

Module 3. Practice LCP optimization

Github repository: <https://github.com/kirillkuts/LCP-optimization-demo>

Project setup

```
git clone git@github.com:kirillkuts/LCP-optimization-demo.git
cd LCP-optimization-demo
npm install
npm run start
```

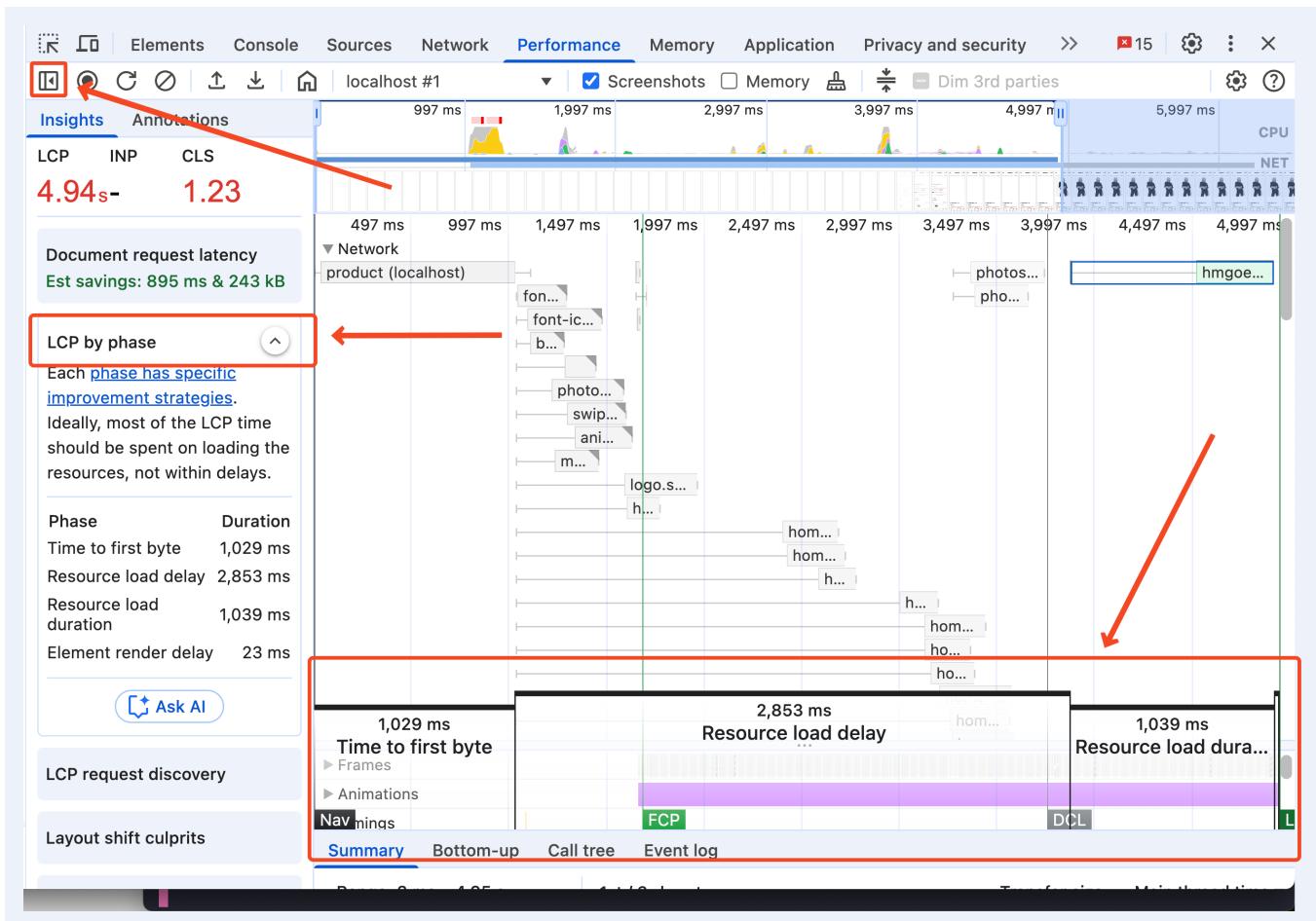
1. Navigate to `http://localhost:3000/product` in Chrome
2. Open DevTools, navigate to `Performance` tab
3. Observe the local LCP value

The screenshot shows the Google Chrome DevTools Performance tab open on a product page for a 'Cotton jersey top' from the 'ecomus' website. The 'Performance' tab is selected, and the 'Screenshots' checkbox is checked. The 'Local and field metrics' section displays the Largest Contentful Paint (LCP) value as 5.12 s, which is highlighted with a red box. Below it, the Cumulative Layout Shift (CLS) value is shown as 0.00. The 'Interactions to Next Paint (INP)' section shows values for Local and Field 75th percentile. The 'Next steps' section includes a 'Field data' card and an 'Environment settings' card. A red arrow points from the 'Screenshots' checkbox in the DevTools header to the LCP value in the performance panel.

4. Run the Performance Analysis by clicking on the “Record and reload” button on the top left

The screenshot shows the Chrome DevTools Performance tab. At the top, there are several checkboxes: 'Disable JavaScript samples' (unchecked), 'Enable advanced paint instrumentation (slow)' (unchecked), 'Enable CSS selector stats (slow)' (unchecked), 'Screenshots' (checked), 'Memory' (unchecked), 'Dim 3rd parties' (unchecked). Below the checkboxes, it says 'CPU: No throttling' and 'Network: Slow 4G'. There's also a checkbox for 'Show custom tracks' with a link to 'Learn more'. The main area has two cards: 'Largest Contentful Paint (LCP)' showing '5.12 s' (Local) and 'Field 75th', and 'Cumulative Layout Shift (CLS)' showing '0.00' (Local) and 'Field 75th'. On the right, there's a 'Next steps' section with a 'Field data' card showing 'Collection period: N' and 'URL - No data'.

