

I FEEL THE NEED FOR WEB PAGE SPEED



BILL DINGER

Managing Directory, Technology | @adazlian

wdinger@gmail.com

<https://github.com/BillDinger/WebPageSpeed>

THINGS PEOPLE WOULD NEVER SAY

“I LOVE AUTOPLAY VIDEOS.” – AN ACTUAL MORON

“ADS ARE GOOD AND ALSO RELEVANT TO ME.” – PAINT CHIP
CONNOISSEUR

“CHROME USING 4GB OF RAM IS GOOD.” – A SLACK DEVELOPER

“THIS WEBPAGE LOADS JUST TOO FAST” – DARWIN AWARD WINNER

OUR CURRENT GRIMDARK HELLSCAPE OF WEB
PERFORMANCE

DESPITE 5G, POCKET SUPERCOMPUTERS AND BILLIONS IN
BROWSER DEVELOPMENT WEBPAGES STILL LOAD LIKE ITS
2001 ON MY DAD'S ISDN LINE

WHY IS THIS SO SLOW???

“Byte-for-byte, JavaScript is still the most expensive resource we send to mobile phones, because it can delay interactivity in large ways.” – Addy Osmani, Engineering Manager, Chrome.

“My page is perfectly fast if you ignore all this third-party tags and dumb images.” – Me, 2017.



THE IMPROVEMENT PATTERN

1. UNDERSTAND THE ARCHITECTURE

2. MEASURE PERFORMANCE

3. OPTIMIZATION TECHNIQUES

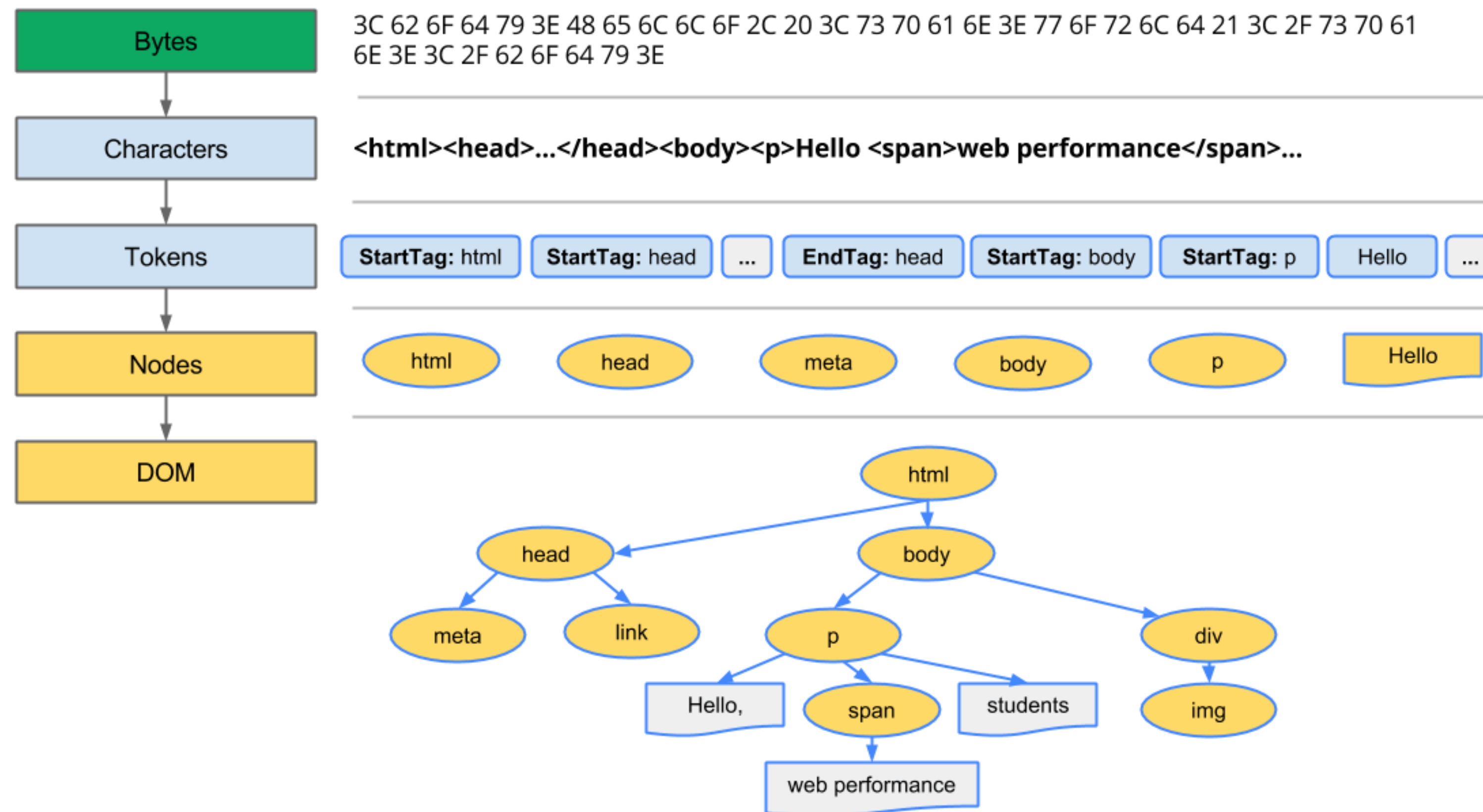
1.) UNDERSTAND THE ARCHITECTURE

The critical rendering path is the minimal set of steps the browser must take to render a webpage.

- 1.) Construct the Object Model.
- 2.) Create a render tree.
- 3.) Layout.
- 4.) Paint.

