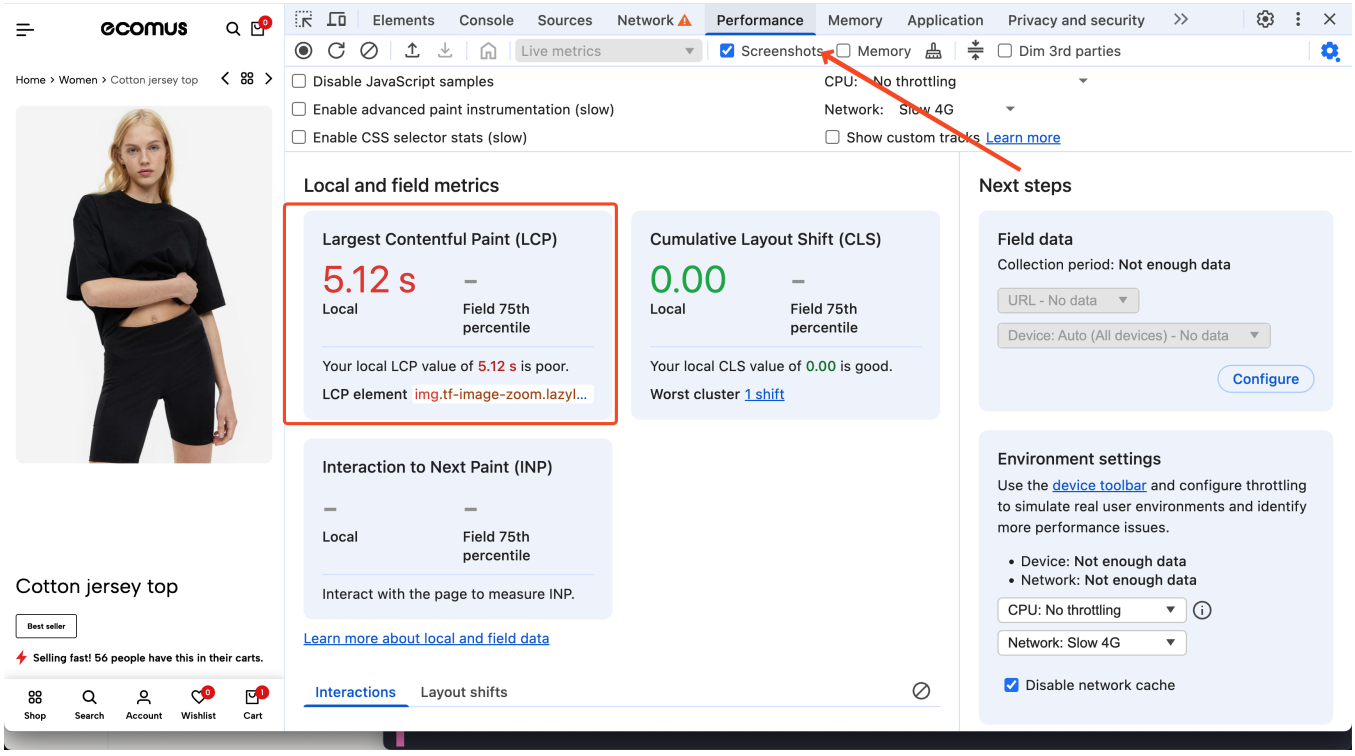
Module 3. Practice LCP optimization

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

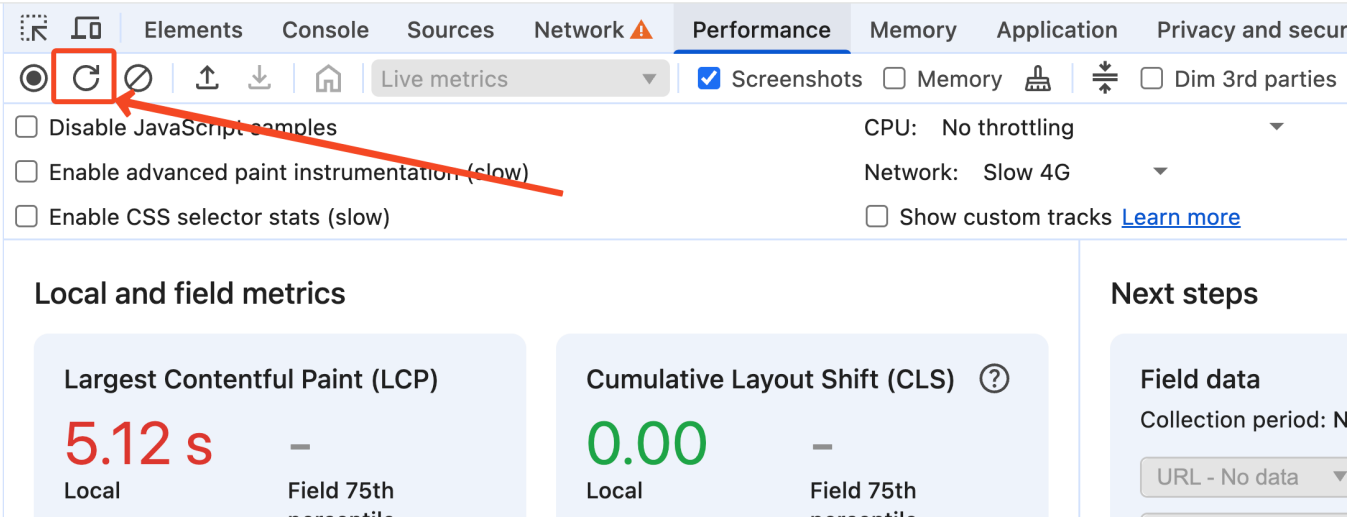
Project setup

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

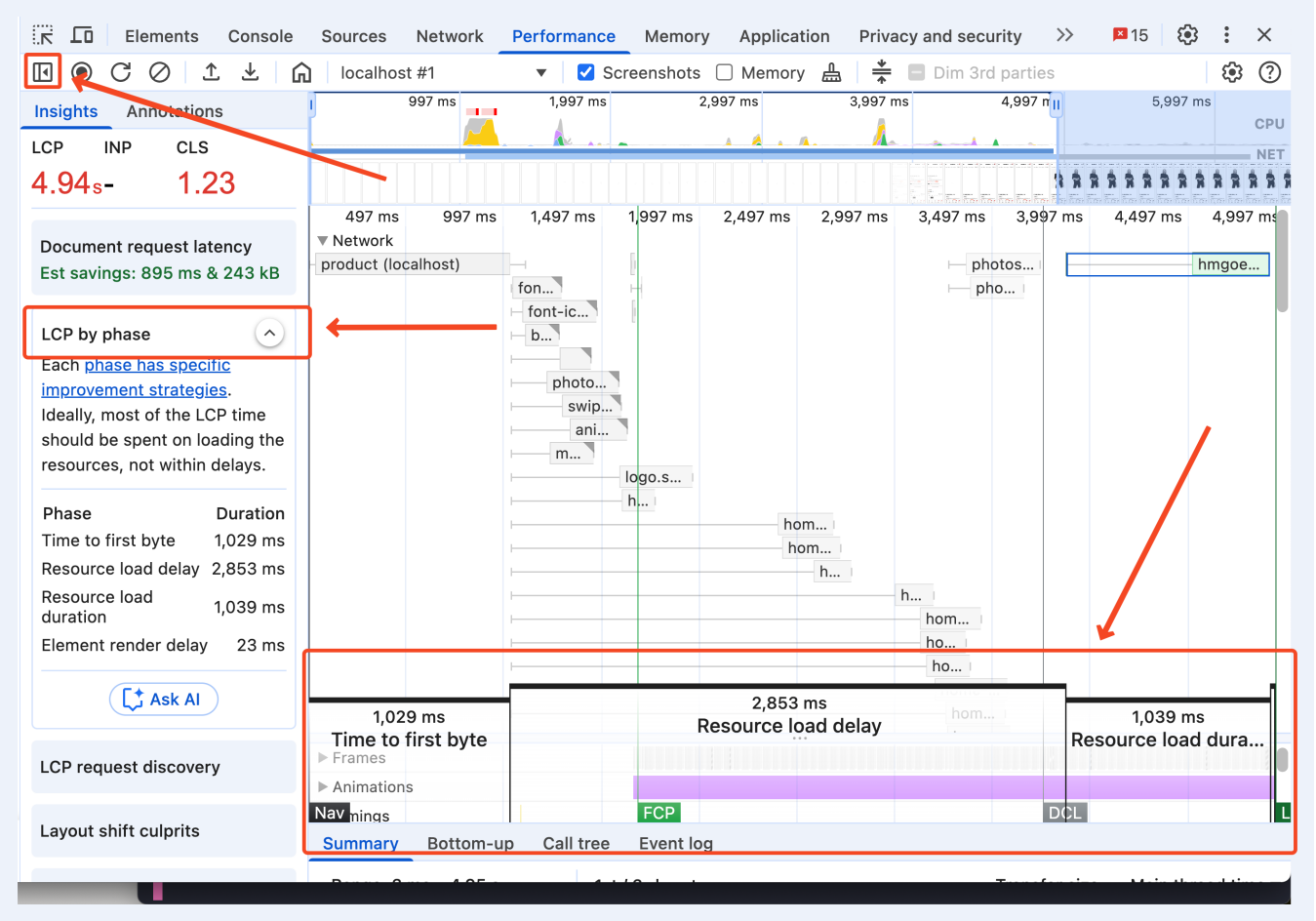
1. Navigate to `http://localhost:3000/product` in Chrome
2. Open DevTools, navigate to `Performance` tab
3. Observe the local LCP value