5. Toggle the sidebar and click on the “LCP by phase” insight, that should activate the LCP buckets annotations



6. Feel free to click on the sidebar toggler again since we won't need it

Step 1: Optimize Resource Load Delay

While there are plenty of changes that could be implemented I'd suggest taking the “heal where it bleeds most approach” - look for the biggest contributor to the LCP, in this case “Resource Load Delay”

Resource load delay is the wait time before the browser begins loading the LCP resource after the TTFB.

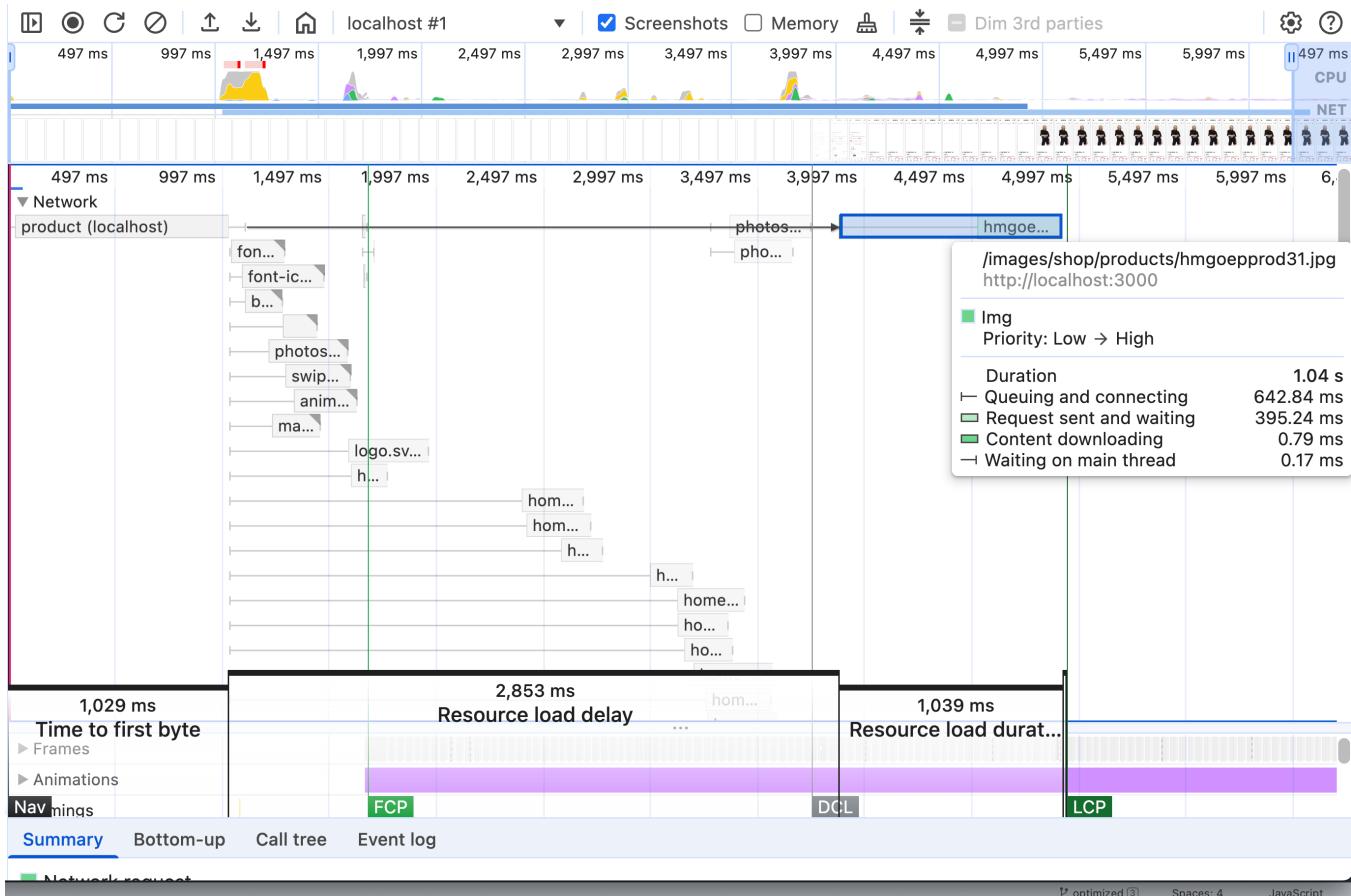
It's one of the most common LCP problems! And is usually caused by the fact that browser prioritizes render-blocking resources (CSS, JS) over images, attempting to optimize for the FCP by default.

Optimization steps

1. Identify the LCP element by hovering over the element block in the performance tab

The screenshot shows the Chrome DevTools Performance tab for a website named 'ecomus'. The main area displays the 'Local and field metrics' section. The 'Largest Contentful Paint (LCP)' metric is highlighted in red, showing a value of **4.95 s**. A red arrow points from the text 'Your local LCP value of 4.95 s is poor.' to the LCP value. Below this, another red arrow points from the text 'Worst cluster 2 shifts' to the 'Cumulative Layout Shift (CLS)' value of **1.23**. The 'Cumulative Layout Shift (CLS)' section also shows a 'Field 75th percentile' value of **1.23**. The 'Interactions to Next Paint (INP)' section is also visible. On the right side of the screen, there is a 'Next steps' panel with sections for 'Field data' and 'Environment settings'.

