

Bootcamp

Web Performance

Simple yet Complex



/devkode.io



/sunnypuri



/sunnypuri_

SUNNY PURI
Frontend Developer

By the end of this session...

You will

- ... learn about devtools
 - ... learn about network waterfall
 - ... learn about resource caching
 - ... learn about timing breakdown
 - ... learn about gZip compression
 - ... learn about preload and prefetch
 - ... learn about Web vitals
 - ... learn about Lighthouse
 - ... learn about Layout thrashing, page jank
- ... and many more optimization techniques.



@devkode.io

Devkode is an open learning platform where anyone can join as a speaker/participant/mentor

We are having Frontend related technical sessions/quiz/challenges on weekly basis

Currently we have two active social platforms where we connect with each other

learn.devkode.io
t.me/teamdevkode

First Contentful Paint

Time to Interactive

Largest Contentful
Paint

Cumulative Layout
Shift

Lighthouse

Waterfall

Cache-Control

Timing
breakdown

gZip

Network

WEB PERFORMANCE

Full day Bootcamp

21st March, 2021

devkode.io/web-performance

Profiling

Reflow / Repaint

Page jank

Layout / Paint /
Composite

60 FPS

Optimization

Async / Defer

Tree shaking

Prefetch /
Preconnect

Lazy loading

14

Performance

Metrics

▲ First Contentful Paint	4.9 s	▲ Time to Interactive	5.4 s
▲ Speed Index	5.3 s	▲ Total Blocking Time	740 ms
▲ Largest Contentful Paint	8.4 s	■ Cumulative Layout Shift	0.114

Values are estimated and may vary. The [performance score is calculated](#) directly from these metrics. [See calculator.](#)

[View Original Trace](#)

