {
      port: chrome.port
    };
    lighthouse(url, opts);
  });
};
```

If you were to execute this code, you'll notice that something definitely seems to be happening. We just aren't getting any feedback in the console to confirm the Lighthouse audit has definitely run, nor is the Chrome instance closing by itself like before.



```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');

const launchChromeAndRunLighthouse = (url) => {
    chromeLauncher.launch()
    .then(chrome => {
        const opts =  {
            port: chrome.port
        }
        lighthouse(url, opts);
    });
}

launchChromeAndRunLighthouse('https://www.lukeharrison.dev');
```

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.

lukeharrison at Lukes-Evo-MBP in ~
$ g

lukeharrison at Lukes-Evo-MBP in ~/git
$ cd tests/l
LH dev/         Lighthouse CLI/

lukeharrison at Lukes-Evo-MBP in ~/git
$ cd tests/LH\ dev/

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev on master [!?]
$ 
```

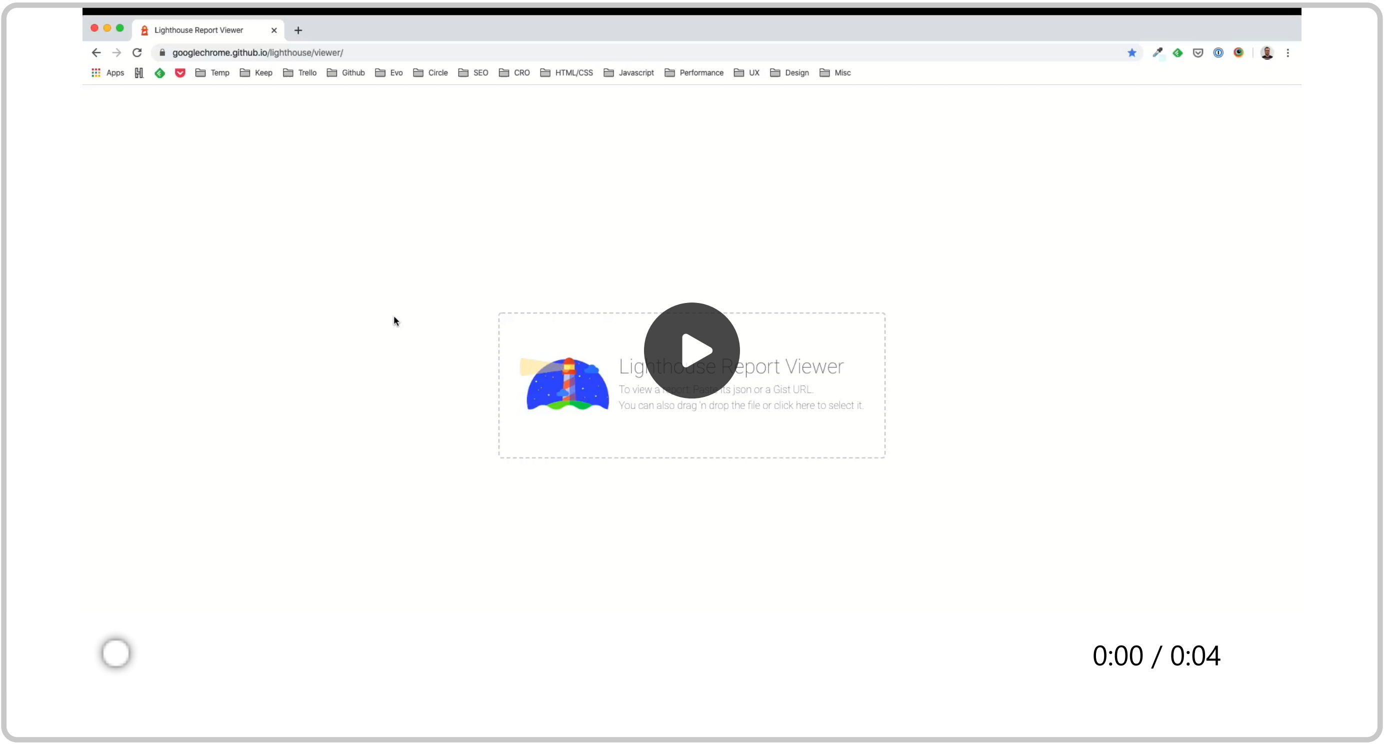
Thankfully, the `lighthouse()` function returns a promise which lets us access the audit results.

Let's kill Chrome and then print those results to the terminal in JSON format via the report property of the results object.

JavaScript

```
const launchChromeAndRunLighthouse = url => {
  chromeLauncher.launch().then(chrome => {
    const opts = {
      port: chrome.port
    };
    lighthouse(url, opts).then(results => {
      chrome.kill();
      console.log(results.report);
    });
  });
};
```

While the console isn't the best way to display these results, if you were to copy them to your clipboard and visit the Lighthouse Report Viewer (https://googlechrome.github.io/lighthouse/viewer/) , pasting here will show the report in all of its glory.


Lighthouse Report Viewer
googlechrome.github.io/lighthouse/viewer/
Lighthouse Report Viewer
To view a report: Paste its json or a Gist URL.
You can also drag 'n drop the file or click here to select it.
0:00 / 0:04

At this point, it's important to tidy up the code a little to make the `launchChromeAndRunLighthouse()` function return the report once it's finished executing. This allows us to process the report later without resulting in a messy pyramid of JavaScript.

JavaScript

```
const lighthouse = require("lighthouse");
const chromeLauncher = require("chrome-launcher");

const launchChromeAndRunLighthouse = url => {
  return chromeLauncher.launch().then(chrome => {
    const opts = {
      port: chrome.port
    };
    return lighthouse(url, opts).then(results => {
      return chrome.kill().then(() => results.report);
    });
  });
};

launchChromeAndRunLighthouse("https://www.lukeharrison.dev").then(results => {
  console.log(results);
});
```

One thing you may have noticed is that our tool is only able to audit a single website at the moment. Let's change this so you can pass the URL as an argument via the command line.

To take the pain out of working with command-line arguments, we'll handle them with a package called yargs (https://www.npmjs.com/package/yargs) .

Terminal
```
npm install --save-dev yargs
```

Then import it at the top of your script along with Chrome Launcher and Lighthouse. We only need its `argv` function here.

JavaScript
```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
const argv = require('yargs').argv;
```

This means if you were to pass a command line argument in the terminal like so:

Command Line
```
node lh.js --url https://www.google.co.uk
```

...you can access the argument in the script like so:

JavaScript
```
const url = argv.url // https://www.google.co.uk
```

Let's edit our script to pass the command line URL argument to the function's `url` parameter. It's important to add a little safety net via the `if` statement and error message in case no argument is passed.

JavaScript
```
if (argv.url) {
  launchChromeAndRunLighthouse(argv.url).then(results => {
    console.log(results);
  });
} else {
  throw "You haven't passed a URL to Lighthouse";
}
```

Tada! We have a tool that launches Chrome and runs a Lighthouse audit programmatically before printing the report to the terminal in JSON format.