2. Identify the image path by clicking on the image element (`images/shop/products/hmgoepprod31.jpg`)
3. Run the performance analysis again and search for the image path request



4. Optimize the image loading by:

- removing lazy loading
- adding `fetchpriority="high"` attribute

Horray! We went from 5s to 3.8s!

You can find the code change here <https://github.com/kirillkuts/LCP-optimization-demo/commit/f370b970bb959b849f62c6441a97922bba0a7146>



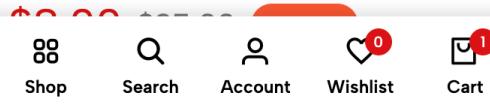
Home > Women > Cotton jersey top



Cotton jersey top

Best seller

⚡ Selling fast! 56 people have this in their carts.



Elements Console Sources Network

Live metrics

Disable JavaScript samples

Enable advanced paint instrumentation (slow)

Enable CSS selector stats (slow)

Local and field metrics

Largest Contentful Paint (LCP)

3.80 s

Local

Field 75th percentile

Your local LCP value of **3.80 s** needs improvement.

LCP element `img.tf-image-zoom`

Interaction to Next Paint (INP)

Local

Field 75th percentile

Interact with the page to measure INP.

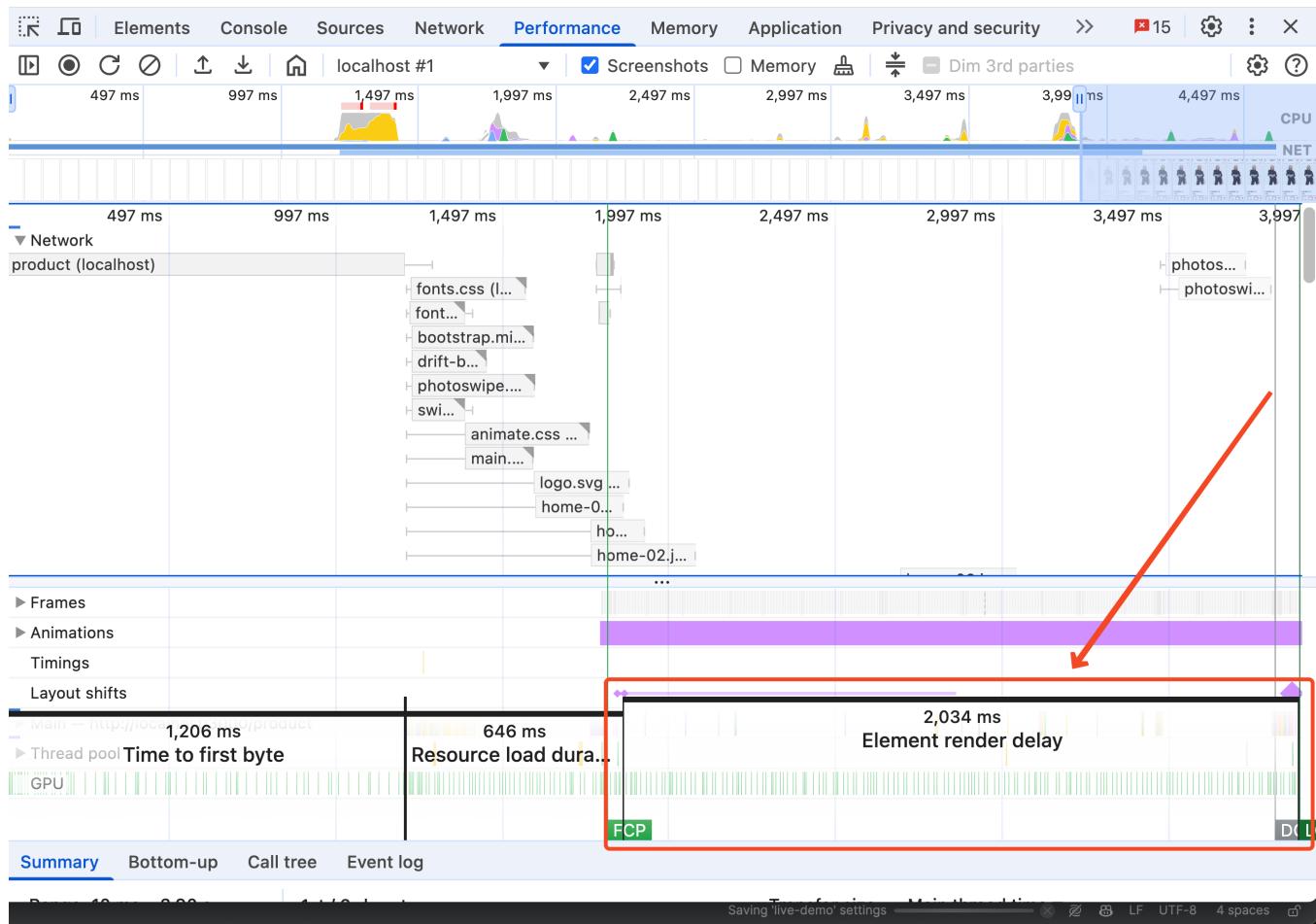
[Learn more about local and field data](#)

[Interactions](#) [Layout shifts](#)

Step 2: Optimize Element Render Delay

1. Re-run the performance analysis and activate LCP annotations again
2. Identify the biggest area - **element render delay**.