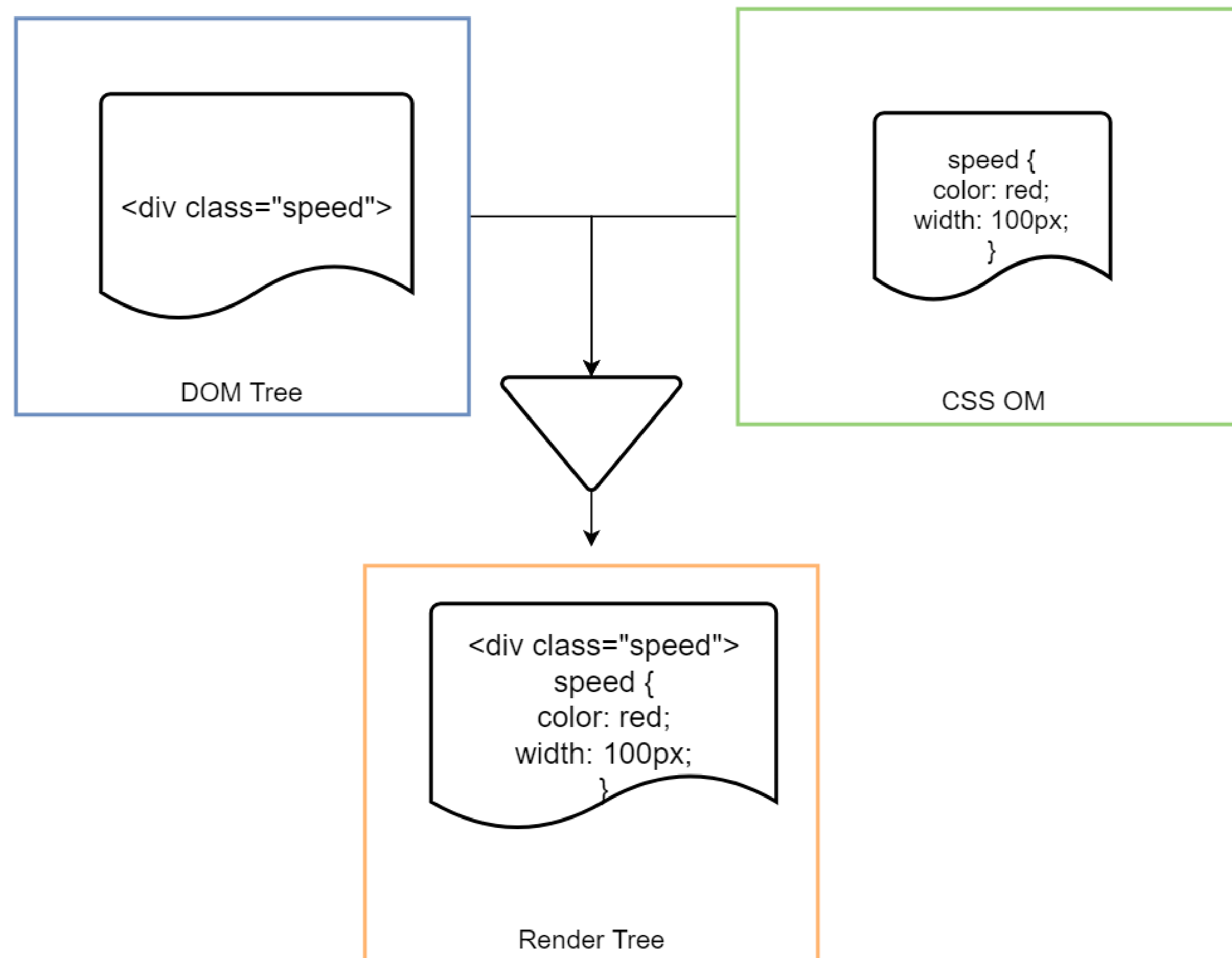
Step #1: Construct the Object Model

Two-part process to fetch and parse HTML (to turn into the DOM) and CSS (to turn into the CSSOM).