100

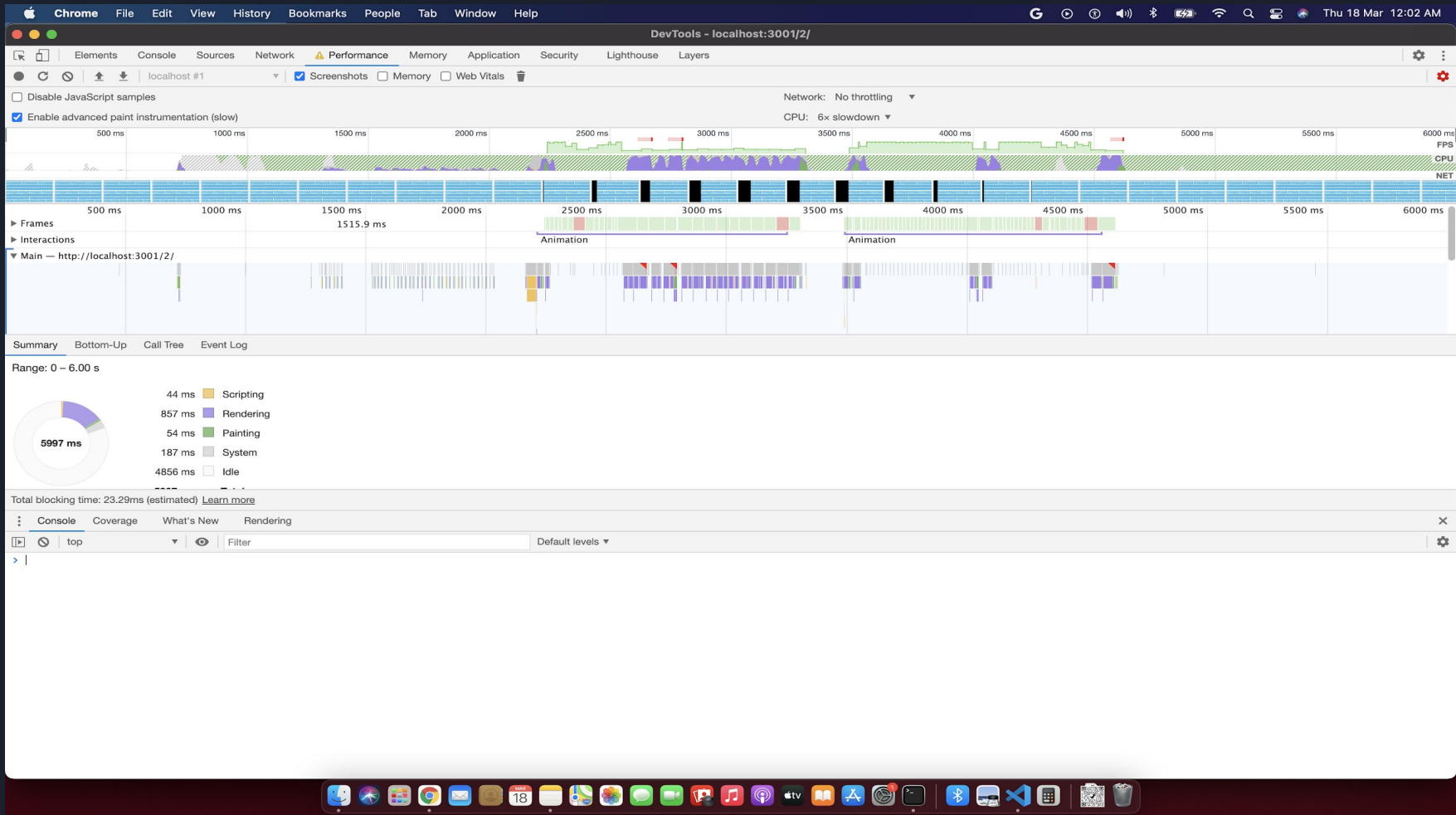
Performance

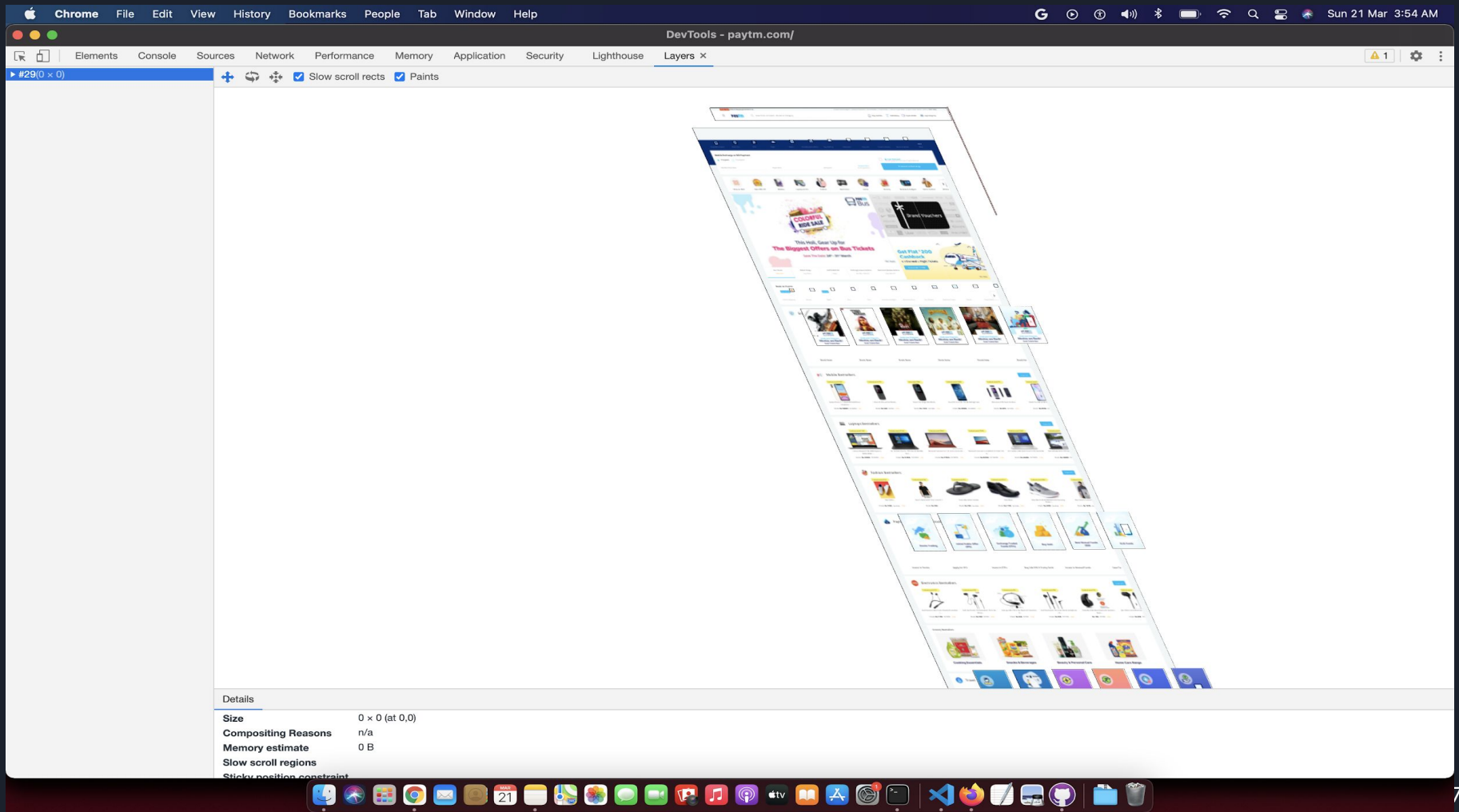
Metrics

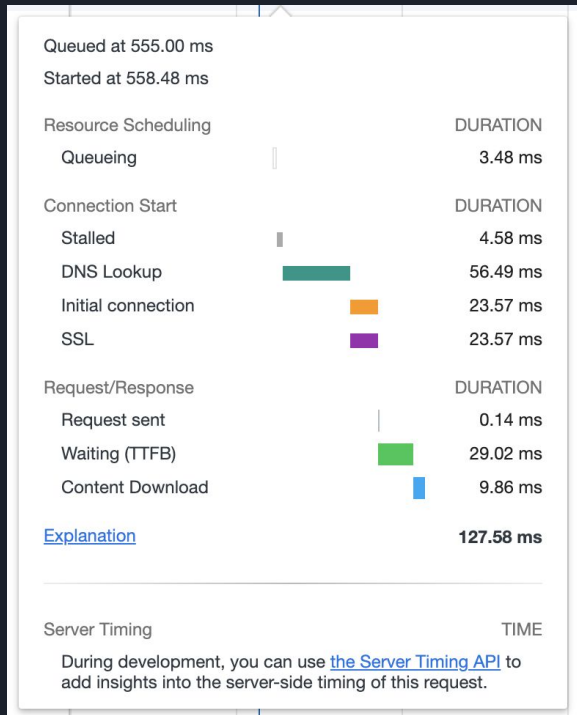
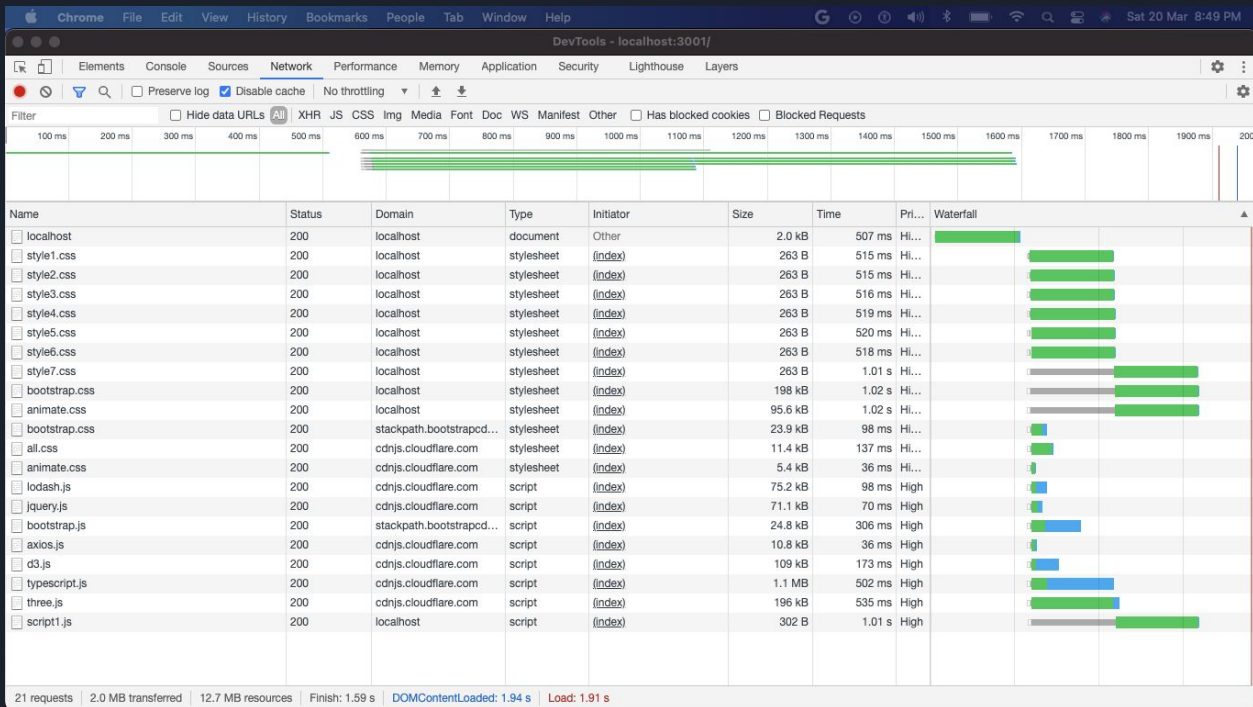
● First Contentful Paint	0.3 s	● Time to Interactive	0.3 s
● Speed Index	0.3 s	● Total Blocking Time	0 ms