(#aa-saving-lighthouse-reports) Saving Lighthouse reports

Having the report printed to the console isn't very useful as you can't easily read its contents, nor are they aren't saved for future use. In this section of the tutorial, we'll change this behavior so each report is saved into its own JSON file.

To stop reports from different websites getting mixed up, we'll organize them like so:

- lukeharrison.dev
  - 2020-01-31T18:18:12.648Z.json
  - 2020-01-31T19:10:24.110Z.json
- cnn.com
  - 2020-01-14T22:15:10.396Z.json
- lh.js

We'll name the reports with a timestamp indicating when the date/time the report was generated. This will mean no two report file names will ever be the same, and it'll help us easily distinguish between reports.

There is one issue with Windows that requires our attention: the colon (`:`) is an illegal character for file names. To mitigate this issue, we'll replace any colons with underscores (`_`), so a typical report filename will look like:

- 2020-01-31T18_18_12.648Z.json

(#aa-creating-the-directory) Creating the directory

First, we need to manipulate the command line URL argument so we can use it for the directory name.

This involves more than just removing the `www`, as it needs to account for audits run on web pages which don't sit at the root (eg: `www.foo.com/bar`), as the slashes are invalid characters for directory names.

For these URLs, we'll replace the invalid characters with underscores again. That way, if you run an audit on `https://www.foo.com/bar`, the resulting directory name containing the report would be `foo.com_bar`.

To make dealing with URLs easier, we'll use a native Node.js module called url (https://nodejs.org/api/url.html) . This can be imported like any other package and without having to add it to the`package.json` and pull it via npm.

JavaScript
```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
const argv = require('yargs').argv;
const url = require('url');
```

Next, let's use it to instantiate a new URL object.

JavaScript
```
if (argv.url) {
  const urlObj = new URL(argv.url);

  launchChromeAndRunLighthouse(argv.url).then(results => {
    console.log(results);
  });
}
```

If you were to print `urlObj` to the console, you would see lots of useful URL data we can use.

Command Line
```
$ node lh.js --url https://www.foo.com/bar
URL {
  href: 'https://www.foo.com/bar',
  origin: 'https://www.foo.com',
  protocol: 'https:',
  username: '',
  password: '',
  host: 'www.foo.com',
  hostname: 'www.foo.com',
  port: '',
  pathname: '/bar',
  search: '',
  searchParams: URLSearchParams {},
  hash: ''
}
```

Create a new variable called `dirName`, and use the string `replace()` method on the host property of our URL to get rid of the `www` in addition to the `https` protocol:

JavaScript
```javascript
const urlObj = new URL(argv.url);
let dirName = urlObj.host.replace('www.','');
```

We've used `let` here, which unlike `const` can be reassigned, as we'll need to update the reference if the URL has a pathname, to replace slashes with underscores. This can be done with a regular expression pattern, and looks like this:

JavaScript
```javascript
const urlObj = new URL(argv.url);
let dirName = urlObj.host.replace("www.", "");
if (urlObj.pathname !== "/") {
  dirName = dirName + urlObj.pathname.replace(/\//g, "_");
}
```

Now we can create the directory itself. This can be done through the use of another native Node.js module called fs (https://nodejs.org/api/fs.html) (short for "file system").

JavaScript
```javascript
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
const argv = require('yargs').argv;
const url = require('url');
const fs = require('fs');
```

We can use its `mkdir()` method to create a directory, but first have to use its `existsSync()` method to check if the directory already exists, as Node.js would otherwise throw an error:

JavaScript
```javascript
const urlObj = new URL(argv.url);
let dirName = urlObj.host.replace("www.", "");
if (urlObj.pathname !== "/") {
  dirName = dirName + urlObj.pathname.replace(/\//g, "_");
}
if (!fs.existsSync(dirName)) {
  fs.mkdirSync(dirName);
}
```

Testing the script at the point should result in a new directory being created. Passing `https://www.bbc.co.uk/news` as the URL argument would result in a directory named `bbc.co.uk_news`.

(#aa-saving-the-report) Saving the report

In the then function for `launchChromeAndRunLighthouse()`, we want to replace the existing `console.log` with logic to write the report to disk. This can be done using the fs module's `writeFile()` method.

JavaScript
```
launchChromeAndRunLighthouse(argv.url).then(results => {
  fs.writeFile("report.json", results, err => {
    if (err) throw err;
  });
});
```

The first parameter represents the file name, the second is the content of the file and the third is a callback containing an error object should something go wrong during the write process. This would create a new file called `report.json` containing the returning Lighthouse report JSON object.

We still need to send it to the correct directory, with a timestamp as its file name. The former is simple — we pass the `dirName` variable we created earlier, like so:

JavaScript
```
launchChromeAndRunLighthouse(argv.url).then(results => {
  fs.writeFile(`${dirName}/report.json`, results, err => {
    if (err) throw err;
  });
});
```

The latter though requires us to somehow retrieve a timestamp of when the report was generated. Thankfully, the report itself captures this as a data point, and is stored as the `fetchTime` property.

We just need to remember to swap any colons (`:`) for underscores (`_`) so it plays nice with the Windows file system.

JavaScript
```
launchChromeAndRunLighthouse(argv.url).then(results => {
  fs.writeFile(
    `${dirName}/${results["fetchTime"].replace(/:/g, "_")}.json`,
    results,
    err => {
      if (err) throw err;
    }
  );
});
```

If you were to run this now, rather than a `timestamped.json` filename, instead you would likely see an error similar to:

Console
```
UnhandledPromiseRejectionWarning: TypeError: Cannot read property 'replace' of undefined
```

This is happening because Lighthouse is currently returning the report in JSON format, rather than an object consumable by JavaScript.

Thankfully, instead of parsing the JSON ourselves, we can just ask Lighthouse to return the report as a regular JavaScript object instead.

This requires editing the below line from:

JavaScript
```
return chrome.kill().then(() => results.report);
```

...to:

JavaScript
```
return chrome.kill().then(() => results.lhr);
```

Now, if you rerun the script, the file will be named correctly. However, when opened, it's only content will unfortunately be...

```
[object Object]
```

This is because we've now got the opposite problem as before. We're trying to render a JavaScript object without stringifying it into a JSON object first.