Element render delay - time between when the resource finishes loading and when the LCP element is rendered on screen



3. Element render delay optimization can take different form. To eliminate some common problems look for the following symptoms:

- **Have all render-blocking resources have been loaded?** In our case the FCP (first contentful paint) happened way before so that's not a problem
- **Is there a long-running JS task, blocking the main thread?** Look for the long red lines in the performance chart, non happened during the render delay phase in our case
- **Is there a JS logic preventing the LCP element from showing?** YES! In our case (as in a lot of other cases that I saw), JS only removes the loader on `DOMContentLoaded` event! The event only fires when ALL deferred/async scripts have loaded.

Note: deferred/async scripts would get a lower priority since they are not render-blocking

4. Add a code that would hide the preloader as soon as the image loads

You can find the code change here <https://github.com/kirillkuts/LCP-optimization-demo/commit/66f886c4dad85b9d995fb94cca19b20b57b03a0>

Supplementary reading: https://web.dev/articles/optimize-lcp/#2_eliminate_element_render_delay



Elements Console Sources |

Live metrics

Disable JavaScript samples

Enable advanced paint instrumentation (slow)

Enable CSS selector stats (slow)

Local and field metrics

Largest Contentful Paint (LCP) ?

1.83 s

Local

-

Field 75th percentile

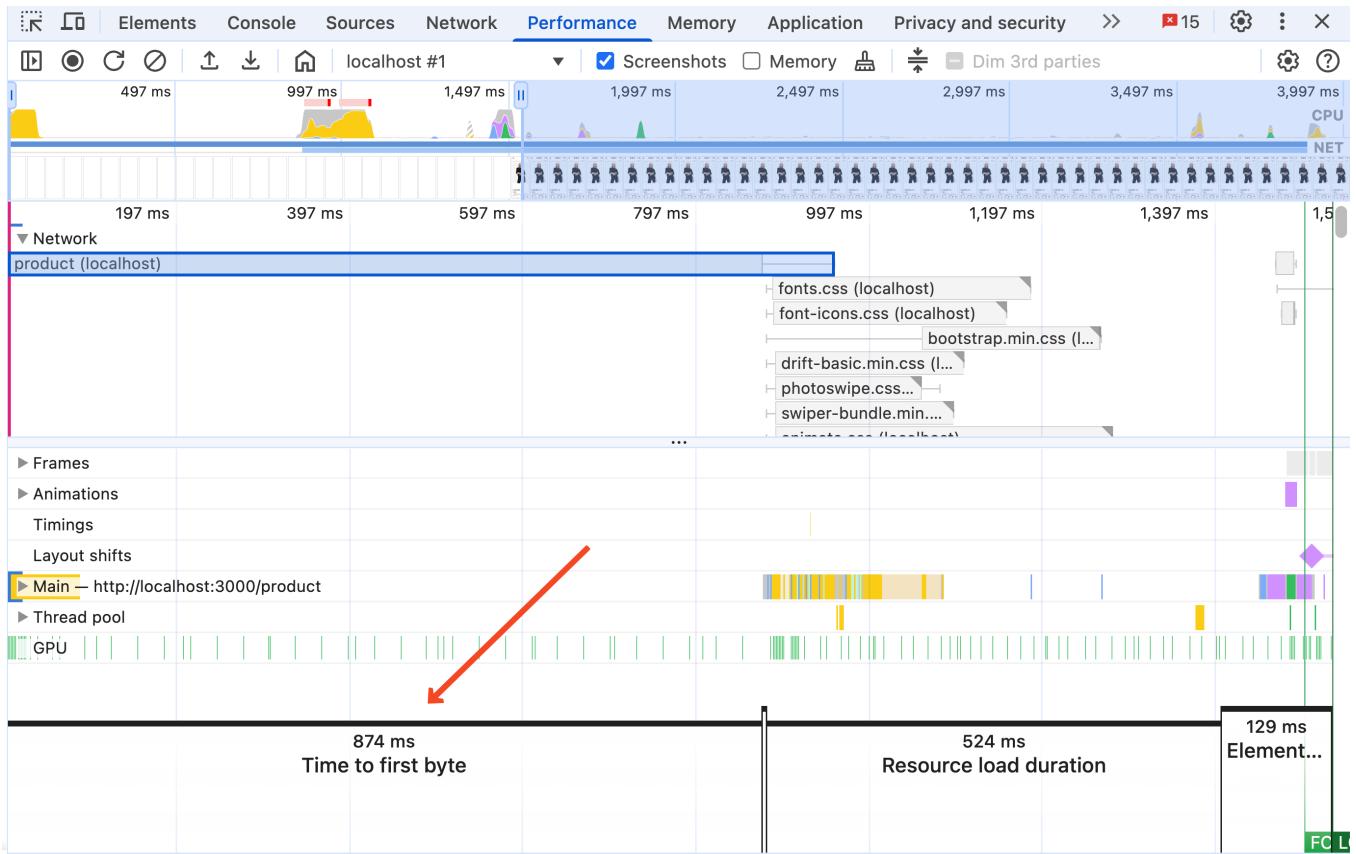
Your local LCP value of 1.83 s is good.