● Largest Contentful Paint	0.8 s	● Cumulative Layout Shift	0

Values are estimated and may vary. The [performance score is calculated](#) directly from these metrics. [See calculator.](#)

[View Original Trace](#)

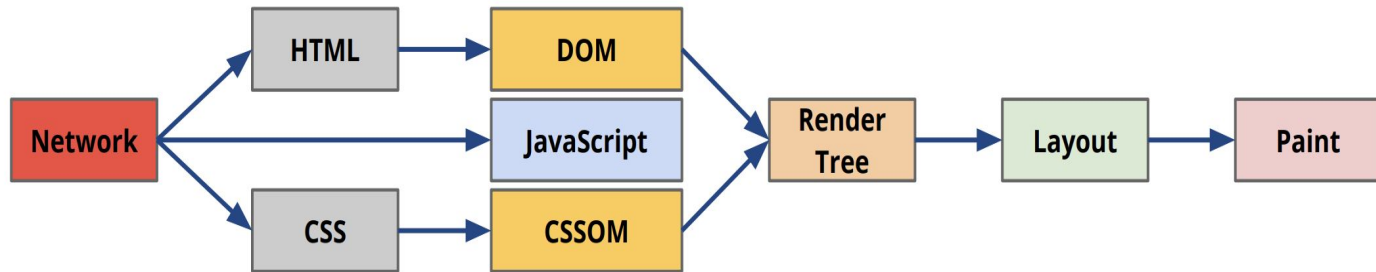






Critical Rendering Path (CRP)

