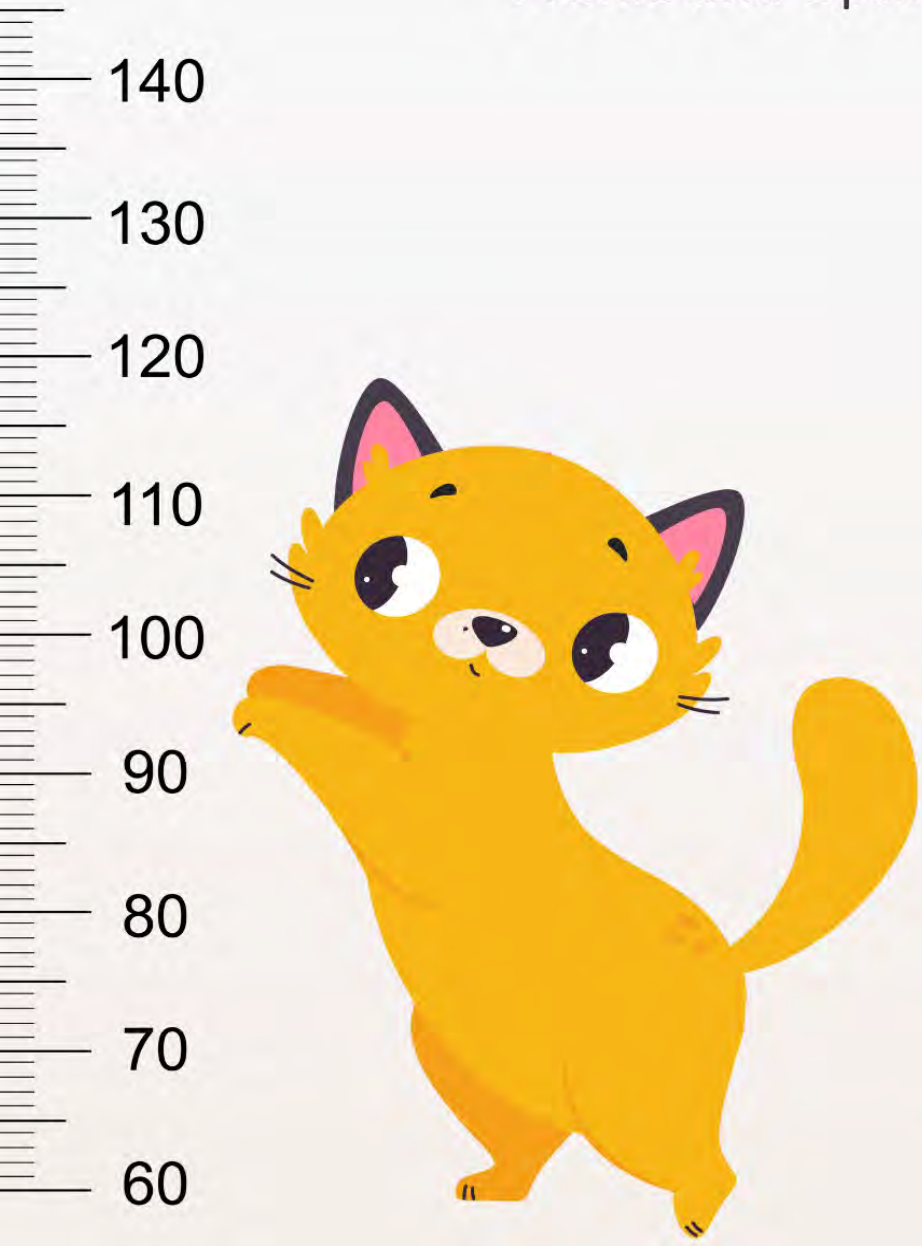
INVESTIGATE, UNDERSTAND, MEASURE

WEB PERFORMANCE FUNDAMENTALS

A Frontend Developer's Guide to Profile and Optimize **React** Web Apps



NADIA MAKAREVICH

Web Performance Fundamentals

A Frontend Developer's Guide to Profile and Optimize React Web Apps

by Nadia Makarevich

Illustrations and book cover design: Nadia Makarevich
Technical review: Andrew Grischenko

Table Of Contents

1. Hello and Welcome!

Performance has been my favorite topic in the last few years. Throughout my career as a software developer, what I've enjoyed most is playing detective: investigating tricky bugs or debugging some weird issues is my idea of fun. There is something incredibly satisfying about pulling a thread, following a lead, getting sidetracked a few times, but then discovering some hidden clue, and finally uncovering the treasure at the end.

Anything performance-related is like the ultimate investigation game. So many variables, all tangled together in a beautiful mess! Some of them influence the performance picture consistently, others only when other variables are present, under certain conditions, when the moon is aligned with certain constellations. What's not to love?

In this book, I want to show you how to play this game. Better yet, we will play it together. This book is not something you can use as documentation. Or a step-by-step guide for building React apps. Treat it as a journey of discovery.

I'm not going to tell you facts. At least not right away. Instead, we'll have an app that we'll be poking and prodding from different angles, then see what happens and measure the result. You won't need facts from me, you'll discover them all by yourself. I'll just be there to guide you and help you stay focused.

At least, this is the intention. Let me know if it worked or not 😉.

Who This Book Is For

The primary target audience for this book is middle-to-senior React developers who want to deepen their knowledge and are interested in learning about web performance. This book assumes that you've written an app or two already and that I don't need to explain such concepts as "state", "props", "components", or "conditional rendering".

If you're a seasoned frontend developer but completely new to React, you also might gain a lot. About two-thirds of the book focuses on fundamental knowledge applicable to any web app. However, be warned: all the code examples and the Study Project are written in React, so you'll need to be able to navigate through them effectively. A few chapters focus on React-specific APIs like Server Components or Suspense. And a few chapters, even though the concept is universal, cover aspects specific to React apps. Such as Server-Side Rendering (SSR) and how to implement it properly in React.

What the Book Is About

The book is about the fundamentals of web performance that are particularly relevant for React developers. This is not a book about how to write React code. There are plenty of books for that already. It's almost the opposite - it's about everything that is *not* React code. Everything that surrounds it and contributes to what we know as Performance.

In this book, you won't learn different state management techniques or how to split components. However, you will learn on a very deep level how to record and understand Performance Flame Graphs, different types of server and client rendering, how different data fetching strategies contribute to initial load performance, and so much more.

In the next chapter, **"Let's Talk About Performance"**, we'll look into why bother with performance in the first place, other than for personal entertainment. Plus, what the CrUX report is, why it's useful, and a few other resources to get you started with measuring performance.

In **"Intro to Initial Load Performance,"** we'll take a first look at the Chrome DevTools Performance and Lighthouse panels, understand what Initial Load is and how to measure it via metrics like FCP and LCP, and make our first few discoveries. While doing so, we'll investigate how different network conditions influence initial load, what a CDN is for, and learn the basics of Cache Control.

In the **"Client-Side Rendering and Flame Graphs"** chapter, we'll become proficient in recording and understanding the Performance Flame Graphs - you know, those colorful monstrosities that confuse everyone who sees them for the first (or tenth) time. While doing so, we'll understand what the Client-Side Rendering (CSR) pattern is and why it's important.

In the **"SPAs and Introducing INP"** chapter, we'll continue with CSR and its natural extension, the Single-Page Application (SPA). While doing so, we'll learn more about DevTools, touch on the next performance metric, INP, and look more into the Lighthouse report.

In the **"Intro to Rendering on the Server (SSR)"**, we'll investigate, you've guessed it, Server-Side Rendering. We'll implement our own pre-rendering strategy, understand the cost of introducing SSR to a previously CSR app, how SSR affects initial load metrics, what hydration is, and how to implement SSR properly in React code.

The **"Bundle Size and What to Do About It"** chapter focuses on the size of JavaScript we're producing and why it matters (or doesn't matter). In the process of this investigation, you'll learn about compression, which aspects of initial load the size of JavaScript can affect, become a master of bundle size analysis, touch on the difference between HTTP/1 and HTTP/2 & 3, learn about preloading, code splitting, tree shaking, different module formats, transitive dependencies, and probably more. It's a huge chapter!

In the **"Intro to Lazy Loading and Suspense"**, we'll continue our investigation into the bundle size and talk about, no surprise here, lazy loading and Suspense. How to implement them properly, and as always, what the consequences are if you don't.

In the **"Advanced Lazy Loading"**, we're still going to talk about lazy loading! You'd be surprised at how complicated and nuanced the topic is. But as a result, we'll know the most important concepts behind frontend frameworks like Next.js, React Router, or Tanstack, why we need code splitting per route, why all frameworks give you a `Link` component, how to do preloading, how SSR comes into play with preloading, and even migrate our Study Project to the Tanstack framework and measure the results.

The **"Data Fetching and React Server Components"** chapter finally leads us away from bundles. We're going to talk about data fetching: on the client, on the server, and in between. How to implement and analyze client-side data fetching, including where and how to perform data prefetching, what happens if data is fetched with Server-Side Rendering (SSR) enabled, and the associated costs. This leads us to React Server Components - what problem they solve, how they solve it, and how to make sense of them generally. Plus, we'll throw Streaming into the mix and polish everything by trying out Next.js in addition to Tanstack.

Tired of initial load performance? In the **"Interaction Performance"** chapter, we'll finally move away from it and focus on interactions. How to navigate Chrome DevTools when it comes to interactions, what a Long Task is and how to fight it by yielding to main, how it's relevant to React developers, and what React DevTools have to offer here.

The **"Getting Rid of Unnecessary Re-renders"** chapter is the most React-y chapter here and focuses entirely on the React API. We're going to improve interaction performance here by fighting and winning the battle against sneaky unnecessary re-renders. All the classics: moving state around, passing components as children, Context and props drilling, and of course, memoization.

Does everything about React memoization make your blood boil? In the **"React Compiler"** chapter, we'll explore what the new, still experimental at the time of writing this book, tool has to offer for solving it. And is it worth it?

Hope that's enough content to keep you busy for a while! Now let's set up the development environment and the Study Project, and let's get started.

Setting Up the Study Project

You need to ensure that you have Node[1], npm[2], and Git[3] installed on your computer.

Clone the Study Project from here: https://github.com/developerway/web-perf-fundamentals[4] and install all dependencies via `npm install`.

Technically, you're good to go! The Study Project is structured the following way:

- `frontend` folder - this is where all the shareable code, like buttons, modals, and common pages, lives.
- `src` folder - this is for the chapter-specific code.

Each chapter with code examples has its own folder inside `src`, with all the necessary setup already done.

Every chapter-specific project is organized as a "workspace". Which in this case just means that it has a `package.json` file inside, and you can run all the scripts declared inside that `package.json` from the root via `npm run [script-name] --workspace=[chapter-name]`.

If you look inside the `src/chapter5-spa-and-inp/simple-frontend/package.json` file, for example, you'll see this:

```json
{
  "name": "chapter5-simple-frontend",
  "scripts": {
    "dev": "vite",
    "build": "rm -rf dist && vite build",
    "start": "tsx watch server.ts"
  }
}
```

That means that from the root of the repository, you can run:

```
npm run dev --workspace=chapter5-simple-frontend
```

or:

```
npm run build --workspace=chapter5-simple-frontend
```

And so on, you get the idea.

You'll also need to install the Chrome browser[5]. We're going to use its DevTools to measure everything throughout the book.

Also, at some point, you'll need the React DevTools[6] Chrome plugin. We'll use it when we get to investigating interaction performance.

That's it, you're ready to put your Sherlock hat on!

2. Let's Talk About Performance

Before investigating anything, we first need to answer "why". It surely is fun, but no business owner will approve months-long work just because developers want to have some fun. Is there a practical reason for it?

Why Performance Matters

There is actually a lot of research into the topic of whether improving web performance has a visible impact on business metrics.

There is, for example, research by Deloitte[7] where they analyzed over a period of 4 weeks mobile data from various brands across Europe and the US. The conclusion: even a 100 ms improvement in site speed results in improved funnel progression, improved page views, bounce rate, and conversion rate. The full report[8] has over 50 pages and lots and lots of numbers.

There are also hundreds of case studies on Google's web performance website: https://web.dev/case-studies. For example:

- Vodafone case study[9] highlights that a 31% improvement in LCP led to an 8% increase in total sales, a 15% improvement in lead-to-visit rate, and an 11% improvement in the cart-to-visit rate.
- QuintoAndar, the largest housing platform in Brazil, reduced the INP of their website by 80% and saw an increase in conversions by 36%[10].
- Disney+ Hotstar[11] reduced INP by 61% and managed to increase weekly card views by 100% as a result.

Shopify invests a lot of time and energy into performance[12], and just last year confirmed that performance[13] still matters, even in 2024.

Also, https://wpostats.com/[14] has a good collection of case studies, starting from 2016 up to 2023.

Hope it's convincing enough?

How to Start Measuring: CrUX

After you decide that performance should be your next focus, the next question is where to start.

If you've never measured anything performance-related on real users, then the very first stop should be the Chrome User Experience Report (CrUX)[15]. It's basically the performance data of the entire internet gathered by Google. It's collected via Chrome from all eligible users[16] that visit your public website and is used by Google Search to inform how the website is ranked by the search.

The data is available in various formats: as a weekly visualization[17], as a dashboard[18], BigQuery[19], and even a developer API[20].

This data and these tools are invaluable. They give you a quick overview of the most important performance metrics for your website and the historical trends for each of the Core Web Vitals metrics. With them, you can see the results of your performance improvement initiatives at zero cost. Plus, since the data is public, it's always useful to compare how your website is doing vs your competitors 😏.

However, this data is not perfect. First, it measures only Core Web Vitals - a set of metrics defined by Google itself. While it's already useful as-is, at some point, you'd want to have more nuanced measurements that are tied more to your product specifics.

The second big downside is that the data is extracted from Chrome only, and only from eligible users. After all, it's the *Chrome* UX Report. That means some of your potential users are excluded.

And lastly, it's only for public websites. If you're hiding your functionality behind authentication, this report is useless to you.

Next Step: Real User Monitoring

If the CrUX report is not enough, the next step is Real User Monitoring (RUM). This step is crucial. Measuring the real customers is the only way to identify whether you *actually* have performance problems worth solving. And where they are, of course.

For this, you need two parts: something that extracts performance-related analytics from your website when it's live and sends it somewhere. And then a system that receives this data and transforms it into valuable insights and visually appealing dashboards.

Which system you're going to use for receiving the data is up to you, your requirements, and your budget. There are so many options, from Google Analytics[21] to Sentry[22] to DataDog[23] to every possible analytics and monitoring solution out there. You probably already have some form of monitoring or analytics solution anyway, so use that until you need something more advanced.

For extracting the data part, it really depends on which data and in which format you need. If you want to report the Core Web Vitals metrics, the same as the CrUX report, to keep yourself aligned with what Google Search is using, then you'd need the web-vitals[24] library maintained by Google itself.

If you need something more custom, then you're probably better off using the web client of whichever tool you're using for dashboarding and combining it with the JavaScript performance API to extract the times of events you want to monitor.

Investigating Locally

After you've monitored performance for some time and identified pain points you'd want to solve, there is a final step: actually solving those pain points!

This is what the rest of the book is all about. How to identify what causes a pain point you discovered, whether it's a very long initial load time or a very slow interaction in a certain place. What can influence this particular pain point in each direction, and which solutions can move the needle in the direction you want.

Ready to start investigating?

3. Intro to Initial Load Performance

When it comes to writing good React code, the actual code itself is just a small piece of the puzzle these days. Before even writing the first component, we need to figure out where and how to do it. And this is where we're bombarded with choices and opinions.

Should I start the project with Vite or Next.js? Or Remix, which mutated into React Router now? Do I even need a framework at all? There are also different rendering methods - should I go with CSR, or SSG, or SSR, or RSC, or ISR? What do those letters even mean? Since when do I need a PhD to start writing simple React code? 😭

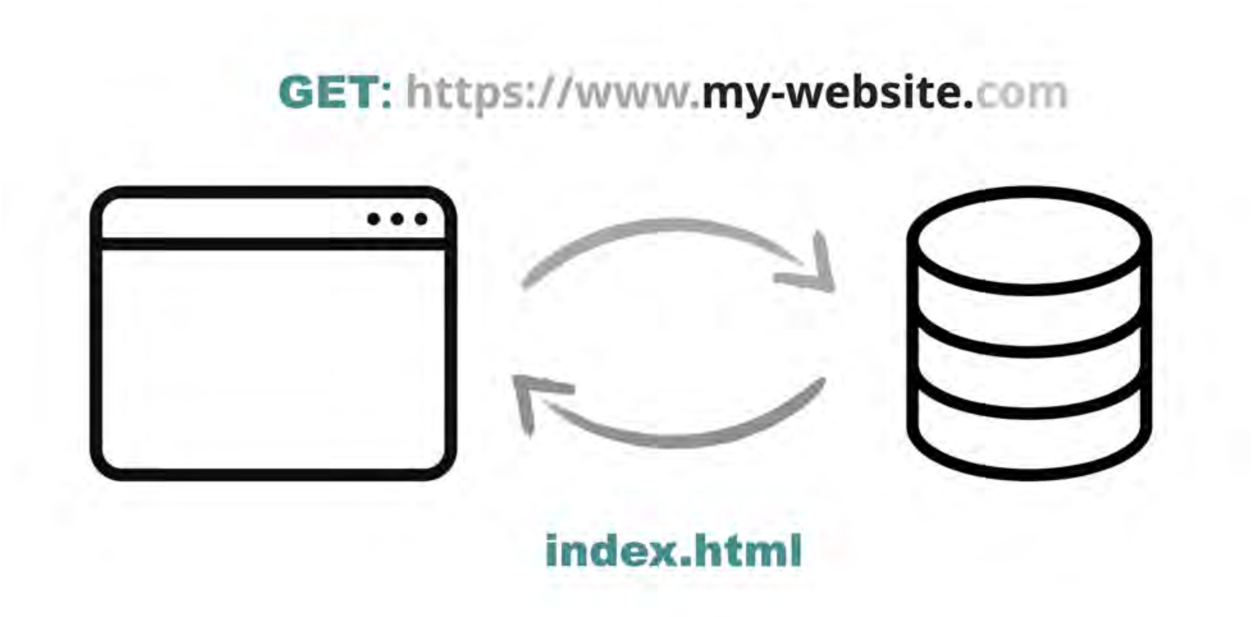
But before even starting to talk about rendering methods and frameworks, we need to do some pre-work. Because the most important aim of them, other than improving the developer experience with fancy tooling, is to improve performance and SEO.

So, step zero in understanding them and being able to make an informed choice about which rendering pattern to use is to understand how front-end performance works. And what SEO is, of course - but that one in the next chapter.

Let's focus on performance first.

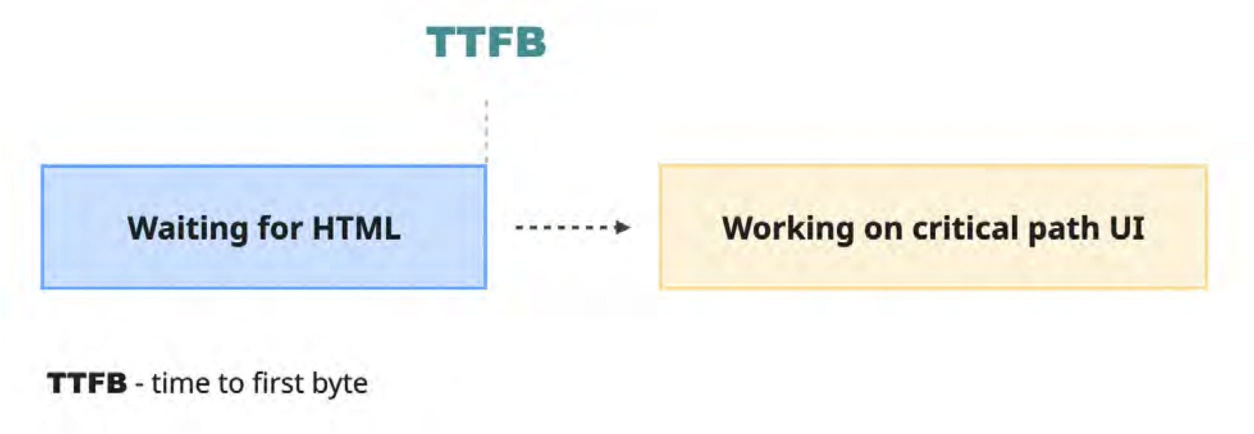
Introducing Initial Load Performance Metrics

What happens when I open my browser and try to navigate to my favorite website? I type "http://www.my-website.com[25]" into the address bar, the browser sends a GET request to the server, and receives an HTML page in return.



The time it takes to do that is known as "Time To First Byte"[26] (TTFB): the time between when the request is sent and when the result starts arriving. After the HTML is received, the browser now has to convert this HTML into a usable website as soon as possible.

It starts by rendering on the screen what is known as the "critical path"[27]: the minimal and most important amount of content that can be shown to the user.



What exactly should be in the critical path is a complicated question. Ideally, everything, so that the user sees the complete experience right away. But also - nothing, since it needs to be as fast as possible, since it's a "critical" path. They are mutually exclusive, so there needs to be a compromise.

The compromise looks like this. The browser assumes that to build the "critical path", it absolutely needs at least these types of resources:

- The initial HTML that it receives from the server, to construct the actual DOM elements from which the experience is built.
- The important CSS files that *style* those initial elements - otherwise, if it proceeded without waiting for them, the user would see a weird "flash" of unstyled content at the very beginning.
- The critical JavaScript files that modify the layout synchronously.

The first one (HTML) is what the browser gets in the initial request from the server. It starts parsing it, and while doing so, it extracts links to the CSS and JS files it needs to complete the "critical path". It then sends requests to get them from the server, waits until they are downloaded, processes them, combines all of this together, and at some point at the end, paints the "critical path" pixels on the screen.

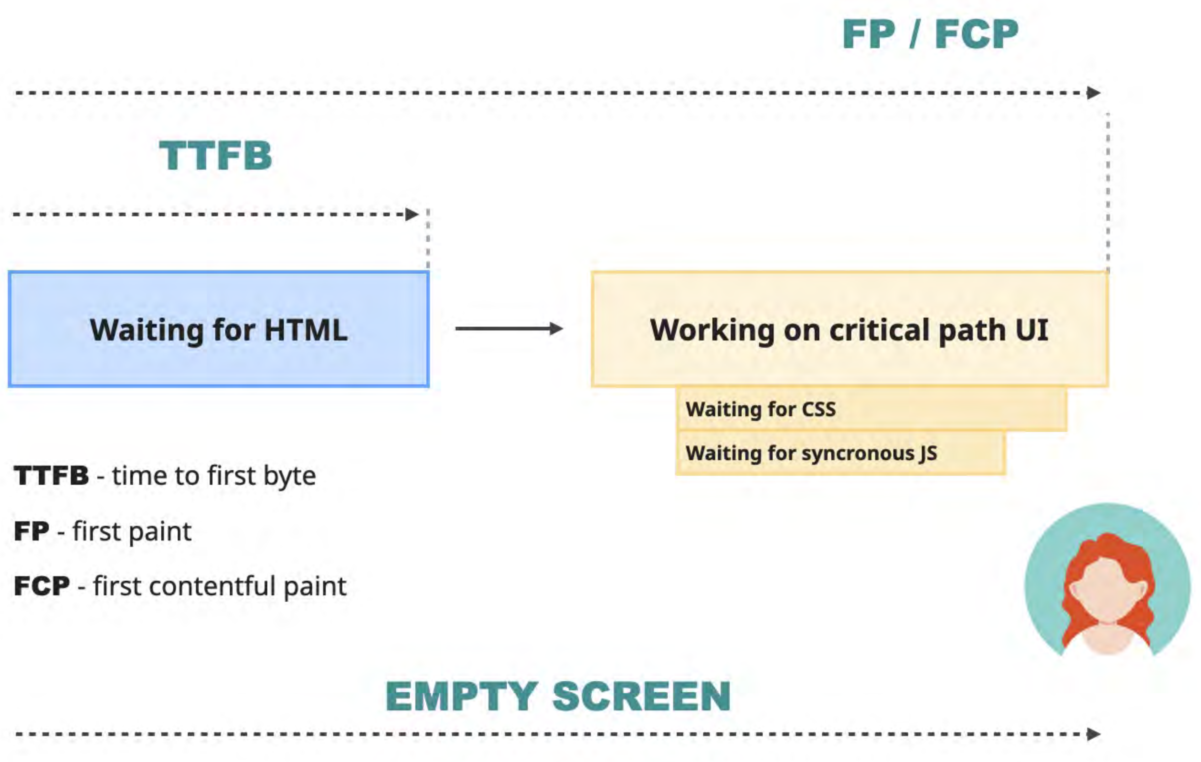
Since the browser can't complete the initial rendering without those critical resources, they are known as "render-blocking resources". Not all CSS and JS resources are render-blocking, of course. It's usually only:

- Most of the CSS, inline or via the `<link>` tag.
- JavaScript resources in the `<head>` tag that are not `async` or `deferred`.

The overall process of rendering the "critical path" looks something like this (roughly):

- The browser starts parsing the initial HTML.
- While doing so, it extracts links to CSS and JS resources from the `<head>` tag.
- Then, it kicks off the downloading process and waits for the blocking resources to finish downloading.
- While waiting, it continues processing HTML if possible.
- After all the critical resources are received, they are processed as well.
- And finally, it finishes whatever needs to be done and paints the actual pixels of the interface.

This point in time is what we know as **First Paint (FP)**. It's the very first time the user has an opportunity to see something on the screen. Whether it will happen or not depends on the HTML the server sent. If there is something meaningful there, like text or an image, then this point will also be when the First Contentful Paint[28] (FCP) happens. If the HTML is just an empty div, then the FCP will happen later.


FP / FCP
TTFB
Waiting for HTML
Working on critical path UI
Waiting for CSS
Waiting for syncronous JS
TTFB - time to first byte
FP - first paint
FCP - first contentful paint
EMPTY SCREEN

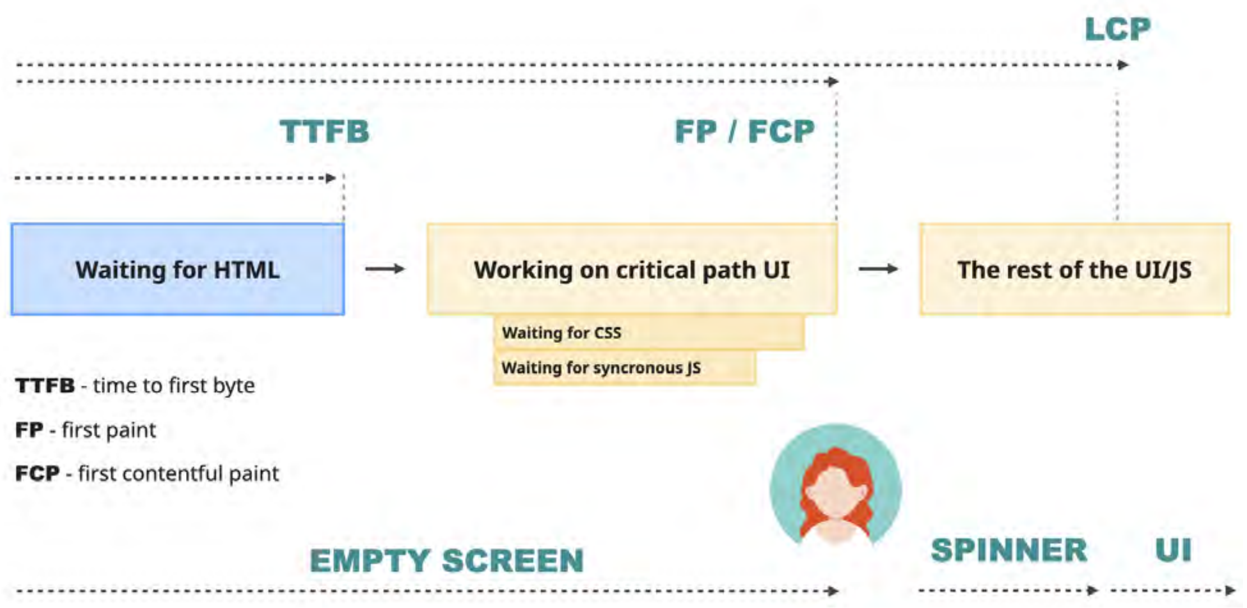
First Contentful Paint (FCP) is one of the most important performance metrics since it measures *perceived initial load*. Basically, it is the user's first impression of how fast your website is.

Until this moment, users are just biting their nails while staring at the blank screen. According to Google[29], a good FCP number is **below 1.8 seconds**. After that, users will start losing interest in what your website can offer and might start leaving.

However, FCP is not perfect. If the website starts its load with a spinner or some loading screen, the FCP metric will represent that. But it's highly unlikely that the user navigated to the website just to check out the fancy loading screen. Most of the time, they want to access the content.

For this, the browser needs to finish the work it started. It waits for the rest of the non-blocking JavaScript, executes it, applies changes that originated from it to the DOM on the screen, downloads images, and otherwise polishes the user experience.

Somewhere during this process is when the Largest Contentful Paint[30] (LCP) time happens. Instead of the very first element, like FCP, it represents the main content area on the page - the largest text, image, or video visible in the viewport. According to Google[31], this number should ideally be **below 2.5 seconds**. More than that, and the users will think the website is slow.


LCP
TTFB
FP / FCP
Waiting for HTML
Working on critical path UI
The rest of the UI/JS
Waiting for CSS
Waiting for syncronous JS
TTFB - time to first byte
FP - first paint
FCP - first contentful paint
EMPTY SCREEN
SPINNER
UI

All of these metrics are part of Google's Web Vitals[32] - a set of metrics that represent user experience on a page. LCP is one of the three **Core Web Vitals** - three metrics that represent different aspects of the user experience. **LCP** is responsible for the ***loading performance***.

These metrics can be measured by Lighthouse[33]. Lighthouse is a Google performance tool integrated into the Chrome DevTools and can also be run via a shell script, web interface, or a Node module. You can use it as a Node module to run it inside your build and detect regressions before they hit production. Use the integrated DevTools version for local debugging and testing. And the web version to check out the performance of competitors 😉.

Overview of the Performance Tools

All of the above is a very brief and simplified explanation of the process. But it's already a lot of abbreviations and theory to make a person's head spin. For me personally, reading something like this is of no use. I instantly forget everything unless I can see it in action and play around with it with my own hands.

For this particular topic, I find the easiest way to fully understand the concepts is to simulate different scenarios on a semi-real page and see how they change the outcome. So let's do exactly that before doing even more theory (and there is so much more!).

Setting Up the Project

You can do all of the simulations below on your own project if you wish - the results should be more or less the same. For a more controlled and simplified environment, however, I would recommend using the Study Project. You can access it here: https://github.com/developerway/web-perf-fundamentals[34].

Start by installing all the dependencies:

```
npm install
```

Building the project:

```
npm run build --workspace=chapter3-simple-frontend
```

And starting the server for this chapter:

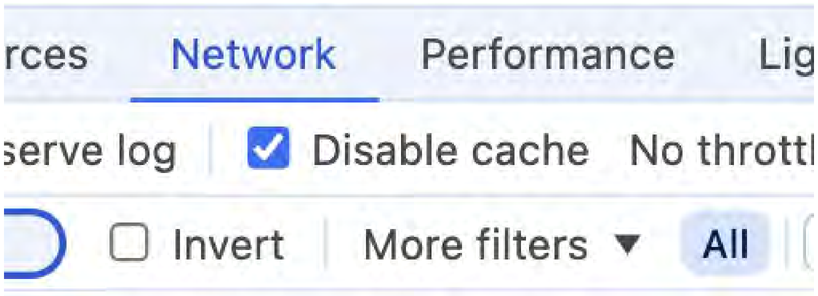
```
npm run start --workspace=chapter3-simple-frontend
```

You should see a nice dashboard page at http://localhost:3000[35].

Exploring the Necessary DevTools

Open the website you want to analyze in Chrome and open Chrome DevTools. Find the "Performance" and "Lighthouse" panels there and move them closer together. We'll need both of them.

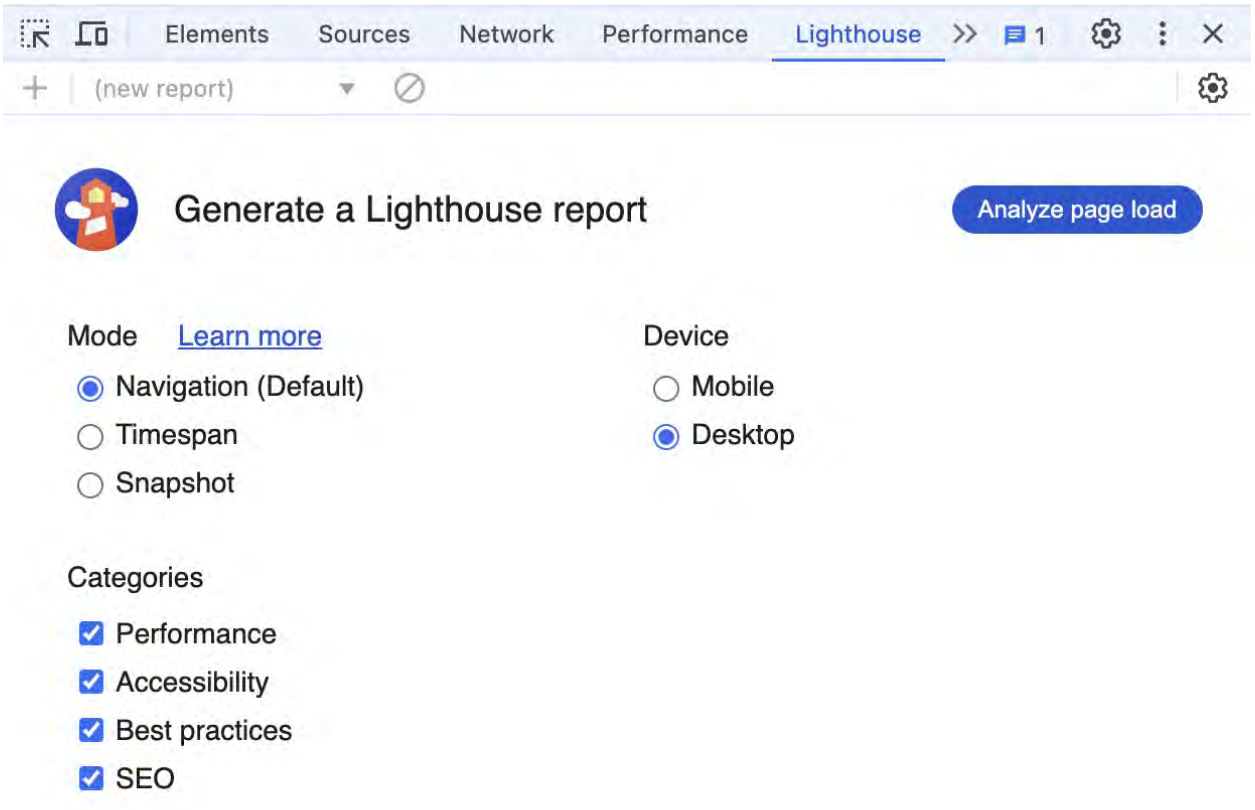
Also, before doing anything else in this chapter, make sure you have the "Disable cache" checkbox enabled. It should be in the Network panel at the very top.



This is so that we can emulate first-time visitors - people who've never been to our website before and don't have any resources cached by the browser yet.

Exploring the Lighthouse Panel

Open the Lighthouse panel now. You should see a few settings there and the "Analyze page load" button.

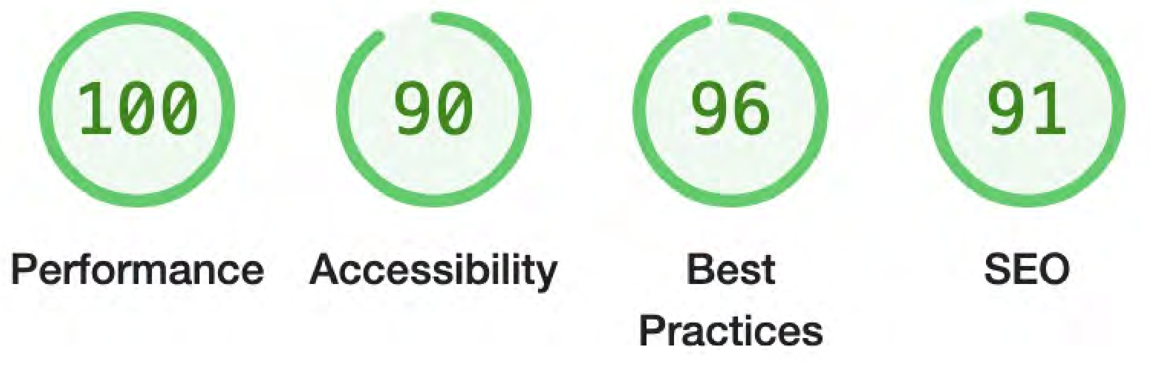


Mode: [✓ Navigation (Default)] Timespan Snapshot

Device: Mobile [✓ Desktop]

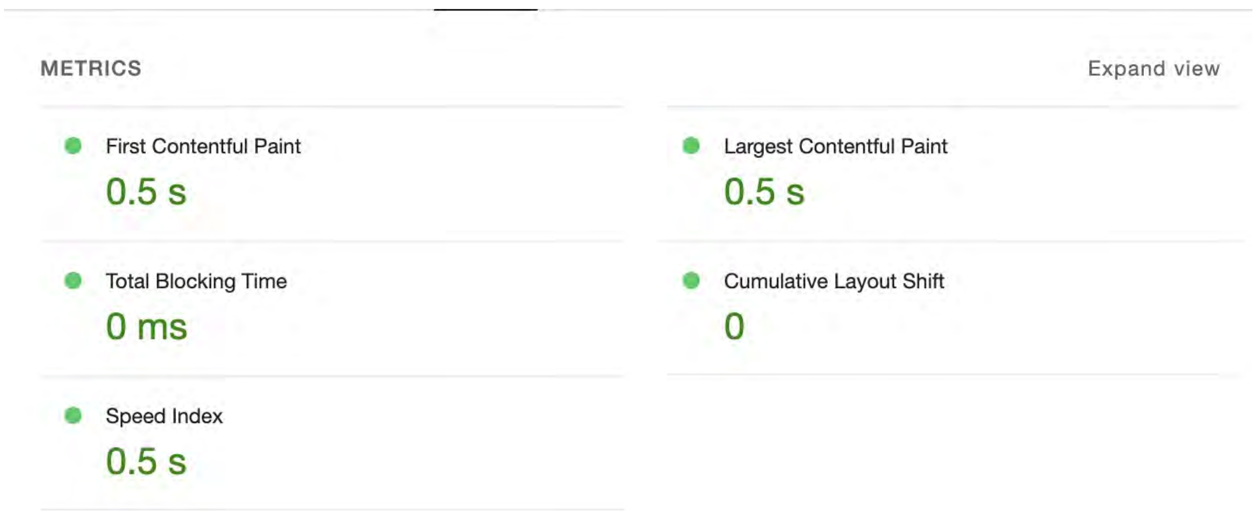
Categories
- [✓] Performance
- [✓] Accessibility
- [✓] Best practices
- [✓] SEO

"Navigation" mode is the one we're interested in for this section - it will run a detailed analysis of the page's initial load. The report will give you scores like this:



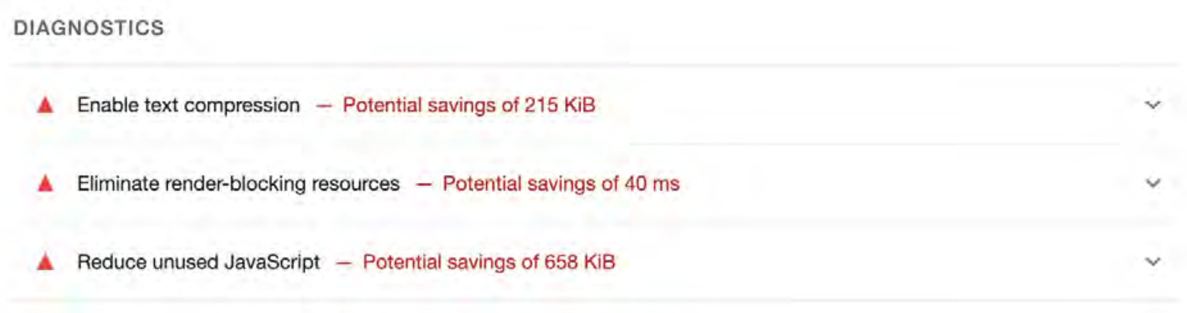
The local performance is perfect, no surprise there - everything always "works on my machine".

There will also be metrics like this:



The FCP and LCP values described earlier are right at the top.

Below, you'll see a list of suggestions that can help you improve your scores.



DIAGNOSTICS

▲ Enable text compression — Potential savings of 215 KiB

▲ Eliminate render-blocking resources — Potential savings of 40 ms

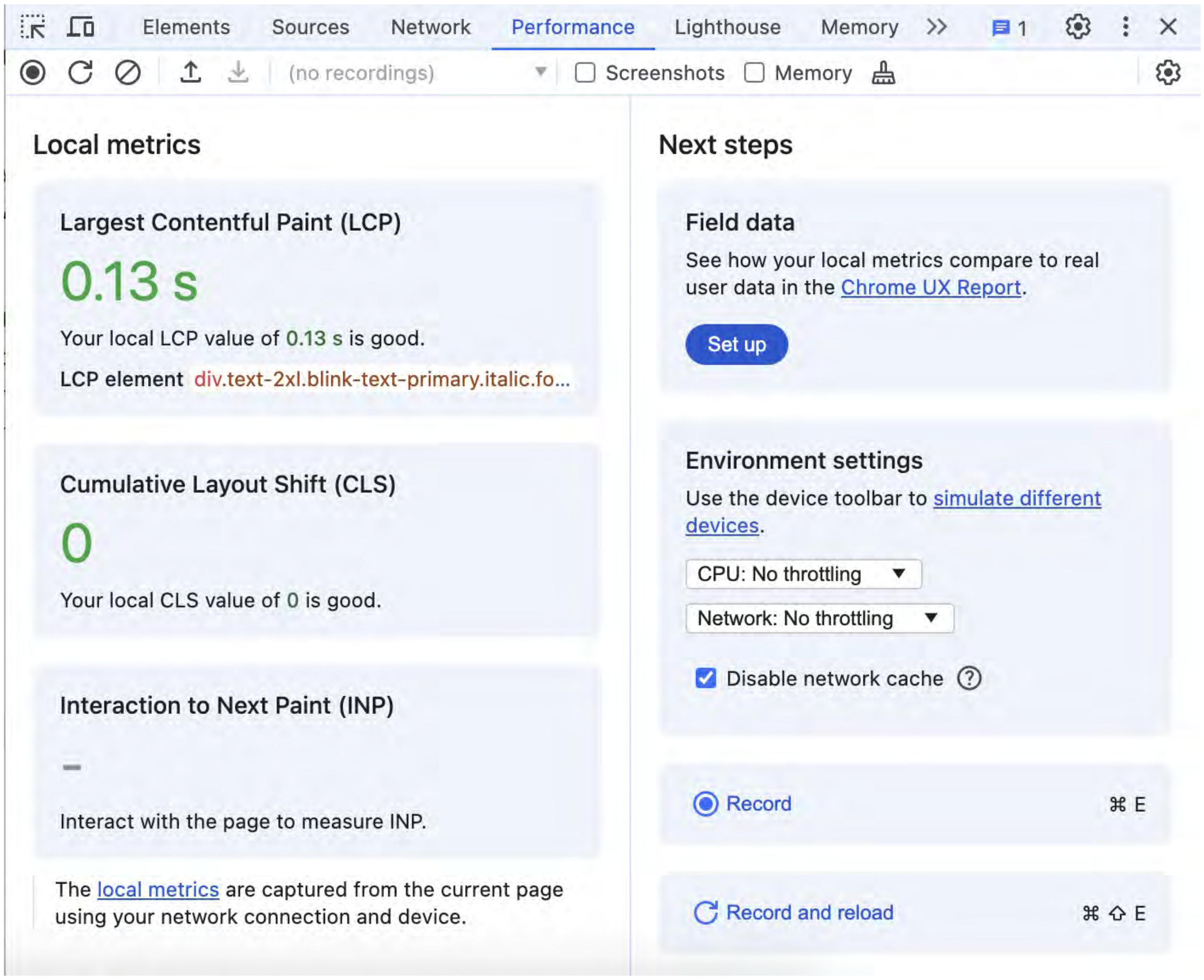
▲ Reduce unused JavaScript — Potential savings of 658 KiB

Every suggestion can be expanded. You'll find more detailed information when you expand a suggestion, and sometimes links that explain that particular topic. Not all of them can be actioned, but it's an incredible tool to get started on performance and learn more about different things that can improve it. It's possible to spend hours just reading through those reports and the related links.

Lighthouse, however, only gives surface-level information and doesn't allow you to simulate different scenarios, like a slow network or low CPU. It's just a great entry point and an awesome tool to track performance changes over time. To dig deeper into what is happening, we need the **"Performance"** panel.

Exploring the Performance Panel

When first loaded, the Performance panel should look something like this:



Local metrics

Largest Contentful Paint (LCP)

0.13 s

Your local LCP value of 0.13 s is good.

LCP element div.text-2xl.blink-text-primary.italic.fo...

Cumulative Layout Shift (CLS)

0

Your local CLS value of 0 is good.

Interaction to Next Paint (INP)

–

Interact with the page to measure INP.

The local metrics are captured from the current page using your network connection and device.

Next steps

Field data

See how your local metrics compare to real user data in the Chrome UX Report.

Set up

Environment settings

Use the device toolbar to simulate different devices.

CPU: No throttling

Network: No throttling

Disable network cache

Record ⌘ E

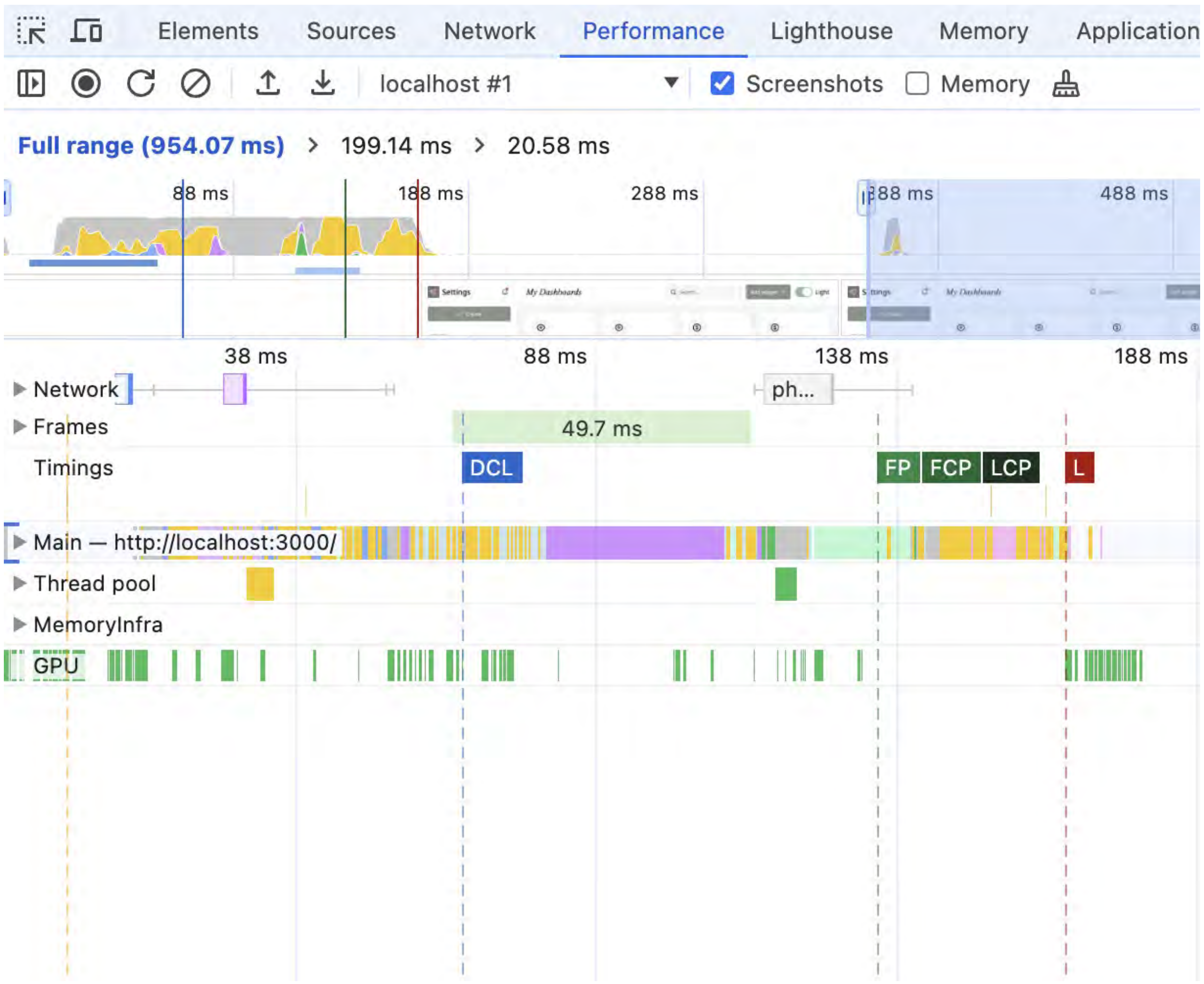
Record and reload ⌘ ⇧ E

It shows the three Core Web Vitals[36] metrics, one of which is our LCP, gives you the ability to simulate slow Network and CPU, and the ability to record performance details over time.

Find and check the "Screenshots" checkbox at the very top of the panel, then click the "Record and reload" button, and when the website reloads itself, stop the recording. This will be your detailed report on what is happening on the page during the initial load.

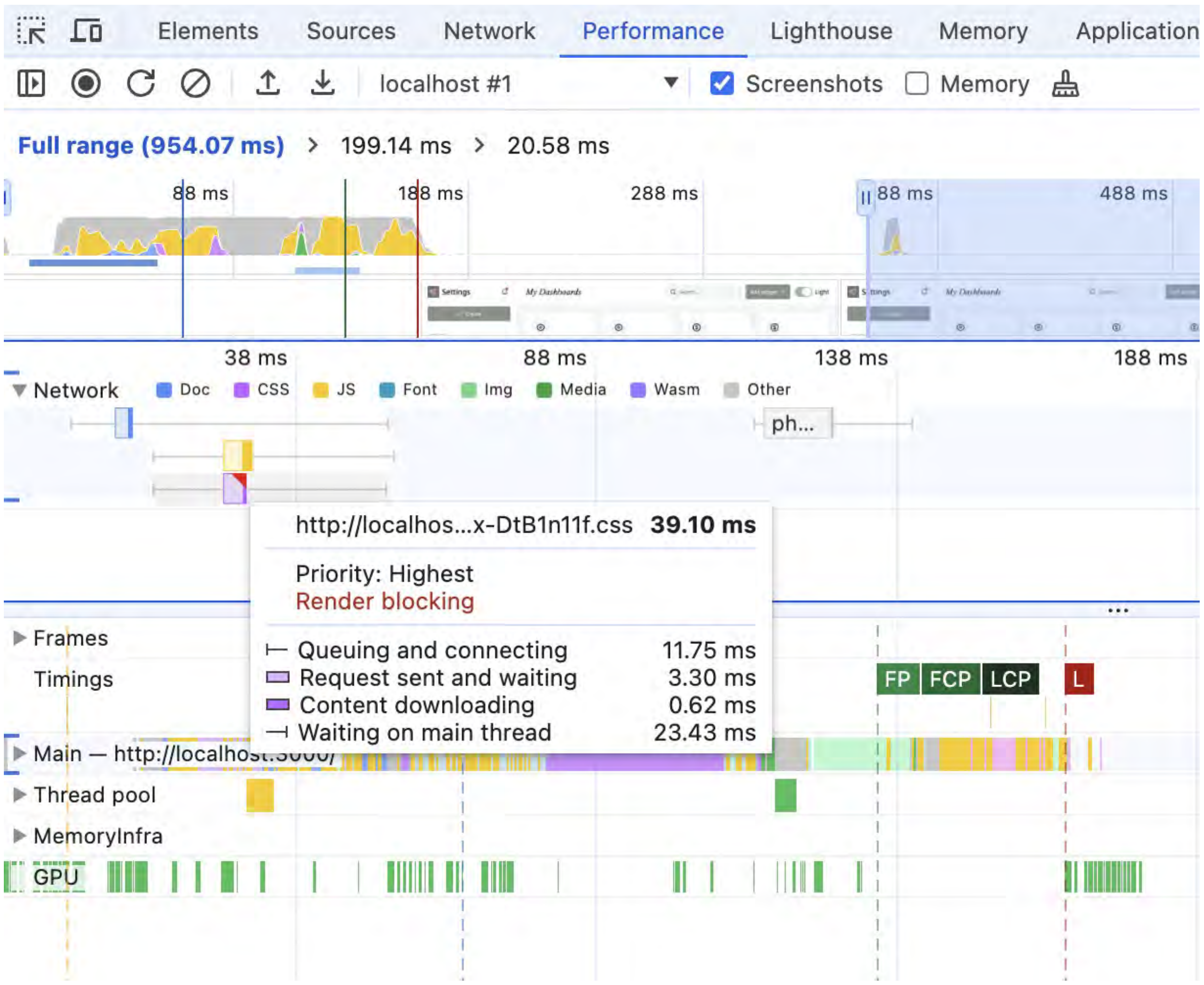
This report will have a few sections.

At the very top sits the general "**timeline overview**" section.



You'll be able to see here that something is happening on the website, but not much more. When you hover over it, a screenshot of what was happening will appear, and you'll be able to select and zoom in on a particular range to get a closer look.

Underneath, there is a **Network section.** When expanded, you'll see all the external resources being downloaded and the exact time they appear on the timeline. When hovering over a particular resource, you'll see detailed information on how much time was spent on each stage of the download. The resources with red corners will indicate the blocking resources.



http://localhos...x-DtB1n11f.css **39.10 ms**

Priority: Highest
Render blocking

Queuing and connecting	11.75 ms
Request sent and waiting	3.30 ms
Content downloading	0.62 ms
Waiting on main thread	23.43 ms

If you're working on the Study Project, you'll see exactly the same picture, and this picture matches what we went through in the previous section to the letter:

- At the beginning, there is the blue block - a request to get the HTML for the website.
- After it's finished loading, there is a brief pause (to parse the HTML), and two requests for additional resources are sent out.
- One of them (the yellow one) is for JavaScript, it's not blocking.
- Another one (the purple one) is for CSS, and this one is blocking.

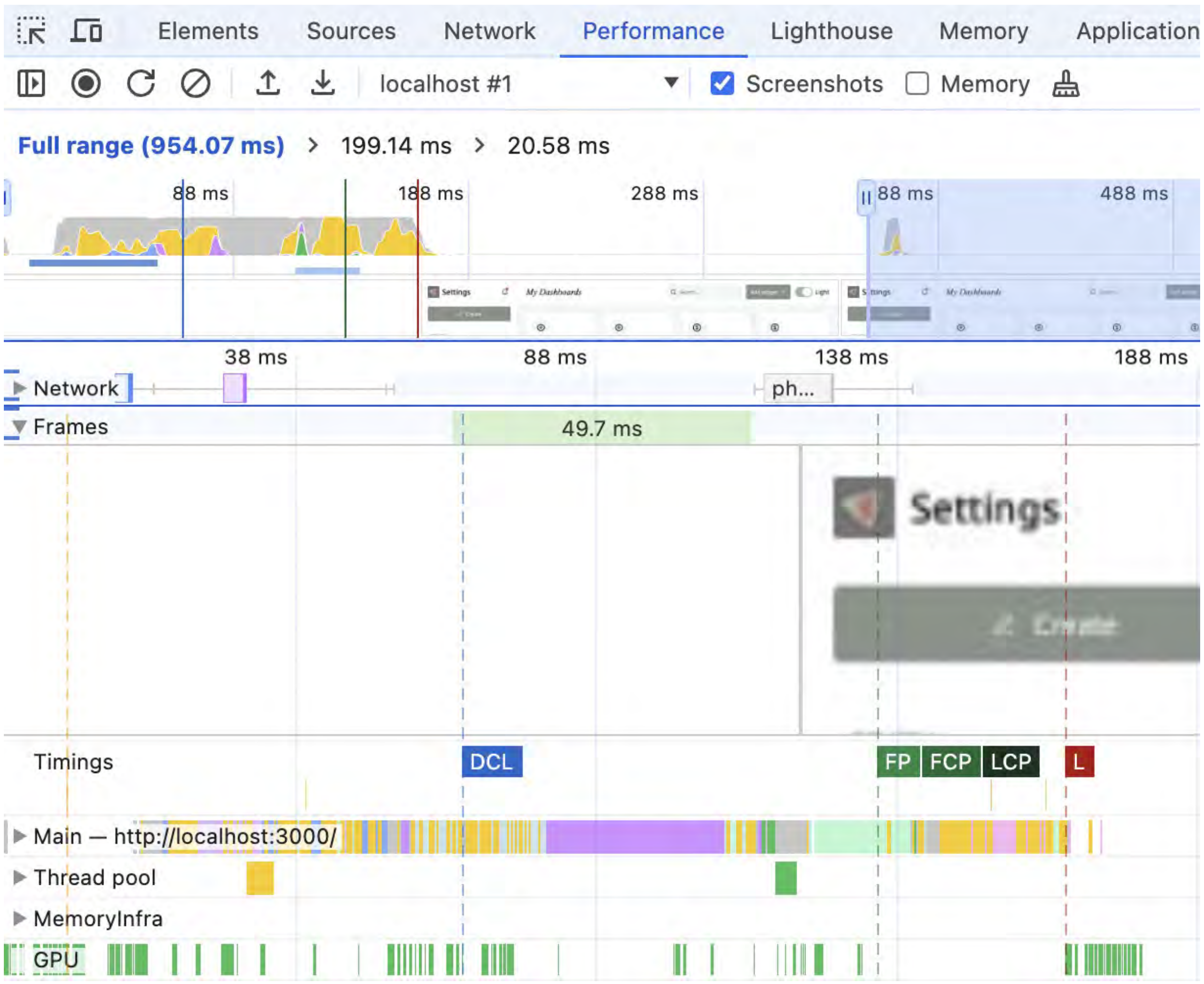
If you open this chapter's Study Project folder now (`src/chapter3-initial-load-performance/simple-frontend`) and peek into the `dist` folder, the source code matches this behavior:

- There will be an `index.html` file, an `assets` folder, and `.css` and `.js` files inside.

- Inside the `index.html` file in the `<head>` section, there will be a `<link>` tag that points to the CSS file. As we know, CSS resources in the `<head>` are render-blocking, so that checks out.
- Also, inside `<head>`, there is a `<script>` tag that points to the JavaScript file inside the `assets` folder. It's neither deferred nor async, but it has `type="module"`. Those are deferred automatically[37], so this also checks out - the JavaScript file in the panel is non-blocking.

> **Additional Challenge**.
> If you have a project you're working on, record the initial load performance for it and look into the Network panel. You'll likely see many more resources downloaded.
>
> - How many render-blocking resources do you have? Are all of them necessary?
> - Do you know where the "entry" point for your project is and how blocking resources appear in the `<head>` section? Try building the project with your variation of `npm build` and search for them. Hint:
>   - If you have a purely webpack-based project, look for the `webpack.config.js` file. Paths to the HTML entry points should be inside.
>   - If you're on Vite, look into the `dist` folder - same as with the study project.
>   - If you're on the Next.js App router - take a peek into the `.next/server/app` folder.

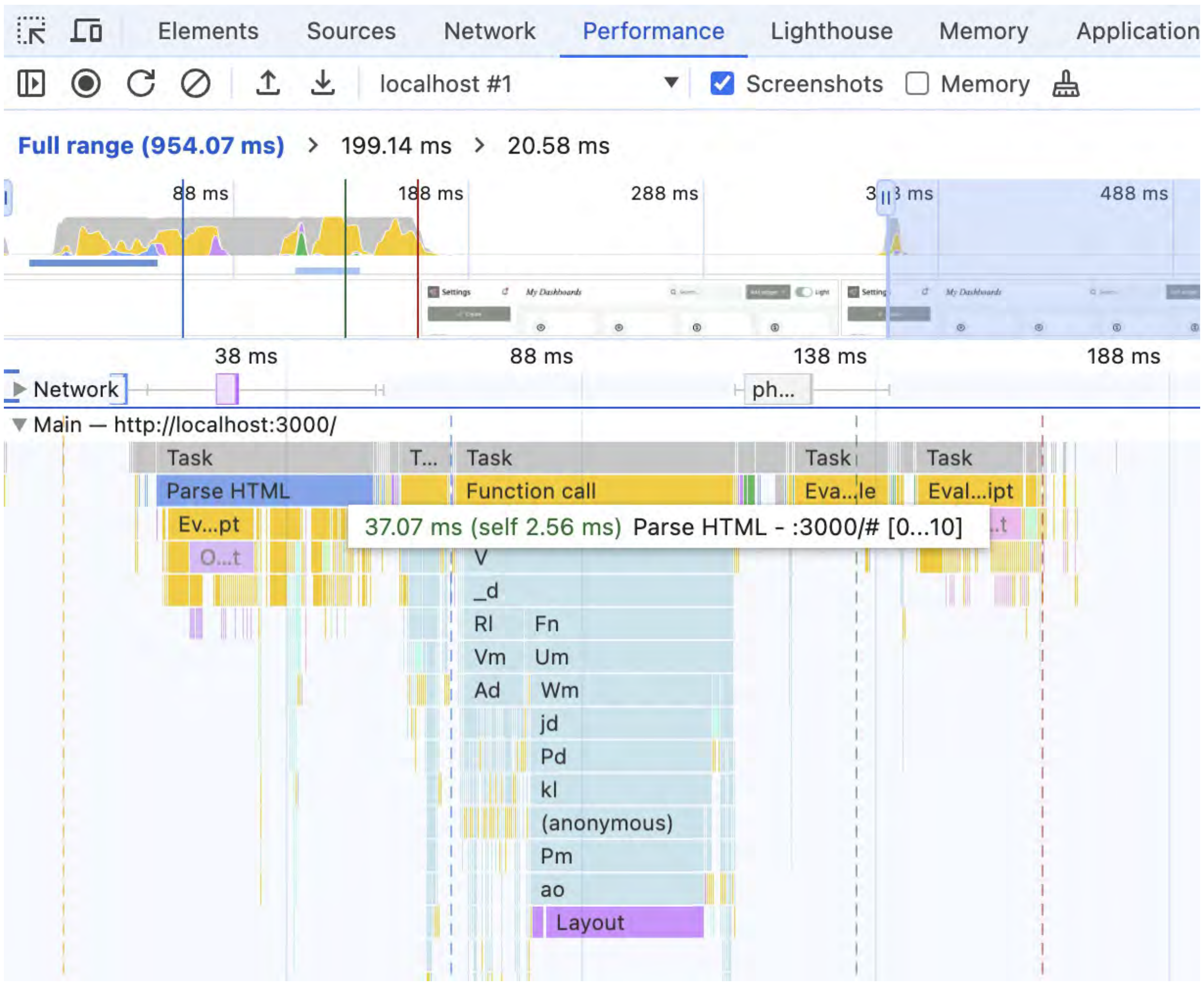
Under the Network section, you can find the **Frames** section.



This one will show you what was rendered on the screen during your timeline. Pretty cool when you start correlating this with the downloaded resources and the Main section.

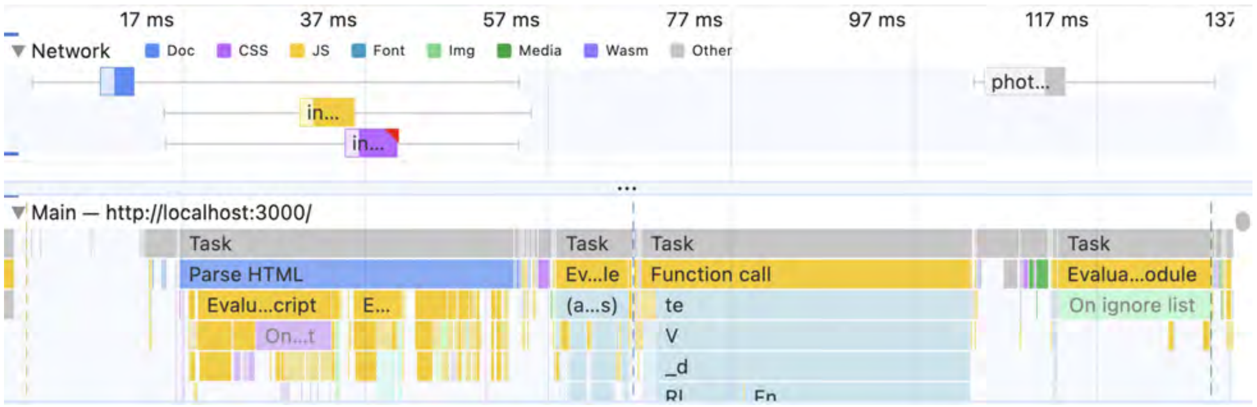
At the very bottom, you can see the metrics we discussed before (FCP, LCP). When hovering over them, you can see their exact time. Clicking on them will update the "summary" tab, where you'll find information on what this metric is and a link to learn more. DevTools are all about educating people these days.

Then, finally, the **Main** section. This is what is happening in the main thread during the timeline recorded.



We can see things here like "Parse HTML" or "Layout" and how long they took. The yellow blocks are JavaScript-related, and they are a bit useless since we're using a production build with compressed JavaScript. But even in this state, it gives us a rough idea of how long the JavaScript execution takes compared to HTML parsing and drawing the Layout, for example.

It's especially useful for performance analysis when both **Network** and **Main** are open and zoomed in so they take up the full screen.



From here, I can see that I have an incredibly fast server, a very fast network, and small bundles. None of the network tasks is a bottleneck. They don't take any significant time, and between them, the browser is just chilling and doing its own thing. So, if I wanted to speed up the initial load here, I need to look into why "Parse HTML" is so slow - it's the longest task on the graph.

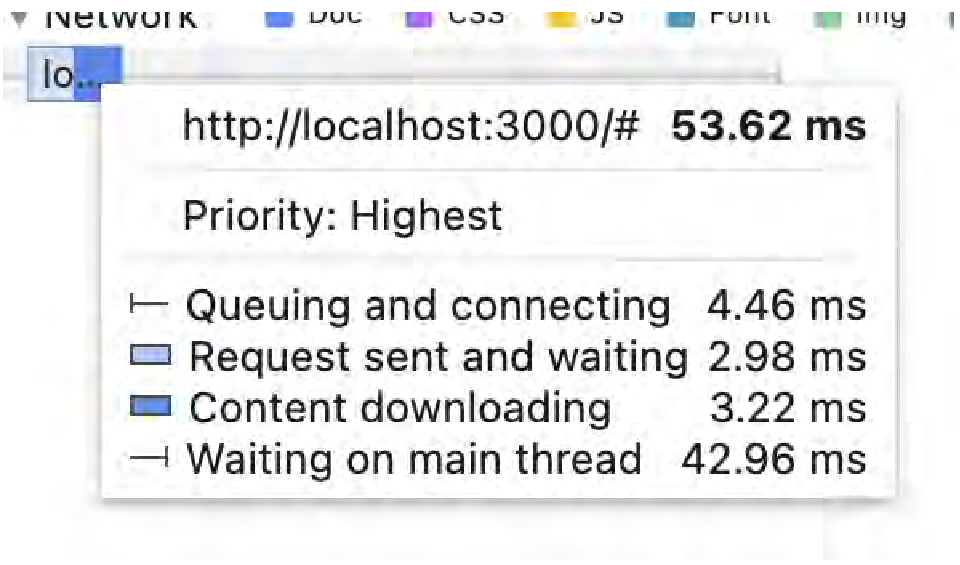
Or, if we look at the absolute numbers, I shouldn't do anything here, performance-wise. The entire initial load takes less than 200 ms and is well below Google's recommended threshold 😳. But this is happening because I'm running this test locally (so no actual network costs), on a very fast laptop, and with a very basic server.

Time to simulate real life.

Exploring Different Network Conditions

Very Slow Server

First of all, let's make the server more realistic. Right now, the very first "blue" step takes about 50 ms, 40 ms of which is just waiting.



http://localhost:3000/# **53.62 ms**

Priority: Highest

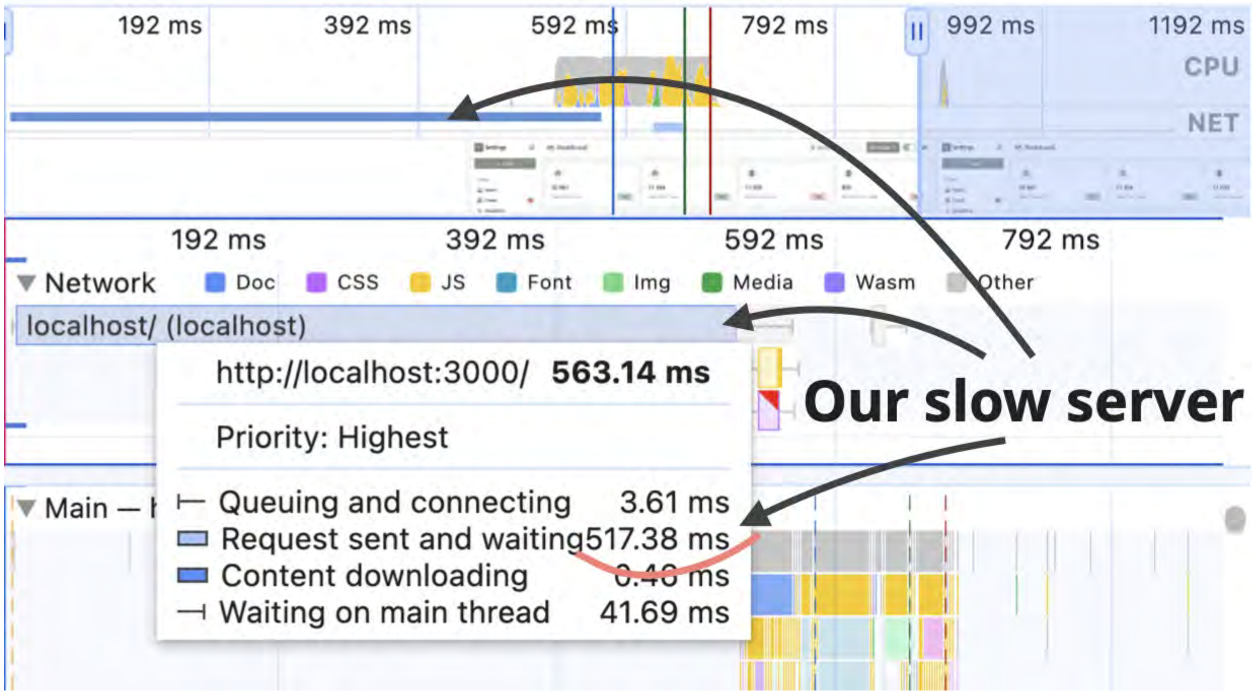
Queuing and connecting 4.46 ms
Request sent and waiting 2.98 ms
Content downloading 3.22 ms
Waiting on main thread 42.96 ms

In real life, the server will do stuff, check permissions, generate stuff, check permissions two more times (because it has lots of legacy code and that triple-checking got lost during PR reviews), and otherwise will be busy.

Navigate to the `server.ts` in this chapter's Study Project. Find the commented-out `// await sleep(500)`, and uncomment it. This will give the server a 500 ms delay before it returns the HTML - it seems reasonable enough for an old and complicated server.

Rebuild the project, restart it, and rerun the performance recording.

Nothing has changed on the timeline except for the initial blue line: it's now incredibly long compared to the rest of the bars.



http://localhost:3000/ **563.14 ms**

Priority: Highest

Queuing and connecting	3.61 ms
Request sent and waiting	517.38 ms
Content downloading	0.46 ms
Waiting on main thread	41.69 ms

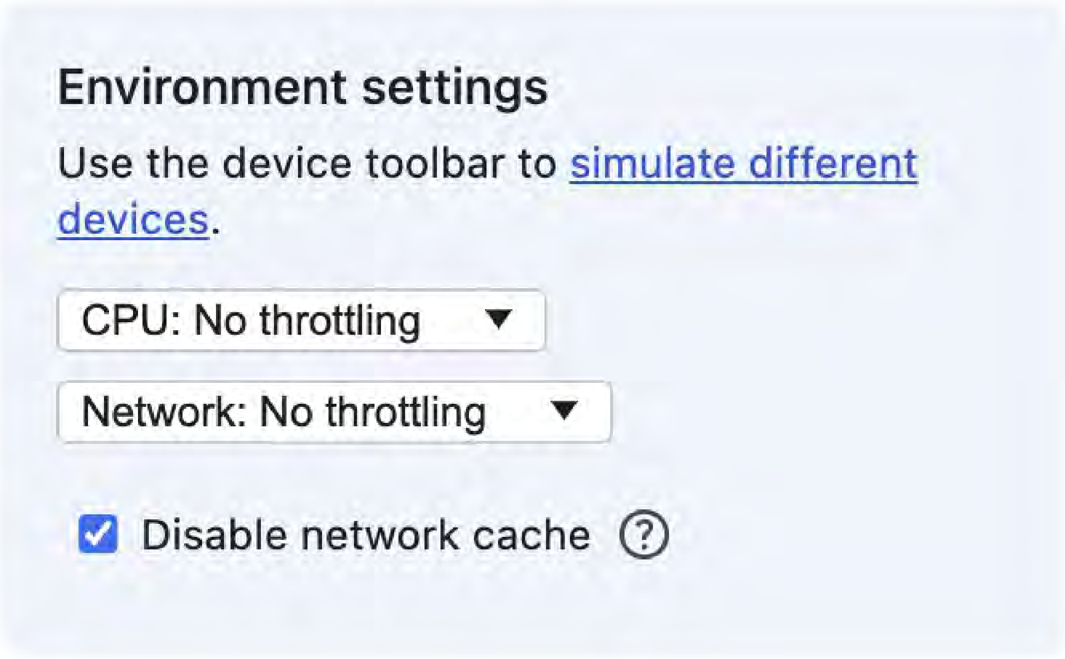
This situation highlights the importance of looking at the whole picture and looking for bottlenecks before doing any performance optimizations. The LCP value is ~650 ms, out of which ~ 560 ms is spent waiting for the initial HTML. The React portion of it is around 50 ms.