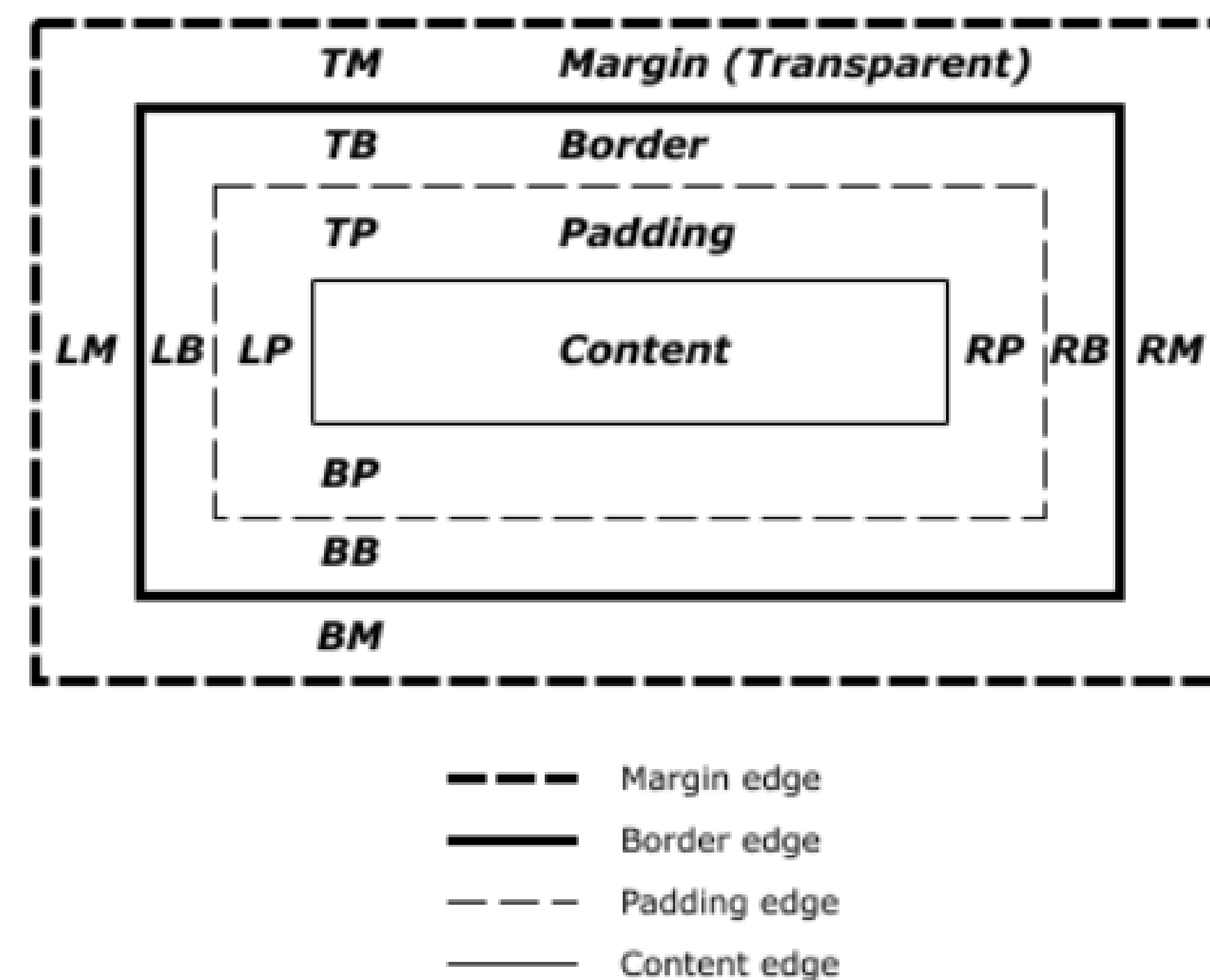
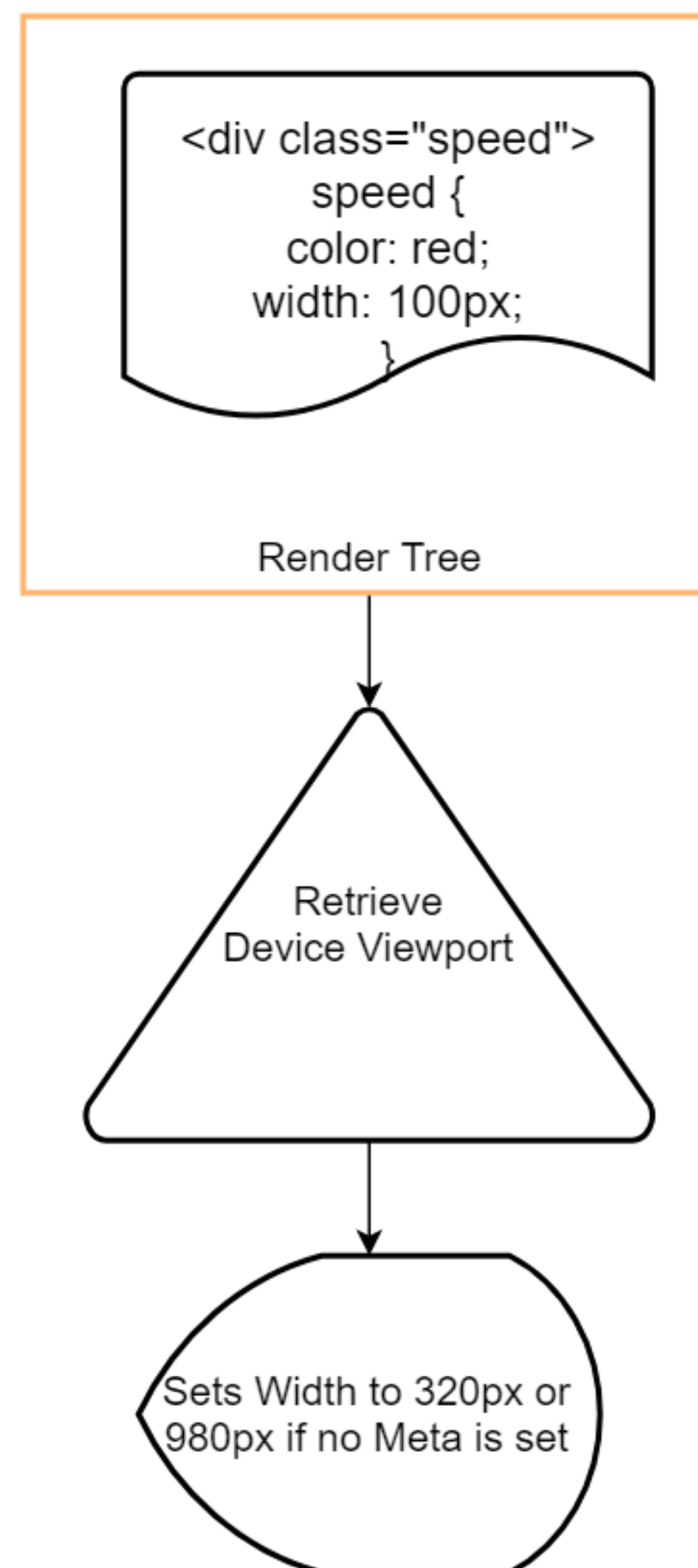
Step #2: Create the Render Tree

Combines the CSSOM & DOM into one tree structure. Only consists of *visible* elements.



Step #3: Layout

Determines size and placement (width/height/float) of elements on the page.



Step #4: Paint

Actually rendering the pixels onto the screen.
In modern browsers, typically will make use of the GPU if present to speed up render.

COOL COOL COOL. WHAT ABOUT
JAVASCRIPT??

Four steps to run JavaScript once it is encountered

- 1.) Download
- 2.) Parse
- 3.) Compile
- 4.) Execute

- 1.) Download – **Network Bound**
- 2.) Parse – **CPU Bound**.
- 3.) Compile – **CPU Bound**.
- 4.) Execute – **CPU Bound**.

JavaScript execution speed is mainly CPU bound!

465.7 ms	0.2 ms	268.3 ms	▼ Evaluate Script
465.7 ms	10.7 ms	10.7 ms	Compile Script
476.6 ms	0.1 ms	257.5 ms	▶ (anonymous)



DEMO: GMAIL



LIKE CSS BY DEFAULT JAVASCRIPT BLOCKS
RENDER UNTIL ALL 4 STEPS HAVE BEEN
COMPLETED!