The solution is simple. To avoid having to waste resources on parsing or stringifying this huge object, we can return *both* types from Lighthouse:

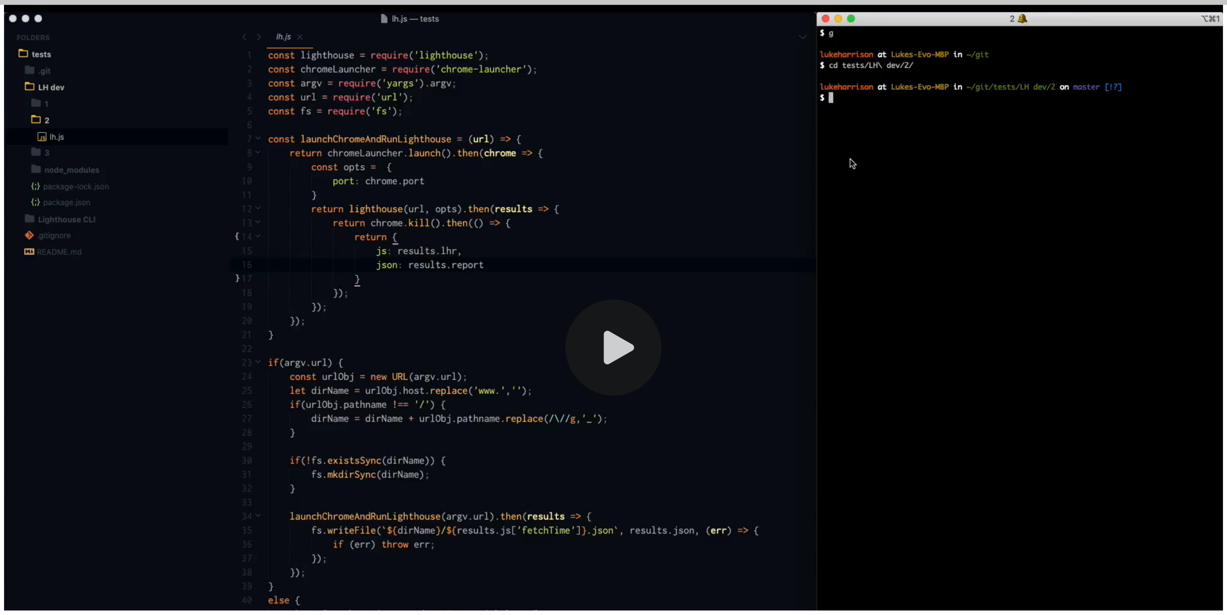
JavaScript
```javascript
return lighthouse(url, opts).then(results => {
  return chrome.kill().then(() => {
    return {
      js: results.lhr,
      json: results.report
    };
  });
});
```

Then we can modify the `writeFile` instance to this:

JavaScript
```javascript
fs.writeFile(
  `${dirName}/${results.js["fetchTime"].replace(/:/g, "_")}.json`,
  results.json,
  err => {
    if (err) throw err;
  }
);
```

Sorted! On completion of the Lighthouse audit, our tool should now save the report to a file with a unique timestamped filename in a directory named after the website URL.

This means reports are now much more efficiently organized and won't override each other no matter how many reports are saved.



```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
const argv = require('yargs').argv;
const url = require('url');
const fs = require('fs');

const launchChromeAndRunLighthouse = (url) => {
    return chromeLauncher.launch().then(chrome => {
        const opts =  {
            port: chrome.port
        }
        return lighthouse(url, opts).then(results => {
            return chrome.kill().then(() => {
                return {
                    js: results.lhr,
                    json: results.report
                }
            });
        });
    });
}

if(argv.url) {
    const urlObj = new URL(argv.url);
    let dirName = urlObj.host.replace('www.','');
    if(urlObj.pathname !== '/') {
        dirName = dirName + urlObj.pathname.replace(/\//g,'_');
    }

    if(!fs.existsSync(dirName)) {
        fs.mkdirSync(dirName);
    }

    launchChromeAndRunLighthouse(argv.url).then(results => {
        fs.writeFile(`${dirName}/${results.js['fetchTime']}.json`, results.json, (err) => {
            if (err) throw err;
        });
    });
}
else {
```

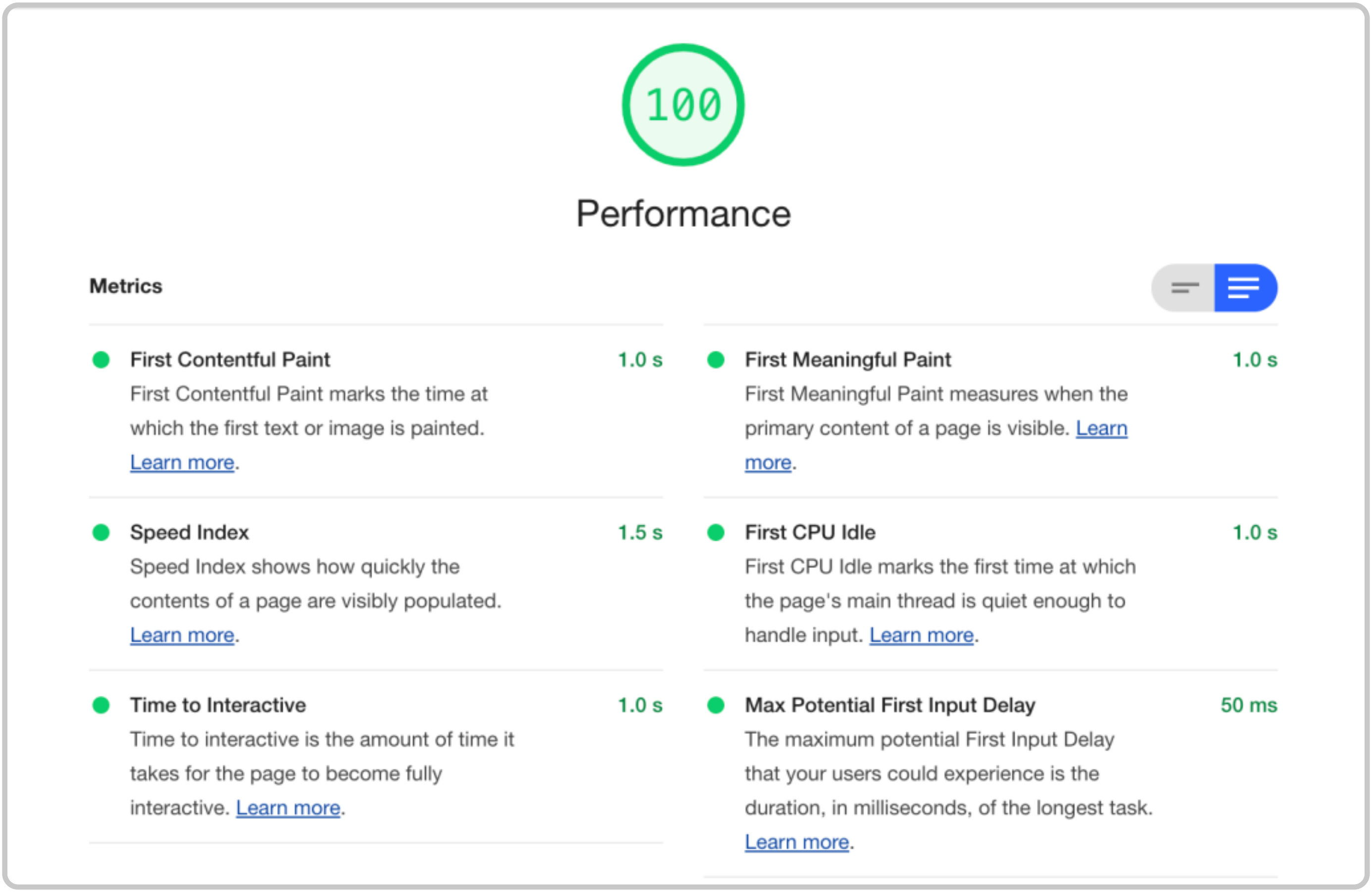
```
$ g

lukeharrison at Lukes-Evo-MBP in ~/git
$ cd tests/LH\ dev/2/

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev/2 on master [!?]
$
```

(#aa-comparing-lighthouse-reports) Comparing Lighthouse reports

During everyday development, when I'm focused on improving performance, the ability to very quickly compare reports directly in the console and see if I'm headed in the right direction could be extremely useful. With this in mind, the requirements of this compare functionality ought to be:

1. If a previous report already exists for the same website when a Lighthouse audit is complete, automatically perform a comparison against it and show any changes to key performance metrics.
2. I should also be able to compare key performance metrics from any two reports, from any two websites, without having to generate a new Lighthouse report which I may not need.