Even if I somehow manage to halve it and reduce it to 25 ms, in the overall picture, it will be just 4%. And reducing it by half will be *a lot* of effort here. A much more effective strategy might be to focus on the server and figure out why it's so slow.

Emulating Different Bandwidth and Latency

Not everyone lives in the world of a 1-gigabit connection. In Australia, for example, 50 megabits per second is considered a high-speed internet connection, and it will cost you around 90 Australian dollars a month. It's not 3G, of course, on which plenty of people around the world are stuck. But still, I cry every time I hear people in Europe bragging about their 1 gigabit per second internet plans for cheap.

So let's emulate the Australian internet and see what happens to the performance metrics. To do that, clear the existing recording in the Performance tab (the button near the reload and record). The panel with network settings should show up:


Environment settings

Use the device toolbar to simulate different devices.

CPU: No throttling

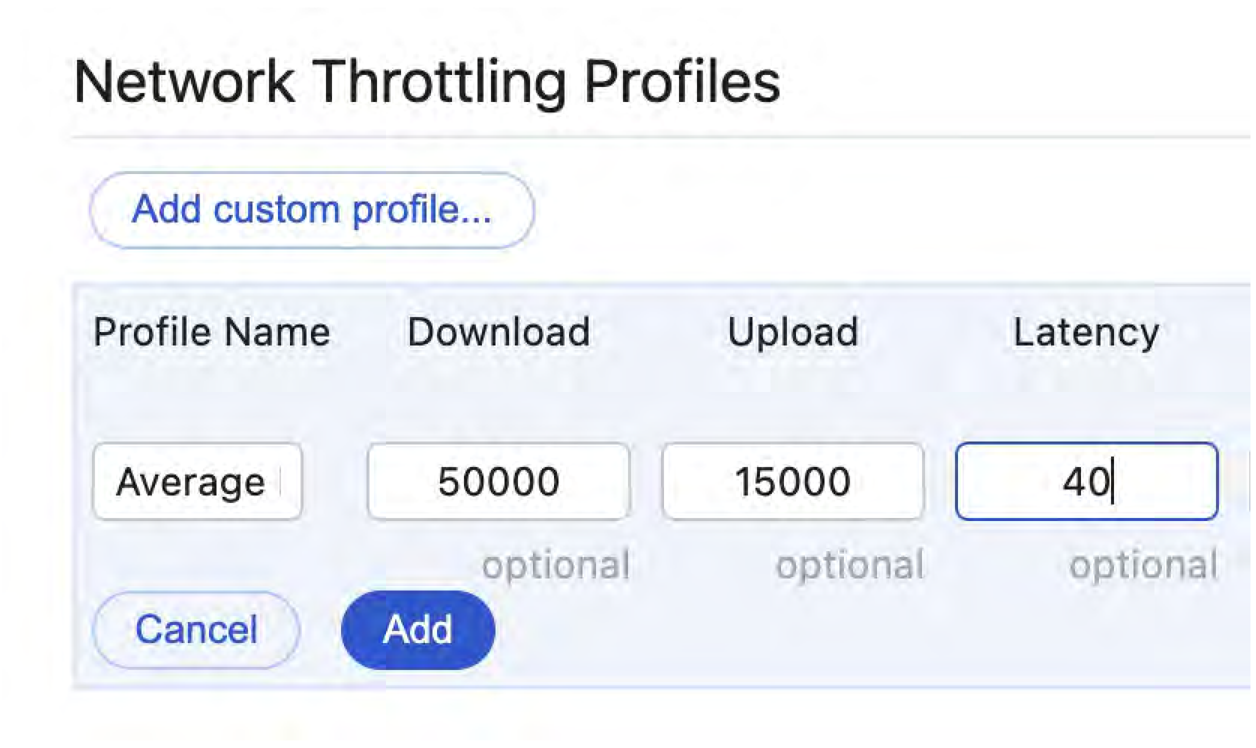
Network: No throttling

☑ Disable network cache

If it's not there in your version of Chrome, the same setting should be available in the Network tab.

Add a new profile in the "Network" dropdown with the following numbers:

- Profile Name: "Average Internet Bandwidth".
- Download: 50000 (50 Mbps).
- Upload: 15000 (15 Mbps).
- Latency: 40 (about average for a general internet connection).



Network Throttling Profiles

Add custom profile...

Profile Name	Download	Upload	Latency
Average	50000	15000	40
	optional	optional	optional

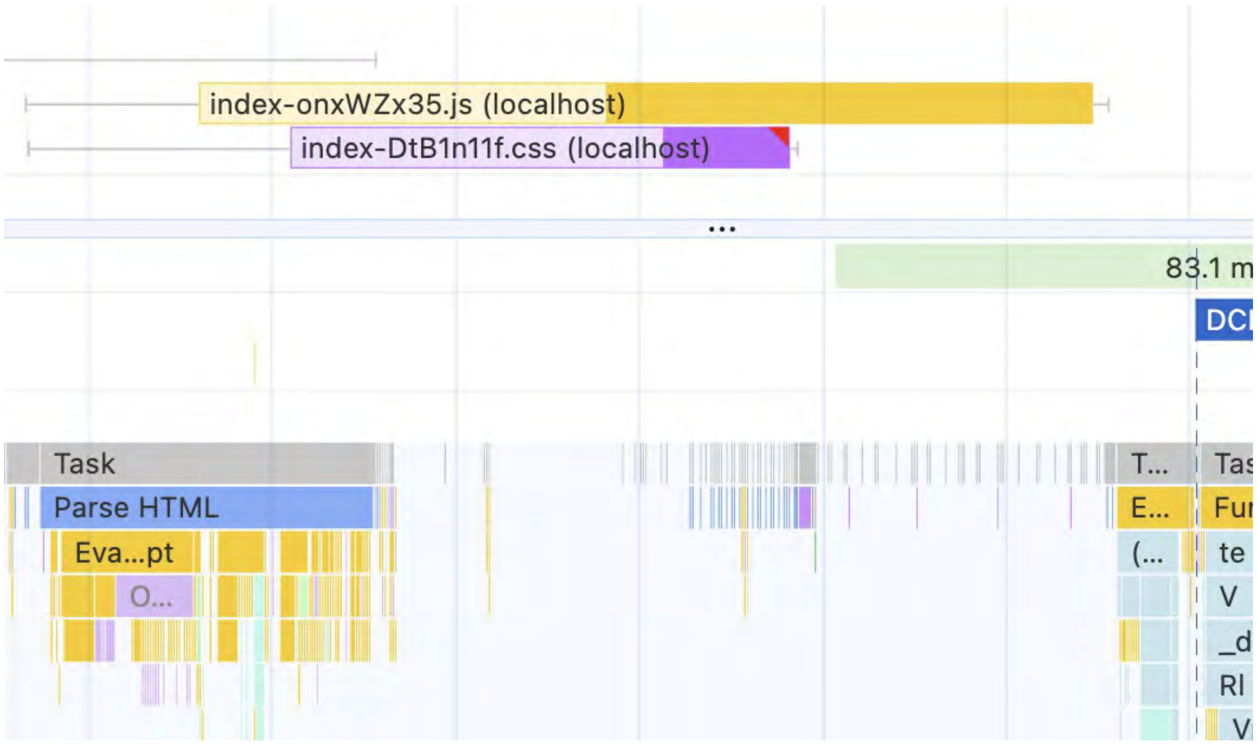
Cancel Add

Now select that profile in the dropdown and rerun the performance recording again.

What do you see? For me, it looks like this.

The **LCP** value barely changed - a slight increase from 640 ms to 700 ms. Nothing changed in the initial blue "server" part, which is explainable: it sends only the bare minimum HTML, so it shouldn't take long to download it.

But the relationship between the downloadable resources and the main thread changed drastically.

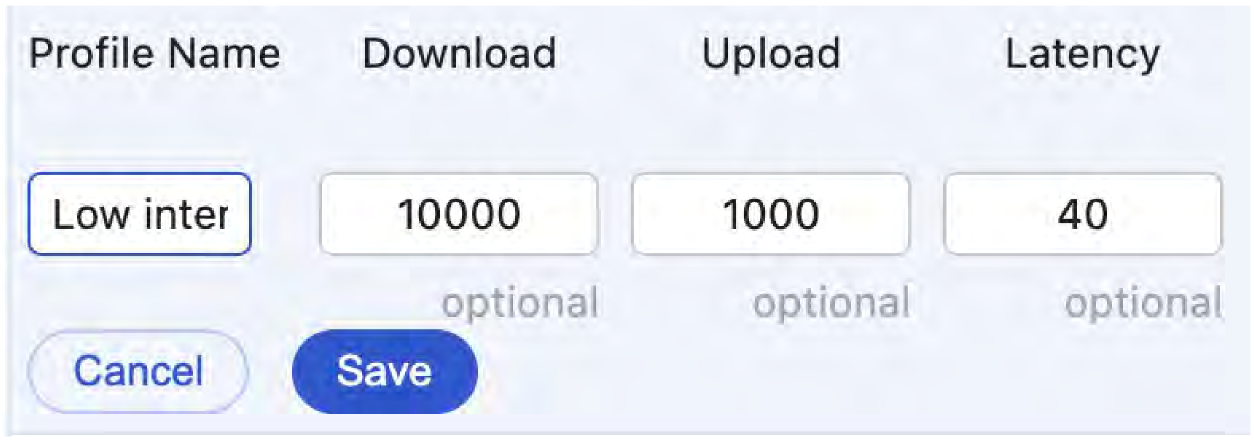


I can clearly see the impact of the **render-blocking CSS** file now. The *Parse HTML* task has finished already, but the browser is chilling and waiting for the CSS - nothing can be painted until it's downloaded. Compare it with the previous picture, where the resources were downloaded almost instantly while the browser was parsing HTML.

After that, technically, the browser could've painted something. But there isn't anything, we're sending only an empty div in the HTML file. So the browser continues to wait patiently until the JavaScript file is downloaded and can be executed.

This approximately 60 ms gap of waiting is exactly the increase in the **LCP** that I'm seeing.

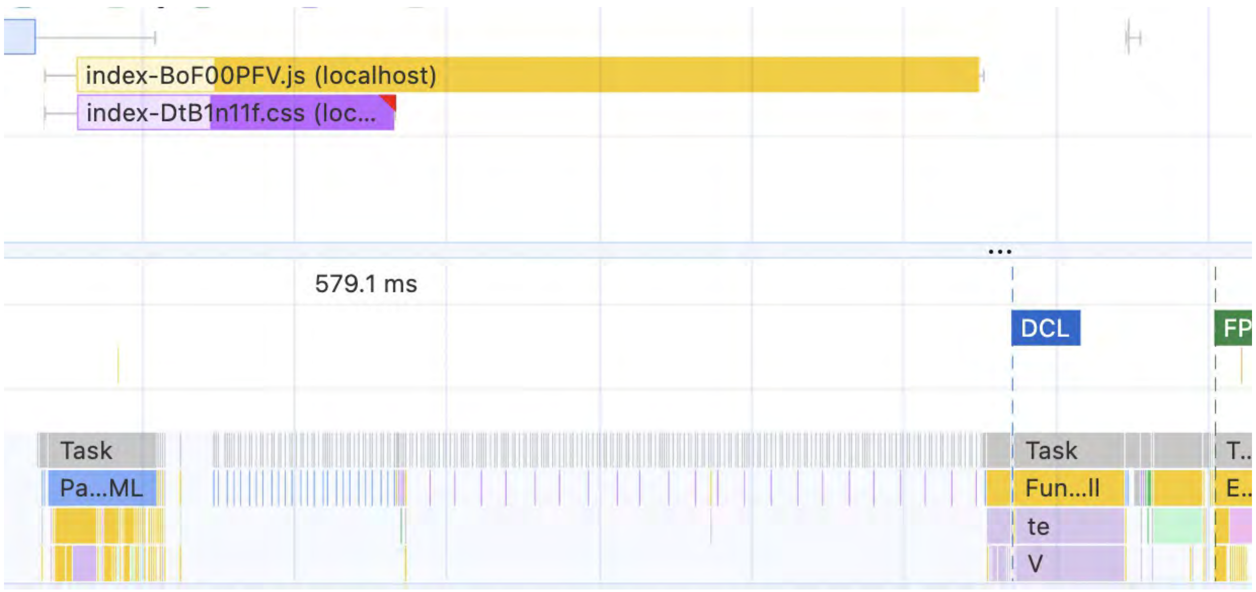
Downgrade the speed even more just to see how it progresses. Create a new Network Profile with 10 Mbps/1 Mbps for Download and Upload, keep the 40 latency, and name it "Low Internet Bandwidth".



Profile Name	Download	Upload	Latency
Low inter	10000	1000	40
	optional	optional	optional

Run the test again.

The LCP value has increased to almost 500 ms now. The JavaScript download takes almost 300 ms. The Parse HTML task and JavaScript execution tasks are shrinking in importance, relatively speaking.



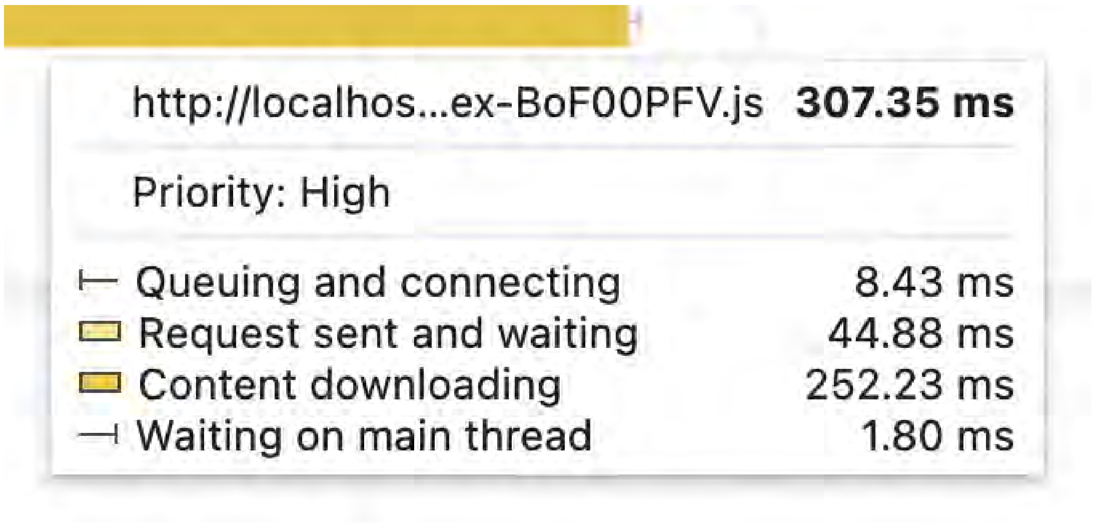
Additional Challenge.
If you have your own project, try to run this test on it.

- How long does it take to download all the critical path resources?
- How long does it take to download all the JavaScript files?
- How much of a gap does this download cause after the Parse HTML task?
- How large are the Parse HTML and JavaScript execution tasks in the main

thread relative to the resource downloading?

- How does it affect the LCP metric?

What's happening inside the resources bar is also quite interesting. Hover over the yellow JavaScript bar. You should see something like this there:



```
http://localhos...ex-BoF00PFV.js  307.35 ms
Priority: High
⊢ Queuing and connecting            8.43 ms
  Request sent and waiting         44.88 ms
  Content downloading             252.23 ms
⊣ Waiting on main thread            1.80 ms
```

The most interesting part here is the "Request sent and waiting", which takes roughly 40 ms. Hover over the rest of the Network resources - all of them will have it. That's our Latency[38], the network delay, that we set to 40 ms. Many things can influence the latency numbers. The type of network connection is one of them. For example, an average 3G connection has a bandwidth of 10/1 Mbps and a latency between 100 and 300 ms.

To emulate that, create a new Network Profile, call it "Average 3G", copy the download/upload numbers from the "Low Internet bandwidth" profile, and set the latency to 300 ms.

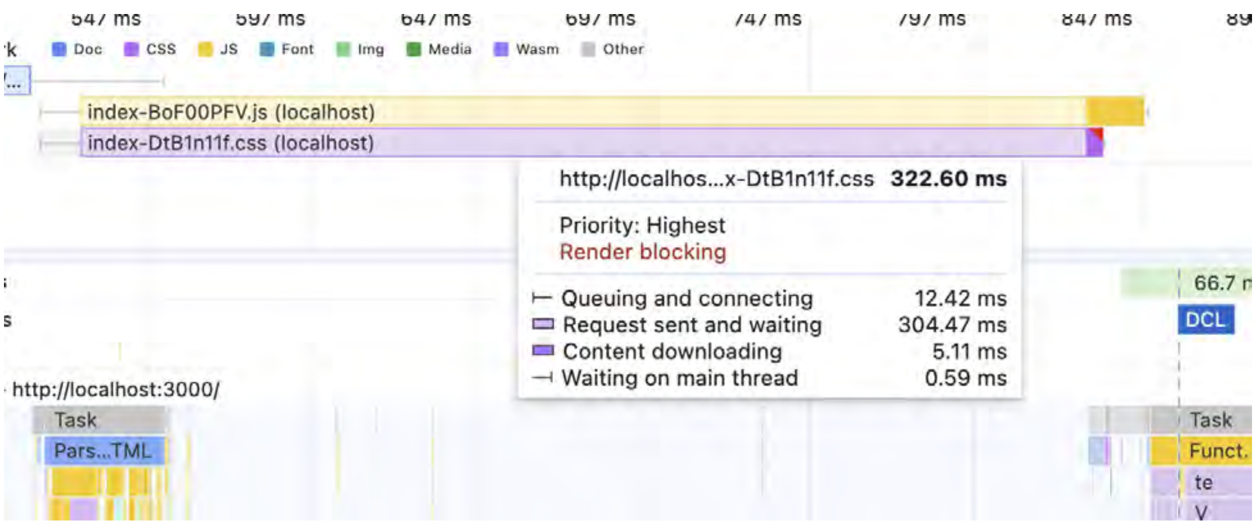
Run the profiling again. All the Network resources should have "Request sent and waiting" increased to around 300 ms. This will push the **LCP** number even further: **1.2 seconds** for me.

And now the fun part: what will happen if I revert the bandwidth to ultra-high speeds but keep the low latency? Let's try this setting:

- **Download**: 1000 Mbps
- **Upload**: 100 Mbps
- **Latency**: 300 ms

This can easily happen[39] if your servers are somewhere in Norway, but the clients are rich Australians.

This is the result:



http://localhos...x-DtB1n11f.css **322.60 ms**

Priority: Highest
Render blocking

Queuing and connecting	12.42 ms
Request sent and waiting	304.47 ms
Content downloading	5.11 ms
Waiting on main thread	0.59 ms

The **LCP** number is around **960 ms**. It's worse than the slowest internet speed we tried before! In this scenario, bundle size doesn't matter much, and the CSS size doesn't matter at all. Even if you halve both of them, the LCP metric will barely move. High latency trumps everything.

This brings me to the very first performance improvement everyone should implement if they haven't yet. It's called "make sure that the static resources are **always** served via a CDN".

The Importance of CDN

The CDN is basically step zero in anything frontend-performance related, before even beginning to think about more fancy stuff like code splitting or Server Components.

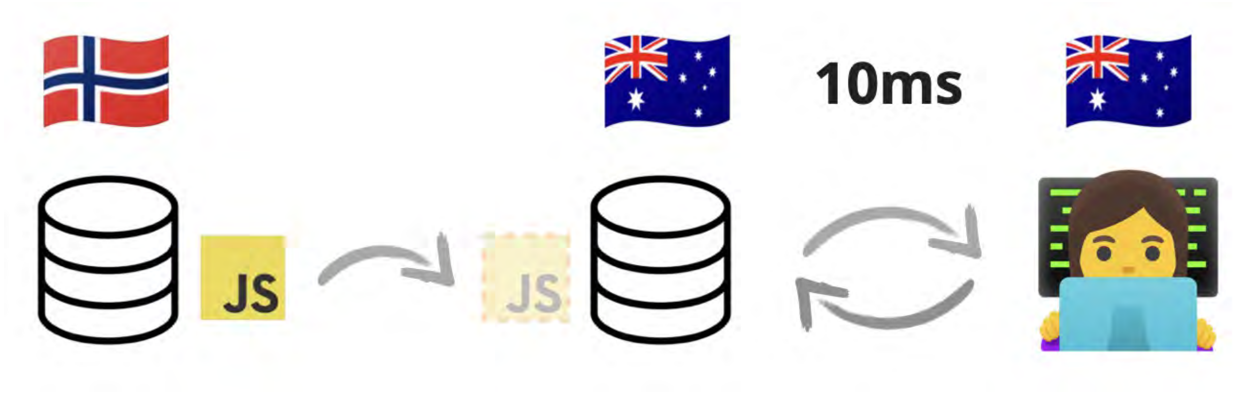
The primary purpose of any CDN[40] (Content Delivery Network) is to reduce latency and deliver content to the end user as quickly as possible. They implement multiple strategies for this. The two most important ones for this chapter are "distributed servers" and "caching".

A CDN provider will have several servers in various geographical locations. These servers can store a copy of your static resources and send them to the user when the browser requests them. The CDN is basically a soft layer around your origin server that protects it from outside influence and minimizes its interaction with the outside world. It's kind of like an AI assistant for an introvert, which can handle typical conversations without the need to involve the real person.

In the example above, where we had servers in Norway and clients in Australia, we had this picture:

With the CDN in between, the picture changes. The CDN will have a server somewhere closer to the user, likely in Australia as well. At some point, the CDN will receive copies of the static resources from the origin server. After it does that, any user from Australia or anywhere nearby will receive those copies rather than the originals from the server in Norway.

This achieves two important things. First, the load on the origin server is reduced since users no longer need to access it directly anymore. Second, users will get those resources much quicker now since they don't have to reach across oceans to download a few JavaScript files.



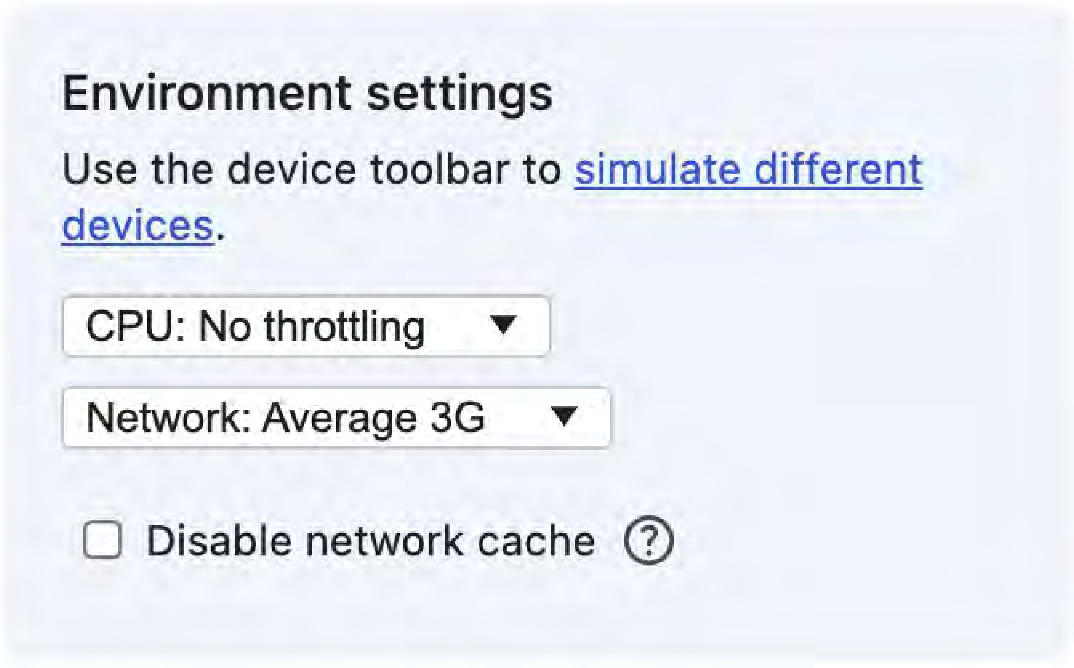
And the LCP value in our simulation above drops **from 960 ms back to 640 ms** 🎉.

Repeat Visit Performance

Up until now, we have only been talking about first-time visit performance - performance for people who've never been to your website before. But hopefully, the website is so good that most of those first-time visitors turn into regulars. Or at least they don't leave after the first load, navigate through a few pages, and maybe buy something. In this case, we usually expect browsers to cache static resources like CSS and JS. I.e., to save a copy of them locally rather than having to download them each time.

Let's take a look at how the performance graphs and numbers change in this scenario.

Open the Study Project again. In the DevTools, set the Network to the "Average 3G" we created earlier. The one with high latency and low bandwidth, so that we can see the difference right away. And make sure that the "disable network cache" checkbox is unchecked.



Environment settings

Use the device toolbar to simulate different devices.

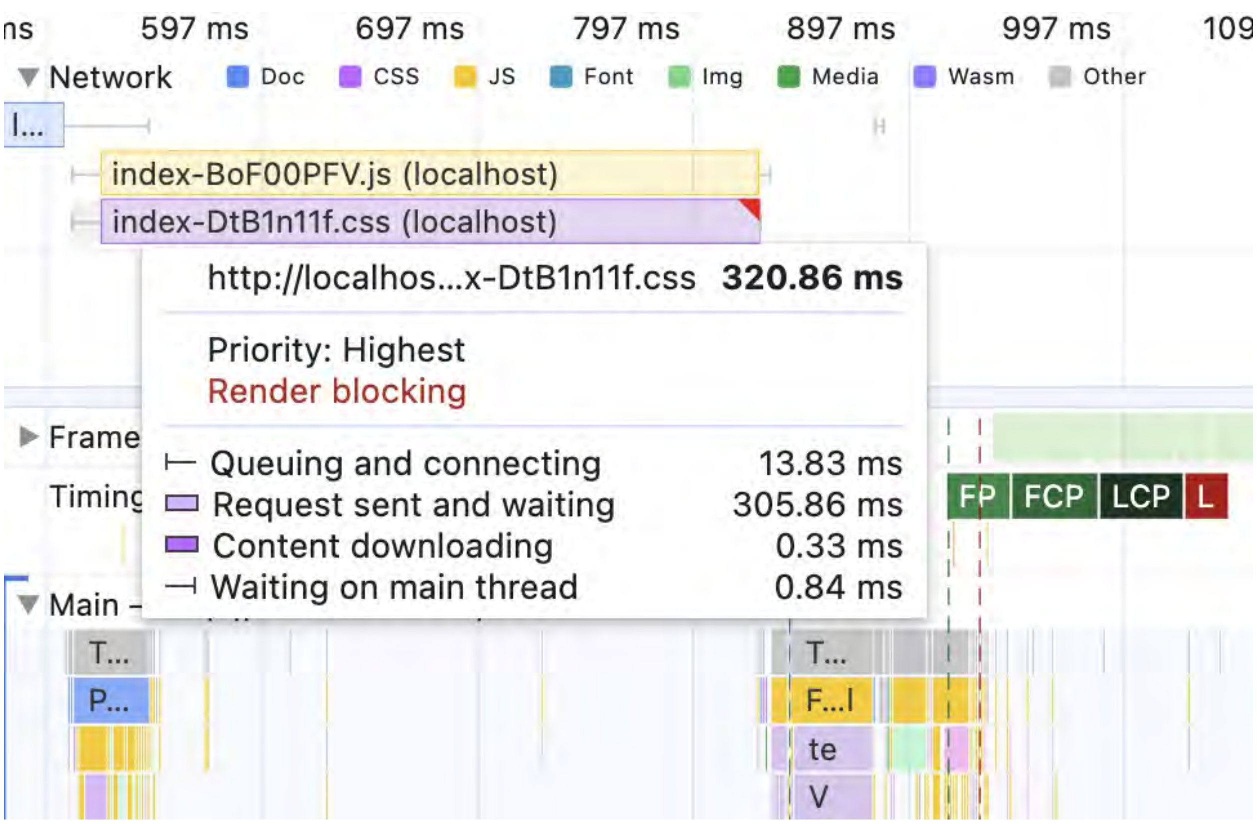
CPU: No throttling

Network: Average 3G

☐ Disable network cache

First, refresh the browser to make sure that we're eliminating the first-time visitor situation. Then reload and measure the performance.

If you're using the Study Project, the end result should be slightly surprising because it will look like this:



http://localhos...x-DtB1n11f.css **320.86 ms**

Priority: Highest
Render blocking

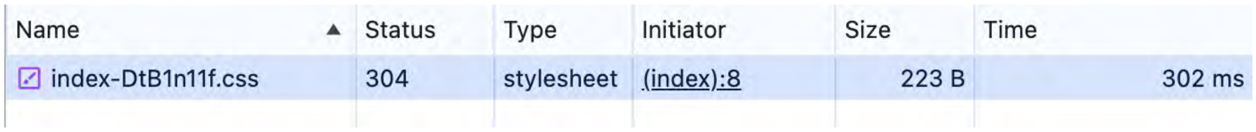
Queuing and connecting	13.83 ms
Request sent and waiting	305.86 ms
Content downloading	0.33 ms
Waiting on main thread	0.84 ms

The CSS and JavaScript files are still very prominent in the network tab, and I see ~300 ms for both of them in "Request sent and waiting" - the latency setting we have in the "Average 3G" profile. As a result, the LCP is not as low as it could be, and I have a 300 ms gap when the browser just waits for the blocking CSS.

What happened? Wasn't the browser supposed to cache those things?

Controlling Browser Cache with Cache-Control Headers

We need to use the Network panel now to understand what's going on. Open it and find the CSS file there. It should look something like this:



Name	Status	Type	Initiator	Size	Time
index-DtB1n11f.css	304	stylesheet	(index):8	223 B	302 ms

The most interesting things here are the "Status" column and the "Size" column. In "Size", it's definitely not the size of the entire CSS file. It's too small. And in "Status", it's not our normal 200 "all's okay" status, but something different - 304 status.

Two questions here - why 304 instead of 200, and why was the request sent at all? Why didn't caching work?

First of all, the 304 response[41]. It's a response that a well-configured server sends for conditional requests[42], where the response varies based on various rules. Requests like this are quite often used to control browser cache.

For example, when the server receives a request for a CSS file, it could check when the file was last modified. If this date is the same as in the cached file on the browser side, it returns the 304 with an empty body (that's why it's just 223 B). This indicates to the browser that it's safe just to re-use the file it already has. There is no need to waste the bandwidth and re-download it.

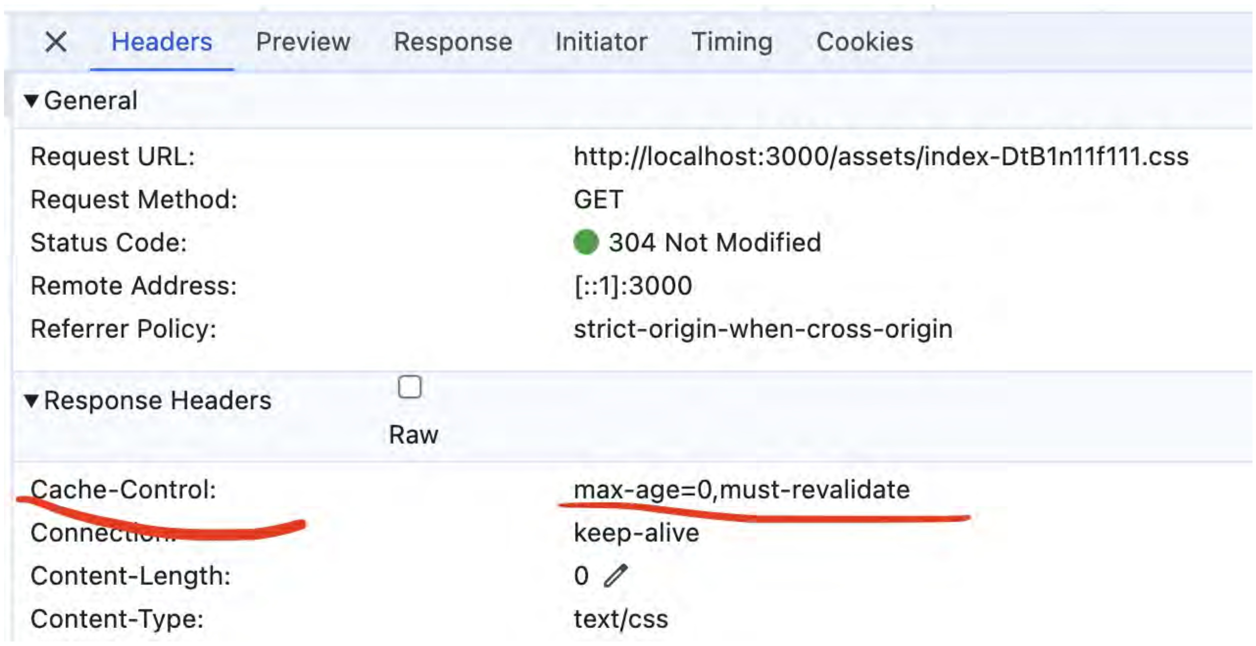
That's why we see the large "request sent and waiting" number in the performance picture - the browser asks the server to confirm whether the CSS file is still up-to-date. And that's why the "content downloading" there is 0.33 ms - the server responded with "304 Not Modified" and the browser just re-used the file it downloaded before.

Additional Challenge.

1. In the Study Project, go to the "dist/assets" folder and rename the CSS file.
2. Go to the dist/index.html file and update the path to the renamed CSS file.
3. Refresh the already opened page with the opened Network tab. You should see the CSS file appear with the new name, 200 status, and the proper size - it was downloaded again. It's known as **"cache-busting"** - a way to force the browser to re-download resources it might have cached.
4. Refresh the page again - it's back to the 304 status and re-using the cached file.

Now, to the **second question** - why was this request sent at all?

This behavior is controlled by the Cache-Control[43] header that the server sets to the response. Click on the CSS file in the Network panel to see the details of the request/response. Find the "Cache-Control" value in the "Headers" tab in the "Response Headers" block:



General

Request URL:	http://localhost:3000/assets/index-DtB1n11f111.css
Request Method:	GET
Status Code:	● 304 Not Modified
Remote Address:	[::1]:3000
Referrer Policy:	strict-origin-when-cross-origin

Response Headers

Cache-Control:	max-age=0,must-revalidate
Connection:	keep-alive
Content-Length:	0
Content-Type:	text/css

Inside this header can be multiple directives in different combinations, separated by a comma. In our case, there are two:

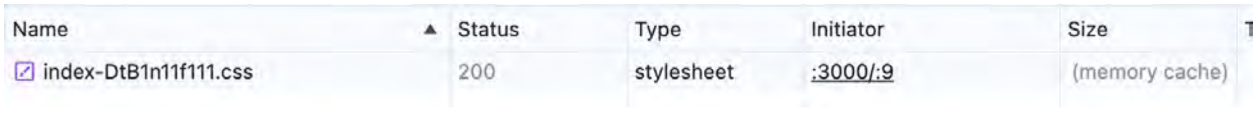
- **max-age** with a number - it controls for how long (in seconds) this particular response is going to be stored.
- **must-revalidate** - it directs the browser to always send a request to the server for a fresh version if the response is stale. The response will turn stale if it lives in the cache for longer than the `max-age` value.

So basically, what this header tells the browser is:

- It's okay to store this response in your cache, but double-check with me after some time to make sure.
- By the way, the time that you can keep that cache is exactly **zero** seconds. Good luck.

As a result, the browser *always* checks with the server and never uses the cache right away.

We can easily change that, though. All we need is to change that `max-age` number to something between 0 and 31536000 (one year, the maximum seconds allowed). To do that, in your Study Project, go to the `server.ts` file, find where `max-age=0` is set, and change it to 31536000 (one year). Refresh the page a few times, and you should see this for the CSS file in the Network tab:

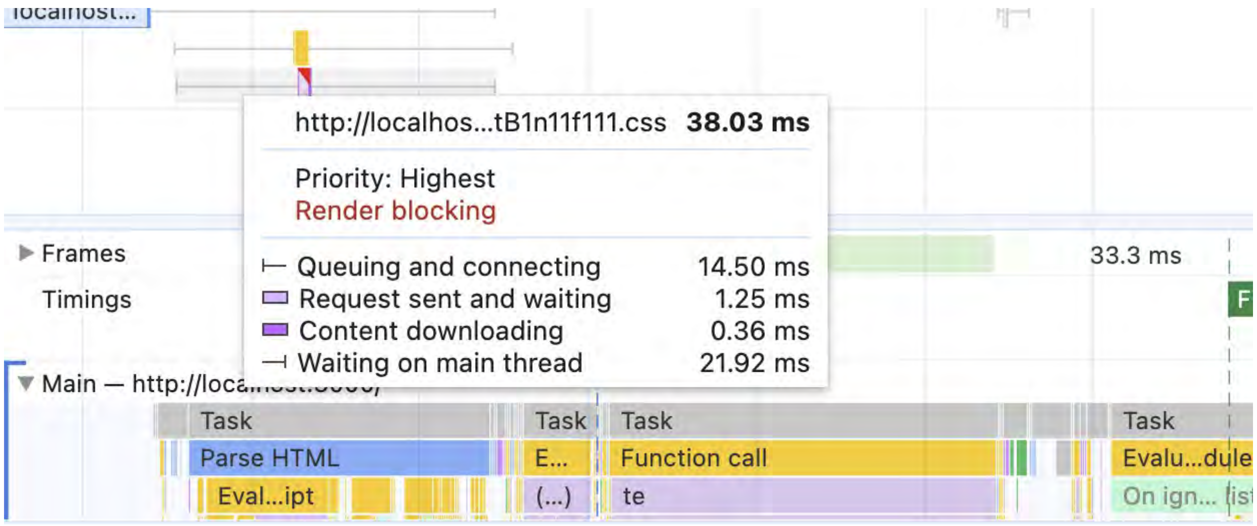


Name	Status	Type	Initiator	Size
index-DtB1n11f111.css	200	stylesheet	:3000/:9	(memory cache)

Notice how the `Status` column is grayed out now, and for `Size`, we see "(memory cache)". The CSS file is now served from the browser's cache, and it will be so for the rest of the year. Unless we enable the "disable cache" checkbox or clean it some other way, of course. Refresh the page a few times to see that it doesn't change.

Now, to the whole point of messing with the cache headers: let's measure the performance of the page again. Don't forget to set the "Average 3G" profile setting and keep the "disable cache" setting unchecked.

The result should be something like this:



http://localhos...tB1n11f111.css 38.03 ms

Priority: Highest
Render blocking

Queuing and connecting	14.50 ms
Request sent and waiting	1.25 ms
Content downloading	0.36 ms
Waiting on main thread	21.92 ms

The "Request sent and waiting" part collapsed to almost zero despite the high latency, the gap between "Parse HTML" and JavaScript evaluation almost disappeared, and we're back to ~650 ms for the LCP value.

Additional Challenge.

1. Change the `max-age` value to 10 now (10 seconds).
2. Refresh the page with the "disable cache" checkbox checked to drop the cache.

3. Uncheck the checkbox and refresh the page again - it should be served from the memory cache this time.
4. Wait for 10 seconds, and refresh the page again. Because the max-age is only 10 seconds, the browser will double-check the resource, and the server will return a 304 response again.
5. Refresh the page immediately - it should be served from memory again.

Cache-Control and Modern Bundlers

Does the above information mean that the cache is our performance silver bullet and that we should cache everything aggressively as much as possible? Absolutely not! Aside from everything else, the chance to create a combination of "not tech-savvy customers " and "need to explain over the phone how to clear browser cache" will cause panic attacks for the most seasoned developers.

There are a million ways to optimize the cache, a million combinations of directives in the Cache-Control header in combination with other headers that may or may not influence how long the cache lives, which also may or may not depend on the server's implementation. Probably a few books' worth of information can be written just on this topic alone. If you want to become the Master of Cache, start with articles on https://web.dev/[44] and MDN resources[45], and then follow the breadcrumbs.

Unfortunately, nobody can tell you, "This is the five best cache strategies for everything". At best, the answer can be: "If you have this use case, in combination with this, this, and this, then this cache settings combination is a good choice, but be mindful of those hiccups". It all comes down to knowing your resources, your build system, how frequently the resources change, how safe it is to cache them, and what the consequences are if you do it wrong.

There is, however, one exception to this. An exception in a way that there is a clear "best practice": JavaScript and CSS files for websites built with modern tooling. Modern bundlers like Vite, Rollup, Webpack, etc., can create "immutable" JS and CSS files. They are not truly "immutable", of course. But those tools generate file names with a hash string that depends on the file's content. If the file's content changes, then the hash changes, and the name of the file changes. As a result, when the website is deployed, the browser will re-fetch a completely fresh copy of the file regardless of the cache settings. The cache is "busted", exactly like in the exercise before when we manually renamed the CSS file.

Take a look at the `dist/assets` folder in the Study Project, for example. Both JS and CSS files have `index-[hash]` file names. Remember those names and run the build command a few times. The names stay exactly the same since the content of those files hasn't changed.

Now go to `src/App.tsx` file and add something like a `console.log('bla')` somewhere. Run the build command again, and check the generated files. You should see that the CSS file name stays exactly as it was before, but the JS file name changes. When this website is deployed, the next time a repeat user visits it, the browser will request a completely different JS file that never appeared in its cache before. The cache is busted.

> **Additional Challenge.**
> Find the equivalent of the `dist` folder for your project and run your build command.
>
> - What do the names of the files look like? Similar with hashes, or plain `index.js`, `index.css`, etc?
> - Do the names of the files change when you rerun the build command again?
> - How many file names change if you make a simple change somewhere in the code?

If this is how your build system is configured, you're in luck. You can safely configure your servers to set the maximum `max-age` header for generated assets. If you similarly version all your images, even better, you can include images in the list as well.

Depending on the website and its users and their behavior, this might give you a pretty nice performance boost for the initial load for free.

Do I Really Need to Know All of This For My Simple Use Case?

By this time, you might be thinking something like, "You're insane. I built a simple website over the weekend with Next.js and deployed it to Vercel/Netlify/HottestNewProvider in 2 minutes. Surely, those modern tools handle all of this for me?". And fair enough. I also thought that. But then I actually checked, and boy, was I surprised 😅

Two of my projects had `max-age=0` and `must-revalidate` for CSS and JS files. Turned out it's the default in my CDN provider 🤷‍♀️. They, of course, have a reason for this default. And luckily, it's easy to override, so no big deal. But still. Can't trust anyone or anything these days 😅.

What's Next?

Now that the initial load and how to find and measure it are clear, why CDN is step zero when it comes to performance, and Cache Control is step one, it's time to talk more about things that are actually related to React. It is the "React Performance" book after all.

In the next chapter, we'll take a look at React's "default" render pattern. I.e., Client-Side Rendering. And while doing so, as a nice consequence, we'll learn how to record and read Performance Flame Graphs. You know, those giant flame-like structures that everyone loves to show off when performance is involved, but confuse the hell out of anyone who's trying to read them for the first (or tenth) time.

4. Client-Side Rendering and Flame Graphs

In the previous chapter, we figured out how to start measuring performance and measured Initial Loading and LCP for a few different scenarios. Armed with this information, we can now start talking about different render patterns and what's the point of them.

We're going to start with the most popular pattern in the React world: Client-Side Rendering. We'll look into what it does and the cost of doing it. For that, we'll look into what a Performance Flame Graph is, how to read it in a simple scenario, and how to extract something useful from it for more real-life apps.

Client-Side Rendering

Let's start with a definition. What is Client-Side Rendering?

Actually, it's what we investigated in the previous chapter! There, we had a React project that, when built, turned out to be just an `index.html` file with an empty div, a JS file, and a CSS file inside. Something like this:

```html
<html lang="en">
  <head>
    <script type="module" crossorigin src="/assets/index-Cx2U5bbX.js"></script>
    <link rel="stylesheet" crossorigin href="/assets/index-BjPt9w-2.css" />
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

The entire beautiful Home page that we see when we start this project is generated entirely by the JavaScript file and React inside of it.

React takes everything that we wrote, converts it into DOM nodes, finds the element with the id `root`, and appends the newly created DOM elements to that element using your regular JavaScript commands. It looks something like this:

```
// React grabs our App and does whatever it needs to do with it
const domElements = ReactDoYourThing(App);

// Get the "root" element with your regular JavaScript
const rootDomElement = document.getElementById('root');

// Append the elements to the root
rootDomElement.appendChild(domElements);
```

After the browser is done executing all this JavaScript, the user suddenly sees the entire page appear out of the blue.

If you look inside the `main.tsx` file, you'll see almost exactly the same code there:

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './index.css';
import App from './App';

createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
);
```

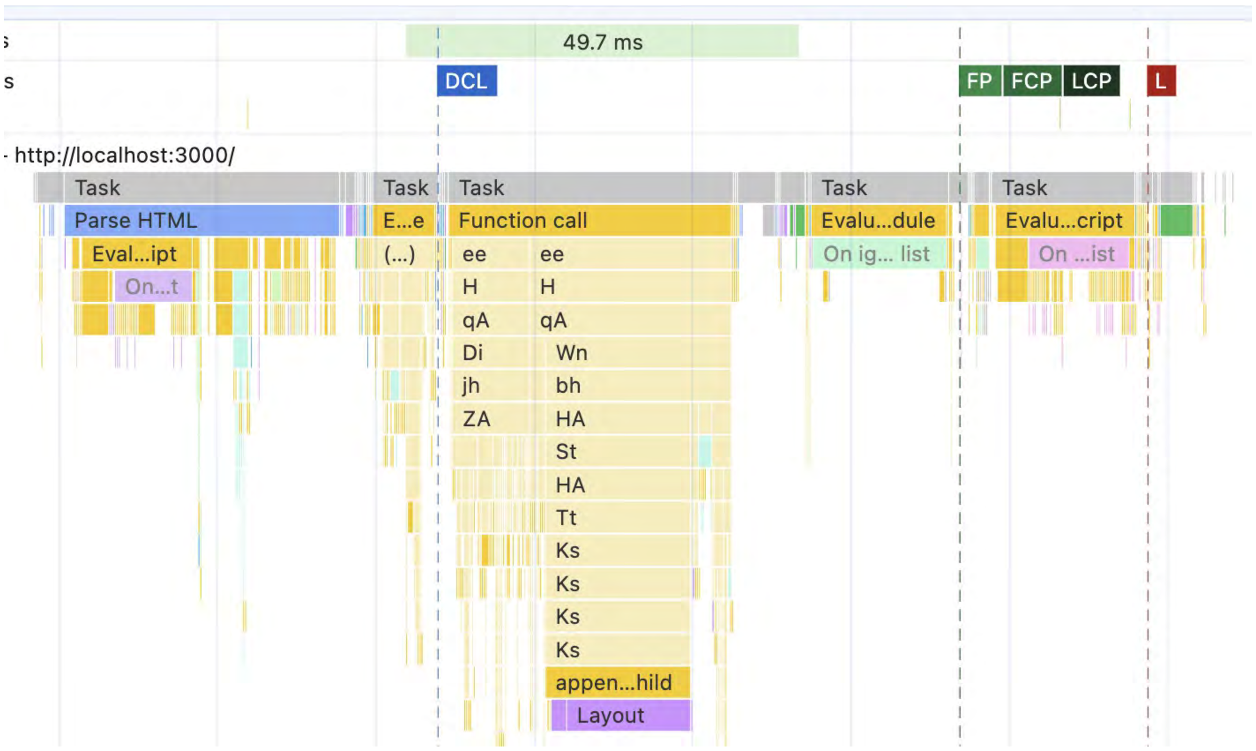
This is the entry point to any React app. This process of generating the full DOM with JavaScript and then injecting it into an empty page is what we know as **"Client-Side Rendering"**.

What does it mean from a performance perspective?

To understand that truly, we need the Performance panel again, this time the "main" section. Let's build and start this chapter's Study Project (`src/chapter4-client-side-rendering/simple-frontend`):

```
npm run build --workspace=chapter4-simple-frontend
npm run start --workspace=chapter4-simple-frontend
```

Open the Chrome DevTools, go to the Performance panel, record the performance, uncollapse the "main" section, and you should see a pretty obvious picture like this:



Okay, I'm joking, of course, about "pretty obvious". Unless you've been reading these graphs every day for years for fun and profit, this picture will probably tell you nothing. So, before continuing with React again, let's figure out what it means and how to extract something useful from it.

Reading the Flame Graph

This picture is known as the "Flame Graph[46]" or sometimes a Flame Chart. Graphs like this represent an application's call stack relative to the resources you want to investigate, like memory or, in our case, time in milliseconds.

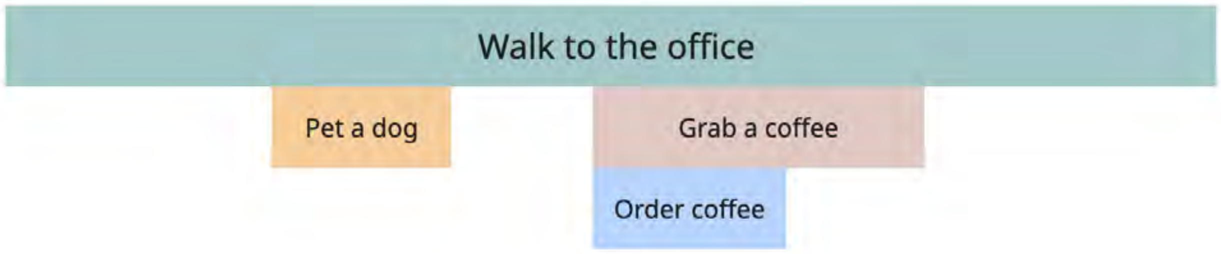
Imagine a function that describes the daily commute of a developer:

- A Developer leaves home and starts walking while humming a tune and listening to a podcast.
- Then they stop to pet a cute dog.
- After some more walking, they stop to grab a coffee, which consists of:
  - Waiting in line for a minute or two to order the coffee.
  - Waiting a few more minutes to actually get the coffee.
- After that, they continue walking while drinking coffee.

The entire flow is the function `goToWork`. While executing, this function at some point triggers the `petADog` function and waits until it's finished. After that, the function `getSomeCoffee` is triggered. This function also triggers another function - `orderCoffee`. After its execution, the `getSomeCoffee` function waits for the coffee and returns it after it's done. After that, the core function `goToWork` continues until the Developer arrives at their destination.

```
const goToWork = async () => {
  await petADog();
  const coffee = await getSomeCoffee();
  await finishWalking(coffee);
};
```

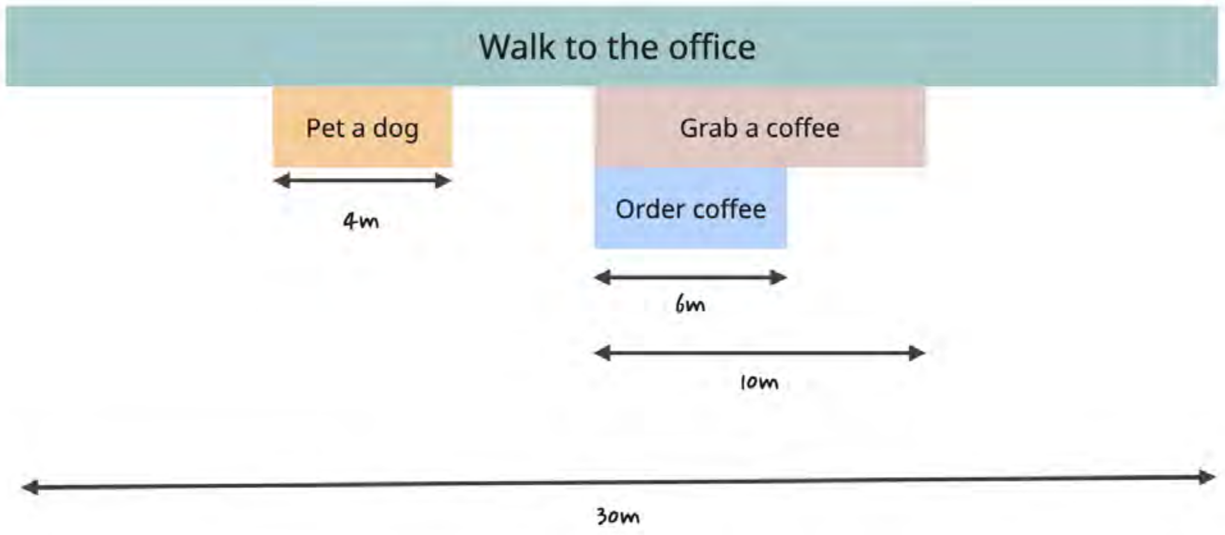
This describes what is known as the Call stack[47]. In the flame graph, it will look like this:



The most important thing about this graph right now is that it clearly shows us which task originated from where:

- The "Pet a dog" task was triggered by "Walk to the office," and it's completely independent of the coffee situation.
- The "Order coffee" task originated from the "Grab a coffee" task, which was also triggered by the "Walk to the office" task.
- The "Order coffee" task has nothing to do with the "Pet a dog" task - it's not within its hierarchy. If the "Grab a coffee" task disappears one day, the "Order coffee" will disappear with it since it's its "child," but the "Pet a dog" task will never notice it.

Another important thing here is that the graph shows the duration of all the tasks.



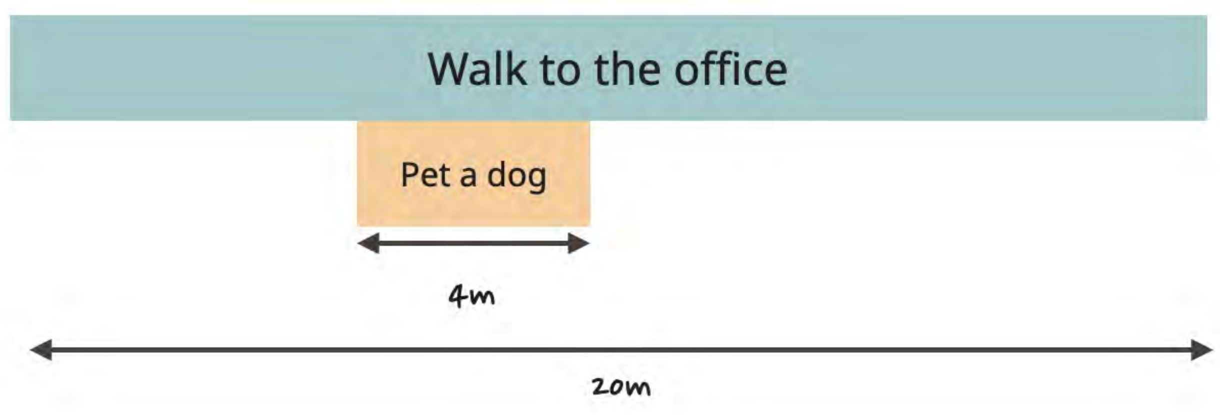
For example, the "Grab a coffee" task took 10 minutes, of which the developer spent 6 minutes waiting in line to place the order. These 10 minutes are known as **"Total Time"** - the time it takes to execute a function, including the execution time of *all* its children. The Total Time for the "Walk to the office" is 30 minutes, out of which it took 4 minutes to execute "Pet a dog" and 10 minutes for "Grab a coffee," during which 6 minutes were spent in line ordering the coffee.

The difference between the Total Time and the time it took for the children to execute is known as **"Self Time"**. In this example (30 - 4 - 10), it's 16 minutes for the "Walk to the office" function. These 16 minutes are the time the developer spent actually walking. For "Grab a coffee," it's 4 minutes.

This knowledge of Self Time, Total Time, and the structure of the children enables us to make some performance optimizations. Most importantly, it allows us to calculate trade-offs and the cost of optimizations.

Let's say I want to reduce the "Walk to the office" time to 20 minutes. What are my options?

I could just get rid of the "coffee" task completely and accomplish the goal immediately.

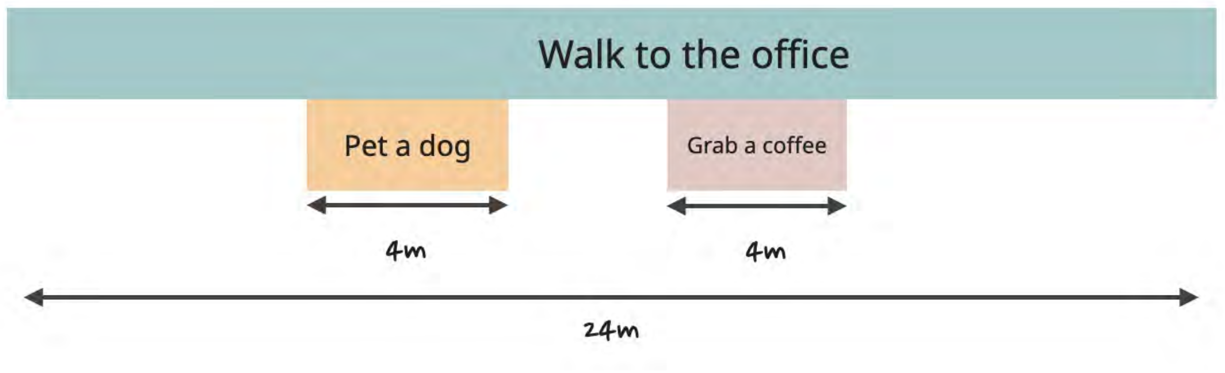


The cost here would be a significant impact on the quality of life of the developer, plus productivity for the rest of the day. It might not be the best option unless you *really* need a quick win *right now*.

Instead, I could get rid of the "Order coffee" task by pre-ordering coffee through an app while walking. The cost here is pretty minimal, just need to download an app and create an account there.

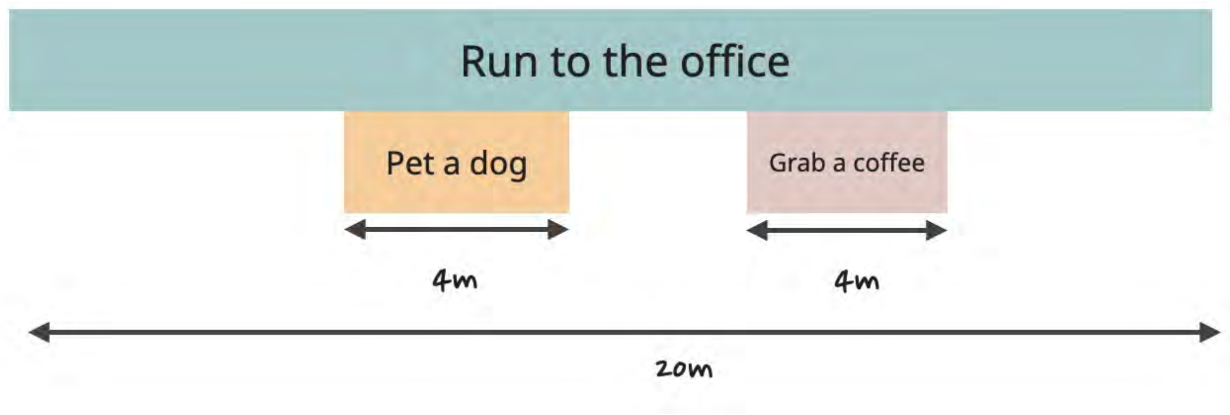
The "Grab a coffee" task is still present - the developer still needs to walk to the counter, find the coffee there, and squeeze through the crowds to get out. So, the function's Self

Time will remain.



Now I just need to find another 4 minutes. I could get rid of the "Pet a dog" task, but similar to the "coffee" task, the cost here would be the impacted mood of the developer and their productivity for the rest of the day.

Another option would be to reduce the Self Time of the "Walk to the office" task itself. I could buy the developer a scooter or a bike, for example, to speed it up. Or arrange some sort of transport that picks up employees and delivers them to work. Or work with the government to organize more footpaths with no streetlights that slow down the walk. Or just encourage the developer to run instead of walk somehow. Imagine that! 😅

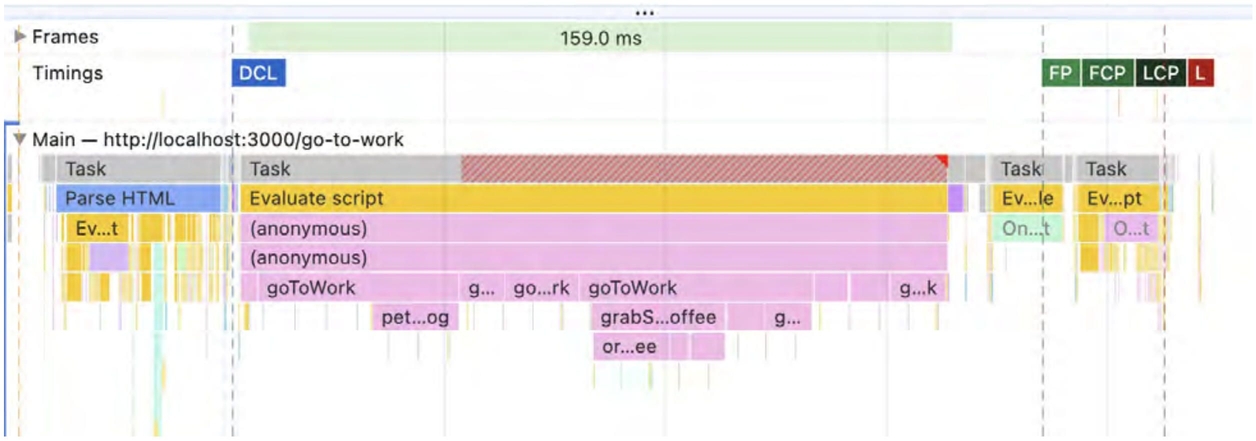


In any case, the performance goal is accomplished again, assisted by the Flame Graph.

Reading the Flame Graph in DevTools

Time to switch back to the real graph.

Start this chapter's Study Project again, navigate to http://localhost:3000/go-to-work, and record the performance again. The graph for this page looks exactly like the illustration above! Kinda.



Okay, it slightly resembles the illustration above. That's because the script is run in the browser, and the profiler records *everything* that is happening, which is still a lot, even for the simplest script. But at least now we can read it and see that:

1. At first, there was the blue bar of "Parse HTML" - we know that one already. It happens when the HTML is received from the server. It triggered some JavaScript for some reason, but we don't care since it has nothing to do with our "walk to work" task. If you right-click on it and select "Hide children," we can even eliminate that visual distraction.
2. Then, there was a loooong yellow line of JavaScript. This is the script that implements the "go to work" functionality. That one triggers all of our functions, and their sequence looks pretty much like the pic above.
3. Only after the long yellow JavaScript line is done, the FCP/LCP metrics are triggered. This tells us that the metrics can happen only when both "Parse HTML" and the JavaScript tasks are done.

To confirm the latest assumption, we can take a look at the actual script and HTML page. You can find them in `src/chapter4-client-side-rendering/simple-frontend/assets`.

The HTML looks something like this:

```html
<html>
  <head>
    <script async src="./main.js"></script>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

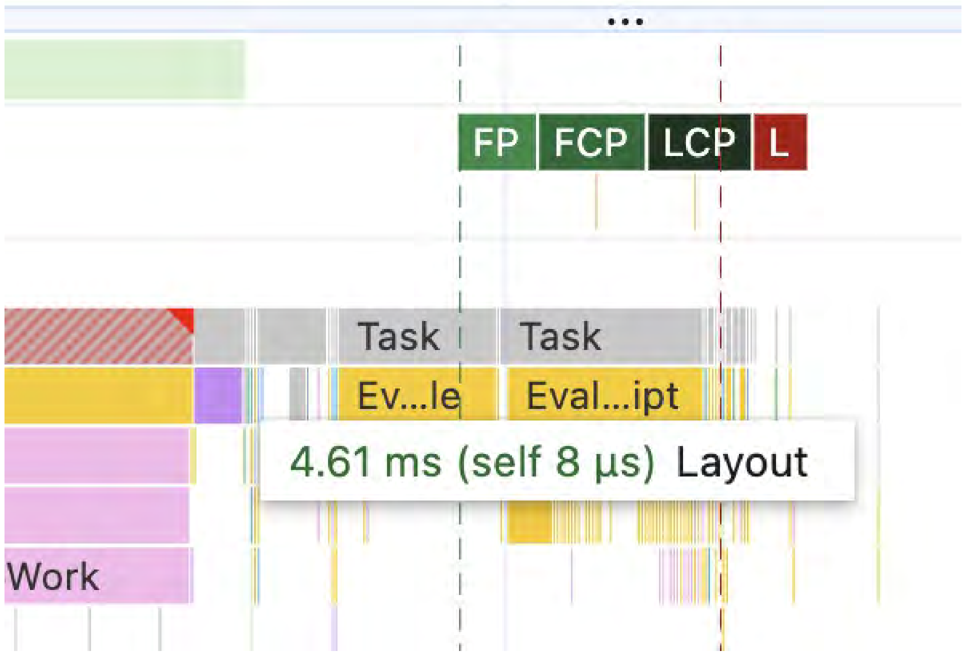
Very typical Client-Side rendering HTML. And in the `main.js` script, at the very end, there is this:

```javascript
const div = document.createElement('div');
div.innerText = 'Arrived!';
document.getElementById('root').appendChild(div);
```

After the synchronous function above is done, JavaScript creates a DOM element and appends it to the existing `<div>` on the page. This corresponds really nicely with what's happening in the Flame Chart - until the JavaScript finished its work, there was no content on the page, so the FCP/LCP metrics couldn't be triggered.

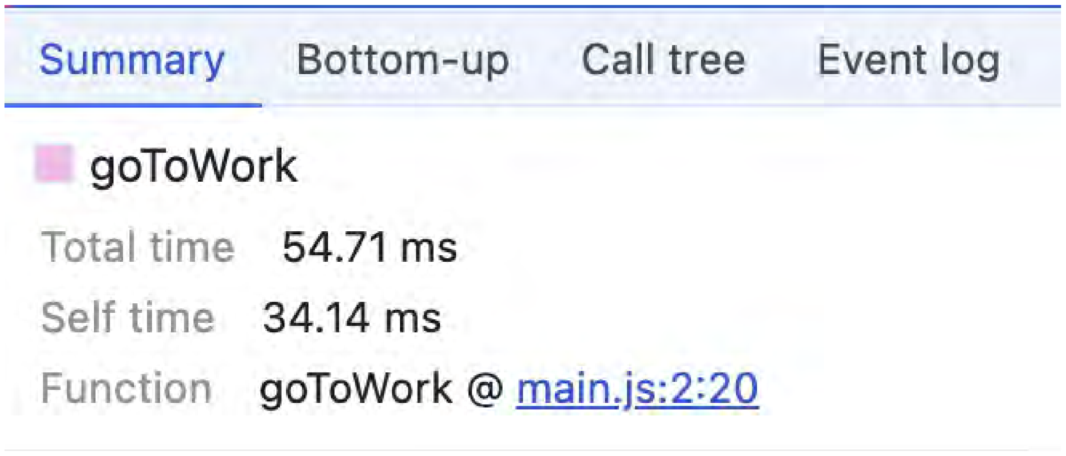
What else is interesting here before we go back to the Flame Chart of an actual app?

First of all, there is a tiny purple block on the same line as the yellow JavaScript line. If you hover over it, you'll see that it's a "Layout"[48] task.



That one is quite important. This is where the browser calculates how the elements should be positioned on the screen and their exact dimensions. In this case, it's tiny since we're only adding one div to the page. But for large apps, it can be very visible.

Another important thing here is that all those lines are interactive. Clicking on any of them will update the detailed information in the tabs at the very bottom. Click on one of the "goToWork" bars, for example. In the "Summary" tab, there will be something like this:



goToWork

Total time	54.71 ms
Self time	34.14 ms
Function	goToWork @ main.js:2:20

We found our **"Total time"** and **"Self time"**. And more!

There is also the "**Function"** part here: this is what triggered this particular execution.

Clicking on it will open the exact code where it started.

Additional Challenge.

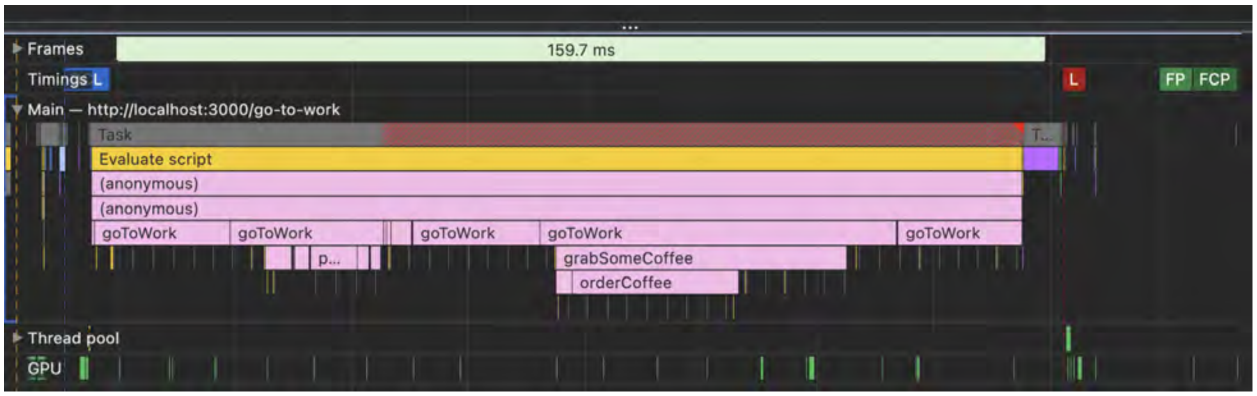
- Click on the "goToWork" block that produced the "petADog" function. Notice the Total and Self time.
- Click on the "petADog" block. Its total time should roughly match the difference between the Total and Self time in the block that produced it.
- Now click on the link in the "Function" block. It should navigate you to the exact file and the exact line where this function was called.
- Click on the yellow JavaScript line that produced the functions. Instead of "Function", it should have a "Script" there with a link to the script that produced that line.

Different blocks will have different information in the Summary tab.

This actually allows us to solve another mini-mystery of this graph: where are all those additional yellow JavaScript blocks coming from? The ones under the Parse HTML block and after our main "goToWork" task?

If you click through them, you should see their origin in the "Summary" tab in the "Script" part. All of them are various Chrome plugins! To see how much those plugins are messing with the performance graph (and the page performance in general), open the "Go to work" exercise in Chrome incognito mode or in Chrome "Guest" profile and re-record the performance there.

For me, it looks like this now:



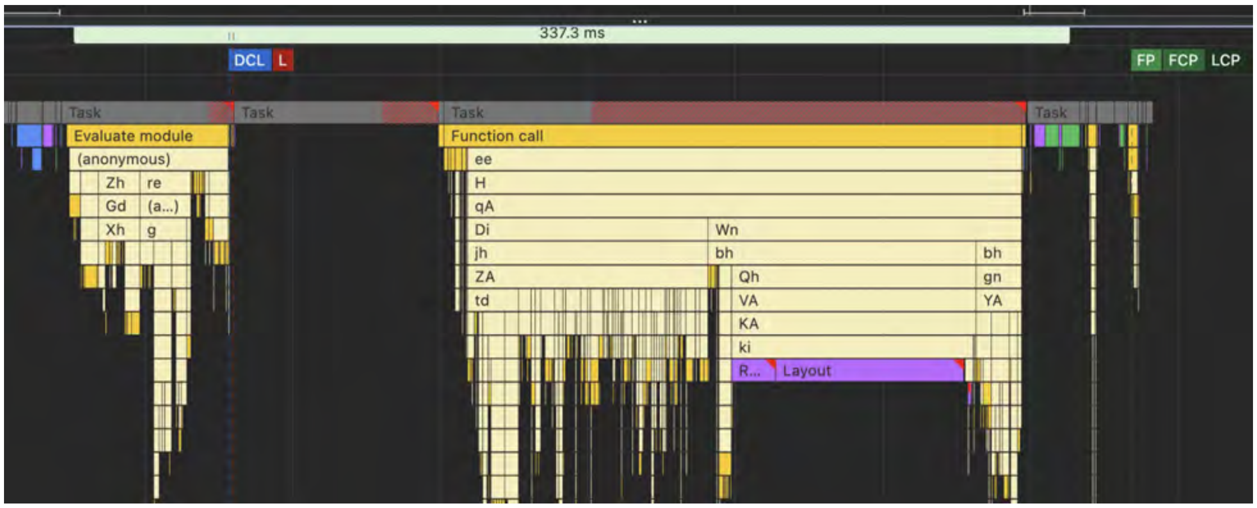
The "Parse HTML" task collapsed to oblivion. No more long-tail JavaScript tasks. The graph is so much clearer.

So this is a big lesson here: before doing anything with this graph, like panicking about the "Parse HTML" task being too long, make sure the outside influence, like third-party plugins, is eliminated.

Client-Side Rendering of a Real App in the Performance Panel

Now that we know how to read the Flame Chart, let's take a look at the chart of our semi-real Home page again. This time, let's be smart right away and open it from the "guest" profile to get rid of all visual distractions. If you're on a very fast computer, it might be helpful to also enable "CPU slowdown" - that will make the bars longer and easier to read. It's in the same place where we configured Network throttling, so it's easy to find.

Can you read what's happening here now?

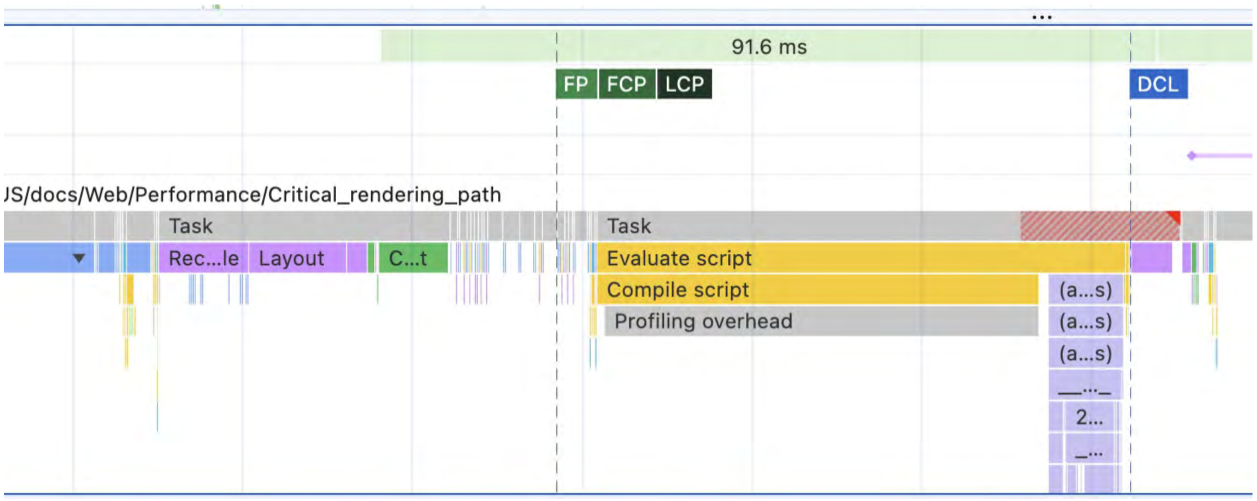


At first, there's "Parse HTML" again - the tiny blue bar at the very beginning. Then, there are two long yellow bars of JavaScript. Clicking on them shows us where they originated from - our `index.js` file with all the React stuff and the app I wrote.

Only after the JavaScript is done, the FCP/LCP metrics appear. That also checks out - the app uses Client-Side Rendering, so there is no content to trigger those metrics until the JS is done.

The "Layout" block is now huge and takes up almost a third of the second yellow JavaScript bar. That's because it's now the entire page, not just a tiny div like we had in the previous exercise.

The fun part here comes from comparing this graph with a graph from a more traditional, non-client-side rendered website. Let's run it on any of the MDN pages[49], for example. The picture will be drastically different, almost in reverse:



As always, there's a blue "Parse HTML" at the beginning. But then, it's followed by a purple block of Layout right away and a green block of Painting[50] (this is when the browser actually paints all the pixels of the calculated Layout). We have the entire page of the MDN docs visible on the screen by this time, so FCP/LCP is triggered.

Only after that do I see a yellow JavaScript bar that does something. And somewhere far, far after the Holy Grail of initial loading, LCP, is triggered, we see the DCL metric (DOMContentLoaded event) - this is when the browser is done parsing HTML and waiting for the scripts to load and execute.

So, what exactly was the point of looking at all of this, you might ask?

Because reading those graphs demonstrates very clearly the cost of doing Client-Side Rendering:

1. The initial load and LCP metric will ***always*** suffer, even when the scripts are already downloaded and cached by the browser. We always need to wait for their execution to see anything on a page.
2. Without JavaScript, there will be no website at all, the user will just get a blank page. Try disabling JavaScript on the MDN page we used to measure performance and on the Study Project to see the difference.

Wait, this feels terrible! Why would anyone want to do that??

First of all, because it's easy and cheap. ***Incredibly*** easy and cheap, to be precise. It's possible to build something complex and fully functional, serve it to thousands of users daily, and still pay zero dollars for hosting and not worry about scaling the system, outages, memory leaks, and all the other "backend-y" stuff. The entire app is just a few static files that can be deployed anywhere without the need for any supervision.

And second, and sometimes even more importantly, LCP and initial load are not the only things that matter when it comes to users and perceived performance. Sometimes, no one cares whether the app loads in 1 second, 1.2 seconds, or even 5 seconds, especially if there's a cute animation while the loading happens. Maybe the users came to the app not to look at it but to interact with it, and they are happy to wait a bit during the initial loading time to get exceptional performance later.

Think of something like project management software. No one goes there just to admire the view or read poems (I hope). People open those apps to do important daily tasks, which almost always involve lots and lots of interactions with the app. After it's loaded!

What's Next?

Now that we know what Client-Side Rendering is and how to read Performance Flame Graphs, we're well prepared for the next step: SPAs (Single-Page Applications) and why they're so cool that they've captivated developers for a very long time.

SPAs are a natural extension of Client-Side Rendering, and they really shine when smooth and *very fast* transitions between pages take priority over the initial load. To measure those, we need to know metrics like INP (Interaction to Next Paint) and where to find them, to start with.

That's exactly what we'll go through in the next chapter.

5. SPAs and Introducing INP

In the previous chapter, we figured out what Client-Side rendering is, how it affects the initial load, and, as a nice consequence, learned how to read the Flame Graph in the Performance panel.

In this chapter, let's talk a bit more about Client-Side rendering, more specifically, about the part that makes it worth it from a performance perspective - SPAs. Here, we'll look into what an SPA is, the difference between SPA transitions and the "traditional" ones, and take a first look at the INP (Interaction to Next Paint) metric, what it means, and how to measure it.

SPA (Single-Page Applications)

An SPA (Single-Page Application) is a client-side rendered app that has multiple pages and handles routing between them without involving the server. To see why it's so cool, we need the Study Project again (`src/chapter5-spa-and-inp/simple-frontend`). Build it and run it in production mode, as usual:

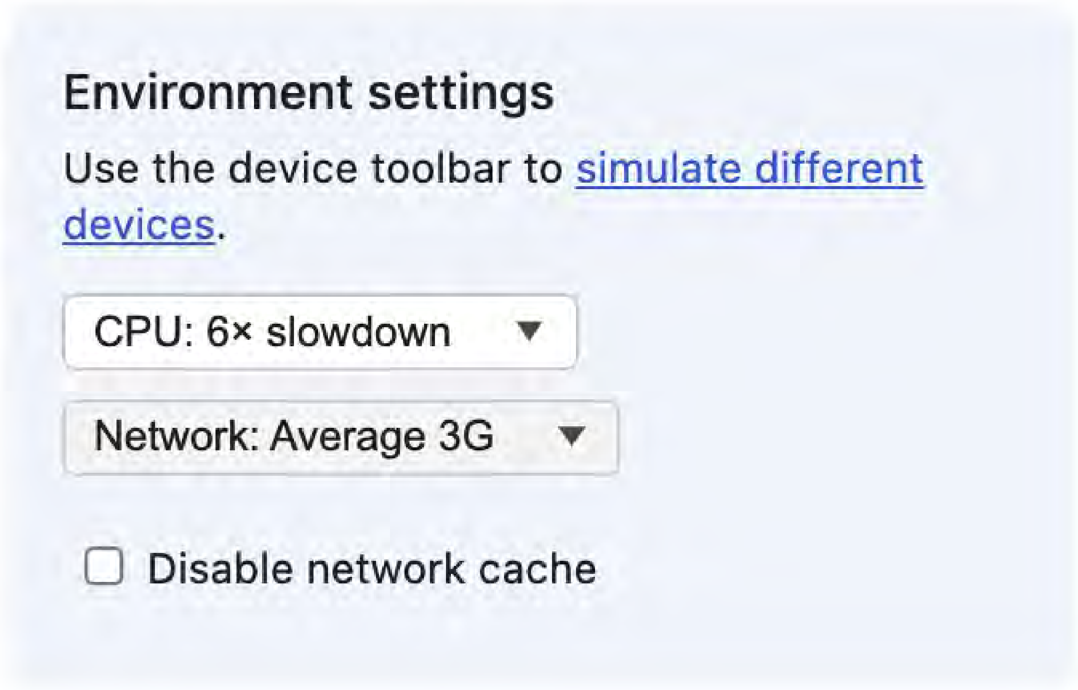
```
npm run build --workspace=chapter5-simple-frontend
npm run start --workspace=chapter5-simple-frontend
```

You should be able to see a beautiful dashboard on the screen.

We're going to work with a few pages here. Click on the logo, and you'll be redirected to the "Login" page. Click on the "Welcome to Product" title to return to the "Dashboards" page. From there, click on "Settings" in the sidebar navigation. This will redirect you to the "Settings" page. Click on the "Home" item to return to the "Dashboards" page.

Do you notice the difference? Even on the latest MacBook, the response should be, "Yeah, something feels off about the navigation to the Settings page compared to the Login page".

Let's downgrade ourselves to a slow device on an average 3G network to really feel it. Open the Performance tab and set the CPU to 6x slowdown and the Network to the "Average 3G" we configured in the previous chapter. Or just the default 3G if you skipped that part.


Environment settings

Use the device toolbar to simulate different devices.

CPU: 6× slowdown

Network: Average 3G

☐ Disable network cache

Navigate back and forth again. This time, the difference should be obvious. Navigating between "Dashboards" and "Settings" takes 1 second. Between the "Dashboards" and "Login" pages, the transition is almost instantaneous.

This is because the navigation between the "Dashboards" and "Login" pages is implemented as client-side routing, which makes them an SPA transition. Somewhere in the code, there is a `Link` component, inside of which there is this code:

```
<a
  onClick={(e) => {
    e.preventDefault();
    navigate(href);
  }}
>
  {children}
</a>
```

A normal `<a>` tag, where the default behavior on click is prevented - i.e., the "normal" link redirect won't happen. Instead, the navigation to the next page is triggered by JavaScript via `navigate(href)`. Which is not a "navigation to the next page" per se, actually. Inside this `navigate` function, there will be something like this:

```
window.history.pushState({}, '', newPath);
dispatchEvent(new PopStateEvent('popstate', { state: {} }));
```

Where `window.history.pushState` will simply update the URL part of the browser, and that's it, no redirects. The `dispatchEvent` part then dispatches a JavaScript event that can be listened to via `addEventListener`.

Somewhere else entirely, there will be a part that listens to that event and sets the state with the `pathname` value:

```
window.addEventListener('popstate', () => {
  setPath(window.location.pathname);
});
```

And then somewhere in the third place, there will be code that renders different pages based on that state value:

```
switch (path) {
  case "/login":
    return <LoginPage />;
  default:
    return <DashboardPage />;
}
```

That's what all the frameworks, like Next.js/Remix/Tanstack, etc, do for routing with various degrees of abstraction. All of them will give you a `Link` component that prevents the default link behavior and some ways to render different pages for different routes without reloading the page itself. From regex pattern matching to folder-based routing.

Additional Challenge.

- Open `App.tsx` file of the Study Project - it should look almost identical to the `switch` above.

- Trace down the implementation of the Logo in the sidebar navigation, the "Welcome to Product" title on the login page, and the rest of the items in the Sidebar.
- Why are some of them SPA transitions, and why are some of them not?
- What needs to be done here to create SPA transitions for all Sidebar items?

The end result is the same for them all: there is no more "traditional" page navigation, which includes asking the server for a new HTML, parsing it, and so on, the full cycle that we investigated before. Instead, the already initialized JavaScript just destroys the entire page (or part of it) with something like:

```
const root = document.getElementById('root');

root.innerHTML = '';
```

And then generates a new page and injects it to the old place:

```
const newChildren = generateNewPage();
root.appendChild(newChildren);
```

This part is usually invisible to us and is handled entirely by React.

What SPA Transitions Look Like in the Performance Panel

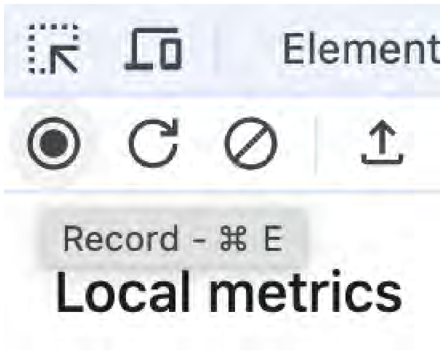
In the performance panel, those types of navigation look very different.

Build and run the Study Project if you haven't done it yet:

```
npm run build --workspace=chapter5-simple-frontend
npm run start --workspace=chapter5-simple-frontend
```

Open the website in a Chrome Guest Profile (or Incognito Mode). Always remember to build the production version when investigating general performance, and open the website in "guest" or "incognito" mode to eliminate outside influences.

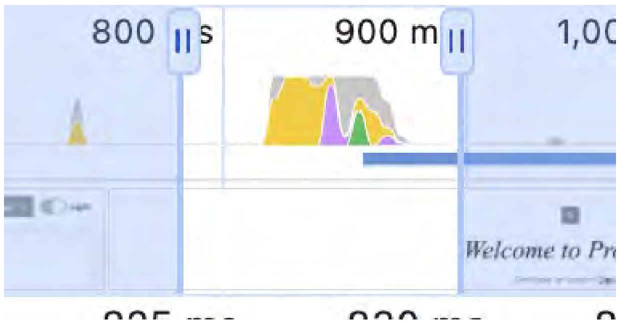
Open the performance panel, slow down the CPU by 6x, and set the network to Average 3G. Start the recording without reloading (the top left button).



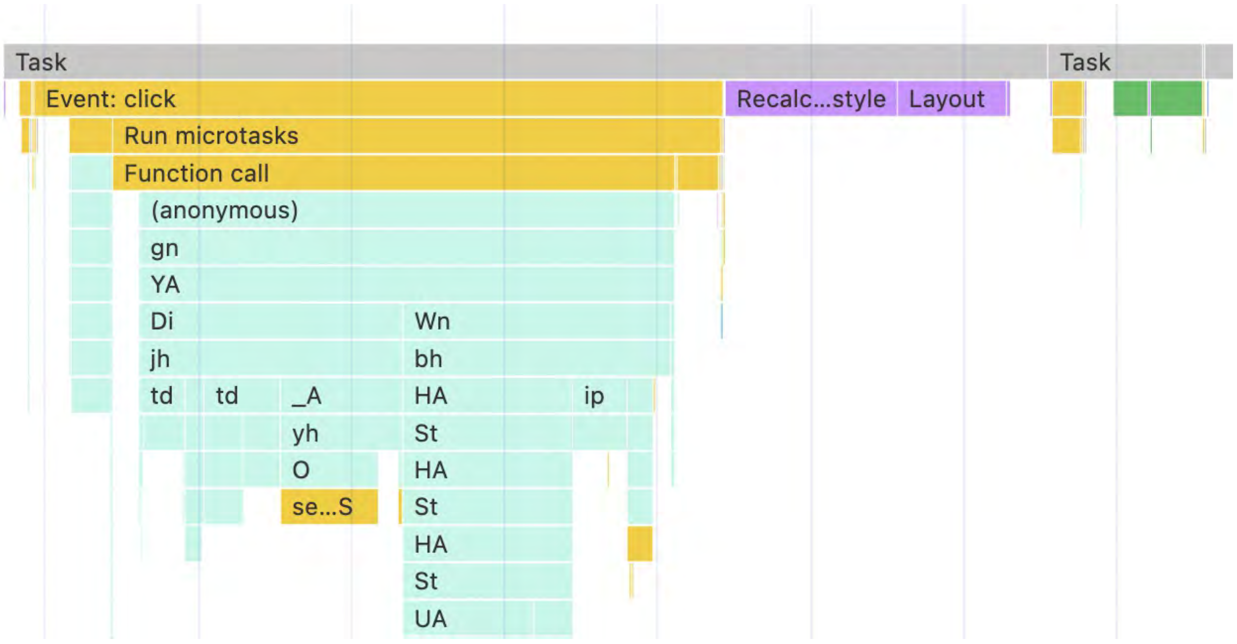
Click on the logo in the top left corner of the website to navigate to the Login page and stop the recording. What should be the result?

First of all, the **Network panel**. It should be absolutely empty, with no requests for external resources and no attempts to download anything.

Second, in the overall **Timeline overview**, you should see a spike in JavaScript activity. Zoom in on the largest one. Hover over it back and forth to confirm that this spike indeed caused the change in the UI. If you had the "screenshots" checkbox enabled, you'll see the exact moment when the UI changed from the "Dashboards" page to "Login".



In the **Main section**, you should see a more detailed picture of what was happening.



We can see that there was a "click" event here, which caused JavaScript to do something (we don't really care what exactly right now). After it was done, the browser recalculated styles (the first purple block), then did some Layouting (the second purple block), then the green blocks of Painting/Committing, and then the new Login page appeared.

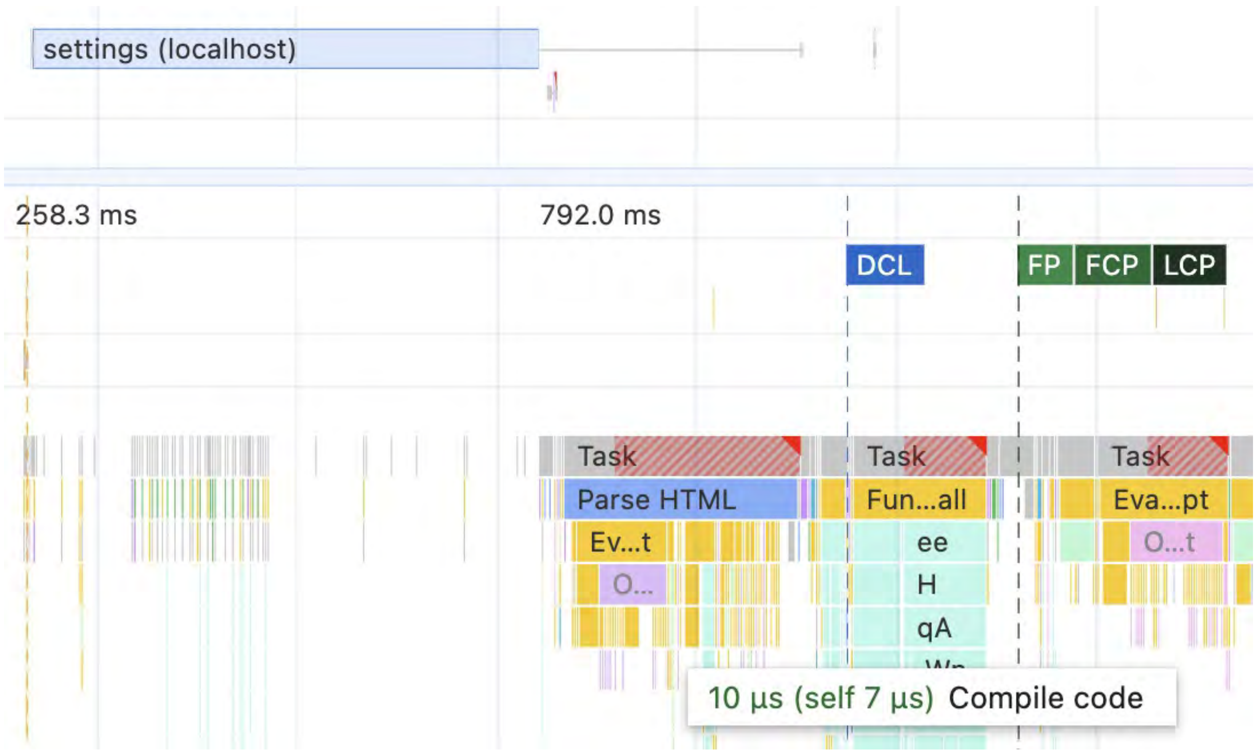
There are no FCP/LCP metrics anymore.

Overall, the entire navigation task took around 60 ms for me (with a slowed-down CPU and Network).

Additional Challenge.
Record performance while navigating from the Login page to the Dashboards by clicking on the "Welcome to Product" title.

- Which tasks took longer? Why do you think it happened?

Now, return to the "Dashboards" screen and repeat the same exercise, this time navigating to the "Settings" page (the one without the SPA transitions).



The LCP/FCP metrics are back again. There is also a new line that shows when the click interaction happened and when the navigation started.

The **"Network"** section is not empty anymore. The request to the server is back, CSS and JS files are back (although from the browser cache), and the Parse HTML task is back.

Overall, this interaction looks almost like a normal "Initial Load". This makes sense since this is exactly what happens when regular, non-SPA transitions via normal `<a>` tags are made.

The entire transition took almost a second for me. A little bit faster than the initial load for this page, but **more than 10 times slower** than an SPA transition.

Those super-fast "snappy" transitions between pages that are implemented so easily are what make SPA so attractive compared to the "traditional" page load. And the fact that we can make them as granular as we want, by the way. Who said that we should limit ourselves to only pages?

Additional Challenge.
Take a look at the "tabs" row on the "Settings" page. Right now, switching between tabs just changes the content, the URL doesn't change. As a result, when a tab is selected and the page is refreshed, the selected tab falls back to the default value. Can you make those tabs feel like "proper" navigation? So that:

- When I select the "Personal Profile" tab, the URL changes to "/settings/personal-profile".
- If I refresh the page, the "Personal Profile" tab remains selected. Hint:
    - Find how the `Link` component implements navigation.
    - Find how the current path value is extracted from the URL in the `App.tsx` file.

Okay, those transitions are super fast. But if there is no FCP/LCP anymore, how do I measure them then?

This is where the "Interaction to Next Paint"[51] (INP) metric comes in.

Measuring Interaction to Next Paint (INP)

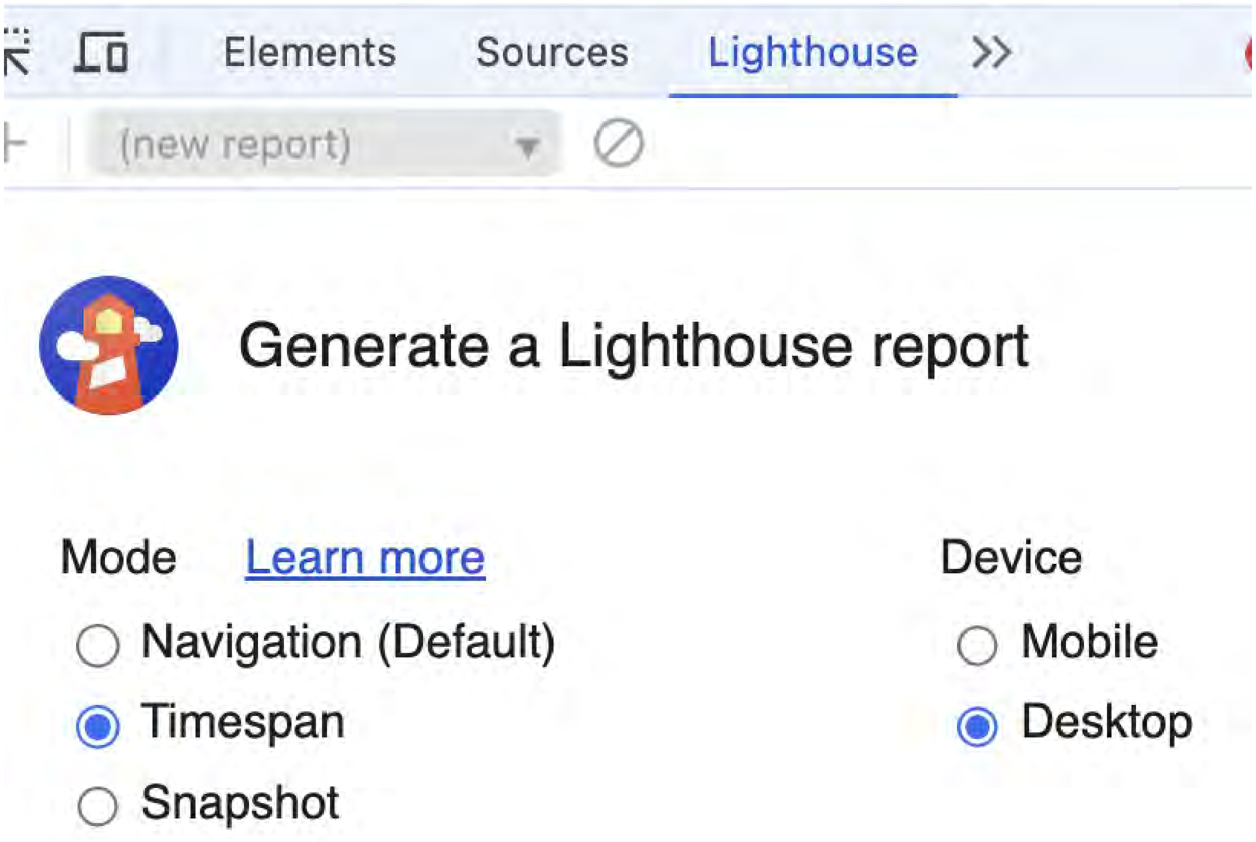
"Interaction to Next Paint" (INP) is a relatively new metric (introduced in 2023[52]) that measures how fast the app responds to users interacting with it. Basically, how "snappy" the interactions feel.

It's the second of the three Core Web Vitals[53] metrics, which are used by Google to measure the "quality" of a website. It's helpful to think of LCP (Largest Contentful Paint, the first Core Web Vital) as the "first impression" of the website, and INP as building continuous relationships with the users afterward.

It can be found in a few places in the Chrome DevTools.

Lighthouse Report

First, as a Lighthouse report. Similar to LCP, it will just give you a number to trace over time, but not much detail. To generate it, open the Lighthouse panel and select the "Timespan" there.



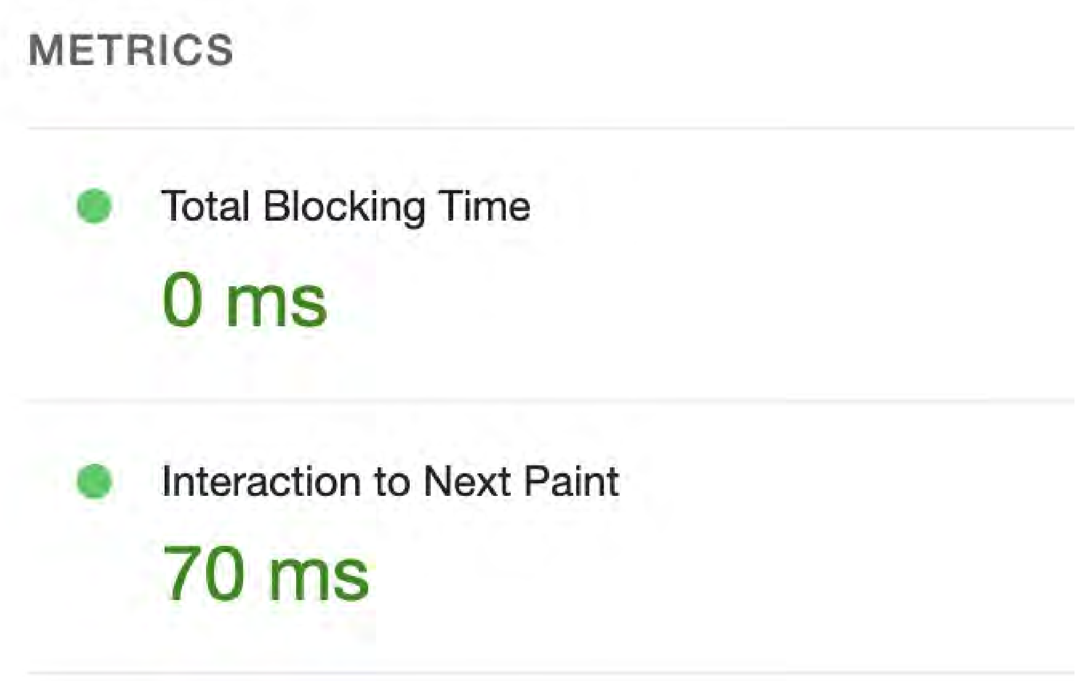
Mode

Navigation (Default) [✓ Timespan] Snapshot

Device

Mobile [✓ Desktop]

Start the recording and click through the website a few times to give the tool some material to work with. For example, in the Study Project, navigate from "Dashboards" to the Login page and then back a few times. Then stop the recording. Lighthouse will generate a report similar to what we saw for the initial load: overall scores, more detailed scores, and a bunch of optimizations you might want to consider.


METRICS

● Total Blocking Time

0 ms

● Interaction to Next Paint

70 ms

This INP score represents the *highest* value of INP among all the interactions you performed while the recording was active. So it will really depend on the interactions. Try toggling the toggle button on and off at the top right corner, instead of navigating back and forth. The result should be very different.

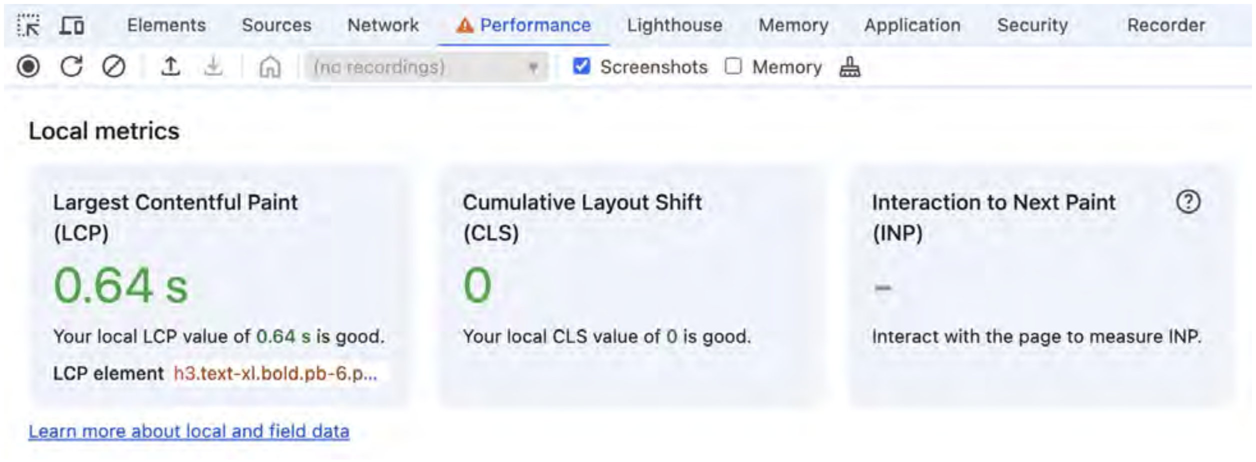
What is a "good" INP value is highly subjective, even more so than LCP. A 150 ms INP for navigating to another page is very good from a user perspective. But the same value in a complicated animation will cause it to look "junky" and definitely won't feel "good". However, to measure and improve something, we need to have at least some baseline, don't we?

We can use Google's guidelines[54] for this; they are pretty much the industry standard these days. Google considers the INP value "good" if it's below 200 ms. Between 200 ms and 500 ms, it's "meh, needs improvement". Over 500 ms is a terribly poor performance.

Performance Panel - Overall View

This overall number from Lighthouse could be useful for monitoring the general "health" of crucial interactions over time. However, it doesn't provide us with information on what exactly went wrong and how. For that, we again need the performance panel.

Open the panel, open the Study Project URL, clear everything, and refresh the page. In the panel, you should see something like this:



Local metrics

Largest Contentful Paint (LCP)
0.64 s
Your local LCP value of 0.64 s is good.
LCP element h3.text-xl.bold.pb-6.p...

Cumulative Layout Shift (CLS)
0
Your local CLS value of 0 is good.

Interaction to Next Paint (INP)
–
Interact with the page to measure INP.

Learn more about local and field data

The third block that is currently empty here is the INP value for the page. It's empty because the page just loaded, I haven't interacted with it yet. Try to click on stuff, open and close menus, and navigate back and forth to the SPA-backed pages.

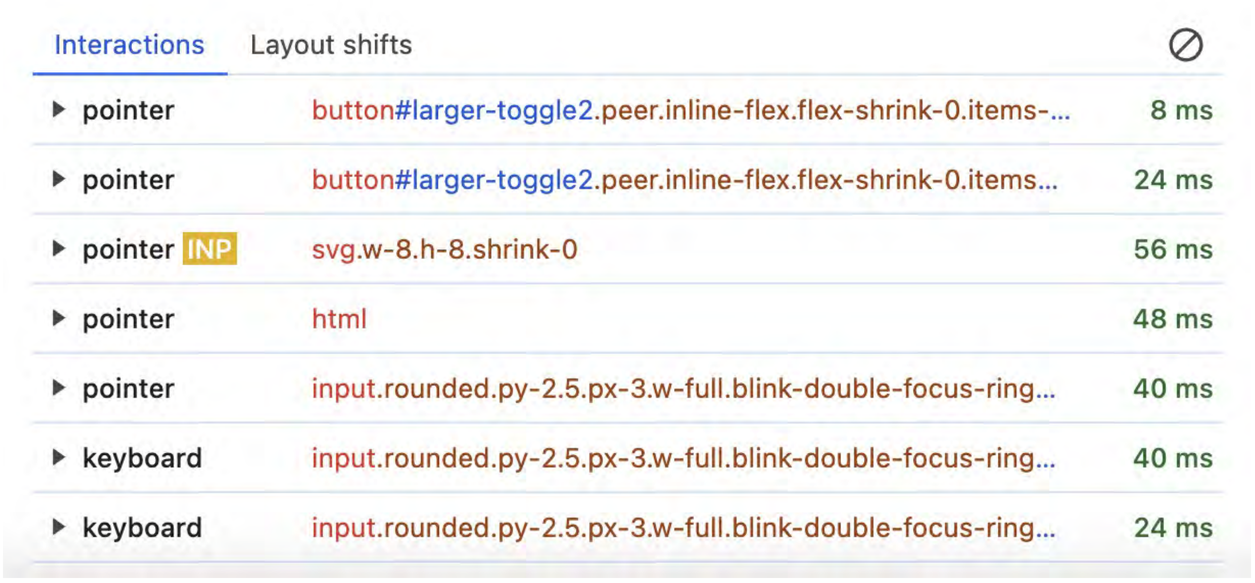
The number will constantly update itself to the highest value recorded on a page. If you start with the toggle and turn it on and off, it will record something like 20 ms there. Opening/closing the menus might update it to something like 40 ms. Navigating back and forth between Dashboards and the Login page will give you 140 ms. Your numbers will be very different and highly depend on the computer you use and the CPU slowdown you set.

Notice how navigating between the SPA pages (Dashboards and Login) keeps the score between the pages, but navigating between non-SPA pages (Dashboards and Settings) resets everything. This distinction is important to keep in mind when measuring and tracking performance as an overall "score". Especially if this score is used for something other than local debugging or detecting regressions in known scenarios.

In SPAs, navigation between pages is likely going to be the largest INP number. As a result, other interactions might easily be overlooked, and the overall user experience might not be as good as it could be just because of hyper-focusing on improving the overall score.

Performance Panel - Interactions View

The total page score is useful, but again, only to track overall performance over time. For debugging purposes, we need more information. We can find this in the "interactions" block in the Performance panel. In my version of Chrome, it's just below the three "total" blocks and looks like this:



▸ pointer	button#larger-toggle2.peer.inline-flex.flex-shrink-0.items-...	8 ms
▸ pointer	button#larger-toggle2.peer.inline-flex.flex-shrink-0.items...	24 ms
▸ pointer INP	svg.w-8.h-8.shrink-0	56 ms
▸ pointer	html	48 ms
▸ pointer	input.rounded.py-2.5.px-3.w-full.blink-double-focus-ring...	40 ms
▸ keyboard	input.rounded.py-2.5.px-3.w-full.blink-double-focus-ring...	40 ms
▸ keyboard	input.rounded.py-2.5.px-3.w-full.blink-double-focus-ring...	24 ms

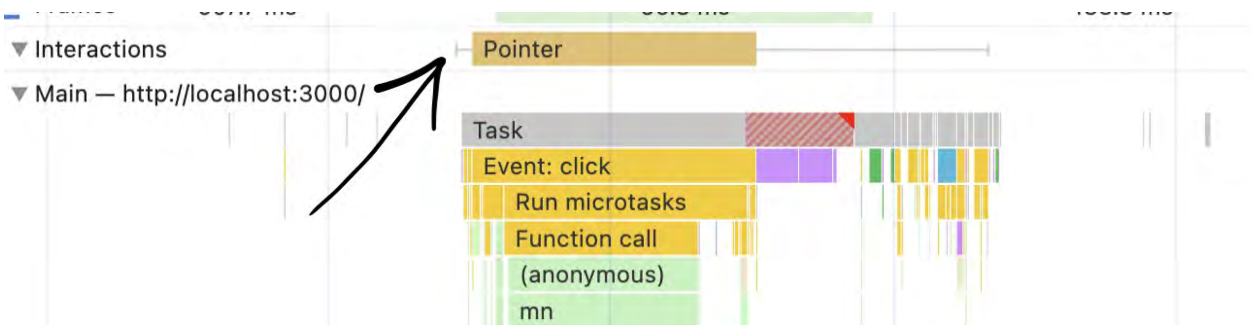
Here, we can see what caused an interaction (pointer, i.e., mouse or keyboard), where it originated, and how long it took. Hovering over the origin highlights the element on the screen, and clicking on it directs you to the exact code.

The yellow "INP" badge highlights the value that is used for the overall page's score. You'll see that value in the initial three blocks, and as you can see, it's the largest one here.

For even more detail, when we need to deeply understand the interactions' performance situation, we have the combination of "Main" and "Interactions" sections in the Performance panel recording.

Performance Panel - Performance Recording

We need the recording of the performance of the SPA transition again to find it. Navigate to the "Dashboards" page, start the recording, navigate to the "Login" page, and stop the recording. Zoom in on the JavaScript spike in the overview timeline where the screen changed to eliminate the distraction. And this time, pay attention to the new section above the "Main" called "Interactions".



That's exactly the INP recording we looked at before. Hovering over it will give us the exact value in milliseconds that we've seen before. But looking at it in this chart gives us much more than just numbers: it's now very visible what exactly is happening between the time the interaction was triggered and the time the browser recorded it as "done".

The dark yellow bar, which corresponds with the JavaScript execution time and represents exactly that, takes only half of this particular interaction. The rest of it is spent on the purple blocks of recalculating styles and layout, green blocks of Painting/Committing, and the rest of the tasks the browser needs to do in order to actually *show* the user the result of the interaction.

Additional Challenge.

1. Record the transition from Login to Dashboards. Which part of the interaction takes longer this time, compared to the navigation between Dashboards and Login?
2. Record transitions between Login and Dashboards with a 20x CPU slowdown. Which part of the interaction takes longer now? How do those numbers and graphs change if those transitions are "traditional" full-page reloads?
3. Navigate to the "Settings" page and record navigations between different tabs there. Which tab takes longer, what is the longest part of the interaction, and why?

What's Next?

Now that we know the basics of measuring and investigating performance locally and have a good idea of two of the most important performance metrics, it's time to put that knowledge to good use. Because SPAs and Client-Side rendering, while being very good for interaction performance, as we already know, have two big downsides:

- They don't work with disabled JavaScript.
- They affect initial load performance.

It's time to investigate and understand why exactly these two can become a problem, what can be done to fix them, and which problems the fixes introduce in return 😅. As a nice consequence, among other things, we'll learn about other rendering patterns, like pre-rendering, SSR, SSG, and what hydration is.

6. Intro to Rendering on the Server (SSR)

In the previous chapters, we learned the basics of the most important performance metrics these days (FP, FCP, LCP, INP), what CSR (Client-Side Rendering) and SPA (Single-Page Application) are, and why they are so popular. We also learned about two of the most significant downsides of CSR/SPA: it negatively affects the initial load and doesn't work in environments without JavaScript.

In this and the following chapters, we're going to focus on how to fix those downsides, what kind of problems those fixes bring, and how to fix those problems. Because relatively cheap and easy fixes have already ended, unfortunately. 😅 Everything else is a delicate balance between the time and resources invested and the expected gain.

Let's start with no-JavaScript environments. Those might be the most puzzling downside. Who disables JavaScript in their browser these days? It's enabled by default everywhere, pretty much nothing will work without it, and most people won't even know what JavaScript is to disable it anyway. Right?

Why No-JavaScript Environments Are So Important

The answer here is in the word "people". Or, more precisely, in the fact that real people are not the only ones who can access your website. Two major players in this area are:

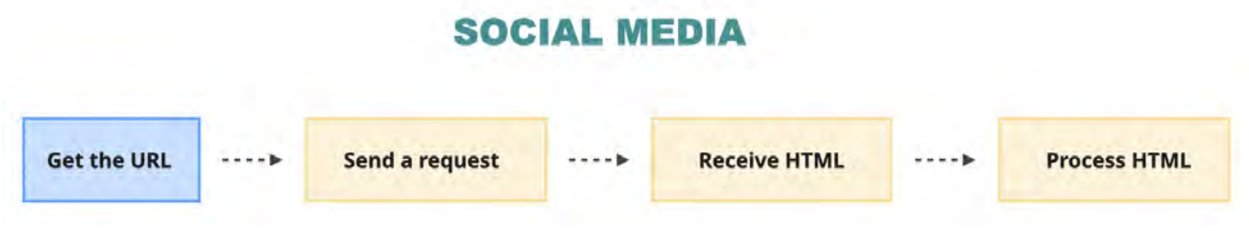
- Search engine robots (crawlers), especially the Google crawler.
- Various social media and messengers' "preview" functionality.

All of them work in a similar manner. **First**, they somehow get the URL to your website's page. This usually happens when a user tries to share a link to your page with their social media friends. Or when a search bot mindlessly crawls through the millions and millions of publicly available pages online. That's why they are called crawlers, by the way.

Second, the bots send the request to the server and receive the HTML, just like the browser does at the very beginning.

Third, from that HTML, they extract the information they came for and process it. Search engines extract stuff like text, links, meta-tags, etc. Based on that, they form the search index, and the page becomes "googleable". Social media previewers grab the meta-tags and create the nice preview we all have seen, with a large picture, title, and sometimes a short description.

And finally, **fourth**... Actually, there is no fourth sometimes. That's it. No JavaScript, just pure HTML. Because rendering the page properly with JavaScript means the robots need to spin up an actual browser, load JavaScript, and wait for it to finish generating the page. Which is quite costly from a resource and time perspective.



You can see it in action in the Study Project for this chapter (`src/chapter6-intro-to-ssr/simple-frontend`). Build and start it:

```
npm run build --workspace=chapter6-simple-frontend
npm run start --workspace=chapter6-simple-frontend
```

Navigate through the pages there. You should see that the title of the page changes with navigation. "Study project: Home" for the home page, "Study project: Settings" for settings, etc.

This title is injected with React using simple code like this:

```
useEffect(() => {
  updateTitle('Study project: Home');
}, []);
```

Where inside, it's just this:

```
export const updateTitle = (text: string) => {
  document.title = text;
};
```

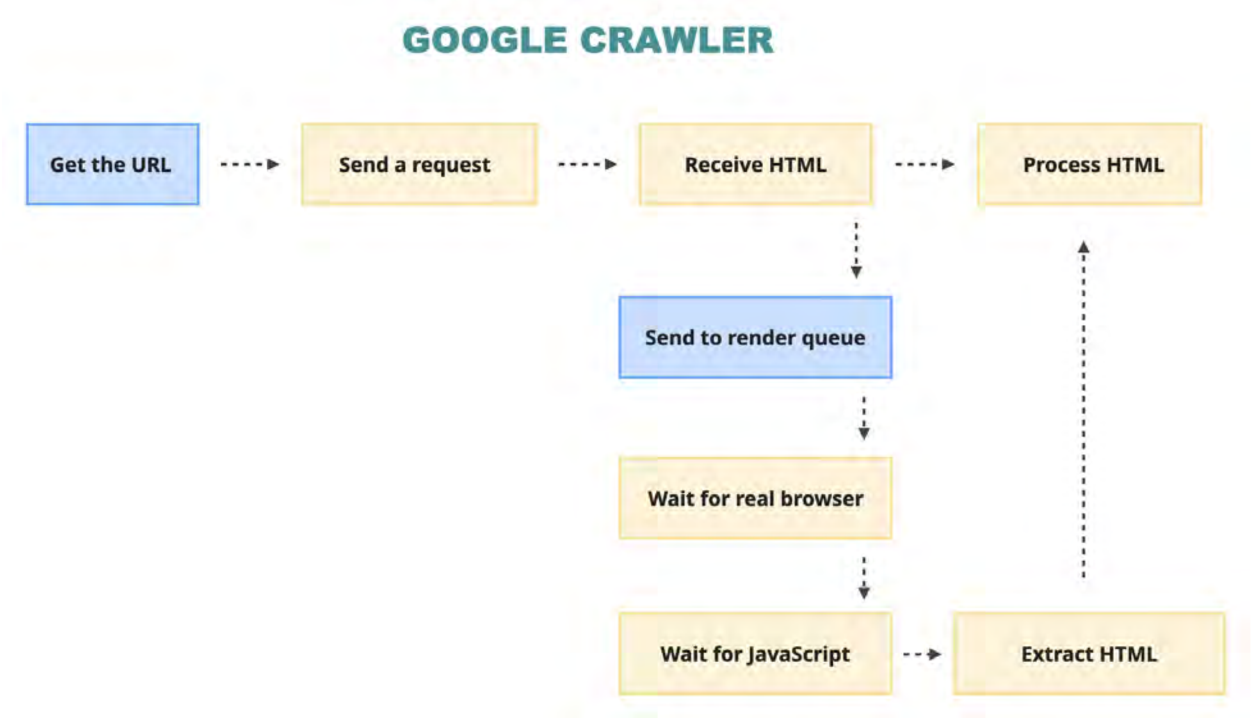
However, you also might see that it briefly "flashes" when initially loaded. That's because the default title is "Vite + React + TS". This is the title I have in `index.html`, and as a result, this is the title I receive from the server.

Now, expose the website to the outside world with ngrok[55] (or a similar tool if you have it):

```
ngrok http 3000
```

Try posting the URL it generates for you on the social media of your choice. In the generated preview, you'll see the old "Vite + React + TS" title. No JavaScript was loaded.

Although this is not entirely true for some of the bots. Most of the popular search engines wait for JavaScript. Google, for example, has a two-step process[56] where it parses the "pure" HTML and also puts the page into a "render" queue, where it actually spins up a headless browser, loads the website, waits for JavaScript rendering, and extracts everything it can again.



However, this process means that the indexing of a website that relies heavily on JavaScript might be slower and budgeted[57].

So if for your website it's:

- Mission-critical to be discoverable by as many search engines as possible, as fast as possible.
- Crucial to be shareable on social media platforms and to look good in the process.

Then it's very important for the server to return the "proper" HTML with all the critical information on the very first request. Typical examples of such websites are:

- Read-first websites, i.e., various forms of blogs, documentation, knowledge bases, forums, Q&A websites, news outlets, etc.
- Various forms of e-commerce websites.
- Landing pages.
- And so on - pretty much everything that you can search for on the World Wide Web.

That means the "classic" client-side rendered SPA, with its empty div as the first HTML response, is a bad idea here.

It doesn't mean, however, that we need to throw away React in anger. There are a few solutions we can try first.

Server Pre-rendering

For this one, we'd need to introduce a server into the equation. Right now, in the Study Project, the server implementation looks like this:

```
app.get('/*', async (c) => {
  const html = fs.readFileSync(path.join(dist, 'index.html')).toString();

  return c.html(html);
});
```

When the server receives any request, it reads the `index.html` file that the build step generated in advance, converts it into a string, and sends it back to whoever requested it. It's basically what all hosting platforms that support SPA will do for you. It's just not something we directly control or modify with platforms like this.

However, to fix the "no-JavaScript" problem, we need to modify the server now. Fortunately, not much. The fact that we're only working with a string simplifies matters a lot. Because nothing stops us from modifying that string before sending it back.

Let's find the existing title and replace it with "Study project", for example:

```
app.get('/*', async (c) => {
  const html = fs.readFileSync(path.join(dist, 'index.html')).toString();

  const modifiedHTML = html.replace('<title>Vite + React + TS</title>', `<title>Study
project</title>`);

  return c.html(html);
});
```

That's slightly better, but in real life, the title should change for every page: there's no point in keeping it static like this. Luckily, each server always knows exactly where the request is coming from. For the framework that I'm using (Hono[58]), it's a matter of asking for `c.req.path` to extract it.

After that, we can generate different titles based on that path:

```
app.get('/*', async (c) => {
  const html = fs.readFileSync(path.join(dist, 'index.html')).toString();

  const title = getTitleFromPath(pathname);

  const modifiedHTML = html.replace('<title>Vite + React + TS</title>', `<title>${title}
</title>`);

  return c.html(html);
});
```

Where in `getTitleFromPath` we can do something like this:

```
const getTitleFromPath = (pathname: string) => {
  let title = 'Study project';

  if (pathname.startsWith('/settings')) {
    title = 'Study project: Settings';
  } else if (pathname === '/login') {
    title = 'Study project: Login';
  }

  return title;
};
```

In real life, this likely needs to be co-located with the actual pages or even extracted from the page code itself. Otherwise, it will go out of sync with them almost immediately. But for the Study Project, this will do just fine.

One last thing to make it pretty: in the `index.html` file, we can replace the original title `<title>Vite + React + TS</title>` with something like `<title>{{title}}</title>` and turn it into a template.

```
<html lang="en">
  <head>
    <title>{{ title }}</title>
  </head>
  ...
</html>;

// on the server then do this instead:
const modifiedHTML = html.replace('{{ title }}', title);
```

In the future, we can convert it into any of the templating languages if there is a need.

And, of course, we're not limited to only the title tag - we can pre-render all the information in the `<head>` section like this. This gives us a relatively easy and cheap way to solve the "no-JavaScript" problem for social media preview functionality. They usually don't need more. Most of them rely on the Open Graph protocol[59], which is a bunch of `<meta>` tags with information.

We can even pre-render the entire page, not only meta-tags! But that one we'll cover in the separate SSR block below, there are many more things to learn there.

Additional Challenge.

1. In `src/chapter6-intro-to-ssr/simple-frontend/server` replace the contents of the `index` file with the content of `src/chapter6-intro-to-ssr/simple-frontend/server/pre-rendering-index.ts`
2. In `src/chapter6-intro-to-ssr/simple-frontend/index.html` replace the contents of the `<title>` tag with `{{title}}`
3. Refactor both the frontend and backend code to support the required meta-tags for social media sharing (see the list here)[60]
4. Build the project, expose it via `ngrok` again, and try to share each page (login, home, settings) on your favorite social media. The preview should work properly now and display the details of each page.
5. Bonus question: How would you refactor the project so that the meta-tag information is not duplicated between the Client and the Server?

The Cost of Server Pre-rendering

I mentioned above that meta-tag prerendering is relatively cheap. What exactly does this mean, though? How cheap, especially compared to the price and effort before introducing it?

Unfortunately, there is no good news compared to the completely static SPA. By adding a simple pre-rendering script, I introduced two problems other than the obvious increase in complexity that I now have to deal with.

Where to Deploy?

The very first problem is, where should I deploy the app now? Before the change, I could keep the hosting cost at zero for a very long time - hosting static resources is practically free these days. Now, I need to have a server. And those usually ain't cheap.

There are two most common solutions here.

We can use the **serverless functions** of the hosting provider that serve the static resources: Cloudflare Workers[61], Netlify Functions[62], Vercel Functions[63], Amazon Lambdas[64], etc. Most of the static resource hosting providers probably have them in some form or another.

The **advantage** here is that we still don't need to *think* about the server and its maintenance. Those Cloud Functions are like mini-servers that the provider deals with for us. Our job is to write code, and it magically Just Works. Everything else is their concern. For study projects, some niche projects, projects at the very beginning of their journey, and those that don't have a viral nature built in, cloud functions will be the optimal choice.

Cloud functions are usually very easy to configure and deploy, they are priced per usage, and the usage comes from actually hitting the endpoint. There is no chance of incurring an unexpected bill here by accidentally leaving the container running over the weekend.

The **downside** is the "price per usage" part. The more popular the website is, the greater the chances that the usage will exceed the limits. I read a few horror stories where a project became popular on HackerNews or TikTok, suddenly got a few million visitors instead of the regular hundred, and the owner woke up to a surprise $5000 bill. Setting spending limits, monitoring spending closely, and having a plan for what to do in this situation are crucial when it comes to serverless solutions.

If serverless functions are not your choice, you can simply keep it as a tiny node (or anything else) **server** and deploy it to any cloud platform, from AWS[65] to Azure[66] to Digital Ocean[67] to ...insert your favorite hosting provider....

This solution has its **advantages**. Everything is under your control. Migrating from one solution to another won't require code changes, unlike serverless functions, which are vendor-locking you a bit. Prices are usually much more predictable, much simpler, and much lower when the usage increases. Also, you can use whatever tech stack you want, whereas serverless functions are typically very limited.

The **disadvantages** are exactly the same as the advantages. Everything is up to you now. You need to monitor CPU/memory usage. Worry about observability. Worry about scaling. Memory leaks will keep you up at night.

And you'll have to worry about geographic regions. Which leads me to the second problem that arises with introducing *any* kind of server to a previously pure SPA app.

Performance Impact of Having a Server

Remember the very first chapter and the impact of **latency** and **CDN** on initial load performance? By introducing even a rudimentary server that only pre-renders meta-tags, I'm introducing a mandatory, unavoidable round-trip to the server for every initial load request, regardless of whether the user is new or repeating.

I just made the initial load performance of an SPA app, which was never great to begin with, slightly worse. And how much worse will depend a lot on where exactly the server is deployed.

If it's deployed as one of the Serverless Functions, then there is a chance that it's not that bad. Some of the providers can run those functions "on Edge". I.e., those functions are distributed to different servers that are closer to the end user. Pretty much the same as CDN for static resources. In this case, the latency will be minimal, and the performance degradation will be minimal.

If, however, I went with the self-managed server, I wouldn't have the advantages of a distributed network. I'd have to deploy it to one particular region. So, users on the opposite side of the planet from this region have a chance to *really* feel the impact of the performance degradation.

If this performance impact is critical, you'd have to deal with it somehow. Prepare yourself for complicated caching strategies, deployments to different regions, etc. Basically, it's not a simple no-server frontend app anymore. It's a full-stack or even backend-first app now.

Next.js on Vercel/Netlify

A question that immediately might pop into mind: "I'm just writing my frontend with Next.js and deploying it to Vercel/Netlify. Do I really need to know any of this?"

The answer here is, "Unfortunately, yes, you can't escape it". Because that's exactly what those Next.js-first hosting providers do by default: they convert your app into JavaScript files and a bunch of small Serverless Functions. It's just happening without your control or even knowing about it.

So, unless you've explicitly set up your Next.js project to export as "static", everything in "The cost of server pre-rendering" applies.

Additional Challenge.

1. If you have a Next.js app that is "natively" deployed (i.e., 1-click deployment) to some serverless platform like Vercel/Netlify, try to find how many functions

were created for it.

2. Are those "edge" functions or normal functions? Can you find out how usage is calculated for them? How many visitors can your website handle before you're over the limit?
3. If you have a combination of "normal" and "edge" functions, can you map out which function is responsible for what, and what effect they have on your deployed project?

Pre-rendering the Entire Page on the Server (SSR)

Let's talk about pre-rendering some more. In the section above, we pre-rendered only meta-tags because it was easy to replace an existing string with another string. But what stops us from messing around beyond the `<head>` tag? Let's take a look at the contents of the `<body>` tag of our HTML page that is sent by the server:

```html
<body>
  <div id="root"></div>
  <script type="module" src="./main.tsx"></script>
</body>
```

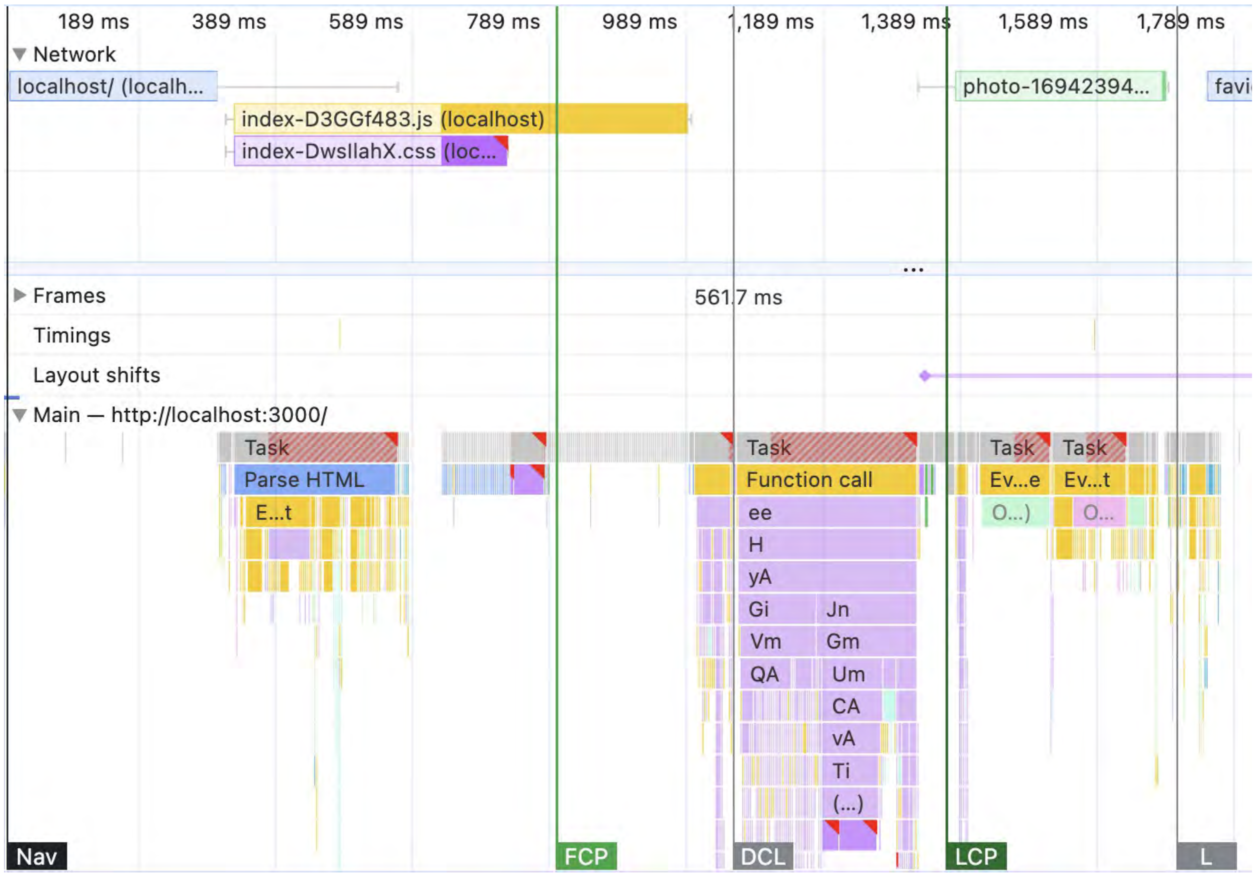
Remember how Client-Side rendering works? When the scripts are downloaded and processed, React takes the "root" element and appends the generated DOM elements to it. So what will happen if, instead of the empty div, I return a div with some content inside? Let's make it a big red block:

```html
<div id="root">
  <div style="background:red;width:100px;height:100px;">Big Red Block</div>
</div>
```

Try adding it to the chapter's `index.html`, build the project, start it, and don't forget to disable the cache and slow down the CPU and Network for better visibility.

When you refresh the page, you should see the momentary flash of the Big Red Block, which is replaced by the normal dashboard page. The first great news is that the red block didn't stick around - obviously, React clears the "root" div before inserting anything inside. Or overrides the existing children, doesn't matter for this chapter.

The second great news comes from staring at the performance graph. Record it now. The result should be something like this:



Pay special attention to the order of things and timing here.

In the beginning, it looks exactly like the graphs we've seen before. First, waiting for HTML from the server, which results in a blue HTML parsing block in the "main" section. That one triggered the download of CSS and JavaScript (yellow and purple blocks in "Network") at some point.

But after the CSS was downloaded, different things started happening. First, we see a somewhat longer block of purple "Layout" (same level as the blue HTML block). That didn't happen before! Almost immediately after it was done, the FCP (First Contentful Paint) was triggered. But the JavaScript bar at the top is still loading! After that, things continue as usual - JavaScript finishes loading, it's processed, painted, and then LCP (Large Contentful Paint) is triggered.

If you hover over the very top section, where the screenshots of the frames appear, you'll see that the gap between FCP and LCP is exactly the period when our Big Red Block was present on the page. The gap between the FCP and LCP is around **500 ms**, with FCP around **800 ms** and LCP around **1.3 s**.

Looks like this 500 ms is pretty much the cost of Client-Side rendering for the initial load. This is huge! If I somehow manage to shave off those 500 ms from LCP, that's a 40% improvement! I might get a promotion for this.

Fortunately, everything is possible. React gives us a few methods[68] that can pre-render the entire app, and which we can theoretically use here. For example, there is "renderToString[69]". It can render our app into a string, according to the documentation:

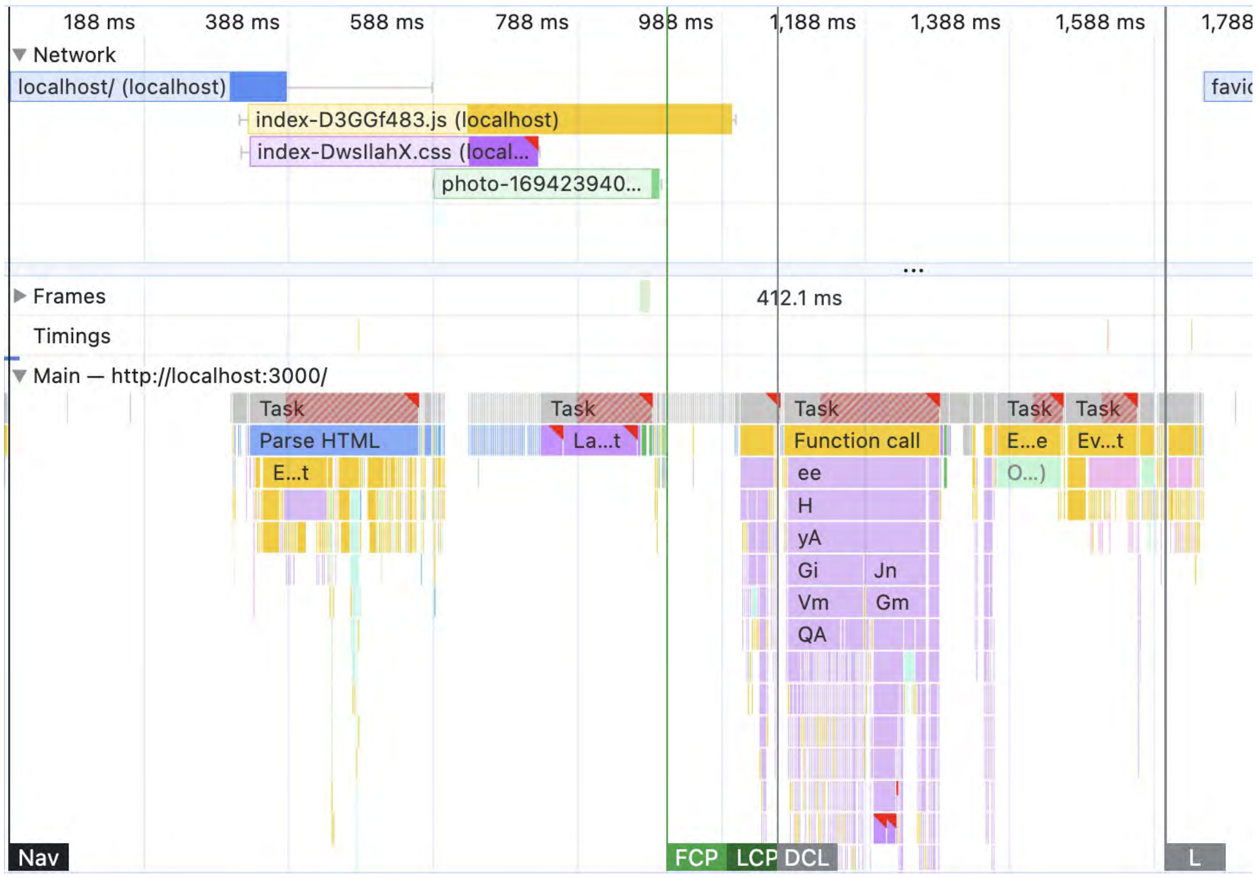
```
const App = () => <div>React app</div>;

// somewhere on the server
const html = renderToString(<App />); // the output will be <div>React app</div>
```

Since we're already dealing with strings on the server, this seems perfect. All that I'd need to do is replace the empty "root" div with the output of this function. Exactly the same as we did for meta-tags. Let's try?

Go to the `src/chapter6-intro-to-ssr/simple-frontend/server/index.ts` and clean it up from any modifications we made above. Find the commented-out code:

```
// return c.html(preRenderApp(html));
```

And uncomment it. Re-record the performance again. The end result should be something like this:



The difference is immediately visible: FCP and LCP happened at the same time. Before the main React-produced JavaScript was triggered, and even before JavaScript finished loading. That means that content pre-rendering is working! 🎉 Happy days ☀️☺️. Hover over the screenshots at the very top to verify that it is indeed the beautiful dashboard that showed up then, not some random fluctuation.

There is, however, a tiny abnormality - the FCP is triggered later than I promised. I was hoping to see it at 800 ms, but it is actually around 900 ms. The recurring lesson in everything performance: never promise exact numbers in advance 😅. But where did I lose 100 ms?

First of all, look at the very top left corner, the "Network" section, where we have the initial request to the server. Notice the solid blue line there appearing? This is our HTML content downloading. We're sending many more elements now, not just a simple empty `<div>`. Hover over that block to see the exact numbers - around a third of the missing 100 ms is spent on downloading the content.

Also, pay attention to the purple "Layout" block after the "Parse HTML" task. It looks much longer now, doesn't it? Hover over it again for the exact numbers - and here's your missing two-thirds of the 100 ms. The browser needed not only to download some extra HTML but also to calculate the positions of many more elements before painting them.

Hence the missing time. But still worth it, isn't it? I shredded 400 ms from the LCP timing and improved initial load performance by 30%! And here's another cool part: disable JavaScript now and refresh the page. The dashboard is still there! And even links work, although they cause a full page reload.

This part is what makes SSR worthwhile. Now, every search engine and every other robot that you want to give access to your page will see everything without loading any JavaScript. Performance improvement is a nice bonus here. And an unstable one at that.

SSR can make LCP Worse

Unstable, because there are no silver bullets in performance. If someone tells you SSR will a 100% increase the initial load of your SPA app, they are mistaken. Now that you know how network conditions, Client-Side and Server-Side rendering work, can you think of a scenario when SSR worsens LCP?

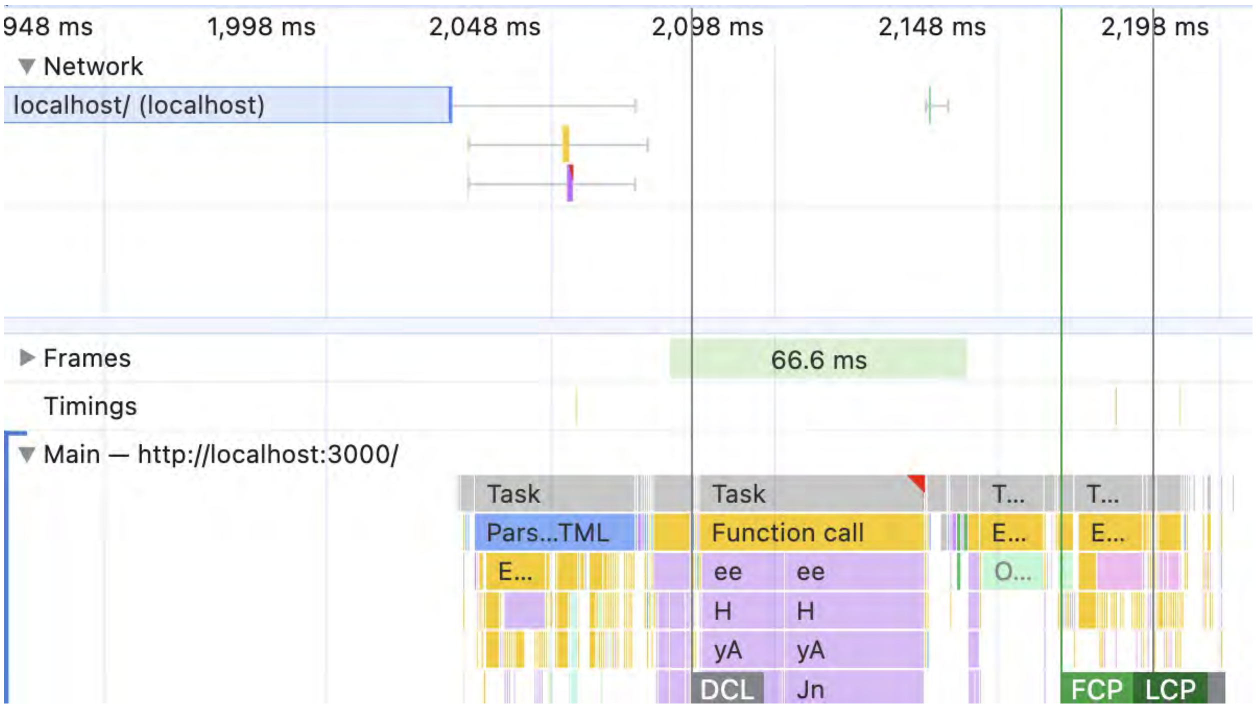
Here's how.

Disable the CPU throttling, let your machine be fast again. Set Networking to the slowest possible simulation. For me, default Chrome 3G does the trick, but you might need to go slower - it will depend on how fast your machine is. *Uncheck* the "disabled cache" checkbox. I want those CSS and JS files to be served from the browser memory.

Now, measure the LCP with and without pre-rendering.

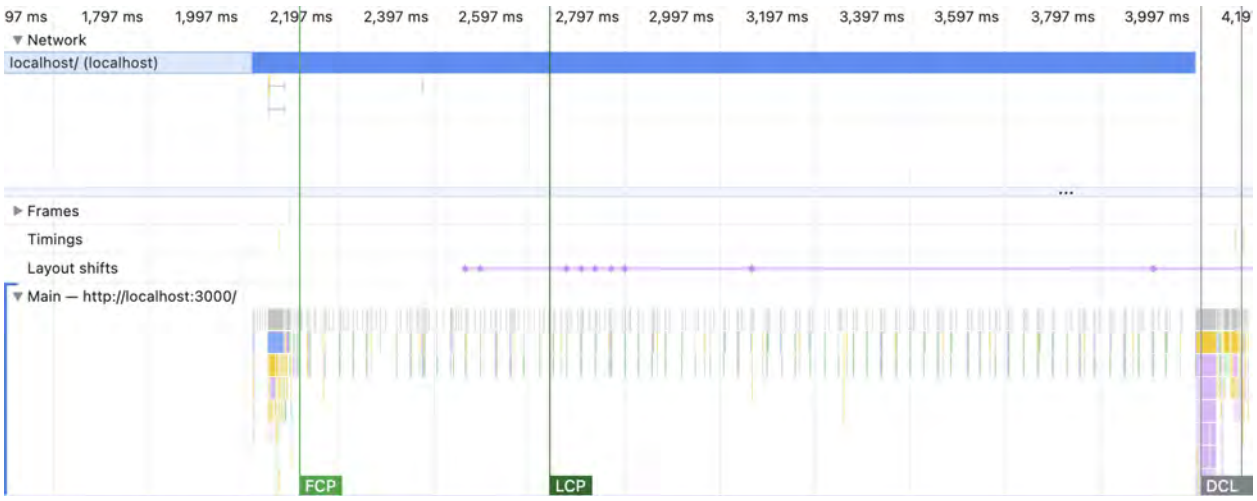
For me, the results are like this. *Without* pre-rendering, the "SPA" mode, the LCP is around 2.13 seconds. *With* pre-rendering, in the "SSR" mode, it's around 2.62 seconds. Almost 500 ms longer!

The performance charts are a fascinating read for this situation. The "SPA" mode looks like this:



At first, there is a loooong block (2 seconds) of waiting for the server's response in the Network section. That's the latency of the slow network connection. Then, almost instantaneous access to JavaScript and CSS resources: they come from the browser cache, no network. Plus, almost instantaneous download of the HTML content - it's just an empty div. Then, the regular and quite fast, since the CPU is not slowed down, JavaScript execution. That's our React generating the page. And finally, the page is visible.

Now, the same Network/CPU conditions with the SSR mode enabled:



Same initial waiting time - latency never went away. Then, HTML starts downloading. But since now we have a lot of it, the download takes a looooooong time - the bandwidth is very reduced.

Then the most fascinating part: while the content is downloading, I see spikes of activity in the Main section. Enlarge and hover over them - it will be mostly Layout tasks. The browser already has CSS and JavaScript downloaded (from the cache), so it has all the necessary information to paint the layout as soon as it gets small pieces of it. And it does.

You should be able to actually ***see*** the gradual build-up of the interface on the page: first, the sidebar shows up, then the top nav, then the top charts, then the table. All of this is in the order of the HTML that is coming through slowly. If this is not cool, I don't know what is.

While this example feels like a weird edge case, it's actually not. The combination of a slow network + huge latency + fast laptop happens quite often for business travelers, for example. Or wildlife photographers. Or travel bloggers. Or engineers who are sent to remote locations. So if your app is targeting that specific niche primarily, and your app is already SPA, trying to introduce SSR to it might make it *worse*.

Or it might not, of course. It will depend on the size of the HTML that is downloaded, how fast the devices actually are, and how much JavaScript the app needs to render. Essentially, it all comes down to two things: knowing your customers and measuring, measuring, measuring everything.

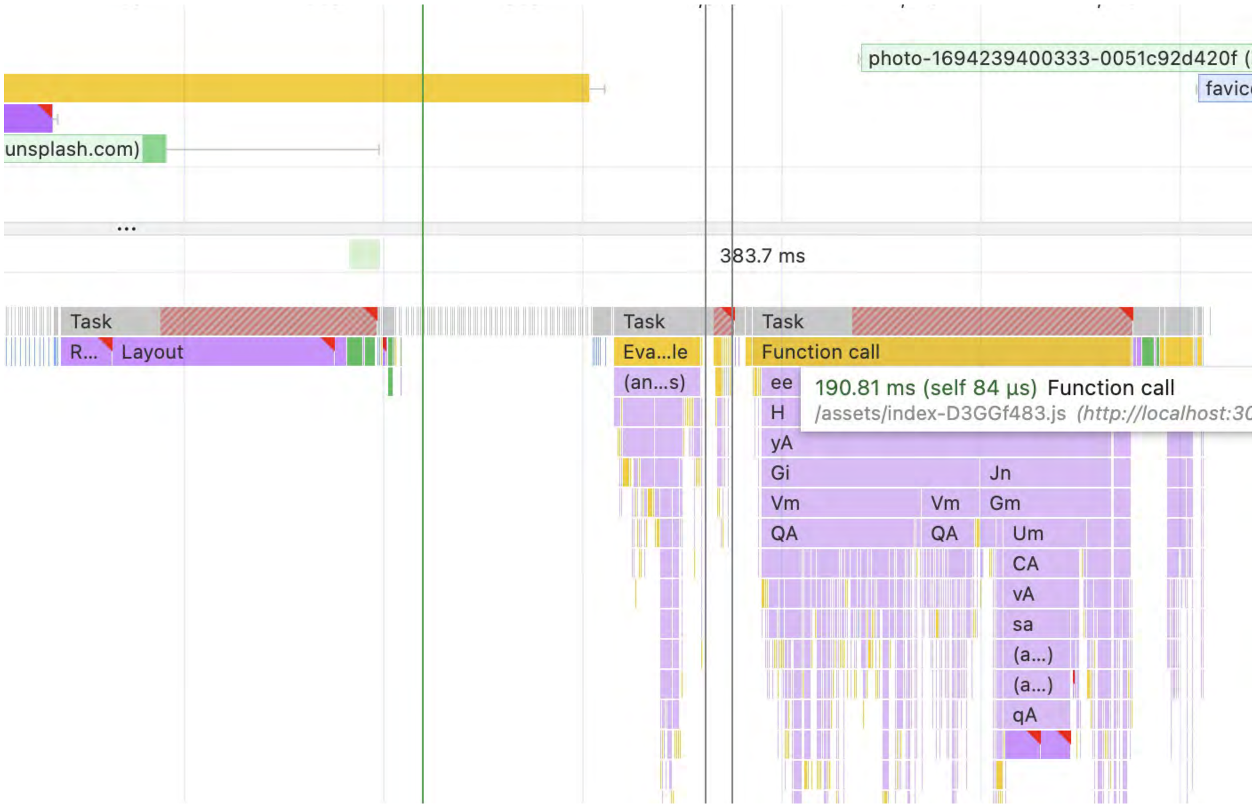
SSR and Hydration

In all this excitement about showing the content sooner, we forgot to investigate what happens after the content is loaded.

Remember the Big Red Block behavior? After React has loaded and generated its own elements, it completely replaces the content of the "root" div and everything inside, including the Big Red Block. But what happens when, instead of the weird red block, I send the proper HTML of the future page?

Actually nothing. I haven't told React in any way that this content is important, so it will behave in exactly the same way: clear the entire content of the "root" div and replace it with its own. It just happens to be exactly the same content from the HTML perspective, so we don't see the difference with the naked eye.

But we *can* see it in the performance profile. Slow down the CPU and Network to make the behavior slightly more visible and re-record the performance for the SSR example. Pay attention to what is happening after the CSS and JavaScript received:



On the top left, I have the network section, where the resources are finished downloading. Almost immediately after the CSS is received, I see a large purple "Layout" section below - that's when our SSR'd content shows up. After the JavaScript yellow block on the top left finishes loading, React kicks in. The somewhat longer task (190 ms) is when React builds the UI. At the very bottom right, I again see a small-ish Layout block.

This is a typical picture of the Client-Side Rendering that we've seen many times already. This is when React clears the "root" div and injects whatever it generates instead. And also, it's completely unnecessary. React already has all the DOM elements present, it could've just reused them instead. Surely, it should be faster.

This is when what is known as **"hydration"** comes in. "Hydration" does exactly what I wished for above - it shows React that there is already HTML on the page that matches exactly the HTML it will generate. So, React can just reuse the existing DOM nodes, add event listeners to them, prepare whatever it needs internally for future functionality, and call it a day. No unnecessary mounting components from scratch!

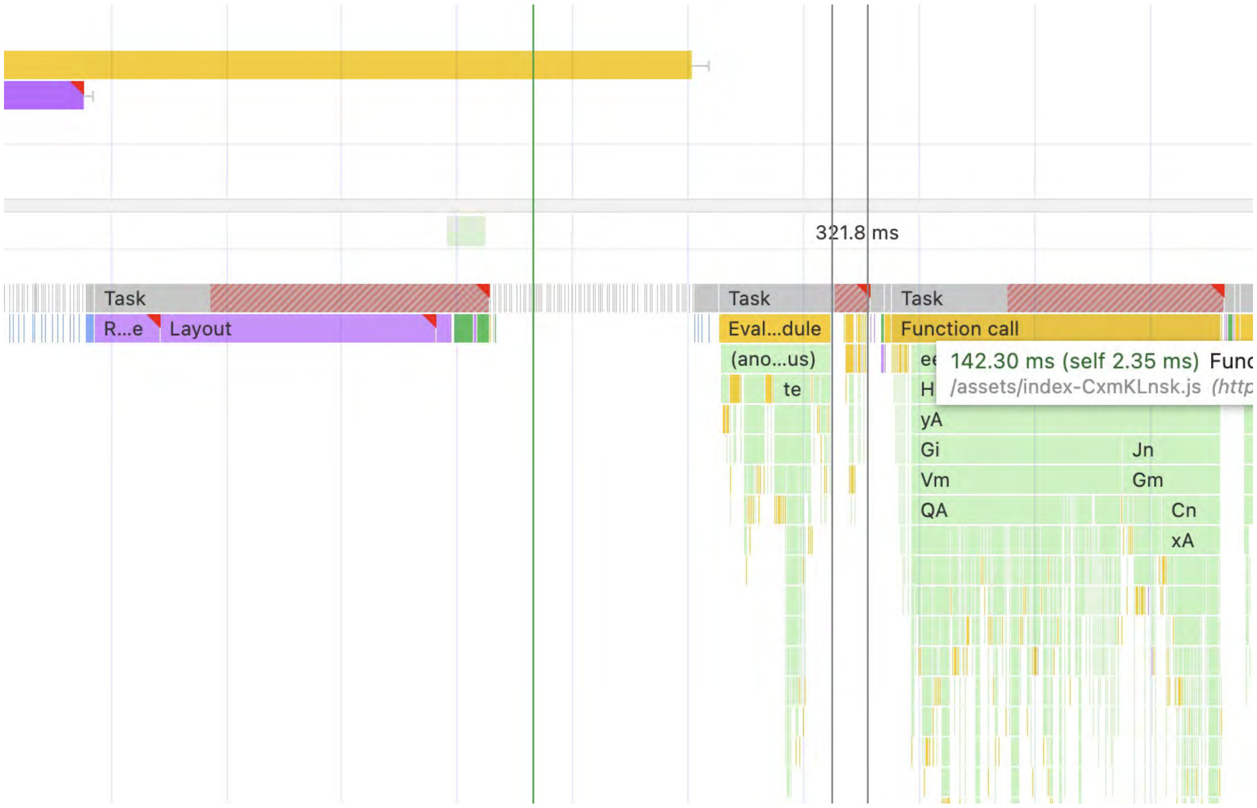
Hydration in React is actually very simple to implement, for once: it's just one function[70] call. All we need to do is replace the `createRoot` entry point with this:

```
hydrateRoot(
  document.getElementById('root')!,
  <StrictMode>
    <App />
  </StrictMode>,
);
```

You can find this code in `src/chapter6-intro-to-ssr/simple-frontend/main.tsx` - comment out the `createRoot` part and uncomment the hydration part. Then rebuild and restart the project:

```
npm run build --workspace=chapter6-simple-frontend
npm run start --workspace=chapter6-simple-frontend
```

Measure the performance again:



There is no more purple stuff in the React-related JavaScript execution. And it's slightly faster now - 142 ms instead of 180. This might not seem like much now, especially considering that LCP was triggered before. But it's not always going to stay like this.

Try, for example, unchecking the "disable network cache" and removing Network throttling while keeping the CPU down. Emulating repeated visitors with fast internet but slow devices. For me, without hydration, it separates FCP from LCP and pushes the LCP beyond the JavaScript task at the very end. LCP, in this case, is around **550 ms**. With hydration enabled, LCP moves closer to FCP and hovers around **280 ms**, right at the beginning of the JavaScript task.

There is also the matter of blocking the main thread and reducing it as much as possible, which Hydration helps with. But we'll cover it in more detail in the next chapters.

Finally, hydration can be not only about JavaScript listeners. It also allows fetching and injecting some initial data into the app, so we can avoid loading spinners or flashes of content. We'll cover this in more detail in a dedicated data-fetching chapter.

Should I Implement SSR Like This?

Now that it seems obvious that SSR might be very, very useful for certain cases, and implementing it seems kinda trivial, the question might arise: can I just use the code from the Study Project and implement my own SSR?

The answer will be very rare for this book: absolutely not! The solution is fine for study purposes, to explore how pre-rendered content behaves from different perspectives while turning on and off one thing.

But it's actually not trivial at all. I hid half of the things I had to do to make it work. And half of the things have not been implemented yet. It's a very basic and almost deprecated version of the backend that doesn't support the latest React features.

For example, there is no SSR for the dev server here. So, there is no way to debug the SSR other than constantly rebuilding the project. That's half the reason you had to rebuild it all the time to apply changes, by the way. (The other half is because performance should always be measured on the production build, so I don't feel particularly guilty about it).

If you want nice things like hot reload, then you have to implement it by yourself. There is a whole large set of instructions[71] on how to integrate SSR with Vite properly. And that's Vite, for Webpack it will be very different and likely not very well documented. For something more exotic, I don't even know where to start.

Also, the pretty string `const html = renderToString(<App />);` that I showed from the React docs is a myth and is never actually going to work by default. The problem here is this part - `<App />`. This is JSX, it's how we write React code most of the time, so it seems so normal now. But the only reason why it works is because your build system has a transformation step, which is likely (or maybe not, it always depends) powered by Babel. "Pure" Node or any other server framework won't support it.

Look into `src/chapter6-intro-to-ssr/simple-frontend/server/pre-render.ts` to see how it's actually implemented.

First, I extracted the transformed `App` code from Vite itself:

```
const { default: App } = await vite.ssrLoadModule(p.join(process.cwd(), 'App.tsx'));
```

If you're using Webpack, you'll likely need to manually configure and register Babel plugins for that. So, even the very first step would mean you need to have an understanding of what is happening here and how to implement it for your app.

The second step, the actual `renderToString`:

```
const reactHtml = renderToString(React.createElement(App, { ssrPath: path }));
```

Still doesn't look like the docs - JSX support for the backend file is not the same as extracting it from Vite. I'm using native React syntax here.

Also, if you read the docs, you'll notice that `renderToString` doesn't support streaming and waiting for data[72].

So, to actually implement proper SSR, you'd need to understand whether you need those new features in your app or not. And if yes, how to implement them on the backend. There is *some* documentation[73] for the recommended method and also a few[74] discussion threads[75] on GitHub on the topic, so at least it's a start.

But it's *a lot* of work. The things mentioned are just the beginning, and before you know it, you're three months behind on the project and basically implementing your own Next.js. Why do you think there aren't that many competitors to it?

So, unless there is a very valid business reason and a lot of support in terms of time, resources, and expertise for this, it might be easier just to use an already existing SSR framework. Especially considering that the backend part here is only one piece of the puzzle. There is also a lot of complexity involved on the front side.

SSR and Frontend

Depending on the size of the app and how optimized it is for SSR, it might be even more complicated than the backend part. Yep, you heard me right, that's another thing I hid from you while implementing the SSR above: we need to make changes to our frontend code as well.

Browser API and SSR

Remember how I extracted the HTML that is sent to the browser? I generated a string with React's `renderToString` and then injected that string into another string. There was no browser in the vicinity of this process, and there never will be.

What do you think will happen to all the calls to the browser variables that we're so used to on the frontend? All those `window.location` and `window.history` and `document.getElementById`? Nothing good. `window`, `document`, etc., will turn into `undefined`. There is no browser that can inject them into the global scope.

So, the second React tries to call a function (i.e., render a component) that tries to access them directly, it will fail with the `window is not defined` error. The entire app will just explode. Not just explode. The server part will explode, which is even worse - there won't be any chance for the frontend part to catch the error and show some pretty screen with "we're working on it, here's a cookie". Error handling would need to be handled on the server, and you'd have to have a special "server" error screen.

Additional Challenge.

- Try to add a simple `console.info(window.location);` to any part of the frontend code, for example, here: `src/chapter6-intro-to-ssr/simple-frontend/App.tsx`.
- Rebuild the app and restart it.
- You should see the `Internal Server Error` string on the screen.
- Can you come up with a way to fix it?

A typical way to fix it would be to check whether the `window` and all other global variables are declared before trying to access them:

```
if (typeof window !== 'undefined') {
  // do something when the global window API is available
}
```

If you look at the code in `frontend/utils/use-client-router.tsx`, this is exactly what I had to do. And I would have to do this any time I need to access `window`, `document`, or anything else at runtime.

useEffect and SSR

And speaking of the `use-client-router` file. If you look closely, you'll see that I didn't have to do that check for `typeof window` inside `useEffect`:

```
useEffect(() => {
  const handlePopState = () => {
    setPath(window.location.pathname);
  };
  window.addEventListener('popstate', handlePopState);
  return () => window.removeEventListener('popstate', handlePopState);
}, []);
```

This is because when running on the server (i.e., via `renderToString` and friends), React doesn't trigger `useEffect`. And `useLayoutEffect,` for that matter. Those hooks will run only on the client after hydration happens. Take a look at this short explanation[76] and some lengthy discussion[77] on the topic from the core React team members if you want more details on the reasoning for this behavior.

So it's definitely something to keep in mind in case you're expecting some UI changes as a result of `useEffect` - they will cause a "flash" of content when the JavaScript is loaded.

Conditional SSR Rendering is a Big No

Some parts of your code might have so many dependencies on the browser API that you might think it's easier just to skip rendering that part entirely while in SSR mode. The natural temptation would be to do something like this:

```
const Component = () => {
  // don't render anything while in SSR mode
  if (typeof window === "undefined") return null;

  // render stuff when the Client mode kicks in
  return ...
}
```

Nope. That's not going to work. Or, more precisely, it will. But it will confuse React big time. React expects that the HTML produced by the "server" code is ***exactly the same*** as the HTML produced by the client code.

It will confuse it so much that it will just fall back to the Client-Side Rendering pattern. I.e., it will erase the entire content of the "root" div and replace it with the freshly generated elements. It will behave as if the hydration never happened, with all the downsides that come with that.

Additional Challenge.

1. Somewhere in `./pages/dashboard.tsx` (or anywhere else you like), create a `ClientOnlyButton` component with the code:

```
const ClientOnlyButton = () => {
  if (typeof window === 'undefined') return null;
  return <button>Button</button>;
};
```

1. Render it somewhere on the page.
2. Rebuild and restart the project as usual.
3. Record the performance profile. It should show the picture we saw when hydration was not implemented yet, with Layout blocks inside the React JavaScript task.

If you're lucky! Sometimes it can just introduce really weird layouting bugs[78] instead, so the website will look completely broken.

The correct way to do this is to rely on React's lifecycle to "hide" the non-SSR compatible blocks. For this, we need to introduce a state and track whether a component is mounted or not:

```
const Component = () => {
  // initially, it's not mounted
  const [isMounted, setIsMounted] = useState(false);
};
```

Then flip this state to `true` when the component has been mounted, i.e., inside a `useEffect`:

```
const Component = () => {
  // initially, it's not mounted
  const [isMounted, setIsMounted] = useState(false);

  useEffect(() => {
    setIsMounted(true);
  }, []);
};
```

Remember: `useEffect` doesn't run on the server, so the state will turn `true` only when the client-side version of the website is fully initialized by React.

And finally, render what we wanted to render that is not SSR compatible:

```
const Component = () => {
  // initially, it's not mounted
  const [isMounted, setIsMounted] = useState(false);

  useEffect(() => {
    setIsMounted(true);
  }, []);

  // don't render anything while in SSR mode
  if (!isMounted) return null;

  // render stuff when the Client mode kicks in
  return ...
}
```

Additional Challenge.

- Rewrite the `ClientOnlyButton` from the previous exercise to work correctly with SSR.
- Rebuild and restart the project as usual.
- Record the performance profile. It should revert to the SSRed look.

Third-party Libraries

Not all of your external dependencies will be supportive of SSR. It's always a gamble with libraries. For some of them, you'll be able to opt out of SSR using the solution above. Some of them will be rejected by the bundler, so you'll have to import them dynamically after the client-side JavaScript is loaded. Some of them you'll need to remove from the project and replace with a more SSR-friendly library.

This is going to be especially painful if the non-SSR-compatible library is something fundamental to the entire project. Like a state-management solution or a CSS-in-JS solution.

For example, try to use Material UI icons somewhere in the Study project:

```
// anywhere, for example, in src/chapter6-intro-to-ssr/simple-frontend/App.tsx
import { Star } from '@mui/icons-material';

function App() {
  // the rest of the code is the same
  return (
    <>
      ...
      <Star />
    </>
  );
}
```

Rebuild it and restart it - you should see that the SSR collapsed with:

```
[vite] (ssr) Error when evaluating SSR module ...: deepmerge is not a function
```

Have fun figuring out how to fix it 😳

Static Site Generation (SSG)

Okay, let's assume that we absolutely have to have "proper" server-rendered pages and we're ready to deal with the consequences on the frontend for this. For example, we're implementing a fancy "promo" website. Those obviously need to be indexed by all possible search engines as fast as possible and should be shareable via everything that can share a link. That's the whole point of a website like this.

Let's also assume that all the info on the website is "static", i.e., there is no user-generated content, no permissions to take into account, no complicated data generation per request. The website is just a few pages that introduce the product, some standard pages like "Terms and Conditions", and a blog that is updated once a week.

This situation is a rare use case where we can have our cake and eat it too. We already know that pre-rendering the website on the server is relatively easy. It's simply a matter of calling `React.renderToString` on our app (more or less).

So, the big question here is: what stops us from running `React.renderToString` during build time, right after we run `npm run build --workspace=chapter6-simple-frontend`? In theory, we're pre-rendering and sending a proper HTML page to the browser anyway. And the pre-rendered content is always the same. We can probably just do it in advance, save it as a bunch of actual `HTML` files, like in the good old days, and save ourselves the pain of having a "proper" server. Right?

The answer: there is absolutely nothing that stops us from doing that. Try running this:

```
npm run build:ssg --workspace=chapter6-simple-frontend
```

It will first build our website the usual way with Vite, and then run a very primitive script (`src/chapter6-intro-to-ssr/simple-frontend/server/generate-static-pages.ts`) that replaces the empty `<div id="root"></div>` with the content generated by `renderToString`. Exactly what the server does. Only now, we no longer need the server.

Take a look at the built files in the `dist` folder. You'll see two additional files: `login.html` and `settings.html`. Open any of the HTML files - you'll see that `<div id="root">` is filled with content.

This is our "static" website, which we can start with absolutely any web server:

```
npx serve src/chapter6-intro-to-ssr/simple-frontend/dist
```

Or upload it pretty much anywhere, same as any Client-Side Rendered app. Only this time it won't have CSR downsides, all search engines will be able to index it properly right away, and social media shares will work beautifully.

Static websites are so good that they even have their own three-letter abbreviation: **SSG (Static Site Generation)**. And of course, there are plenty of frameworks that generate them for you, no need for manual labour: Next.js supports SSG[79], Gatsby[80] is still pretty popular, lots of people love Docusaurus[81], Astro[82] promises the best performance, and probably many more.

What's Next?

The idea of generating parts of a React application during server time or even during build time is pretty powerful. We'll be returning to those concepts in the next chapters from time to time. Starting from the next chapter right away, where we'll take a look at how the size of JavaScript affects performance with and without SSR.

7. Bundle Size and What to Do About It

Until this moment, we were just writing our React code without much concern for the size of the JavaScript we're producing. But if you read any performance-related discussions on the Internet, the issue of bundle size and how to reduce it dominates the conversation. And even without participating in those discussions, having a few megabytes of JavaScript on the page just *feels wrong*.

But why exactly is this the case? How much of an impact does it have? And most importantly, if I decide to reduce the bundle size, how do I do that, and what improvements can I expect?

Let's investigate!

Initial Setup

First of all, let's set up the *Baseline Project*. We're going to use it to compare the impact. We can use the project from the previous chapter for this. First of all, build it with:

```
npm run build --workspace=chapter6-simple-frontend
```

Notice the size of the JavaScript it produces:

```
390.98 kB │ gzip: 141.43 kB
```

Run it like this:

```
npm run start --workspace=chapter6-simple-frontend -- port=1234
```

So that we can access it via `http://localhost:1234` and run it with this chapter's Study

Project in parallel.

Now to this chapter's Study Project. Build it:

```
npm run build --workspace=chapter7-simple-frontend
```

Write down the JavaScript size:

```
5,321.89 kB │ gzip: 1,146.59 kB
```

Fall to the floor in disbelief. More than **5 megabytes**! Good Lord 🙀 What exactly did you implement there, you might ask, and did it take more than a year? That surely took some effort and talent! 🤪

The answer: just added one form and another simple page with not that much functionality and less than an hour. It's actually incredibly easy to make a small mistake here and there and explode your JavaScript beyond reason. You'll see how and why while reading the chapter. For now, let's start the project and peek into the changes:

```
npm run start --workspace=chapter7-simple-frontend
```

First, open the Settings page and switch to the "Personal Profile" tab. There is a nice form there now.

Second, click on "Inbox" in the navigation - it's another nice page with a list of messages. Clicking on any of the messages will open a drawer with an editor. It's even semi-functional, some of the buttons will work 😉. If you hover over messages in the list, a row of buttons appears. Clicking on "Delete" or "Archive" opens the respective modal dialogs.

That's it. The rest is exactly the same as before! Including the "Home" page, which is identical to the Chapter 4 Study Project. This makes it ideal for measuring the impact of these five megabytes of JavaScript.

Bundle Size and Network

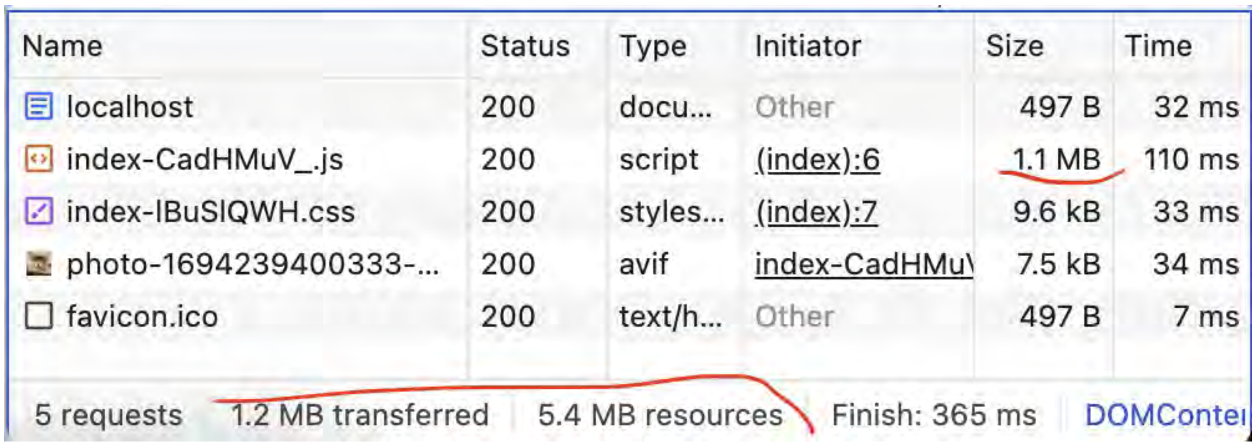
The most intuitive and most noticeable impact of those 5 MB of JavaScript will be, of course, the download time. Open both the baseline project and Chapter 5 project together, open the Performance panel, check the "Disable network cache" to imitate the first-time visitor, set the Network throttling to some reasonable speed, and refresh the page.

Even with "Fast 4G", the difference will be drastic: 0.88 s vs 6 s LCP for me. On Chrome's default 3G setting, you'll probably fall asleep before the website completely loads. Seriously, I gave up after 80 seconds of loading. If a website seriously intends to serve users 5 MB of uncompressed render-blocking JavaScript, it better pay them for every visit.

Bundle Size and Compression

Speaking of uncompressed. Until now, we've been measuring the actual size of the files produced by the production build. But in real life, when sending those files to the user, we often would first compress them with something like gzip[83] or brotli[84].

In the `src/chapter7-bundle-size/simple-frontend/server/index.ts` file, find the `app.use(compress());` line and uncomment it. This will enable `gzip` compression for all of the assets in the app. Open the network panel now and take a peek at the size of the files that are served to the browser:



Name	Status	Type	Initiator	Size	Time
localhost	200	docu...	Other	497 B	32 ms
index-CadHMuV_.js	200	script	(index):6	1.1 MB	110 ms
index-lBuSIQWH.css	200	styles...	(index):7	9.6 kB	33 ms
photo-1694239400333-...	200	avif	index-CadHMu\	7.5 kB	34 ms
favicon.ico	200	text/h...	Other	497 B	7 ms

5 requests | 1.2 MB transferred | 5.4 MB resources | Finish: 365 ms | DOMConte

The JavaScript file is now 1.1 MB; the rest of them have also shrunk. At the bottom, you'll see that we "only" have 1.2 MB transferred. The improvements to LCP are huge: on fast 4G, it dropped from 6 seconds to just 1.73, which is actually in the "good" category. On 3G, it's still sad, though - almost 30 seconds for me. So still needs improvement.

Some (if not most) of the hosting providers have compression enabled by default, by the way. Especially if you use a CDN. So, chances are, your website already has it enabled. If it doesn't, for some reason, enable it immediately. As you can see, it can drastically boost your performance numbers.

Compression is also very important to take into account for any efforts to reduce bundle size. Reducing JavaScript size by 200 KB, while it seems impressive, could result in just a 40 KB decrease in the transferred size. Which will have a much lesser impact (if any) on the initial load than one might hope for when they hear the "200 KB" number.

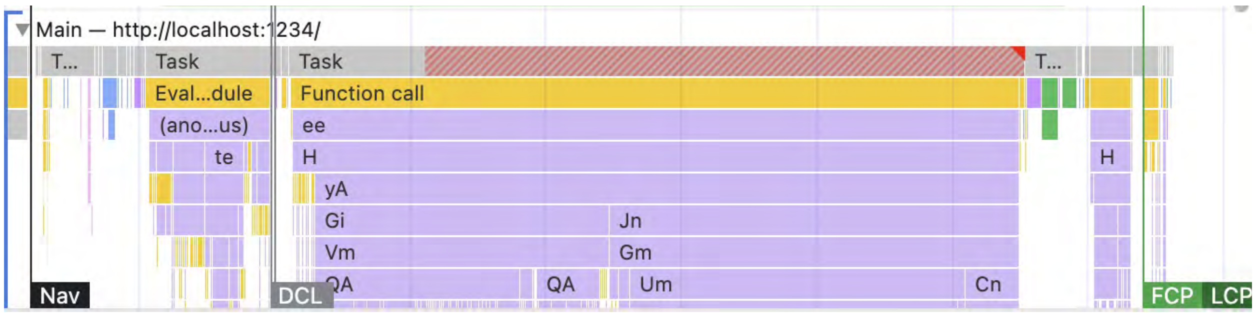
As a downside, this makes the cost of adding yet another small library to the bundle almost negligible. After all, if it solves a Huge Problem right now, and it's only 30 KB of uncompressed JavaScript, where's the harm? It probably won't even move the LCP metric. Left unchecked, and six months later, you wake up to 5 MB of uncompressed JavaScript and the need to push for yet another "let's reduce our bundle sizes" initiative for the entire organization.

Fortunately, there are two more arguments in favor of keeping the JavaScript size small that can help with that. And sometimes, they can be even more important.

Bundle Size and JavaScript Evaluation Time

After the browser downloads the JavaScript, it needs to evaluate and execute it. My gut feeling tells me that the more JavaScript it has, the harder and slower this process is going to be. Let's check?

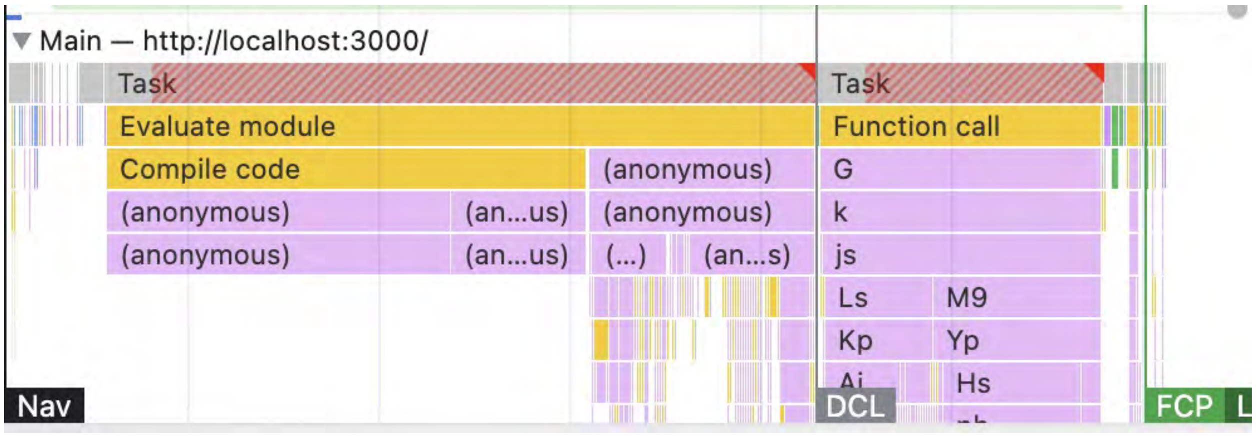
Open the tab with the **Baseline Project** we started earlier, get rid of all the Network throttling, uncheck the "disable cache" checkbox, and instead set the CPU throttling to 6x slowdown. This will imitate repeated visitors on slow devices. Record the initial load performance graph. It should look something like this:



There will be yellow JavaScript bars followed by the LCP metric - our usual Client-Side rendering pattern. The first bar is the Evaluate[85] task - this is when the browser processes all the downloaded JavaScript and prepares it to be executed. The second bar is exactly that - the browser executes the JavaScript from the first task. If you click on it, you'll see in the Summary tab that it originated from the `index.js` file. The first one didn't.

The first yellow task takes around **45 ms**, the second around **270 ms** for me. Your numbers might be different.

Now open the tab with this chapter's Study Project (the one with 5 MB JavaScript), and record the same profile on the same page with the same settings. The picture should look something like this now:



Still the same two yellow bars of JavaScript followed by the LCP metric. Only this time, the first task takes around **740 ms**. Almost ***16 times*** longer! The second task stays unchanged and hovers around 270 ms.

This is the cost of too much JavaScript right there. The page is exactly the same, the code for it is exactly the same, but all this JavaScript that is never even used on this page is slowing it down.

In a way, this slowing down is even more important than the network cost, even though it seems insignificant in comparison. The network costs users will pay only once, the very first time they visit the website. After that, the static resources should be cached by the browser. However, the slow Evaluate task will happen on every visit.

Bundle Size and SSR

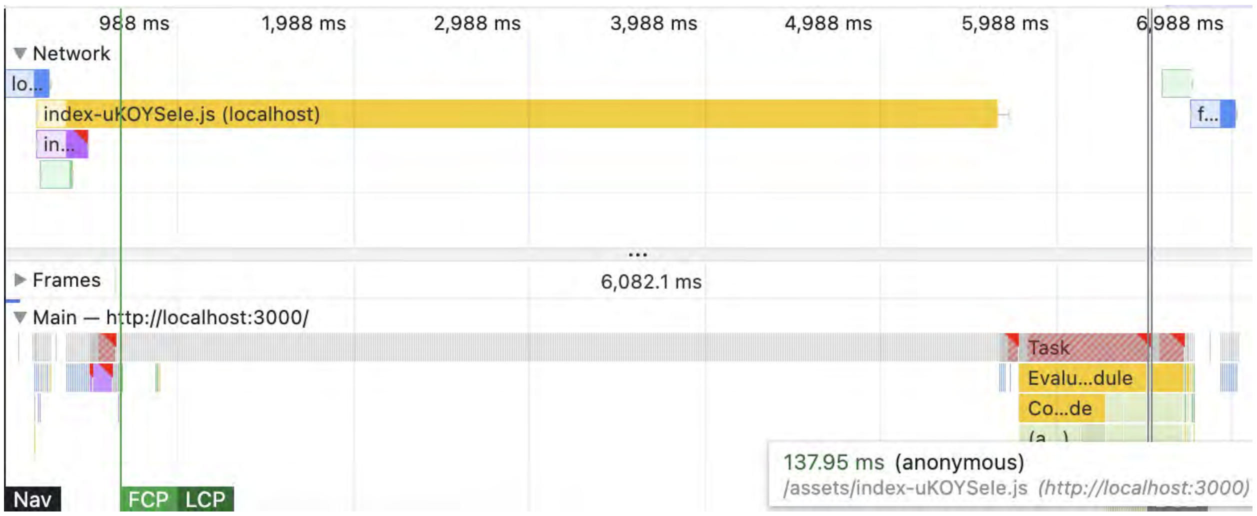
The combined effect of slow Network and/or low CPU makes large bundle sizes **very** problematic for SSR. Which is a bit counterintuitive - isn't SSR great for performance in most cases, like we proved in the previous chapter? Let's see for ourselves.

Navigate to `src/chapter7-bundle-size/simple-frontend/server/index.ts` and uncomment the row that enables SSR:

```
return c.html(preRenderApp(html, c.req.path));
```

Now open the Performance panel again, enable CPU throttling, and set Network throttling to slow 4G. Start the Performance recording, and as soon as the page shows up, try to trigger the toggle at the top right or open a menu by clicking the three dots button. It won't work for a while. Can you guess why?

The answer is in the Performance profile.



At first, as usual for SSR, we have the HTML downloading, then the CSS/JS download is triggered, and then LCP happens. The page is visible now. Since the setting is 4G, all of this happened relatively quickly, the LCP value is around 1.5 ms. But then, we have a loooooooong yellow bar (almost 6 seconds) of JavaScript downloading. After that, almost a second is spent processing and executing JavaScript. Only after the JavaScript is executed is hydration done, and all the event listeners are attached.

During these 7 seconds, there is no JavaScript on the page. It's just pure pre-rendered HTML. The only things that will work are links and native CSS things like focus selectors and animations, if they are present. The page *seems* very fast, but in reality, it is *broken* for 6 seconds. And that's a 4G setting! Set it to "3G", and you'll probably be able to take a nap before the website becomes functional.

This time when the page becomes fully interactive is known as the Time To Interactive (TTI)[86] metric. Unfortunately, it was removed from Lighthouse and Core Web Vitals, but it's still a useful metric to measure and take into account when analyzing the performance of an SSR-ed website.

Comment out the SSR part back - we'll be dealing with Client-Side rendering for the rest of the chapter, for simplicity.

Reducing Bundle Size with Code Splitting

Now that we know the impact huge bundle sizes can have on our website's performance, it's time to do something about those 5 MB of JavaScript. There are multiple tools and techniques for that. We'll start with Code Splitting.

The idea behind Code Splitting is this. Let's say we have 5 MB of JavaScript we need to send to the browser. If those 5 MB are packed into just one file, as it is now, downloading it will take more than a minute. As we've seen. However, browsers can send requests in parallel. What if we split that huge file into several files and request them at the same time? In theory, the download time should shrink.

However, we can't just cut that file into 10 random pieces - that's not how JavaScript works. We need to split it into isolated, independent modules, for starters, and then make sure those modules can call functions from other modules.

Doing this manually would be a huge headache. But luckily, we don't have to: modern bundlers can do it for us. They can trace all the dependencies and imports of all the code we ask them to compile, build a huge dependency tree out of it, and then slice and dice that graph in any way we want.

Those smaller pieces are known as "chunks". Some of the bundlers will create them automatically without your control to simplify the configuration. Some of them, like Vite, the bundler that the Study Project is using, have a simple configuration[87] that gives you some way to control it. Some, like Webpack,[88] give you absolute freedom to build as robust a Chunking Strategy as you want, and you can even configure it so that it splits the JS into chunks automatically when it becomes too big.

Let's try the manual approach first. We need to understand what strategies and best practices are possible, and why, before we can delegate that job to the advanced frameworks.

"Vendor" Chunks

The very first thing to do when starting on the code splitting path is to separate the "app" code - everything you write in your repo, from the "vendor" code - everything you use as a dependency. There are two reasons for that. First, of course, is the code splitting itself: the parallelization of the requests idea.

The second reason is the file names and caching. As you hopefully remember from the previous chapters, modern tools will generate file names with hashes in them by default. I.e., our `index.js` file in reality will look like `index.blabla.js`, where the `blabla` part is a hash value[89] of the file content. It only changes when the file's content changes.

If the entire app's code, including libraries like React itself, is bundled together in just one file, the file will change every time you change something in the app. This is terrible for the initial load since we can't rely on browser caching in this case. Every time there is a deployment, the large JavaScript file gets a new name, and all your users are downgraded to first-time visitors from a performance perspective.

Additional Challenge.

1. Build the Study Project.
2. Navigate to the `dist/client` folder. In the `assets` folder, you should see a JavaScript file with a hash in its name.
3. Remember the name of the file and delete the `dist` folder.
4. Rebuild the project again.
5. The file's name should be the same since the content hasn't changed.
6. Try to add anything to the project, a simple `console.log('bla')` in the `App.tsx` file would do.
7. Rebuild the project again.
8. The file's name will change.

If, however, we extract the dependencies part into `vendor.ayayay.js`, this file will change only when you update your dependencies. Which happens much more rarely than writing regular code. As a result, the browsers will be able to cache that file for much longer.

Different bundlers will have different ways of configuring it with different granularities. In Vite, the bundler we use for the Study project, we can do this by configuring manual chunks via Rollup[90]. Open the `vite.config.ts` file at the root of the Study Project and find and uncomment this code:

```
manualChunks: (id) => {
  if (id.includes('node_modules')) {
    return 'vendor';
  }

  return null;
};
```

The `id` here is the absolute path of all the files included in your project, including everything installed in `node_modules`, our external dependencies. As you can see, we're commanding Vite to group everything that includes `node_modules` in their file path under the `vendor` name.

Now rebuild the project. For the fun of it, you can include `console.log(id)` inside that configuration and see for yourself all the files it needs to process to build our beautiful app.

After rebuilding, peek into the `dist/client` folder. In the `assets`, you should now see two JavaScript files: `vendor.smth.js` and `index.bla.js`. Remember the exact names the bundler produced. Now, make some changes to the app's code and rebuild the project again. Only the `index.bla.js` file's name should change. The `vendor.smth.js` stays the same.

Additional Challenge.

1. Launch the project.
2. Open DevTools, navigate to the Network panel, make sure that the "disable cache" checkbox is turned off, and refresh the page.
3. Both `index` and `vendor` files should have `(disk cache)` or `(memory cache)` in the `Size` column - they were cached.
4. Now keep the project launched, make some changes in the app, and rebuild it in a separate terminal window.
5. Refresh the page.
6. Only the `index` file should be downloaded. The `vendor` file stays cached.
7. Slow down the Network and measure performance for both scenarios, and for the same scenarios without the vendor chunk, to compare the difference.

Funnily enough, if you measure the JavaScript compile time with and without the `vendor` chunk, you'll see that just adding the `vendor` chunk reduces it by half. So we're improving performance on two fronts here.

We can make those chunks as granular as we want. We could, for example, try to extract everything that is considered "design systems", i.e., buttons, checkboxes, modal dialogs, etc., into their own chunk.

```
manualChunks: (id) => {
  // everything that is inside this folder, will end up in its own chunk
  if (id.includes('frontend/components')) {
    return 'components';
  }

  return null;
};
```

Split the code per page or even per feature:

```
manualChunks: (id) => {
  // The code for the dashboard page will end up in its own chunk
  if (id.includes('/pages/dashboard')) {
    return 'dashboard';
  }

  return null;
};
```

Or even extract certain libraries into their own chunk if we know that some of them are heavy:

```
manualChunks: (id) => {
  if (id.includes('node_modules')) {
    // All external dependencies will go into "vendor" except for Radix UI
    if (id.includes('@radix')) {
      return 'radix';
    }
    return 'vendor';
  }
  return null;
};
```

Additional Challenge.

1. Go to the `frontend/components/button/index.tsx` file, it's a normal Button component that is used everywhere in the code.
2. Add a console.log somewhere there with some unique string that you can search for, for example, `console.log('Some identifiable string');`
3. Delete the `dist` folder in the Study Project, remove all the chunks from the `vite.config.ts`, and rebuild the project. The build should produce only one `index.bla.js` file.
4. Search for the string you put in the button component - it should be present in this `index` file.
5. Add the "vendor" chunk as described above, delete the `dist` folder, and rebuild the project.
6. The string (and the button itself) should still end up in the `index` file, with the `vendor` chunk containing the majority of JavaScript.
7. Add the "components" chunk as described above, delete the `dist` folder, and rebuild the project again. The button and your unique string should end up in the `components` chunk now.
8. Pay attention to the sizes of all the chunks and how they are changing depending on the configuration.

Developing a good chunking strategy for your own application is going to be a lot of trial and error and will largely depend on how many external dependencies you have and how you write your code. There are, however, a few things to watch out for while experimenting with chunks.

Chunks Preloading

All those "manual" chunks, regardless of the tool, will be preloaded as critical resources in one way or another. If you implement some of the chunking strategies from above, build the project, and sneak peek into the generated `index.html` file. You'll see that every one of those chunks, except for the default `index`, ends up inside the `<head>` as `<link rel="modulepreload" />`.

```
<script type="module" crossorigin src="/assets/index-pLa1GZqS.js"></script>
<link rel="modulepreload" crossorigin href="/assets/vendor-DIQ9qPEN.js" />
```

This allows the browser to fetch and compile[91] all this JavaScript in parallel, which is good. However, we didn't indicate to the bundler which of those chunks are not really critical for the initial render. So the browser has no choice but to wait for all of them to render our Study Project.

If you start the project, check the "disable cache" checkbox, and record the initial load performance, you'll see that the structure and order of the bars didn't change. There will still be a long wait for JavaScript download, and then compile/execution bars. Only after that will the UI show up, and the LCP value will be triggered. It's just that instead of one very long yellow JavaScript bar, there will be a few shorter ones in parallel.

There are a few more things to pay attention to here.

HTTP/1 vs Others

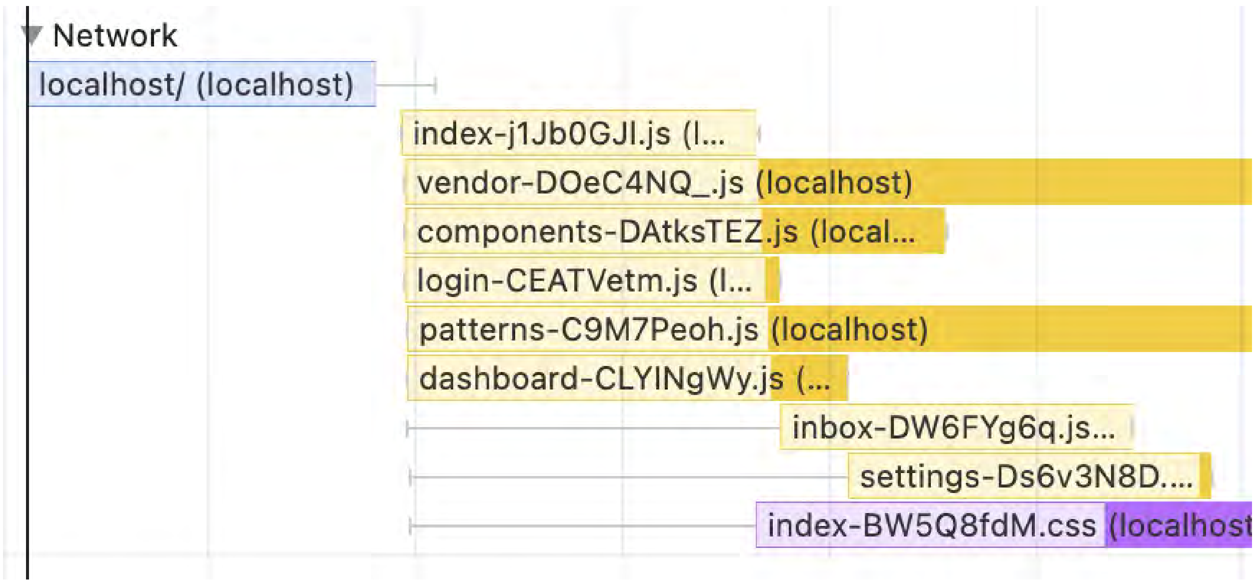
First of all, the HTTP/1 vs HTTP/2 vs HTTP/3[92] protocol and concurrent connections. If your website still uses the HTTP1 protocol[93], the browsers will impose limits on concurrent connections per domain. In Chrome, for example, it's just six requests from the same domain in parallel.

So if you try to preload too many resources like this, and your website uses an HTTP/1 server, the browser will have no choice but to delay the downloads of other resources. Including critical CSS files!

You can see this effect for yourself if you introduce more than six chunks into the Study Project. Try, for example, something like this:

```
manualChunks: (id) => {
  if (id.includes('node_modules')) {
    return 'vendor';
  }

  if (id.includes('frontend/components')) {
    return 'components';
  }
  if (id.includes('pages/inbox')) {
    return 'inbox';
  }

  if (id.includes('pages/dashboard')) {
    return 'dashboard';
  }

  if (id.includes('pages/login')) {
    return 'login';
  }

  if (id.includes('pages/settings')) {
    return 'settings';
  }

  if (id.includes('frontend/patterns')) {
    return 'patterns';
  }

  return null;
};
```

It will extract chunks for vendor, components, patterns, and every page in the app - 8 chunks in total. Build the project, start it, record the performance profile, and look at what is happening in the Network section of it:


Network
localhost/ (localhost)
index-j1Jb0GJl.js (l...
vendor-DOeC4NQ_.js (localhost)
components-DAtksTEZ.js (local...
login-CEATVetm.js (l...
patterns-C9M7Peoh.js (localhost)
dashboard-CLYINgWy.js (...
inbox-DW6FYg6q.js...
settings-Ds6v3N8D....
index-BW5Q8fdM.css (localhost

There are only six JavaScript chunks requested first. The download of the `inbox` and `settings` chunks is triggered only when one of the previous chunks stops downloading.

And what's worse, the CSS file is also delayed! So if the page was SSRed, splitting JavaScript into chunks here would've delayed the initial load because of the CSS file, and as a result, we'd make our performance numbers ***worse***.

How much of a problem is it in real life, though?

Limits like this are **only for the HTTP/1 protocol**. HTTP/2 and HTTP/3 can handle many more requests in parallel. According to Cloudflare statistics[94] from 2022-2023, HTTP/1 is used in about 10% of the traffic that goes through them. I checked a few other CDN providers, and all of them seemed to be either on HTTP/2 or HTTP/3 by default. So chances are, your "production" website is fine if it uses a CDN.

Our *local* website, however, was using HTTP/1 by default. HTTP/2[95] needs to be implemented separately if I want to see it locally. The same story with Express[96] (a very popular Node framework) and even with everyone's favorite Next.js[97]. So, even if your "production" website is fine, your local environment will likely not be.

This raises a very interesting problem. If you're not aware of this difference and start experimenting with different chunking strategies and load orders locally, for the purpose of improving initial load, you might see exactly zero gains from those experiments in production. Or the opposite - locally it might seem like not a good idea (like creating more chunks), but in production, you'll see a massive improvement.

Chunk Sizes

There is another downside to splitting our code into too many chunks. The compression ratio actually decreases[98] as the chunks become smaller. So, the overall network transfer of compressed JavaScript might increase, even if the initial load decreases. This may be important for users with limited bandwidth, especially those using mobile devices.

Unnecessary Code Is Still Loaded

There is another problem with over-relying on splitting code into chunks manually like this. We're still loading a massive amount of unnecessary JavaScript on the user's critical path. There is no way a simple dashboard like ours actually *needs* all those 5 MB. There are also multiple pages on the website, like "Inbox" or "Login". We don't need to load them to show the dashboard.

If we just delay loading those pages and all the libraries that are not necessary for the dashboard page, surely the initial load would improve?

This is where the concept of "dynamic import"[99] comes in, which is the ability to do exactly that - load JavaScript modules "on the fly".

However, it's not as simple as just loading everything that we can "dynamically". Introducing dynamic imports comes with its own complexity and trade-offs. And more importantly, we still need to know what exactly should be extracted into a "dynamic" load.

If you implemented the "vendor" chunk, take a peek into the `dist` folder and notice the sizes of the produced chunks. You'll have a small-ish `index` file and a 5 MB `vendor` file. A good chunk size[100] is around 50-100 KB, anything else needs to be split. So there is not much point in splitting the `index` file here, whether manually or dynamically, while we still have 5 MB of dependencies left.

What I'm leading to here is that it's time to understand how we ended up with this outrageous amount of JavaScript in the first place. Maybe we can just get rid of most of it and won't need to do any more code splitting at all? We'll return to the dynamic imports after we slim down the bundle a bit.

Analyzing Bundle Size

Okay, so how exactly did we end up with 5 MB of JavaScript in the Study Project? If you implemented any of the chunking strategies from above, let's get rid of them and leave only "vendor" and "index" chunks:

```
manualChunks: (id) => {
  if (id.includes('node_modules')) {
    return 'vendor';
  }

  return null;
};
```

This will produce two JavaScript files: one with all the external dependencies and one with all the code we wrote. The "vendor" file is the biggest offender, with 5 MB of minified JavaScript, so let's look into that one first.

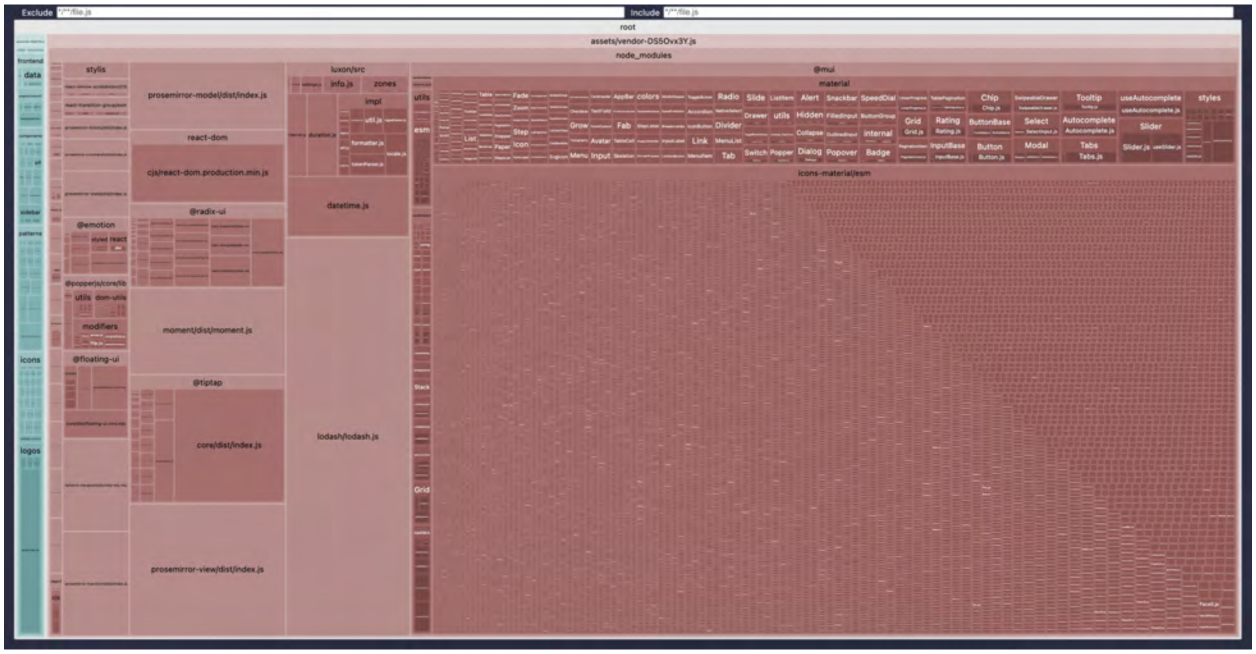
By "look", I don't mean actually opening it in the IDE and staring at it. While we can certainly do that, it will give us exactly zero information and might even cause a headache.

This is the job of the tools known as "bundle analyzers". For our Study Project, we'll use a "Rollup Plugin Visualizer"[101] library. If you're using a different bundler for your project, just google "name-of-your-bundler + bundle analyzer" - there are lots of them out there.

In your Study Project, open the `vite.config.ts` file and find and uncomment this code:

```
visualizer({
  filename: 'stats.html',
  emitFile: true,
  template: 'treemap',
});
```

It will enable the analyzer. Rebuild the project and take a look inside the `dist/client` folder. There should be a `stats.html` file there. Open it in your browser and wait until it fully loads (might take a while!). When it finishes loading, it should show a graph like this:



It's a hierarchical visualization of every single JavaScript file in the project. It starts from the "root" at the very top - this is the root of the project, i.e., our `dist` folder. Inside, there are two blocks:

- The largest red-ish one is `assets/vendor` - this is our vendor chunk.
- The teal at the left is `assets/index` - this is our own code.

The size of the blocks is relative to the size of the code, so it's pretty obvious from even a brief glance at the picture that the "index" is tiny compared to the red "vendor". Inside the `vendor` block, there is a `node_modules` block, different libraries, and so on, and every file is grouped by the file path parts.

Hovering over every block will give you the exact path.

Clicking on any block allows you to "zoom in" and peek at what's inside.

Additional Challenge.

- Click on the `node_modules` block - it will eliminate the visual distraction of the teal block on the left.
- Click on the `@mui` block - it will zoom in on what is bundled from the `@mui` library.
- Click on the `material` block inside - those are components inside that library.

- Click on any of the blocks inside to zoom in even further.
- Click on the `node_modules` block again to zoom out back to the content of the entire `node_modules`.

If you want to have more fun with it, you can generate different types of visualization. For example, if you change the config to this:

```
visualizer({
  filename: 'stats.html',
  emitFile: true,
  // different types of visualizations
  template: 'flamegraph',
});
```

And rebuild the project, it will give you an already familiar Flame Graph instead of this two-dimensional map.

After you have the visualization, it's a matter of putting your detective hat on and starting the investigation. Usually, it just means staring at the map until your eyes are watering, noticing unreasonably large areas, recognizing the library from it (or googling it if you have no idea), then going through the code and trying to understand whether you can remove the usages of this library and what it will cost.

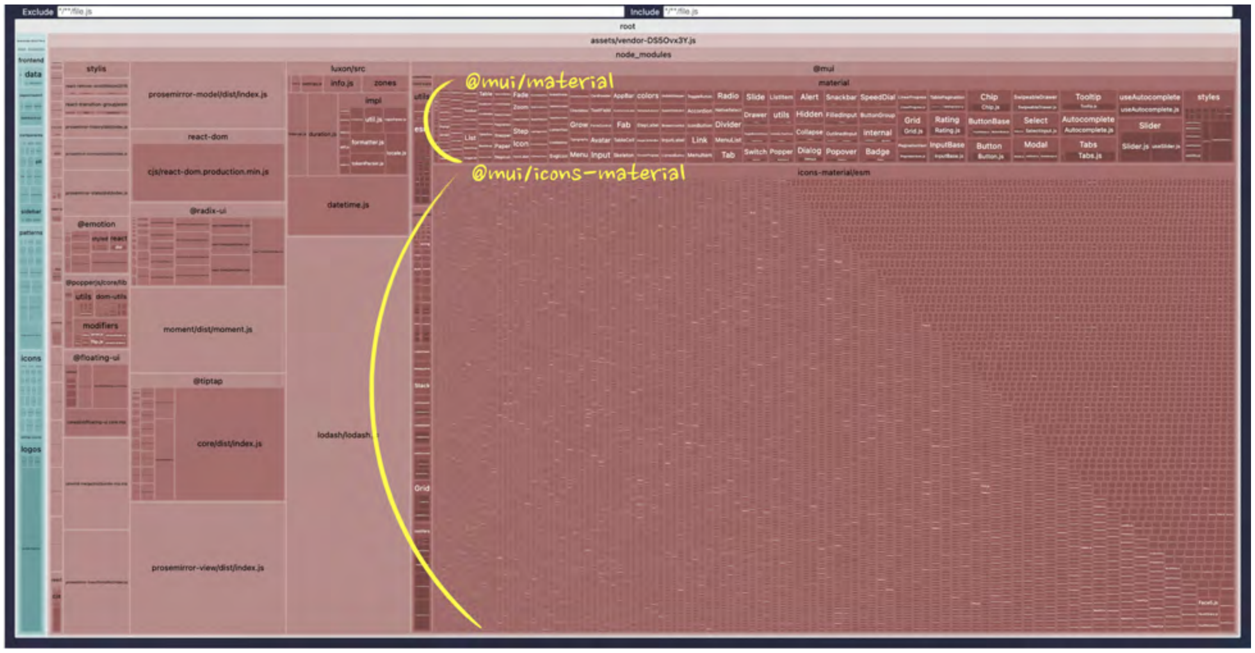
Let's do it together to understand the process. We'll follow this process for every package we're going to investigate.

Investigation Process

Step 1: Identify a Package to Eliminate

The very first huge block that I immediately notice here is everything under the `@mui` title inside the `node_modules` block, which contains a number of npm packages installed as the project dependencies. And we know naming convention[102] for npm packages - it's either one word (with dashes), or two words separated by `/`, where the first word is a namespace and starts with `@`. So everything directly under the `node_modules` title is either a package or a namespace for multiple packages.

Since `@mui` starts with `@`, it's a namespace, and everything directly under it will be a package. This gives us two packages: `@mui/material` and `@mui/icons-material`.



Everything inside is the content of those packages.

Step 2: Understand the Package

Quick googling tells us what those packages are: the `material` package is Google's Material UI[103] components library, and `icons-material` is a set of icons[104] for this library that is installed separately.

If we zoom in on the "material" package, we'll see that it includes *all* of the possible components. I can see Snap, Alert, Tooltip, etc - hundreds of them. The same story if I zoom in on the "icons-material" block - looks like the entire set of 2000 icons is included in the bundle.

No wonder the bundle is 5 MB!

Step 3: Understand the Usage of the Package

For this step, we need to read a lot of code - we need to understand where exactly those packages are coming from. The very first thing we can confirm is whether we're using those packages directly in our code or if something else is using them indirectly. Luckily, for those two, it's easy: we just need to do a text search across the code inside the `frontend` and `src` folders for `@mui` - those are the only places where we have our frontend code.

The search will give us two files: `frontend/icons/index.tsx` has an import from `"@mui/icons-material"`, and `frontend/utils/ui-wrappers.tsx` imports from `"@mui/material"`.

The code that uses the icons looks like this:

```
import * as Material from "@mui/icons-material";

export const Icons = {
  ...Material,
  BellIcon,
  ... // other icons
};
```

Clearly, someone was trying to unify the usage of all the icons in the project. The intent here likely was that all the icons would be grouped under the same `Icons` namespace, with the assumption that it would help reduce the chance of having icons with the same name in the project and make it easier in the future to move icons to a new library if there is a need. With a pattern like this, in the code you wouldn't import icons directly from `"@mui/icons-material"`, but rather import them all from this file and use them like this: `<Icons.BellIcon />`.

If you do the search through the project for `Icons.` (the dot at the end allows us to narrow it down), you'll see that this is exactly what is happening in three files: two dialogs and a message-list component. Or you can search for "usages" if your IDE supports that, of course.

In theory, it's quite a noble idea, and indeed would've made future refactoring much easier - you'd need to refactor just one file if you want to replace the icons, and the rest of the code wouldn't even know that something changed. Plus, it makes it super easy to see which icons are available in the project via autocomplete (if your IDE supports it).

In practice, we ended up with two thousand icons in our bundle 😳.

And exactly the same story with the "material" library usage:

```
import * as Material from '@mui/material';

export const StudyUi = {
  Library: Material,
  Button: Button,
};
```

Someone wanted to expose all the available components through a unified interface, probably for exactly the same reason as the icons. That's one of the reasons why we're investigating them together here 🙄 Search for `StudyUi.Library` to confirm that this is indeed what is used somewhere in the code.

Step 4: Confirm That This Is the Problem

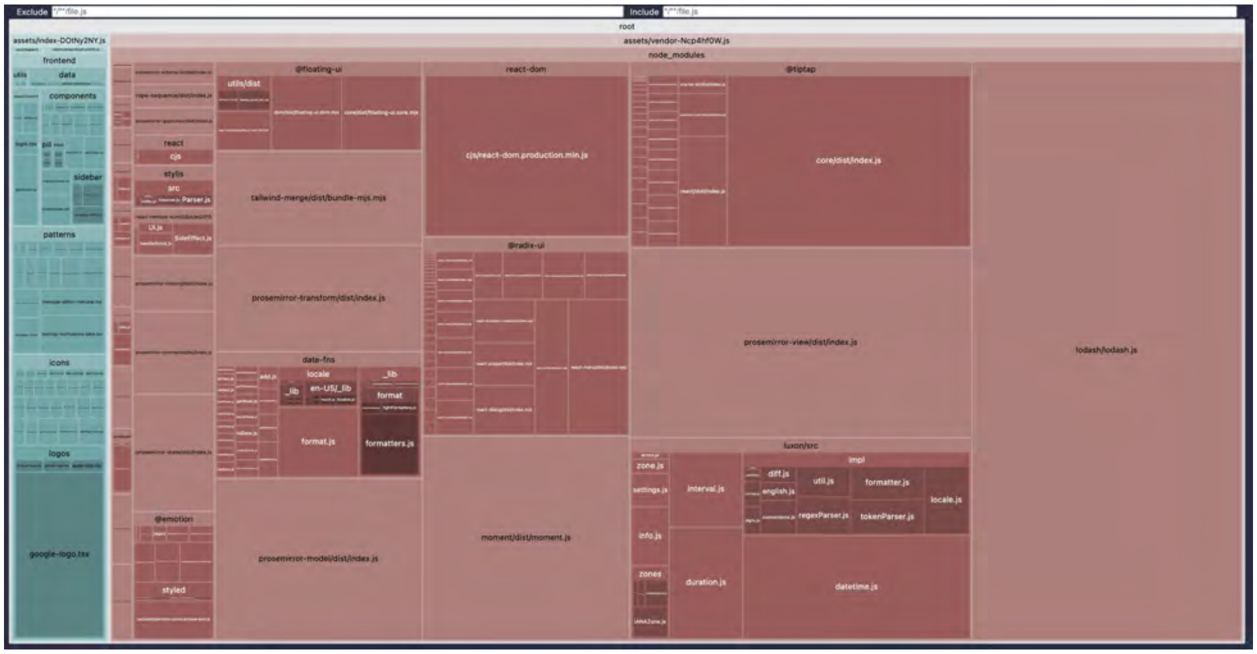
Before attempting any refactorings, which in real life could be very costly, we first need to confirm that we have identified the problem correctly.

For now, let's just comment out the import part for both of those libraries.

```
// Just comment out those imports everywhere

// import * as Material from "@mui/material";

// import * as Material from "@mui/icons-material";
```

And then rebuild the project. It won't start since we haven't fixed the usage of those libraries yet. But it will be enough to see whether the bundle size has reduced or not, and confirm whether those imports were the problem.

And indeed it works! The "vendor" file shrunk **from 5 MB to 811 KB**, and the visualization now looks like this:



The teal "index" chunk is much more visible since we shrunk the `vendor` so much. There are no more `@mui` packages, and other libraries like `prosemirror-view` and `lodash` have become much more visible.

Now, all we need to do as a final step is to fix the problem properly. But first, we need to identify what the actual problem here is. Surely not everyone who uses MUI components and icons ends up with 5 MB bundles because of them? No one would use them in this case. So, something is wrong in our code.

To understand this, we need to know a concept known as "tree-shaking".

Tree Shaking and Dead Code Elimination

Modern bundlers not only merge JavaScript modules together. They also try to identify and remove "dead code", i.e., the code that is not used anywhere. And they are pretty good at it.

Try, for example, adding this code somewhere:

```
export const MyButton = () => <button>Click me</button>;
```

Let's say in `frontend/components/button/index.tsx`, where we keep all our buttons.

And then rebuild the project.

You should notice that the `index` chunk stays with exactly the same name and exactly the same size with or without this code. That's because we're not using this button anywhere, it just sits there.

Now, try to add a new `MyDialog` component somewhere in `frontend/components/dialog/index.tsx` that uses this button:

```
import { MyButton } from '@fe/components/button';

export const MyDialog = () => {
  return (
    <>
      <MyButton />
      <div>My dialog</div>
    </>
  );
};
```

Then rebuild the project. The result should be exactly the same! Same name for the chunk, same size. We *still* haven't used this code - the `MyDialog` component still sits there and does nothing. The bundler was able to detect that and got rid of both `MyDialog` and `MyButton` in the production files. Crazy smart, right?

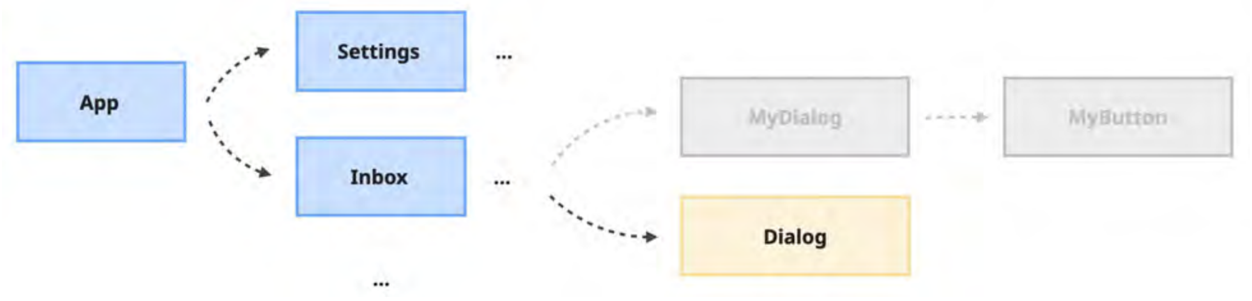
Only when the component is used in the code that forms the app *for real* will it be included. Try to render the `MyDialog` somewhere inside `App.tsx`, for example:

```
import { MyDialog } from '@fe/components/dialog';

export default function App() {
  // keep the rest of the code as is
  if (path.startsWith('/settings')) {
    return (
      <>
        <SettingsPage />
        <MyDialog />
      </>
    );
  }
  // keep the rest of the code as is
}
```

And rebuild the project. The `index` chunk name changes, the size slightly increases. You can even open the `index` chunk and search for the "Click me" string to verify that the new button is included.

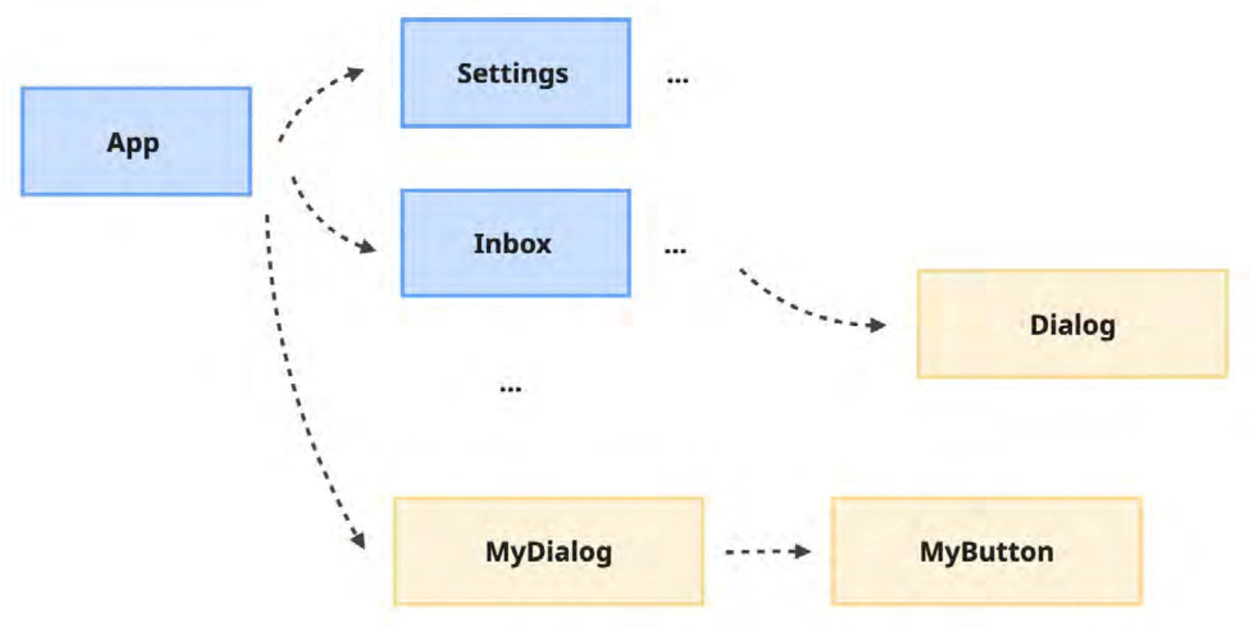
This process of eliminating unused code is known as "tree-shaking"[105].

It's called this way because the bundler creates an abstract "tree" from all the files and exports/imports within the files, tracks down "alive" and "dead" branches of that tree, and then removes the "dead" ones. Before we included the `MyDialog` in `App.tsx`, the "tree" would look something like this (simplified):



The `index.tsx` file inside the `frontend/components/dialog` folder exports multiple components, including a generic `Dialog` that is used in a few places. Our `MyDialog`, which is not used anywhere, is marked in gray (i.e., a "dead branch"). The gray branches will be excluded from the final bundle.

When we explicitly included `MyDialog` in the `App.tsx`, the tree changed into this:



The `MyDialog` branch is not dead anymore, and as a result, it's included in the bundle.

Modern bundlers are getting smarter and smarter, and it becomes harder and harder to fool them when it comes to tree-shaking. It's still possible, however, for a determined person 😅

One of the things they can't deal with yet is the `*` import in combination with renaming. `*` import is this:

```
import * as Buttons from '@fe/components/button';
```

It's basically a command to import *everything* from the module and alias it as `Buttons`. Then, we can use the buttons we need via dot notation:

```
<Buttons.SmallButton />
```

This pattern is quite popular, especially when there are many exports from one module, when you want to avoid importing them one by one:

```
import { Button, SmallButton, LargeButton } from '@fe/components/button';
```

And by itself, this `*` import is actually not enough to confuse the bundler - told ya they are smart! However, when it's used as a variable, not just the means to extract what's inside... The bundlers can't handle it yet.

This scenario is a classic example:

```
import * as Buttons from "@fe/components/button";
import * as Dialogs from "@fe/components/dialog";
// the rest of the components

export const Ui = {
  Buttons,
  Dialogs,
  ...
};
```

Try to add this code to the `App.tsx` file instead of the previous example and render a "normal" button using the dot pattern:

```
import * as Buttons from '@fe/components/button';
import * as Dialogs from '@fe/components/dialog';

export const Ui = {
  Buttons,
  Dialogs,
};

export default function App() {
  // keep the rest of the code as is
  if (path.startsWith('/settings')) {
    return (
      <>
        <SettingsPage />
        <Ui.Buttons.SmallButton />
      </>
    );
  }
  // keep the rest of the code as is
}
```

Then, rebuild the project, open the `index` chunk inside the `assets` folder, and search for the "Click me" string - the string we used in our `MyButton` button. Although we didn't use the `MyButton` explicitly, its code is now included there.

If you've never seen this pattern before, it might look a bit ridiculous. Why would anyone do that?

One of the reasons namespacing like that gained popularity is because it allows for much simpler imports and much more explicit code. For example, try putting this code in the `index.tsx` file in `frontend/components` and add the rest of the components from `@fe/components` to the imports there.