... Although there are some caveats

JS	Blocks Render?	Network Load Priority
Inline JS, i.e <code><script> Console.log('hello world'); </script></code>	Yes	N/A
<code><script async src="https://www...></code>	No	Medium/High
<code><script defer src="https://www"></code>	No	Lowest
<code><script></code> in head	Yes	Highest
<code><script></code> in body	No	Medium/High

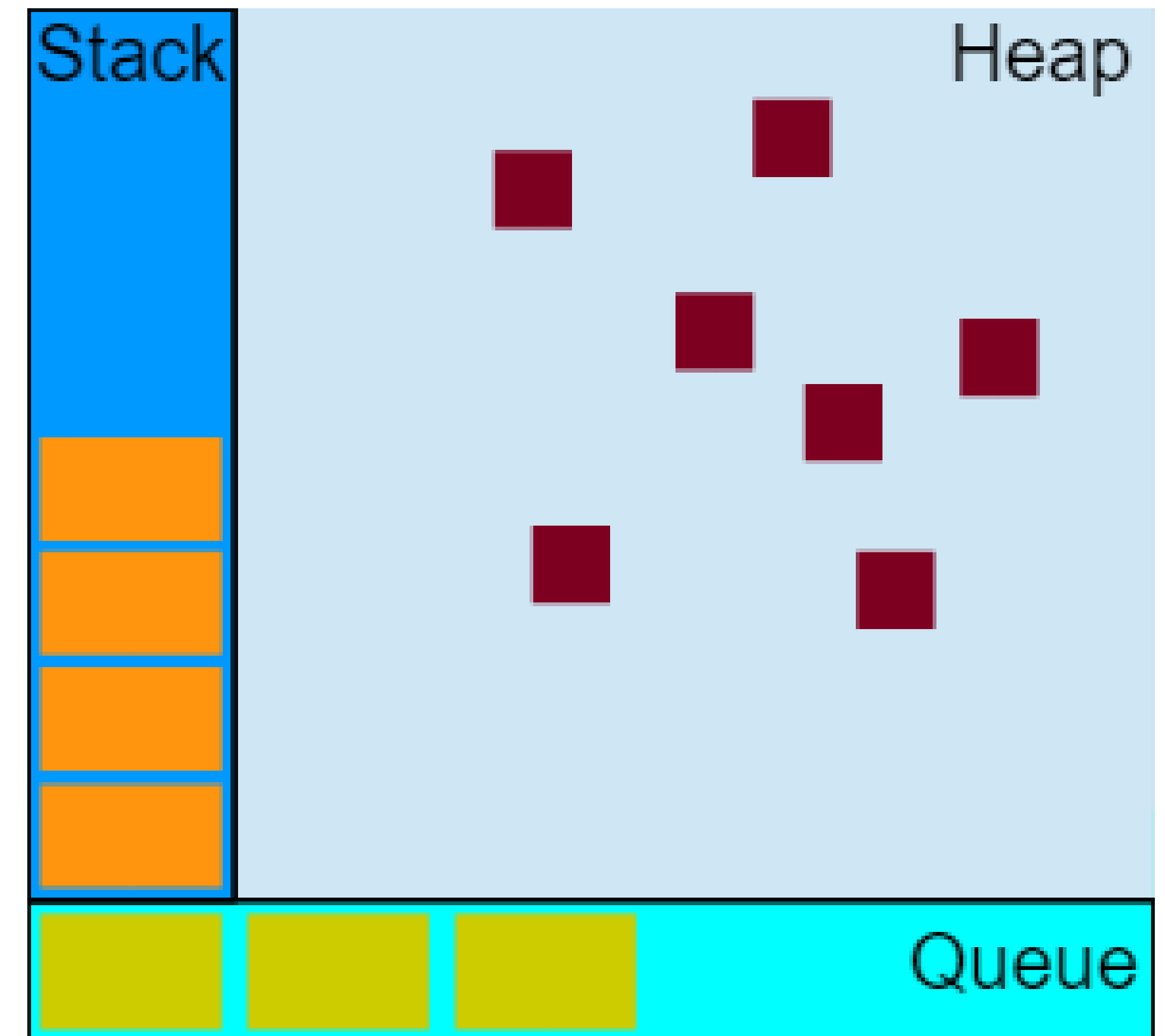
Source: bit.ly/script-scheduling

JavaScript Event Loop and its Friends.

Stack: A stack of Functions and their arguments. When a function is called it is added to the top of the stack when it returns it is popped off the top and next function down is executed.

Heap: Where objects get stored in memory.

Queue: A queue of messages (events, promises, functions) to process



Code:

```
function callingMe(first,second) {  
  console.log(`Calling me with ${first}  
and ${second}`);  
}  
  
function doingThing() {  
  callingMe('hello', 'world');  
}  
  
function startMeUp(param) {  
  console.log(`starting up my program  
for ${param}`);  
  doingThing()  
}  
  
startMeUp('KCDC 2019');
```

Stack:

```
1.)callingMe('first','second')  
2.)doingThing()  
3.)startMeUp('KCDC 2019')
```

The *stack* runs from top to bottom and executes until completion.

Event loop implementation: example

```
class Loop {
  get messages() {
    return this._messages;
  }
  get waitForMessages() {
    return this._waitForMessages();
  }
  set waitForMessages(wait) {
    this._waitForMessages = wait;
  }

  execute() {
    if (Array.isArray(this._messages) && this._messages.length > 0) {
      const funcToExecute = messages.pop();
      funcToExecute();
    } else {
      this.waitForMessages(false);
    }
  }

  loop() {
    if (this.waitForMessages()) {
      this.execute();
    }
  }

  constructor(messages) {
    this._waitForMessages = true;
    this._messages = messages;
  }
}
```

```
const messages = [
  function() {
    start;
  },
  function() {
    startMeUp('KCDC 2019');
  },
  function() {
    doingThing();
  },
  function() {
    callingMe('hello', 'world');
  },
];
const loop = new Loop(messages);
loop.loop();
```

YOU GET TO DO ONE THING AT A TIME...
WHICH IS WHERE THE QUEUE COMES IN

Queue: handles non-blocking code and works with the event loop.