What parts of a report should be compared? These are the numerical key performance metrics collected as part of any Lighthouse report. They provide insight into the objective and perceived performance of a website.



100

Performance

Metrics

● First Contentful Paint 1.0 s
First Contentful Paint marks the time at which the first text or image is painted. Learn more.

● First Meaningful Paint 1.0 s
First Meaningful Paint measures when the primary content of a page is visible. Learn more.

● Speed Index 1.5 s
Speed Index shows how quickly the contents of a page are visibly populated. Learn more.

● First CPU Idle 1.0 s
First CPU Idle marks the first time at which the page's main thread is quiet enough to handle input. Learn more.

● Time to Interactive 1.0 s
Time to interactive is the amount of time it takes for the page to become fully interactive. Learn more.

● Max Potential First Input Delay 50 ms
The maximum potential First Input Delay that your users could experience is the duration, in milliseconds, of the longest task. Learn more.

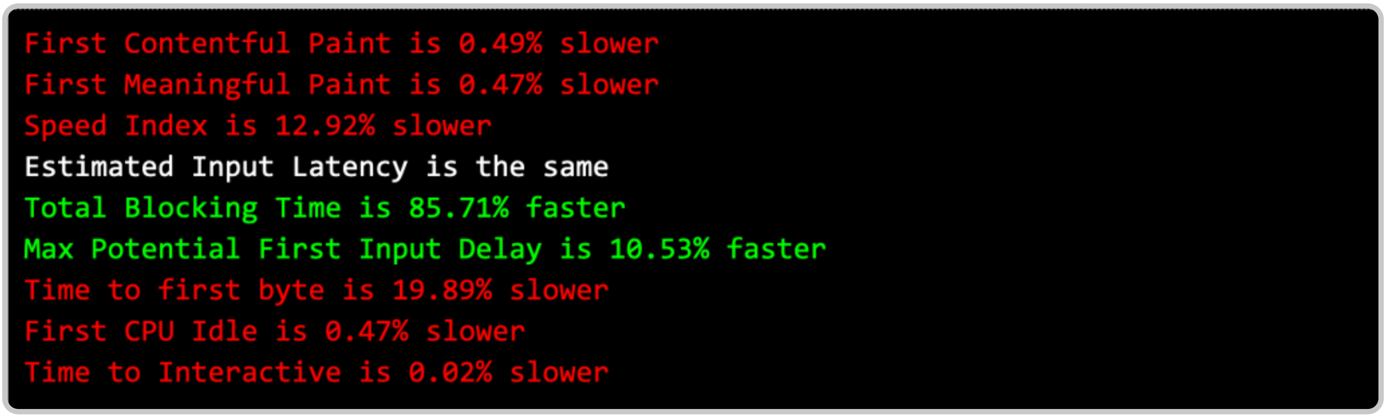
In addition, Lighthouse also collects other metrics that aren't listed in this part of the report but are still in an appropriate format to be included in the comparison. These are:

- **Time to first byte –** Time To First Byte identifies the time at which your server sends a response.
- **Total blocking time –** Sum of all time periods between FCP and Time to Interactive, when task length exceeded 50ms, expressed in milliseconds.
- **Estimated input latency –** Estimated Input Latency is an estimate of how long your app takes to respond to user input, in milliseconds, during the busiest 5s window of page load. If your latency is higher than 50ms, users may perceive your app as laggy.

How should the metric comparison be output to the console? We'll create a simple percentage-based comparison using the old and new metrics to see how they've changed from report to report.

To allow for quick scanning, we'll also color-code individual metrics depending on if they're faster, slower or unchanged.

We'll aim for this output:



```
First Contentful Paint is 0.49% slower
First Meaningful Paint is 0.47% slower
Speed Index is 12.92% slower
Estimated Input Latency is the same
Total Blocking Time is 85.71% faster
Max Potential First Input Delay is 10.53% faster
Time to first byte is 19.89% slower
First CPU Idle is 0.47% slower
Time to Interactive is 0.02% slower
```

(#aa-compare-the-new-report-against-the-previous-report) Compare the new report against the previous report

Let's get started by creating a new function called `compareReports()` just below our `launchChromeAndRunLighthouse()` function, which will contain all the comparison logic. We'll give it two parameters —`from` and `to` — to accept the two reports used for the comparison.

For now, as a placeholder, we'll just print out some data from each report to the console to validate that it's receiving them correctly.

JavaScript

```
const compareReports = (from, to) => {
  console.log(from["finalUrl"] + " " + from["fetchTime"]);
  console.log(to["finalUrl"] + " " + to["fetchTime"]);
};
```

As this comparison would begin after the creation of a new report, the logic to execute this function should sit in the `then` function for `launchChromeAndRunLighthouse()`.

If, for example, you have 30 reports sitting in a directory, we need to determine which one is the most recent and set it as the previous report which the new one will be compared against.

Thankfully, we already decided to use a timestamp as the filename for a report, so this gives us something to work with.

First off, we need to collect any existing reports. To make this process easy, we'll install a new dependency called glob, which allows for pattern matching when searching for files. This is critical because we can't predict how many reports will exist or what they'll be called.

Install it like any other dependency:

Command Line

```
npm install glob --save-dev
```

Then import it at the top of the file the same way as usual:

JavaScript

```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
const argv = require('yargs').argv;
const url = require('url');
const fs = require('fs');
const glob = require('glob');
```

We'll use `glob` to collect all of the reports in the directory, which we already know the name of via the `dirName` variable. It's important to set its `sync` option to `true` as we don't want JavaScript execution to continue until we know how many other reports exist.

JavaScript

```
launchChromeAndRunLighthouse(argv.url).then(results => {
  const prevReports = glob(`${dirName}/*.json`, {
    sync: true
  });

  // et al

});
```

This process returns an array of paths. So if the report directory looked like this:

- lukeharrison.dev
    - 2020-01-31T10_18_12.648Z.json

- 2020-01-31T10_18_24.110Z.json

...then the resulting array would look like this:

```
JavaScript
[
 'lukeharrison.dev/2020-01-31T10_18_12.648Z.json',
 'lukeharrison.dev/2020-01-31T10_18_24.110Z.json'
]
```

Because we can only perform a comparison if a previous report exists, let’s use this array as a conditional for the comparison logic:

```
JavaScript
const prevReports = glob(`${dirName}/*.json`, {
  sync: true
});

if (prevReports.length) {
}
```

We have a list of report file paths and we need to compare their timestamped filenames to determine which one is the most recent.