```
import * as Buttons from "@fe/components/button";
import * as Dialogs from "@fe/components/dialog";
// all other frontend components

export const Ui = {
  Buttons,
  Dialogs,
  ...
  // all other components
};
```

Now, I can collapse individual imports of components *everywhere* to just this:

```
import { Ui } from '@fe/components';
```

Look at the `frontend/patterns/confirm-archive-dialog.tsx` file, for example. All of those:

```
import { NormalToLargeButton } from '@fe/components/button';
import { Dialog, DialogBody, DialogClose, DialogDescription, DialogFooter, DialogTitle }
from '@fe/components/dialog';
// one million other imports
```

Could've been just `import { Ui } from "@fe/components";`

And everything else would've been used via a namespace:

```
<Ui.Dialogs.Dialog />
```

Many people love the clarity this pattern gives. For every component I use, I see where it originated from immediately within the context of the function. Plus, no name collisions, which is always nice.

But as a result, this pattern confuses the bundler, **tree-shaking on this code doesn't work**, and our final JavaScript size is larger than it should be.

For our own code, this might not be that big of a deal - after all, everything that we write, we write with the intention of using it. And one or two forgotten functions won't make much of a difference.

When it comes to external libraries, however, it's a completely different story. Because this is exactly the pattern we used for our `@mui` components:

```
import * as Material from '@mui/material';

export const StudyUi = {
  Library: Material,
};
```

And icons:

```
import * as Material from "@mui/icons-material";

export const Icons = {
  ...Material,
  BellIcon,
  ... // other icons
};
```

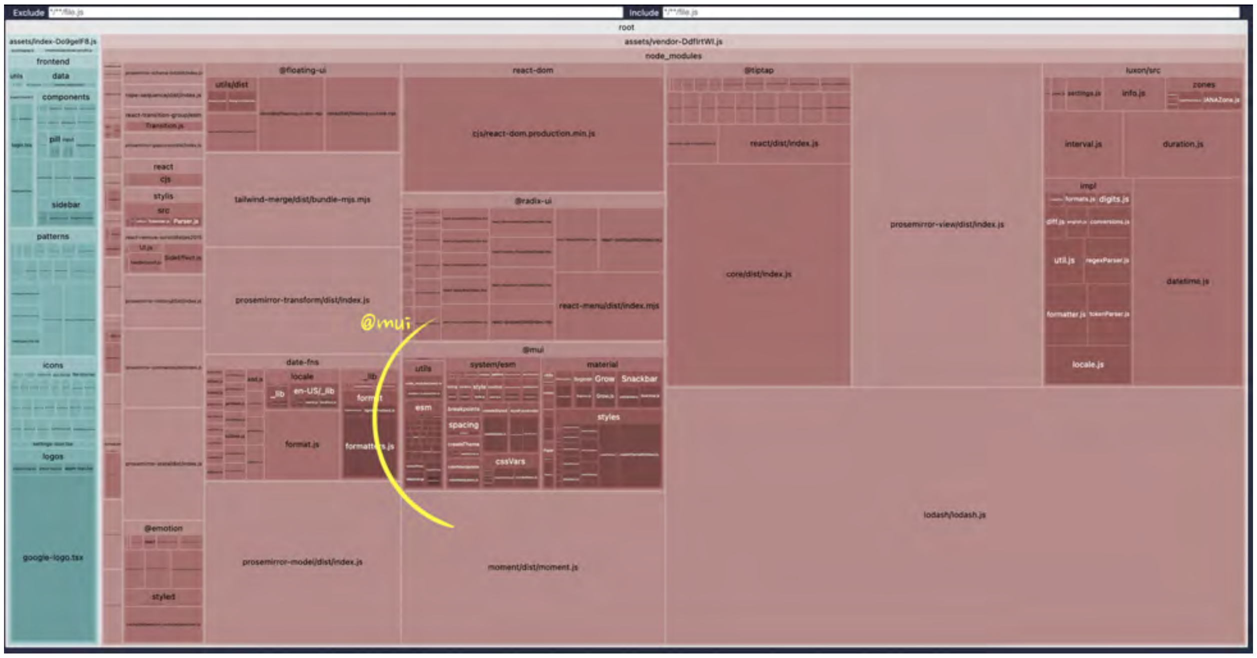
The quick fix here, if we want to preserve the pattern and namespaces and avoid global refactoring, is to get rid of the `*` import and import only the components and icons we use. Get rid of all the changes that we made in the frontend part of the project, and instead do this:

```
// frontend/utils/ui-wrappers.tsx file

import { Button } from '@fe/components/button';
import { Snackbar } from '@mui/material';
```

```
export const StudyUi = {
  Library: {
    // this is the only component we use from the Material library
    Snackbar: Snackbar,
  },
  Button: Button,
};
```

```
/// frontend/icons/index.tsx file

import { Star } from "@mui/icons-material";

// keep the rest of the imports

export const Icons = {
  Star: Star, // this is the only icon we use from the Material set
  ... // the rest of the icons
};
```

Rebuild the project again. The bundle is now **878** KB instead of 5 MB - we clearly got rid of unnecessary icons and components from `@mui`. Open the `stats.html` file - it now looks like this:

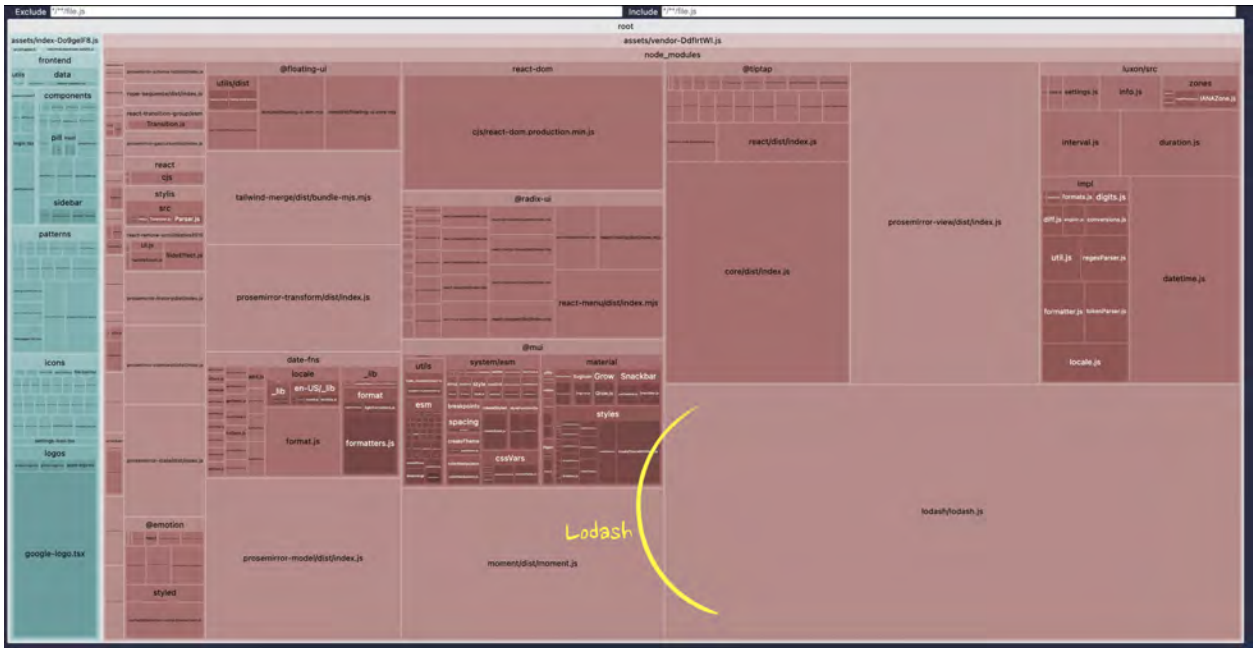


We still have the `@mui` block here since we do use it. But now it's much smaller and overshadowed by other larger blocks. So let's consider our "mui" problem solved for now and look at the other problems.

But before that, we need to make sure that the fix didn't break the app. Start the project and navigate to the "Inbox" page: you should see the gold star at the beginning of each message - that's the star icon we used from MUI. Click on the "delete" button that appears when you hover over any message, and click the "Yep, do it!" button - a notification should appear at the bottom left corner of the page. That's our `Snackbar` component from MUI. Everything works as expected!

ES Modules and Non-tree-shakable Libraries

Now that we fixed the `@mui` dependencies and their block doesn't take the entire screen, we can see other problematic inclusions into the bundle more clearly. For example, there is this big lump of "lodash" on the bottom right. What's going on there? Why is it so big?



We'll apply exactly the same process for the investigation. First, quick research into the Lodash[106] library - it's a JavaScript library that implements quite a number of utilities[107] for arrays, objects, lists, and so on, that are mostly not available as native JavaScript functions.

Search for its usage throughout the project files gives us a single place - in the `./pages/inbox.tsx` file. This is the code, a bit simplified:

```
// FILE: ./pages/inbox.tsx
import _ from "lodash";

export const InboxPage = () => {
  const onChange = (val: string) => {
    // This is the only place where we use the library
    const cleanValue = _.trim(_.lowerCase(val));

    // Send cleanValue to the server
    console.info(cleanValue);
  };
  return ...
};
```

We import the entire library via `import _` and then use `trim` and `lowerCase` utils on a text string before sending it to the backend. Since it's a search field, it's safe to assume it's going to be used for async autocomplete, so the usage seems legit. Let's ignore for a second that we probably didn't need the library at all here, since all modern browsers support `trim` and `toLowerCase` already. The point of this exercise is to focus on bundle investigation and what kind of gotchas we can expect.

Let's focus on the fact that we use just two simple utils from a huge library here. There is no way those two simple functions need so much JavaScript. It's a clear indication that the tree-shaking has failed, and we imported the entire library and all of its content.

To validate this assumption (as we should do with absolutely any assumption when it comes to performance investigations), we can simply remove one of the utils:

```
// remove the lowerCase util, keep only trim
const cleanValue = _.trim(val);
```

If tree-shaking works correctly, the size of the `vendor` chunk should decrease a little and the name of the chunk should change since the unused `lowerCase` util will be "shaken out".

Notice the name and size of the `vendor` chunk, make the change from above, and rebuild the project.

Nothing changes. The tree-shaking doesn't work. Maybe it's because we're importing the entire library with `import _ from` and it confuses the bundler somehow? Change it to be an explicit import and try again:

```
import { trim } from 'lodash';

// inside onChange callback
const cleanValue = trim(val);
```

The bundle name changes, and the size changes by two bytes, which is clearly not enough to eliminate an unused util. It's probably just because we changed the name in the import. If you compare the resulting `vendor` chunks "before" and "after" this change in your IDE (if it supports this type of comparison), you'll see that this is indeed the case - just a few minified variables were renamed, the rest is still the same.

Tree-shaking doesn't work at all, as we have just proven.

The problem here is that we have **different module formats** in JavaScript: ESM, CJS, AMD, UMD. "Module" is a single reusable piece of code that can be loaded into another piece of code. These formats define how this reuse happens.

The full history of those modules, the comprehensive differences, and how they are used and distributed in modern tools would need another book. Fortunately, we only need to know one thing for the bundle size investigations.

When you see `import { bla } from "bla-bla"` or `export const bla` or `export { bla }` - it's **ESM format**. Our entire project is ESM, and it's pretty much the standard these days, at least when it comes to writing frontend code. Modern bundles can **easily tree-shake ESM format,** as we've seen already while experimenting with tree-shaking in our own code. Everything else that is **not ESM is very hard to tree-shake**.[108]

ESM is a relatively new format, and not all libraries have caught up with it yet. You can check whether a library is ESM or not with a tiny is-esm[109] npm package. It's a CLI tool that gives you a Yes/No answer. If Yes, it's ESM and it will be tree-shaken.

```
npx is-esm lodash
```

The answer here is **No**.

For comparison, run it on `@mui/icons-material` and `@mui/material` - the result will be **Yes**.

This answers the question of why the material packages were tree-shaken, but lodash was not.

Now, what to do about it? Unfortunately, the answer for some libraries, especially really old ones, will be "nothing". We either need to accept the consequences of the bundle size or get rid of the library altogether.

Some libraries, however, especially if they are actively maintained, will provide a workaround. While the "main" entry file is not ESM, they might provide additional entries for smaller pieces of the library that allow importing only what you need.

`@mui/icons-material` actually does that in addition to ESM format. You can import the icon that you need directly from the package and rely on tree-shaking to work:

```
import { Star } from '@mui/icons-material';
```

Or you can import the icon directly from its own entry point and not live in fear of tree-shaking failing for some obscure reason:

```
import Star from '@mui/icons-material/Star';
```

Whether a library provides this additional way to import is usually documented in some way or form. Material icons,[110] for example, suggest the precise import as a default way to use the icons.

If we look at Lodash documentation[111], they also mention those types of imports:

```
// Cherry-pick methods for smaller browserify/rollup/webpack bundles.
var at = require('lodash/at');
```

Let's try to use this in our project and see what it does to the bundle size.

The code we started with is this:

```
import _ from "lodash";

export const InboxPage = () => {
  const onChange = (val: string) => {
    // this is the only place were we use the library
    const cleanValue = _.trim(_.lowerCase(val));

    // Send cleanValue to the server
    console.info(cleanValue);
  };
  return ...
};
```

With the bundle size for the `vendor` chunk being around **878** KB.

We need `trim` and `lowerCase` utils. If we use the precise imports, it will transform into this:

```
// change the imports to be precise
import trim from "lodash/trim";
import lowerCase from "lodash/lowerCase";

export const InboxPage = () => {
  const onChange = (val: string) => {

    // Get rid of the _ and use the utils names
    const cleanValue = trim(lowerCase(val));

    // Send cleanValue to the server
    console.info(cleanValue);
  };
  return ...
};
```

Rebuild the project, and the bundle size goes down to **812.95** KB! Looks like it worked. Open the `stats.html` file to see that the huge Lodash block is barely visible now.

Although, if we're being serious, in an actual non-study project, I'd just remove those two: if I don't need IE9 support, `trim` can be replaced with native JavaScript trim[112], and `lowerCase` can be replaced with native toLowerCase[113]. The code will then turn into this:

```
export const InboxPage = () => {
  const onChange = (val: string) => {

    // Get rid of lodash completely
    const cleanValue = val.toLowerCase().trim();

    // Send cleanValue to the server
    console.info(cleanValue);
  };
  return ...
};
```

And after rebuilding, the bundle size goes down by another ~10 KB.

Common Sense and Repeating Libraries

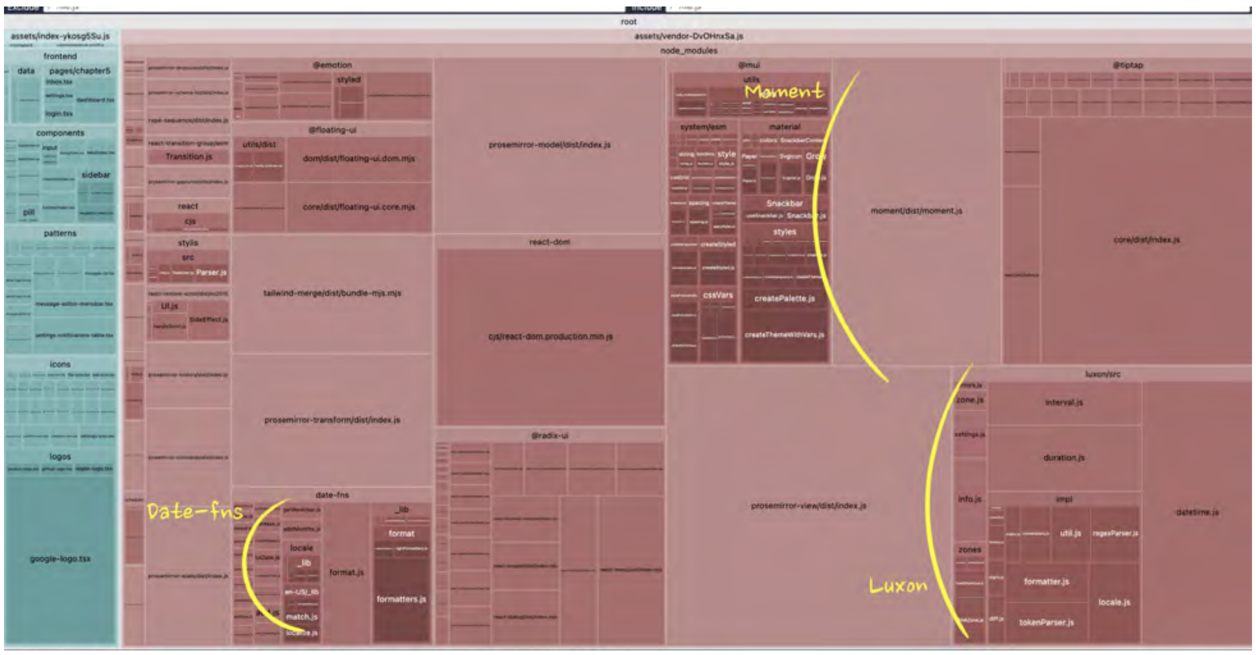
I'm getting more and more excited as our bundles shrink! There is something satisfying in removing code. Let's do more of this!

The next important thing to look at when investigating bundle sizes is common sense. I know, it sounds ridiculous, but you'll see what I mean in a second 😜.

The big advantage of using the modern open source ecosystem is that you can find a few libraries for pretty much any use case. The big disadvantage is exactly the same - for pretty much any use case, there will be a few libraries. And in big projects, especially when there are multiple teams involved, there is a pretty high chance that someday a few libraries that solve exactly the same use case will show up in the bundle.

This especially often happens with stuff like dates, animations, resizing, infinite scrolling, forms, charts, and so on - pretty much everything that is too painful or too complicated to implement from scratch and generic enough to be extracted into a library.

Let's look at our already slightly cleaned-up bundle chart and squint really hard at the highlighted areas.



We have three quite significant in size libraries: `date-fns`, `moment`, and `luxon`. Quick googling reveals that:

- date-fns[114] is a library for manipulating Dates in JavaScript.
- moment[115] is also a library for manipulating Dates.
- And luxon[116], you guessed it, also a library for manipulating Dates.

🤦‍♀️ Someone really didn't do any due diligence before introducing yet another Dates library.

What to do in this situation really depends on how much code would need to be refactored to get rid of some of them, how much effort it will take, and how many KB of bundle size we're ready to tolerate for the functionality a library gives us.

It might happen, especially in old projects, that the Moment library is used pretty much everywhere, and the newer Luxon and Date-fns are just in a few places. So, in this case, it could make more sense to get rid of the newer ones as a quick win if the time dedicated to the bundle size initiative is restricted, and focus on other areas. Or it could be the opposite, and Moment could be a leftover of a large refactoring that someone forgot to remove in a few places.

In our case, the project is very new, and each of the libraries is used only once. So the refactoring to unify usage will be easy.

In this case, it all comes down to which library allows tree-shaking or specific imports, which API I like the most, which one is maintained, and all the other things you usually consider when choosing a library.

Running the tree-shaking check reveals that `moment` is not tree-shakable, and a quick scroll through its documentation doesn't show anything that allows targeted imports like `lodash` does. So this one is out.

Luxon seems to be tree-shakable, but looking at our bundle chart, it's still much bigger than date-fns. So either tree-shaking is flawed there, or it's just naturally large. Doesn't really matter here, since date-fns is an option, I like its API anyway, and it's much smaller.

There is also the option of just removing all three of them - our use cases are pretty simple. But personally, a proper dates library will be the last library I remove from any project. I hate dealing with the native Date API in JavaScript. So I'll just refactor everything to date-fns.

In `frontend/patterns/message-editor.tsx` file, I have Luxon and this code that uses it:

```
// FILE: frontend/patterns/message-editor.tsx
import { DateTime } from 'luxon';

// inside MessageEditor component
const formattedDate = DateTime.fromMillis(timestampDate).toFormat('MMMM dd, yyyy');
```

It just converts a milliseconds value to a human-readable format, easy enough. In `date-fns` the alternative will be this:

```
import { format } from 'date-fns';

// inside MessageEditor component
const formattedDate = format(new Date(timestampDate), 'MMMM dd, yyyy');
```

In the `frontend/patterns/messages-list.tsx` file, I have Moment and this code that uses it:

```
// FILE: frontend/patterns/messages-list.tsx
import moment from 'moment';

// inside MessageList component
moment(message.date).format('MMMM Do, YYYY');
```

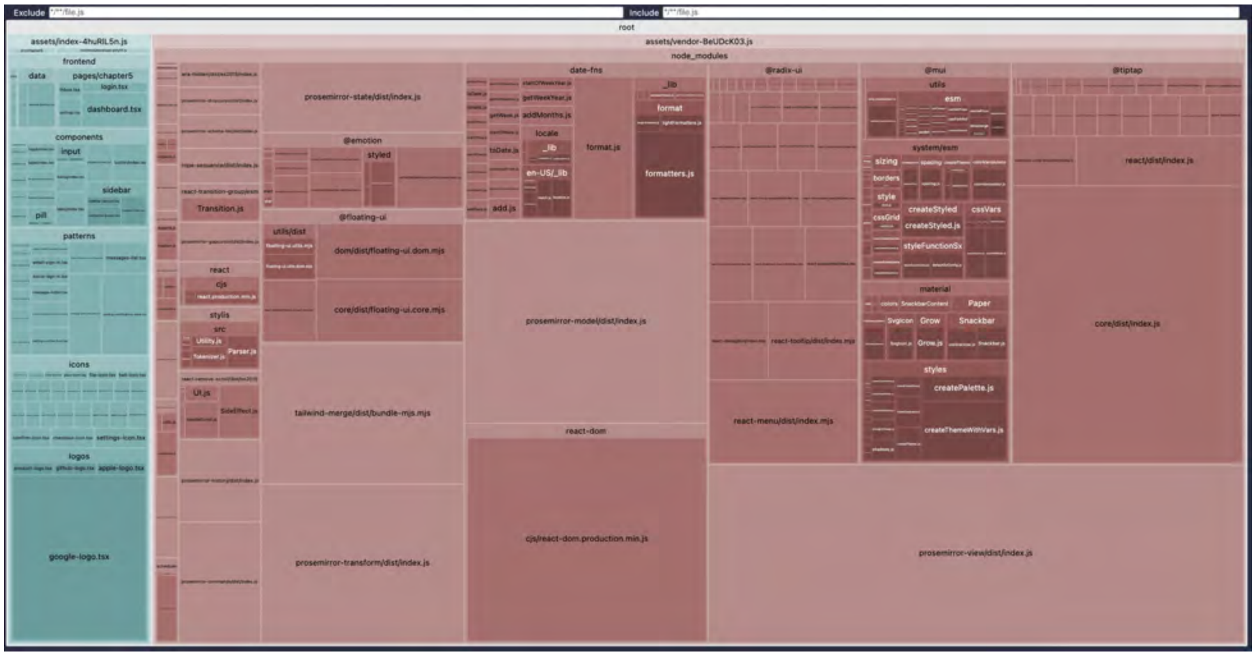
Exactly the same use case as with Luxon - I have a date in milliseconds that I convert to a human-readable format.

Refactoring it into date-fns:

```
import { format } from 'date-fns';

// inside MessageList component
format(new Date(message.date), 'MMMM do, yyyy');
```

Rebuilding the project, and boom! Bundle size drops by 20%, from **804.34** KB to **672.52** KB. Check the `stats.html` to enjoy the lack of huge Moment and Luxon blocks 💃 ▨.

Let's remove something else while we're on a roll. The chart now looks like this:



There are lots of packages with `prosemirror` and `tiptap` in the name - we'll deal with them in the next section.

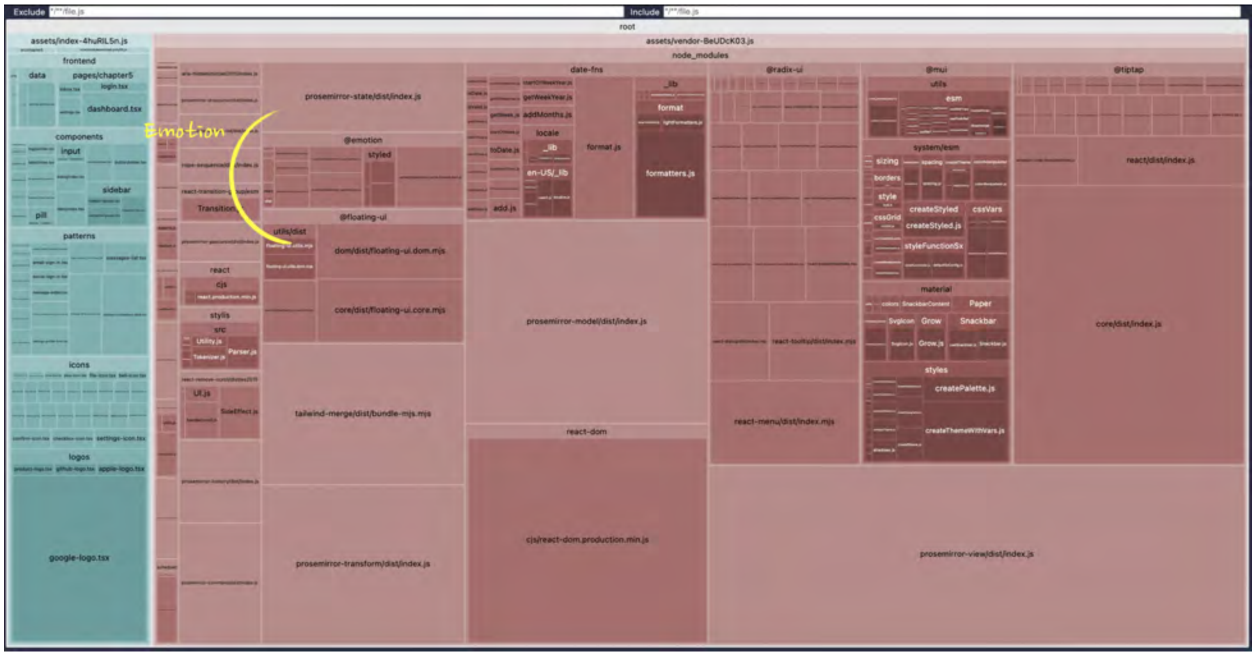
There are visible `@mui` and `date-fns` blocks, which we have shrunk already.

There is a big chunk of `@radix-ui`. These are UI primitives that I use to build the "core" components. I'm not going to touch them now, since I'm definitely not migrating away from Radix in the scope of this project.

There is the `@floating-ui` library - a quick google reveals that this is a library[117] used for positioning dropdowns, tooltips, and other floating elements. Libraries like that are a high risk for duplicates, as happened with Dates. A few more minutes staring at the chart, and I don't see anything visible that could be a library with similar functionality. So this one can stay.

There is also the `tailwind-merge` library. The Study Project uses Tailwind[118] for styling, and this library is essential[119] when dealing with Tailwind, so this one can also stay.

And finally, there is a block of `@emotion`.



While it's relatively small, especially compared to all the prosemirror-related blocks, its presence here raises an eyebrow. Emotion[120] is a CSS-in-JS library, i.e., it's used to style the website instead of dealing with pure CSS. However, we already have Tailwind for this!

If not for the bundle concern, we should remove it just for the sake of reducing the complexity of the project, if we can.

Searching for usages of `@emotion` in the project reveals just one place where it's used:

```
// FILE: frontend/patterns/confirm-delete-dialog.tsx file
import styled from '@emotion/styled';

const Center = styled.div`
  text-align: center;
`;

// inside ConfirmDeleteDialog component
<DialogDescription className="px-8">
  <Center>Are you sure you want to delete this message? You won't be able to recover it.
</Center>
</DialogDescription>;
```

This is a typical case of refactoring going slightly wrong in a large project. Most likely, someone refactored the project to Tailwind in one pull request, and at the same time, someone added this div in another pull request in parallel, and they were merged at the same time. Happens all the time. Now, removing it is our duty. No broken windows[121] should be left broken.

Luckily, it's still easy. We just need to kill the import and the div and add a classname to center the text instead of this component:

```
// FILE: frontend/patterns/confirm-delete-dialog.tsx file

// remove the import and the Center component

// inside ConfirmDeleteDialog component
// remove the Center element and add the new className
<DialogDescription className="px-8 text-center">
  Are you sure you want to delete this message? You won't be able to recover it.
</DialogDescription>
```

Rebuild the project, and...

The `vendor` chunk stays exactly the same size 🤔 What?.. The Emotion package is still there for some reason. Open the `stats.html` file to confirm that.

But why?

Transitive Dependencies

A library can't end up in the bundle by accident. If it's there, it was used somewhere. If it's not used in our project code directly, that means that it was used by some library, which in turn was used in the code. This happens quite often with "foundation" level libraries, i.e., libraries people use to build something on top of them, like positioning libraries and various utils libraries like lodash.

Dependencies like this are called "transitive" dependencies. To identify where a library comes from, we can use npm-why[122] util:

```
npx npm-why @emotion/styled
```

It will give us the list of all the places where the package `@emotion/styled` is used, either directly or indirectly:

```
Who required @emotion/styled:

study-project > @emotion/styled@11.14.0
study-project > @mui/icons-material > @mui/material > @emotion/styled@11.14.0
study-project > @mui/icons-material > @mui/material > @mui/system > @emotion/styled@11.14.0
study-project > @mui/icons-material > @mui/material > @mui/system > @mui/styled-engine >
@emotion/styled@11.14.0
study-project > @mui/material > @emotion/styled@11.14.0
study-project > @mui/material > @mui/system > @emotion/styled@11.14.0
study-project > @mui/material > @mui/system > @mui/styled-engine > @emotion/styled@11.14.0
```

The result, hopefully, is self-explanatory. We use `@emotion/styled` directly in the Study Project, which checks out - we indeed do have it in our `package.json`. And then both `@mui/icons-material` and `@mui/material` use it through a chain of other libraries.

That's a bummer - I thought we could forget about `@mui` when we fixed its import. But it looks like we need to make a hard decision.

Because the only solution here now, if we want to remove the `@emotion` libraries from the bundle, is to remove *everything* that uses them - our direct usage of `@emotion` and usages of both `@mui` libraries.

This instantly escalated the solution from a "quick fix" to a potentially very time-consuming refactoring, especially in real code.

In the Study Project, we can still do it, just to see how far we can push the bundle size reduction. In the real code, it will always be a trade-off between the time spent on refactoring and the potential benefits.

Removing `@mui/material`

First, find where it's used:

```
// FILE: frontend/utils/ui-wrappers.tsx file
import { Snackbar } from '@mui/material';

export const StudyUi = {
  Library: {
    Snackbar: Snackbar,
  },
  Button: Button,
};
```

And delete the import and the usage:

```
import { Button } from '@fe/components/button';

export const StudyUi = {
  Button: Button,
};
```

Then find where the `Snackbar` component is used:

```
// FILE: frontend/patterns/messages-list.tsx file
import { StudyUi } from '@fe/utils/ui-wrappers';

```

```
// Inside MessageList component
<StudyUi.Library.Snackbar open={openSnackbar} onClose={() => setOpenSnackbar(false)}
message="Message deleted!" />;
```

This is a notification[123] component shown when a message is deleted. Since we use Radix for everything else, we can replace this component with Radix's Toast[124] component, which does exactly the same thing. We haven't used this component before, so it might increase our bundle size. However, I hope that the removal of `@mui` and `@emotion` will compensate for this increase. We'll measure the result when we're done with refactoring.

For now, just replace the usage with this:

```
// FILE: frontend/patterns/messages-list.tsx file
import * as Toast from '@radix-ui/react-toast';

// Inside MessageList component
<Toast.Provider swipeDirection="left" duration={3000}>
  <Toast.Root
    className="grid grid-cols-[auto_max-content] bg-blinkNeutral50 items-center gap-x-4
rounded-md bg-white p-4 shadow-
[hsl(206_22%_7%_/_35%)_0px_10px_38px_-10px,_hsl(206_22%_7%_/_20%)_0px_10px_20px_-15px]
[grid-template-areas:_'title_action'_'description_action'] data-[state=open]:animate-slide-
in-left"
    open={openSnackbar}
    onOpenChange={() => {
      setOpenSnackbar(false);
    }}
  >
    <Toast.Title className="text-base font-medium p-2 [grid-area:_title]">Message deleted!
</Toast.Title>
  </Toast.Root>
  <Toast.Viewport className="fixed bottom-4 right-4 z-50 m-0 flex w-[390px] max-w-[100vw]
list-none flex-col gap-2.5 outline-none" />
</Toast.Provider>;
```

Removing `@mui/icons-material`

Find where it's used:

```
// FILE: frontend/icons/index.tsx
import { Star } from "@mui/icons-material";
```

```
export const Icons = {
  Star: Star,
  ... // other icons
};
```

It's a generic "Star" icon that we use to highlight whether a message is in favorites or not. We actually already have a "Star" icon in our collection of local icons, so we can just reuse it instead:

```
// frontend/icons/index.tsx file
import { StarIcon } from "@fe/icons/star-icon";

export const Icons = {
  Star: StarIcon,
  ... // other icons
};
```

You don't even need to find its usage anywhere. That's the beauty of this namespacing pattern in this case - it should just work.

Rebuild the project, start it, and navigate to "Inbox". All messages should now have an "empty" Star icon, that's the new non-MUI one. In a real project, you'd want to replace the old icon with exactly the same one as before, but in our case, it's useful that they are different - at least we see that the changes work.

Hover over any message, click "delete", and click the "confirm" button. The toast component should now appear at the bottom right and be white.

The size of the `vendor` chunk is now **600.98** KB - it went down by ~70 KB! Looks like our refactoring helped.

Finally, open the `stats.html` file - everything related to `@mui` and `@emotion` should disappear, and a new `react-toast` block inside `@radix` should appear.

What's Next?

I hope this was a fun investigation, and you'll now be able to go through your own projects, identify, and quickly fix all the bundle size issues you have. But be sure to return here after you're done with your investigation! We haven't finished talking about bundle size yet - there is a whole issue of all the `@tiptap` and `prosemirror` -related dependencies that need to be dealt with.

So this is what we're going to be looking at in the next chapter. And while doing so, we'll learn about Lazy Loading and start our first look into Suspense and where it can be useful.

8. Intro to Lazy Loading and Suspense

In the previous chapter, we did *a lot* of bundle-size-related stuff:

- We extracted the "vendor" chunk.
- We split the code into chunks per page and per feature.
- We even investigated the content of the bundle, found a few very disturbing dependencies that inflated it in size, and got rid of them.

There is, however, one aspect of the fight with an ever-increasing amount of JavaScript that is still missing. Lazy Loading! Which is impossible to discuss without introducing the concept of `Suspense`.

Knowing these two concepts will lead us to even more advanced topics, almost reimplementing our own routing framework and starting a competitor to Next.js, and eventually to understanding React Server Components.

But all in time. Let's focus on Suspense and Lazy Loading for this part.

Lazy Loading and Code Splitting

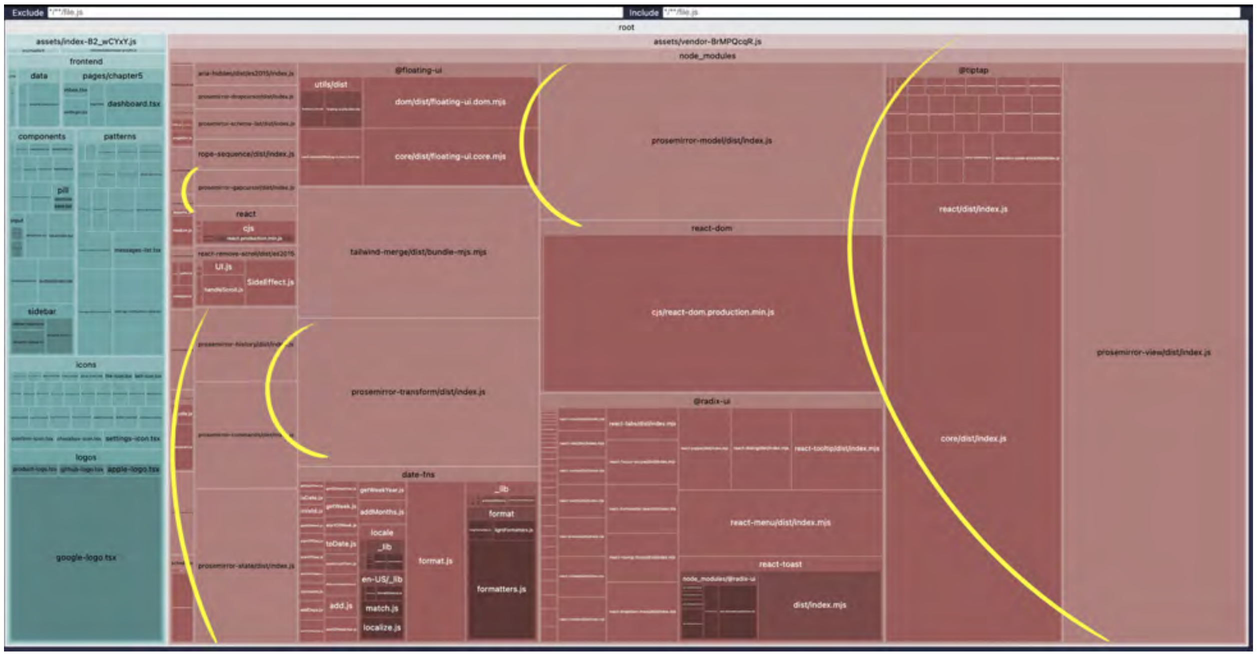
Let's continue the investigation where we left off in the previous chapter. We fixed pretty much everything weird in the bundle, except for a bunch of `prosemirror`-related dependencies. These are like half the vendor chunk now!

If you skipped the work in the previous chapter, you can pick it up from here: `src/chapter8-lazy-loading-intro`. This is the "fixed" code that should be the result of all the manipulations with the dependencies we did.

Let's again build the project and produce its bundle analysis chart:

```
npm run build --workspace=chapter8-simple-frontend
```

The bundle stats file is going to be in `./dist/client/stats.html` and will look like this:



As you can see, `prosemirror` stuff is all around it.

A quick search through the codebase shows that we're not using anything "prosemirror" related directly, so it must be a transitive dependency.

Let's do `npm-why` on any of them to see where they are coming from:

```
npx npm-why prosemirror-model
```

This should give you a huge list with one single origin: `@tiptap/starter-kit`. Searching through the codebase gives us this file: `frontend/patterns/message-editor.tsx`, which implements the `MessageEditor` component.

This component, in turn, is rendered here:

```
// FILE: ./pages/patterns/messages-list.tsx
import { MessageEditor } from '@fe/patterns/message-editor';

// inside MessageList component
{
  clickedMessage ? (
    <MessageEditor
      onClose={() => {
        setClickedMessage(null);
      }}
    />
  ) : null;
}
```

This is our future WYSIWYG-style editor for a message that appears when a message is clicked. Start the project, navigate to "Inbox" and click on any of the messages, if you haven't done it yet - it will appear in a drawer. The editor even works (somewhat)!

So it's clearly something that we can't just remove - it's an essential feature by the look of it.

On the other hand, this feature is not visible during the initial load, especially if we start the initial load from a page other than "Inbox". This combination of "very heavy feature" + "invisible on the initial load" is a clear sign that we can benefit from the pattern known as "lazy loading"[125].

The idea behind "lazy loading" is that we extract this huge feature and all of its associated libraries into its own chunk, away from the `vendor`. Then, load it only after all critical resources have been loaded.

To achieve this in React, we need to do four things:

1. Mark a part of the code as "unnecessary on the initial load".
2. Extract the marked code into its own chunk.
3. Control when exactly the downloading starts.
4. Control what happens when the download is in progress.

Unfortunately, the answer to how exactly to do all of this is slightly dependent on which framework and bundlers you're using. In some cases, all you need to do is use the framework's version of "lazy/dynamic import". Sometimes, React's default "lazy" import will work. Sometimes, you'd need to combine it with the bundlers' configuration changes. Sometimes, you'd need to install a plugin or a library for it to work.

But while the exact implementation might differ slightly, those four steps at their core will remain the same. We'll implement them for the Study Project with a detailed explanation of what's happening. As homework, try to implement it for your own projects using your own tools.

Mark Code as "Lazy"

To mark some components as "not critical", typically we'd need React's lazy[126] utility.

In the case of the Study Project code, we have a heavy `MessageEditor` component that we render when a user clicks on a message:

```
// FILE: ./pages/patterns/messages-list.tsx
import { MessageEditor } from '@fe/patterns/message-editor';

// inside MessageList component
{
  clickedMessage ? (
    <MessageEditor
      onClose={() => {
        setClickedMessage(null);
      }}
    />
  ) : null;
}
```

To mark it as lazy, we'd need to get rid of the direct `import` from the `@fe/patterns/message-editor` and use the `lazy` in combination with a dynamic (i.e., async) `import` instead:

```
// FILE: ./pages/patterns/messages-list.tsx
// Remove the direct import of the MessageEditor component and add a lazy import
import { lazy } from 'react';

// lazy import for a named component
const MessageEditorLazy = lazy(async () => {
  return {
    default: (await import('@fe/patterns/message-editor')).MessageEditor,
  };
});
```

`lazy` returns us a component that can be rendered anywhere as any other component. The difference is that the dynamic import (and the download of the code as a result) will be triggered **only when the "lazy" component is mounted.**

Let's mount it and see how it works. First, find where the not-lazy component is used:

```
// inside MessageList component
{
  clickedMessage ? (
    <MessageEditor
      onClose={() => {
        setClickedMessage(null);
      }}
    />
  ) : null;
}
```

And replace the `MessageEditor` with `MessageEditorLazy`. That's literally it.

Overall, these are all the changes we need to make now to turn a component into "lazy":

```
// FILE: ./pages/patterns/messages-list.tsx
// Remove the direct import of the MessageEditor component and add a lazy import
import { lazy } from 'react';

// lazy import for a named component
const MessageEditorLazy = lazy(async () => {
  return {
    default: (await import('@fe/patterns/message-editor')).MessageEditor,
  };
});

// inside MessageList component - replace MessageEditor with MessageEditorLazy
{
  clickedMessage ? (
    <MessageEditorLazy
      onClose={() => {
        setClickedMessage(null);
      }}
    />
  ) : null;
}
```

Although it's not going to work as expected yet, we haven't completed all the steps.

"Lazy" Chunk

In the second step, we need to extract the "lazy" component into its own chunk to free our `vendor` from all its weight. This step will heavily depend on the bundler or framework you're using. Sometimes, you won't have to do anything. The splitting will happen automatically.

The Study Project is on Vite[127], and by default, it supports "lazy" loading.

Let's see what will happen if we just build the project without any configuration changes:

```
npm run build --workspace=chapter8-simple-frontend
```

It indeed produced one extra chunk named `message-editor-bla.js`. However, the size of this chunk is tiny, and the `vendor` chunk hasn't changed at all:

```
dist/client/assets/message-editor-DEpy3GEC.js    4.86 kB │ gzip:   1.23 kB
dist/client/assets/vendor-BbjbBVYU.js          600.97 kB │ gzip: 186.55 kB
```

If you open the `stats.html` file, you'll see that this tiny file only has two files from our project. Everything related to editor libraries is still in the `vendor` chunk.

If we look at the `vite.config.ts` configuration we implemented earlier, you'll see that we still have this code:

```
manualChunks: (id) => {
  if (id.includes("node_modules")) {
    return "vendor";
  }

  return null;
},
```

Looks like this rule takes priority over the lazy loading patterns in Vite. So it's time we get creative with those rules again. For example, we can do this:

```
manualChunks: (id) => {
  if (
    id.includes("node_modules/@tiptap") ||
    id.includes("node_modules/prosemirror")
  ) {
    return "editor-vendor";
  }
  if (id.includes("node_modules")) {
    return "vendor";
  }

  return null;
},
```

This will extract editor-related heavy libraries into their own `editor-vendor` chunk. Rebuild the project again, and the result will be this:

```
dist/assets/message-editor-wugmzKv-.js    4.90 kB │ gzip:  1.24 kB
dist/assets/editor-vendor--ExOB3X7.js   295.78 kB │ gzip: 89.41 kB
dist/assets/vendor-DiJDHLYi.js          304.56 kB │ gzip: 96.90 kB
```

This is more like it - the `vendor` chunk has been halved in size, and the new `editor-vendor` appeared. Open the `stats.html` file to confirm that all those libraries indeed moved their location. The `vendor` chunk is now as clean as it can possibly be for this project, with only the necessary libraries left there.

But remember that at the very beginning of the bundle size investigation in the previous chapter, we discovered that all the chunks were injected into `index.html` and, as a result, preloaded during the initial render? I would assume this wouldn't happen for the "lazy" chunks. Otherwise, what's the point of them? But it never hurts to check. Open `./dist/client/index.html` and take a look at which JavaScript files are loaded there.

Only `vendor` and `index` files should be present. This is more great news: it looks like our bundle is smart enough to realize that the `editor-vendor` chunk, although declared manually in the configuration, originated from a "lazy" chunk, and its loading can be postponed.

This verification step is important to do if you're messing with any bundler configuration manually.

Additional Challenge.
If the `message-editor` and `editor-vendor` chunks don't appear in `index.html`, how exactly are they downloaded then?
How to investigate:

- Disable minification in `vite.config.ts` by adding `minify: false` to the `build` setting, to make the produced code readable. Rebuild the project.
- Search for `message-editor` and `editor-vendor` inside the built files - you'll see which one is referenced where and how.
- After following the breadcrumbs, you should see that the bundler injected a piece of code into one of the files. This code creates a `script` tag dynamically and injects it into `document.head` via `appendChild`.

Control the Start of Download

The next step in the lazy loading process is to control when the download starts exactly. As we already know, with the use of lazy[128] util, it will happen when the "lazy" component is mounted. So, essentially, this step is all about knowing when the "lazy" component is mounted and what happens afterward.

To investigate this, let's peek into the "Network" section of the performance profile.

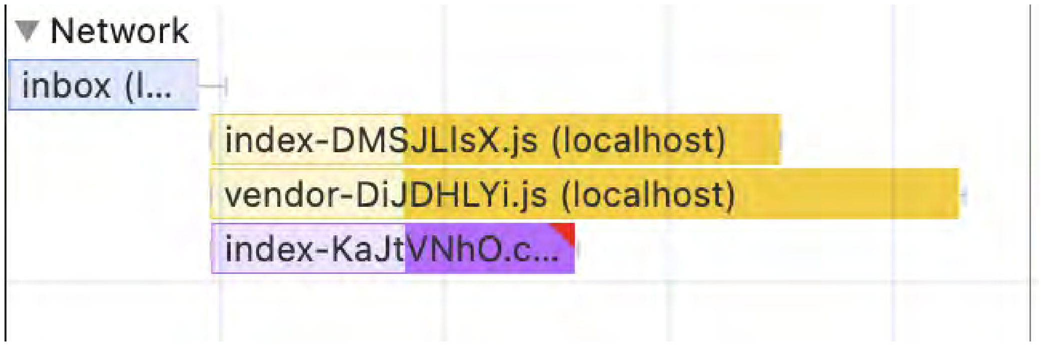
Build and start the project:

```
npm run build --workspace=chapter8-simple-frontend
npm run start --workspace=chapter8-simple-frontend
```

Navigate to the "Index" page and record the performance of the initial load with:

- 6x CPU throttling.
- Slow 4G of the network throttling.
- Checked the "disable network cache" checkbox.

If you disabled minification in the previous step, enable it again. In the "Network" section, you should see only two JavaScript bars: index and vendor chunks being loaded in parallel.



There are no "lazy" chunks.

Now open the "Network" tab instead of "Performance" and click on any of the messages. The interface will fall apart, but that's okay - it's because we haven't completed the last step yet. The important thing here is that only now did the `editor-vendor` and the `message-editor` show up.

This is happening because we render our lazy component conditionally:

```
// FILE: ./pages/patterns/messages-list.tsx

// inside MessageList component
{
  clickedMessage ? (
    <MessageEditorLazy
      onClose={() => {
        setClickedMessage(null);
      }}
    />
  ) : null;
}
```

Where `clickedMessage` is a state variable. By default, this variable was `false`. `MessageEditorLazy` was never mounted during the initial load. As a result, the lazy code was never loaded.

After we clicked on a message, we moved that variable to a "truthy" value:

```
<div
  className="flex flex-col w-full cursor-pointer"
  tabIndex={1}
  onClick={() => setClickedMessage(message.id)}
>
```

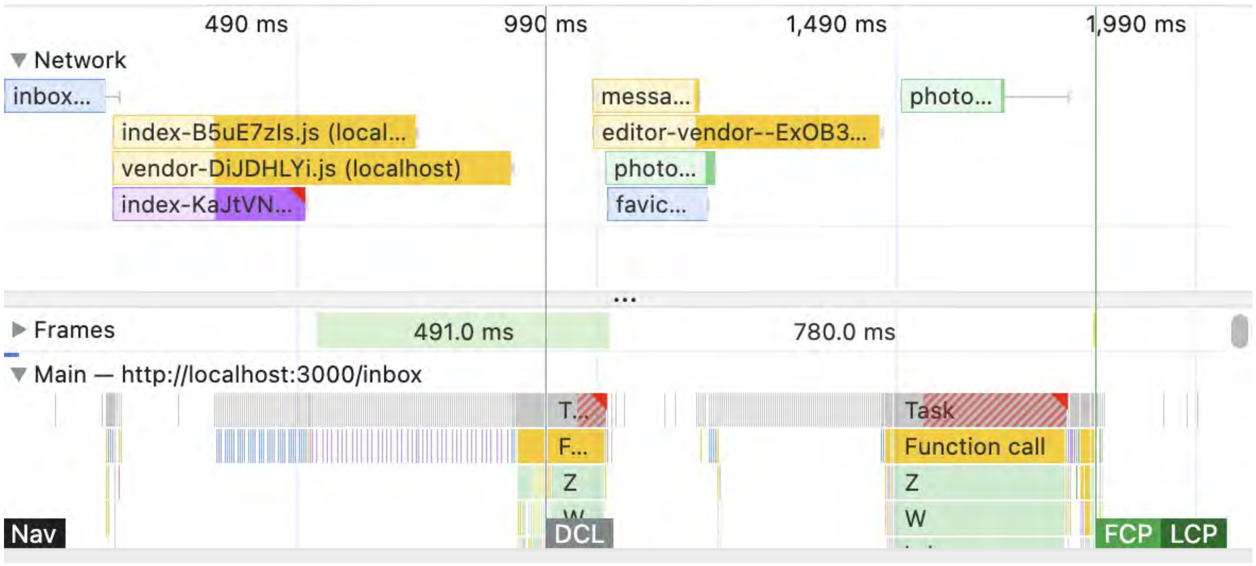
The `MessageEditorLazy` component was mounted for the first time, and the download of all the files in the associated chunk was triggered.

```
// The chunk associated with this lazy load started downloading
// when the MessageEditorLazy was mounted
const MessageEditorLazy = lazy(async () => {
  return {
    default: (await import('@fe/patterns/message-editor')).MessageEditor,
  };
});
```

Now, what will happen if we remove the conditional rendering but keep the editor "lazy"?

```
// FILE: ./pages/patterns/messages-list.tsx

// inside MessageList component
// remove the variable and the condition, make this component render as others
<MessageEditorLazy
  onClose={() => {
    setClickedMessage(null);
  }}
/>
```

Make the change, rebuild and restart the project, and record the initial load performance again. The picture now should be this:



The browser first downloads the critical path JavaScript - our `index` and `vendor` chunks. Then it processes them and triggers some React stuff - that's when we mount the components, including the now non-conditional `MessageEditorLazy`. That triggers the download of the "lazy" chunks. If you click on any of those yellow bars of the "lazy" chunks, you'll see an arrow that shows the chunk of origin, which is the `index` chunk. In the "Summary" at the bottom, you'll also see the "Initiated by" field, which also points to the index chunk.

While this download is happening, the browser simply waits and shows nothing. After everything is downloaded, the browser finally finishes the JavaScript, calculates the layout, and paints it on the screen, including the open drawer. Only after that is the LCP metric triggered, with a value of around **1.8 s**.

With this "semi-lazy" loading, we essentially made the initial loading slightly worse. Instead of fetching all this JavaScript in parallel, we're doing it in sequence in the worst possible way: while showing absolutely nothing to the users in the process.

We can actually measure how much worse it is. All we need is to change `MessageEditorLazy` back to the direct import:

```
// FILE: ./pages/patterns/messages-list.tsx
import { MessageEditor } from '@fe/patterns/message-editor';

// comment out the lazy chunk
// const MessageEditorLazy = lazy(async () => {
```

```
//   return {
//     default: (await import("@fe/patterns/message-editor")).MessageEditor,
//   };
// });

// inside MessageList component
<MessageEditor
  onClose={() => {
    setClickedMessage(null);
  }}
/>;
```

As always, rebuild and restart the project, and measure the initial load.

We'll still have the `editor-vendor` chunks since the `vite.config.ts` stays the same. But this time, you'll see that all three chunks are fetched in parallel at the very beginning, JavaScript is no longer sequenced, and the LCP drops to **1.6 s.**

Instead of improving performance, the incorrectly implemented lazy-loading cost us 200 milliseconds.

This is why there are four steps here. We need to control what happens when the "lazy" download is in process. For that, we need to use Suspense[129]. But before that, we need to understand what it is and how it works.

Intro to Suspense

Suspense is a special component provided to us by React itself. It can detect whether some of the elements passed to it as `children` are in the process of lazy-loading, and if yes - it can "suspend" the entire tree. "Suspended" children will be marked by React as "not critical, deal with them later". React then will skip the entire sub-tree, render a fallback component passed to `Suspense`, and focus on rendering everything else that is not "suspended". Lazy-loaded components will be rendered when they are fully downloaded.

For example, let's say you have a `Button` component:

```
const Button = () => {
  return <button>Button</button>;
};
```

And then some `Parent` component that renders that button:

```
const Parent = () => {
  return (
    <>
      <h1>Welcome!</h1>
      <Button />
    </>
  );
};
```

If you just render it like this on the screen, you'll see the "Welcome!" string followed by the "Button" string. Just a regular React component hierarchy.

If you wrap the `Button` in `Suspense`:

```
const Parent = () => {
  return (
    <>
      <h1>Welcome!</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <Button />
      </Suspense>
    </>
  );
};
```

Nothing actually will change. The `Button` here is not lazy-loaded. The `Suspense` component detects this and renders the `Button` right away.

If, however, I wrap the `Button` in `lazy` like this:

```
const LazyButton = lazy(async () => {
  return {
    default: Button,
  };
});
```

And then render it in place of the normal button:

```
const Parent = () => {
  return (
    <>
      <h1>Welcome!</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyButton />
      </Suspense>
    </>
  );
};
```

The situation will change. The "Welcome!" string and the "Loading..." fallback will be rendered first, and then "Loading..." will be replaced by the "Button". If you implement this code and refresh the page, you'll see a momentary flash of content. Slow down the CPU 20x, refresh the page again, and you'll clearly see the sequence.

By the way, look closely at how I used `lazy`:

```
const LazyButton = lazy(async () => {
  return {
    default: Button,
  };
});
```

And read the description[130]. Lazy accepts a "load" function as an argument, which should be a Promise. When the Promise is resolved, React renders the `default` part of the returned value. That's why the code above looks slightly weird - it's just mimicking the "load" function.

This knowledge, other than being a cool fact, gives us an opportunity to mimic actual lazy loading even more. Since it's just a regular promise[131], nothing stops us from introducing a small delay before the return:

```
const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

const LazyButton = lazy(async () => {
  // let's wait a bit
  await sleep(3000);
  return {
    default: Button,
  };
});
```

Add this code, and you won't need to slow down the CPU anymore to see the Suspense fallback while the button is "loading".

Everything that goes inside `Suspense` will be "suspended" while the lazy code is loading. I.e. while we're waiting for the promise to resolve. For example, try to introduce something before the button like this:

```
const Parent = () => {
  return (
    <>
      <h1>Welcome!</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <>
          <h3>This is suspended</h3>
          <LazyButton />
        </>
      </Suspense>
    </>
  );
};
```

Both the `h3` tag and the button will be hidden under "Loading" until the promise inside `lazy` resolves itself.

Applying Suspense in the Study Project

Now that we know how Suspense works, it's time to finish with the code-splitting task and finally fix the bundles.

We already have the lazy-loaded `MessageEditorLazy` component. All we need to do is wrap it in `Suspense`:

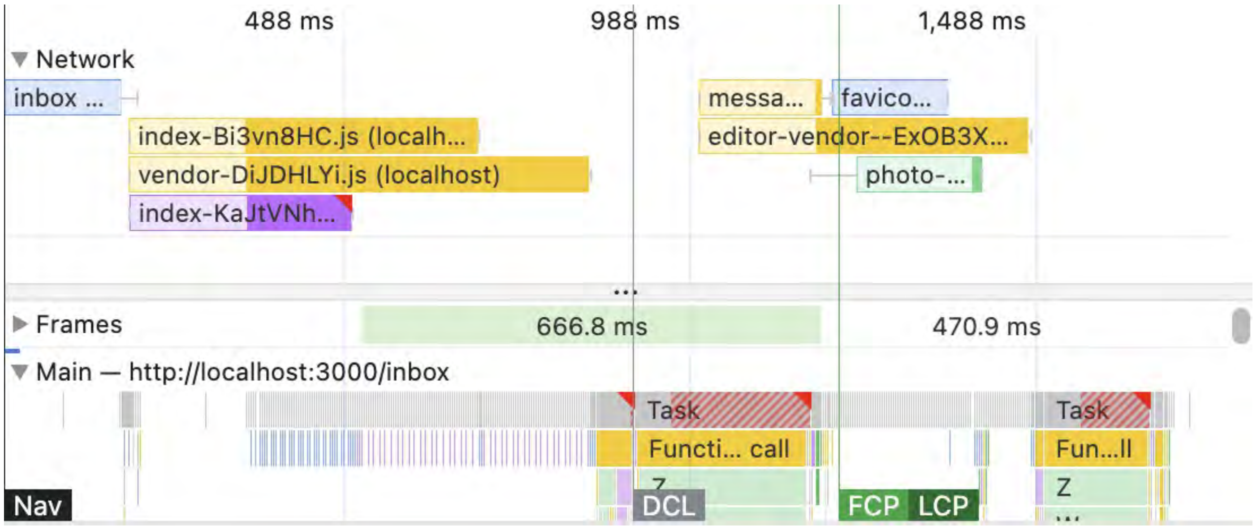
```
// FILE: ./pages/patterns/messages-list.tsx
// remove the direct MessageEditor import
```

```
import { lazy, Suspense } from 'react';

// uncomment the lazy chunk
const MessageEditorLazy = lazy(async () => {
  return {
    default: (await import('@fe/patterns/message-editor')).MessageEditor,
  };
});

// inside MessageList component wrap lazy editor in Suspense
<Suspense>
  <MessageEditorLazy
    onClose={() => {
      setClickedMessage(null);
    }}
  />
</Suspense>;
```

Keep the `MessageEditorLazy` as a permanently opened drawer again to compare the result in the UI with and without Suspense. Rebuild the project and restart it.

This time, you should see how the UI is built in "stages". First, the list of messages will appear: React renders the full page first since it's not "suspended" in any way. Only after React finishes with the critical tasks will the "lazy" drawer appear. Compare it with the rendering before introducing Suspense, where everything was considered "critical", and the initial render of the page was slower since it was waiting for the downloading and rendering of the drawer.

If you record the performance profile again with Suspense enabled, you should see this picture now:



`index` and `vendor` JavaScript in the Network section first - they are downloading. Then, they are processed, and React execution kicks in. In the middle of this, the "lazy" chunks started downloading when the `MessageEditorLazy` is mounted. But React ***does not wait*** for them. It finishes rendering the critical path non-suspended components.

This is when the list of messages shows up on the screen, and the LCP is triggered. Download of the "lazy" JavaScript above is still in progress. The LCP value is **1.2 s** now, 400 milliseconds lower than what we had before. The page is perfectly interactive at this point. If you slow down the Network to 3G, you can see for yourself - as soon as the list of messages appears on the screen, you'll be able to open/close menus and turn the toggle at the top on and off.

While all of this is happening, the "lazy" download continues slowly, and the MessageEditorLazy component is still "suspended". Only after this download finishes does React take over again, render the component that was "suspended", and inject it into the page. This is when the open drawer finally appears.

Suspense with Fallback

With the permanently open drawer, we didn't really need the fallback and "loading" state. However, with conditional rendering, it's pretty much mandatory.

Let's restore conditional rendering now:

```
// FILE: ./pages/patterns/messages-list.tsx
// remove the direct MessageEditor import
import { lazy, Suspense } from 'react';

// uncomment the lazy chunk
const MessageEditorLazy = lazy(async () => {
  return {
    default: (await import('@fe/patterns/message-editor')).MessageEditor,
  };
});

// restore the conditional rendering
{
  clickedMessage ? (
    <Suspense>
      <MessageEditorLazy
        onClose={() => {
          setClickedMessage(null);
        }}
      />
    </Suspense>
  ) : null;
}
```

Now, the "lazy" component is mounted only when the state variable is "truthy", i.e., when we click on a message. Which makes the user experience terrible the first time they click.

The lazy code download will only kick in when the `MessageEditorLazy` mounts. The drawer will appear *only* after the download finishes. Until this moment, the user will just stare in confusion at the screen, with no visible reaction to their action. For them, this would look like the UI is broken.

This is why it's vitally important to do at least *something* right away. And this is where the `fallback` prop is crucial. If we were experimenting with some inline always-open functionality, like a heavy "comments" section, we could show a spinner or some nice shimmering animation where the content is supposed to be injected.

In our case, since the lazy component is a drawer, we can show an "overlay" div that grays out the content.

```
<Suspense fallback={<div className="w-full h-full fixed top-0 left-0 opacity-50 bg-
blinkNeutral300 z-50"></div>}>
```

```
  <MessageEditorLazy
    onClose={() => {
      setClickedMessage(null);
    }}
  />
</Suspense>
```

Rebuild, restart the project, and click on any message now. You'll see that immediately after the click, the overlay div shows up and blocks the page, preventing you from doing anything, and also indicating that something is happening. Soon after, the drawer shows up.

In the Network tab, you'll see that the "lazy" chunks behave properly now. During the initial load, they are not visible and appear only after the first click.

All consecutive clicks on other messages will show the drawer immediately - we already have all the associated JavaScript downloaded and cached by Suspense.

What's Next

There is more to lazy loading and Suspense, the above is just an introduction. We also need to learn chunks preloading and why it's important, how Suspense works with SSR, and how it's compatible with data fetching. We'll talk about all of this in the next chapter.

For now, if you want to practice different combinations of lazy loading and see how it affects user experience and performance, you can try to solve the challenges below. Those will prepare you better for the next chapter as well.

Challenge 1: Heavy Hidden Widget

Start the Study Project and navigate to any of the pages. At the top right corner, you'll see the "Add widget" button. Right now, it's just a button, but in the future, it will likely open a modal dialog-like functionality, where people will be able to configure their own widgets. This future modal dialog is a perfect candidate for lazy loading!

1. Find the "Add widget" button in the code and refactor it to open a modal dialog

when clicked. Use the Dialog from `frontend/components/dialog` for this.
2. Pull some heavy component into this dialog to make it feel realistically "heavy".
3. Lazy-load the dialog with a fallback, the same way we lazy-loaded the drawer before.
4. Refactor the code to make the button lazy-loaded with the dialog. So that during the initial page load, the button is not rendered, but appears after the necessary code is downloaded, and clicking on the modal doesn't trigger the fallback of Suspense.

What kind of pros and cons do those two patterns have?

Challenge 2: Router-Based Lazy Loading

Currently, in the Study Project, all the pages are bundled together in the `index` chunk. But do we really need to load all the JavaScript from the "Inbox" or "Settings" page if we're loading the "Home" page? In real projects, there will be much more code in all of them.

So let's split and lazy-load code per page! Modern frameworks like Next.js or React Router do that kind of thing by default for us, but this exercise could be useful to understand what exactly they are doing and what else is missing in our rudimentary solution.

1. Navigate to the Study Project's `App.tsx` file - this is our rudimentary "routing" solution.
2. Make all pages "lazy-loaded" here with some fallback.
3. Measure the initial load performance - it should decline. Why do you think this happened? What can be done to improve it? Are there situations that you can think of in which this "naive" split per route could improve performance?
4. Try to navigate to a different page - you should see that the Sidebar disappears when you do that. Why do you think this happens now? What can be done to improve it?

Challenge 3: Going to the Real World

Let's take our new knowledge to the real world now. Open a few of your favorite websites and record initial load performance for them. Analyze their JavaScript loading strategy:

1. What kind of chunking strategy are they using, if any?
2. Do they use lazy loading for some parts of the website? If yes, what is their lazy loading strategy?
3. If you were hired as their performance expert tasked with improving their initial load performance, what would you start investigating first in their JavaScript situation? What seems to be a low-hanging fruit to fix on your first day?

9. Advanced Lazy Loading

In the previous chapter, we learned about Lazy Loading, how to enable and control it, and where the Suspense component plays a role.

But Lazy Loading is not only about the delayed loading of modal dialogs and drawers. It's a very powerful tool that allows us to control precisely what the user sees first and squeeze every millisecond from the initial load performance. Or not. Or sometimes.

This chapter is when the performance talk gets real, truly fuzzy, and totally dependent on the environments, users, and sometimes phases of the moon.

Initial Project Setup

As usual, we'll need a semi-real Study Project. You can find it in `src/chapter9-advanced-lazy-loading/simple-frontend` of the Study Project repository[132]. The code here is the same as the "cleaned up" version in the previous chapter, with minor refactoring in the layout area and a different Vite config.

As always, build and start it like this:

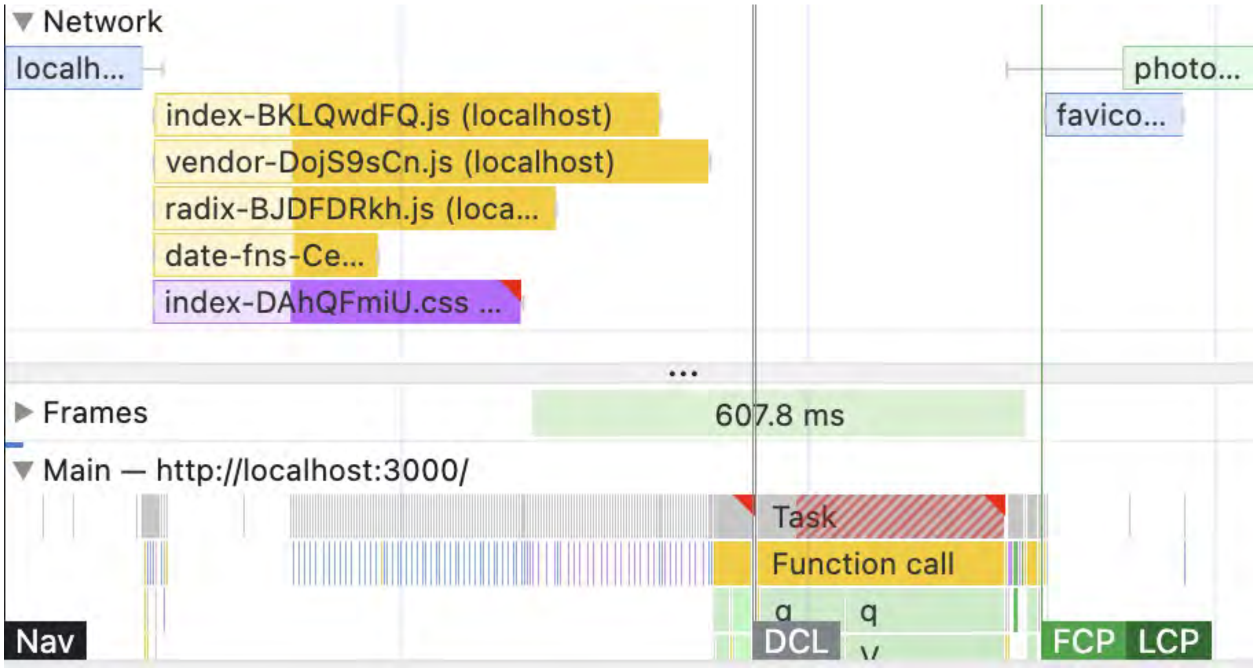
```
npm run build --workspace=chapter9-simple-frontend
npm run start --workspace=chapter9-simple-frontend
```

In the `dist` folder, you'll see multiple JavaScript chunks: I split `vendor` into a few more to reduce the size even further. You can see how it's configured in `vite.config.ts`.

If you open the project in the browser, you'll see an already familiar app: Sidebar navigation on the left with Home, Inbox, and Settings pages working. Plus, clicking on the logo will navigate you to the Login page, and clicking on the title will navigate you back.

For all measurements, unless otherwise specified, I'm going to use **"Fast 4G"** in the Network setting, **"CPU: 6x slowdown"**, and the **"Disable cache"** checkbox enabled.

The Home page performance should already be a familiar Client-Side Rendering picture:



Network
localh...
index-BKLQwdFQ.js (localhost)
vendor-DojS9sCn.js (localhost)
radix-BJDFDRkh.js (loca...
date-fns-Ce...
index-DAhQFmiU.css ...
photo...
favico...
Frames
607.8 ms
Main — http://localhost:3000/
Task
Function call
q
q
Nav
DCL
FCP
LCP

A few initial chunks are downloaded in parallel, then JavaScript is executed, followed by FCP/LCP metrics at the same time. The LCP time is **915 ms.**

Let's see what we can squeeze out of it.

Manual Code Splitting per Route

If you open the `stats.html` file and peek into the content of all the chunks, you'll see that literally *everything* that I coded in the project is bundled in the `index` file. Everything else is coming from `node_modules`, i.e., it's an external dependency.

On an intuitive level, that doesn't seem right. Why would I have the content of the "Login" or "Inbox" page loaded when I open the Home page? Wouldn't it slow down the initial render? We already did some serious code-splitting with those vendor chunks.

Why not continue on this path and slice that JavaScript even more?

We surely can, but this is when it gets really tricky and environment/people/phases of the moon dependent.

The very first thing that comes to mind is to continue with slicing code in the build config file. We already wrote plenty of code like:

```
if (id.includes('node_modules')) {
  return 'vendor';
}
```

We could also add conditions for every page like this:

```
if (id.includes('dashboard')) {
  return 'dashboard';
}
if (id.includes('inbox')) {
  return 'inbox';
}
if (id.includes('login')) {
  return 'login';
}
if (id.includes('settings')) {
  return 'settings';
}
```

It doesn't look too pretty, but it's easy to add, requires no code changes, easy to read, understand, and modify. In the future, if the app grows even bigger, we could probably just generate it.

Try adding exactly that to `vite.config.ts` and rebuilding the project. You'll see that in addition to the existing vendor chunks, it now produces a chunk for each page, and they even seem reasonable sizes, and combined are roughly the size of the `index` file before the split (plus a bit of overhead), so it clearly worked as expected:

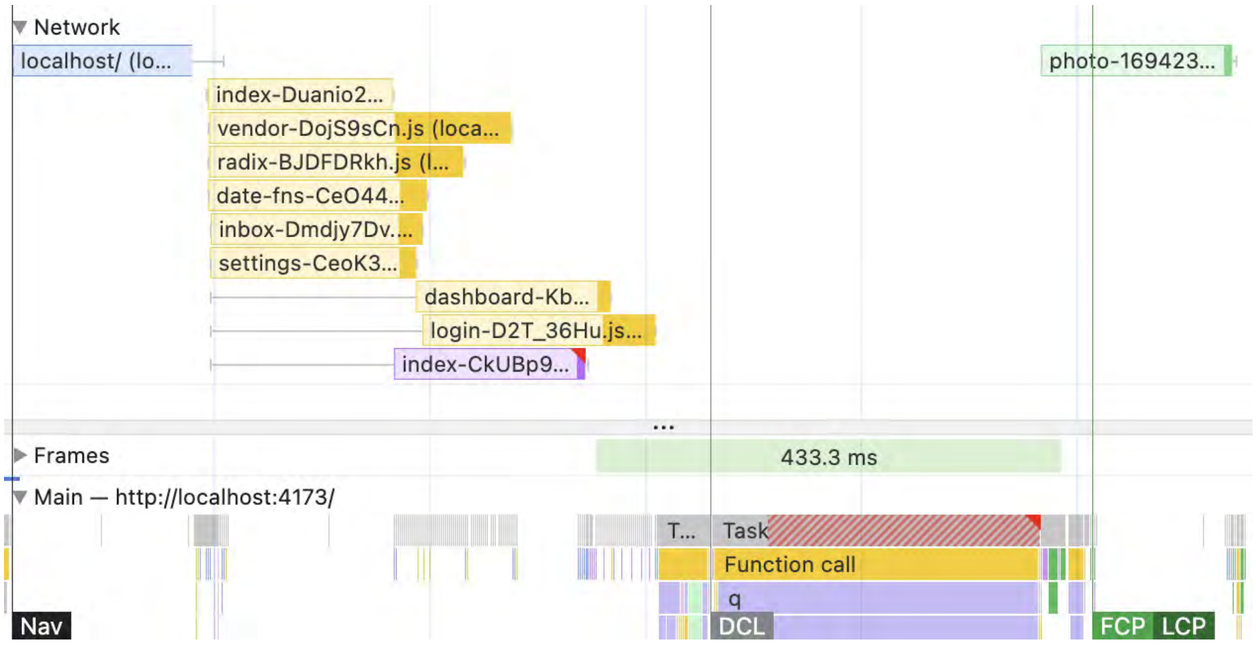
```
dist/assets/index-B22iro7S.js                    1.23 kB │ gzip:  0.66 kB
dist/assets/settings-CeoK36Hz.js                14.55 kB │ gzip:  3.99 kB
dist/assets/inbox-Dmdjy7Dv.js                   15.39 kB │ gzip:  5.14 kB
dist/assets/dashboard-Kb80EKUn.js               38.66 kB │ gzip:  8.14 kB
```