Examples of non-blocking code include things like `setTimeout`, `setInterval`, event handlers, and most I/O related things in JavaScript such as ajax calls.

Technically not part of the JavaScript spec, rather implemented by the various engines (Node, V8, etc) to offload I/O and not block the event loop.

Queue: Quiz Time!

```
const XMLHttpRequest = require('xmlhttprequest').XMLHttpRequest;

try {
  console.log('Starting our program.');
```

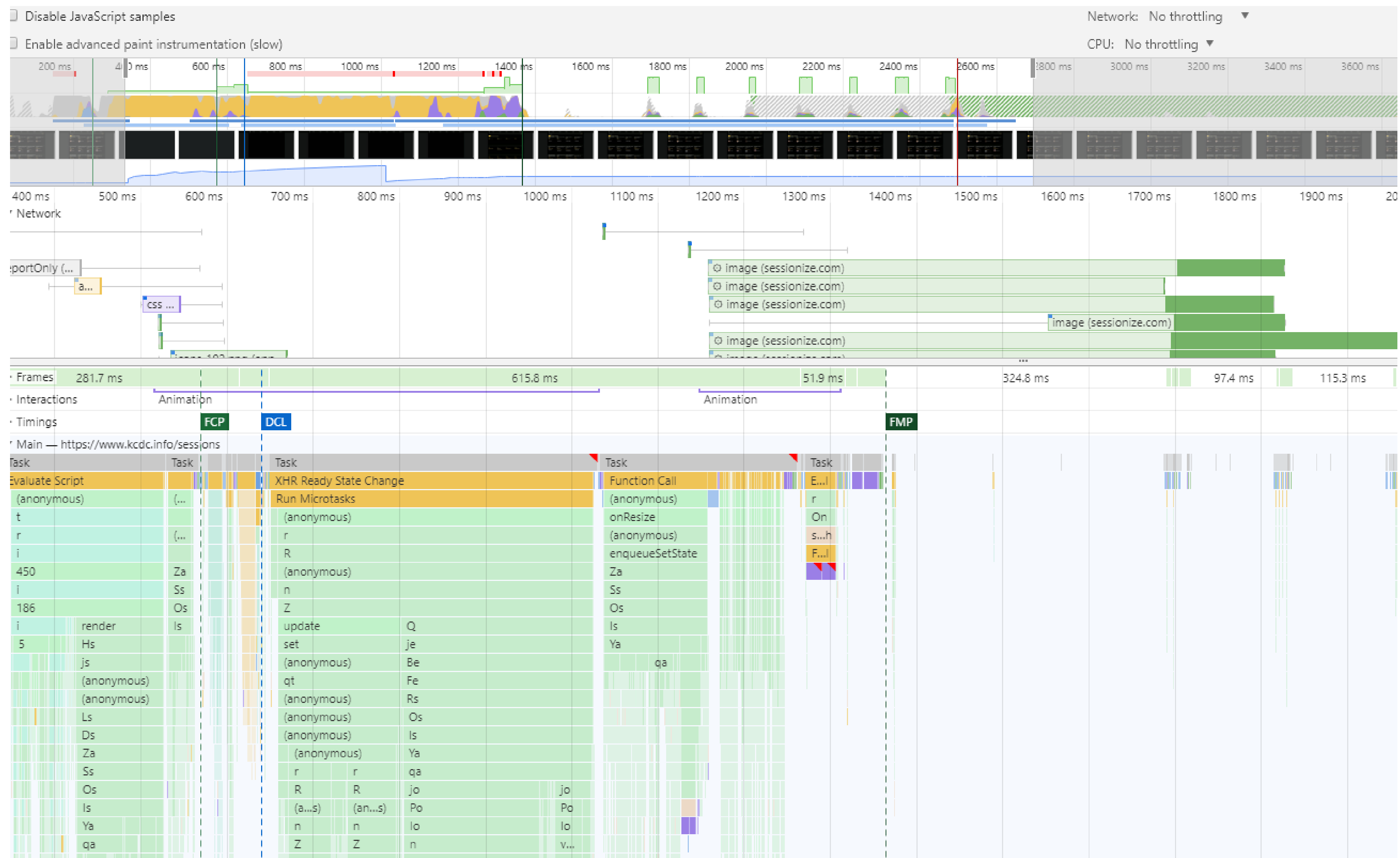
```
  const request = new XMLHttpRequest();
  request.onloadend = () => {
    console.log('Non Blocking complete!');
  };
  request.open('GET', 'https://jsonplaceholder.typicode.com/comments', true);
  request.send();
  console.log('Out of try block.');
```

```
  } catch (err) {
    console.log('Error found.', err);
  } finally {
    console.log('Finally thats done');
```

```
  }
}
```


2.) MEASURE PERFORMANCE

Chrome & Firefox both ship with great performance monitoring tools



DEMO: CHROME PERFORMANCE TOOLS

Real User Monitoring

- Important to capture & collect statistics from a wide array of devices, countries, regions, network conditions, etc.
- Collect *meaningful* stats and send them to a centralized analytics store for analysis.
- Third party paid tools are available (Tealeaf, New Relic, Dynatrace).

Built in Events

- `domInteractive` - When the browser has finished parsing the document.
- `domContentLoaded` – browser has finished parsing the document (DOM construction complete). Scripts such as `defer` and `inline` have finished parsing and executing.
- `domComplete` – All scripts run and all dependent resources (styles, images) finished downloading.
- `First Paint` – first time the browser rendered something after navigation, the first time anything on the screen changes from what was on the screen prior to the navigation.
- `First Contentful paint` – first time the browser renders any text, image, non-white canvas or SVG from the DOM.

Performance APIs: Monitoring Crack

A series of high-resolution timers & observers built around navigation, resources, and your own custom events.