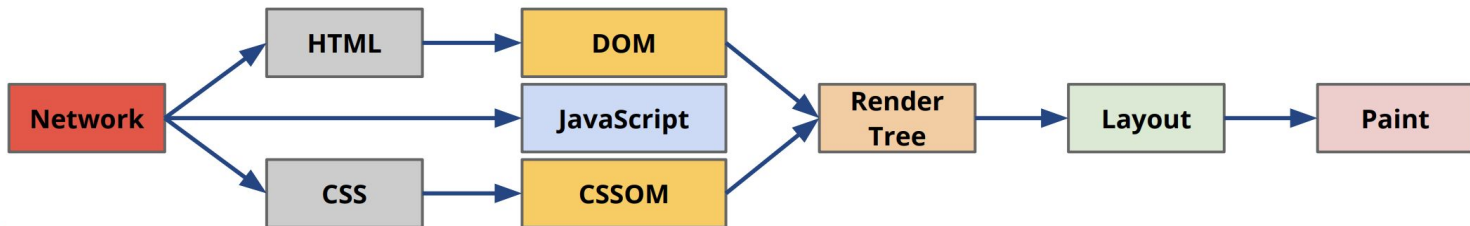
The **Critical Rendering Path** is the sequence of steps the browsers goes through to convert the HTML, CSS and JavaScript into pixels on the screen.



2

Critical rendering path: resource loading

1



*Latency,
bandwidth
3G / 4G / ...*

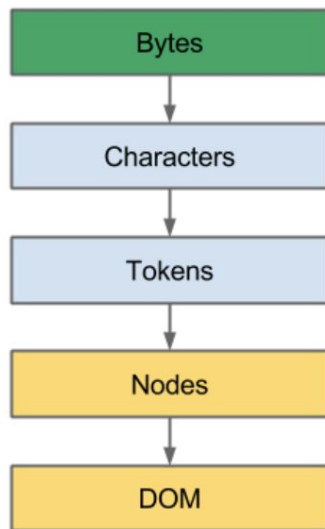
3

In-app performance: CPU + Render

Document Object Model (DOM)



```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div></div>
  </body>
</html>
```

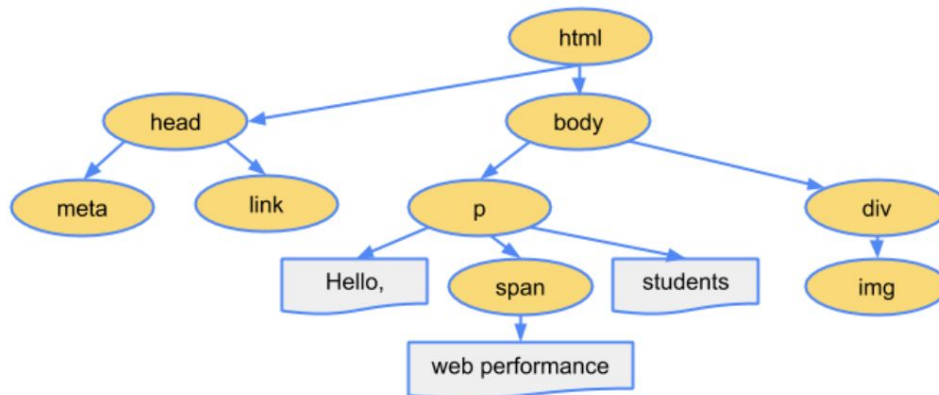


3C 62 6F 64 79 3E 48 65 6C 6C 6F 2C 20 3C 73 70 61 6E 3E 77 6F 72 6C 64 21 3C 2F 73 70 61 6E 3E 3C 2F 62 6F 64 79 3E

`<html><head>...</head><body><p>Hello web performance...`

StartTag: html StartTag: head ... EndTag: head StartTag: body StartTag: p Hello ...

html head meta body p Hello

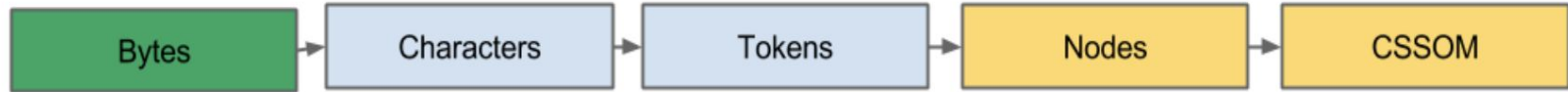


CSS Object Model (CSSOM)