```
dist/assets/login-D2T_36Hu.js                    75.80 kB │ gzip: 44.79 kB
.. // the rest of the chunks
```

So it would be reasonable to expect some improvement in the LCP, since a large part of it is the JavaScript download time.

Record the performance profile again to see that this change made the LCP **worse**. It's **992 ms** now instead of 915 ms. That's slightly counterintuitive 😳. What happened? Can you investigate and find out before looking into the answer?

This is the picture that you should see in your recording:



Yep, there are more chunks in parallel. But also, there are chained chunks now! So, overall, it takes longer.

If you remember, in Chapter 7, we covered why: in Chrome, there is a connection limit for the HTTP/1 protocol. Everything that is outside the limit will have to wait for its turn, as it happens here. So, in production, which is hopefully HTTP/2 or 3, those requests should be parallel, and this won't be a problem.

However, it's unlikely that LCP will improve anyway, even if all of them were in parallel. Take a closer look at which chunk is the *longest* one here - this one will determine the overall wait time for all the JavaScript. This one is the **bottleneck**. And it's the `vendor` chunk, which hasn't changed from our splitting efforts this time.

So we probably did all this work for nothing.

But did we? 🤔

What we're measuring here is just one quite specific use case: the **website's first-time visitors**. So if the majority of the visitors open your website only once in the website's lifetime, then the work was wasted indeed, and you'd need to come up with something else to improve the LCP.

And although I say the use case is specific, it doesn't mean that it's not popular. Pretty much every standalone landing page or promo page fits this description. You open it once, and then either buy something from it ("convert" in marketing terms) or not. Either way, you'll likely never return to this page again. In this case, all effort should be focused on splitting and trimming the bottleneck chunk, the `vendor`.

However, on a website with a high return ratio, like any SaaS platform, **returned visitors** might take priority, and the question of whether the website is updated between visits plays a crucial role. Because now it's all about how many of those chunks are stored in the **browser's cache**.

If the website is almost never updated, then all the chunks will be cached, the network won't have any effect, and splitting any of them will be pretty much pointless. You can simulate this scenario by unchecking the "Disable network cache" checkbox. Uncheck it, refresh the page, and then record the LCP number. Then rebuild the project without the code splitting (disable all of it in the config), refresh the page, and record the performance again. The LCP number should be exactly the same since the JavaScript download plays no part here.

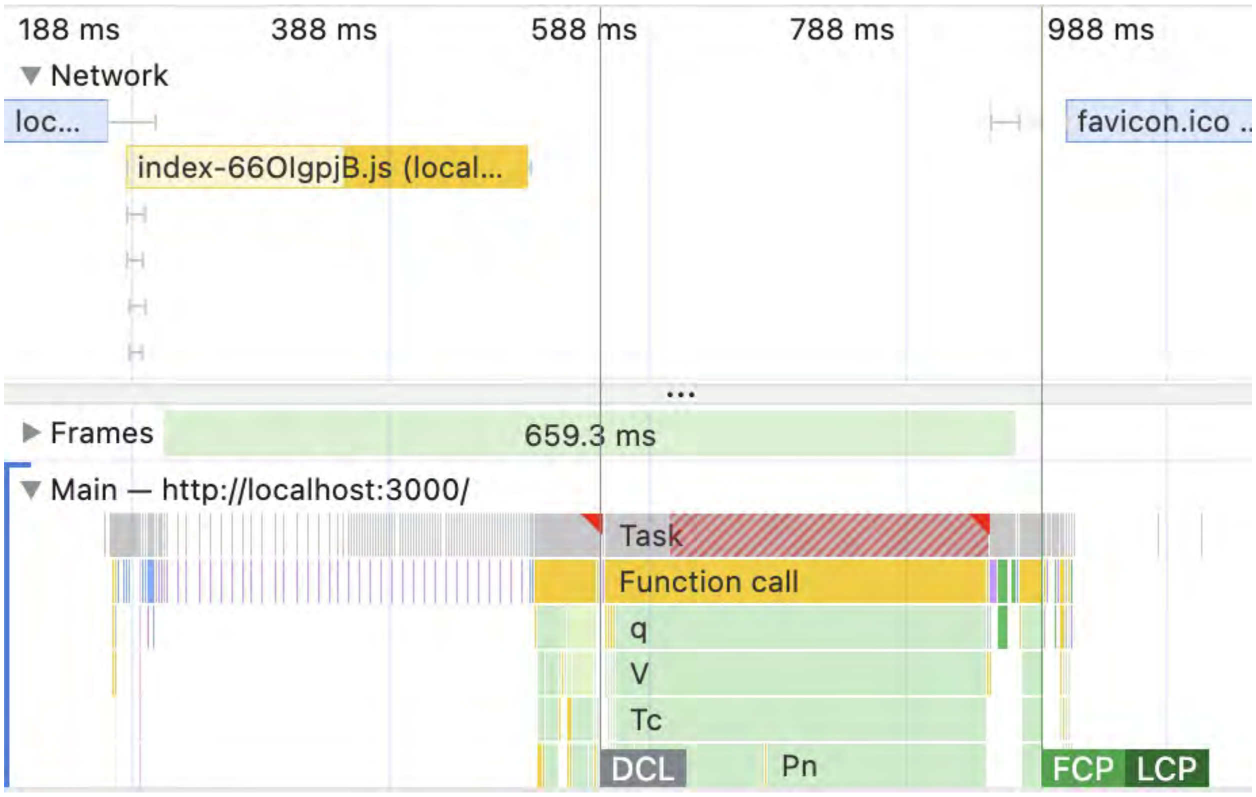
However, in real life, it's highly unlikely that we'd care about the performance of a website that is never updated. The website is probably under constant development, with deployments at least a few times a week, maybe even a day.

To emulate this, keep the "Disable network cache" checkbox *disabled* and then do this:

1. Build the project *without* the manual chunks we introduced here.

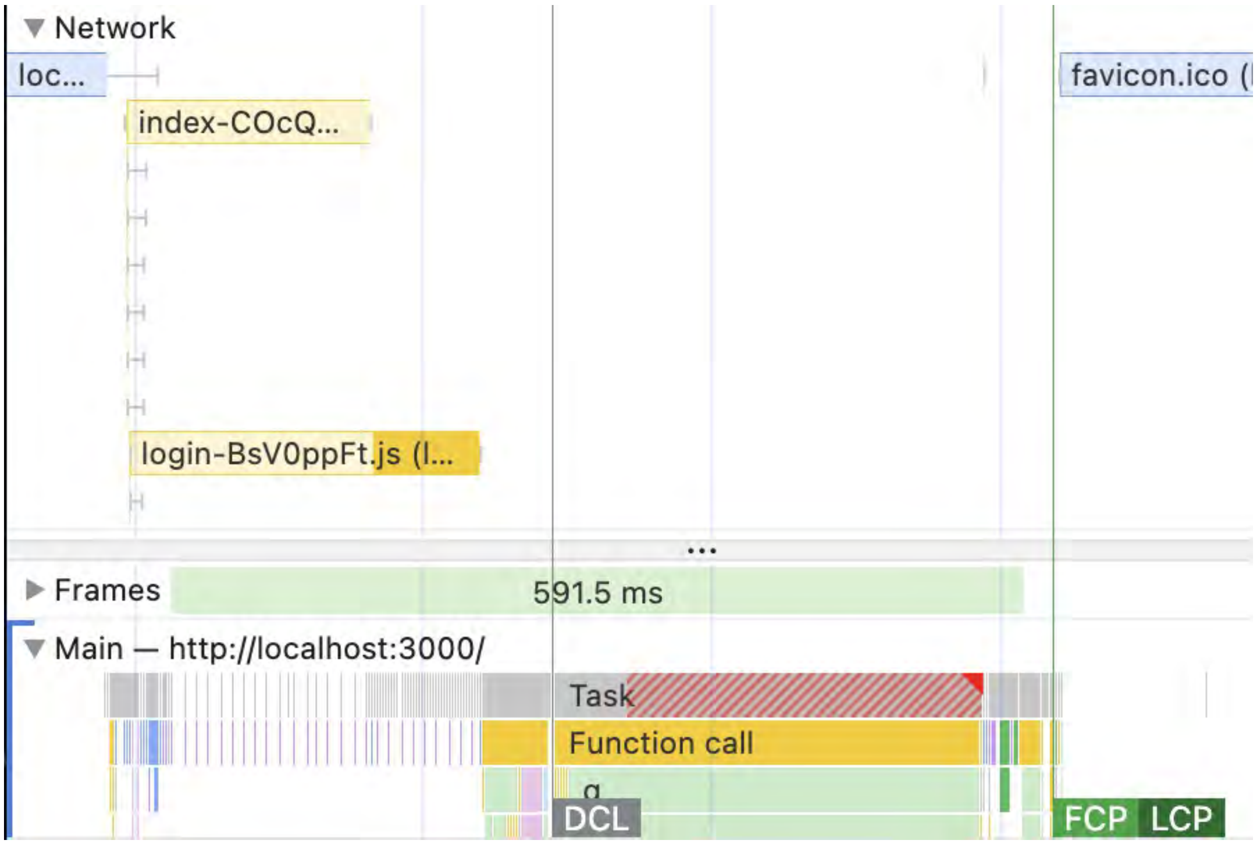
2. Start the website and refresh the page to allow the browser to cache all the available files.
3. Make some changes in `./pages/login.tsx`, that's our Login page. Even adding a `console.log` would do.
4. Rebuild the project without closing the browser tab with the open website.
5. Record performance (with the reload button).

The picture should be this:



All the "vendor" chunks haven't changed and were previously cached by the browser. In the "Network" tab, the `index` chunk that contains the entire project, including the Login page, is the biggest contributor now. The LCP value is **640 ms**.

In this case, splitting the `index` chunk like we did before should, in theory, produce some visible results. Restore per-page manual chunking, follow exactly the same steps, and record the final performance. The picture now is this:



Most of the JavaScript chunks are cached, and their download is skipped. We're now downloading only the tiny `index` chunk that has a reference to the Login page, and the `login` chunk itself - this is where we made the change.

The LCP value is **620 ms** now. A 20 ms difference. Hmmm. It's not impressive at all, but at least slightly visible. I'll take that, considering that we got it pretty much for free. Although it's highly dependent on which code changes and which page we're loading. Try to make changes to the "Settings" page instead of the "Login" - you probably won't see any gains at all in this case. Try to figure out why, by the way, if you want a challenge.

Surely, we can do better than that. Let's put all the numbers in a table for comparison and try a different approach.

	Simulation	Home page LCP
1	First-time visitors, non-split `index` file (baseline)	915 ms
2	First-time visitors, manual chunks via config	992 ms/915 ms

3	Repeated visitors with changes in Login, non-split `index` file	640 ms
4	Repeated visitors with changes in Login, manual chunks via config	620 ms

Lazy-loading per Route

In the previous section, we investigated what would happen if we split that index chunk into per-page chunks manually. If you did the exercises above, restore the project to its initial state, with just one `index` chunk that has everything and the LCP value at **915 ms.** In this section, we'll try to do exactly the same thing, only instead of manual chunks, we'll lazy-load them.

To do that, we only need to change the code in `App.tsx`. If you remember the previous chapter, we'd need to:

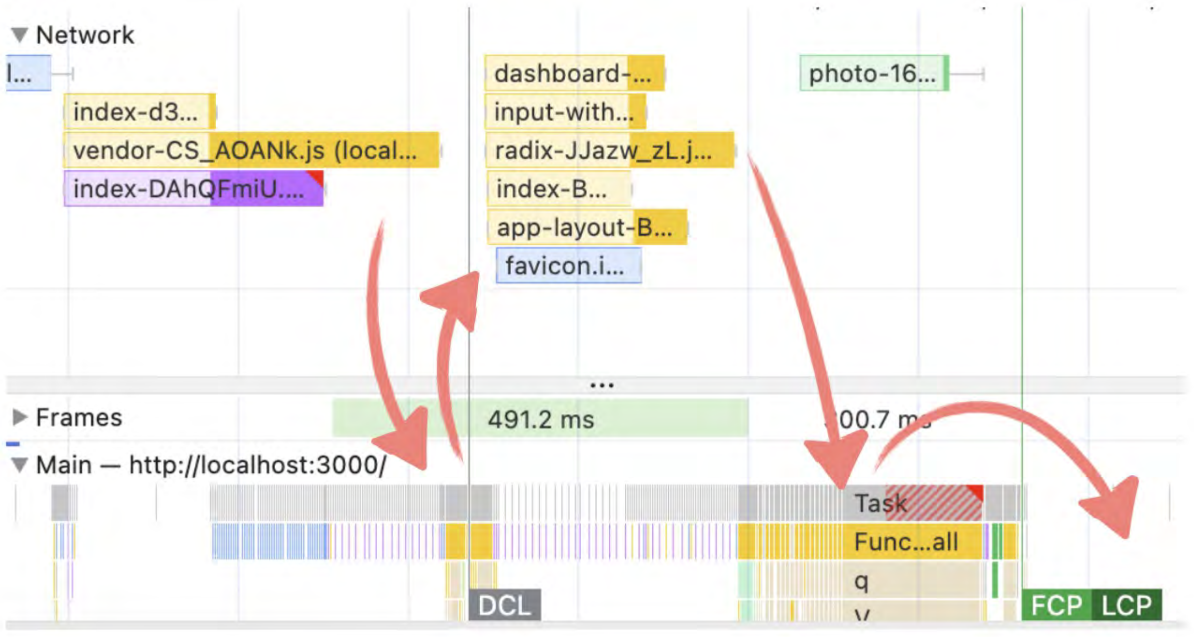
- Replace direct imports with "lazy" imports.
- Replace the components with "lazy" components.
- Wrap the "lazy" components in `Suspense`.

For each page in the `App.tsx` file, do this:

```
// replace direct import with lazy import
const DashboardPageLazy = lazy(async () => {
  return {
    default: (await import('./pages/dashboard')).DashboardPage,
  };
});

// inside App component, replace the previous usage with this:
<Suspense>
  <DashboardPageLazy />
</Suspense>;
```

You can grab the solution from `App-lazy.tsx` if you get stuck at some point.

Build the project as usual, and you'll see that the build produced a bunch of new chunks for our lazy pages. Start the project, enable the **"Disable cache"** checkbox again, and record the performance profile. This time, it will be very different.



First, we're waiting for the download of the critical resources and the initial JavaScript, i.e., `index` and `vendor` chunks. You can verify that those are the only two chunks that will be initially downloaded if you open `index.html` inside the `dist` folder.

After they finish downloading, we see a tiny burst of JavaScript in the main section - this is the browser initializing React and React rendering everything that is not lazy. In our case, it's nothing - we're lazy-loading everything.

At some point during the previous step, the "lazy" page component is mounted, and the download of all the lazy chunks is triggered. We see a burst of downloads in the Network panel.

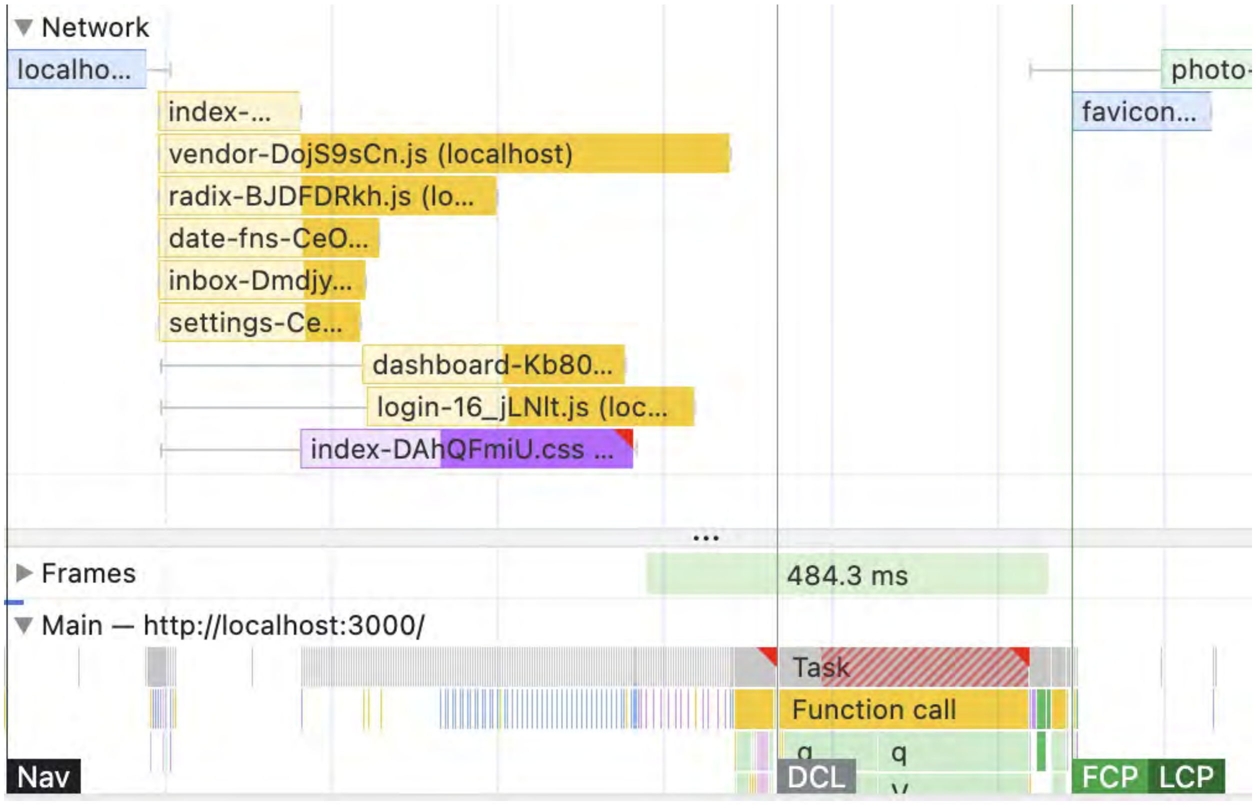
After that one is done, we see another burst of JavaScript in the main section - this is React finally rendering the lazy page.

And after that, finally, we see the LCP/FCP metrics.

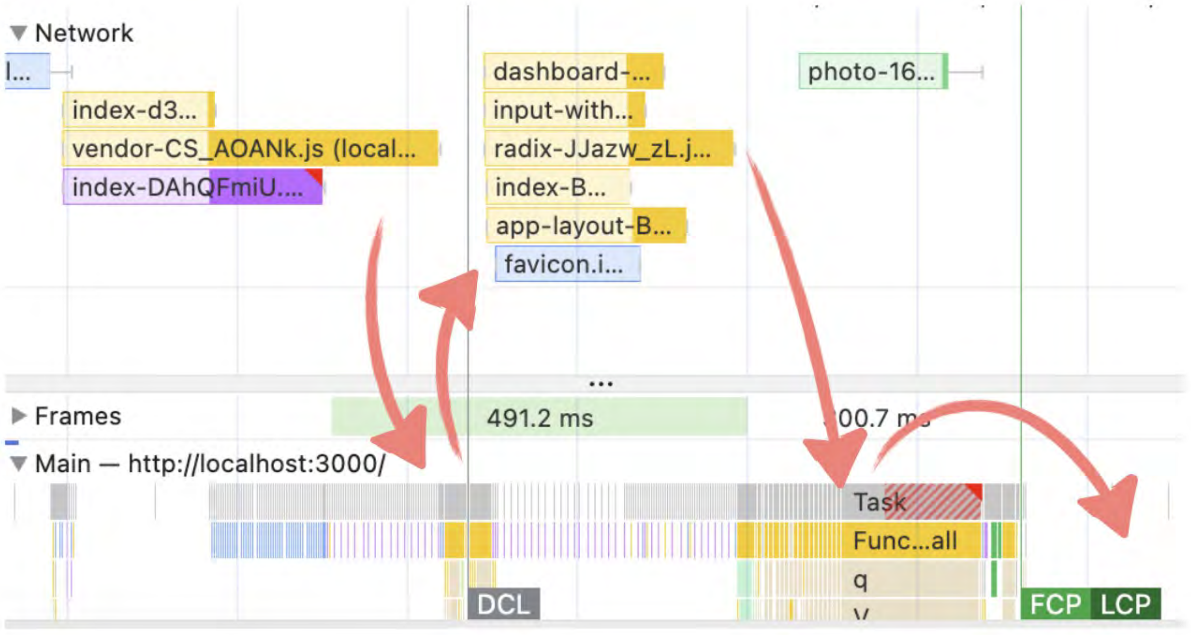
The **LCP** value this time is **1.07**, which is... slightly more than the initial non-split version with full code in the `index` chunk. Ooops. In the effort to improve performance, we actually managed to make it slightly worse.

If we compare those performance profile pictures, the answer to why should be obvious.

Before:



After:



While yes, technically we're downloading less code now, but we're also no longer downloading it in *parallel*. Instead of a simple linear "download everything → render everything", we now have a two-step process, which adds its own overhead. As a result, the benefit we gained from reducing the bundle was mitigated by the overhead of the additional step. And the picture will remain the same for repeat visitors.

So, why exactly would we do that then, if there is no visible benefit?

Two reasons. First, there are no visible benefits in *this* project. Mostly because it's just not big enough. On a project with a few dozen different pages and complicated logic, the result could be better. But this may or may not happen and will highly depend on the project itself, so don't take it as a promise.

The second, and much more important, benefit, however, is very concrete. With lazy-loading like this, we essentially defined the "critical path" from a React perspective in the app. And now we can control it with great precision.

Loading Critical Elements First

By "controlling", I mean we can identify which React elements we absolutely have to have as the top priority and render them first. Let's say we want to prioritize the LCP element as much as possible. Everything else is a second-class citizen.

First, we need to find the element. Open the Home page, record its performance profile, click on the LCP green label there, look in the Summary tab for the reference to the DOM element (it should have "related node" there), and then trace down in the code which React element rendered it. It should be the "My Dashboards" title.

In the code, it lives in the `TopbarForSidebarContentLayout` component (`frontend/patterns/topbar-for-sidebar-content-layout.tsx`), which is in turn used inside `AppLayout` (`frontend/patterns/app-layout.tsx`), which is used on all pages except Login.

The root "Dashboard" page, the one that we lazy-load, looks like this:

```
// FILE: frontend/pages/shared/dashboard-with-layout.tsx
```

```
export const DashboardPage = () => {
  return <AppLayout>... // all the dashboards code</AppLayout>;
};
```

And it's lazy-loaded code inside `App.tsx`:

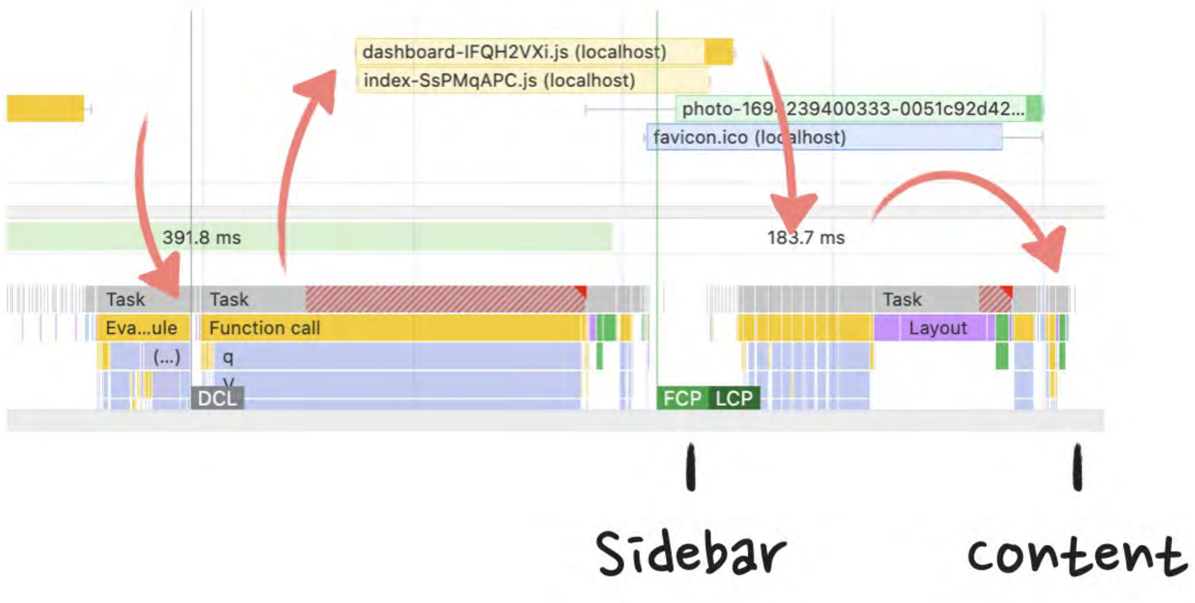
```
<Suspense>
  <DashboardPageLazy />
</Suspense>
```

As we now know, everything that is not wrapped in `Suspense` will be on the "critical" path and will be prioritized. So, what if we move the `AppLayout` component outside of the dashboard page, lazy-loading, and suspense boundary?

Do this:

```
export default function App({ ssrPath }: { ssrPath?: string }) {
  // the rest of the code stays the same

  // wrap Suspense in AppLayout
  return (
    <AppLayout>
      <Suspense>
        <DashboardPageLazy />
      </Suspense>
    </AppLayout>
  );
}
```

And remove `AppLayout` from the `DashboardPage` component. Then, rebuild the project yet again and record the performance profile of the Home page. This time, the picture should be this:



Sidebar and the page title are on the "critical path" now, so they are rendered immediately when the initial JavaScript is loaded. After that, the LCP marker is triggered (because of the title), and the value is now **763 ms**. Almost a 300 ms improvement! The full page is not rendered at that time, but that's okay.

We can go even further and say that the Sidebar is not actually *that* important for our users compared to the page title, and lazy-load the Sidebar as well!

```
// FILE: frontend/patterns/app-layout.tsx

// replace direct Sidebar import with lazy:
const FixedWidthPrimarySidebarSPALazy = lazy(async () => {
  return {
    default: (await import('@fe/patterns/fixed-width-primary-sidebar-
spa')).FixedWidthPrimarySidebarSPA,
  };
});

// in the AppLayout component, replace FixedWidthPrimarySidebarSPA with this
<Suspense fallback={<div className="sidebar-fallback"></div>}>
  <FixedWidthPrimarySidebarSPALazy />
</Suspense>;
```

Rerun the build and re-record the performance. The LCP value is around **695 ms** now. We squeezed another 65 ms out of it at almost no cost. If you refresh the page, you'll see the order in which the elements appear now, with the title appearing much earlier. Slow down the Network even more to get a clearer picture.

Record the repeated visitor's performance for the full picture. Disable the "cache" checkbox, make some changes on the Login page, rebuild, and re-record. The LCP number is now **545 ms**!

Let's add them to our table and celebrate a little 🎉. A 15-25% improvement in numbers is not to be sneezed at.

	Simulation	**Home page LCP**
1	First-time visitors, non-split `index` file (baseline)	915 ms
2	First-time visitors, manual chunks via config	992 ms/915 ms
3	First-time visitors, lazy-loading, LCP element on the critical path	695 ms
4	Repeated visitors with changes in Login, non-split `index` file	640 ms
5	Repeated visitors with changes in Login, manual chunks via config	620 ms
6	Repeated visitors, lazy-loading, LCP element on the critical path	545 ms

Additional challenge

- Find the LCP element on the "Settings" page and refactor the code to minimize its value
- Find the LCP element on the "Inbox" page and refactor the code to minimize its value
- Find the LCP element on the "Login" page and refactor the code to minimize its value

However, there is another cost to those nice numbers, other than the delayed areas on the page. Try to navigate between different pages. Even if you did the Challenge above, it still doesn't look that great. There is a visible blank screen in the content area when you navigate between Home and Settings, for example. And it's especially bad when navigating from pages with the Sidebar to the Login page.

To improve this, we need preloading.

Preloading Lazy Chunks Manually

The idea is simple. When we load the Home page as a result of our chunking efforts, we load only the code that is needed on this page. This means that when we navigate to another page, we need to wait until all the new code is downloaded. Hence, the blank screen - this is our Suspense boundary's fallback rendered until the code is fully loaded.

One way to improve it would be to introduce a nice "loading" fallback that imitates the future layout. So that the page looks like a "skeleton" of the real page - you probably have seen this pattern everywhere.

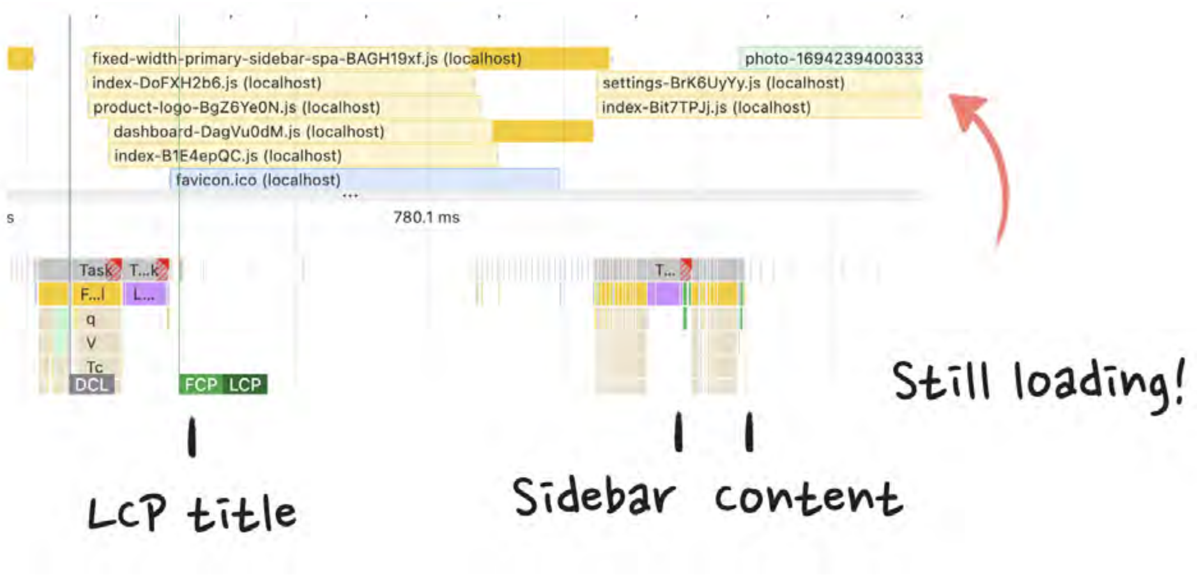
There is, however, another option. What if we just download all the needed JavaScript in advance, but *after* all the critical resources are loaded and ideally rendered? That way, we won't slow down the LCP metric, but also won't have to wait when navigating.

The correct way to do it might depend on the bundler. In Webpack, for example, there is a chance you'd need to install a plugin for this. In Vite, however, we're in luck: all we need to do is add an `import` statement at the top of the file, and it will take over from here.

Let's, for example, preload the "Settings" page for the Home page:

```
// Just add this at the top of ./dashboard.tsx
import('./settings');
```

Rebuild the project, and now navigation from Home to Settings is close to how it was before. If you record the initial load performance of the Home page, you'll see the `settings` chunk is somewhere at the end. And this chunk is not blocking anything. The page content is rendered *before* its download has finished. This will be especially visible on slower Network settings:


fixed-width-primary-sidebar-spa-BAGH19xf.js (localhost)
index-DoFXH2b6.js (localhost)
product-logo-BgZ6YeON.js (localhost)
dashboard-DagVu0dM.js (localhost)
index-B1E4epQC.js (localhost)
favicon.ico (localhost)
photo-1694239400333
settings-BrK6UyYy.js (localhost)
index-Bit7TPJj.js (localhost)
780.1 ms
DCL
FCP
LCP
Still loading!
LCP title
Sidebar
content

In theory, we could do this for every page easily. We could even be smart about it and preload only those chunks that are needed on every page. For example, from the Login page, we can navigate only to the "Home" page. So there is no need to preload *everything* there, just the "Home" page chunk will be enough.

And as a quick and easy win in small projects, this solution is totally okay. However, when a project grows, adding those imports manually everywhere will become tiresome pretty quickly. Especially if you need to track which page or component is used where. Surely there is a way to do it better?

Preloading Lazy Chunks With the Link Component

There is, actually! If you look at the code, you'll see that all links are implemented using the `Link` component (`frontend/utils/link.tsx`) underneath. As it should be in an SPA. So in theory, this `Link` can be a bridge between all the URLs and all the dynamic chunks.

The simplest and "naive" way to achieve this is to introduce some sort of mapping between the paths and the associated lazy imports needed for this path:

```
const preloadingMap = {
  '/': () => import('./pages/dashboard'),
  '/settings': () => import('./pages/settings'),
  '/inbox': () => import('./pages/inbox'),
  '/login': () => import('./pages/login'),
};
```

Then, in the `Link` itself, we can trigger the "preloading" functions associated with the path:

```
// inside Link component
useEffect(() => {
  if (href && preloadingMap[href]) {
    const preload = preloadingMap[href];
    preload();
  }
}, [href]);
```

Add this code, rebuild the project, and record the initial load of the Home page.

You'll see a bunch of new chunks being preloaded, including the `login` chunk. All of them are triggered only when the sidebar is rendered and the Link component is mounted, which is even better than what we had with the manual import. Now we don't have the problem of accidentally cluttering the Network with too many preloads and delaying the downloading and rendering of the main content. Which, in theory, could happen with too many preloads too early.

From the Home page, navigate to the Login page (by clicking on the logo) and enjoy the instantaneous transition. The speed of transition is comparable to the baseline project, where we had all the code bundled together in just one `index` chunk. But now we have the best of both worlds: improved initial load plus almost infinite scalability. New pages that we'll inevitably add to the project won't slow down the existing pages.

This Link-based preloading is great, but its implementation is **rudimentary** here at best. If you refresh the Login page and then try to navigate to the Home page (by clicking on the title), you'll see that the `Sidebar` component is still taking its time: it was split into its own **independent chunk** by the bundler, and we haven't preloaded it. To do this properly, we would need to extract this information from the bundler by generating a manifest file that maps routes to all the chunks used on this route, and then pass this information to the frontend somehow.

Also, preloading *all* Links on the page is excessive. It's fine on a small-ish project that we have, but in the real world, there will be dozens of links on a page. Preloading all of them at the same time is not only unnecessary but, yet again, can clog the network and potentially delay the loading of something critical. In the ideal world, we'd want to have more control over what's preloaded. Typical strategies that you'll see in the wild are preloading only the links that are in the viewport (i.e., visible on the screen) or only triggering preloading on hover.

At this point, if we continue down this path, we'll start implementing a full-blown custom framework and start competing with the likes of Next.js and Remix/React Router. However, this book is not about how to implement your own framework and take over Vercel. It's more about customer-facing websites and different patterns that affect their performance.

So, since we now know exactly what we want and why, there is no harm in using some existing solution.

Bundles, Loading, and Frameworks

Choosing the right solution is a complex topic. Especially in the frontend world, where best practices tend to change radically every two years, and frameworks are constantly evolving, being deprecated, or replaced by the new shiny tool.

At the moment of writing this chapter (April 2025), the frameworks recommended by the React[133] team are Next.js and React Router v7 (ex Remix), if we're talking about the web, not native apps. These are the current major players that have been out there for a while. There is also a list of up-and-coming frameworks[134], which includes a new kid on the block: "TanStack Start[135]". Personally, I have lots of experience with Next.js and React Router, but I've never used TanStack. So it's a good time to try it out now.

Because the actual framework doesn't really matter if you know exactly what you need and what's possible. Learn the fundamentals first, and you'll be able to switch between frameworks at a moment's notice, quickly understand their strengths and weaknesses, take advantage of their features, compensate for their flaws, and adjust their default behavior to your exact needs.

The fundamentals that are important to migrate our Study Project to *any* of the above-mentioned frameworks are:

1. How to initialize a project? Typically, there will be either some command or some repository to clone.
2. Is the framework **SSR** (server-side rendering) by default or **CSR** (client-side rendering)?
3. How is **routing** implemented? Typically, it will be either file-based routing, where each page has to be its own file, or code-based routing, where you declare routes in code, or some combination of both.
4. How is **navigation** implemented? Typically, there will be a `Link` component and a few hooks for imperative control, like `useNavigate`, `useLocation`, `useRouter`, etc.

This is it. There will be, of course, many, many more features in each of them. It will be very easy to get lost in the tons of documentation and start thinking that mastering it would take months. Ignore all of it for now. You'll figure all of them out when/if you need them, as long as you know the fundamentals.

Migrating Study Project to TanStack

So, let's migrate and see what it can give us and whether it's better than our custom solution.

If you get stuck at some point or just want to skip to the end and start measuring performance on a working app right away, the final version is in the `src/chapter9-advanced-lazy-loading/tanstack-router` folder. Build and start it as usual:

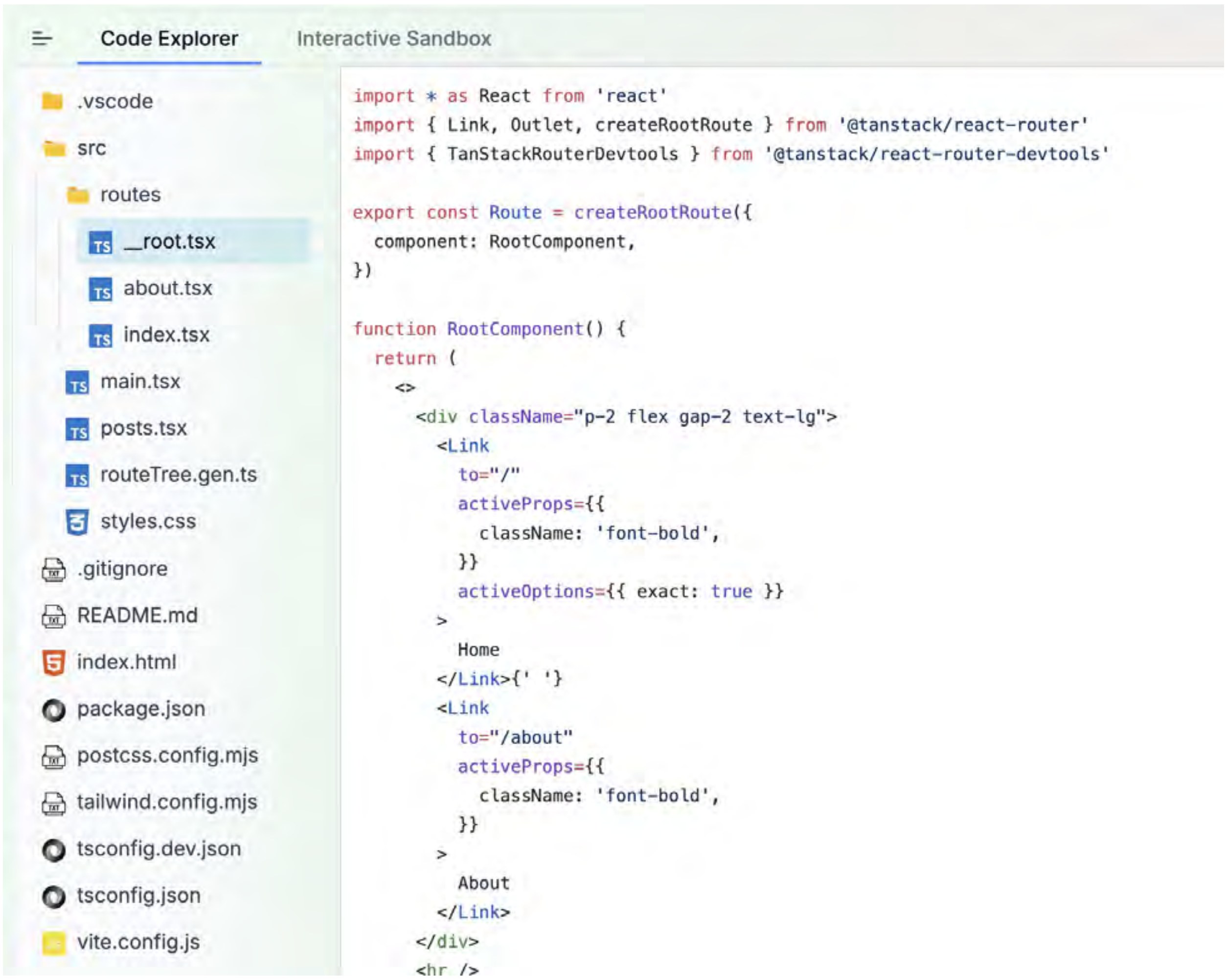
- In dev mode with `npm run dev --workspace=chapter9-tanstack-router`.
- Build the production version with `npm run build --workspace=chapter9-tanstack-router`.
- Start the production version for performance measurements with `npm run start --workspace=chapter9-tanstack-router`.

Considering that the framework is in "beta" mode at the moment of writing the chapter, chances are that when you read it, half of the API and documentation links will change. So I'll just describe the steps that you need to take and hope for the best.

However, I highly encourage you to stop reading right now and try to figure everything out for yourself first. Only then, look at the steps I took and the changes I made, and compare your solution to mine.

Step 1: Initialize the New Project

I used the "Quick start (file-based)"[136] example[137] from the list. Clone it and install all dependencies in a fresh repository outside of the Study Project. Then navigate through the files/folders and take a good read.



```
import * as React from 'react'
import { Link, Outlet, createRootRoute } from '@tanstack/react-router'
import { TanStackRouterDevtools } from '@tanstack/react-router-devtools'

export const Route = createRootRoute({
  component: RootComponent,
})

function RootComponent() {
  return (
    <>
      <div className="p-2 flex gap-2 text-lg">
        <Link
          to="/"
          activeProps={{
            className: 'font-bold',
          }}
          activeOptions={{ exact: true }}
        >
          Home
        </Link>{' '}
        <Link
          to="/about"
          activeProps={{
            className: 'font-bold',
          }}
        >
          About
        </Link>
      </div>
      <hr />
```

Even without reading the docs, just from the structure, we can already make certain assumptions.

First, inside the "routes" folder, there are `about` and `index` files. Since I chose the "file-based" example, it's reasonable to assume that the example project has two routes - "About" and "Index". This is actually why I usually prefer file-based routing - it makes the structure very clear. If you open any of those files, you'll see regular React component code. This is where our pages like Home and Settings will go.

There is also a `__root` file. Since it starts with `__`, which is a naming convention for internal/private methods, it's probably not actually a route, but a special "root" file. Most likely, this is where stuff like meta-tags and shared layout will be. If you search the documentation for "root", the assumption will be confirmed - this is indeed the entry point[138] to the entire app. The code there is always rendered, according to the docs. So this is obviously where our Study Project's `AppLayout` will live.

There is also a `main` file - this is pretty much the same as in the Study Project, where we call React's `render`. It just has a bit more wiring for the TanStack router.

Then, there is the `posts` file - that's a bunch of fetch calls, we don't fetch data yet, so we can just delete it. Then, there is some sort of generated file `.gen.ts` - don't care, it's generated by the framework. And `styles.css` - that one is obvious, our CSS goes here.

And finally, there is `vite.config.js` - we already know that one, looks like they are also using Vite underneath. Makes things easier.

You'll find the scripts for running the project in `package.json`. Should be `dev` for dev mode, `build` and `start` for building and starting.

Step 2: Copy the Config and Study Project Files

Copy everything to your fresh repo!

- Copy the entire `frontend` folder.
- Contents of `index.css` to `styles.css`.
- The entire `tailwind.config.js` file.
- From `tsconfig.json`, copy the `paths` part - that's the `@` aliases we use to reference code in the `frontend` folder.
- From `vite.config.ts`, copy the `resolve` part - that's again the `@` aliases. Otherwise, you'd have to change all the imports in the code, and that's a very boring job.
- Also, from `vite.config.ts`, copy the `preview` part - we need to make sure the "production" config is the same in both projects, so that we can compare apples to apples.

- From `package.json` - copy all dependencies. Some of them will be unused, but it doesn't really matter here.

Don't forget to install all dependencies after you copy them with `npm install` and then double-check that the project still runs with all the config changes.

Step 3: Create Routes

Get rid of the `about` file from the `routes` - we don't need that one. Override the content of our `main.tsx` file with the content of the example main.tsx[139].

Then open our `App.tsx` file, where we had our routes and their pages. All we need to do is recreate that setup. Create `login`, `settings`, and `inbox` files instead of the deleted `about`, and render the required pages in each of them.

The code for each file, including the `index`, will be something like this:

```
import * as React from 'react';

// that will be different for each file
import { DashboardPage } from '../pages/dashboard';
import { createFileRoute } from '@tanstack/react-router';

// route as well
export const Route = createFileRoute('/')({
  component: HomeComponent,
});

// and render the component here
function HomeComponent() {
  return <DashboardPage />;
}
```

In the `__root.tsx` file, change the `RootComponent` to only render the `Outlet` component - that's the content of each route, i.e., our pages. And wrap it in our `AppLayout`, same as we did in the previous section, for all pages to speed up the LCP.

```
function RootComponent() {
  return (
    <AppLayout>
      <Outlet />
    </AppLayout>
  );
}
```

If you skipped the previous section, then in addition, go through all pages and get rid of the `AppLayout` component there.

We also need to opt out of the layout for the Login page - that one doesn't have navigation. So we need to detect which route we're on in this root component and render only the `Outlet` for those:

```
function RootComponent() {
  if (isLogin) return <Outlet />;

  return (
    <AppLayout>
      <Outlet />
    </AppLayout>
  );
}
```

For this, we need to read the docs of the actual router[140] - all of them will have something like what we need in one way or another. For example, we can try the useLocation[141] hook and extract the `pathname` from it:

```
function RootComponent() {
  // Get current pathname
  const { pathname } = useLocation();

  // If it's the login route, don't use AppLayout
  if (pathname === '/login') {
    return <Outlet />;
  }

  // Otherwise, use AppLayout for all other routes
  return (
    <AppLayout>
      <Outlet />
    </AppLayout>
  );
}
```

At this point, the app should work and behave as expected. Except for one thing.

Step 4: Fix Navigation

We still have our rudimentary `Link` component for navigating between pages. To take advantage of the built-in framework features, we need to use the one that is provided to us. Luckily, it's easy: we just need to trace where we use our custom `Link` and replace it with the router's Link[142].

A quick search through the project reveals that it's used in just three places:

- `SidebarLinkItem` component in `frontend/components/sidebar/navigation-items.tsx` for the Sidebar items.
- `LoginPage` component in `./pages/login.tsx` for the navigation from the title to the Home page.
- `PrimarySidebarHeading` component in `frontend/patterns/primary-sidebar-heading.tsx` for the navigation between the logo and the Login page.

In all of the usages, replace the import with `import { Link } from "@tanstack/react-router";` and change the prop on the `Link` from `href` to `to`.

That's it, we're finally good to go!

Build and start the finished project, and let's see what we've got by default.

Exploring the Performance of the Study Project on Tanstack

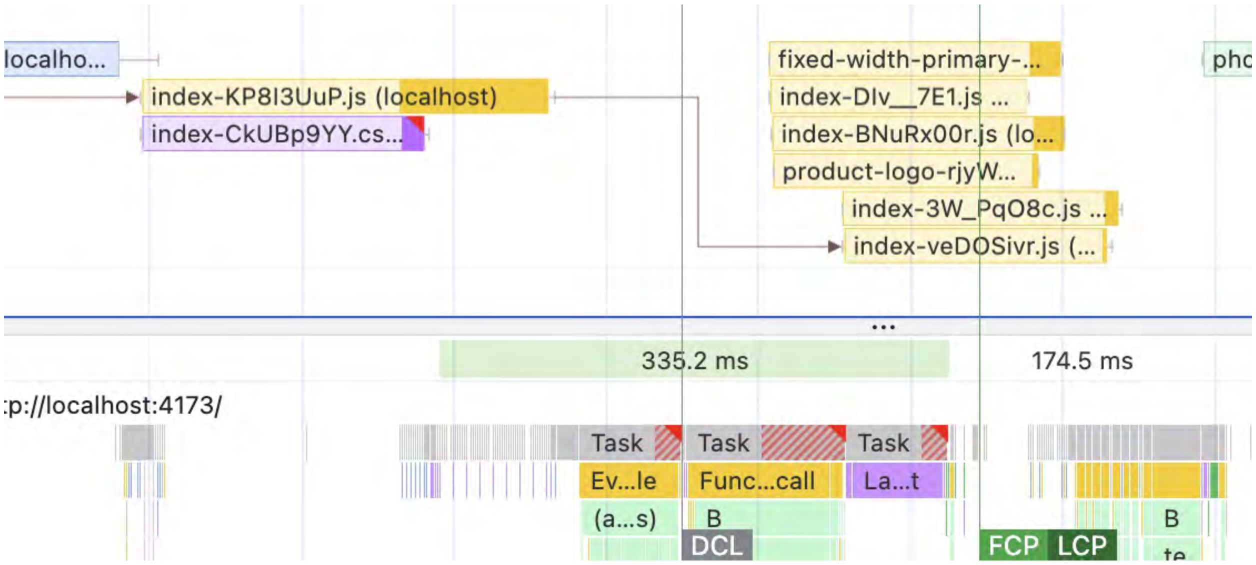
These are the simulations we recorded before:

	Simulation	**Home page LCP**
1	First-time visitors, non-split `index` file (baseline)	915 ms
2	First-time visitors, manual chunks via config	992 ms/915 ms
3	First-time visitors, lazy-loading, LCP element on the critical path	695 ms
4	Repeated visitors with changes in Login, non-split `index` file	640 ms
5	Repeated visitors with changes in Login, manual chunks via	620 ms

	config	
6	Repeated visitors, lazy-loading, LCP element on the critical path	545 ms

Let's now add the "First-time and Repeated visitors" on Tanstack to the list. Not for the sake of shaming or getting excited about the framework. But to see and understand what the framework is doing.

First, record the performance profile of the Home page with the enabled "no cache" checkbox, 6x CPU, and Fast 4G throttling - our "First-time visitors" simulation. The picture should look like this:



Which is pretty much ***identical*** to our implementation of route-based lazy-loading. At first, the must-have resources (index JS and CSS) are loaded, the JS file triggers the download of all other JS files, and the LCP value is triggered way before the rest of them finish downloading.

Only this time, we didn't have to do those manual `lazy` elements everywhere. The framework handled that for us. The LCP number is almost the same as well, just a little bit higher (**726 ms** vs 695 ms), most likely because of the framework's overhead.

There are, however, a few differences. Now that we have implemented that thing from scratch, we can tell exactly what, why, and how those differences will affect performance.

Framework's Chunking Strategy

The first thing that immediately jumps to mind is that there are no "vendor" chunks as we had. The framework does code splitting automatically. Which is good. We didn't have to change any configurations for this. And it does it really well, since the end result matches our very optimized LCP number.

But the libraries are sitting in the same bundles as our "own" code. As we already know, this could potentially lead to a worsening of the repeated visitors' performance. And indeed, when I measure it in exactly the same way as before, I get **628 ms**. Our manual solution got me **545 ms**.

	Simulation	Home page LCP
1	First-time visitors, non-split `index` file (baseline)	915 ms
2	First-time visitors, manual chunks via config	992 ms/915 ms
3	First-time visitors, lazy-loading, LCP element on the critical path	695 ms
4	First-time visitors with a framework, LCP element on the critical path	726 ms
5	Repeated visitors with changes in Login, non-split `index` file	640 ms
6	Repeated visitors with changes in Login, manual chunks via config	620 ms
7	Repeated visitors, lazy-loading, LCP element on the critical path	545 ms
8	Repeated visitors with a framework, LCP element on the critical path	628 ms

If we're unhappy with that slight increase in numbers, we'd have to tinker with the chunks' configs manually again. The default installation of the framework has Vite underneath, so it shouldn't be a problem if there is a need.

Additional challenge

Speaking of Vite. Tanstack Router can be installed with other bundlers[143]. For example, with Webpack[144]. Try to migrate the Study Project on Tanstack from Vite to Webpack, and then measure the same scenarios.

- Do the numbers go up or down?

- What about other bundlers?

Framework's Preloading Strategy

One big thing for which we introduced quite a lot of complexity was preloading. Remember - we had to create and maintain a map of all the paths and their corresponding bundles, modify the `Link` component, import chunks manually, and implement their preloading manually.

Tanstack implemented all of this for us. If you open our default Tanstack Study Project and look at what is happening in the Network panel, you'll see that initially, other pages' chunks are not loaded. Neither `settings` nor `login` show up right away. However, when you hover over the links that lead to those pages, the new chunks appear.

This is the "intent" preloading strategy[145], which is enabled by default. The framework also supports "viewport" and "render" - two other most popular preloading strategies, where preloading happens when a component appears on the screen or mounted. Switching between them is just a matter of setting a different prop on a `Link` or a setting inside `vite.config.ts` .

Experimenting with different preloading strategies and measuring the performance impact of them can be a ton of fun! I highly recommend you spend some time here and play around with all the settings.

Additional challenge

Now that you know all the steps needed to migrate the Study Project to Tanstack, try to do exactly the same migration to the React Router[146] framework.

- What will be different in the default configuration of Tanstack vs React Router?
- How does React Router handle chunking compared to Tanstack? What about preloading?
- Which default configuration is better for which use case?
- Are performance numbers the same or different between those frameworks? After you're done with React Router, try to do exactly the same thing with Next.js[147]. Pay attention to the fact that Next.js is SSR by default.

- What changes in the performance profile because of SSR by default?
- What are the pros and cons compared to the previous frameworks?
  If you do this, you probably won't even need to read the next section, you'll learn it by doing 😎.

Lazy Loading + SSR

We have one last thing to verify before moving on from lazy loading. Until this moment, the website was on CSR (Client-Side Rendering). But what about SSR (Server-Side Rendering)? In the previous chapters, we proved that SSR could be really great for the initial loading and LCP metrics, but that it can delay interactivity on a page. Does lazy loading help with it, or make it worse?

It's really easy to find out now that we have all the knowledge and tools.

The LCP time of the baseline project is **915 ms**. Since this is rendered on the client, this value is also the value when everything becomes interactive.

Reset all the changes we've made so far to the baseline project. Don't remove them, just stash or commit to a separate branch - we'll need to return to them in a moment.

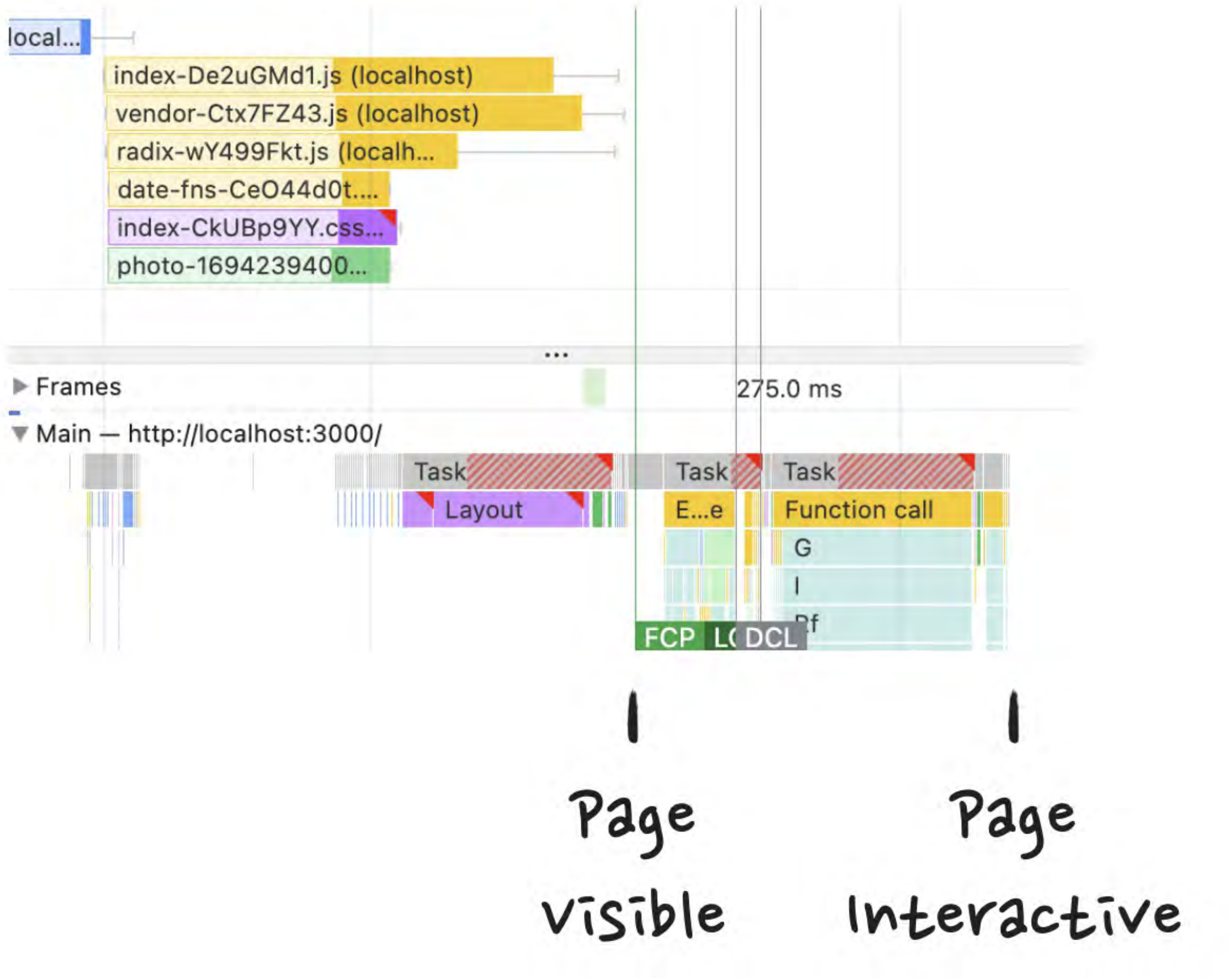
Build both server and client:

```
npm run build --workspace=chapter9-simple-frontend
npm run build:server --workspace=chapter9-simple-frontend
```

Uncomment the following line in `server/index.ts` - this will enable our rudimentary SSR for the baseline project.

```
return c.html(preRenderApp(html, c.req.path));
```

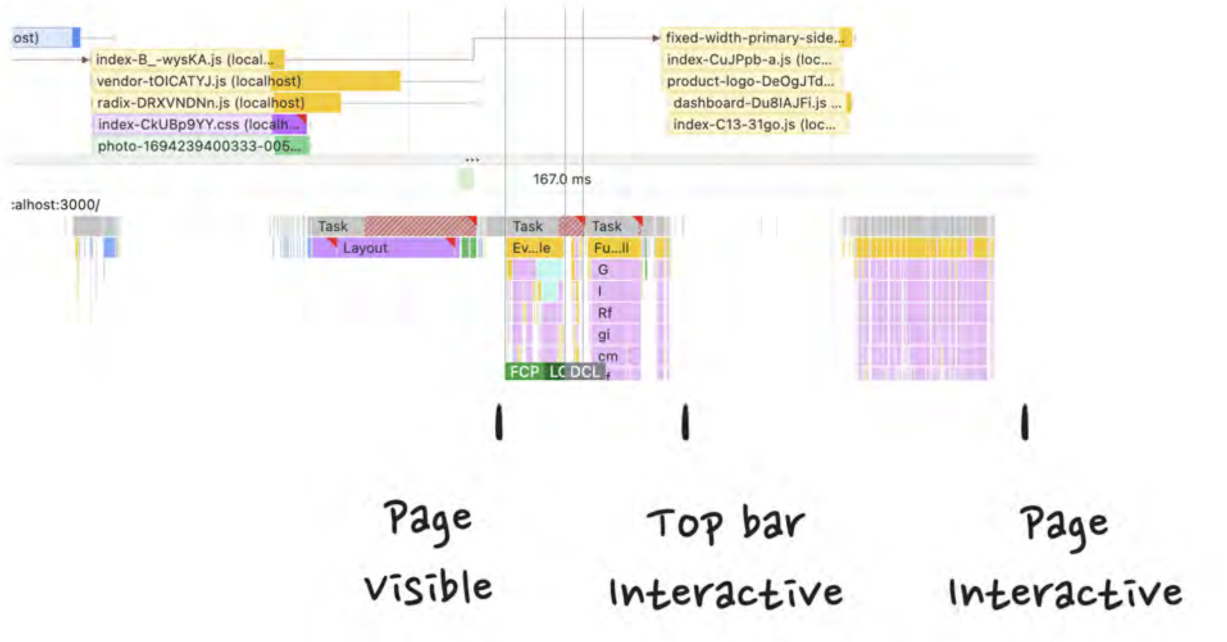
Start the project and measure the initial performance. The picture should be a typical SSR picture: the page becomes visible before JavaScript is downloaded and interactive after it's executed.



The LCP value in this scenario is **600 ms**. You'll have to manually extract the "interactive" time from the timeline since there is no highlighted metric here. For me, this value is around **850 ms**. The gap between them is **250 ms**. This is SSR's biggest downside - when the page is visible, but nothing on it works except for links. And this is on fast 4G, by the way! On something like slow 3G, this gap could be 3 seconds.

Now with lazy loading. Restore everything we did: code splitting per page, extracted layout, and lazy-loaded sidebar. Record it now.

This time, the picture should be different. We'll have the same pattern as before with CSR in the Network section - some initial JavaScript that, at some point, triggers more JavaScript downloads. But this time, this pattern will be applied to the SSR "reverted" pattern of the "main" section. Where first the page is rendered as pure HTML, becomes visible, and only then the JavaScript is executed, which then "hydrates" the already visible page - this is when it becomes interactive.

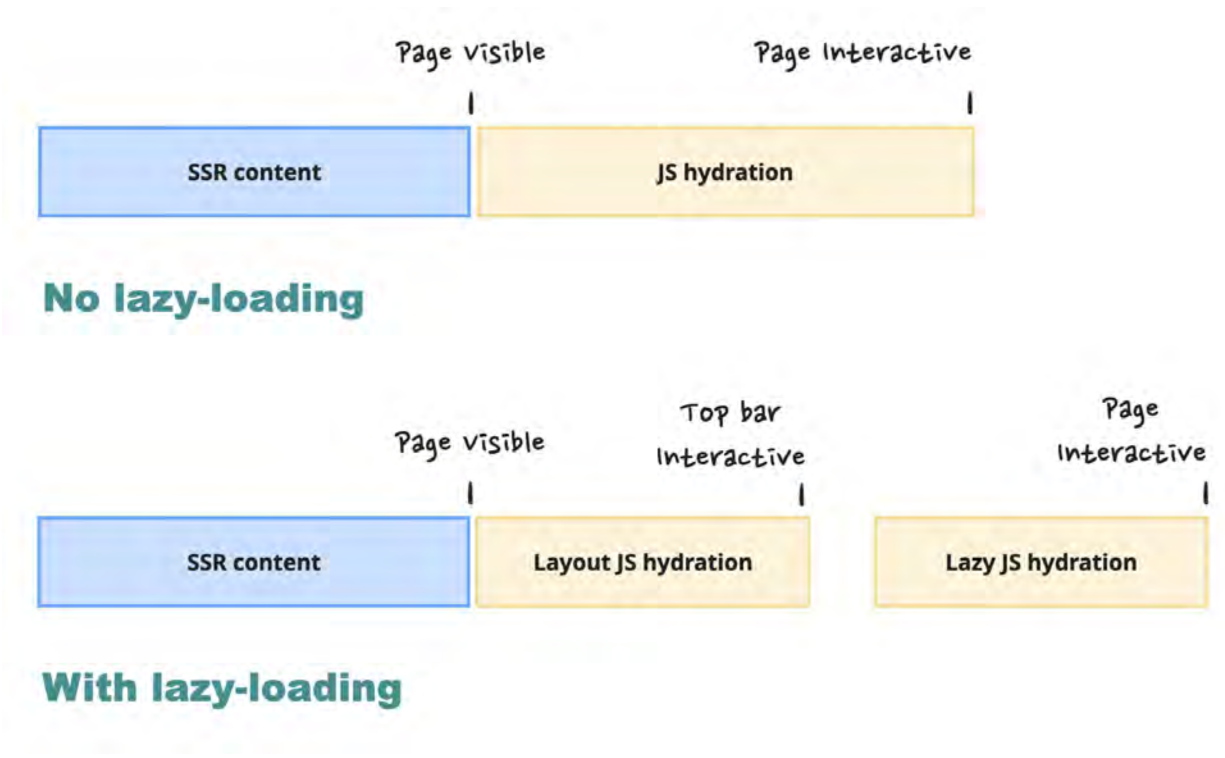

ost)
index-B_-wysKA.js (local...
vendor-tOICATYJ.js (localhost)
radix-DRXVNDNn.js (localhost)
index-CkUBp9YY.css (localh...
photo-1694239400333-005...
fixed-width-primary-side...
index-CuJPpb-a.js (loc...
product-logo-DeOgJTd...
dashboard-Du8IAJFi.js ...
index-C13-31go.js (loc...
167.0 ms
:alhost:3000/
Task
Layout
Task
Ev...le
Task
Fu...ll
FCP
LC
DCL
Page visible
Top bar Interactive
Page Interactive

Considering that half of the JavaScript is lazy-loaded, we have this structure. The page becomes visible at the same time as before, with the LCP **600 ms**. This is because the bottleneck here is the render-blocking CSS file, which hasn't changed with lazy-loading.

Then, the initial JavaScript, which processes our routing and layout, finishes downloading and executing. This is when everything in our AppLayout becomes interactive. This number is around **720 ms** for me.

At the same time, the "lazy" JavaScript is triggered - this is when components inside our routing, i.e., our lazy page and the Sidebar, are mounted. After they are downloaded, they are also executed, and finally, those blocks become interactive too. This happens around the **1 s** timeline.

So, for SSR, the overall benefit of lazy-loading is pretty much the same as for CSR. It can help you speed up the *most crucial* part of the page, in our case, the "top bar" component, at the expense of the rest of the page. Which part is the most crucial is always up to you. If we assume that the most crucial part of this page from an interactivity point of view is the "search" functionality, then with lazy-loading, we managed to speed up its time-to-interactive by **130 ms**.



Additional challenge

- Try to implement SSR with the Tanstack framework. Will the results be the same?
- Try to enable SSR in React Router - will the result change? Compare the output code of the two of them.

What's Next?

I hope you can now explain how a page will behave in different conditions with different rendering scenarios in your sleep. However, this is still not the end of it! There is also a thing called "React Server Components", which adds even more on top of all of those concepts. As if it's not complicated enough.

But at least we're fully technically and mentally prepared to deal with the most hyped topic of recent years in the React community. So, finally, everything you want to know about React Server Components, right in the next chapter!

10. Data Fetching and React Server Components

Have you heard of React Server Components[148]? You probably have. It's one of the most talked-about and controversial topics in the React world over the last few years. And it's also the most "magical" and most misunderstood feature.

So it's time to make it understood. Luckily, we already have so many tools we can use: we now know what initial load is, how to measure it, how to record performance graphs, what SSR and CSR are, and how to find them on those graphs. We'll need all of it to figure out that magic.

Although we still need more. We need to talk about data first: how to fetch data on the client and on the server, how to pass data from the server to the client, and why existing tools are not good enough.

This is probably why Server Components are so misunderstood: you need to have a PhD in React development before touching them 😅

Setting Up the Project

We're going to work with the same beautiful app that should be very familiar by now. Only this time, we'll make it more "real" by fetching some data from remote sources.

This chapter probably has the most complicated setup. We're going to start with a baseline project in `src/chapter10-data-fetching/frontend/baseline`. It's going to be useful for the initial measurements. For this chapter, we'll focus entirely on the Home page.

Build, start, and open the project as usual:

```
npm run build --workspace=chapter10-baseline-frontend
npm run start --workspace=chapter10-baseline-frontend
```

The project has a Sidebar with menu links on the left, and a few dashboards and tables in the content area. Right now, all of this data is "static" - i.e., it's hard-coded into the components themselves.

Over the course of this chapter, we'll make the sidebar links and table data dynamic. We're going to work only with those entities, but I highly encourage you to make other charts dynamic as well on your own. It will be much more fun to play around with different combinations when you have five dynamic entities on a page instead of just two.

The backend API for those two endpoints lives in `src/chapter10-data-fetching/backend-api`. Start it:

```
npm run start --workspace=chapter10-backend-api
```

Navigate to `http://localhost:5432/api/sidebar` and `http://localhost:5432/api/statistics`. You should see the JSON responses with all the necessary information.

The endpoints implementation is extremely complicated and looks like this:

```
// FILE: src/chapter10-data-fetching/backend-api/index.ts
app.get('/api/sidebar', async (c) => {
  await sleep(500);

  return c.json(sidebarData);
});
```

In real life, you'd need to connect to a remote resource like a database to extract this data. Fortunately, we don't really need to go that far to investigate frontend performance. We can just imitate the inevitable delay with the `sleep` function.

That's it for the setup, let's do things now.

Data Fetching on the Client

Implementing Client-Side Fetching

Let's start with the "classic". We have some remote data. And a client-side rendered SPA. The easiest way to get this data is to fetch it on the client.

To do this for the Sidebar, we need to find a component that renders the items:

```
// FILE: frontend/patterns/primary-sidebar-primary-group-spa.tsx

export const PrimarySidebarPrimaryGroupSPA = ({ ...props }) => {
  return (
    <NavigationGroup header="general" {...props}>
      <SidebarRegularLinkItem href="/" before={<HomeIcon className="w-8 h-8 sm:w-6 sm:h-6"
/>}>
        Home
      </SidebarRegularLinkItem>
      ... // all other static items
    </NavigationGroup>
  );
};
```

Fetch the data from the API we have:

```
useEffect(() => {
  const fetchSidebarData = async () => {
    const response = await fetch('http://localhost:5432/api/sidebar');
    const data = await response.json();
  };

  fetchSidebarData();
}, []);
```

Introduce some state to store this data and the progress of fetching it:

```
const PrimarySidebarPrimaryGroupSPA = ({ ...props }) => {
  const [sidebarData, setSidebarData] = useState(undefined);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    const fetchSidebarData = async () => {
      const response = await fetch('http://localhost:5432/api/sidebar');
      const data = await response.json();

      setSidebarData(data);
      setIsLoading(false);
    };

    fetchSidebarData();
  }, []);
};
```

And then, instead of the static items, render the dynamic data when we have it, or some nice Sidebar skeleton while we're loading.

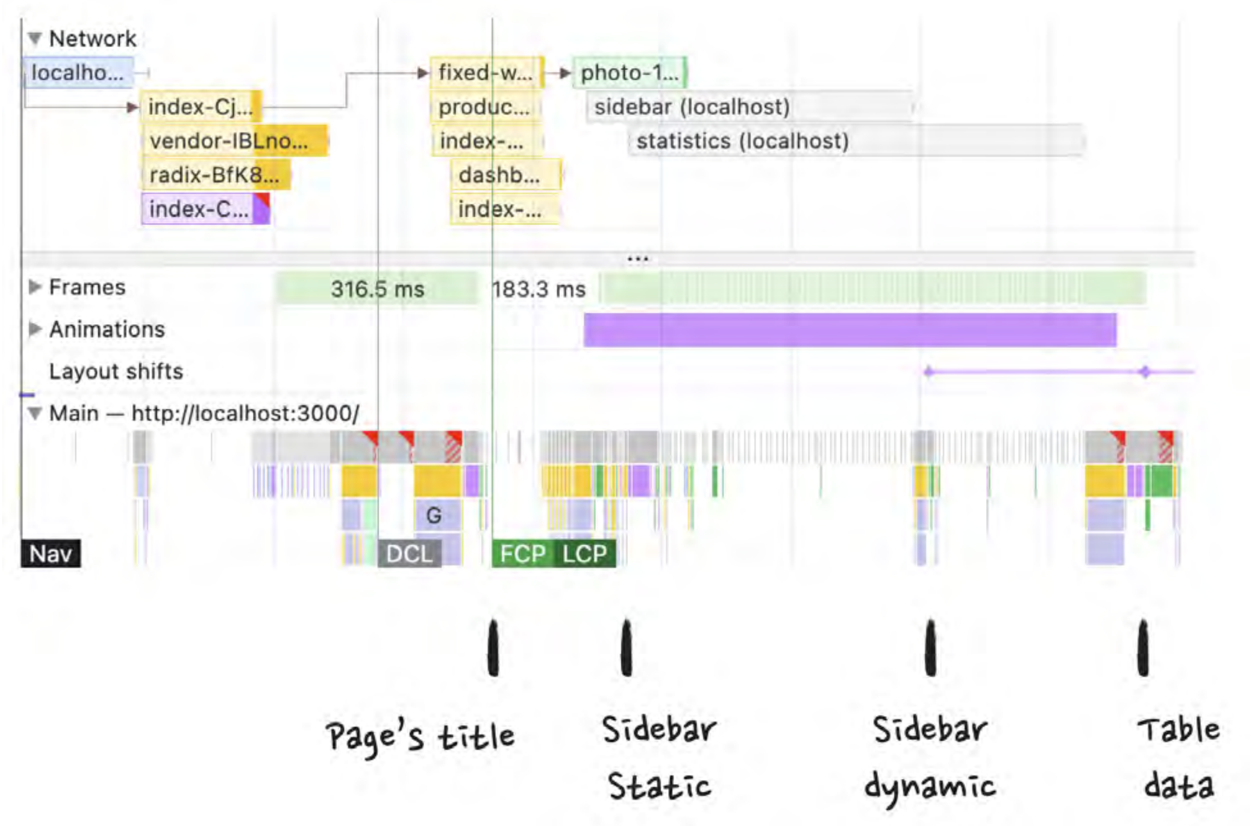
```
const PrimarySidebarPrimaryGroupSPA = ({ ...props }) => {
  return (
    <NavigationGroup header="general" {...props}>
      {isLoading ? <SidebarSkeleton /> : renderSidebar(sidebarData)}
    </NavigationGroup>
  );
};
```

There will be a bit of coding to implement that rendering and skeleton. If you don't feel like doing it yourself, use the `renderSidebar` function and the `SidebarSkeleton` component from `frontend/utils/sidebar.tsx`. Then do exactly the same thing for the table data inside the dashboard component inside `frontend/pages/shared/dashboard-without-layout.tsx`.

Alternatively, if you really don't want to bother with all the coding above, you can switch to the `src/chapter10-data-fetching/frontend/client-fetch`. It has all of this implemented already.

Analyzing Client-Side Fetching

Build and start the project as usual, open it in the browser, record initial load performance, and take a peek inside. By now, you should be able to read what's happening there in your sleep.



First, we fetch and then render the "critical path" assets. The user can see the page's title. Then, the "lazy" components are downloaded, and the "static" parts of those components are rendered. This includes a few items in the Sidebar, like the "Create" button, and the charts on the main page. After that, the fetch for the dynamic items kicks in. Only after we have the fetch results can we render the dynamic Sidebar items and the table data.

The **LCP** number here is around **730 ms**. Which, technically speaking, is pretty good. But the users will see the Sidebar items around the **1.4 s** mark and the table around the **1.8 s** mark. Between those times, for around 700 ms, they admire the loading animation. Whether this situation is good or not depends entirely on you.

Let's say we're not happy with such long loading times and want to shorten them as much as possible. In this case, there are only two things we can do. Either shorten the fetch time, which would mean optimizing the backend logic, which is completely out of scope here. Or move the beginning of the fetch to the left, so that it finishes sooner. This means **prefetching**.

Prefetching Data via Promises

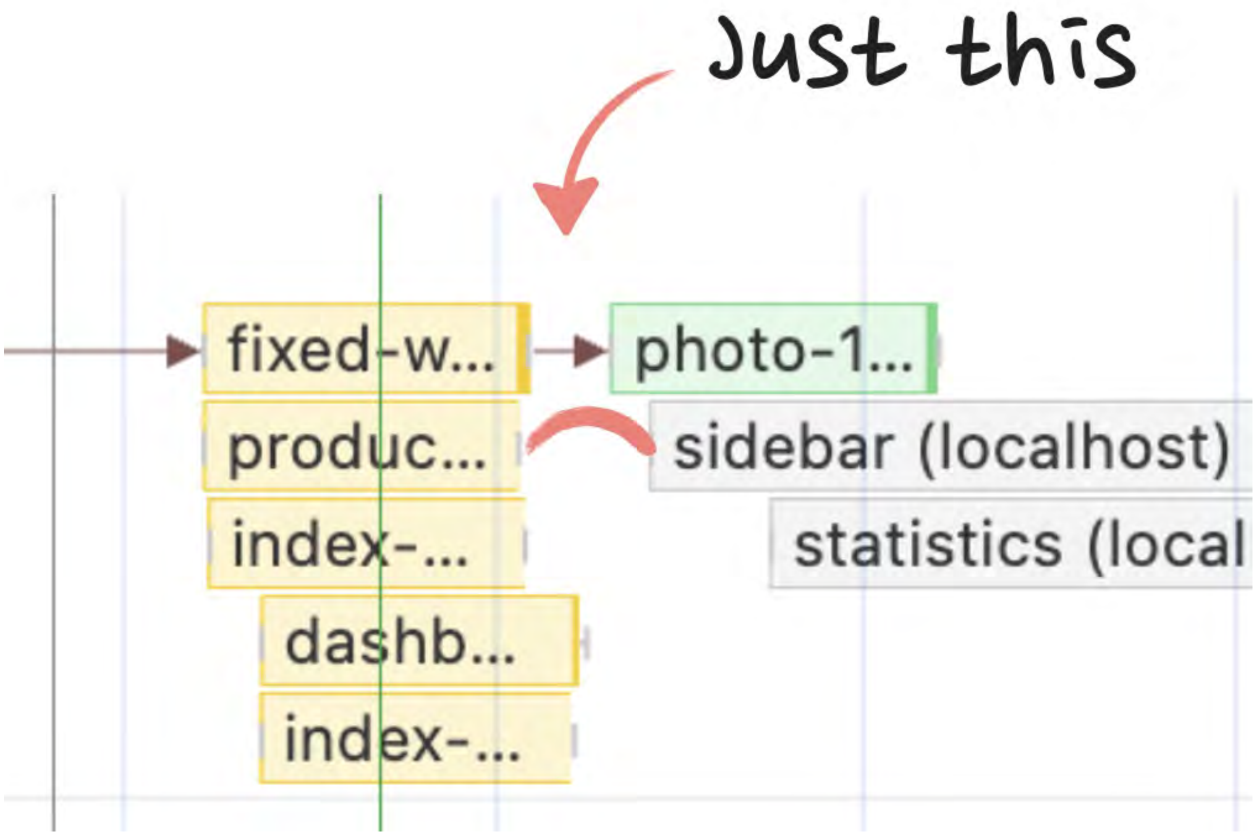
From the previous chapter, we already know about preloading the lazy bundles. With data, it's the same concept, only we often refer to it as **prefetching** sometimes.

Currently, we're calling `fetch` inside `useEffect`. I.e., only after the component that needs the data is *mounted*. So, in theory, we could've applied the same solution as we had for bundles - trigger the loading outside of the component. In the case of data, it would need a bit more work since the bundler doesn't help us here, but it is still doable.