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



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

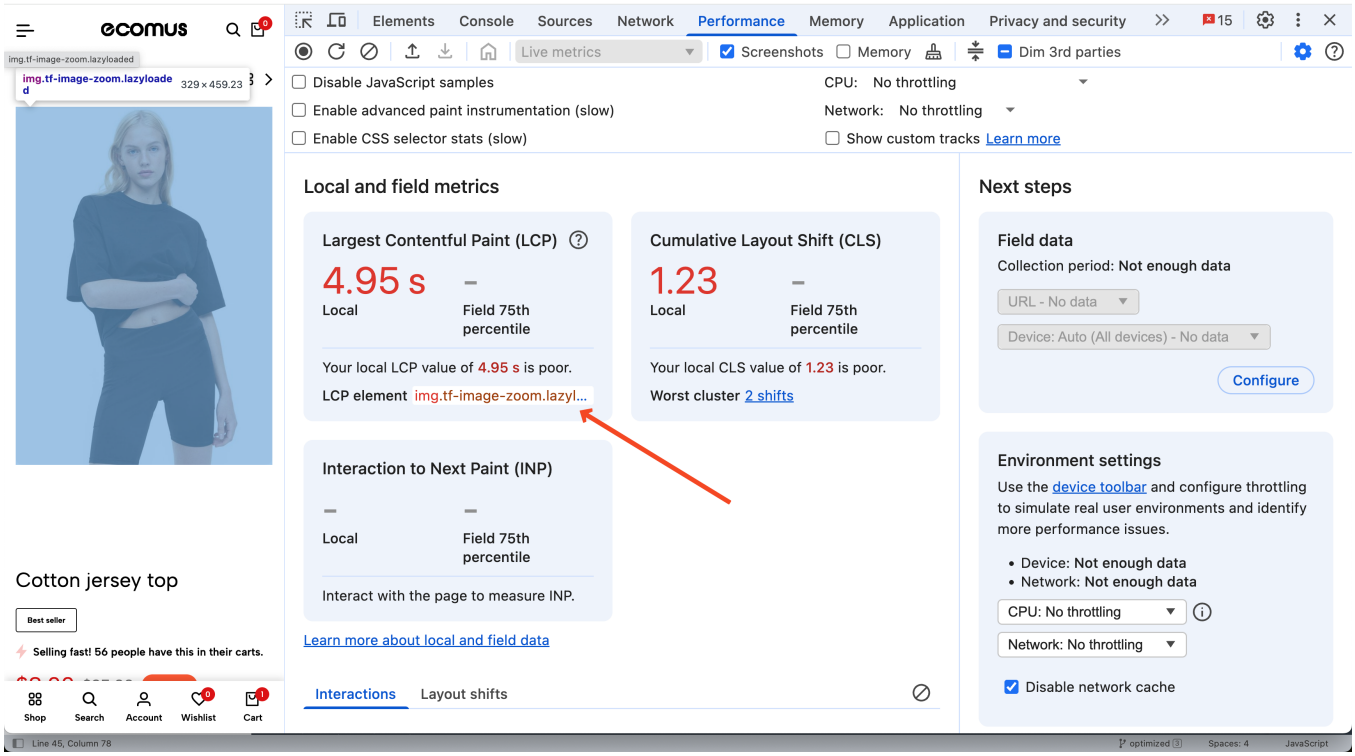
While there are plenty of changes that could be implemented I'd suggest taking the "heal where it bleeds most approach" - look for the biggest contributor to the LCP, in this case "Resource Load Delay"

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

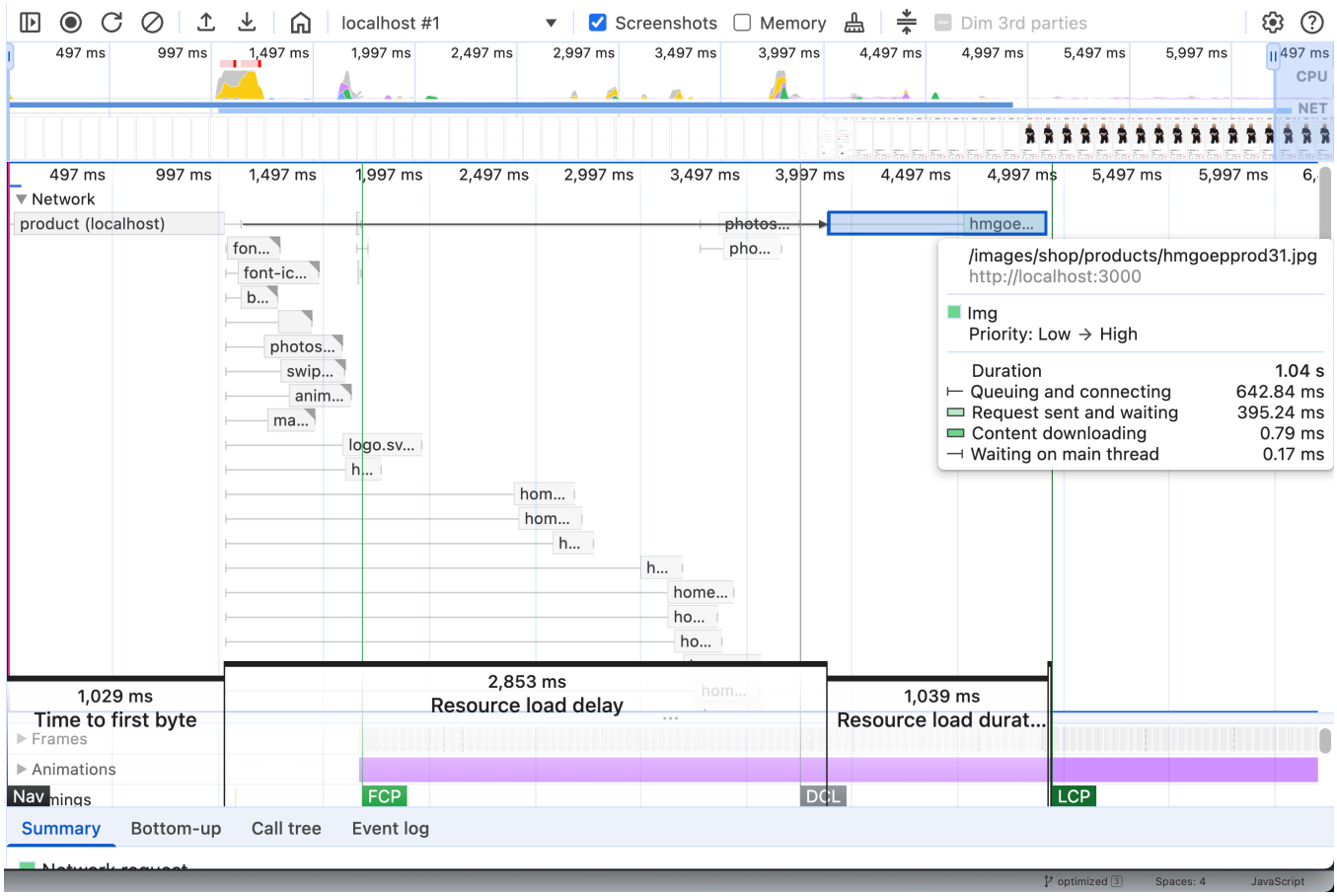
It's one of the most common LCP problems! And is usually caused by the fact that browser prioritizes render-blocking resources (CSS, JS) over images, attempting to optimize for the FCP by default.