LCP element [img.tf-image-zoom](#)

Step 3: Optimize TTFB, DB calls

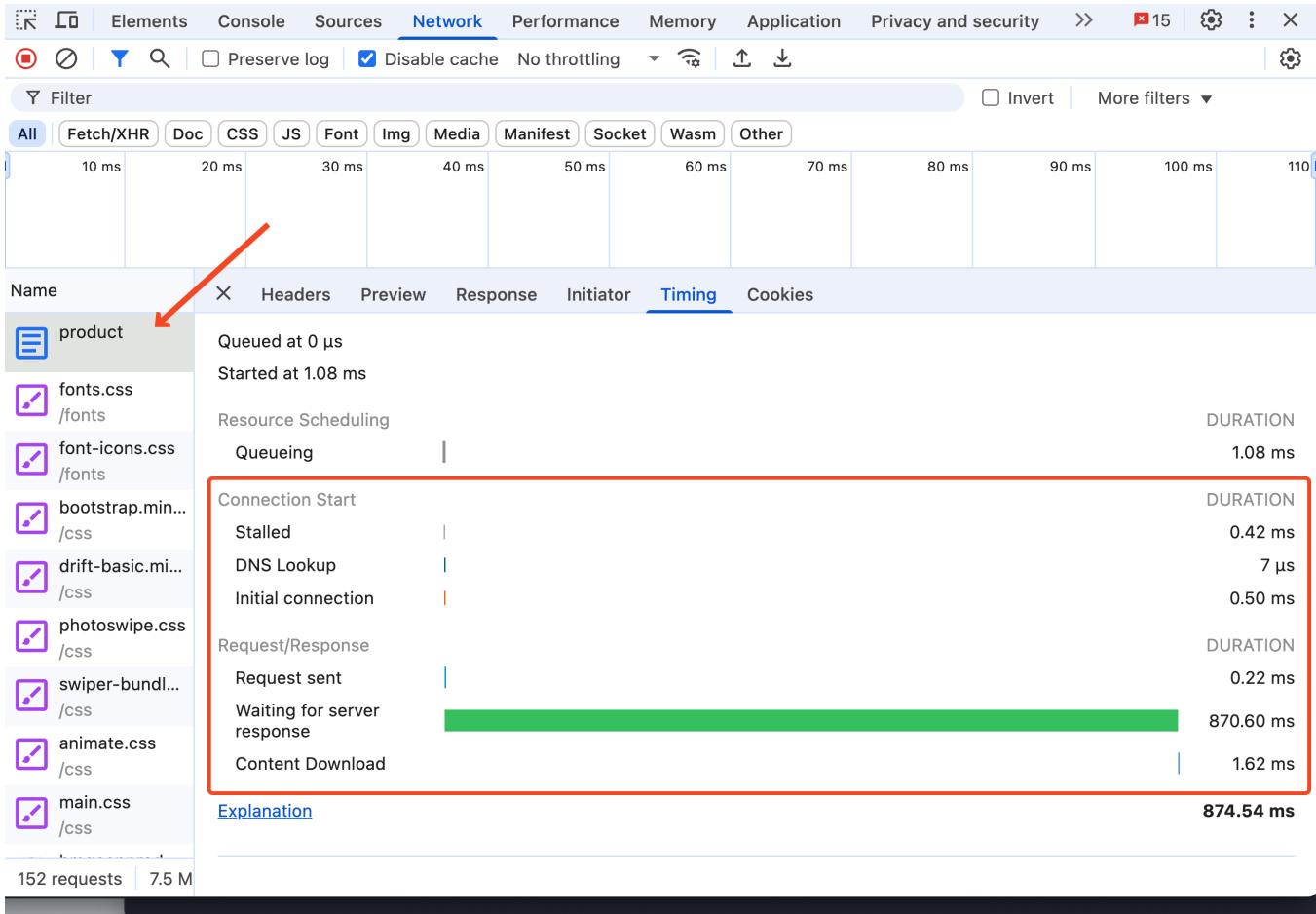
We've got pretty good results so far going from 5s to 1.8s , however we can do even better!

1. Re-run the performance analysis. TTFB seems to be the biggest problem



- Find the first request in the network tab (that should be the request that returns the HTML for the web-page), navigate to the `Timings` tab

Review the time distribution. As you can see most of the time is taken by “Waiting for server response” which is mapping to the “Server latency” (e.g. How much time it took to process my request?)



3. Let's review the codebase and identify potential bottlenecks

4. `server.js` contains the route resolver, which in itself makes 2 sequential API calls. That has a room for improvements since we could make those 2 requests in parallel

```

30     const id = req.params.id;
31     const productDetails = await getProductDetails(id);
32
33     let output = html.replace('{{ gallery }}', ` 
34         <div dir="ltr" class="swiper tf-product-media-main" id="gallery-swiper-started">
35             <div class="swiper-wrapper">
36                 ${productDetails.images.map((img, i) => `
37                     <div class="swiper-slide" data-color="beige">
38                         <a href="#${img}" target="_blank" class="item" data-pswp-width="770px" data-pswp-height="`+
39                             img.height +
40                         `"
41                             data-zoom="${img}"
42                             ${` 
43                                 i === 0 ?
44                                     `src="${img}" fetchpriority="high" onload="hidePreloader()"`+
45                                     `:
46                                     `data-src="${img}"`+
47                                 ``
48                             }`+
49                             ` alt=""`+
50                         `>
51                     </a>
52                 `).join('')}
53             </div>
54         </div>
55     `);
56
57     output = output.replace('{{ priceOld }}', `${productDetails.priceOld}`);
58     output = output.replace('{{ priceNew }}', `${productDetails.priceNew}`);
59     output = output.replace('{{ title }}', `${productDetails.title}`);
60
61     const productStock = await getProductStock(id);
62
63     output = output.replace('{{ sizes }}', ` 
64         <div class="variant-picker-values">
65             ${Object.keys(productStock).map(option => `
66                 <input type="radio" name="size1" id="values-${option}" checked>
67                 <label class="style-text size-btn" for="values-${option}" data-value="S">
68                     <p>${option.toUpperCase()}</p>
69                 </label>
70             `).join('')}
71         </div>
72     `);
73

```

5. Let's parallelize those API calls

<https://github.com/kirillkuts/LCP-optimization-demo/commit/0a46c707143536a8482ec41fe51581cca4632143>

One more 300ms win! Congrats!