This means we first need to collect a list of all the file names, trim any irrelevant data such as directory names, and taking care to replace the underscores (_) back with colons (:) to turn them back into valid dates again. The easiest way to do this is using `path (https://nodejs.org/api/path.html)` , another Node.js native module.

```
JavaScript
const path = require('path');
```

Passing the path as an argument to its `parse` method, like so:

```
JavaScript
path.parse('lukeharrison.dev/2020-01-31T10_18_24.110Z.json');
```

Returns this useful object:

JavaScript
```
{
  root: '',
  dir: 'lukeharrison.dev',
  base: '2020-01-31T10_18_24.110Z.json',
  ext: '.json',
  name: '2020-01-31T10_18_24.110Z'
}
```

Therefore, to get a list of all the timestamp file names, we can do this:

JavaScript
```
if (prevReports.length) {
  dates = [];
  for (report in prevReports) {
    dates.push(
      new Date(path.parse(prevReports[report]).name.replace(/_/g, ":"))
    );
  }
}
```

Which again if our directory looked like:

- lukeharrison.dev
  - 2020-01-31T10_18_12.648Z.json
  - 2020-01-31T10_18_24.110Z.json

Would result in:

JavaScript
```
[
 '2020-01-31T10:18:12.648Z',
 '2020-01-31T10:18:24.110Z'
]
```

A useful thing about dates is that they're inherently comparable by default:

JavaScript
```
const alpha = new Date('2020-01-31');
const bravo = new Date('2020-02-15');

console.log(alpha > bravo); // false
console.log(bravo > alpha); // true
```

So by using a `reduce` function, we can reduce our array of dates down until only the most recent remains:

```javascript
dates = [];
for (report in prevReports) {
  dates.push(new Date(path.parse(prevReports[report]).name.replace(/_/g, ":")));
}
const max = dates.reduce(function(a, b) {
  return Math.max(a, b);
});
```

If you were to print the contents of `max` to the console, it would throw up a UNIX timestamp, so now, we just have to add another line to convert our most recent date back into the correct ISO format:

```javascript
const max = dates.reduce(function(a, b) {
 return Math.max(a, b);
});
const recentReport = new Date(max).toISOString();
```

Assuming these are the list of reports:

- 2020-01-31T23_24_41.786Z.json
- 2020-01-31T23_25_36.827Z.json
- 2020-01-31T23_37_56.856Z.json
- 2020-01-31T23_39_20.459Z.json
- 2020-01-31T23_56_50.959Z.json

The value of `recentReport` would be `2020-01-31T23:56:50.959Z`.

Now that we know the most recent report, we next need to extract its contents. Create a new variable called `recentReportContents` beneath the `recentReport` variable and assign it an empty function.

As we know this function will always need to execute, rather than manually calling it, it makes sense to turn it into an IFFE (Immediately invoked function expression), which will run by itself when the JavaScript parser reaches it. This is signified by the extra parenthesis:

```javascript
const recentReportContents = (() => {

})();
```

In this function, we can return the contents of the most recent report using the `readFileSync()` method of the native `fs` module. Because this will be in JSON format, it's important to parse it into a regular JavaScript object.

```javascript
const recentReportContents = (() => {
  const output = fs.readFileSync(
    dirName + "/" + recentReport.replace(/:/g, "_") + ".json",
    "utf8",
    (err, results) => {
      return results;
    }
  );
  return JSON.parse(output);
})();
```

And then, it's a matter of calling the `compareReports()` function and passing both the current report and the most recent report as arguments.

```javascript
compareReports(recentReportContents, results.js);
```

At the moment this just print out a few details to the console so we can test the report data is coming through OK:

```
https://www.lukeharrison.dev/ 2020-02-01T00:25:06.918Z
https://www.lukeharrison.dev/ 2020-02-01T00:25:42.169Z
```

If you're getting any errors at this point, try deleting any `report.json` files or reports without valid content from earlier in the tutorial.

(#aa-compare-any-two-reports) Compare any two

reports

The remaining key requirement was the ability to compare any two reports from any two websites. The easiest way to implement this would be to allow the user to pass the full report file paths as command line arguments which we'll then send to the `compareReports()` function.

In the command line, this would look like:

Terminal

```
node lh.js --from lukeharrison.dev/2020-02-01T00:25:06.918Z --to cnn.com/2019-12-16T15:12:07.169Z
```

Achieving this requires editing the conditional `if` statement which checks for the presence of a URL command line argument. We'll add an additional check to see if the user has just passed a `from` and `to` path, otherwise check for the URL as before. This way we'll prevent a new Lighthouse audit.

JavaScript

```
if (argv.from && argv.to) {

} else if (argv.url) {
 // et al
}
```

Let's extract the contents of these JSON files, parse them into JavaScript objects, and then pass them along to the `compareReports()` function.

We've already parsed JSON before when retrieving the most recent report. We can just extrapolate this functionality into its own helper function and use it in both locations.

Using the `recentReportContents()` function as a base, create a new function called `getContents()` which accepts a file path as an argument. Make sure this is just a regular function, rather than an IFFE, as we don't want it executing as soon as the JavaScript parser finds it.

JavaScript

```
const getContents = pathStr => {
  const output = fs.readFileSync(pathStr, "utf8", (err, results) => {
    return results;
  });
  return JSON.parse(output);
};

const compareReports = (from, to) => {
  console.log(from["finalUrl"] + " " + from["fetchTime"]);
  console.log(to["finalUrl"] + " " + to["fetchTime"]);
};
```

Then update the `recentReportContents()` function to use this extrapolated helper function instead:

JavaScript
```
const recentReportContents = getContents(dirName + '/' + recentReport.replace(/:/g, '_') + '.json');
```

Back in our new conditional, we need to pass the contents of the comparison reports to the `compareReports()` function.

JavaScript
```
if (argv.from && argv.to) {
  compareReports(
    getContents(argv.from + ".json"),
    getContents(argv.to + ".json")
  );
}
```

Like before, this should print out some basic information about the reports in the console to let us know it's all working fine.

Terminal
```
node lh.js --from lukeharrison.dev/2020-01-31T23_24_41.786Z --to lukeharrison.dev/2020-02-01T11_16_25.221Z
```

Would lead to:

```
https://www.lukeharrison.dev/ 2020-01-31T23_24_41.786Z
https://www.lukeharrison.dev/ 2020-02-01T11_16_25.221Z
```

(#aa-comparison-logic) Comparison logic

This part of development involves building comparison logic to compare the two reports received by the `compareReports()` function.

Within the object which Lighthouse returns, there's a property called `audits` that contains another object listing performance metrics, opportunities, and information. There's a lot of information here, much of which we aren't interested in for the purposes of this tool.