Optimization steps

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



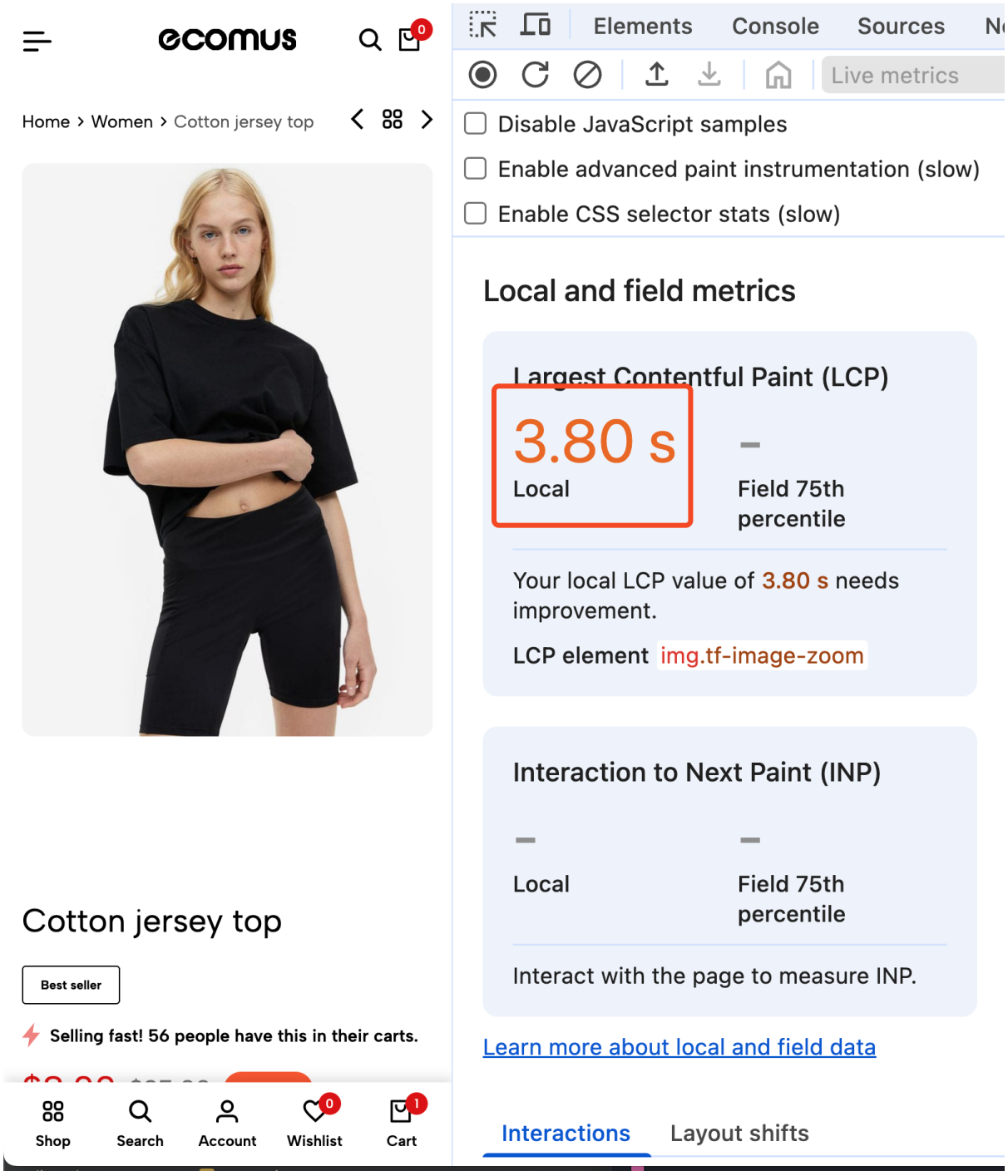
2. Identify the image path by clicking on the image element (`images/shop/products/hmgoepprod31.jpg`)
3. Run the performance analysis again and search for the image path request



4. Optimize the image loading by:
    - removing lazy loading
    - adding `fetchpriority="high"` attribute

Horray! We went from 5s to 3.8s!

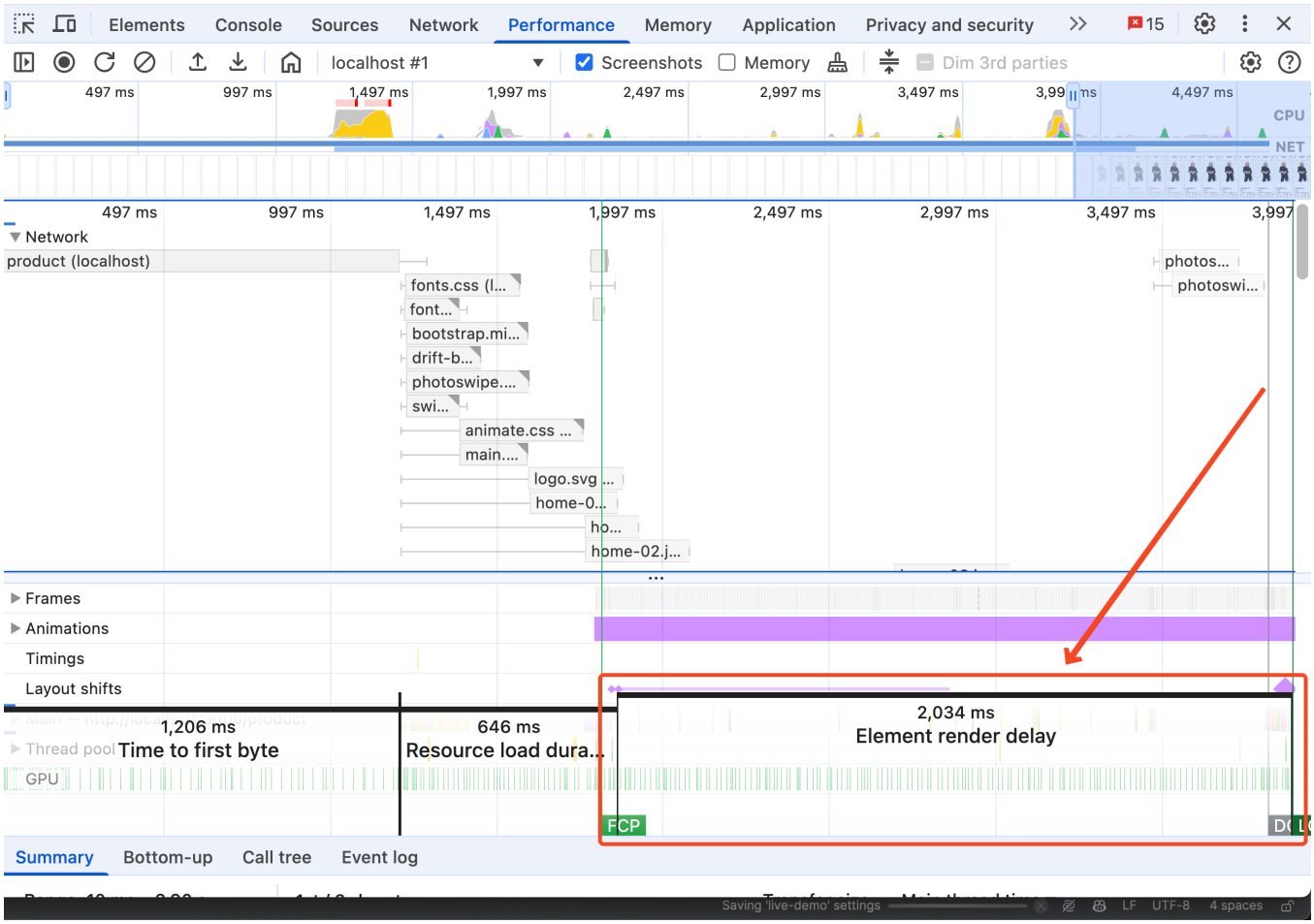
> You can find the code change here https://github.com/kirillkuts/LCP-optimization-demo/commit/f370b970bb959b849f62c6441a97922bba0a7146