Elements Console Sources N

Live metrics

Disable JavaScript samples

Enable advanced paint instrumentation (slow)

Enable CSS selector stats (slow)

Local and field metrics

Largest Contentful Paint (LCP)

1.34 s

Local

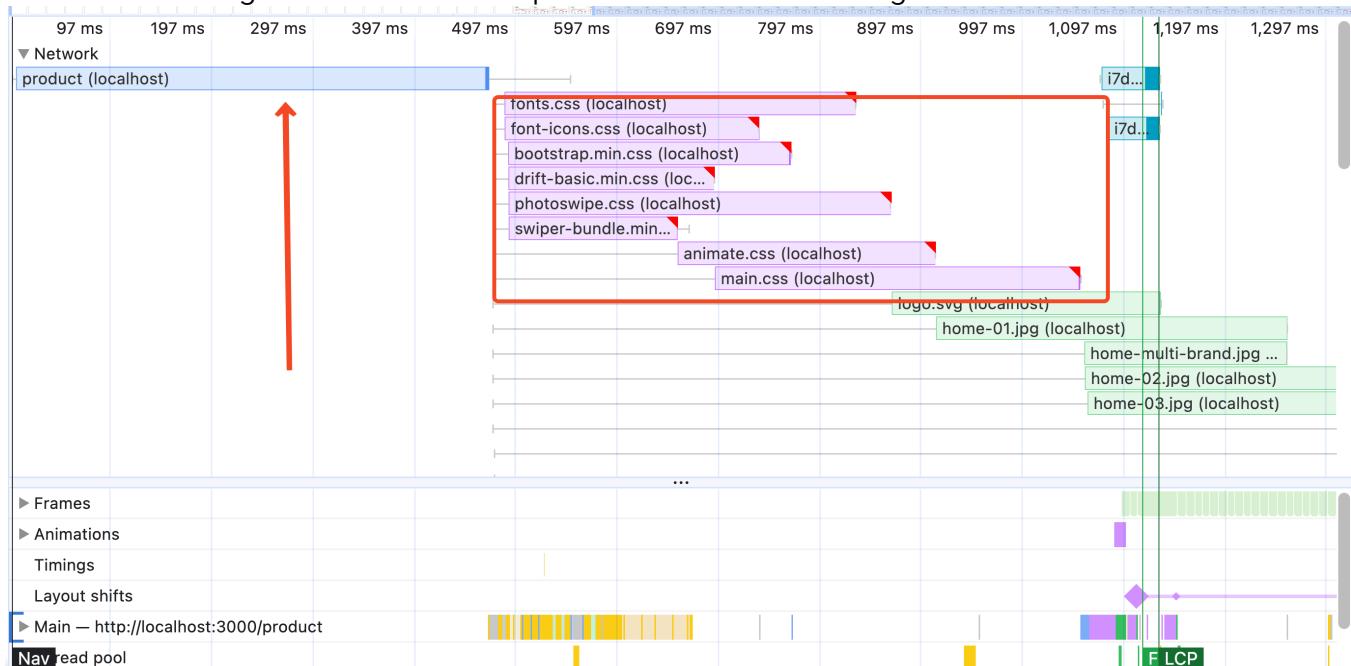
Field 75th percentile

Your local LCP value of 1.34 s is good.

LCP element

Step 4: Optimize TTFB, early head

We are still waiting for the DB to return product data before starting to load common assets.