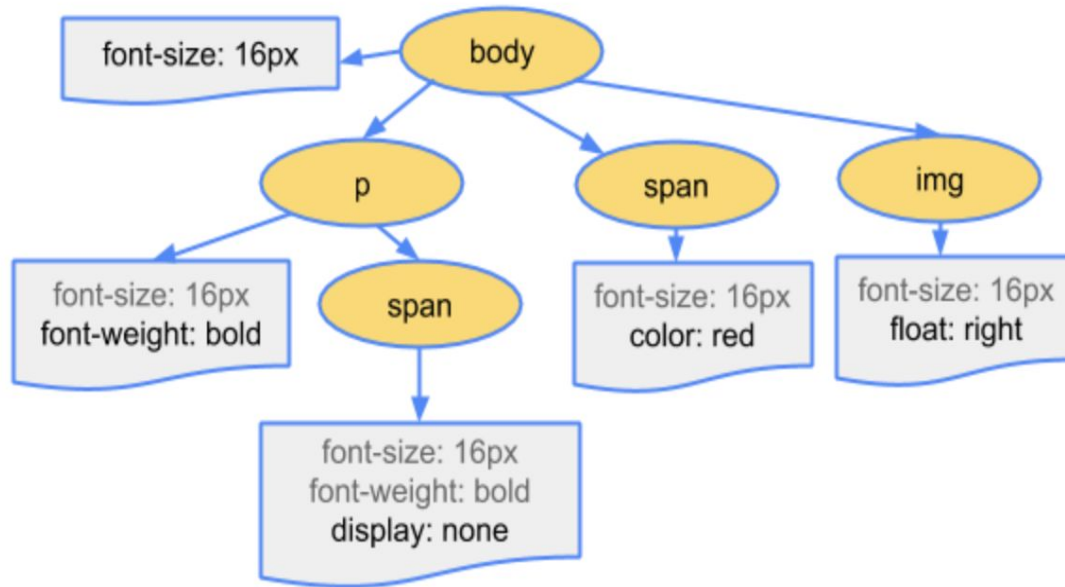
While the browser was constructing the DOM of our simple page, it encountered a link tag in the head section of the document referencing an external CSS stylesheet: `style.css`. Anticipating that it needs this resource to render the page, it immediately dispatches a request for this resource, which comes back with the following content:

```
body { font-size: 16px }  
p { font-weight: bold }  
span { color: red }  
p span { display: none }  
img { float: right }
```

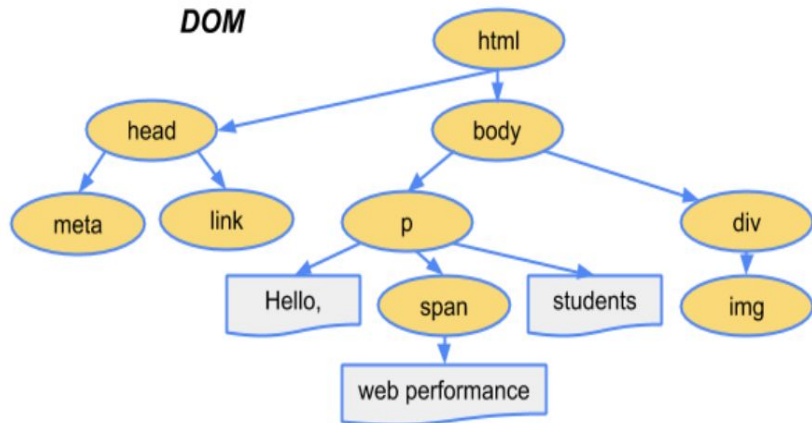




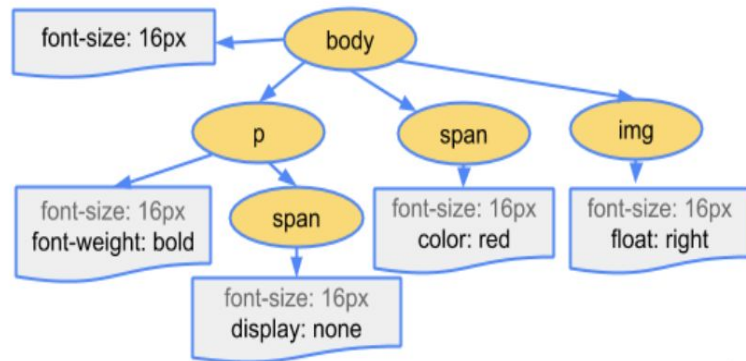
The CSS bytes are converted into characters, then tokens, then nodes, and finally they are linked into a tree structure known as the "CSS Object Model" (CSSOM):