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



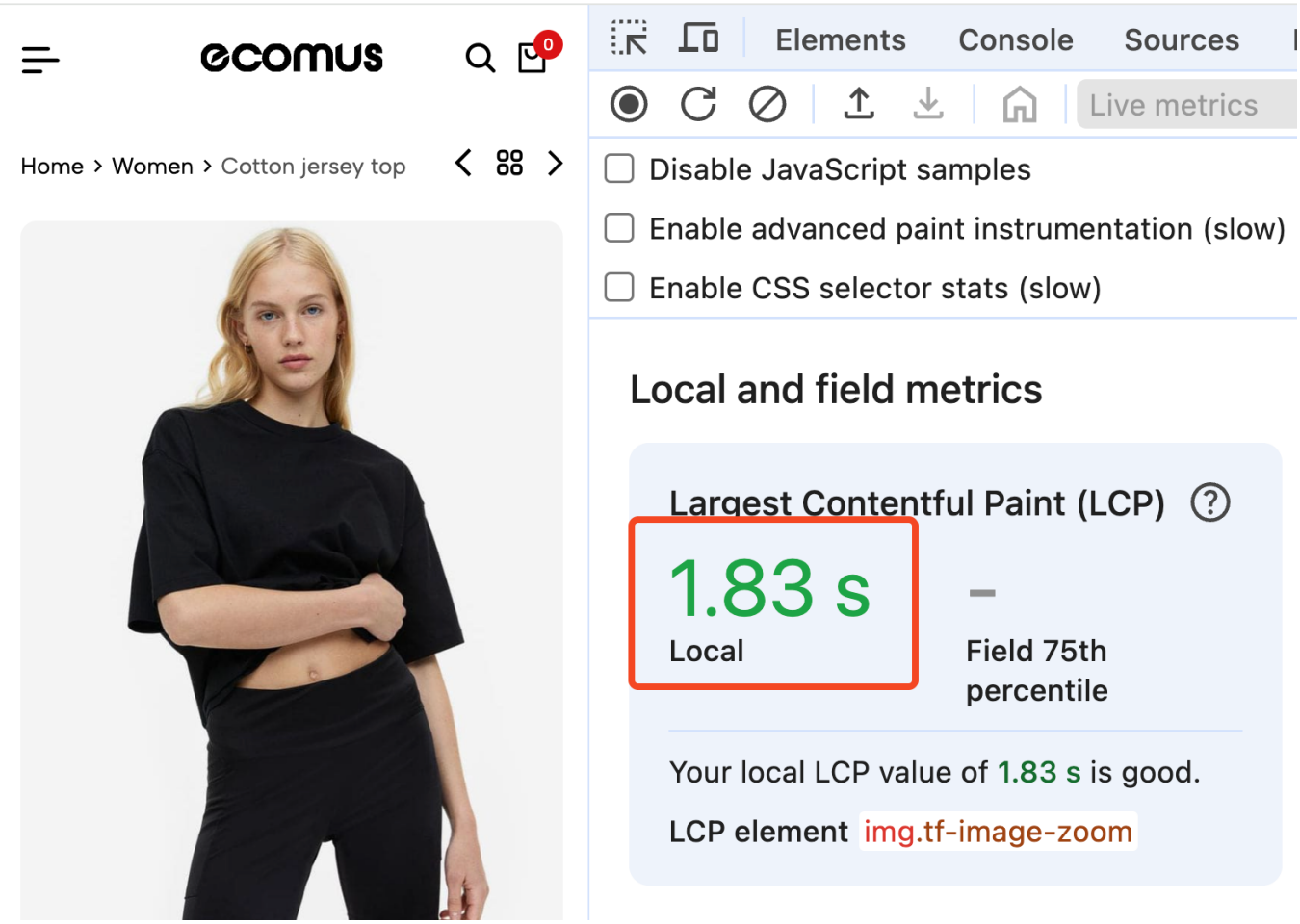
3. Element render delay optimization can take different form. To eliminate some common problems look for the following symptoms:
    - **Have all render-blocking resources have been loaded?** In our case the FCP (first contentful paint) happened way before so that's not a problem
    - **Is there a long-running 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/66f886c4dad85b9d995bfb94cca19b20b57b03a0

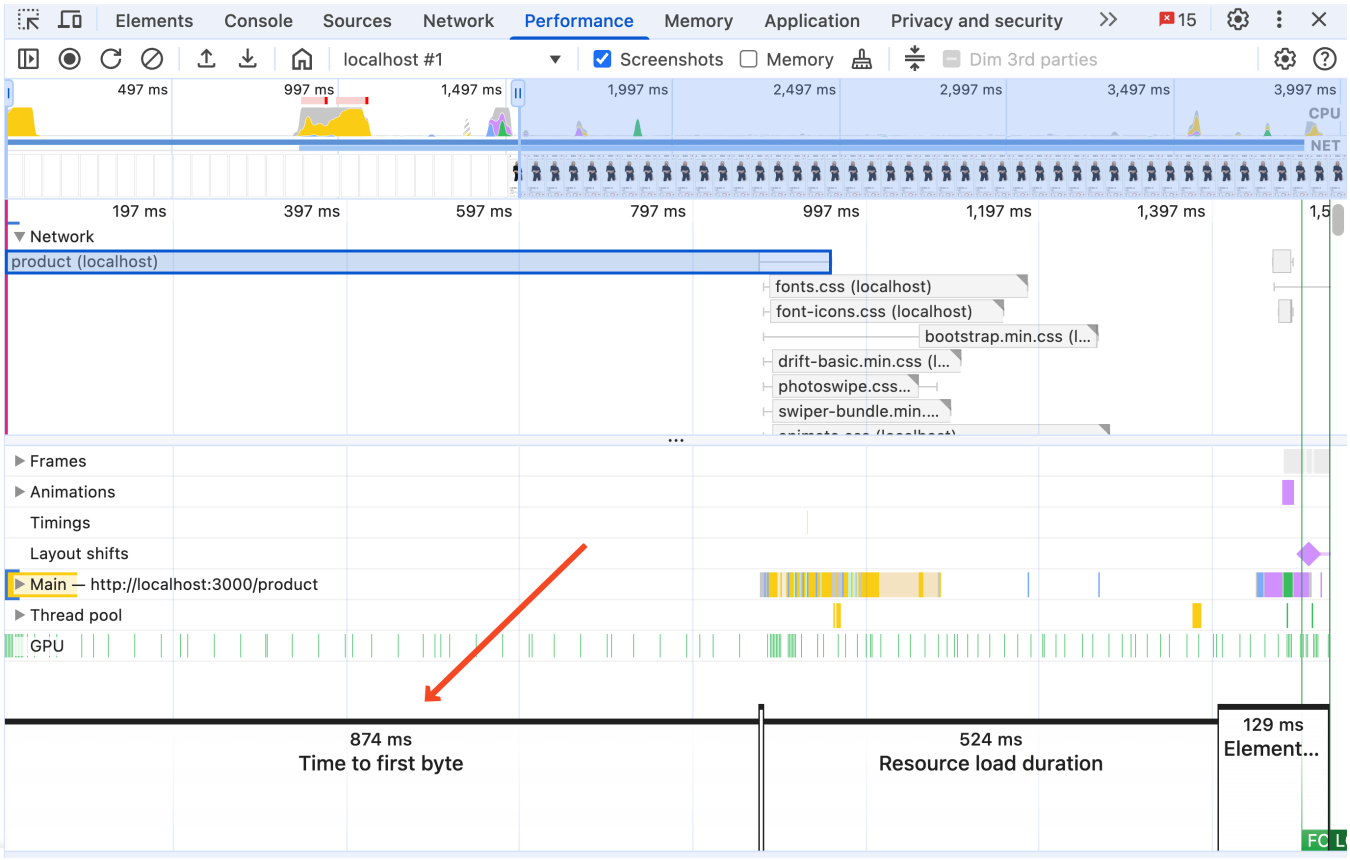
Supplementary reading: https://web.dev/articles/optimize-lcp\#2_eliminate_element_render_delay