Here's the entry for First Contentful Paint, one of the nine performance metrics we wish to compare:

JSON

```json
"first-contentful-paint": {
  "id": "first-contentful-paint",
  "title": "First Contentful Paint",
  "description": "First Contentful Paint marks the time at which the first text or image is painted. [Learn more
  "score": 1,
  "scoreDisplayMode": "numeric",
  "numericValue": 1081.661,
  "displayValue": "1.1 s"
}
```

Create an array listing the keys of these nine performance metrics. We can use this to filter the audit object:

JavaScript

```javascript
const compareReports = (from, to) => {
  const metricFilter = [
    "first-contentful-paint",
    "first-meaningful-paint",
    "speed-index",
    "estimated-input-latency",
    "total-blocking-time",
    "max-potential-fid",
    "time-to-first-byte",
    "first-cpu-idle",
    "interactive"
  ];
};
```

Then we'll loop through one of the report's `audits` object and then cross-reference its name against our filter list. (It doesn't matter which audit object, as they both have the same content structure.)

If it’s in there, then brilliant, we want to use it.

JavaScript

```
const metricFilter = [
  "first-contentful-paint",
  "first-meaningful-paint",
  "speed-index",
  "estimated-input-latency",
  "total-blocking-time",
  "max-potential-fid",
  "time-to-first-byte",
  "first-cpu-idle",
  "interactive"
];

for (let auditObj in from["audits"]) {
  if (metricFilter.includes(auditObj)) {
    console.log(auditObj);
  }
}
```

This `console.log()` would print the below keys to the console:

```
first-contentful-paint
first-meaningful-paint
speed-index
estimated-input-latency
total-blocking-time
max-potential-fid
time-to-first-byte
first-cpu-idle
interactive
```

Which means we would use `from['audits'][auditObj].numericValue` and `to['audits'][auditObj].numericValue` respectively in this loop to access the metrics themselves.

If we were to print these to the console with the key, it would result in output like this:

```
first-contentful-paint 1081.661 890.774
first-meaningful-paint 1081.661 954.774
speed-index 15576.70313351777 1098.622294504341
```

```
estimated-input-latency 12.8 12.8
total-blocking-time 59 31.5
max-potential-fid 153 102
time-to-first-byte 16.859999999999985 16.096000000000004
first-cpu-idle 1704.8490000000002 1918.774
interactive 2266.2835 2374.3615
```

We have all the data we need now. We just need to calculate the percentage difference between these two values and then log it to the console using the color-coded format outlined earlier.

Do you know how to calculate the percentage change between two values? Me neither. Thankfully, everybody's favorite monolith search engine came to the rescue.

The formula is:

```
((From - To) / From) x 100
```

So, let's say we have a Speed Index of 5.7s for the first report (from), and then a value of 2.1s for the second (to). The calculation would be:

```html
5.7 - 2.1 = 3.6
3.6 / 5.7 = 0.63157895
0.63157895 * 100 = 63.157895
```

Rounding to two decimal places would yield a decrease in the speed index of 63.16%.

Let's put this into a helper function inside the `compareReports()` function, below the `metricFilter` array.

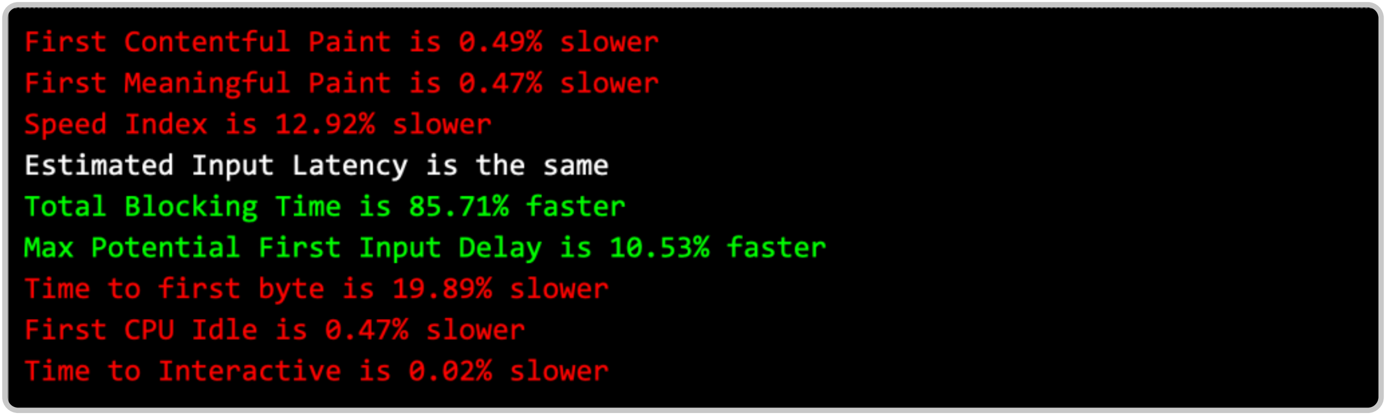
```javascript
const calcPercentageDiff = (from, to) => {
  const per = ((to - from) / from) * 100;
  return Math.round(per * 100) / 100;
};
```

Back in our `auditObj` conditional, we can begin to put together the final report comparison output.

First off, use the helper function to generate the percentage difference for each metric.

```javascript
for (let auditObj in from["audits"]) {
  if (metricFilter.includes(auditObj)) {
    const percentageDiff = calcPercentageDiff(
      from["audits"][auditObj].numericValue,
      to["audits"][auditObj].numericValue
    );
  }
}
```

Next, we need to output values in this format to the console:



```
First Contentful Paint is 0.49% slower
First Meaningful Paint is 0.47% slower
Speed Index is 12.92% slower
Estimated Input Latency is the same
Total Blocking Time is 85.71% faster
Max Potential First Input Delay is 10.53% faster
Time to first byte is 19.89% slower
First CPU Idle is 0.47% slower
Time to Interactive is 0.02% slower
```

This requires adding color to the console output. In Node.js, this can be done by passing a color code as an argument (https://css-tricks.com/a-guide-to-console-commands/#article-header-id-4) to the `console.log()` function like so:

```javascript
console.log('\x1b[36m', 'hello') // Would print 'hello' in cyan
```

```
hello
```

You can get a full reference of color codes (https://stackoverflow.com/questions/9781218/how-to-change-node-jss-console-font-color) in this Stackoverflow question. We need green and red, so that's `\x1b[32m` and `\x1b[31m` respectively. For metrics where the value remains unchanged, we'll just use white. This would be `\x1b[37m`.