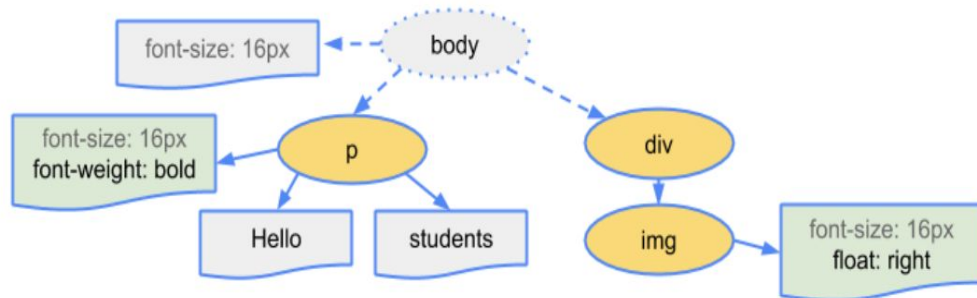
DOM



CSSOM



Render Tree



Performance = 60 FPS

Most devices today refresh their screens **60 times a second**.

If there's an animation or transition running, or the user is scrolling the pages, the browser needs to match the device's refresh rate and put up 1 new picture, or frame, for each of those screen refreshes.

Each of those frames has a budget of just over 16ms ($1 \text{ second} / 60 = 16.66\text{ms}$). In reality, however, the browser has housekeeping work to do, so all of your work needs to be completed inside **10ms**.

When you fail to meet this budget the frame rate drops, and the content judders on screen. This is often referred to as **jank**, and it negatively impacts the user's experience.

If you change a “layout” property, so that’s one that changes an element’s geometry, like its width, height, or its position with left or top, the browser will have to check all the other elements and “reflow” the page. Any affected areas will need to be repainted, and the final painted elements will need to be composited back together.



JS / CSS > Style > Layout > Paint > Composite

If you changed a “paint only” property, like a background image, text color, or shadows, in other words one that does not affect the layout of the page, then the browser skips layout, but it will still do paint.



JS / CSS > Style > Paint > Composite

If you change a property that requires neither layout nor paint, and the browser jumps to just do compositing



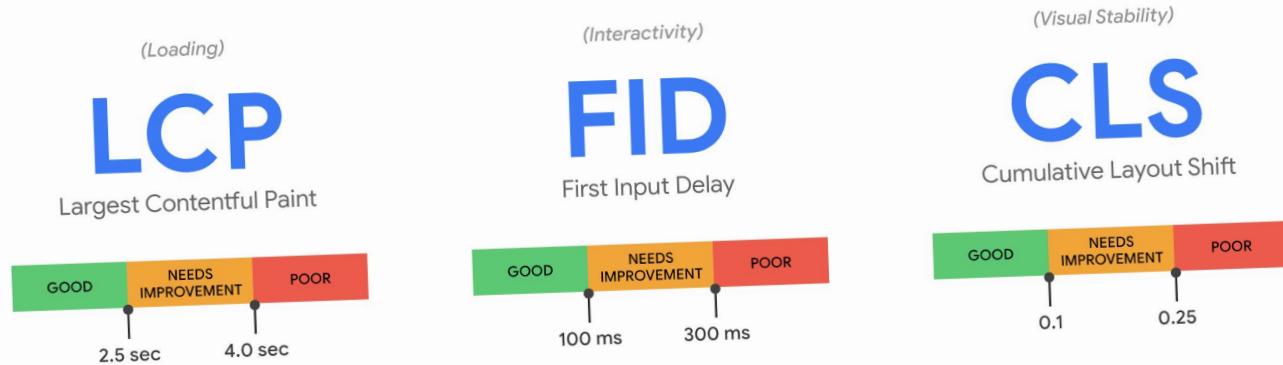
JS / CSS > Style > Composite

csstriggers.com

Web Vitals

Web Vitals is an initiative by Google to provide unified guidance for quality signals that are essential to delivering a great user experience on the web.

Three aspects of the user experience:
Loading, **Interactivity**, and **Visual Stability**



Timing

Queued at 175.23 ms

Started at 177.40 ms

Resource Scheduling

Queueing



DURATION

2.17 ms

Connection Start

DURATION

Stalled



4.95 ms

Proxy negotiation



0.39 ms

DNS Lookup



4.27 ms

Initial connection



47.46 ms

SSL



47.46 ms

Request/Response

DURATION

Request sent



0.14 ms

Waiting (TTFB)