Step 3: Optimize TTFB, DB calls

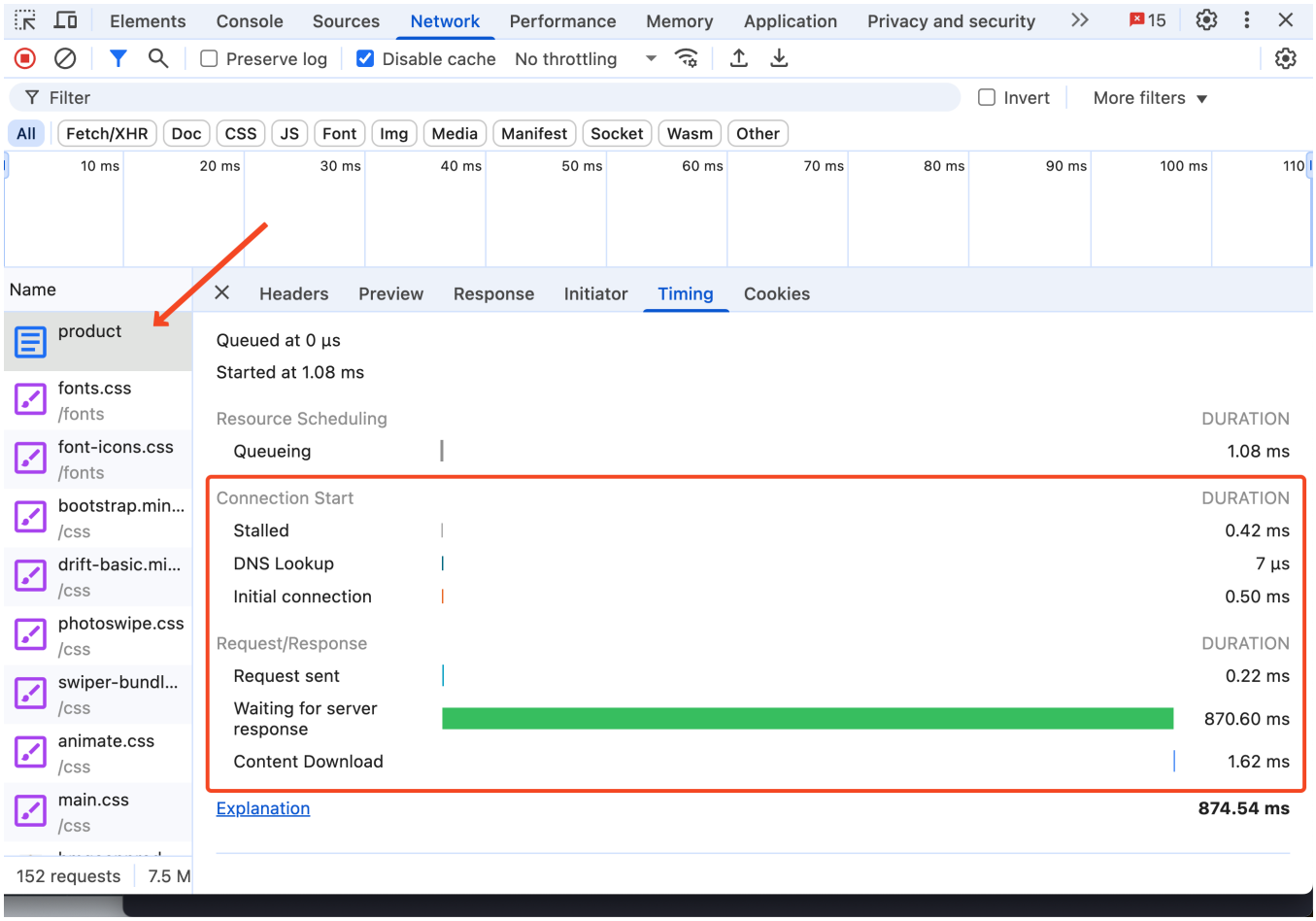
We've got pretty good results so far going from 5s to 1.8s , however we can do even better!

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



2. 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?)



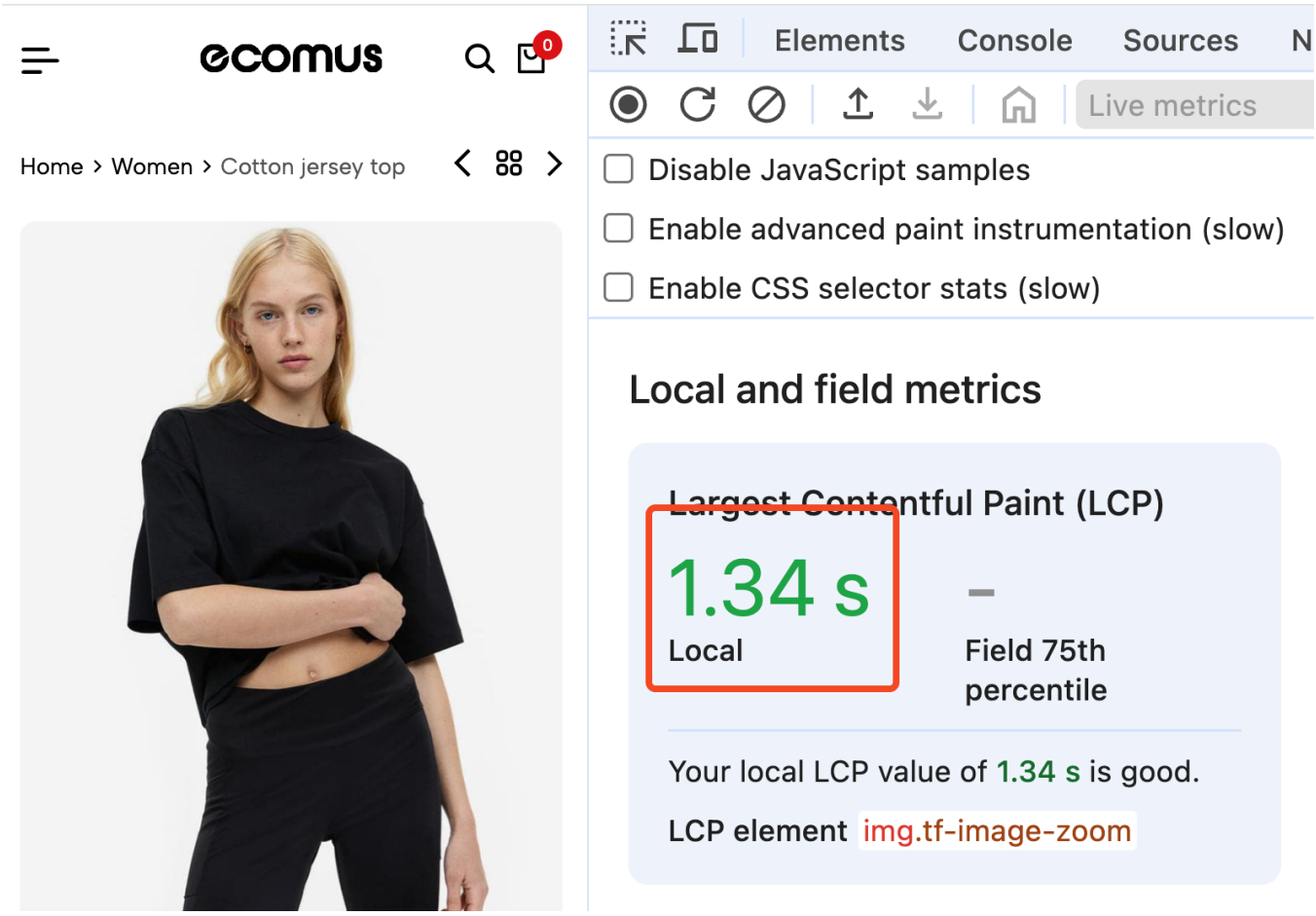
3. Let's review the codebase and identify potential bottlenecks

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

```
    const id = req.params.id;
    const productDetails = await getProductDetails(id);

    let output = html.replace('{{ gallery }}', `
        <div dir="ltr" class="swiper tf-product-media-main" id="gallery-swiper-started">
            <div class="swiper-wrapper">
                ${productDetails.images.map((img, i) => `
                    <div class="swiper-slide" data-color="beige">
                        <a href="${img}" target="_blank" class="item" data-pswp-width="770px" data-pswp-height="
                            <img
                                class="tf-image-zoom ${i > 0 ? 'lazyload' : ''}"
                                data-zoom="${img}"
                                ${
                                    i === 0 ?
                                        `
                                            src="${img}" fetchpriority="high" onload="hidePreloader()"
                                        ` :
                                        `data-src="${img}"`
                                }
                                alt="">
                        </a>
                    </div>
                `).join('')}
            </div>
        </div>
    `);

    output = output.replace('{{ priceOld }}', `$${productDetails.priceOld}`);
    output = output.replace('{{ priceNew }}', `$${productDetails.priceNew}`);
    output = output.replace('{{ title }}', `${productDetails.title}`);

    const productStock = await getProductStock(id);

    output = output.replace('{{ sizes }}', `
        <div class="variant-picker-values">
            ${Object.keys(productStock).map(option => `
                <input type="radio" name="size1" id="values-${option}" checked>
                <label class="style-text size-btn" for="values-${option}" data-value="S">
                    <p>${option.toUpperCase()}</p>
                </label>
            `).join('')}
        </div>
    `);
```