Depending on if the percentage increase is a positive or negative number, the following things need to happen:

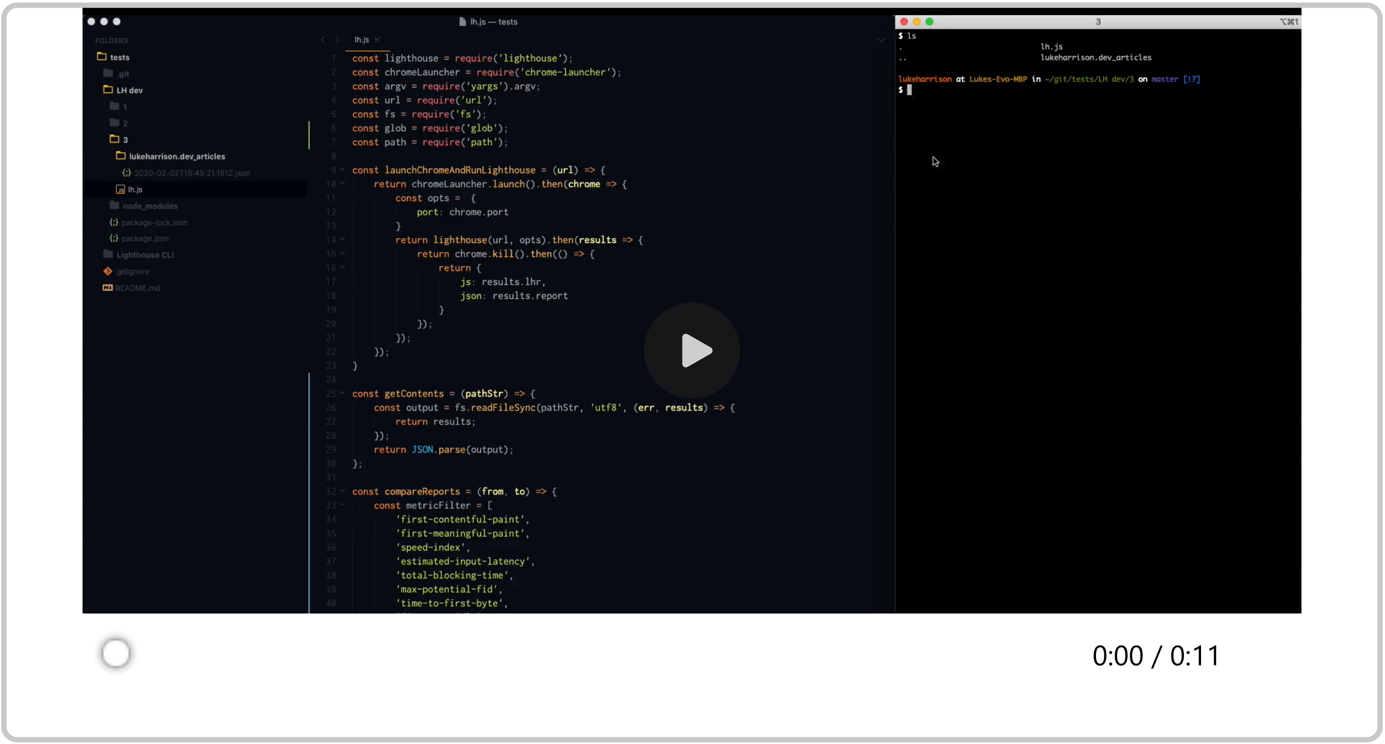
- Log color needs to change (Green for negative, red for positive, white for unchanged)
- Log text contents change.
  - '[Name] is X% slower for positive numbers
  - '[Name] is X% faster' for negative numbers
  - '[Name] is unchanged' for numbers with no percentage difference.
- If the number is negative, we want to remove the minus/negative symbol, as otherwise, you'd have a sentence like *'Speed Index is -92.95% faster'* which doesn't make sense.

There are many ways this could be done. Here, we'll use the `Math.sign()` function, which returns 1 if its argument is positive, 0 if well… 0, and -1 if the number is negative. That'll do.

JavaScript
```javascript
for (let auditObj in from["audits"]) {
  if (metricFilter.includes(auditObj)) {
    const percentageDiff = calcPercentageDiff(
      from["audits"][auditObj].numericValue,
      to["audits"][auditObj].numericValue
    );

    let logColor = "\x1b[37m";
    const log = (() => {
      if (Math.sign(percentageDiff) === 1) {
        logColor = "\x1b[31m";
        return `${percentageDiff + "%"} slower`;
      } else if (Math.sign(percentageDiff) === 0) {
        return "unchanged";
      } else {
        logColor = "\x1b[32m";
        return `${percentageDiff + "%"} faster`;
      }
    })();
    console.log(logColor, `${from["audits"][auditObj].title} is ${log}`);
  }
}
```

So, there we have it.

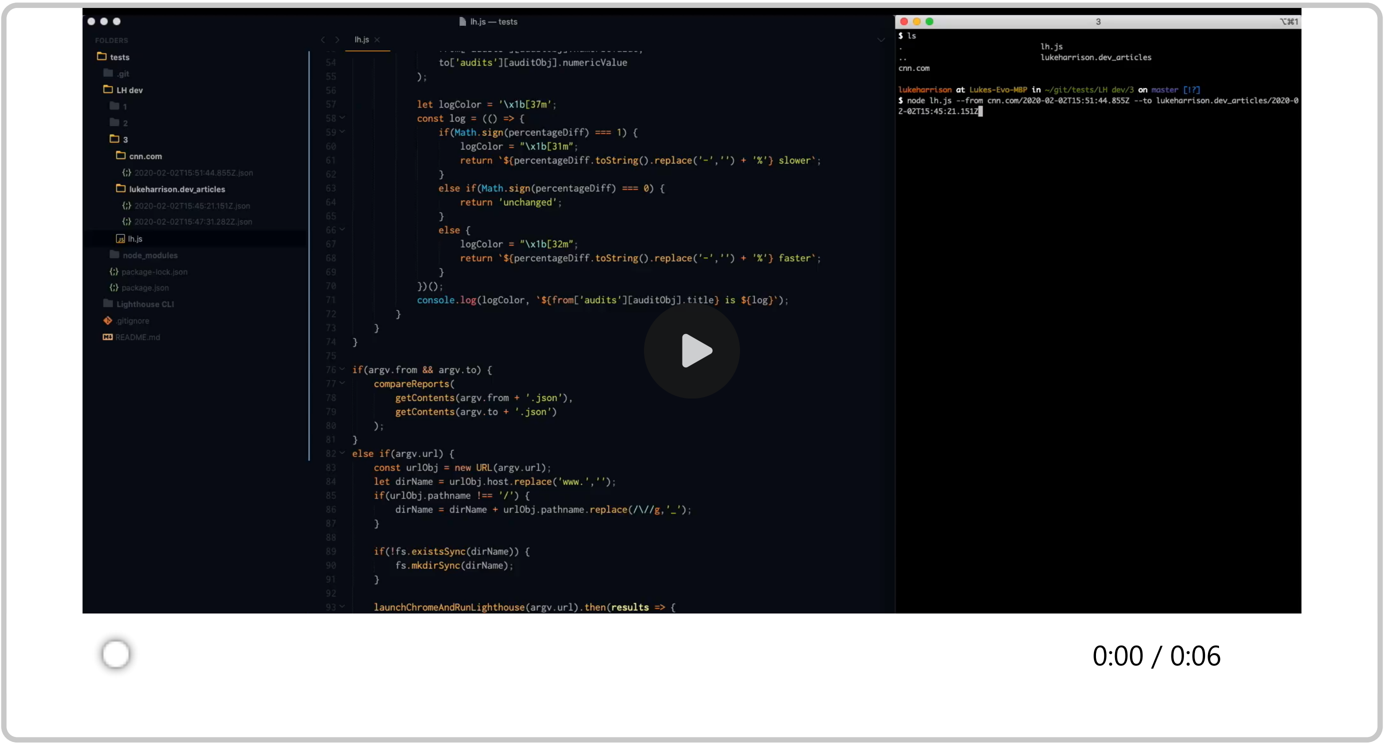
You can create new Lighthouse reports, and if a previous one exists, a comparison is made.



```
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');
const argv = require('yargs').argv;
const url = require('url');
const fs = require('fs');
const glob = require('glob');
const path = require('path');

const launchChromeAndRunLighthouse = (url) => {
    return chromeLauncher.launch().then(chrome => {
        const opts =  {
            port: chrome.port
        }
        return lighthouse(url, opts).then(results => {
            return chrome.kill().then(() => {
                return {
                    js: results.lhr,
                    json: results.report
                }
            });
        });
    });
}

const getContents = (pathStr) => {
    const output = fs.readFileSync(pathStr, 'utf8', (err, results) => {
        return results;
    });
    return JSON.parse(output);
};

const compareReports = (from, to) => {
    const metricFilter = [
        'first-contentful-paint',
        'first-meaningful-paint',
        'speed-index',
        'estimated-input-latency',
        'total-blocking-time',
        'max-potential-fid',
        'time-to-first-byte',
```

```
$ ls
.                               lh.js
..                              lukeharrison.dev_articles

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev/3 on master [!?]
$ 
```