```
// Trigger the data fetch before the component
// And save the promise in a variable
const preloadPromise = fetch('http://localhost:5432/api/sidebar');

const PrimarySidebarPrimaryGroupSPA = ({ ...props }) => {
  // exactly the same code as before

  useEffect(() => {
    const fetchSidebarData = async () => {
      // await for the promise triggered above the component
      const response = await preloadPromise;

      // everything else is exactly the same
    };

    fetchSidebarData();
  }, []);

  return; // same stuff as before
};
```

As you know, `fetch` is a Promise[149]. Nothing stops us from creating this promise before mounting the component, saving it to a variable, and then awaiting it to resolve inside `useEffect`.

The implementation above, however, won't do you much good: both the Sidebar and Table are lazy-loaded. So the fetch Promise will be triggered only when the lazy bundles are downloaded. We're just saving the time it takes the browser to parse and execute the lazy bundles. It's probably around 50 - 100 ms, depending on the computer and complexity, but nothing ground-breaking.



However, now that we know that the fetch can be extracted and shared as a variable, why not create it *before* loading the lazy bundles, somewhere at the very beginning of the critical path?

For example, we could implement a `prefetch` function:

```
// create file frontend/utils/prefetch-critical-resources.tsx

let sidebarCache = undefined;
let tableCache = undefined;

export const prefetch = () => {
  if (!sidebarCache) {
    sidebarCache = fetch('http://localhost:5432/api/sidebar');
  }
  if (!tableCache) {
    tableCache = fetch('http://localhost:5432/api/statistics');
  }

  return {
    sidebar: sidebarCache,
    table: tableCache,
  };
};
```

Here, we create the "fetch" promises when the function is called and cache them into variables for further reuse.

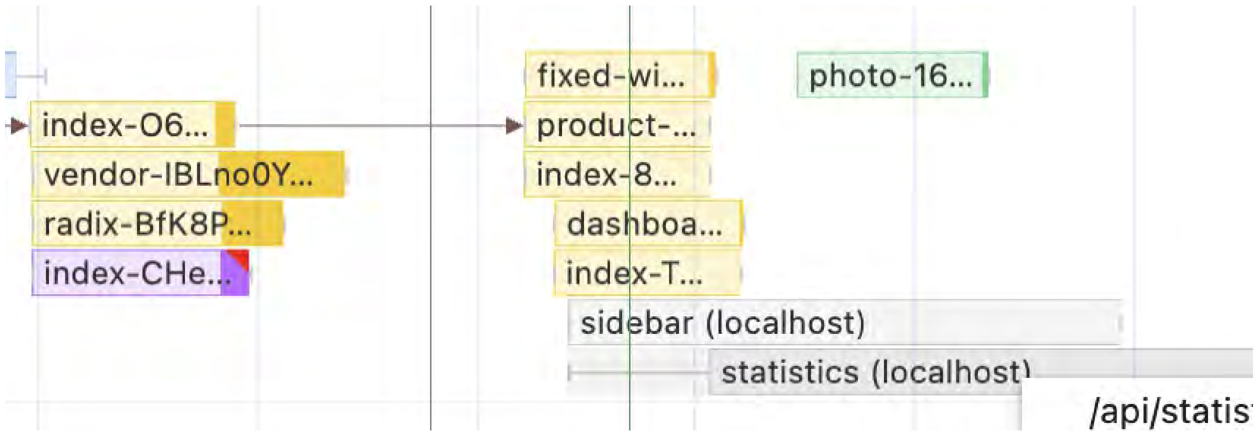
Then call that function in `useEffect` in `App.tsx` - this is our "entry" to the entire app that is on the critical path and therefore not lazy-loaded:

```
// FILE: src/chapter10-data-fetching/frontend/client-fetch

export default function App() {
  // everything else is the same

  useEffect(() => {
    prefetch();
  }, []);
```

And then use the cached promises in the Sidebar (and the Table in exactly the same way):

```
// no more preloading here

const PrimarySidebarPrimaryGroupSPA = ({ ...props }) => {
  // exactly the same code as before

  useEffect(() => {
    const fetchSidebarData = async () => {
      // use critical prefetch promise here
      const response = await prefetch.sidebar;

      // everything else is exactly the same
    };

    fetchSidebarData();
  }, []);

  return; // same stuff as before
};
```

This will give us a much more interesting performance picture:



The beginning of both fetches moved even more to the left. Sidebar items now appear at around 1.2 seconds. 200 ms improvement for "free". The `statistics` endpoint also moved to the left. Kinda.

If you remember, in one of the previous chapters, we discussed the difference between HTTP 1 and 2/3 protocols. With the HTTP 1 protocol, Chrome has a limit of only six parallel connections to the same domain.

With data-fetching, it's especially important to understand that difference. Because, in the case of static resources deployed to a CDN, chances are, you're on the 2/3 version already, so it doesn't matter that much. However, if you're building your own server, you'd need to make sure that it supports HTTP 2/3. Or be very careful with data prefetching. The more non-critical prefetching you add, the more chances you unintentionally delay something important.

Okay, so let's assume that our servers are HTTP 2/3, and the parallel requests are not much of a problem. The only other thing we can do to move the fetches a bit left while staying within the React application is to move them outside of the App component. As with moving them outside of the lazy component, it will win us the time it takes the browser to parse and compile the downloaded JavaScript, but no more.

If we want to move even further left, we'd need to step away from the React environment to the wild west of native JavaScript. We'd want to trigger fetches before or at least in parallel with the rest of the JavaScript we're downloading. So they would have to be injected into our `index.html` file via a separate `script` tag like this:

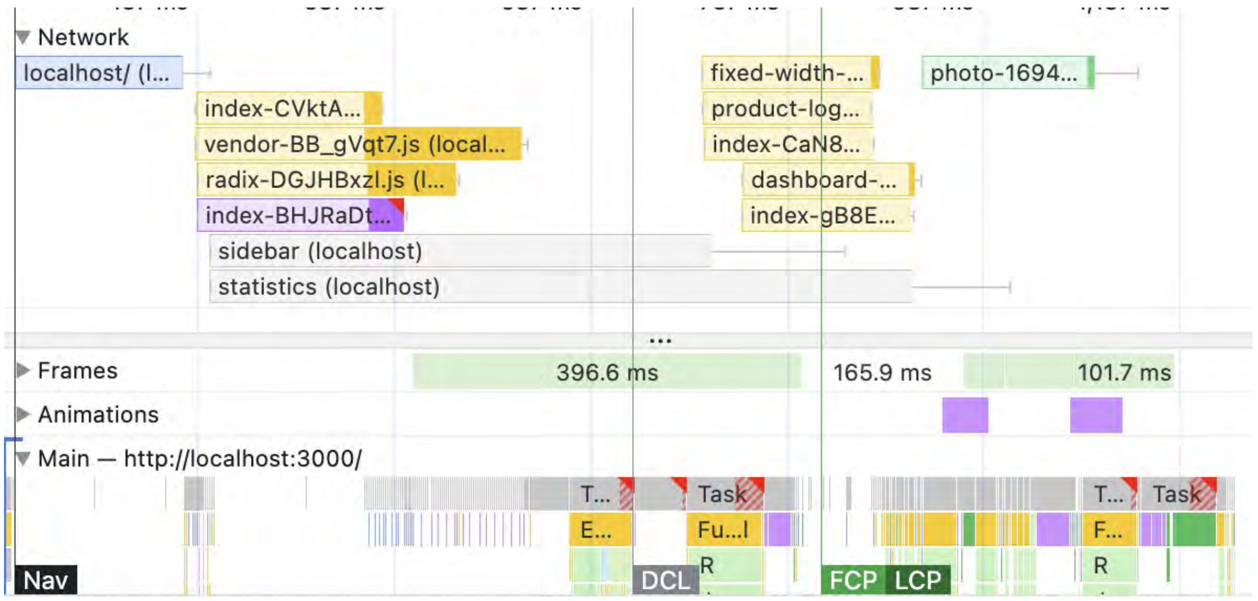
```html
// FILE: src/chapter10-data-fetching/frontend/client-fetch/index.html
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Chapter 10</title>
    <script>
        window.__PREFETCH_PROMISES = {
            sidebar: fetch('http://localhost:5432/api/sidebar'),
            table: fetch('http://localhost:5432/api/statistics'),
        };
    </script>
  </head>
  <body class="bg-blinkGray100">
    <div id="root"><!--ssr--></div>
    <script type="module" src="main.tsx"></script>
  </body>
</html>
```

And then in React, you'd be able to pick them up from the `window`:

```ts
let sidebarCache = window.__PREFETCH_PROMISES?.sidebar;
let tableCache = window.__PREFETCH_PROMISES?.table;

// The rest of prefetching is exactly the same
```

If you want to try it out on the study project, make sure to either guard the use of the `window` against SSR, like we did in the dedicated chapter. Or temporary switch to `createRoot` from `hydrateRoot` in the `main.tsx` file. Otherwise, it will cause hydration errors.

The end result should look something like this:



`sidebar` and `statistics` fetch calls start at the very beginning now, as soon as HTML is downloaded. When everything lazy is finally loaded, the data is already there. So, there will be almost no loading state for either the Sidebar items or the Table. The data will be rendered as soon as the JavaScript is ready. Best possible speed.

There is a cost to it, of course. The API calls are "stealing" a bit of the bandwidth now, which results in LCP moving to the right and hovering over the **790 ms** mark. Used to be 730 ms. We made the table and the sidebar items faster, but slowed down the LCP.

There is also a matter of dealing with the `window` in the app, which introduces complexity to the SSR if you have SSR. Not to mention that we're again prefetching *everything*, regardless of whether it's used on a page or not. And implementing more logic to differentiate what should and shouldn't be prefetched will require more code, which potentially can slow down the critical resources even more.

So I would treat this technique more like an exercise to understand the relationship between when the data fetch is initiated and when the result is rendered. Not something to bring to production, especially these days, when we have plenty of other options to try out first.

Data Fetching with Libraries

Speaking of production. The code above is *really* not ready for real life.

In real life, you'd want to be able to prefetch only the data needed on a page, plus maybe the data needed on pages you can navigate to from the current page if you're feeling fancy. There is no reason to prefetch something for Settings from the Login page, for example, as it happens now.

Plus, you'd want to be able to refetch the data manually if you suspect that it changed. Plus, maybe refetch it automatically. Or support paginated requests. Or abort a fetch from time to time. Not to mention, you'd want a cleaner loading and error state - there is no reason to implement them manually all the time.

What I'm saying here, in real projects, if you're fetching on the client, you'd want to use a library that handles all of this. The major player and the most popular option after the simple `fetch` is Tanstack Query[150]. The second most popular, if we ignore GraphQL here, is SWR[151], according to the 2024 survey[152].

Both of them will give you a `useLibraryFetch` hook (useQuery[153] in Tanstack and useSWR[154] in SWR) to which you'll be able to pass the URL of the API in one way or another. In return, the hook will give you `data`, `loading`, and `error` state. So, converting the existing code to any of those libraries will require very little change:

```
const PrimarySidebarPrimaryGroupSPA = ({ ...props }) => {
  // no more useEffect mess, just this little beauty
  const { data, loading } = useLibraryFetch('/my/url/in/some/form');

  return (
    <NavigationGroup header="general" {...props}>
      {loading ? <SidebarSkeleton /> : renderSidebar(sidebarData)}
    </NavigationGroup>
  );
};
```

However, regardless of the library, it's important to understand that all of them are still JavaScript and React. So all of them will be bound by the same rules when it comes to data fetching and prefetching.

"Regular" data fetching will be triggered when a component that requests the data is mounted. This is pure React. Until the component is mounted, its function is never called, and anything inside doesn't exist.

Switching between `loading` states, setting `data` or `error`, whether it's hidden behind a library or not, is a state update, and as a result, a re-render. Again, this is pure React. There is no other way here.

Everything related to prefetching will be subject to the same restrictions and downsides. This is the nature of JavaScript and its relationship to the browser.

Additional challenge

1. Migrate the study project from custom `fetch` to Tanstack[155].
2. Implement the different prefetching strategies we used for `fetch`: inside the lazy component, outside the lazy component, inside the `App` component, outside the `App` component, outside of React.
3. Does the performance picture differ compared to normal `fetch`?
4. Do the same migration and tests with SWR[156]. What about performance now?

Data Fetching and SSR

Okay, so it seems that we squeezed everything possible from the client-side data fetching. But what about the server-side? Didn't we just discuss a few chapters ago that SSR is very popular these days? Plus, isn't it common knowledge that fetching data on the server is much faster? Can we also benefit here?

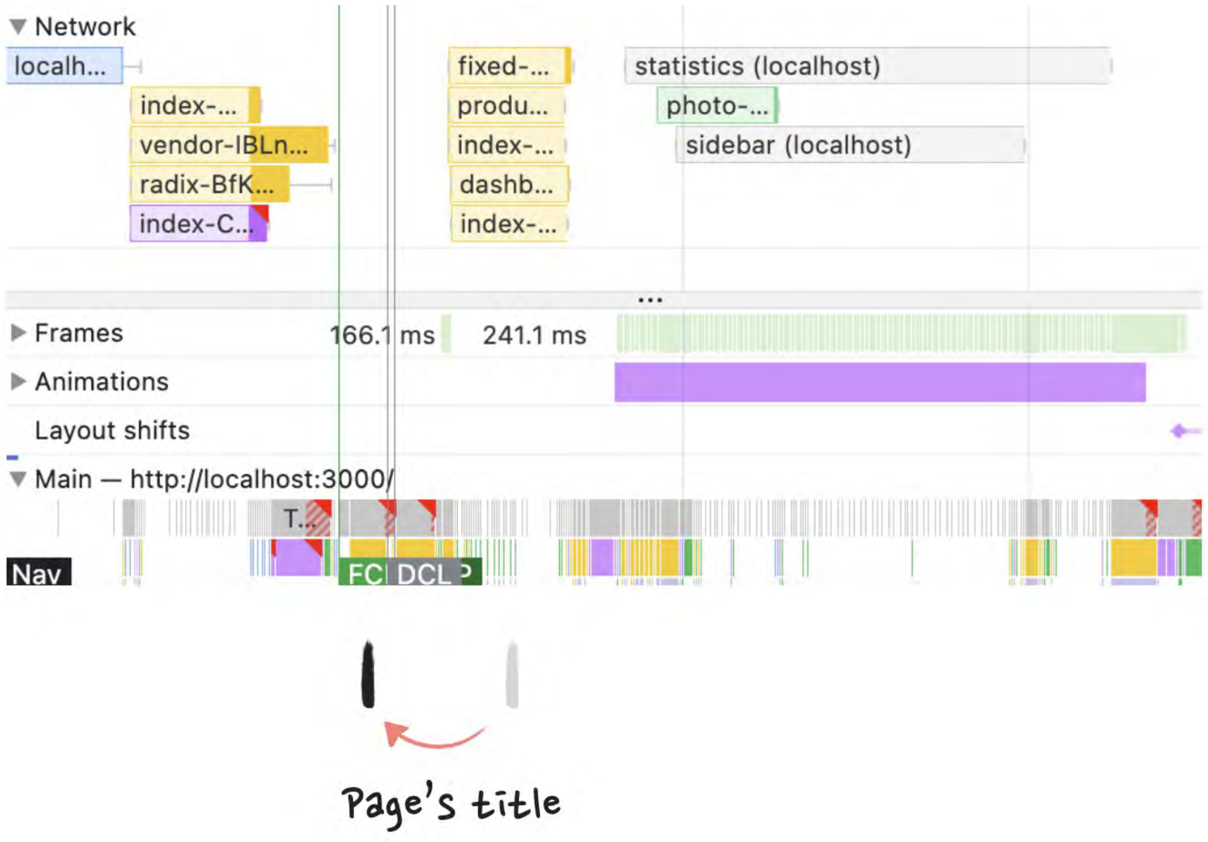
Fair questions. Let's take a look.

SSR and Client-Side Data Fetching

Let's start by introducing the simple SSR with no additional changes to our website. Get rid of all the prefetching we did for a slightly more visible picture and uncomment it in our backend implementation:

```
// FILE: src/chapter10-data-fetching/frontend/client-fetch/server/index.ts

app.get("/*", async (c) => {
  const html = ... // keep the same

  // uncomment this part
  return c.html(simpleSSR(c, html));
});
```

The performance profile for this situation barely changes:



The "critical path" pre-renders now, and the LCP number went down from **730 ms** to **480 ms**. We can see the page's title and the basic page placeholder a bit sooner. But we still have requests for `statistics` and `sidebar` on the frontend, and the table and the sidebar items show up only when those requests have finished.

This is happening because we fetch data inside the `useEffect` hook, which is client-only. So, for the dynamic data to show up, we still need to wait for the fetch requests to complete on the client. The server didn't help us even a bit here.

However, now that we've added a server to the equation, we can experiment some more!

SSR and Server-Side Data Fetching

If you remember from the SSR chapter, our entire "server" implementation is just three lines:

- Read the initial HTML file.
- Generate more HTML with React's `renderToString`.
- Replace the `<!-- ssr -->` comment with the generated HTML.

```
app.get("/*", async (c) => {
  const html = fs.readFileSync(`/dist/index.html`).toString();
  const reactHTML  = renderToString(<App />);
  const finalHTML = html.replace("<!--ssr-->", reactHTML);

  return c.html(finalHTML);
});
```

So nothing stops us from fetching the Sidebar and the statistics there as well.

```
app.get("/*", async (c) => {
  // I can just fetch all the data right here
  const sidebarPromise = fetch(`/api/sidebar`).then((res) => res.json());
  const statisticsPromise = fetch(`/api/statistics`).then((res) => res.json());

  // in parallel!
  const [sidebar, statistics] = await Promise.all([
    sidebarPromise,
    statisticsPromise,
  ]);

  ... // the rest is the same
});
```

The only question now is how to transfer this data to the React app? 🤔 Normally, we'd need to do that in two steps.

Step 1: Data as Props to renderToString

First, we'd need to introduce props to the `App` component. Good regular props that are passed in the good regular React way:

```
const [sidebar, statistics] = await Promise.all([
  sidebarPromise,
  statisticsPromise,
]);

// Yep, just your normal props here
const reactHTML = renderToString(<App sidebar={sidebar} statistics={statistics} />);
```

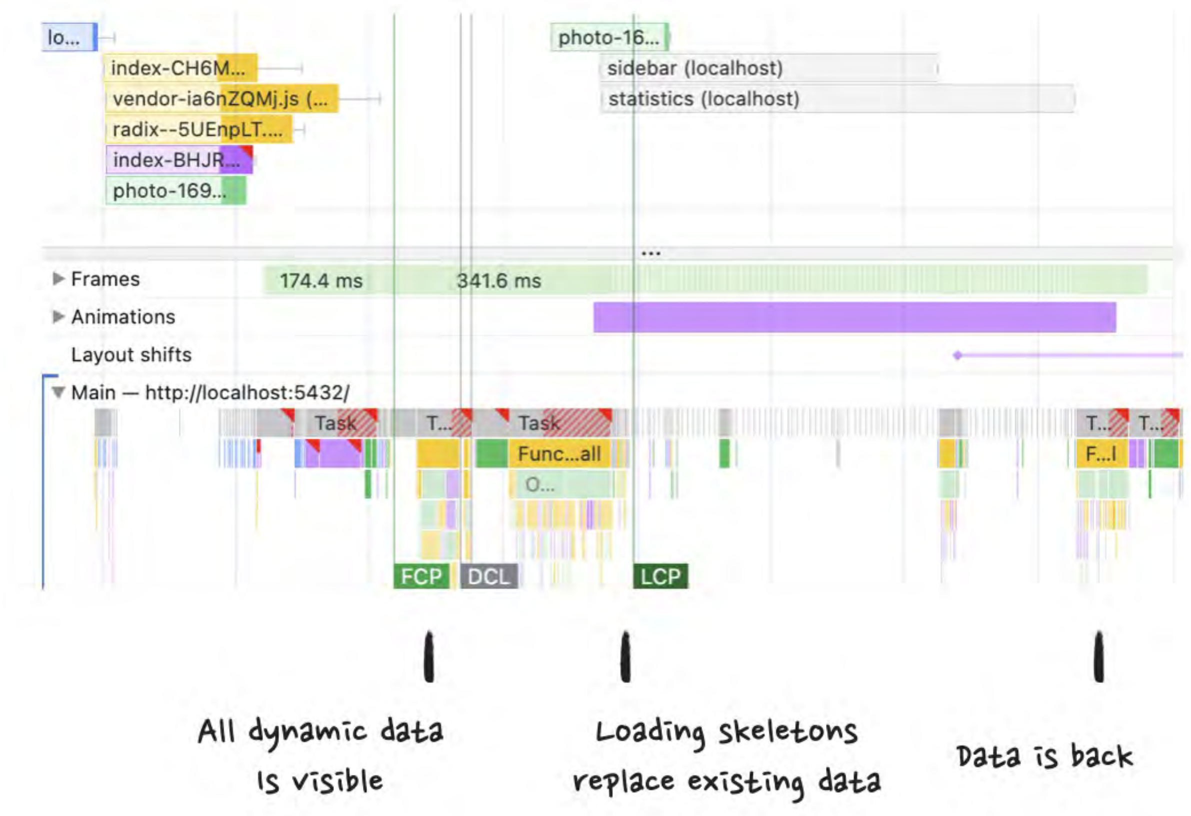
Then, we'd just need to pass that data to the relevant pages, then a bit of props drilling, and then use this data to render whatever we need as any other prop. In our case, it will go into the initial state:

```
const PrimarySidebarPrimaryGroupSPA = (props) => {
  // props as initial state
  const [sidebarData, setSidebarData] = useState(props.sidebar);
```

This would take care of the server part and the initial server HTML. It will be relevant only for the bits that are not suspended, i.e., stuff that is on the critical path. Because everything that is wrapped in Suspense will default to the fallback. So to see something meaningful in the study project, you'd need to get rid of Suspense and all the lazy loading for now. We'll discuss a bit further how it happens in real life.

If you do this and refresh the page, you'll see a very interesting picture. First, the page loads with navigation items and the statistics table present. Then they disappear, the loading skeleton shows, and after a while, they appear again.

You'll see this behavior much clearer if you record the performance profile with the "Screenshots" checkbox enabled. It will be something like this:



That's ~~garbage~~ a really bad UX. We definitely need to fix it.

Step 2: Pass Data to the Client

As you hopefully remember from the previous chapter, when we're doing SSR, we need to do "hydration" to initialize the client-side part of the app properly. It usually happens inside the `main.tsx` file and takes the form of using `hydrateRoot` on the div with the `root` id:

```
hydrateRoot(document.getElementById('root')!, <App />);
```

In this case, React will iterate over the preexisting DOM elements that we supplied with the SSR and attach event listeners to the relevant elements to make the page interactive. If hydration didn't happen or failed for some reason, React has no way of knowing that it can reuse the elements. So it wipes them all out, re-creates the entire interface from scratch, and injects it inside the "root" div again.

Which *feels* like exactly what is happening here. Except it's not on the DOM level, but on the level of the app itself.

The reason for this behavior can be found in the `main.tsx` file:

```
hydrateRoot(document.getElementById('root')!, <App />);
```

On the server, we passed the initial props to the `<App />`. But not here! 🤷‍♀️ React has no way of knowing that we actually already have the data. So it wipes out the interface, reinitializes the app from scratch, kicks in yet another fetch on the client from `useEffect`, and shows us the loading skeleton while we wait. The already downloaded data from the server disappeared into the void.

We need initial props on the client as well, the same as we had in `renderToString`. Except, on the client, we don't have that information yet. We need to inject the data we fetched from the server into the client somehow. The simplest way to do that is to inject it via a `script` tag by attaching the data to the global `window` variable.

```
<script>
  window.__SSR_DATA__ = {
    sidebar: '...', // sidebar data from the promise
    statistics: '...', // statistics data from the promise
  };
</script>
```

And then on the server side, when we're replacing the `<!-- ssr -->` comment with HTML, we'd add that mini-script to the `body` tag:

```
app.get("/*", async (c) => {
  ... // same as before

  const reactHtml = renderToString(
    <App sidebar={sidebar} statistics={statistics} />,
  );

  const htmlWithData = `
    <div id="root">${reactHtml}</div>
    <script>window.__SSR_DATA__ = ${JSON.stringify({
      sidebar,
      statistics,
    })}</script>`;
  const finalHtml = html.replace(
    '<div id="root"><!--ssr--></div>',
    htmlWithData,
  );

  return c.html(finalHtml);
});
```

You can see the full working implementation if you uncomment the following lines in the `server/index.ts` file:

```
// FILE: src/chapter10-data-fetching/frontend/client-fetch/server/index.ts

// uncomment this line
return c.html(simpleSSRWithHydration(c, html));
```

This is also called "hydration", by the way. Only it's about hydrating the data, not JavaScript event handlers.

Step 3: Read Data From the Client

And the final step: accessing the data. If you open the Elements panel in Chrome and look at the `<body>` tag, you should see at the very bottom our mini-script with the data inside. So now it's just a matter of accessing it via `window.__SSR_DATA__` somewhere in the frontend code.

Something like this:

```
hydrateRoot(
  document.getElementById('root')!,
  <App ssrPath="" sidebar={window.__SSR_DATA__?.sidebar} statistics=
{window.__SSR_DATA__?.statistics} />,
);
```

We already introduced props to the `App` component when we did SSR, so we're good to go.

Now you should be able to see the prerendered page in all its glory: the dynamic table and sidebar items show up with the rest of the page in one smooth motion. No more beautiful but mildly annoying skeletons.

The Cost of SSR Data Fetching

At what cost, though? Setting aside the problem of incredibly fragile, unscalable, and just ugly code 😅 that should never hit production in its current form, there is a performance issue here.

If you record the performance profile now, you should see that:

- The blue HTML bar in the "network" section is very long.
- The LCP value dropped from ~480 ms to **1.45 s**.

Almost a second worse. That is quite the performance hit! You know the reason for this by now, right?

```
app.get("/*", async (c) => {
  const sidebarPromise = fetch(`http://localhost:5432/api/sidebar`).then((res) =>
res.json());
  const statisticsPromise = fetch(`http://localhost:5432/api/statistics`).then((res) =>
res.json());

  // We await! For a whole ~700 ms
  const [sidebar, statistics] = await Promise.all([
    sidebarPromise,
    statisticsPromise,
  ]);

  const reactHtml = renderToString(
    <App sidebar={sidebar} statistics={statistics} />,
  );

  ... // the rest is the same
});
```

We moved the data fetching to the server, and we have no choice but to wait for it before passing it to the `renderToString`, which is just a regular JavaScript function with simple input and output. The data just happens to come from a slow data source and takes ~700 ms to load.

Oooops. There is always a price.

But it doesn't mean that data fetching on the server is necessarily bad. The 700 ms delay is a bit extreme, and even with it, the full page load takes 1.45 s, which is still within the "green" zone performance-wise. And if the data fetching is much faster, the 50 - 100 ms delay might be completely unnoticeable and totally worth it for the fully prerendered entire page with no loading states. Especially if SEO is your priority.

The question of the ugly code would need to be resolved, though 😅

SSR Data Fetching for Real

All those naive implementations were to understand the core concepts that are usually hidden inside frameworks. However, none of the above is really suited for "production" use.

At the very least, you'd need to split your server code into pages, same as the client, and fetch only the data that is needed per page. There is absolutely no reason to fetch the statistics table and navigation items on the server with a 700 ms delay when you open a Login page, for example.

You'd need to restore lazy loading and Suspense if you want to preserve the SPA navigation while pre-rendering the initial load. And that is its own pain for the SSR - you'd probably need to generate a single "server" bundle and modify lazy loading to understand that.

I'm not even going to mention dev experience. Right now, it's non-existent. If you did all the exercises above, you probably hate me by now for the constant rebuilding of stuff, rather than a nice dev mode's hot reloading.

This manual `renderToString` implementation is good for understanding the concept and might be useful if you want to pre-render cheaply an existing one-page app. But for anything more complicated, you'd end up building your own SSR framework.

So, unless you work for a company that explicitly builds an SSR framework as part of its business strategy (which could happen, Shopify with React Router and Vercel with Next.js are examples), you'd likely be much better off using an existing solution.

Tanstack[157] supports SSR out of the box and has a bunch of examples on how to set it up. React Router[158] in the "Framework" mode supports SSR as well. And then, of course, there is Next.js[159].

Next.js was an SSR framework from the beginning, and by now, it has evolved into two "variants": App Router[160] and Pages router[161]. App Routers is the latest and greatest bells and whistles, with Streaming and Server Components included. We're going to try it out in the next section.

Pages Router is the "classic" SSR. So if you want a bit of a challenge, try to migrate the Study Project to the Next.js Pages Router[162]. If nothing else, just to appreciate more what a "proper" framework gives you out of the box from the dev experience perspective, compared to our hacky solution.

If you've done the migration to Tanstack in the previous chapter, the steps are exactly the same:

1. Figure out how to set up the framework.
2. Copy-paste all the relevant configs like TypeScript aliases and Tailwind.
3. Figure out routing and render existing pages for each route.
4. Figure out the "critical path" and layouts in the framework, and render the AppLayout there.
5. Replace our custom `Link` with the `Link` component of the framework.

Plus, you'd need to figure out how to fetch[163] our data on the server side and pass it to the client using the framework.

If you get stuck at some point or just want to explore the code without the pain of setting it up yourself, you can find it in `src/chapter10-data-fetching/frontend/next-pages`. I only did the dashboards page there, since the rest of them are irrelevant to the topic. And only the table props, because pushing props to the sidebar would need a bit too much copy-pasting of files with one single change, and the Study Project is way too big already. All you'd need to do to make it work is to prop drill the `sidebar` object to where it's needed.

If you look at the code, the most interesting part there is this:

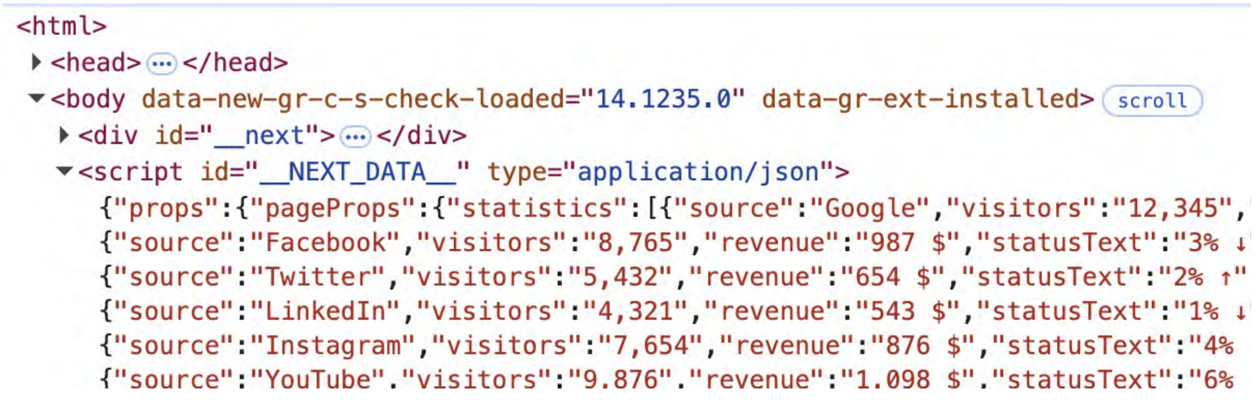
```
// FILE: src/src/chapter10-data-fetching/frontend/next-pages/pages/index.tsx
```

```
export const getServerSideProps = async () => {
  const sidebarPromise = fetch(`http://localhost:5432/api/sidebar`).then((res) =>
res.json());
  const statisticsPromise = fetch(`http://localhost:5432/api/statistics`).then((res) =>
res.json());

  const [sidebar, statistics] = await Promise.all([sidebarPromise, statisticsPromise]);

  // Pass data to the page via props
  return { props: { statistics, sidebar } };
};
```

This is *exactly* what we did in our custom backend. I just copy-pasted the fetch logic. Only this time, it works with dev mode, integrates with the router without weird hacks, co-locates with the app's code, and hides the `renderToString` part.

If you build and start the project, the performance profile picture will be pretty much identical. There will be a loooong line of the blue HTML because we're waiting for the fetch. Then the page will appear with the dynamic data already present. No skeletons.

And now the best part: open the Elements panel in Chrome, find a `script` tag with `id="__NEXT_DATA__"`, and take a peek inside. It Should be something like this:



```
<html>
 ▶<head>⋯</head>
 ▼<body data-new-gr-c-s-check-loaded="14.1235.0" data-gr-ext-installed> scroll
   ▶<div id="__next">⋯</div>
   ▼<script id="__NEXT_DATA__" type="application/json">
      {"props":{"pageProps":{"statistics":[{"source":"Google","visitors":"12,345",
      {"source":"Facebook","visitors":"8,765","revenue":"987 $","statusText":"3% ↓
      {"source":"Twitter","visitors":"5,432","revenue":"654 $","statusText":"2% ↑"
      {"source":"LinkedIn","visitors":"4,321","revenue":"543 $","statusText":"1% ↓
      {"source":"Instagram","visitors":"7,654","revenue":"876 $","statusText":"4%
      {"source":"YouTube","visitors":"9,876","revenue":"1,098 $","statusText":"6%
```

Again, exactly the same thing we implemented manually. No more magic! It's time to form a React Mythbusters team. We're qualified enough now, I'd say.

But before doing that, it's time to finally talk about React Server Components.

Streaming and React Server Components

Server Components Intro

Finally! Server components, the most hyped feature of React in the last few years. So, what's the point of them? What exactly do they solve? And why is streaming here?

One of the semi-obvious disadvantages of SSR is that it doesn't really affect the bundle size in any way - everything that we write in React always ends up in the browser at one point or another. As we know from the previous chapters, this could lead to a situation where a page is already rendered and visible, but it's a pure static HTML page with no interactivity. Leave this gap long enough, and the users will think that the website is terribly broken.

This happens because we used to think of React as a "frontend" framework, with the server part being an afterthought purely for pre-rendering purposes. With this mindset, *everything* that you write in React needs to be in the browser. React doesn't understand that some parts of the app are purely static and could be left alone after their HTML from the server is received.

The introduction of Server Components makes this mindset do a full U-turn (as it may or may not happen every two years in the React ecosystem). Now we can adopt a mindset that *nothing* we write in React we actually need in the browser 😅. Joking, of course, it's more like - if there is something we really need, we'd mark it explicitly. By default, assume server-only for everything.

Implementing proper support for this, however, is hard. In fact, it's so hard that even today, the synonym for "Server Components" is "Next.js". Among the top three most popular ones (Next, React Router, and Tanstack & Vite), Next is the only one that implements them.

So, in order to try them out, apply them to the study project, and see whether they improve anything, we kinda have to do that in Next.js. And not only Next.js, but its latest "App Router"[164] feature. If you don't want to set up App Router from scratch, I've already implemented it in the `src/chapter10-data-fetching/frontend/next-app-router` Study Project.

Exploring Server Components

Let's explore now. We'll need a completely empty page to start with. So, regardless of whether you set up your own or use the provided Study Project, get rid of everything in the `src/app/page.tsx` file and render a button component from `frontend/components/button` instead:

```
import { Button } from '@fe/components/button';

export default function App() {
  return <Button>I'm a button</Button>;
}
```

Build and start the project, and you should see the button on the screen. If you're using the provided Study Project, use the following commands:

```
npm run build --workspace=chapter10-next-app-router
npm run start --workspace=chapter10-next-app-router
```

By default, this button will be your very first Server Component.

Server Components Stay on the Server

The very first advantage of Server Components that you might hear is that they reduce bundle size, which is usually good for performance. So let's first verify that this is indeed the case.

The easiest way to do that is the "naive" way. Open the code of the `Button` component (`frontend/components/button/index.tsx`), grab any of the Tailwind strings at the beginning to reliably identify the Button's code, and search for those strings inside the `.next` folder - this is everything that Next.js builds for you.

You'll see that it's included three times in files inside the `.next/server` folder, but that's it. It's reasonable to assume that nothing from the `server` folder is served to the client.

And it's easy to verify that this is indeed the case: just start the project and open the Network panel in Chrome Dev Tools. You'll see a bunch of scripts downloaded, but all of them are coming from `_next/static/chunks` - this is our client app. And the Button is not there.

Now we just need to verify that the Button appears in the client bundle if the app stops being a "Server" component. To do that, add the `use client` directive at the very top of the `page.tsx` file:

```
'use client';

import { Button } from '@fe/components/button';

export default function App() {
  return <Button>I'm a button</Button>;
}
```

Rebuild the app and again search for the Button's string. This time, it should appear inside the `.next/static/chunks/app/page-xxx.js` file. Yep, it's one of those that is sent to the browser! You should be able to see this file in the Network panel in Chrome, and when you look at its content there, you'll see the button's code.

Okay, that's some powerful magic right there. But how does it work, and are there any downsides?

`use client` vs Nothing

The first and most intuitive concept here is "client" components. These are our "traditional" components, your basic React code that we've been writing until now, which are interactive, bundled together, and sent to the browser. We turn code into a "client" component by explicitly adding `use client` at the top of the file.

After that, the terminology becomes really confusing. In the world of Server Components, the "default" component is Server. We turn a component into a Server component by... doing nothing. At the very beginning, our App with the Button was a Server Component before we added a `use client` directive.

That leads to a very interesting consequence. Look at the code for the Button itself here: `frontend/components/button/index.tsx`. There are no directives at the top, and we didn't change anything there when we turned our app into the Client component. But the fact that we introduced `use client` to the *parent* component made all the difference.

So, components without the `use client` directive can still be Client Components. Their parents will decide. It's possible that in one case a component stays as a Server Component, but in another case, it accidentally ends up in the browser. Exactly as it happened with the Button.

To make things even more complicated, there is also the `use server` directive[165]. Which is used for *server functions* that can be called from the Client Components. Essentially, it's a fancy way to create a REST API and slightly out of scope here.

How the Magic Works - the Theory

As you might know, a normal React Component is a function that returns a React *Element*. Every time we "render" a component, in reality, we create an Element, which will be picked up and converted into something useful by React when it reaches this Element in the render tree.

```
const Component = () => ... // this is a component

<Component /> // this is an Element
```

The Element in this form is syntax sugar for an object, which has the `type` of the element and a bunch of other attributes like props, children, and key.

```
{
  type: Component,
  ... // bunch of other stuff
}
```

So our Button from the App above will be represented by an object like this:

```
{
  type: Button,
  props: {
    children: "I'm a button"
  },
  ... // other stuff
}
```

If this concept is completely new to you, you might need my other book called

"Advanced React"[166]- half of it is dedicated to understanding all of this on a very deep level and why it matters for everyday life.

But for the purpose of this chapter, the thing that we need to know is that when we call React's `render` at the very root of the app, React goes through all the components on the render path starting from the root and constructs that tree of objects. From that tree, it can create the "real" DOM elements and inject them into the page.

Normally, this happens on the client, and this is the basis of Client-Side rendering. If we add Server-Side rendering to the mix, we'll have the DOM elements prefilled already. In this case, React still goes through the tree, constructs the objects, but instead of creating new DOM elements and injecting them onto the page, it reuses the existing DOM nodes and just attaches the needed event listeners to the interactive elements.

So if we look at the simplest App we had:

```
export default function App() {
  return <Button>I'm a button</Button>;
}
```

After all the processing and tree-reconstructing, it will be roughly represented by an object like this:

```
{
  type: "button", // the Button component renders a "button" element
  props: {
    children: "I'm a button",
    className: "inline-flex gap-2 ..." // Button injects quite a lot of classes
  }
}
```

Now, let's add Server Components to the mix. With Server Components, we generate that tree of objects *in advance*, on the server, saving some valuable client processing time. If we do that and *inject* that object into the page for the client to pick up, all React would need to do is construct the DOM elements from the already existing tree. All the work of creating that tree first can be skipped.

We can inject that tree in the same way as we did with SSR data: via a script tag, attached to a global variable.

```
<script>
  window.__SERVER_COMPONENTS = JSON.stringify(...) // add that tree here
</script>
```

Now React just needs to read an object from the `window` and convert it into DOM nodes without much thinking.

How the Magic Works - Verifying the Theory

The content of this book pretty much follows the "trust but verify" principle. No reason for this part to be an exception - let's see whether the theory matches the practice.

It's a little tricky with Next.js, as it does a lot of under-the-hood magic that's quite hard, if not impossible, to turn on and off. For example, it's SSR through and through, with no way to opt out. Considering that Next.js is pretty much the only solution for Server Components, this leads to a big confusion that Server Components are a synonym for SSR, or at least can't exist without SSR.

But we know the theory now, and know how SSR works, so we can hack our way through that.

First of all, let's check whether the theory matches the reality in the default state. Revert the App to be just a button, without the `use client`:

```
import { Button } from '@fe/components/button';

export default function App() {
  return <Button>I'm a button</Button>;
}
```

Build it and take a look inside the `.next/server/app` folder. By default, Next.js pre-renders at build time everything, which is another issue we'll have to disable in the next step. But for now, it serves us nicely.

Open `index.html` now. Two interesting things to notice there. First, you should be able to see the rendered button:

```
<button type="button" class="inline-flex ...">I'm a button</button>
```

That's our SSR at work. The button itself was pre-rendered and injected.

Underneath, you should see a bunch of `<script>` tags with `self.__next_f` code, with some weird but slightly familiar data being pushed to it. That's our Server Components, serialized in a specific React-understandable way! If you search for "I'm a button" in that serialized mess, you'll see that it's indeed there, and it looks almost like the object from the theory.

```
{\"type\":\"button\",\"className\":\"inline-flex ...\",\"children\":[\"$undefined\",\"I'm a button\",\"$undefined\"]}]
```

An object with the `type` "button", lots of classNames, and the `children` with "I'm a button" inside.

Now, we just need to eliminate the influence of SSR. The theory states that if Server Components were injected, React would be able to pick them up and recreate the DOM nodes from those objects.

To do that, manually remove the pre-rendered button in the generated `index.html` and restart the server. The code should be the `<body>` tag followed by some comments with `$` inside (probably something Next.js injects), followed by all the `<script>` tags.

Now, start the server and open the website. The button should be there! To prove 100% that it's the correct file, disable JavaScript on the page and refresh the page. The button will disappear, proving that it was created on the client.

For even more fun, go back to the generated `index.html`, restore the SSR'd button, but change the text inside it to "I'm an SSR button with some very long text". Restart the server and refresh the page with and without JavaScript. Without JavaScript, the button stays with SSR text. With JavaScript, it changes after React is loaded. If you slow down the CPU and Network and record a performance profile, you should even be able to trace the exact moment it happens: after JavaScript is downloaded and executed.

Why Exactly Were We Doing It?

While all of the above is quite fancy, the point of going through such a paradigm shift and so much complexity is still not exactly clear. What was the point of all of this, exactly?

First, the most obvious, although the hardest to see any benefit, IMHO, is the bundle size reduction. If the Button from the example above weighs ~1 MB, that's one MB that would not be included in the client bundle.

Why is it hardest to see benefits? Because in reality, it's highly unlikely that your button will weigh 1 MB. From the bundle size investigation chapter, you already know that most of the time, it will be various small-ish duplicated or unnecessary libraries that just compound. Plus, considering that any component may or may not end up in the client bundle, depending on *any* of the parents in the entire render tree... For a large app with more than one developer working on it, it will be almost impossible to control.

There is a bigger benefit, however. Since Server Components run on the server and only their return matters, they can have access to stuff Client Components cannot dream of. Like a file system, for example. We can totally do this, and it's completely valid React now:

```
export default function App() {
  const json = JSON.parse(fs.readFileSync(path.resolve('test.json')).toString());

  return <Button>I'm a button of id: {json.id}</Button>;
}
```

Assuming that `test.json` exists and has this content:

```
{
  "id": "1"
}
```

Although, wait a second. Didn't I just say that any component may or may not be a Client Component depending on the parent? What will happen if this component ends up in the client bundle?

Okay, the statement wasn't 100% accurate. That kind of shenanigans will just break your build with `Module not found: Can't resolve 'fs'`. So, some components can't be Client components. Access to the file system or other Node-specific APIs is one way to ensure that the component is server-only.

Another way is to make it asynchronous. Asynchronous components can't be Client Components (at least at the moment). You'll see errors in dev mode if you try to do that. With asynchronous components, we can do pretty much anything that requires asynchronous operations, including data fetching.

Remember how we fetched our sidebar and statistics data in the Data Fetching and SSR section? It's pretty much the same, only now it's not limited to the page's entry and can be inside any React component, as long as it's a Server Component:

```
export default async function App() {
  const sidebarPromise = fetch(`http://localhost:5432/api/sidebar`).then((res) =>
res.json());
  const statisticsPromise = fetch(`http://localhost:5432/api/statistics`).then((res) =>
res.json());

  const [sidebar, statistics] = await Promise.all([sidebarPromise, statisticsPromise]);

  return (
    <>
      <p>
        Sidebar data:
        {JSON.stringify(sidebar)}
      </p>
      <p>
        Statistics data:
        {JSON.stringify(statistics)}
      </p>
    </>
  );
}
```

No more props drilling, the data fetching is co-located with the React code that uses it. If you load that page and open the Network panel, you'll see that there are no requests for this data on the client. It just appears there.

You might've also noticed that the page loads instantly. But weren't those endpoints somewhat slow? This is again where Next.js makes it harder (or easier, depending on your point of view). Next.js is aggressively pre-building and pre-caching everything, including data fetching in Server Components. If you attempt to build the project without the backend API service running, you'd see the build fail. Next.js is trying to download this data in advance. You can see what it looks like if you open the `.next/server/app/index.html` again: everything is downloaded and injected in advance.

This could be great if the data is somewhat static or tightly coupled with the App itself: downloading everything in advance makes you independent from those endpoints, and the page will be super-fast. However, if this data is truly dynamic, like it's coming from some sort of CMS and managed independently from the app's release cycle, this might become problematic. Because, the only way to update that data is to rebuild and redeploy the project.

Luckily, in this case, we can opt out of this behavior by adding `export const dynamic = "force-dynamic";` at the top of the file.

```
// Add this to opt out of pre-rendering
export const dynamic = "force-dynamic";

export default async function App() {
 ... // everything is the same
}
```

Rebuild the app and open the `.next/server/app` folder. You'll see that the `index.html` file is gone: from now on, it will be generated only when you navigate to the website in the browser.

Do that, and you'll see the delay of the endpoints: the page is blank until the data comes through. Exactly the same behavior as we had with the SSR approach.

Which is confusing and infuriating, in a way. We went through all those troubles just to co-locate data fetching with components? Where are the performance benefits? 😳 A slight reduction of bundle size and a JavaScript task that may or may not happen doesn't seem like something that justifies the cost.

It's because there is another piece of the puzzle that actually unlocks those benefits. Streaming!

Exploring Streaming

Until this moment, our "server" part of the app, even when it was in the Server Components mode, resembled a simple waterfall. We trigger some data fetch on the server, `await` until the data is there, then feed this data to the components and send the result to the client.

However, we already know that there is such a thing as a Critical Path: the content of the page that the user absolutely must see as soon as possible. We don't really *need* to wait for all the slow data to show it to the user. Plus, our endpoints are different - one is slower than the other. Why exactly are we waiting for the slowest one to resolve to show the data from the less slow endpoint?

We already know that we can co-locate data fetching with the components that use that data, so waiting makes even less sense. If, for example, I refactor the mini-app to be this:

```
export default async function App() {
  return (
    <div>
      <h1>Welcome to the App!</h1>
      <Sidebar />
      <Statistics />
    </div>
  );
}
```

With the `Sidebar` and `Statistics` components being asynchronous Server Components:

```
// exactly the same code for Statistics
const Sidebar = async () => {
  const sidebarPromise = fetch(`http://localhost:5432/api/sidebar`).then((res) =>
res.json());

  const sidebar = await sidebarPromise;

  return <>sidebar: ${JSON.stringify(sidebar)}</>;
};
```

In this case, waiting for everything to finish fetching is even more unnecessary: in an ideal world, I would expect the `h1` part to be independent and show up immediately, and for `Sidebar` to be independent from `Statistics` and vice versa.

However, this doesn't happen.

This is because we haven't commanded our server to do what I described yet. And whether it's even possible or not depends on the server's implementation.

In the "traditional" SSR with `renderToString`, there would've been nothing I could've done, even if I used Server Components for fetching (assuming they work there). `renderToString` is just a JavaScript function: some data comes in, it processes it, async or otherwise, and returns some data. All in one go.

Luckily, Next.js is not "traditional" here and supports what is known as Streaming[167]. Streaming allows Node, which is the engine underneath the Next.js server, to process data gradually, in "chunks". Basically, your Netflix streaming vs downloading the entire movie in advance. Only applied to React rendering.

By the way, implementing streaming by yourself is not an easy task, even a hacky variation for study purposes. I swear, the amount of time I spent trying to do that is just embarrassing. But in the end, I always end up with a half-working version of Next.js, the necessity to include a dozen new concepts and a few new chapters just to cover "how". None of which would be relevant to web performance or writing customer-facing interfaces: the primary focus of this book.

This is why we're using Next.js here again. If the end result is going to be a not-very-understandable black box full of magic anyway, I could just use an existing magic right away and focus on how to use it rather than reimplementing it.

So, back to coding. Next.js is fancy and supports streaming. Streaming is the "ideal" situation that I described above: when it works, it would render the `h1` tag first, then the `Sidebar` and `Statistics` components independently, as soon as their data comes through.

```
// With streaming, we should see h1 first
// Then, Sidebar and Statistics independently
// When their data comes through
export default async function App() {
  return (
    <div>
      <h1>Welcome to the App!</h1>
      <Sidebar />
      <Statistics />
    </div>
  );
}
```

To make it work, however, it is not enough to just use Next.js (or any other framework). The implementation above has one fatal flaw: React still thinks that everything is on the critical path. We need to mark `Sidebar` and `Statistics` as "not that important, focus on something else first until they are ready".

You guessed it - we need to wrap them in the already familiar `Suspense`.

```
export default async function App() {
  return (
    <div>
      <h1>Welcome to the App!</h1>
      <Suspense>
        <Sidebar />
      </Suspense>
      <Suspense>
        <Statistics />
      </Suspense>
    </div>
  );
}
```

Now you should be able to see exactly the "ideal" scenario I described! The "Welcome" `h1` tag appears immediately, followed by the `Sidebar` data (the faster endpoint), followed by `Statistics`.

Where you put `Suspense` boundaries will determine the Streaming "chunks". Basically, one `Suspense` - one chunk. In the implementation above, we have two independent ones. I could also wrap both of the components in just one `Suspense`:

```
export default async function App() {
  return (
    <div>
      <h1>Welcome to the App!</h1>
      <Suspense>
        <Sidebar />
        <Statistics />
      </Suspense>
    </div>
  );
}
```

In this case, `Suspense` will wait for *both* `Sidebar` and `Statistics` to resolve before rendering them. So you'll see the "Welcome to the App!" text show up immediately, and then both the Sidebar and Statistics data at the same time.

We can also render one inside another. Because why not? From a React perspective, those are just components. But be mindful that since they are just components, they would have to follow the components' lifecycle rules. I.e., data fetching inside a component can't be triggered until the component ends up in the render tree. Which, in the case of Suspense, will happen only when the "parent" Suspense resolves itself. I.e., the fetch requests will be chained, and we created a regular data fetching "waterfall", only on the server.

```
// Nesting like this will create a data fetching waterfall
export default async function App() {
  return (
    <div>
      <h1>Welcome to the App!</h1>
      <Suspense>
        <Statistics />
      </Suspense>
    </div>
  );
}

const Statistics = async () => {
  const statisticsPromise = fetch(`http://localhost:5432/api/statistics`).then((res) =>
res.json());

  const statistics = await statisticsPromise;

  return (
    <>
      <Suspense>
        <Sidebar />
      </Suspense>
      statistics: ${JSON.stringify(statistics)}
    </>
  );
};
```

Final Measurements on the Real Page

That was quite a journey for the simple fetch requests! They started on the client inside `useEffect`, migrated to the server outside of React components, and finally ended up back inside React components, but *still* on the server. Wild.

But was it really worth it? From the performance perspective, of course, assuming we're willing to make the effort. There is only one way to answer that: measure and compare. We already have apps for the three of them:

1. Client-side fetching: `src/chapter10-data-fetching/frontend/client-fetch`
2. "Classic" SSR: `src/chapter10-data-fetching/frontend/next-pages`
3. Server Components: `src/chapter10-data-fetching/frontend/next-app-router`

All are implemented the same way, with the "My Dashboards" title being the only thing on the "critical path", and Sidebar items and the Statistics table data loaded dynamically. So, let's measure the LCP (when the title shows up) and the time for dynamic items to show up (which will be the same as the LCP for the classic SSR) for each of those apps.

	Critical Path	**Sidebar Items**	**Statistics Table**
Client-side Fetching	740 ms	1.4 s	1.8 s
"Classic" SSR	1.2 s	1.2 s	1.2 s
Server Components	410 ms	550 ms	1 s

Okay, even I must admit, this is pretty impressive. Server Components outperform everything, some of them even twice over.

Whether it's worth the pain or not, I'll leave for you to decide. ☺

What's Next?

Okay, so now data fetching in React, new and old patterns, should be clear. We can fetch anything now on the client, can optimize with smart prefetching, can move that data fetching to the Server if there is a need, know how to pass that data from the server back to the client, and even solved how Server Components work.

But all of this, and pretty much everything we talked about before, dealt with just one aspect of performance: initial load. Frankly, I'm getting bored of it, so let's switch to something else now. How about some Interactions performance? We briefly touched on that one in the SPA section, so maybe it's time to talk about it in a bit more detail.

11. Interaction Performance

We're two-thirds into the book already, and we're still talking about the initial load performance. Time to change that. The initial load is important, of course, sometimes the most important part of performance, the very first impression of your website, in a way. But ignoring second, third, and all other impressions after the first one is not always the best strategy.

In this chapter, we'll talk about the second most important part of your website's performance: interaction performance. We talked a little bit about it already when we looked into the INP metric and how to measure it.

In this chapter, we'll learn a few of the most useful tools and techniques to improve it.

Chrome DevTools for Interactions

First things first. Before improving things, we first need to understand how to find and debug things to improve.

Until this moment, we used Chrome's Performance panel to record the performance profile of the initial load in the "production" build, even for debugging. We couldn't really use "dev" mode with niceties like Hot Reload - dev mode usually doesn't bundle the files in the same way and doesn't do tree shaking, which are essential for understanding the effect of the code on the initial load.

However, we could extract a lot of useful information from the recorded profile, and we didn't really need to install any additional tools other than the bundle size analyzer.

For interaction performance, the situation is reversed. We can't really extract anything useful from the "production" build other than detecting where the interactions' "bottlenecks" are. As soon as we can say that this particular interaction is slower than we'd like, we kinda *have to* switch to the dev mode.

This is because, for the initial load, we were generally okay with treating the React part as a giant blob of code of unknown quality. For the purpose of initial load debugging, the only thing that usually matters is how large the blob is. With interactions, we need to understand what *exactly* the blob does, and more importantly, why. A generic performance profiler won't give you much other than "React does something weird, your guess is as good as mine".

You'll see what I mean in a second. Let's start doing things now. Build and start the Study Project for this chapter (`src/chapter11-interactions`):

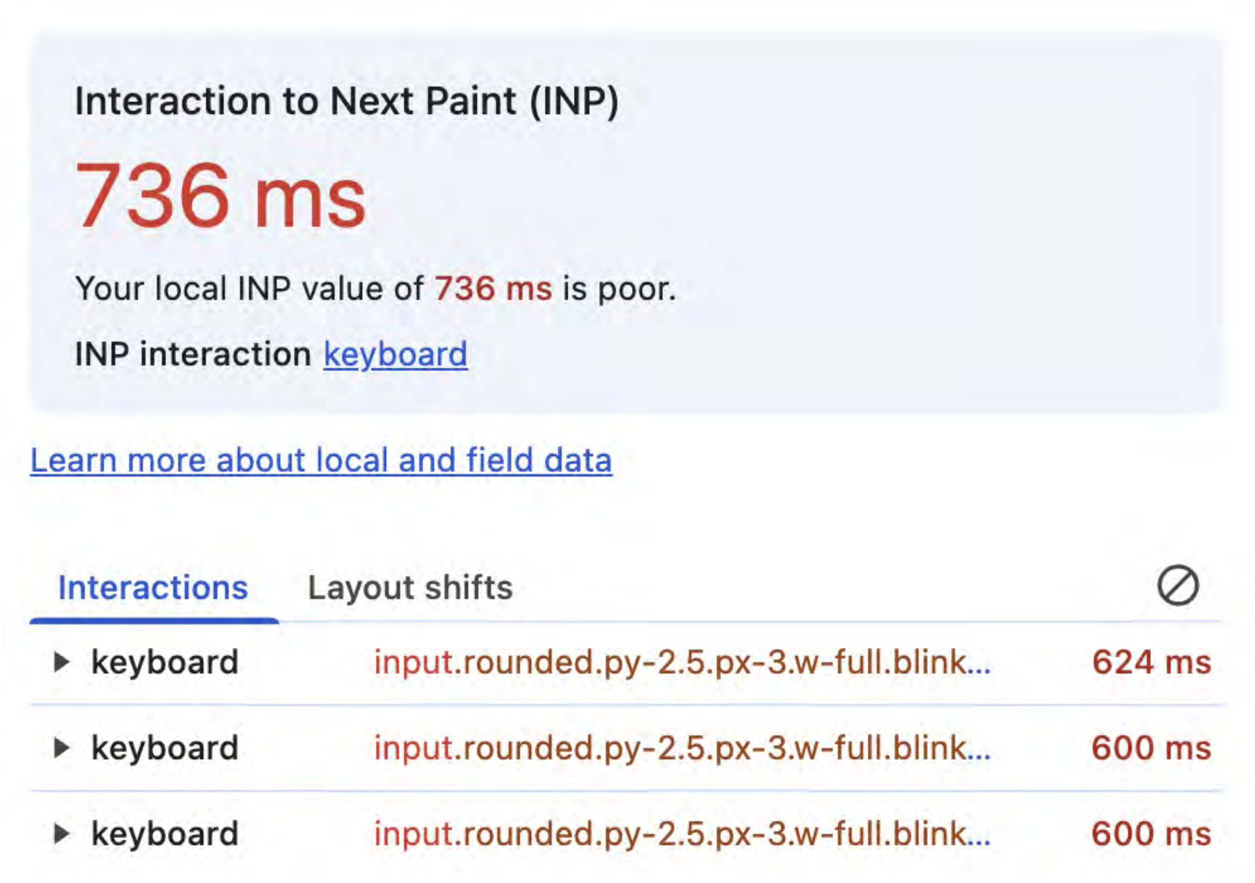
```
npm run build --workspace=chapter11-baseline-frontend
npm run start --workspace=chapter11-baseline-frontend
```

For the rest of the chapter, we're going to investigate the performance of the search field at the very top of the website. Try to type there, although you probably won't notice anything yet.

Because the very first thing to do when investigating interactions is to set the CPU to "low tier" mobile settings. Otherwise, especially if you're on a high-end developer laptop, you likely won't see anything suspicious, regardless of how slow the interaction is in the real world. Unless you're doing something *really* ~~weird~~ heavy, of course, like calculating Pi on the frontend.

I usually set the CPU throttling to 20x to be sure. With the reasoning that if the interaction is "green" on 20x, it's green for everyone.

The second thing is to open the Performance panel and perform the interaction you want to assess while keeping an eye on the bottom of the panel (don't record anything yet). There will be a list of every interaction, and its time will be recorded there. If it's red or just larger than you expected, it's a reason to investigate further.


Interaction to Next Paint (INP)

736 ms

Your local INP value of 736 ms is poor.

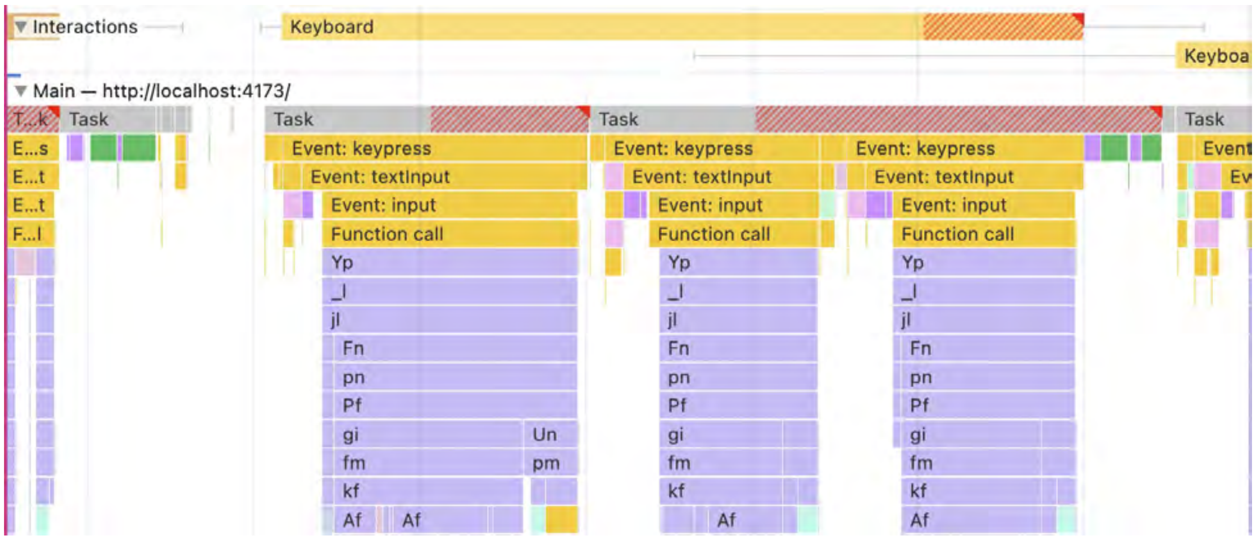
INP interaction keyboard

Learn more about local and field data

Interactions	Layout shifts	
▶ keyboard	input.rounded.py-2.5.px-3.w-full.blink...	624 ms
▶ keyboard	input.rounded.py-2.5.px-3.w-full.blink...	600 ms
▶ keyboard	input.rounded.py-2.5.px-3.w-full.blink...	600 ms

After that, the next natural step is to record the performance profile and examine where those numbers come from. Press the "Record" button in the Performance panel, type in the search field, and then press "Done".

The profile you'll get will be something like this:



There will be the usual "Main" section, where you'll see a flurry of typical and very familiar, by now, yellow JavaScript bars. And then there will be an "Interactions" section on top of it, where you'll see each interaction, i.e., key press, and how long the browser thinks it took, starting from when you first pressed the button until the browser had a chance to react to it with *something* on the screen (i.e., the symbol appears).

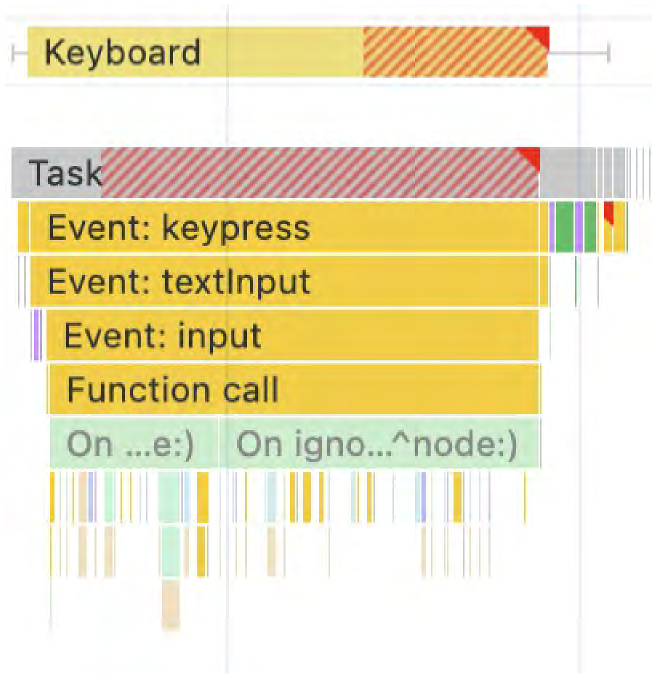
This picture is useful for seeing that, indeed, there is something wrong with how that search field is implemented, judging by all those red diagonal lines on top of grey and yellow bars. And from all we know about the Flame Graphs, we need to look into what's happening inside those "Function call" and make those yellow bars shorter.

The production build is useless for this, as you can see - all the names of all the functions are minified to a completely unreadable wall of symbols. So, the very first thing we need to do after we identify the part of the app to fix is to disable all minification and compression. And the easiest way to do this is to switch to "dev" mode.

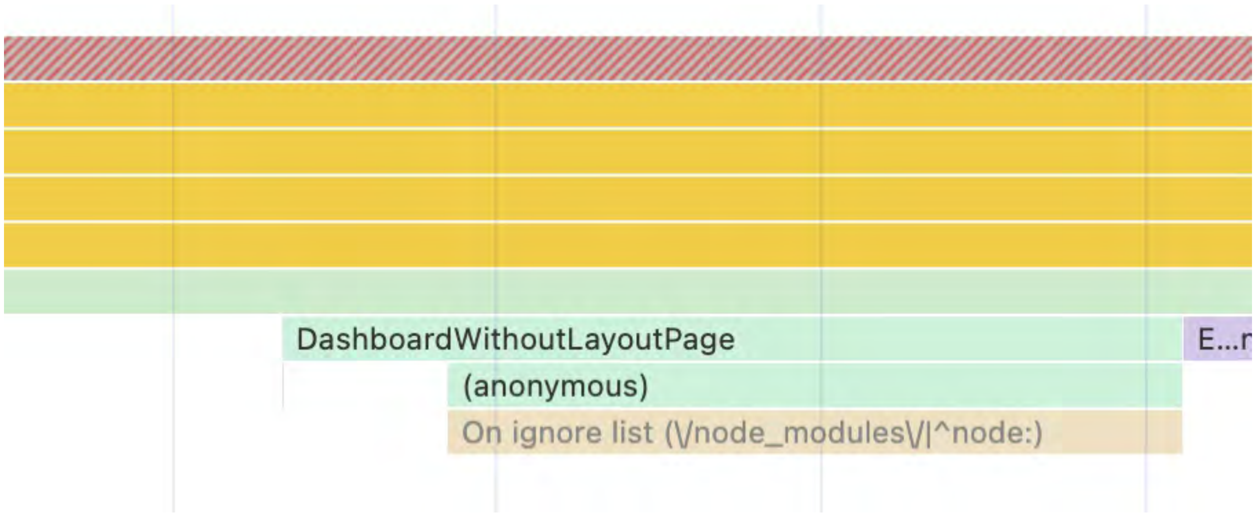
```
npm run dev --workspace=chapter11-baseline-frontend
```

Finally, hot reloading is working! 😅

Record the performance profile for the same interactions, and you'll finally see... that nothing is clear still. 😵



Although if you really zoom in on the graph, you'll see that a few named blocks popped up. And those names look suspiciously familiar.


DashboardWithoutLayoutPage
E...r
(anonymous)
On ignore list (\/node_modules\/|^node:)

If you search the codebase, you'll find those functions - these are normal React components! So, the DevTools at least give us a hint: when we type in the search, React is doing something with those components.

Which just screams "lots of unnecessary re-renders", if you ask me. Because you'll see lots of those function calls inside those blocks. And "re-render" is basically a synonym for "React calls Component's function with some fresh update after a user interaction".

We'll prove that these are re-renders and fix them in a few sections below. Let's finish with non-React stuff first, to wrap them up. Because dealing with re-renders will be spread across several chapters.

The Long Tasks Problem

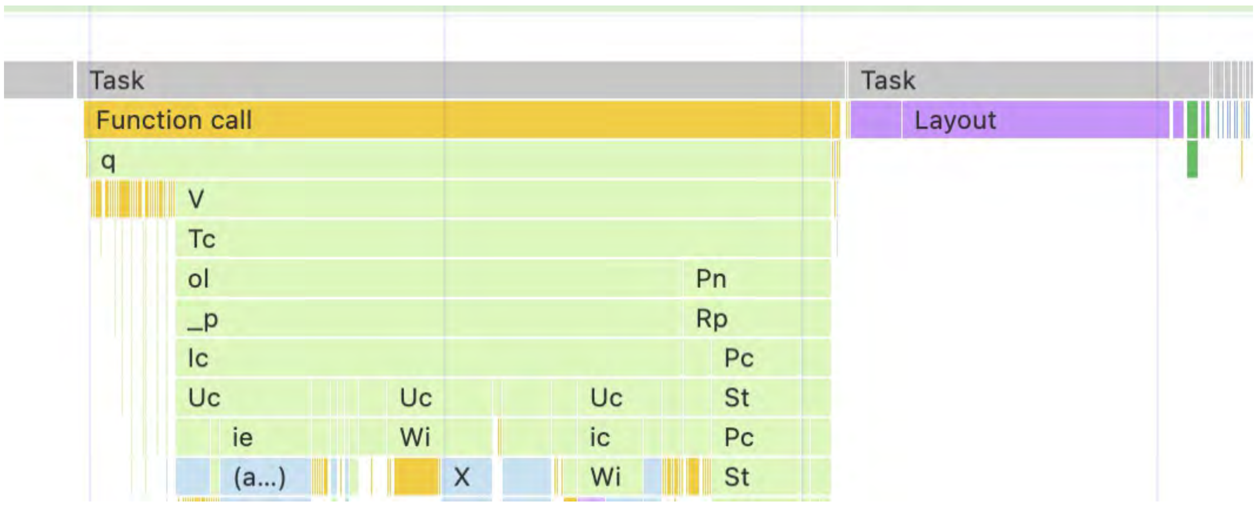
Let's talk a bit more about JavaScript tasks. Or, to be precise, the concept of "long tasks" and why it's a problem.

As you may know, the browser performs everything JavaScript-related as a "task". Simple synchronous JavaScript will be an uninterrupted task. This, for example:

```
const sleep = (ms: number) => {
  const start = Date.now();
  while (Date.now() - start < ms) {
    // busy wait
  }
};

sleep(10);
```

Will be a single task that takes 10 ms. You'll see it in the performance profile as such. If I change it to `sleep(5000);` nothing will change other than duration - it's still going to be one long task that takes 5 seconds.

A first task originates when the JavaScript is loaded - the initial chunk of the synchronous critical path JavaScript. This is what we've seen so many times already: the large yellow line of JavaScript in one unbroken block. You can even see the "Task" indication right on top of it.

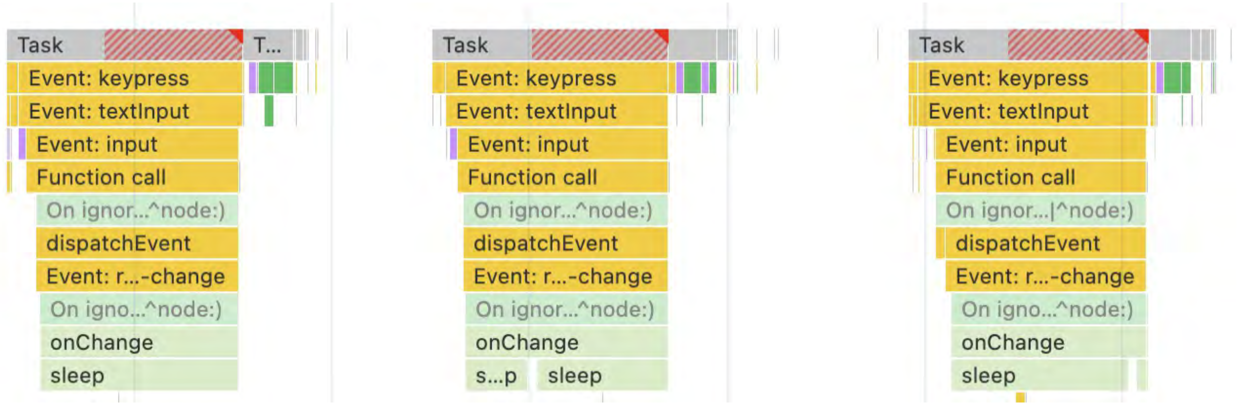


Tasks can also originate from callbacks and other asynchronous operations. Like typing in the search field, whether it's React or not, or from a Promise. In this case, the browser will place these tasks in a queue and process them in due time after it's completed its current task.

If you find that slow Search field we've been playing with, get rid of the state and instead call the `sleep` function in the `onChange` callback:

```
// FILE: frontend/patterns/topbar-for-sidebar-content-layout.tsx

<InputWithIconsNormalToLarge
  // value={search}
  onChange={(e) => {
    // setSearch(e.target.value);
    sleep(100);
  }}
/>
```

You'll see that every keypress produces an unbroken 100-millisecond task.



You'll see that the duration of the task is controlled by the `sleep` function. Which adds a bit more usefulness to the Chrome profiler, by the way. If you see some meaningfully named non-React function here, then the long task is caused by your own code, not React re-renders.

While the browser is busy performing a task, it's unresponsive. Like, completely out of it. If you change the `sleep` value from `100` to `5000` in the `onChange` callback and try to type anything, you'll see that after the very first key press, the entire page freezes and nothing is interactive anymore.

This is exactly what happens when we trigger a re-render of the entire page by typing in the search field: the browser has too much React to do in one go.

When it comes to long tasks like that, solutions can be put into two broad categories: split or shorten. We can split the task into smaller chunks, allowing the browser to queue something in between. Or, we can reduce the task's execution time by making it do less work. I.e., lots of refactoring most of the time.

Let's start with the first one. Because everything from the "shorten" category basically means lots of refactoring to get rid of the unnecessary re-renders.

Dealing with Long Tasks: Yield to Main

The solution of breaking up long tasks is often referred to as "yield to main"[168]. The idea is this. I used a `sleep` function with a `5000` value to imitate a long task - one single function call. However, in real life, it's highly unlikely to see a situation like this. More likely, it will be a set of function calls, each of which won't take *that* long.