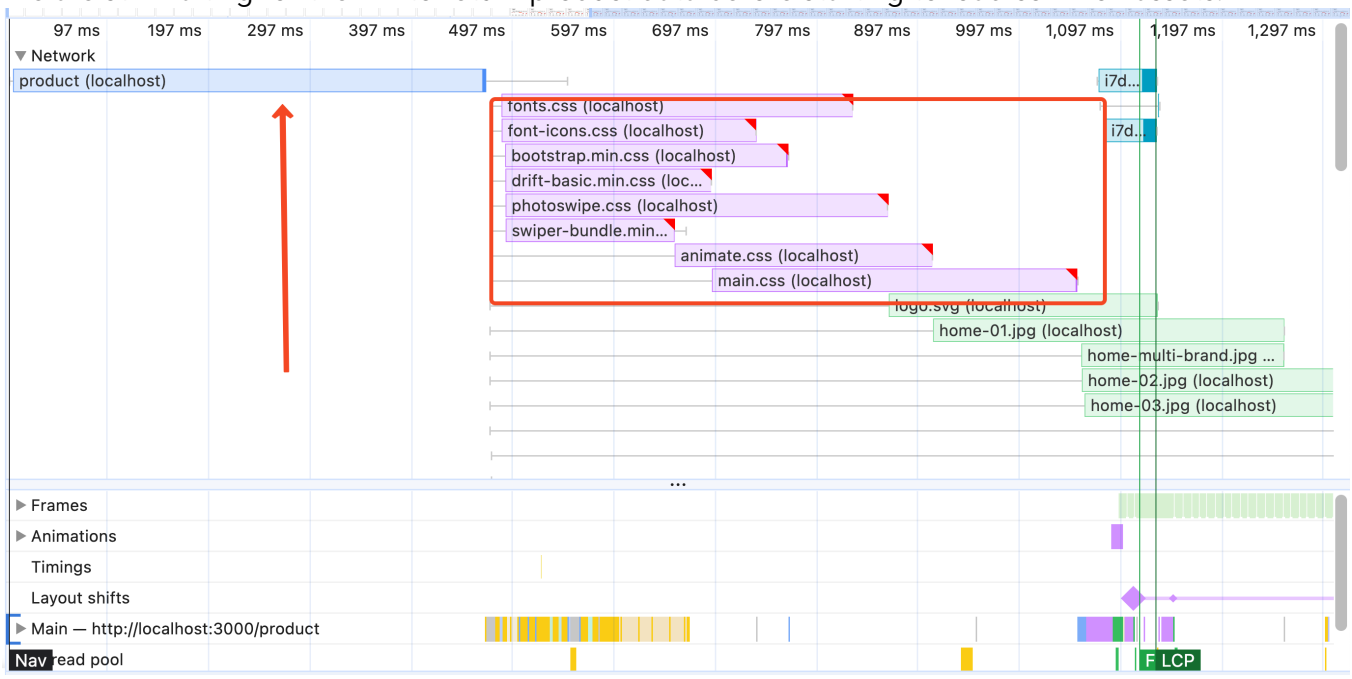
5. Let's parallelize those API calls
   https://github.com/kirillkuts/LCP-optimization-demo/commit/0a46c707143536a8482ec41fe51581cca4632143

One more 300ms win! Congrats!