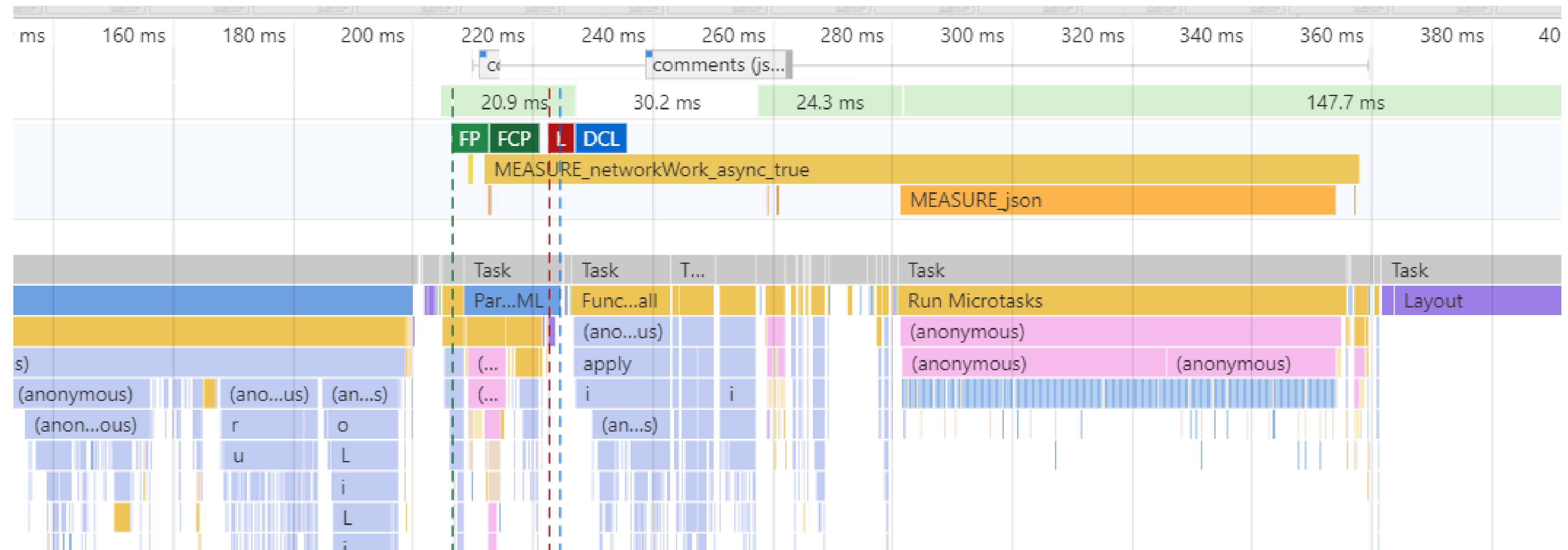
Type	API	Example	Use
Performance	Window.Performance	Window.performance.nav igation	Legacy APIs and root interface all others inherit from
Navigation Timing	Window.performance.timing	domInteractive, loadEventEnd, connectEnd	Built in events created by the browser.
User Timing	Window.performance.mark, window.performance.measure	N/A	For you to create and use.
Resource Timing	Window.performance.getEntries ByType('resource')	Any external resource (Script, SVG, CSS, XHR)	Retrieving detailed performance report on connection times.
Performance Timeline	window.performance.Observe	Window.performance.obs erver('resource')	Performance observers

Instrument your app for insights!

```
const registerPerformance = function(markName) {
  const endName = `END_${markName}`;
  const measureName = `MEASURE_${markName}`;
  window.performance.mark(markName);

  return () => {
    window.performance.mark(endName);
    window.performance.measure(measureName, markName, endName);
    console.log(window.performance.getEntriesByName(measureName, 'measure'));
  };
};
```

- 1.) Create a Mark.
- 2.) Create a end mark.
- 3.) Add a Measure.
- 4.) User timings will display in chrome dev tools.
- 5.) Can also report them to your favorite analytics service for real time user monitoring.





YEAH SURE WHAT ABOUT A REAL SITE?
DEMO TIME!!!!

Instrument Angular

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-top-bar',
  templateUrl: './top-bar.component.html',
  styleUrls: ['./top-bar.component.css']
})
export class TopBarComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  afterViewInit() {
    // After View Init.
  }
}
```

Instrument React

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
  
  componentDidMount() {  
    // measure here  
  }  
}
```

WebPage Test

- Free, open source project that supports user focused performance metrics.
- Website (<https://www.webpagetest.org>) as well as standalone docker containers
- Supported primarily by google, but wide array of industry backers.
- Integrate into your CI/CD pipeline.