And you can also compare any two reports from any two sites.



```
                to['audits'][auditObj].numericValue
            );

            let logColor = '\x1b[37m';
            const log = (() => {
                if(Math.sign(percentageDiff) === 1) {
                    logColor = "\x1b[31m";
                    return `${percentageDiff.toString().replace('-','') + '%'} slower`;
                }
                else if(Math.sign(percentageDiff) === 0) {
                    return 'unchanged';
                }
                else {
                    logColor = "\x1b[32m";
                    return `${percentageDiff.toString().replace('-','') + '%'} faster`;
                }
            })();
            console.log(logColor, `${from['audits'][auditObj].title} is ${log}`);
        }
    }
}

if(argv.from && argv.to) {
    compareReports(
        getContents(argv.from + '.json'),
        getContents(argv.to + '.json')
    );
}
else if(argv.url) {
    const urlObj = new URL(argv.url);
    let dirName = urlObj.host.replace('www.','');
    if(urlObj.pathname !== '/') {
        dirName = dirName + urlObj.pathname.replace(/\//g,'_');
    }

    if(!fs.existsSync(dirName)) {
        fs.mkdirSync(dirName);
    }

    launchChromeAndRunLighthouse(argv.url).then(results => {
```

```
$ ls
.                               lh.js
..                              lukeharrison.dev_articles
cnn.com

lukeharrison at Lukes-Evo-MBP in ~/git/tests/LH dev/3 on master [!?]
$ node lh.js --from cnn.com/2020-02-02T15:51:44.855Z --to lukeharrison.dev_articles/2020-0
2-02T15:45:21.151Z
```

(#aa-complete-source-code) Complete source code

Here’s the completed source code for the tool, which you can also view in a Gist via the link below.

JavaScript

```
const lighthouse = require("lighthouse");
const chromeLauncher = require("chrome-launcher");
const argv = require("yargs").argv;
const url = require("url");
const fs = require("fs");
const glob = require("glob");
const path = require("path");

const launchChromeAndRunLighthouse = url => {
  return chromeLauncher.launch().then(chrome => {
    const opts = {
      port: chrome.port
    };
    return lighthouse(url, opts).then(results => {
      return chrome.kill().then(() => {
        return {
          js: results.lhr,
          json: results.report
        };
      });
    });
  });
};

const getContents = pathStr => {
  const output = fs.readFileSync(pathStr, "utf8", (err, results) => {
    return results;
  });
  return JSON.parse(output);
};

const compareReports = (from, to) => {
  const metricFilter = [
    "first-contentful-paint",
    "first-meaningful-paint",
    "speed-index",
    "estimated-input-latency",
    "total-blocking-time",
    "max-potential-fid",
    "time-to-first-byte",
    "first-cpu-idle",
    "interactive"
  ];
```

```
    const calcPercentageDiff = (from, to) => {
      const per = ((to - from) / from) * 100;
      return Math.round(per * 100) / 100;
    };

    for (let auditObj in from["audits"]) {
      if (metricFilter.includes(auditObj)) {
        const percentageDiff = calcPercentageDiff(
          from["audits"][auditObj].numericValue,
          to["audits"][auditObj].numericValue
        );

        let logColor = "\x1b[37m";
        const log = (() => {
          if (Math.sign(percentageDiff) === 1) {
            logColor = "\x1b[31m";
            return `${percentageDiff.toString().replace("-", "") + "%"} slower`;
          } else if (Math.sign(percentageDiff) === 0) {
            return "unchanged";
          } else {
            logColor = "\x1b[32m";
            return `${percentageDiff.toString().replace("-", "") + "%"} faster`;
          }
        })();
        console.log(logColor, `${from["audits"][auditObj].title} is ${log}`);
      }
    }
  };

  if (argv.from && argv.to) {
    compareReports(
      getContents(argv.from + ".json"),
      getContents(argv.to + ".json")
    );
  } else if (argv.url) {
    const urlObj = new URL(argv.url);
    let dirName = urlObj.host.replace("www.", "");
    if (urlObj.pathname !== "/") {
      dirName = dirName + urlObj.pathname.replace(/\//g, "_");
    }

    if (!fs.existsSync(dirName)) {
      fs.mkdirSync(dirName);
    }

    launchChromeAndRunLighthouse(argv.url).then(results => {
      const prevReports = glob(`${dirName}/*.json`, {
        sync: true
      });
      if (prevReports.length) {
        dates = [];
        for (report in prevReports) {
          dates.push(
            new Date(path.parse(prevReports[report]).name.replace(/_/g, ":"))
          );
        }
        const max = dates.reduce(function(a, b) {
          return Math.max(a, b);
        });
        const recentReport = new Date(max).toISOString();

        const recentReportContents = getContents(
          dirName + "/" + recentReport.replace(/:/g, "_") + ".json"
        );

        compareReports(recentReportContents, results.js);
      }

      fs.writeFile(
        `${dirName}/${results.js["fetchTime"].replace(/:/g, "_")}.json`,
        results.json,
        err => {
          if (err) throw err;
        }
      );
    });
  } else {
    throw "You haven't passed a URL to Lighthouse";
  }
```

(#aa-next-steps) Next steps

With the completion of this basic Google Lighthouse tool, there’s plenty of ways to develop it further. For example:

- Some kind of simple online dashboard that allows non-technical users to run Lighthouse audits and view metrics develop over time. Getting stakeholders behind web performance can be challenging, so something tangible they can interest with themselves could pique their interest.

- Build support for performance budgets, so if a report is generated and performance metrics are slower than they should be, then the tool outputs useful advice on how to improve them (or calls you names).

Good luck!