Step 4: Optimize TTFB, early `head`

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

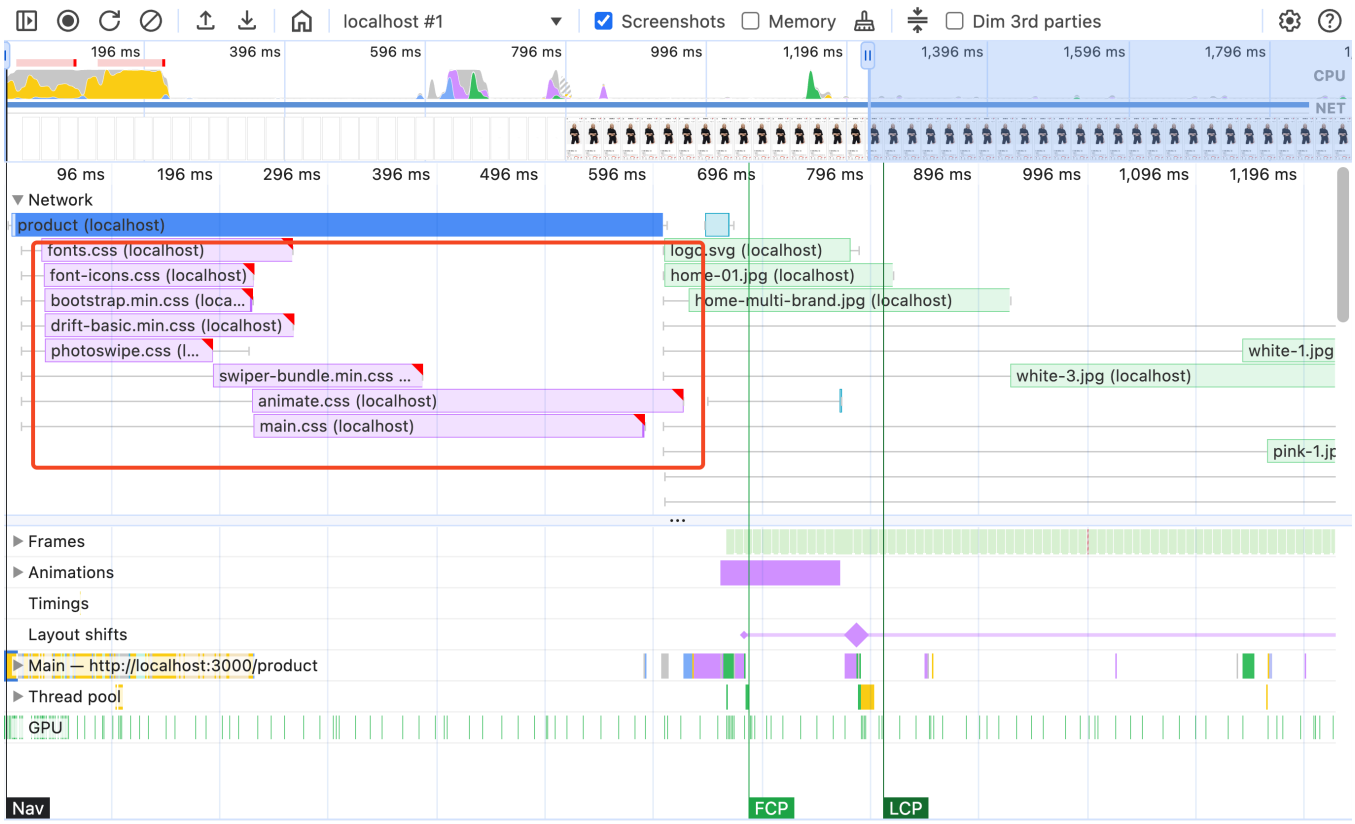


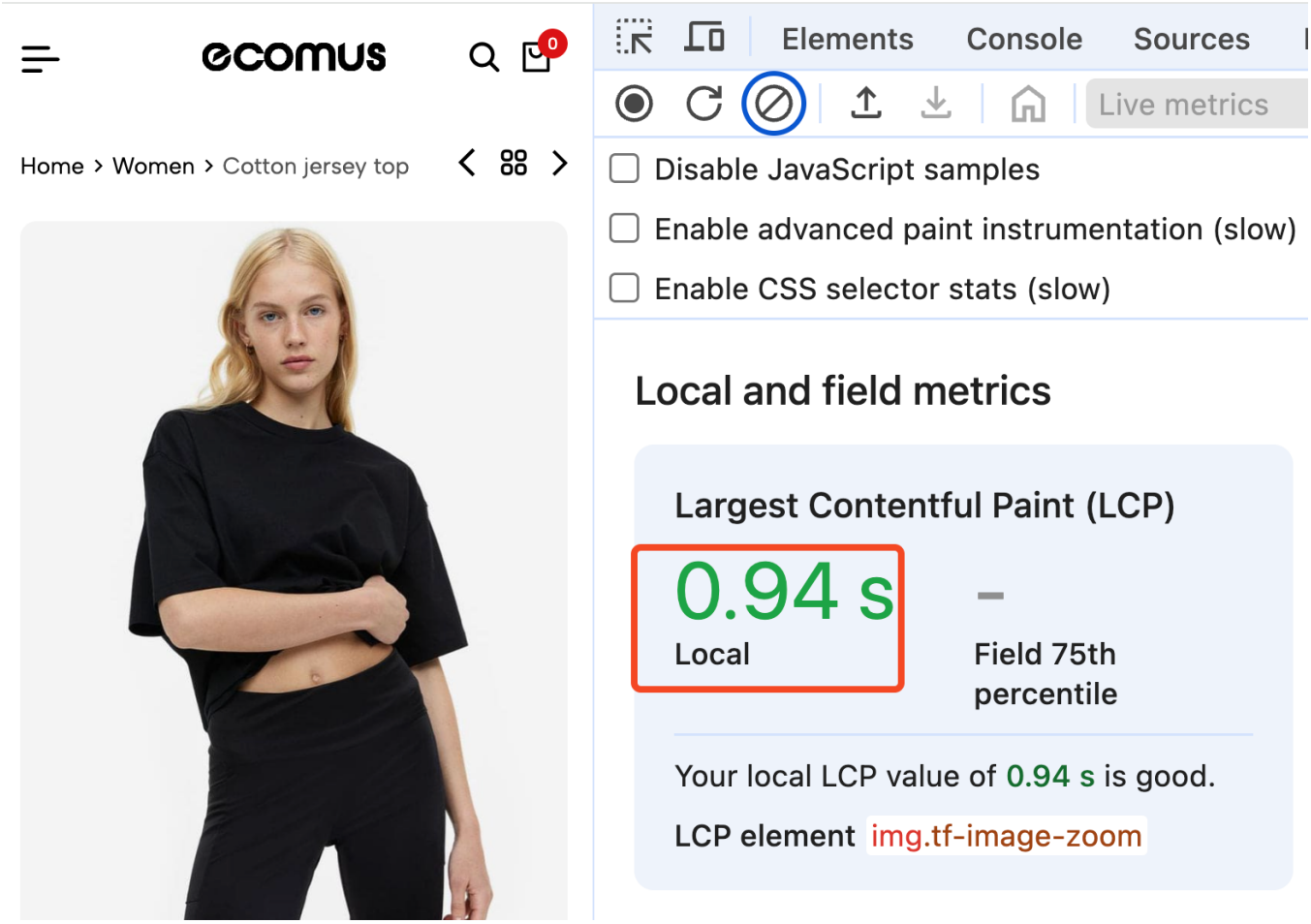
Let's optimize this by ensuring the `head` section is returned before we make the DB call, and the rest

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**!





Step 5: Optimize TTFB, cache LCP src

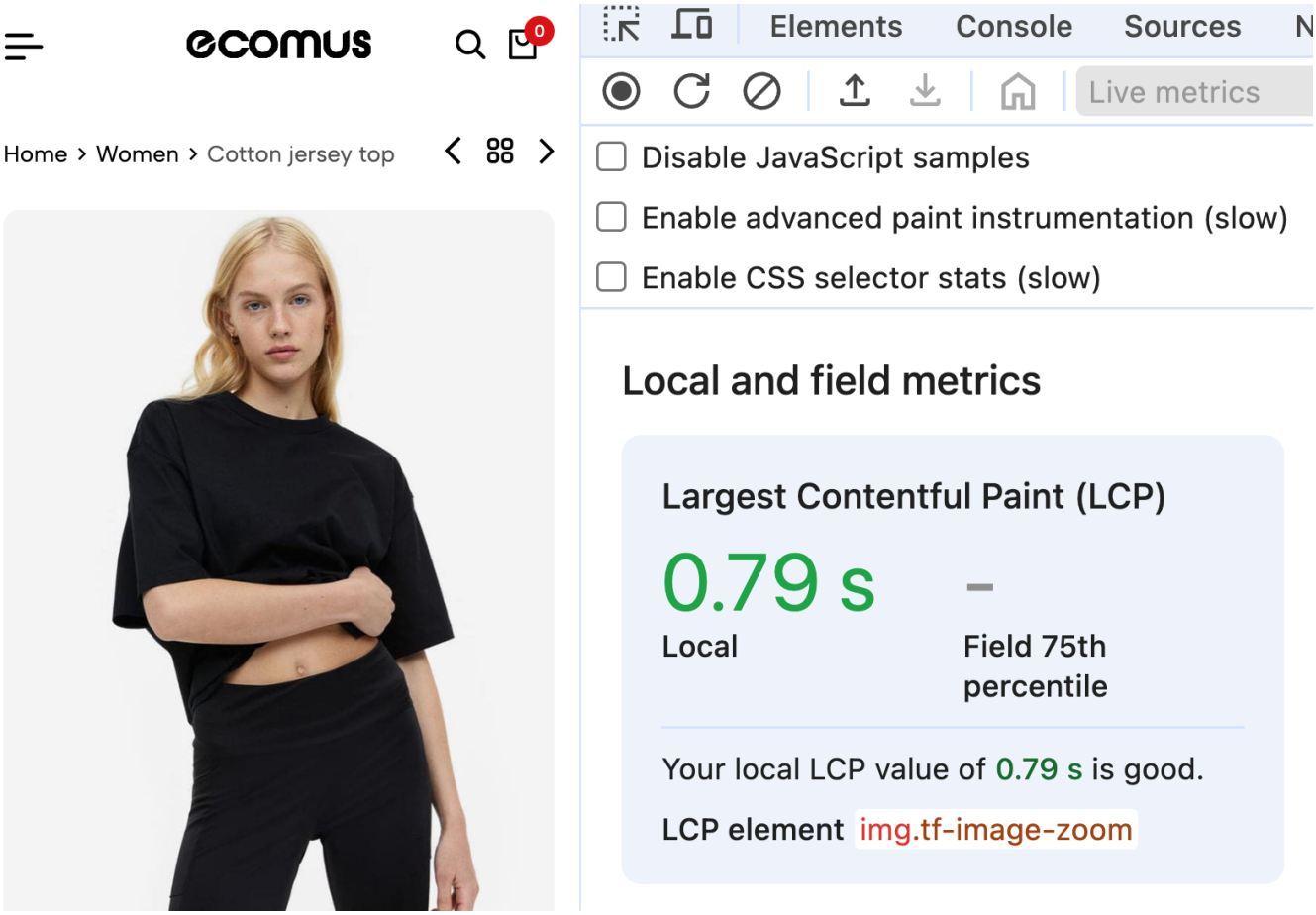
Our current setup awaits for the DB to return product data before adding images to the page. However, the DB call takes a long (300-500ms) time. Let's implement a cache layer that would keep LCP link in-memory

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

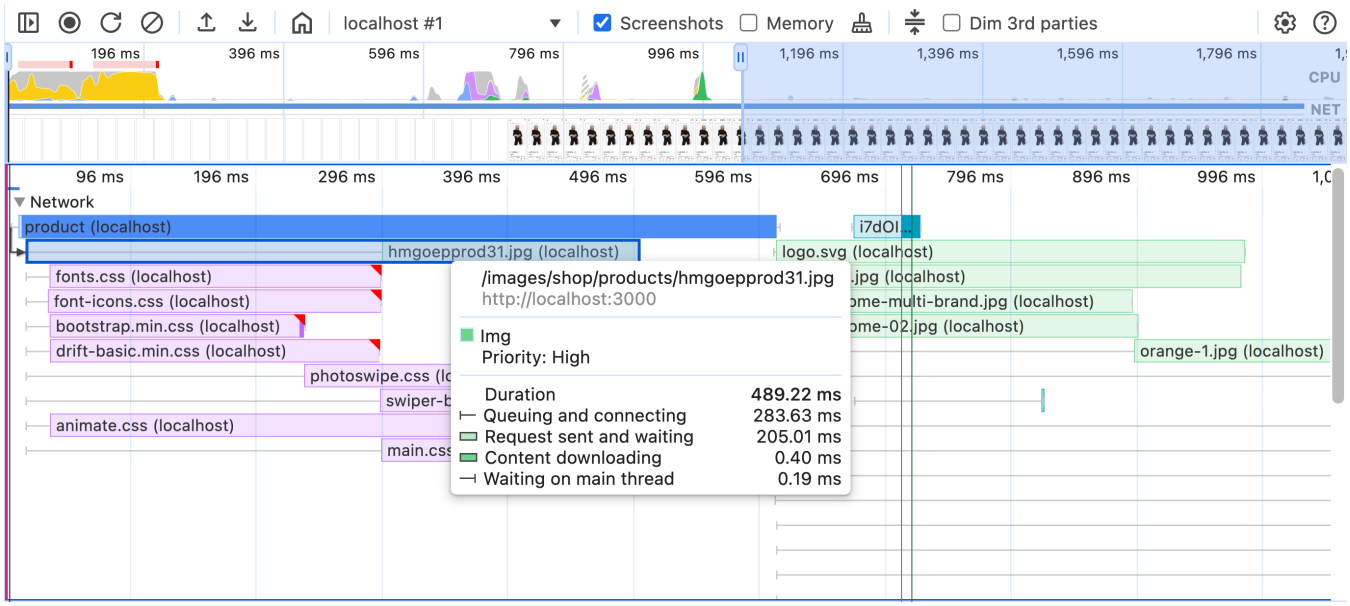
Another ~100ms WIN!