25.43 ms

Content Download



1.07 ms

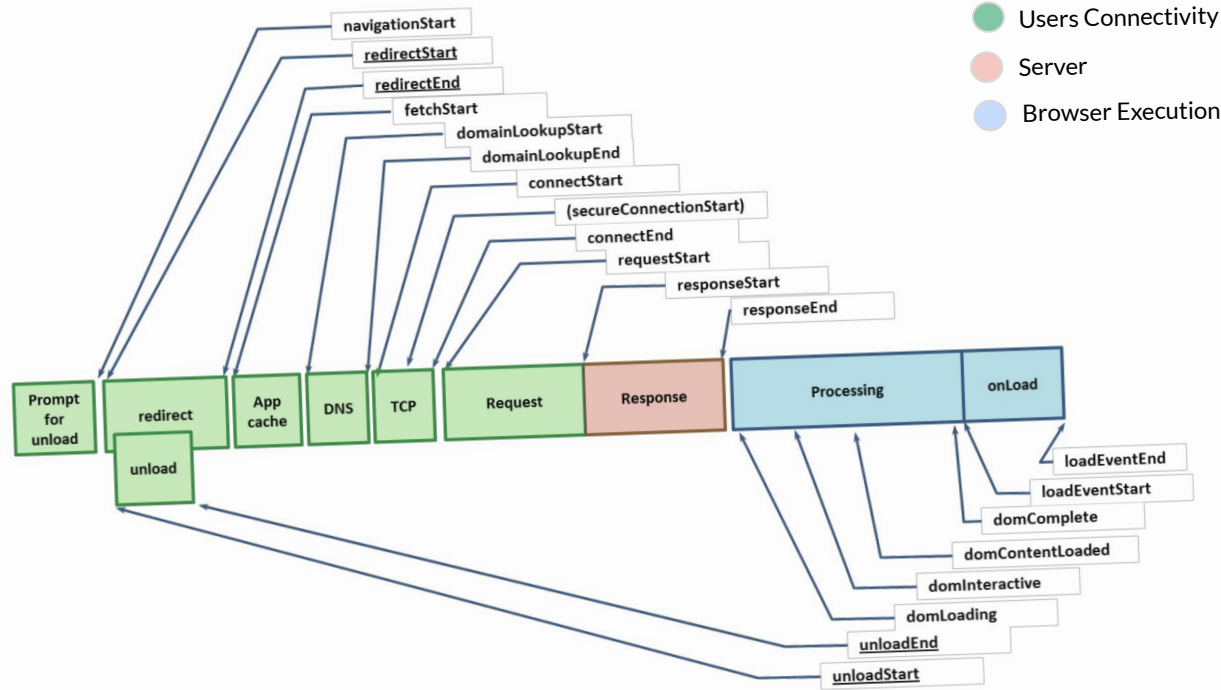
[Explanation](#)

85.93 ms

Server Timing

TIME

During development, you can use [the Server Timing API](#) to add insights into the server-side timing of this request.



Performance timeline API

The things we measure with the Performance API are referred to as entries. These are the performance entries that are available to us:

- mark
- measure
- navigation
- resource
- paint
- frame

Performance Timeline API provides the following three methods, which are included in the performance interface:

- `getEntries()`
- `getEntriesByName()`
- `getEntriesByType()`

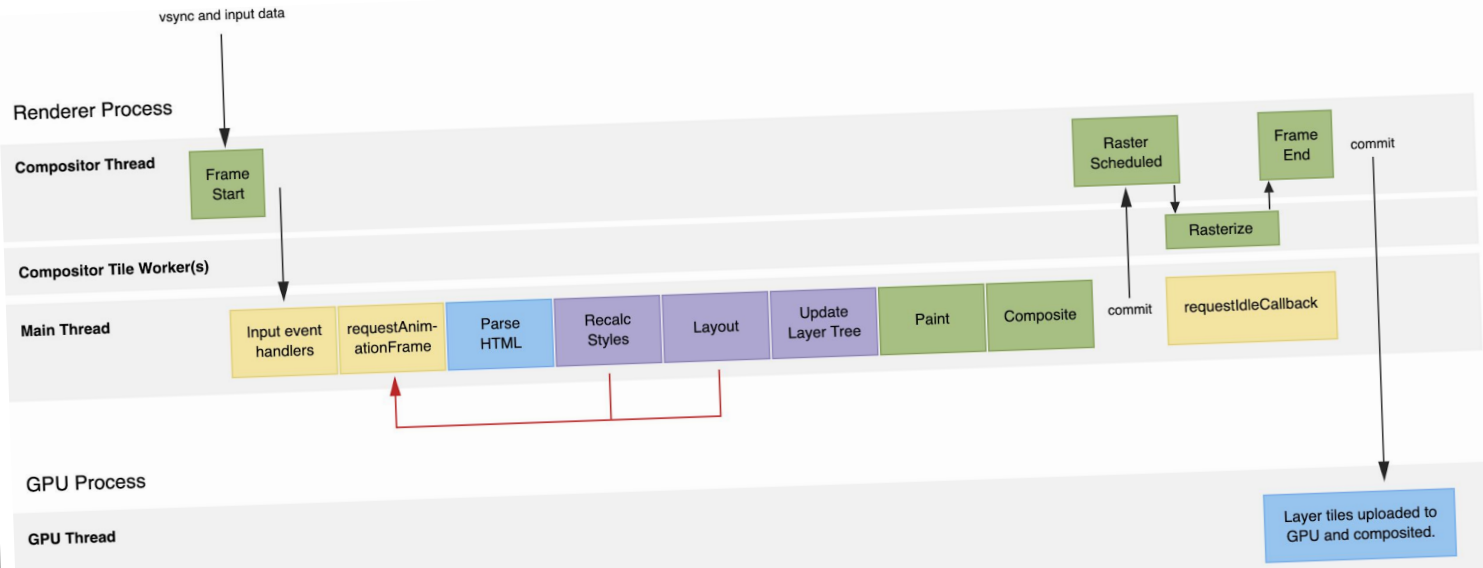
Each method returns a list of performance entries gathered from all of the other extensions of the Performance API.