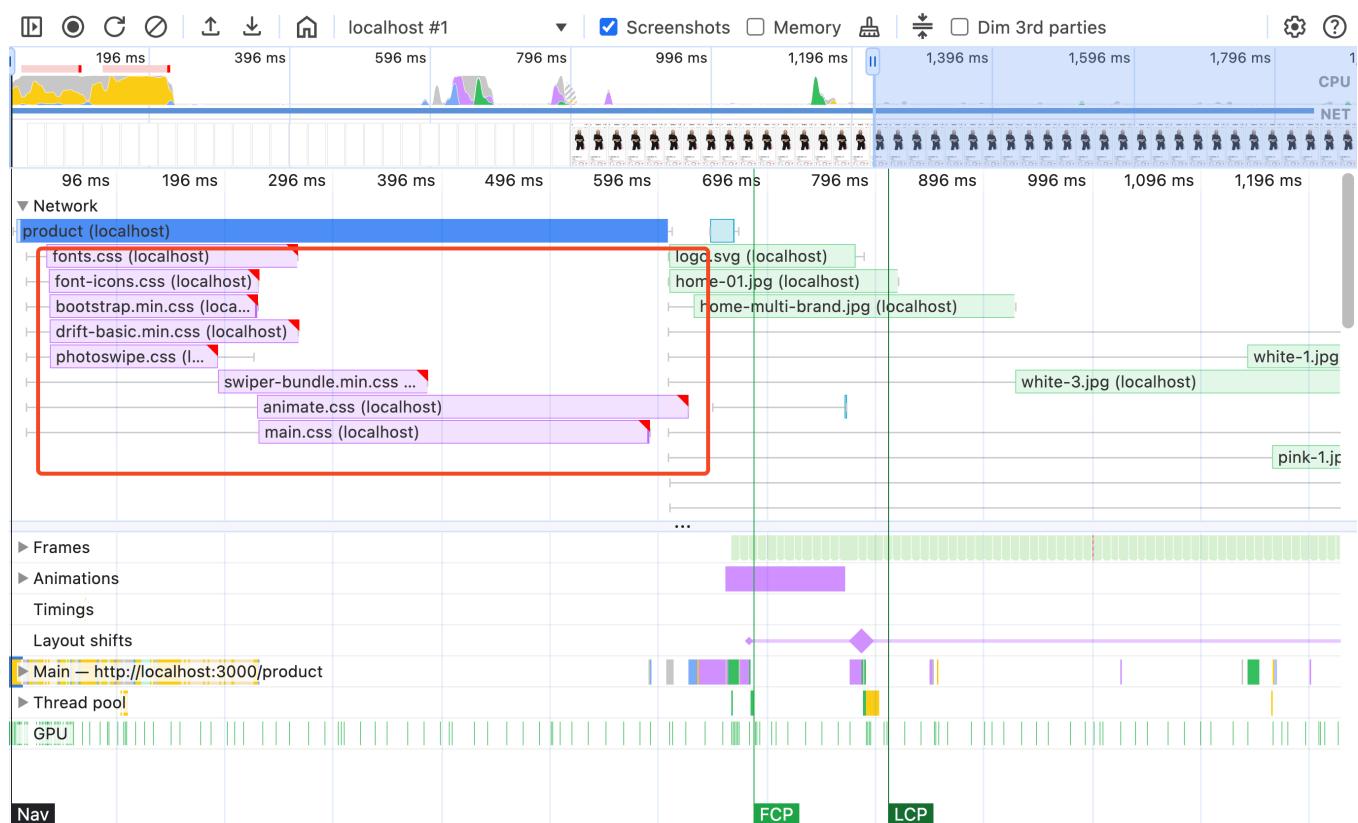


Let's optimize this by ensuring the `head` section is returned before we make the DB call, and the rest

of the page is sent after it

<https://github.com/kirillkuts/LCP-optimization-demo/commit/c9ed997a70b4558501ae5a26ea5df3f9137d29dd>

Much better! The styles start to load early and we effectively parallelized both DB calls on the server and style load on the client, optimizing the LCP by another **300ms**!



Largest Contentful Paint (LCP)

0.94 s Local – Field 75th percentile

Your local LCP value of **0.94 s** is good.

LCP element `img.tf-image-zoom`

Step 5: Optimize TTFB, cache LCP src

Our current setup awaits for the DB to return product data before adding images to the page. However, the DB call takes a long (300-500ms) time. Let's implement a cache layer that would keep LCP link in-memory

For the demo we'll use in-memory map, however for production please reference other non-durable storage tools

<https://github.com/kirillkuts/LCP-optimization-demo/commit/04cfa422e4eb6945af9ad39261e4f7bdf6ddbca0>

Supplementary reading: <https://web.dev/articles/optimize-ttfb/#use-caching>

Another ~100ms WIN!

 Disable JavaScript samples Enable advanced paint instrumentation (slow) Enable CSS selector stats (slow)

Local and field metrics

Largest Contentful Paint (LCP)

0.79 s

Local

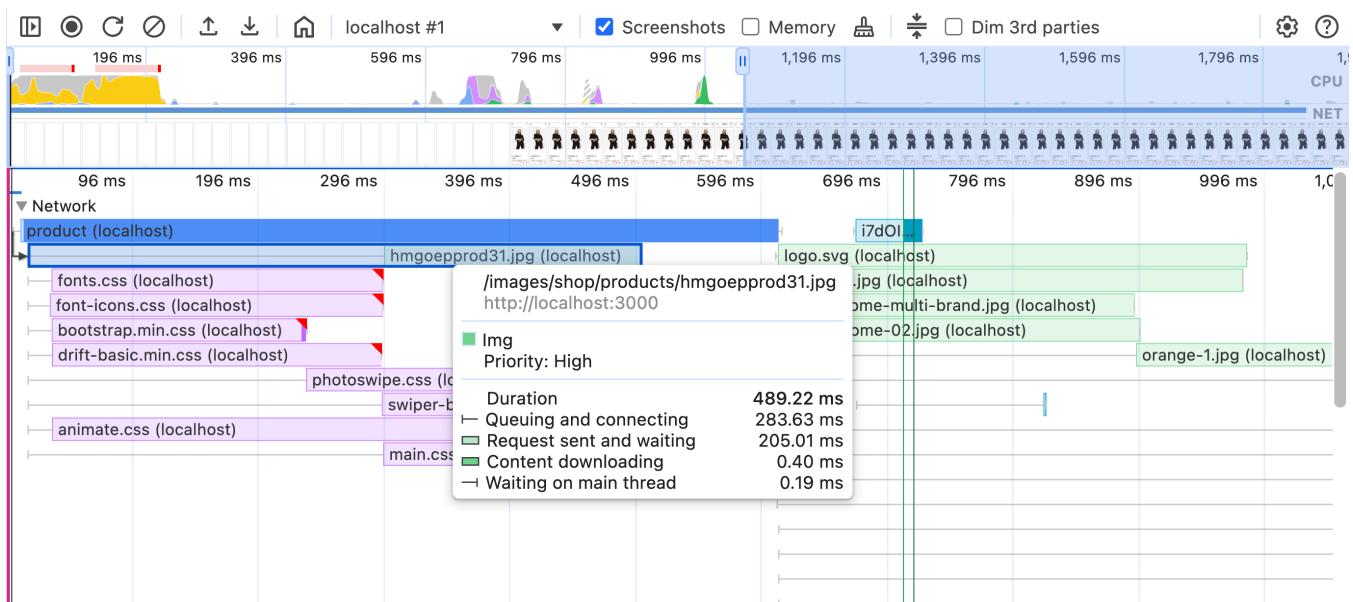
Field 75th percentile

Your local LCP value of **0.79 s** is good.LCP element [img.tf-image-zoom](#)

Step 6: Optimize LCP, add HTTP2

Re-run performance analysis to identify potential improvement areas.