Step 6: Optimize LCP, add HTTP2

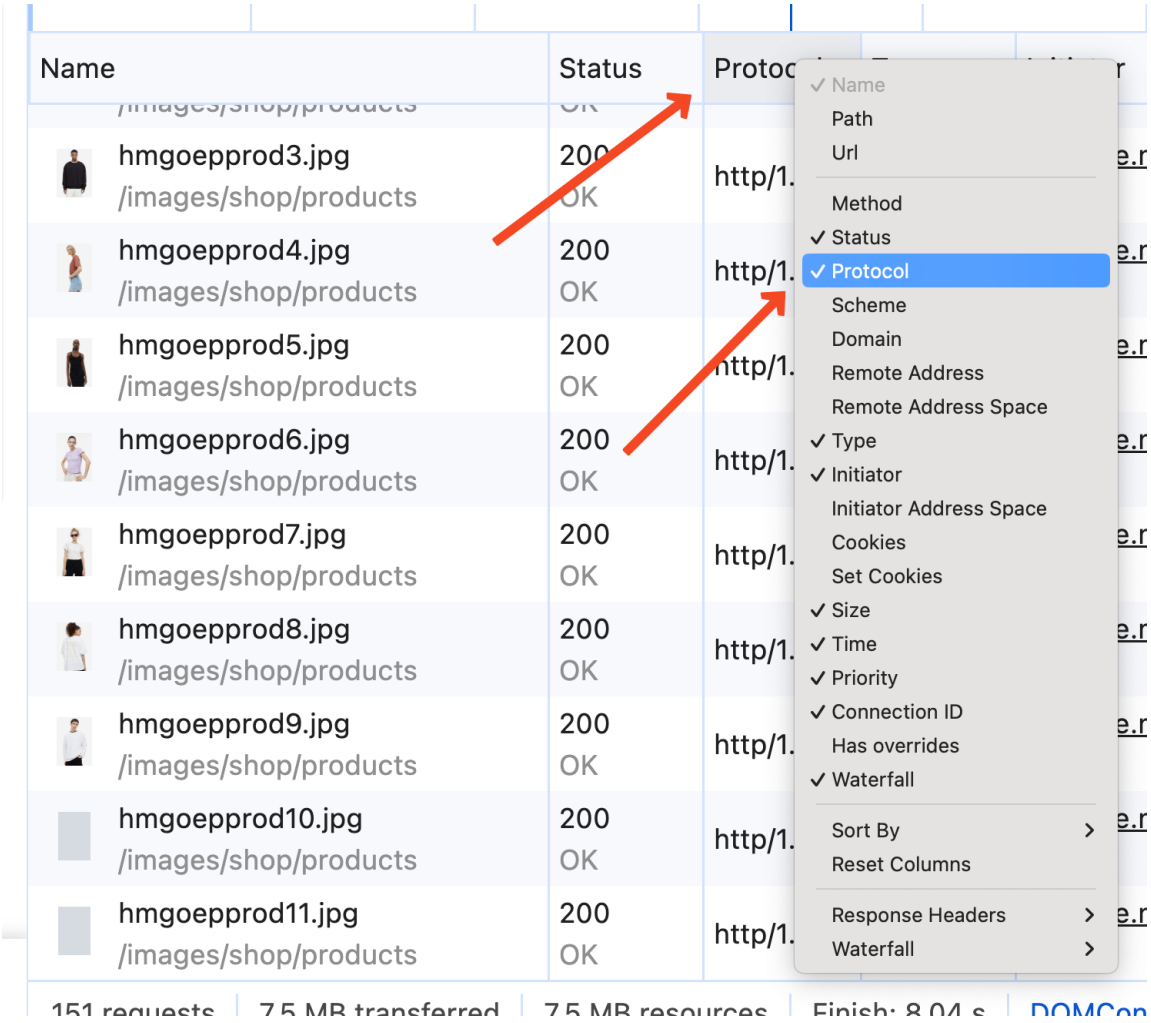
Re-run performance analysis to identify potential improvement areas.

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?

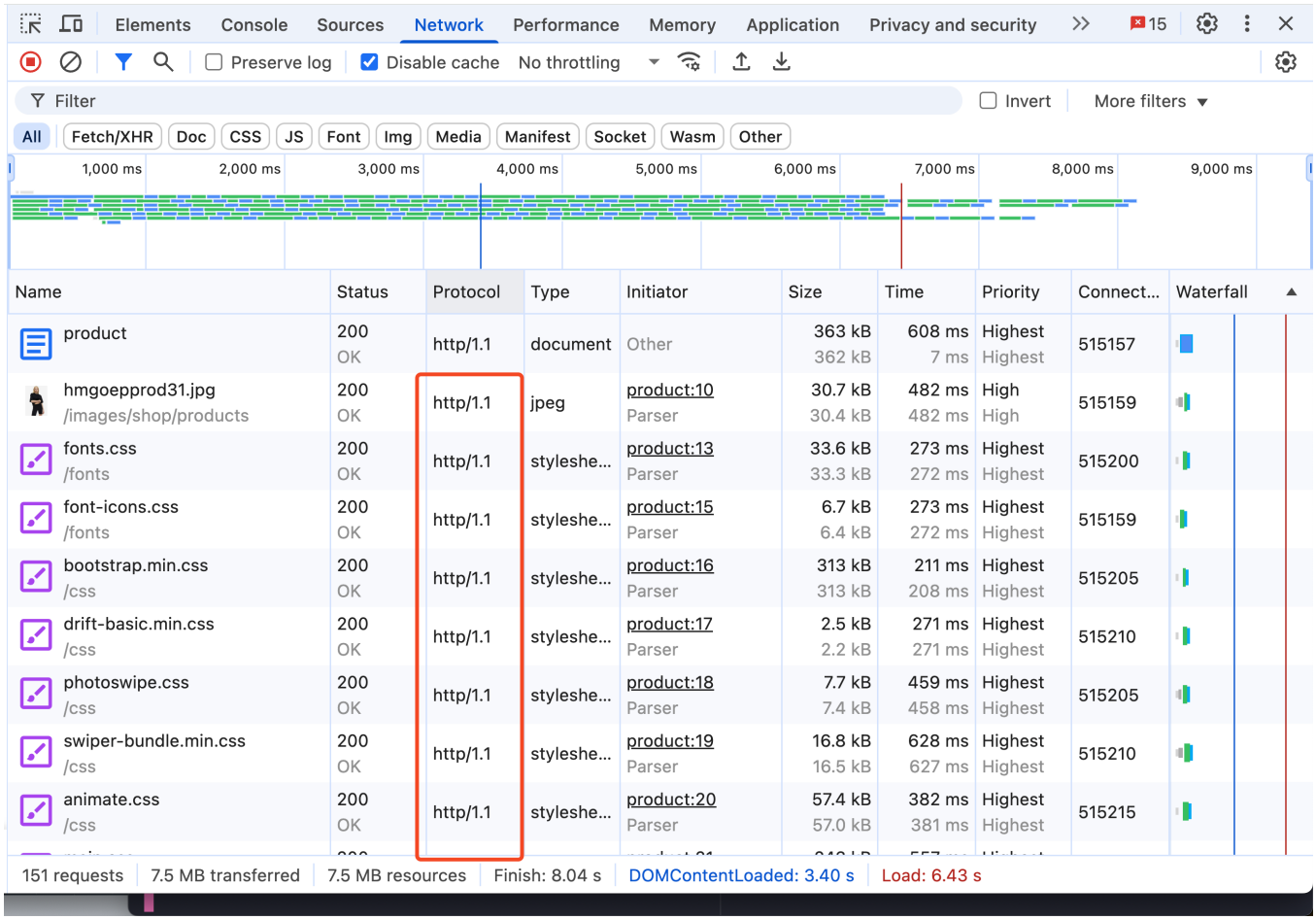


Let’s review the network tab and identify the proocol we use to load additional assets

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



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

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

ecomus

Home › Women › Cotton jersey top

Elements Console Sources Ne

Live metrics

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 img.tf-image-zoom