PerformanceObserver is another interface included in the API. It watches for new entries in a given list of performance entries and notifies of the same.

Example:

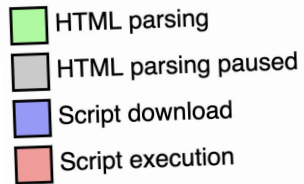
```
const navigationEntries = performance.getEntriesByType("navigation")[0];
```

Performance

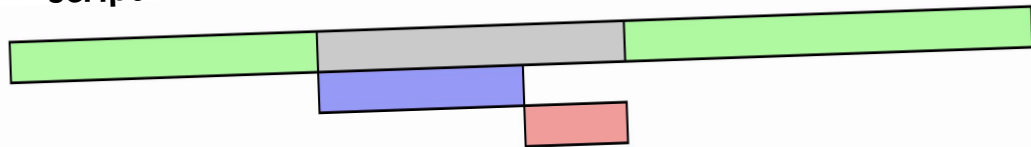


- **Compositor Thread.** This is the first thread to be informed about the vsync event (which is how the OS tells the browser to make a new frame). It will also receive any input events. The compositor thread will, if it can, avoid going to the main thread and will try and convert input (like – say – scroll flings) to movement on screen.
It will do this by updating layer positions and committing frames via the GPU Thread to the GPU directly. If it can't do that because of input event handlers, or other visual work, then the Main thread will be required.
- **Main Thread.** This is where the browser executes the tasks we all know and love: JavaScript, styles, layout and paint. This thread wins the award for “most likely to cause jank”.
- **Compositor Tile Worker(s).** One or more workers that are spawned by the Compositor Thread to handle the Rasterization tasks.

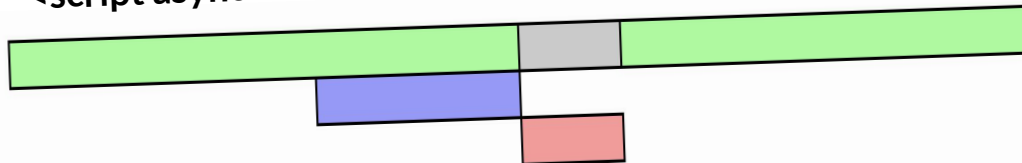
Async/Defer



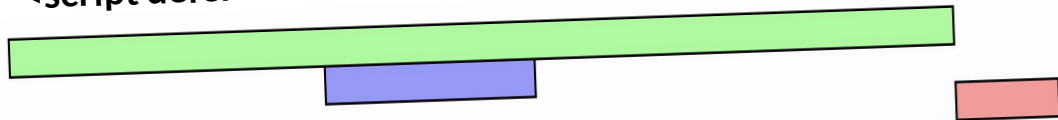
<script>



<script async>



<script defer>



Layout thrashing

What forces layout / reflow

All of the below properties or methods, when requested/called in JavaScript, will trigger the browser to synchronously calculate the style and layout*. This is also called reflow or layout thrashing, and is common performance bottleneck.

Element APIs

Getting box metrics

- `elem.offsetLeft`, `elem.offsetTop`, `elem.offsetWidth`, `elem.offsetHeight`, `elem.offsetParent`
- `elem.clientLeft`, `elem.clientTop`, `elem.clientWidth`, `elem.clientHeight`
- `elem.getClientRects()`, `elem.getBoundingClientRect()`

Scroll stuff

- `elem.scrollBy()`, `elem.scrollTo()`
- `elem.scrollIntoView()`, `elem.scrollIntoViewIfNeeded()`
- `elem.scrollWidth`, `elem.scrollHeight`
- `elem.scrollLeft`, `elem.scrollTop` also, setting them

Setting focus

- `elem.focus()`

Also...

- `elem.computedRole`, `elem.computedName`
- `elem.innerText`

Maximum Browser requests

TABLE 1. MAXIMUM SUPPORTED
CONNECTIONS

Version	Maximum connections
Internet Explorer® 7.0	2
Internet Explorer 8.0 and 9.0	6
Internet Explorer 10.0	8
Internet Explorer 11.0	13
Firefox®	6
Chrome™	6
Safari®	6
Opera®	6
iOS®	6
Android™	6

Big Thanks




Khushboo Singh

Sr Software developer

Company: PTC

Exp: 5 years

 /imkhushboo



Gopalakrishnan C

Software Engineer

Company: Kissflow

Exp: 4 years

 /gopal1996



Sadanand Akshay Pai

Front-end developer

Company: Trelleborg

Exp: 2 years

 /sadanandpai



Thank you!



/devkode.io



/sunnypuri



/sunnypuri_