Web Vitals Metrics

Google's "helpful" metrics – used to SEO Rank Pages as well

LCP Largest Contentful Paint

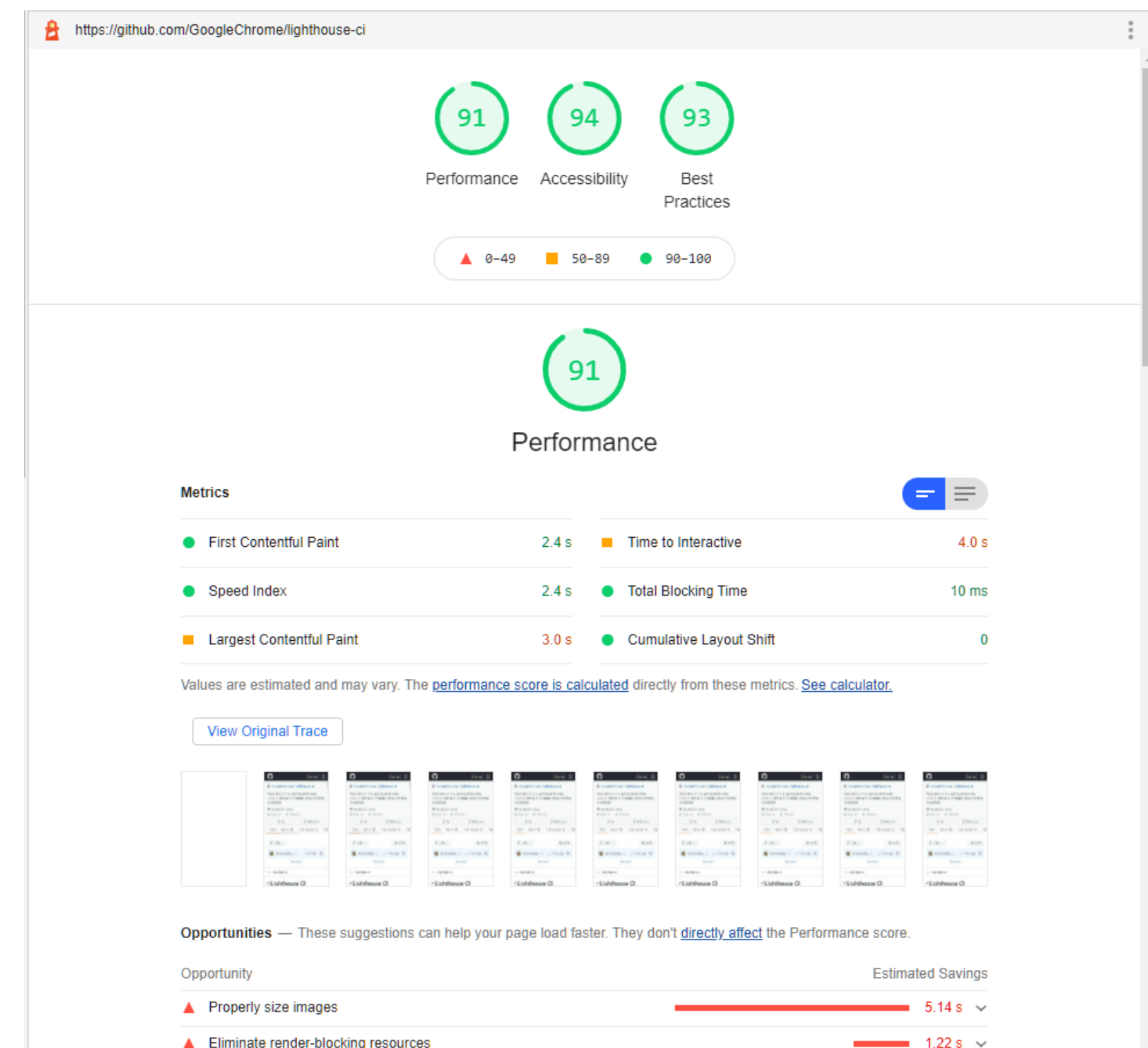
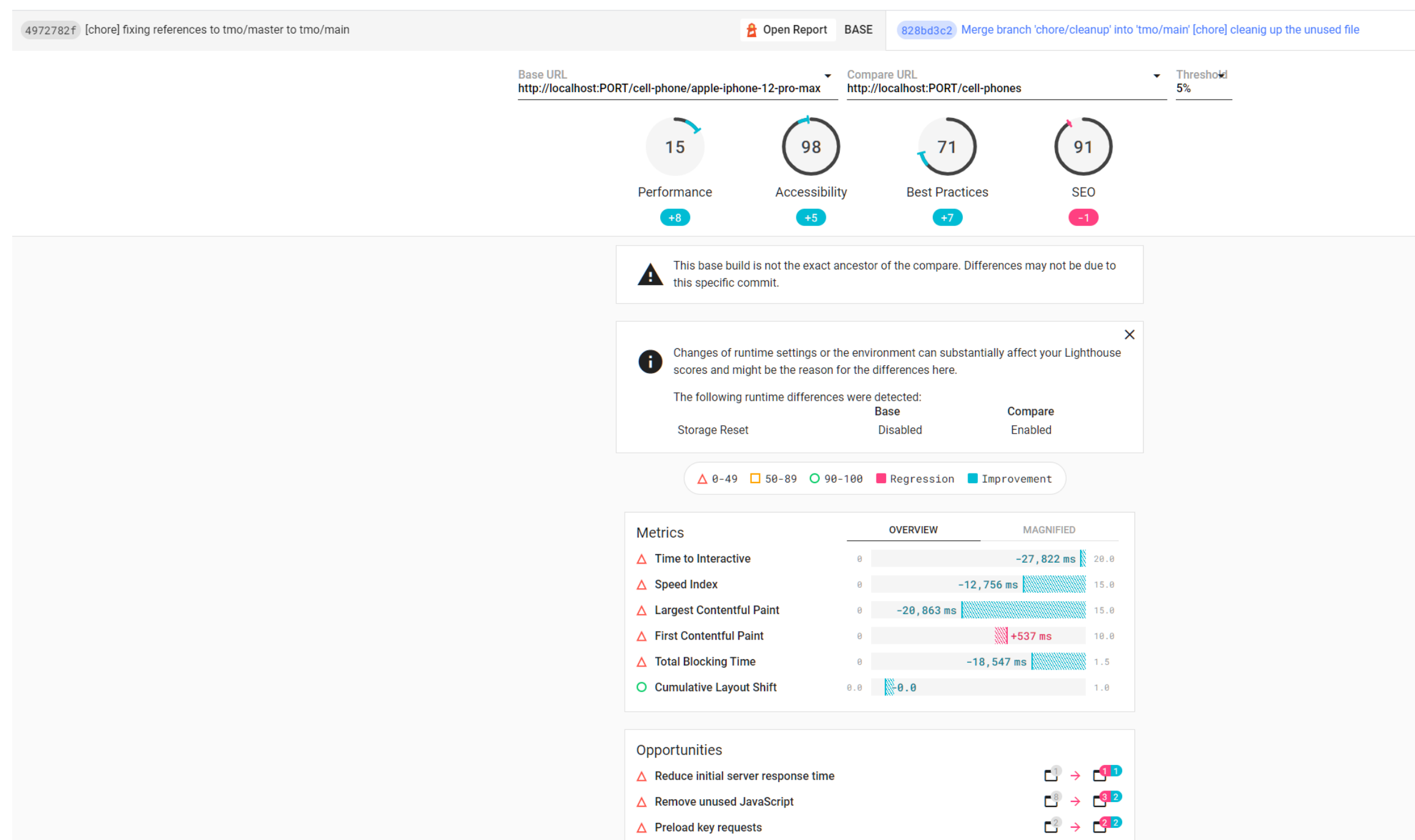
FID First Input Delay