So the LCP resource has a “High” priority, it’s discoverable in the initial HTML response, however the browser would not start loading it until render-blocking CSS files finish loading. Why?



Let's review the network tab and identify the protocol we use to load additional assets

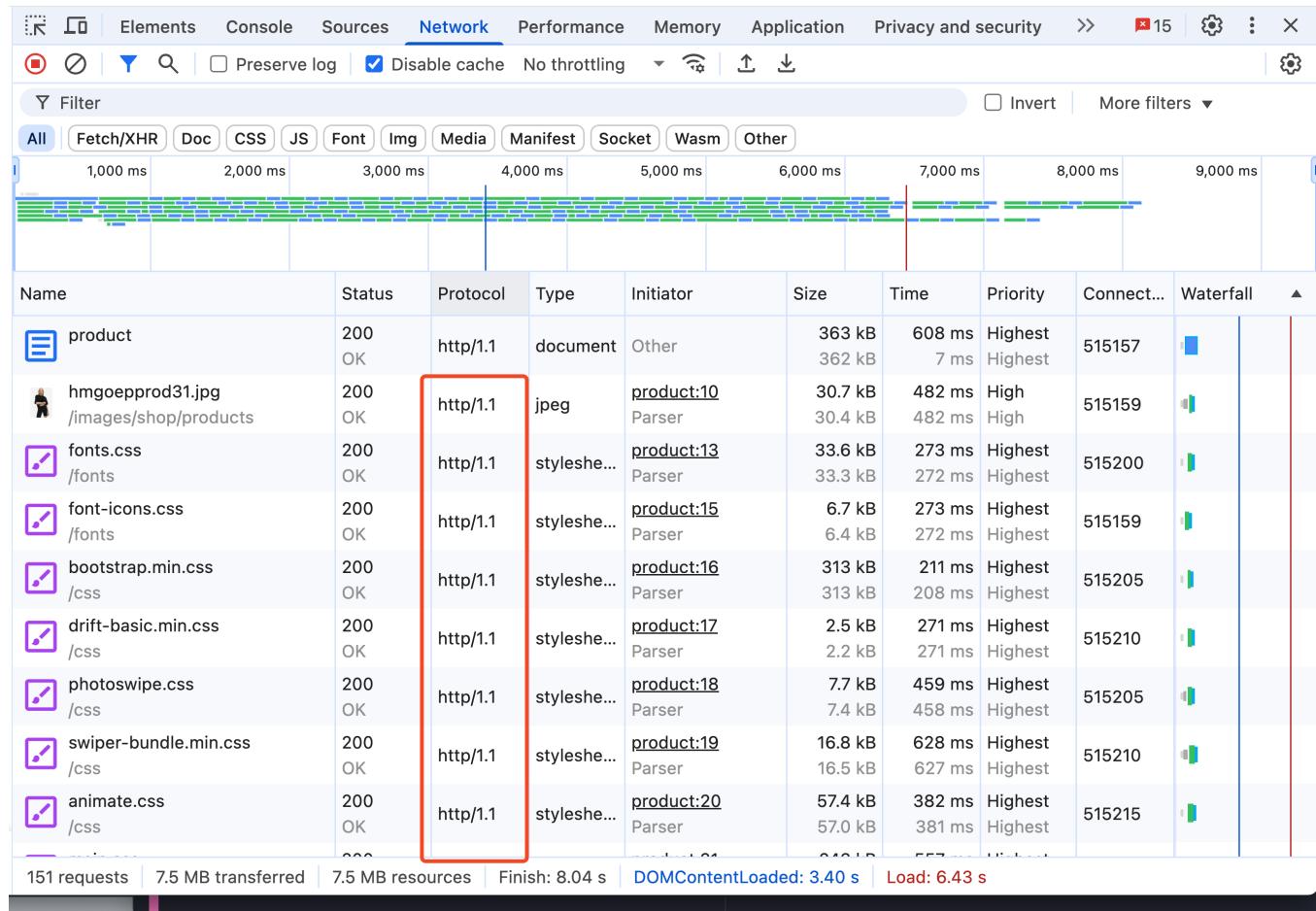
Right click on any column name in the network tab and click on the “Protocol” to add it to the table

Name	Status	Protocol
/images/shop/products	200 OK	http/1.
hmgoepprod3.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod4.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod5.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod6.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod7.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod8.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod9.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod10.jpg	200 OK	http/1.
/images/shop/products	200 OK	http/1.
hmgoepprod11.jpg	200 OK	http/1.

A context menu is open over the "Protocol" column header, with "Protocol" selected. The menu also includes options like Name, Status, Method, Type, Initiator, Size, Time, Priority, Connection ID, Waterfall, Sort By, and Reset Columns.

151 requests | 75 MR transferred | 75 MR resources | Finish: 8.01 s | DOMComplete

As you can see we are using HTTP1.1 to load additional resources, which causes leads to the limitation of <10 concurrent requests. Makes sense why the image load would not start soon!



Let's implement HTTP2

[https://github.com/kirillkuts/LCP-optimization-demo/commit/
de482380c3b3a3142590418f57f30242241b4249](https://github.com/kirillkuts/LCP-optimization-demo/commit/de482380c3b3a3142590418f57f30242241b4249)

Another ~100ms WIN. Great work optimizing from 5s to 0.6s!



- Disable JavaScript samples
- Enable advanced paint instrumentation (slow)
- Enable CSS selector stats (slow)

Local and field metrics

Largest Contentful Paint (LCP)

0.61 s

Local

Field 75th percentile

Your local LCP value of **0.61 s** is good.

LCP element