CLS Cumulative Layout Shift

<https://web.dev/learn-web-vitals/>

Lighthouse & Lighthouse CI

Tools from Google, free can integrate them into your CI Pipeline
(<https://github.com/GoogleChrome/lighthouse-ci>)



RAIL Model

Use your new instrumentation to understand performance from a *user* perspective in these 4 scenarios

Render

Animation

Idle

Load



OPTIMIZATION TECHNIQUES

Once you measure... follow this convenient flowchart

1.) Establish a performance budget
(<https://www.performancebudget.io/>)

2.) Are your file sizes too large?

3.) Is render performance / CPU usage the issue?

Webpack: Code Splitting, Minification, Tree Shaking

Code Splitting – Don't ship one gigantic bundle of code. Conditionally load separate bundles.

Minification – let webpack reduce size of JavaScript by shortening variable names, removing comments, etc

Tree Shaking – parses code and removes uncalled functions.

Webpack Stats (analyzer) – visual graph of the size of your JS.

Media Queries in CSS

Use Media Queries to scope your styles, allows much faster construction of CSSOM, Render Tree & Layout steps.

Has built in support for match syntax (only, not, and) as well as different media types (print, screen) as well as supporting resolution/grid/width of viewport sizes.

```
@media only print and (max-width: 800px) {  
    body {  
        background-color: ■ green;  
        color: ■ black;  
        padding-right: 15px;  
        padding-left: 15px;  
    }  
}  
  
@media (min-width: 768px) {  
    body {  
        background-color: □ white;  
        color: ■ black;  
    }  
}
```


Reflow: Avoid like the plague

Reflow: Anything that forces the browser to recalculate the layout of the page, forcing the render tree & layout steps to re-run.

Includes many commonly used methods such as `innerHeight`, `innerText`, `focus`, `scroll`, etc.

Instead of checking window height, use the `matchMedia` interface!

```
const result = window.matchMedia('(max-width: 1600px)');  
  
if (result.matches) {  
  console.log('Big screen found!');  
}
```

[Violation] Forced reflow while executing JavaScript took 44ms

<https://gist.github.com/paulirish/5d52fb081b3570c81e3a>

Network Optimization Techniques

HTTP/2 – requires HTTPS and somewhat modern web servers, much more efficient at handling multiple requests.

GZIP – natively gzip's files on the server reducing file size in exchange for minimal CPU overhead.

CDN – Content Delivery Network, handles delivering assets to users
With many data centers across the world, reducing transmission delays.

Response Headers

```
accept-ranges: bytes  
cache-control: max-age=172724  
content-encoding: gzip  
content-length: 61225
```

Etags – A specification around hashing content and allowing browser
To check if the local copy cache is valid by only sending the hash to the server.

Cache-Control directives – allows on disk caching by the browser itself.

TTFB – Time to first byte, long delay time indicates overwhelmed servers, misconfigured servers, or network bottleneck.

Cookies: STOP USING COOKIES FOR EVERYTHING! Get sent with *every* request! Adds up.

Service & Web Workers

Service Workers – a separate thread that you can queue work onto that does not take up the main UI thread. Intercepts network traffic leaving the browser and allows you to modify it without the main program necessarily knowing. Think offline requests, caching, and pre-fetching.

Web Workers - A script initiates an XHR request and handles the response back from the server. Response is set to another thread (the web worker). Good use case would be any XHR request that when the response comes back requires heavy CPU such as a large JSON payload

All collected into one central location

https://github.com/BillDinger/WebPageSpeed/blob/master/performance_checklist.md

Performance Checklist

Sourced from industry standard best practices

[Mozilla Developer Network's Performance Best Practices](#) [Google's web performance fundamentals](#) [Google's page speed insight](#) [Akamai's performance best practices](#)

Basics

1. Has a set of baseline scans been completed for existing site?
2. Is there a [performance budget](#) set?
3. Is performance monitoring setup using RUM or other metrics?
4. Are pages [TTFP](#) under 1s?
5. Does the [page load](#) in under 3s on an environment other than local?
6. Don't guess, measure - have you used the [performance api](#) and the [built in browser tooling](#) to evaluate bottlenecks on critical pages?
7. Are you using [caching and etags for static assets](#)

CSS

1. CSS is minified and compressed
2. CSS is not located in body
3. There are 2 or fewer 1st party CSS files (bundle your CSS)
4. Critical path CSS is inline in head
5. For animation use [CSS Transforms](#) instead of absolute positioning to let the GPU do hardware acceleration.

RESOURCES

The Cost of JavaScript in 2019: <https://v8.dev/blog/cost-of-javascript-2019>

The JavaScript event loop: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Event_loop

Google Web Fundamentals: Critical Rendering Path
<https://developers.google.com/web/fundamentals/performance/critical-rendering-path/>

What causes reflow: <https://gist.github.com/paulirish/5d52fb081b3570c81e3a>

Script scheduling: bit.ly/script-scheduling

HTML parsing spec: <https://html.spec.whatwg.org/multipage/parsing.html#the-end>

RAILs monitoring: <https://developers.google.com/web/fundamentals/performance/rail>

Browser Caching: <https://tools.ietf.org/html/rfc7234>

Service Workers: [https://developer.mozilla.org/en-US/docs/Web/API/Service Worker API/Using Service Workers](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers)

Google's Core Web Vitals help: <https://web.dev/>

Jon Mill's talk on service workers: <https://www.youtube.com/watch?v=C6S-SOqAX-k>

THANK YOU.

<https://github.com/BillDinger/WebPageSpeed>