```
// it's probably going to be like this
doSomething();
thenSomethingElse();
andSomethingAfter();
```

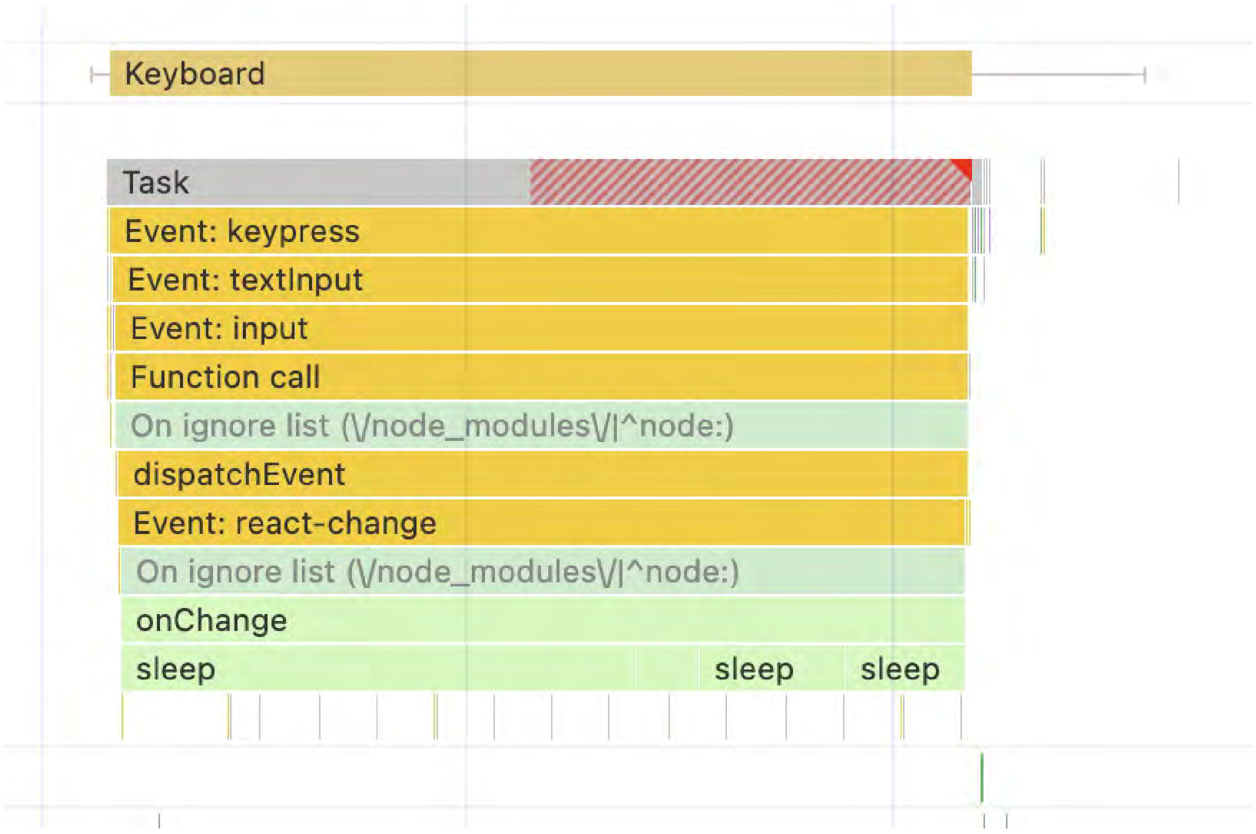
Triggered in a chain like this, this code will produce a single task. We can imitate that situation with the help of the `sleep` function:

```
const sleep = (ms: number) => {
  const start = Date.now();
  while (Date.now() - start < ms) {
    // busy wait
  }
};

for (let i = 0; i < 10; i++) {
  sleep(10);
}
```

If we do something synchronous for 10 ms 10 times, it already gives us a 100 ms delay. Put it in the search field:

```
// FILE: frontend/patterns/topbar-for-sidebar-content-layout.tsx
<InputWithIconsNormalToLarge
  // value={search}
  onChange={(e) => {
    for (let i = 0; i < 10; i++) {
      sleep(10);
    }
  }}
/>
```

Then start typing and record the performance:



You might see a few `sleep` calls at the end of the flame graph, but it's just one single task that blocks the browser.

However, now that we have multiple `sleep` calls, we can tell the browser explicitly that after each of them, it can pause for a bit, unblock itself, and check whether something happened while it was busy. And if something happened, react to it.

scheduler.yield()

There are multiple ways to do that. The latest and greatest is the Scheduler API[169] with `scheduler.yield()`:

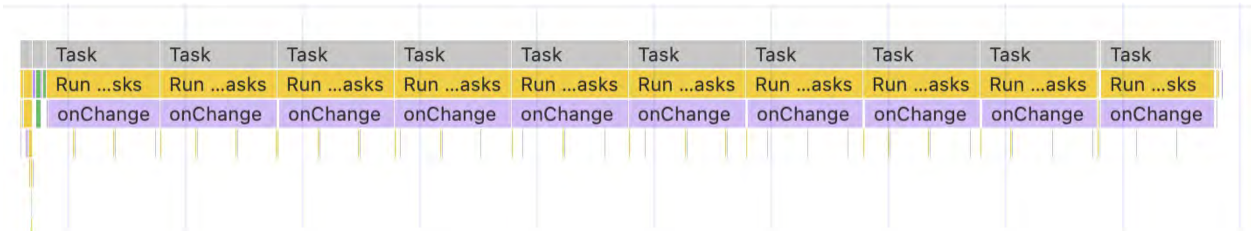
```
for (let i = 0; i < 10; i++) {
  sleep(10);
  // yield to main after each sleep
  await scheduler.yield();
}
```

Or in the search field right away:

```
<InputWithIconsNormalToLarge
  // value={search}
  onChange={async (e) => {
    for (let i = 0; i < 10; i++) {
      await scheduler.yield();
      sleep(10);
    }
  }}
/>
```

Don't forget to make the `onChange` async!

If you record the performance again, you'll see a bunch of microtasks where one unbroken task used to live:



And the best part - nothing is frozen anymore! No matter how fast you type, everything is responsive. It's especially visible when you change the `for` loop to perform a thousand times instead of ten.

Clean up the Performance panel and pay attention to the "Interactions" block. Everything is green regardless of how much the CPU is slowed down!

scheduler.yield() Fallback

Be mindful, however, that since `scheduler.yield` is the latest and greatest, not every browser supports it. Especially old browsers on mobile platforms, where you'd see the greatest benefits from splitting up tasks, which is a bit ironic.

Not to worry, however, we can implement a perfectly valid fallback for this case that works everywhere. It relies on the fact that functions like `setTimeout` or `requestAnimationFrame` naturally create their own task that is put into the browser queue.

Basically, we can implement the `yieldToMain` function as simply as that:

```
const yeildToMainThread = () => {
  return new Promise((resolve) => {
    setTimeout(resolve, 0);
  });
};
```

And use it in place of `scheduler.yield` in exactly the same way:

```
<InputWithIconsNormalToLarge
  // value={search}
  onChange={async (e) => {
    for (let i = 0; i < 10; i++) {
      await yeildToMainThread();
      sleep(10);
    }
  }}
/>
```

With exactly the same result!

When to use it in React?

After reading the pattern above, a question should immediately arise: is this really applicable to our standard UI applications? Even in the search example from the very beginning, we didn't have any loops or anything to yield from. Just an `onChange` callback with `setState` inside.

The answer is probably "very, very rarely". To be completely honest with you, I don't think I've ever used it explicitly in any of my React apps. Most of the heavy computation in React is usually hidden behind state updates and delegated to React itself.

However, this trick might be useful when you have a lot of non-React calculations. Especially if you need to iterate over DOM nodes for these. Think something like implementing your own custom browser-based game or a fancy animation framework. Or creating a competitor to React 😉.

React DevTools for Interactions

Okay, so back to the original problem of search, without any artificial `sleep` tasks. We already guessed that it's slow because of re-renders and looked at the code of the input component:

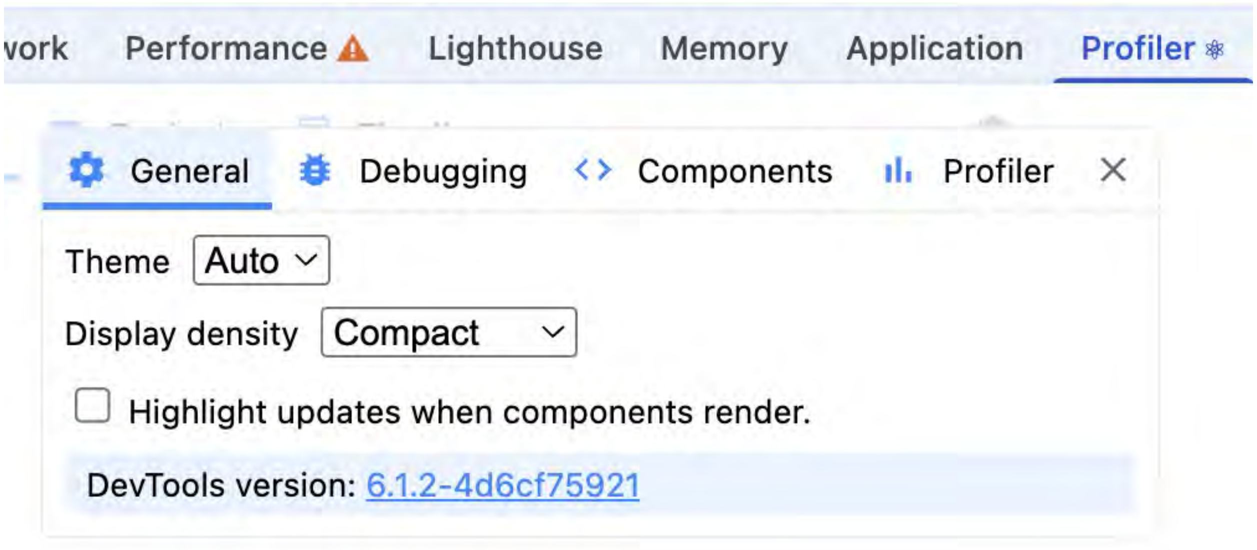
```
<InputWithIconsNormalToLarge value={search} onChange={(e) => setSearch(e.target.value)} />
```

In the `onChange` callback, I see a `setSearch`, which is not a local state but comes from props. So I'd say it's a pretty safe bet that lots of unnecessary re-renders are responsible for the slowness.

But to actually *see* the size of the disaster and confirm it's re-renders, we need something visual. Well, technically speaking, you can just put `console.log` in a few strategic places, and this is what I do 98% of the time, but you don't have to be that old school. Plus, having nice visuals always helps.

Chrome DevTools won't help us here much, as we've seen, so it's time to try something else and switch to React DevTools. React DevTools[170] is a Chrome plugin that, when downloaded and installed, gives you two additional tabs in Chrome DevTools: "Profiler" and "Components", both with a tiny React icon. The plugin works only in "dev" mode, so it's good that we switched to it already.

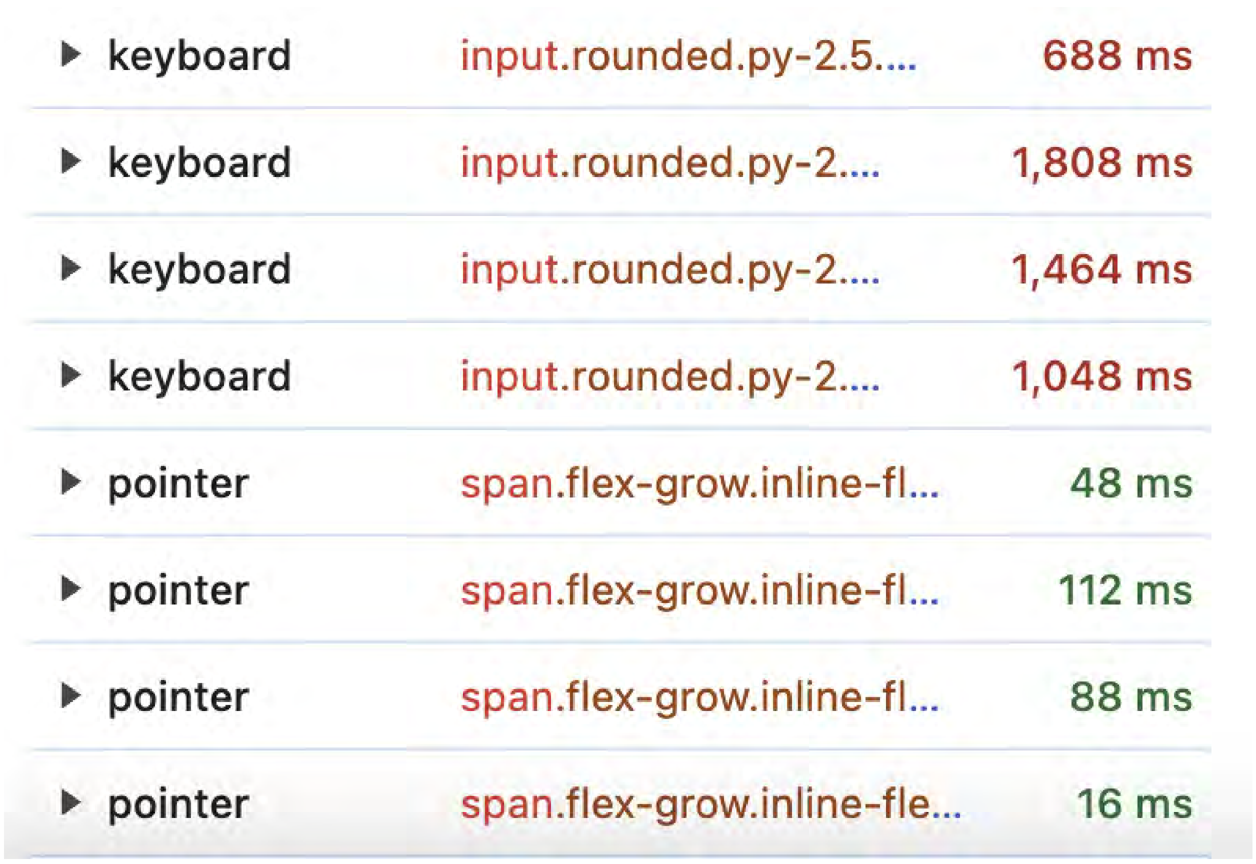
If you haven't installed the plugin yet, now is the time to do so. After that, run the project in "dev" mode, open the new Profiler tab, click on the "cog wheel" icon to open settings, and check the "Highlight updates when components render" checkbox in the "General" tab.



Theme: Auto
Display density: Compact
☐ Highlight updates when components render.
DevTools version: 6.1.2-4d6cf75921

This is the most useful tool when it comes to interactions and React. It does exactly what it says: it highlights all components on the page that re-render on every interaction. Try to type into the search box when this checkbox is "on". The entire page should light up like a Christmas tree.

For contrast, try to open/close the "Tasks" collapsible items in the Sidebar - only the items inside that panel will be highlighted. Slow down the CPU 20x, and you'll see that the INP number is still in the "green" for this interaction. Compare it with typing in the search:



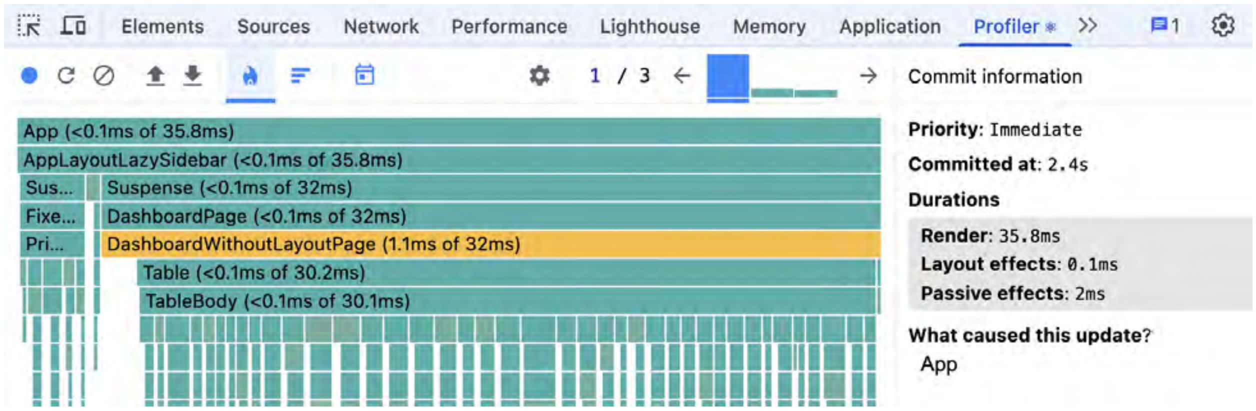
▶ keyboard	input.rounded.py-2.5....	688 ms
▶ keyboard	input.rounded.py-2....	1,808 ms
▶ keyboard	input.rounded.py-2....	1,464 ms
▶ keyboard	input.rounded.py-2....	1,048 ms
▶ pointer	span.flex-grow.inline-fl...	48 ms
▶ pointer	span.flex-grow.inline-fl...	112 ms
▶ pointer	span.flex-grow.inline-fl...	88 ms
▶ pointer	span.flex-grow.inline-fle...	16 ms

That will be the difference between "only legitimately necessary re-renders" and "something went wrong and the entire page re-renders unnecessarily".

Before investigating what happened and how to fix it, though, let's check out another cool toy.

Similar to the default Chrome profiler, the React profiler allows you to record the performance profile. The "Record" button is even in the same place - at the top left corner of the panel. Click it to start the recording and interact with the search again.

After the recording is done, in the "Flamegraph" section, you'll see the same graph we've been staring at for so long over the course of this book:

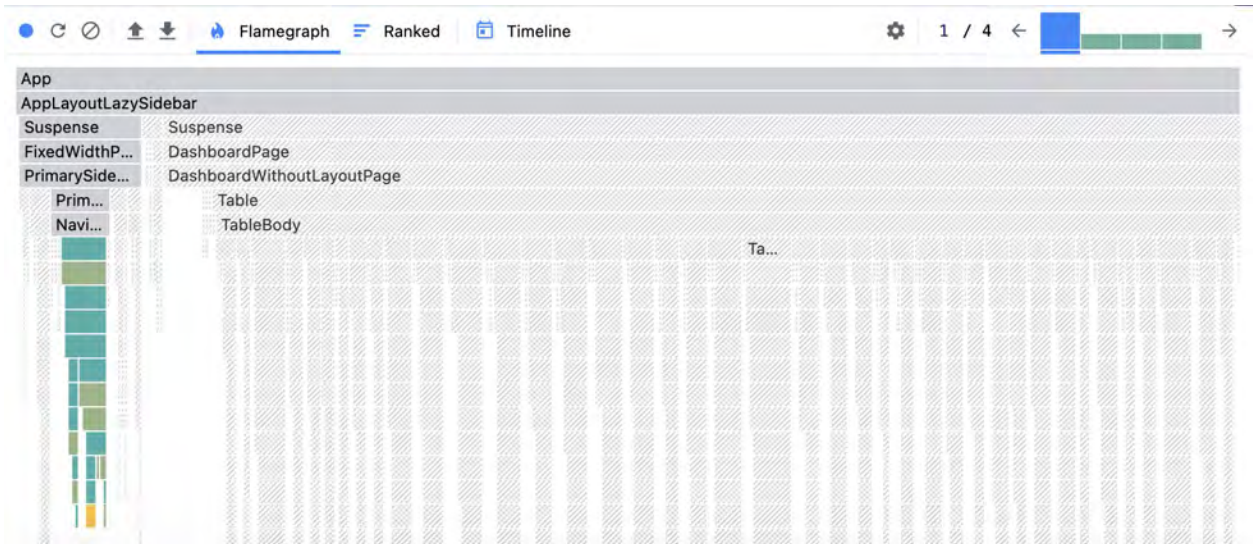


App (<0.1ms of 35.8ms)
AppLayoutLazySidebar (<0.1ms of 35.8ms)
Sus... Suspense (<0.1ms of 32ms)
Fixe... DashboardPage (<0.1ms of 32ms)
Pri... DashboardWithoutLayoutPage (1.1ms of 32ms)
Table (<0.1ms of 30.2ms)
TableBody (<0.1ms of 30.1ms)

Priority: Immediate
Committed at: 2.4s
Durations
Render: 35.8ms
Layout effects: 0.1ms
Passive effects: 2ms
What caused this update?
App

Only this time, it finally makes sense for the React-trained eye. All our components are here, in their hierarchical order, so reading it should be pretty intuitive by now. We have an `App` component that triggers (i.e., renders) AppLayoutLazySidebar, then `Suspense`, `DashboardPage`, etc., etc. The entire tree of React components.

Now, record the same profile for the "Tasks" collapsible items in the Sidebar, and you'll immediately see what the color coding is for:



The colored components are those that are re-rendered during the recorded interaction. Which confirms what we've already seen by enabling the "highlight re-renders" checkbox. Only this time, we can see how much time those re-renders took relative to each other and potentially see if a component is a bottleneck.

So the picture is pretty clear here: when we're typing in the Search field, we trigger re-renders of the entire tree of components. To speed up the typing and return to the "green" numbers, we need to make sure that the re-renders picture is similar to the collapsible items in the Sidebar.

What's Next

The re-renders discovery above means that we're deep in React territory now. As a result, we need to deal with it in a React way. Which, of course, means that we have a thousand and one possible solutions.

So in the next two chapters, we'll look into exactly that: possible solutions for the re-renders problem. In the first one, we'll look into the most efficient manual solutions, and in the chapter after that, how to automate them with React Compiler.

12. Getting Rid of Unnecessary Re-renders

We discovered in the previous chapter that the search field causes the entire page to re-render on every key press. Time to deal with it.

Although I must mention that the amount of content that touches on re-renders and React components, their lifecycle-related behavior, and the reasons for those, could fill another book. So this chapter is more of a "brief" refresher of the most important parts of the full picture that could help us investigate and fix the search.

If any of the content in this chapter is a surprise, and you want to understand how it works *exactly*, you'd need my other book called "Advanced React"[171]. Most of it is dedicated to the behavior touched on in this chapter. I wasn't joking about filling an entire book, you see. You can use the **ADVREACT30** discount code to receive a 30% discount during the checkout process on the website. Just don't share it publicly, let it be our little secret 😉

And if you already know that book by heart, you'll find exactly zero new information in this chapter. So treat it as a refresher to nod along, or just skip it altogether.

Re-renders Basics

As we've just seen, the unnecessary re-render of the entire page caused our search field to be somewhat slow. In fact, probably 90% of all slow interactions in React are caused by either too many re-renders in an interval or re-renders of too much at the same time. Or both.

So, what exactly is a re-render?

When the app appears on the screen for the first time, it's ***mounted***. This is when React initializes every component for the first time, triggers its lifecycle for the first time, creates or hydrates DOM nodes, and does all other important work for the app to function properly.

After all this is done and the user interacts with the page, React needs to update everything that needs to change after this interaction. This process of updating already existing components is what we know as a ***re-render*** in React. A re-render is usually much faster than mounting since React doesn't need to create anything from scratch; it just updates already existing components with some new data.

The one and only way to initially trigger a re-render in React is to update the state. This may take the form of many different APIs: `setState` from the `useState` hook, `useSyncExternalStore` hook, `useReducer` hook, or any of the external state management libraries and their APIs. Underneath all of this, there is a ***state update***.

After the component that triggers the state update is updated (i.e., re-rendered), React then needs to distribute the changes to all other components that might possibly depend on the changed data. So, a re-render of ***every component rendered inside*** is now triggered. Re-renders of those trigger re-renders of components *inside of those components*, and so on and so forth, until the end of this tree of components is reached.

Have you noticed how I didn't mention props even once here? Because props don't matter for the "default" behavior! If a re-render of a component is triggered, ***all components*** rendered inside will re-render regardless of whether their props change or not.

The only time when whether the props change matters is when a component is ***memoized***, either manually via `React.memo`, or automatically via the Compiler. In this case, when React encounters this component in the render tree, it will stop and check whether the props have changed. If even a single prop changes, the entire component's re-render is triggered. So, for a component to skip re-renders, both it and *every single prop* should be memoized.

Re-renders are "hierarchical" and never go "up", unless explicitly "bubbled" up via some callbacks. So, usually, anything that comes from props is "higher" in the hierarchy and is *not* affected by re-renders.

Creating components inside other components is a very bad idea. If a re-render of such a component is triggered, React will ***unmount*** this component first and then ***mount*** it back. It's called ***re-mounting***, and it usually causes performance problems (since mounting is much slower than re-rendering), and various bugs like lost focus or reset state.

There are, however, some cases when you explicitly want to trigger re-mounting or reuse the existing instance of a sibling component. In this case, you need to use the `key` property with some stable value between re-renders.

Re-renders Situation in the Search Field

Okay, so we found the problem with the search - the entire page re-renders. We now know what re-renders are and how they behave by default. Now it's time to identify the problem in the code.

First, we have the `App` component that controls the search's state:

```
// FILE: src/chapter12-re-renders/simple-frontend/App.tsx
export default function App() {
  // state initialized here
  const [search, setSearch] = useState("");

  return (
    <!-- state passed down here -->
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      <DashboardPage />

      <!-- state used here -->
      <div>Search results for {search}</div>
    </AppLayoutLazySidebar>
  );
}
```

This is the very "root" component, so it makes sense why the entire page lights up when we're typing something. By the rules of re-renders, when the state changes, everything that is rendered inside the component with the changed state will re-render. In our case, the entire page.

This state is passed down through a few layers of components via `AppLayoutLazySidebar`, where eventually it ends up attached to the search field:

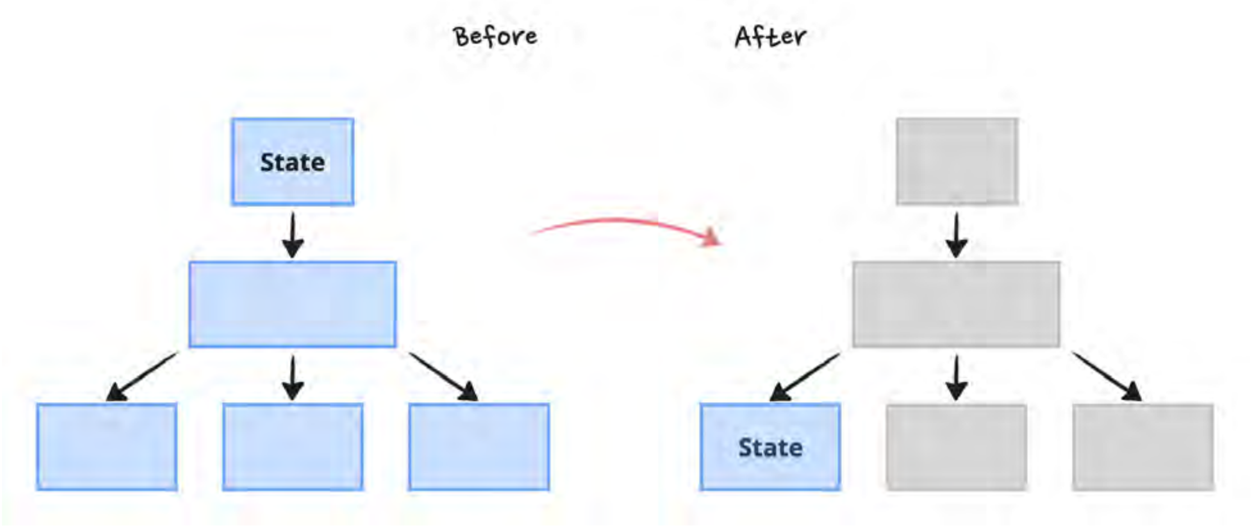
```
// FILE: frontend/patterns/topbar-for-sidebar-content-layout.tsx

export const TopbarForSidebarContentLayout = ({ search, setSearch }) => {
  return <InputWithIconsNormalToLarge value={search} onChange={(e) =>
setSearch(e.target.value)} placeholder="Search..." />;
};
```

Also, this state is used in the div with the future search results. It's probably going to be some sort of drawer that expands over the entire page when implemented, so it makes sense why it's at the very root of the app.

Now to fixing it. In React, there are, of course, one million ways to do that. Let's start with the simplest and the most overlooked one.

Moving State Down

If the state update causes all components inside to re-render, the simplest solution is to reduce the blast radius of the updates by moving the state away from the "root" component down down down the render tree to the very last component that actually needs it.



A very good indication of a situation where it's appropriate is a few layers of components that just pass state around, rather than utilizing it. Which is exactly what happens in our case: the state value passes through a few components until it reaches the search field.

In code, it would mean we need to move the state at the very least to the `TopbarForSidebarContentLayout` inside `frontend/patterns/topbar-for-sidebar-content-layout.tsx`:

```
// FILE: frontend/patterns/topbar-for-sidebar-content-layout.tsx
export const TopbarForSidebarContentLayout = () => {
  const [search, setSearch] = useState('');

  return <InputWithIconsNormalToLarge value={search} onChange={(e) =>
setSearch(e.target.value)} placeholder="Search..." />;
};
```

The div/future drawer with the search results would also have to move there, of course:

```
// FILE: frontend/patterns/topbar-for-sidebar-content-layout.tsx
export const TopbarForSidebarContentLayout = () => {
  const [search, setSearch] = useState('');

  return (
    <>
      <InputWithIconsNormalToLarge value={search} onChange={(e) =>
setSearch(e.target.value)} placeholder="Search..." />
      <div>Search results for {search}</div>
    </>
  );
};
```

Do this simple thing and measure how long it takes to type in the search field now. For me, it improved the interaction **10 times**, from ~500 ms to ~50 ms, and moved it from "insufferable and very red" to "sleek and always green". Just like that!

In reality, I'd want to introduce debouncing here as well: those search requests surely will be sent to some form of backend to get results. And results would have to be rendered in a Portal rather than directly like this, otherwise, you'll suffer from stacking context issues.

Components as Children and Props

Let's assume that Moving State Down is not available for some reason, and the state has to stay inside the `App` component. Maybe we can't move the search results drawer down with the state because it's tied to something else that is happening inside the `App` component.

We can still reduce the state blast radius even in this situation, before resorting to some drastic measures like memoization. If we look at the `App` component again, we'll see that `DashboardPage` is the component we want to avoid re-rendering: there is the entire page underneath. The `AppLayoutLazySidebar` component, although strictly speaking is not necessary to re-render, is not that big of a deal - it's just the layout and the sidebar.

```
// FILE: src/chapter12-re-renders/simple-frontend/App.tsx
export default function App() {
  const [search, setSearch] = useState("");

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      <!-- only this one is a problem -->
      <DashboardPage />

      <div>Search results for {search}</div>
    </AppLayoutLazySidebar>
  );
}
```

The second rule of re-renders: anything that is passed as props is not affected. So, what will happen if I encapsulate everything that depends on the state in one component and pass `DashboardPage` as a prop? Like this:

```
const LayoutWithSearch = ({ content }) => {
  const [search, setSearch] = useState('');

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      {content}
      <div>Search results for {search}</div>
    </AppLayoutLazySidebar>
  );
};
```

```
export default function App() {
  return <LayoutWithSearch content={<DashboardPage />} />;
}
```

There is no more state that can affect the `DashboardPage` component inside the `App`! Typing in the search field is again green and sleek, although *slightly* slower than the first solution (because of the Sidebar). But it's green with 20x CPU slowdown, so who cares?

And now the trick that blows the minds of people who see it for the first time. Rename the `content` prop to `children`:

```
export default function App() {
  return <LayoutWithSearch children={<DashboardPage />} />;
}
```

Everything still works, since it's just a prop. Then rewrite it like this:

```
export default function App() {
  return (
    <LayoutWithSearch>
      <DashboardPage />
    </LayoutWithSearch>
  );
}
```

Everything still works! Because it's exactly the same thing. This nice HTML-like syntax is nothing more than syntax sugar for the `children` prop. It's simply a cleaner look, plus it will become important to know when we get down to memoization.

Avoid Props Drilling

What to do, however, if we want to avoid re-rendering the Sidebar as well, while keeping the state at the "root"? In this case, we can leverage React Context or any external state management solutions like Zustand[172] or Redux[173]. Basically, anything that allows us to bypass layers of components and extract data from some semi-external "storage" directly to the component that needs the data.

At the very root, we'll create the Data Provider that stores the search state in Context[174]:

```
const Context = createContext({
  search: '',
  setSearch: () => {},
});

export const useSearch = () => useContext(Context);

const DataProvider = ({ children }) => {
  const [search, setSearch] = useState('');
  const value = useMemo(() => ({ search, setSearch }), [search, setSearch]);

  return <Context.Provider value={value}>{children}</Context.Provider>;
};
```

Wrap the contents of our `App` in this provider, so that everything can have access to the Context data:

```
export default function App() {
  return (
    <DataProvider>
      <AppLayoutLazySidebar>
        <DashboardPage />

        <div>Search results for ...</div>
      </AppLayoutLazySidebar>
    </DataProvider>
  );
}
```

We already know that the `children` prop won't be affected by the state change, so everything inside the `App` is safe.

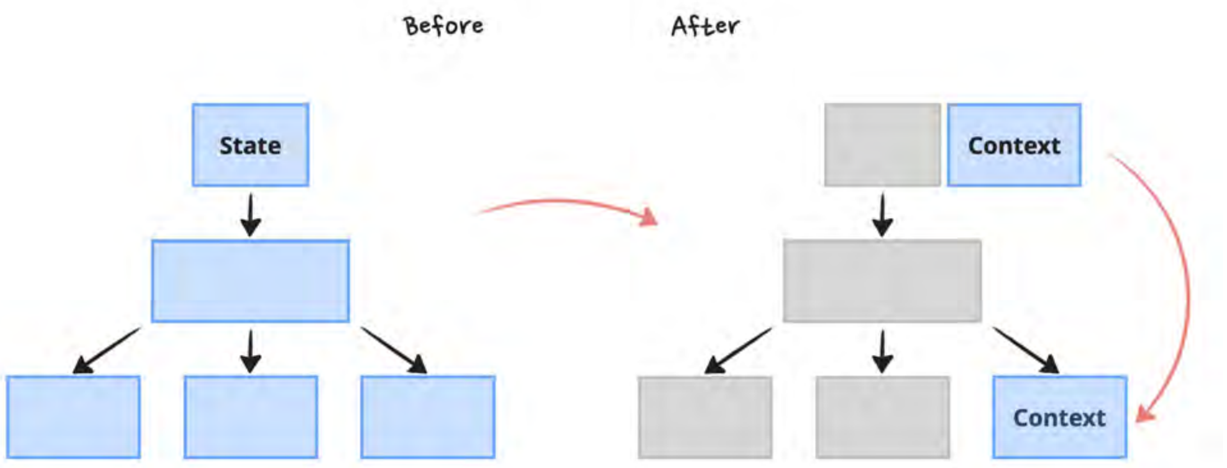
Then extract the Search results into their own component and use the `useSearch` hook to access the search data:

```
const SearchResults = () => {
  // useSearch from the Context provider
  const { search } = useSearch();
  return <div className="hidden">Search results for {search}</div>;
};
```

Get rid of all the props drilling on the `AppLayoutLazySidebar` component, and use Context directly inside the component that renders the search field:

```
// FILE: frontend/patterns/topbar-for-sidebar-content-layout.tsx
export const TopbarForSidebarContentLayout = () => {
  // useSearch from the Context provider
  const { setSearch, search } = useSearch();

  return (
    <>
      <InputWithIconsNormalToLarge value={search} onChange={(e) =>
setSearch(e.target.value)} placeholder="Search..." />
      <div>Search results for {search}</div>
    </>
  );
};
```

Run it, measure it, and enjoy yet again the sleek and green experience even on 20x CPU delay. Plus, as a nice bonus, cleaner code, since you don't need to pass props around anymore. If we visualize this pattern as a tree, it will look like this:



Memoization

The patterns above are impressive, right? However, in real life, it's not always that easy to implement them. The real code is messy, and moving state around like that or introducing state management or Context would require significant refactoring, which is not always feasible.

For this case, we have memoization.

In React, memoization stands on three pillars: `useMemo`, `useCallback`, and `memo`. All three will preserve a reference to whatever they memoize between re-renders.

We'll use `useMemo` when we need to save the reference to arrays and objects:

```
const App = () => {
  const dataMemo = useMemo(() => [{ id: 1 }, { id: 2 }], []);
};
```

Although in real life, the value inside `useMemo` will be derived from something inside the `App` component itself. Otherwise, there is no point: the array/object can be moved outside of the component, and `useMemo` can be avoided altogether.

We'll use `useCallback` when we need to preserve the reference to a function:

```
const App = () => {
  const onClickMemo = useCallback(() => {
    console.log('Button clicked');
  }, []);
};
```

And we'll use `memo` when we need to memoize a *component itself*:

```
const App = () => <div>...</div>;
const AppMemo = memo(App);
```

How would this help us with our search problem? Simple.

As we know, if a component is wrapped in `memo`, React will stop and check its props. If no props change, then re-renders will be skipped for this component. So in our initial case:

```
// FILE: src/chapter12-interactions/baseline/App.tsx
export default function App() {
  const [search, setSearch] = useState('');

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      <DashboardPage />

      <div>Search results for {search}</div>
    </AppLayoutLazySidebar>
  );
}
```

If we want to avoid the re-rendering of the `DashboardPage` component, we just need to wrap it in `memo`:

```
// wrapped in memo to prevent re-renders
const DashboardPageMemo = memo(DashboardPage);

export default function App() {
  // state here

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      {/* wrapped in memo to prevent re-renders */}
      <DashboardPageMemo />

      <div>Search results for {search}</div>
    </AppLayoutLazySidebar>
  );
}
```

If it had some props, we'd need to memoize the props as well:

```
// wrapped in memo to prevent re-renders
const DashboardPageMemo = memo(DashboardPage);

export default function App() {
  // state here

  // wrap objects and arrays in useMemo
  const dataMemo = useMemo(() => [{ id: 1 }, { id: 2 }], []);

  // wrap functions in useCallback
  const onClickMemo = useCallback(() => {
    console.log('Button clicked');
  }, []);

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      {/* don't forget to memoize ALL the props! */}
      <DashboardPageMemo data={dataMemo} onClickMemo={onClickMemo} />

      <div>Search results for {search}</div>
    </AppLayoutLazySidebar>
  );
}
```

And the app is free from unnecessary re-renders once again.

But to do it right is complicated. *Very* complicated, in fact.

Over time, those simple memos turn into chains of `useMemo` and `useCallback`, props start being involved, and to maintain the purity of memoization, you have to track it down through layers and layers of components and other hooks, and eventually, one of them will slip. And that will be enough.

As soon as even a single prop is not memoized correctly, the entire construct falls apart, and the memoized component starts behaving like it was not memoized at all. And I haven't mentioned anything about `children` yet 😉

What if the `DashboardPage` component had children? The pattern from above, which looks like HTML:

```
const DashboardPageMemo = memo(DashboardPage);

export default function App() {
  // state here

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      {/* what if it had children like this? */}
      <DashboardPageMemo>
        <ChildComponent />
      </DashboardPageMemo>
    </AppLayoutLazySidebar>
  );
}
```

A lot of people don't realize that `ChildComponent` in this scenario is just a prop (like we've seen before). So, in order for the memoization on the `DashboardPage` to work properly, we need to *memoize* `ChildComponent` *as well.*

If you didn't get caught by the fact that `ChildComponent` is a prop in this scenario and are ready to memoize it, I can bet almost anything that your first attempt at memoizing it will be like this:

```
const DashboardPageMemo = memo(DashboardPage);
const ChildComponentMemo = memo(ChildComponent);

export default function App() {
  // state here
  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      <DashboardPageMemo>
        <ChildComponentMemo />
      </DashboardPageMemo>
    </AppLayoutLazySidebar>
  );
}
```

Wrap it in `memo` and render the memoized version - exactly the same as the `DashboardPage` component. This feels only natural. And it's entirely **wrong**.

Or, to be precise, `ChildComponent` itself will be memoized, of course, and won't re-render on state change in the `App` component. Memoization of the `DashboardPageMemo`, however, is broken here.

`<ChildComponentMemo />` here is an *Element*, not a component. An Element is just an object with the `type` property that points to the component:

```
const child = <ChildComponentMemo />;
// the same!
const child = {
  "type": ChildComponentMemo,
  ... // bunch of other stuff
}
```

The *object* itself is not memoized, so the reference to it is not stable between re-renders. Essentially, we have this situation:

```
<DashboardPageMemo
  children={{
    type: ChildComponentMemo,
  }}
/>
```

Memoized component with a single non-memoized prop. Memoization is broken. The `DashboardPageMemo` component will re-render with every state change.

To memoize it properly, we need to do it in the same way as any other object. I.e., we need `useMemo`:

```
export default function App() {
  // state here

  const memoChild = useMemo(() => {
    return <ChildComponentMemo />;
  }, []);

  return (
    <AppLayoutLazySidebar search={search} setSearch={setSearch}>
      <DashboardPageMemo>{memoChild}</DashboardPageMemo>
    </AppLayoutLazySidebar>
  );
}
```

This is probably React's most counterintuitive pattern by far, and I've seen even very senior engineers get caught by it. So if you got caught by it, don't feel bad, you're in really good company!

What's Next?

As we've seen, fighting re-renders can involve a significant amount of effort. You'd either need to refactor your app in lots of ways or introduce memoization. And manual memoization in React is probably the most fragile pattern in the history of fragile patterns.

So, it's no surprise that a lot of smart people were not happy with it and worked on a way to automate it. After a few years of effort, the React team produced the solution: React Compiler[175], which does exactly that. We'll investigate it in the next chapter.

13. React Compiler

After React Server Components, the React Compiler[176], is probably the most talked-about feature of React. And probably the most anticipated one. And for a good reason.

As we've seen in the previous chapter, implementing memoization manually is no fun and has more chances to go wrong than right. If any of the content in the previous chapter was a surprise, or if you've never given memoization much thought and were memoizing things just because it kinda sounds like a good idea, I can bet almost anything that half of it is broken in your app right now.

The Compiler[177] is supposed to help with it. So what exactly is it? How can it help? And what are the consequences? This is what we're investigating here.

What Is React Compiler

Many people think that React Compiler is part of React for some reason. It's not! At least it's not at the time of writing this book 😅 Who knows what will happen two weeks after its release.

In fact, it's actually a separate tool implemented as a Babel[178] plugin. I.e., it's part of your build system that transforms the code from something we write daily in React to something that the browser understands.

It was introduced as a concept in December 2021[179], was released as a beta version in October 2024[180], and as a Release Candidate in April 2025[181]. So, technically speaking, it's not ready for production *yet*. In Next.js, for example, this feature is still (i.e., in June 2025) marked as "experimental"[182]. Use it at your own risk!

The fact that it's not part of React is both good and bad news. The bad news is that you need to install it separately. Just updating React to the latest version won't be enough. You'd also need to be on the build system that supports Babel plugins, which is not all of them. You can find the list of all the build tools that support the Compiler on the official React website[183].

The good news, however, is that since it's not part of React 19, you don't *need* to update to the latest version in order to get the benefits. React 17 is the lowest version that the Compiler supports, so even if you're stuck with this five-year-old "legacy", you're still good to try the Compiler.

Setting it up is relatively straightforward if you're on the correct system. Just follow the official guide[184], and it should be fine. Cleaning up the codebase to prepare for the compiler will be more of a challenge. To help with that, the React team also released an ESLint plugin, so make sure to install it and fix everything that it finds.

What the Compiler Does

Like any Babel plugin, it transforms the code you write into something else. Remember all the manual memoization you need to do in order to memoize a component with props? This code, for example:

```
const App = () => {
  return (
    <HeavyComponent
      arrayProp={[1, 2, 3]}
      callbackProp={() => {
        console.log('Callback happened');
      }}
    >
      <ChildComponent />
    </HeavyComponent>
  );
};
```

In order to make sure that `HeavyComponent` doesn't re-render, we need to introduce manual memoization in *four* different places: the component itself, two of its obvious props, and the children. The end result will be this:

```
const App = () => {
  const memoArray = useMemo(() => [1, 2, 3], []);
  const memoCallback = useCallback(() => {
    console.log('Callback happened');
  }, []);
  const memoChild = useMemo(() => <ChildComponent />, []);

  return (
    <HeavyComponent arrayProp={memoArray} callbackProp={memoCallback}>
      {memoChild}
    </HeavyComponent>
  );
};
```

The compiler transforms the first, non-memoized code into code that *behaves* as the second, memoized code.

However, it doesn't wrap things in memoization hooks that literally. It's much smarter than that. It tries to predict things, leverages conditional renders, groups things together, etc. So saying that it just wraps everything in memoization is not correct. But the end result is the same, so it's a good enough mental model.

The "memoized" code of this simple `App` component in reality will look like this:

```
const App = () => {
  const $ = _c(2);
  let t0;
  if ($[0] === Symbol.for('react.memo_cache_sentinel')) {
    t0 = [1, 2, 3];
    $[0] = t0;
  } else {
    t0 = $[0];
  }
  let t1;
  if ($[1] === Symbol.for('react.memo_cache_sentinel')) {
    t1 = (
      <HeavyComponent arrayProp={t0} callbackProp={_temp}>
        <ChildComponent />
      </HeavyComponent>
    );
    $[1] = t1;
  } else {
    t1 = $[1];
  }
  return t1;
};
function _temp() {
  console.log('Callback happened');
}
```

You can play around with it by yourself using the online React Compiler Playground[185]

Pay attention, for example, to how it transformed the inline callback into a `_temp` function and moved it outside of the `App` entirely. Or how, instead of memoizing `ChildComponent` and `HeavyComponent` separately, it grouped them together and stored them as `t1`. Smart, right?

What does it mean for our app's performance, however?

The Performance Impact of the Compiler

Given all that we already know, it should be easy to predict and measure the impact of the Compiler.

First, I would expect Interaction Performance to improve, sometimes by a lot. That search problem we struggled with - I expect the Compiler to be able to deal with it. So, INP in general should improve.

On the other hand, it generates more code than before and forces JavaScript to do more work during the initial critical rendering pass. So it's possible that:

- The bundle size might increase.
- The size of the JavaScript tasks in the "main" thread might increase.

That means it's quite possible that the initial load will worsen, and LCP will suffer as a consequence.

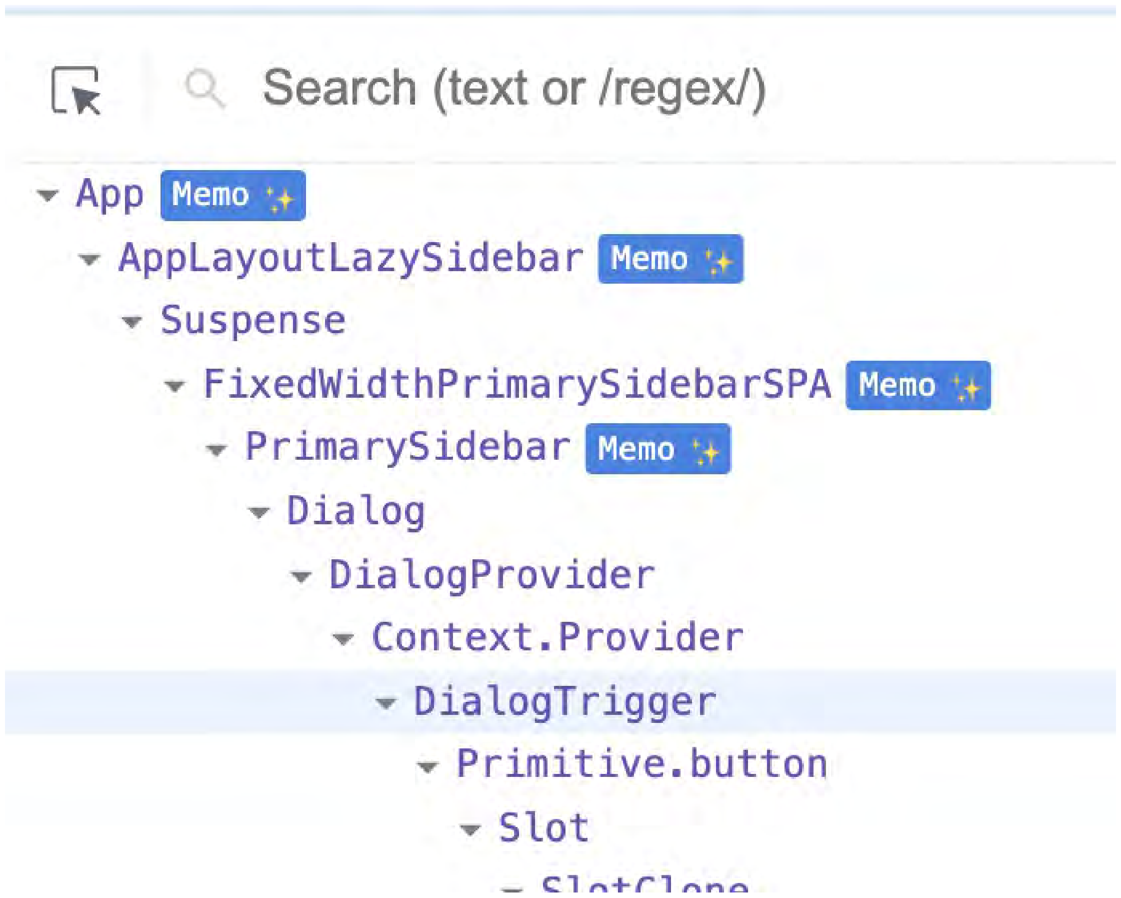
Now we just need to find out by how much.

If you want to try setting up the Compiler[186] by yourself, use the Study Project from the previous chapter. If you'd rather avoid doing that, jump to this chapter's Study Project. It's exactly the same as in the previous chapter, just with the Compiler enabled and working.

We'll start with measuring the impact on the initial load. Start it in dev mode to verify that the Compiler works:

```
npm run dev --workspace=chapter13-baseline-frontend
```

Open the project, open Chrome DevTools, and open the "Components" tab from React DevTools. You should see a bunch of Components with a "memo" label next to them:



That means the code is indeed transpiled by the Compiler.

Now disable the Compiler so that we can create a baseline for our measurements. In `vite.config.ts`, find the `react` plugin part and get rid of the object passed as an argument. Should be just this:

```
plugins: [react()];
```

Start it in dev mode again, and you'll see that the "Memo" label is gone. Now build it:

```
npm run build --workspace=chapter13-baseline-frontend
```

And write down the sizes of the produced bundles:

```
index-rlydUjSN.js                              4.23 kB
message-editor-fixed-DwcLLQhh.js               4.96 kB
inbox-Di1yL7b8.js                              9.58 kB
settings-DAAUbXpO.js                          11.93 kB
dashboard-nayv72DD.js                         12.55 kB
index-DuGuEtcS.js                             13.78 kB
fixed-width-primary-sidebar-spa-Ch2Umc5-.js   18.35 kB
date-fns-Ce044d0t.js                          20.94 kB
login-DGm_5nF8.js                             75.89 kB
radix-BqRmCkXQ.js                             81.95 kB
vendor-DuAMwyl9.js                           205.12 kB
editor-fQknsaKD.js                           295.79 kB
```

Start it:

```
npm run start --workspace=chapter13-baseline-frontend
```

Slow down the CPU to 20x and the Network to Fast 4G. Then write down:

- LCP for the "Home" page.
- The time when the table with statistics shows up.
- Average INP number if you type really fast in the Search field.
- Average INP number if you open/close the "Tasks" section in the Sidebar.
- Average INP number if you navigate from the Login page to the Home page (by clicking on the title).
- Average INP number if you navigate from the Settings page to the Home page.

They will fluctuate a bit, so measure a few times and take the median.

For me, the numbers are:

	Baseline
"Home" page LCP	1.3 s
"Statistics" table	2.3 s
Search INP	650 ms
Home INP from Login	350 ms
Home INP from Settings	260 ms

Now, restore the Compiler, build and start the project, and measure exactly the same things. Bundle sizes, with before and after values:

```
index-rlydUjSN.js                             4.23 kB --> 5.97 kB
message-editor-fixed-DwcLLQhh.js              4.96 kB --> 11.68 kB
inbox-Di1yL7b8.js                             9.58 kB --> 12.90 kB
settings-DAAUbXpO.js                         11.93 kB --> 16.55 kB
dashboard-nayv72DD.js                        12.55 kB --> 17.45 kB
index-DuGuEtcS.js                            13.78 kB --> 19.97 kB
fixed-width-primary-sidebar-spa-Ch2Umc5-.js  18.35 kB --> 20.94 kB
date-fns-CeO44d0t.js                         20.94 kB --> 26.92 kB
login-DGm_5nF8.js                            75.89 kB --> 77.71 kB
radix-BqRmCkXQ.js                            81.95 kB --> 81.95 kB
vendor-DuAMwyl9.js                          205.12 kB --> 211.00 kB
editor-fQknsaKD.js                          295.79 kB --> 295.79 kB
```

Exactly as predicted: the size of the JavaScript slightly increased. But the increase is not that huge, and this is not a compressed one. So it's not something I'd worry about.

Now to the measurements:

	Baseline	**Compiler**
"Home" page LCP	1.3 s	1.4 s
"Statistics" table	2.3 s	2.5 s
Search INP	650 ms	50 ms
Home INP from Login	353 ms	355 ms
Home INP from Settings	263 ms	260 ms

Again, pretty much exactly as predicted.

The initial load went down a little. This one is due to increased load on JavaScript, and it's only noticeable for me on a 20x CPU slowdown. If I reduce it to even 6x, the difference stops being measurable.

The INP for the search field improved more than 10 times, exactly like it was with our manual solutions. So the Compiler solved re-renders in this case completely. Great news!

INP for transitions from the Login and Settings pages to the Home page hasn't changed. Which is interesting. Because I included these transitions here with a secret hope that there would be a difference.

You see, when you navigate from Login to the Home page, the entire page changes. Everything on the Home page is mounted from scratch. However, when you navigate from *Settings* to the Home page, the entire top bar and the Sidebar remain exactly the same. So, in theory, they shouldn't even re-render, let alone re-mount, if they were memoized. So, in theory, the INP number should've improved.

Not Everything Can Be Caught by the Compiler

Sometimes It's Your Code

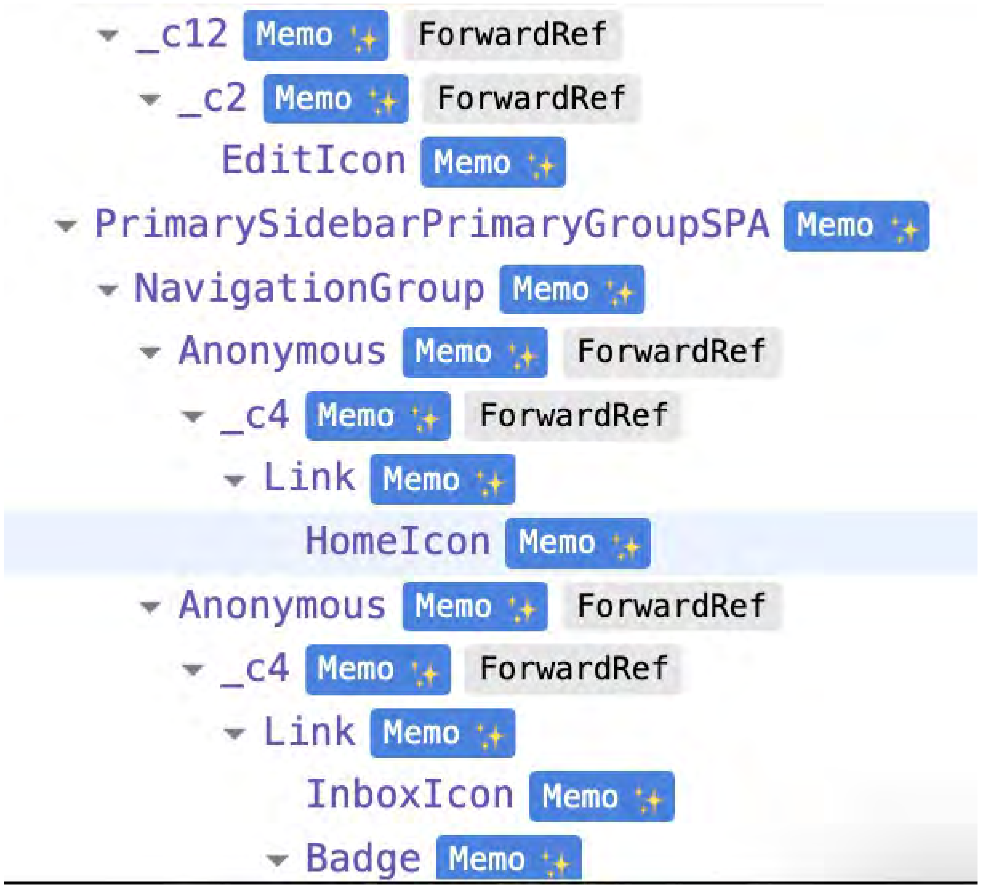
Does the lack of INP improvement when navigating to Home mean that something still re-renders despite the Compiler? It's quite easy to verify. Open the project in dev mode, turn on the "Highlight updates when components render" setting in the Profile tab, and navigate back and forth.

If you do this, you should be able to see that the top bar is not highlighted, and the Sidebar itself and some of its items are not highlighted. However, the "Home", "Inbox", etc., navigation items do re-render! And those have SVG icons, so it's quite plausible that re-rendering those items is the heaviest part, and that's why we didn't see any improvement.

Now we need to figure out what happened.

The very first thing to double-check when working with the Compiler is whether the

Compiler managed to memoize those items. Maybe something went wrong on its side? Open the Components tab in DevTools and select one of the items via the tab's select tool. You should see that every single component that composes those items has a "Memo" label next to it.



```
▾ _c12 Memo ✨ ForwardRef
  ▾ _c2 Memo ✨ ForwardRef
      EditIcon Memo ✨
▾ PrimarySidebarPrimaryGroupSPA Memo ✨
  ▾ NavigationGroup Memo ✨
    ▾ Anonymous Memo ✨ ForwardRef
      ▾ _c4 Memo ✨ ForwardRef
        ▾ Link Memo ✨
            HomeIcon Memo ✨
    ▾ Anonymous Memo ✨ ForwardRef
      ▾ _c4 Memo ✨ ForwardRef
        ▾ Link Memo ✨
            InboxIcon Memo ✨
          ▾ Badge Memo ✨
```

So memoization did happen. But it didn't help with re-renders.

The second thing is to always double-check that re-render actually happens and it's not DevTools false-positively highlighting something.

For this, trace down the implementation of the items themselves and add a good old `console.log` in `useEffect`:

```
// FILE: frontend/components/sidebar/navigation-items.tsx

export const SidebarRegularLinkItem = () => {
  useEffect(() => {
    console.log("SidebarRegularLinkItem renders");
  });

  ... // the rest of the code
};
```

`useEffect` with no dependencies will be triggered on every re-render of a component, which is exactly what we need.

After doing that, navigate back and forth between the Settings and Home screens. You should see a bunch of logs in the Console tab. So, re-renders indeed happen.

The final step is to investigate why. How to do that is a big question, though. There are no tools that can reliably determine the reason with 100% certainty, including the React DevTools. Sometimes they can give you hints, like "hook changed" or "props change", that may or may not be true. But that's it.

The only sure way to do that and find the actual reason, I found, is by turning things on and off and by knowing how React works. For example, in this case, inside the `SidebarRegularLinkItem`, you'll find this line:

```
const path = usePath();
```

If you comment it out, you'll see that none of the items re-render anymore. So clearly, this hook is the reason. If you navigate inside, you'll discover that this is our rudimentary implementation of client-side routing, which sets state every time the URL changes.

As we know, re-render happens on every state change. And since this hook is used in every item component, each of them re-renders on every state change, which happens on every URL change. This happens every time we navigate between pages.

The items re-render not because of some memoization issues or Compiler issues, but because the code is not optimized to minimize re-renders, plain and simple. It's 100% the developer's fault 🙈.

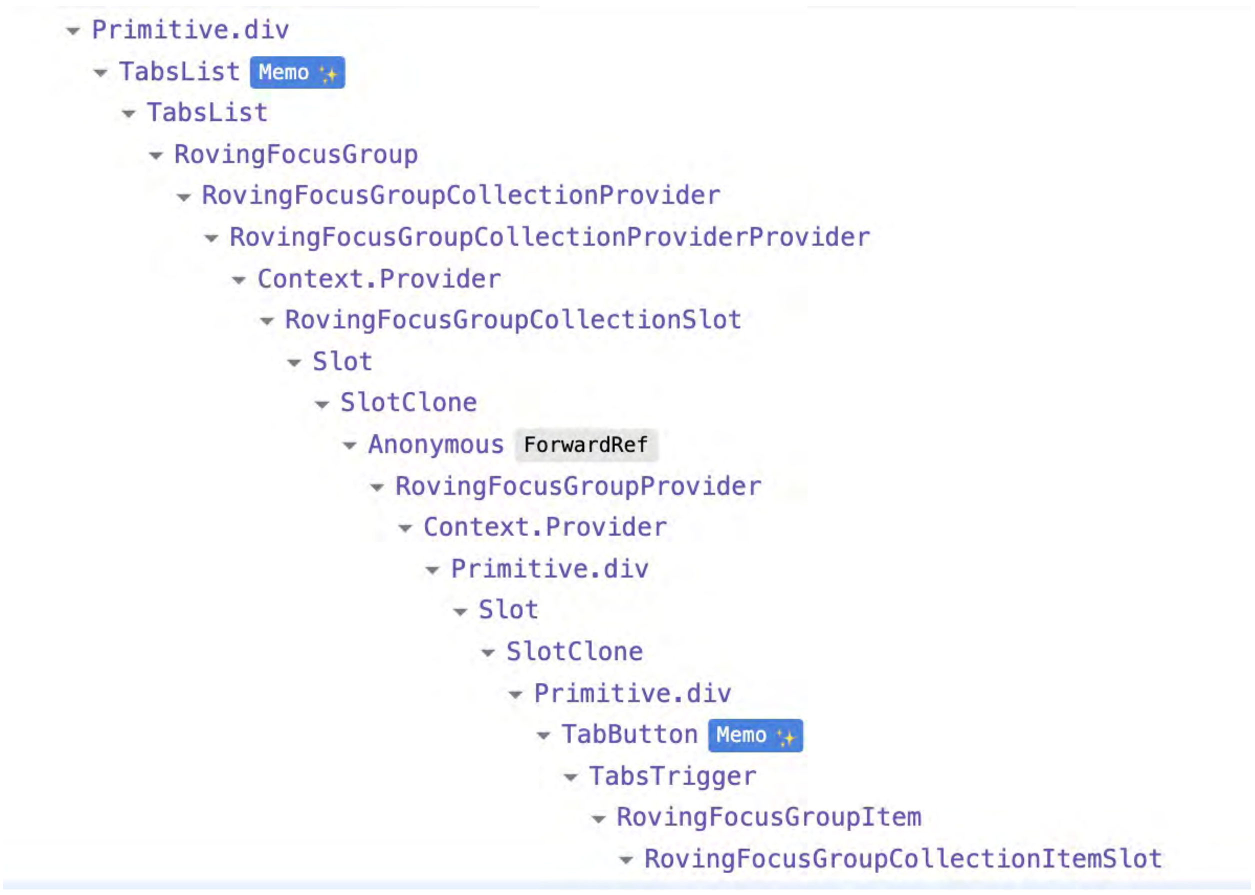
It's not always a matter of state, though.

Compiler and External Libraries

The Compiler transforms your code and yours only. Everything external that you pull as a dependency will be at the mercy of the people who support it. If they don't run their code through the Compiler before distributing the library, it will not be automatically memoized.

This will be especially important if you use UI component libraries, like MUI, Ant, or any other new and shiny UI toy. You can see what it will look like in the Study Project as well - it's built on the Radix UI library, and the version that the project is on wasn't precompiled.

If you open the Settings page and select any of the tab components in the Component tab, you'll see a picture like this:



```
▾ Primitive.div
  ▾ TabsList Memo ✨
    ▾ TabsList
      ▾ RovingFocusGroup
        ▾ RovingFocusGroupCollectionProvider
          ▾ RovingFocusGroupCollectionProviderProvider
            ▾ Context.Provider
              ▾ RovingFocusGroupCollectionSlot
                ▾ Slot
                  ▾ SlotClone
                    ▾ Anonymous ForwardRef
                      ▾ RovingFocusGroupProvider
                        ▾ Context.Provider
                          ▾ Primitive.div
                            ▾ Slot
                              ▾ SlotClone
                                ▾ Primitive.div
                                  ▾ TabButton Memo ✨
                                    ▾ TabsTrigger
                                      ▾ RovingFocusGroupItem
                                        ▾ RovingFocusGroupCollectionItemSlot
```

Only the `TabsList` and `TabButton` components have the "Memo" label. This is because only those two live in our code, and they are basically simple wrappers around Radix UI primitives. Everything else in between is used under the hood of those primitives inside the library itself. So, it's not going to be memoized by the Compiler on our side.

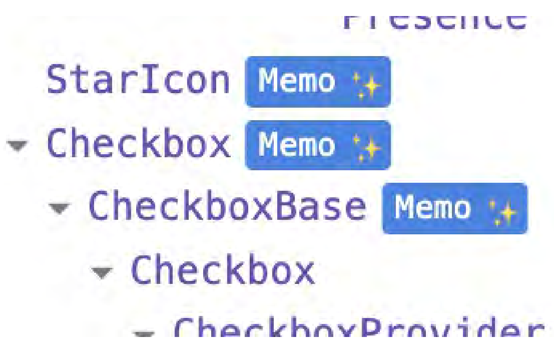
Sometimes It Just Doesn't Work

Another case of when memoization by the Compiler doesn't happen is when... it simply doesn't happen. Sometimes the code is so complicated or unusual that the Compiler tries its best but just fails. It's JavaScript it tries to parse, after all. And JavaScript is notorious for being very flexible and full of behavior that is ~~really weird~~ hard to make sense of.

Take a look, for example, at the Inbox page of the Study Project. If you hover over the list items on that page with the "highlight renders" enabled, you'll see that the Star icon and the Checkbox at the beginning of every item re-render.

The code for them lives in `frontend/patterns/messages-list-fixed.tsx`. The code is somewhat messy and not something I'd push to production. But the icons and checkboxes part is very straightforward:

```
export const MessageListFixed = () => {
  return (
    ... // bunch of code before and after
    <Checkbox />
    <Icons.Star className="text-blinkGold400 w-8 h-8" />
  )
}
```



And there is no state inside either of them that is tied to anything happening outside. So the Compiler should've worked. But it didn't 🤷‍♀️.

Have fun investigating why! It took me a while to figure it out, and I'm already used to investigations like this.

The problem here is this line down below in the code:

```
hoveredMessage === message.id;
```

If you comment out the entire block that is guarded by this condition, the re-renders of icons and checkboxes will stop.

The code, cleaned up from everything unnecessary, looks like this:

```
export const MessageListFixed = () => {
  const [hoveredMessage, setHoveredMessage] = useState(null);

  return messages.map((message) => (
    <div onMouseEnter={() => setHoveredMessage(message.id)} onMouseLeave={() =>
setHoveredMessage(null)}>
      <Checkbox />
      <Icons.Star />

      {hoveredMessage === message.id ? <div>content</div> : null}
    </div>
  ));
};
```

There is state that controls which message is hovered over. Then, there is a list of messages that render divs with "on hover" logic. And finally, there is a div that shows up with the additional buttons when a message is hovered over.

So my guess of what is happening here, based on what we know about how the Compiler works, is that it tried to optimize the memoization here by grouping all components inside the `map` loop into one. Basically, `Checkbox`, `Icons`, and the conditional div have to be assigned to a single variable all together. Then, this variable, from the Compiler's perspective, would depend on the `hoveredMessage` state. Which, as a result, means that memoization will update itself on every state change, i.e., on every hover in our case, and the entire thing would behave as if it's not memoized.

Copy-pasting that reduced example into the React Compiler Playground[187] produced the following code:

```
export const MessageListFixed = () => {
  const $ = _c(2);
  const [hoveredMessage, setHoveredMessage] = useState(null);
  let t0;
  if ($[0] !== hoveredMessage) {
    t0 = messages.map((message) => (
      <div onMouseEnter={() => setHoveredMessage(message.id)} onMouseLeave={() =>
setHoveredMessage(null)}>
        <Checkbox />
        <Icon />

        {hoveredMessage === message.id ? <div>content</div> : null}
      </div>
    ));
    $[0] = hoveredMessage;
    $[1] = t0;
  } else {
    t0 = $[1];
  }
  return t0;
};
```

Which is very similar to what I guessed - the entire mapped `messages` expression is actually assigned to a single t0 variable. The `if` statement above controls the dependency. And in our code, it changes with every hover. Mystery solved.

To be fair, by the time you're reading this, this exact issue might already have been solved. The React team constantly works on improving the Compiler, and numerous issues like these have already been fixed since its release. However, it's still JavaScript, so it's an uphill battle for them. So there will be something else.

Is It Worth It?

So, to summarize: the React Compiler, indeed, can improve interaction performance in your app, sometimes by a little, sometimes by a lot. However, it can't catch every single re-render: there will always be suboptimal usages of state somewhere, or external libraries that are not compiled, or just unusual code that the Compiler can't deal with yet.

I've tried it on a few apps over the last year, and it tends to catch between 40 and 80% of re-renders on a page for me, with various degrees of improvement.

The simpler and cleaner the components, the easier it is for the Compiler to correctly memoize them. We can see this even in the Study Project, where the incorrect memoization in the messages list component was almost entirely due to the fact that the component is too large and does too many things at the same time. And yes, I did it on purpose, if you were wondering 😅. So if you tend to write your code in a similar way, chances are, the Compiler will be less effective for you.

The impact on the initial load seems to be minimal, if present at all. It tends to increase bundle size, but not that significantly. In other projects where I tried the Compiler, the result was the same.

Setting it up is pretty straightforward if you're on the correct system that supports it. For the Study Project, I simply needed to install a plugin and include it in the configuration, and it worked. In Next.js, it's a matter of turning a flag on and off. On other apps I tried, it was a similar experience.

So, is it worth it or not? Up to you and your risk appetite and the health of the codebase, of course. If it works, it could be a performance improvement "for free", which also eliminates the need to do manual memoization as a nice bonus.

However, enabling it also means that you're introducing more "magic" into the way React code works, and you won't ever be able to tell whether a component will re-render or not by just looking at the code. It may or may not, depending on whether the Compiler worked as expected. Debugging the code when something goes wrong in this setup will be more challenging than ever.

14. Final Words

This is it! You're done with the book. Congratulations, this was not an easy achievement! Hope you enjoyed the read, and your performance journey will be an easy and successful one.

If you have any feedback, please don't hesitate to share it at feedback@getwebperf.com[188]. If you bought this book on Amazon, a public review there would always be appreciated. 😉

Feel free to connect with the author on LinkedIn[189], X/Twitter[190], or Bluesky[191] and ask any questions if something is not clear.

Grab the "Advanced React"[192] book with a 30% discount (code: **ADVREACT30**) if you feel like you want to learn about the intricacies of the framework in a similar investigative style.

Read the author's blog for more investigations that were not included in this book: https://www.developerway.com.

And teach everyone around you about web performance! You're qualified now.

Footnotes

[1]: https://nodejs.org/en

[2]: https://www.npmjs.com/

[3]: https://git-scm.com/

[4]: https://github.com/developerway/web-perf-fundamentals

[5]: https://www.google.com/intl/en_au/chrome/

[6]: https://react.dev/learn/react-developer-tools

[7]: https://www.deloitte.com/ie/en/services/consulting/research/milliseconds-make-millions.html

[8]: https://www.deloitte.com/content/dam/assets-zone2/ie/en/docs/services/consulting/2023/Milliseconds_Make_Millions_report.pdf

[9]: https://web.dev/case-studies/vodafone

[10]: https://web.dev/case-studies/quintoandar-inp

[11]: https://web.dev/case-studies/hotstar-inp

[12]: https://performance.shopify.com/

[13]: https://performance.shopify.com/blogs/blog/why-web-performance-still-matters

[14]: https://wpostats.com/

[15]: https://developer.chrome.com/docs/crux

[16]: https://developer.chrome.com/docs/crux/methodology#user-eligibility

[17]: https://developer.chrome.com/docs/crux/vis

[18]: https://developer.chrome.com/docs/crux/dashboard

[19]: https://developer.chrome.com/docs/crux/bigquery

[20]: https://developer.chrome.com/docs/crux/api

[21]: https://developers.google.com/publisher-tag/guides/monitor-performance

[22]: https://sentry.io/for/performance/

[23]: https://www.datadoghq.com/product/real-user-monitoring/

[24]: https://github.com/GoogleChrome/web-vitals

[25]: http://www.my-website.com

[26]: https://web.dev/articles/ttfb

[27]: https://web.dev/learn/performance/understanding-the-critical-path

[28]: https://web.dev/articles/fcp

[29]: https://web.dev/articles/fcp#what-is-a-good-fcp-score

[30]: https://web.dev/articles/lcp

[31]: https://web.dev/articles/vitals#core-web-vitals

[32]: https://web.dev/articles/vitals

[33]: https://developer.chrome.com/docs/lighthouse/overview

[34]: https://github.com/developerway/web-perf-fundamentals

[35]: http://localhost:3000

[36]: https://web.dev/articles/vitals#core-web-vitals

[37]: https://web.dev/learn/performance/optimize-resource-loading#async_versus_defer

[38]: https://aws.amazon.com/what-is/latency/

[39]: https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Europe%2CNorwaySweden

[40]: https://web.dev/articles/content-delivery-networks

[41]: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/304

[42]: https://developer.mozilla.org/en-US/docs/Web/HTTP/Conditional_requests

[43]: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

[44]: https://web.dev/

[45]: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

[46]: https://www.brendangregg.com/flamegraphs.html

[47]: https://developer.mozilla.org/en-US/docs/Glossary/Call_stack

[48]: https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path#layout

[49]: https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path

[50]: https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path#paint

[51]: https://web.dev/articles/inp

[52]: https://developers.google.com/search/blog/2023/05/introducing-inp

[53]: https://web.dev/articles/vitals

[54]: https://web.dev/articles/inp#good-score

[55]: https://ngrok.com/

[56]: https://developers.google.com/search/docs/crawling-indexing/javascript/javascript-seo-basics

[57]: https://developers.google.com/search/docs/crawling-indexing/large-site-managing-crawl-budget

[58]: https://hono.dev/

[59]: https://ogp.me/

[60]: https://ogp.me/#metadata

[61]: https://workers.cloudflare.com/

[62]: https://www.netlify.com/platform/core/functions/

[63]: https://vercel.com/docs/functions

[64]: https://aws.amazon.com/lambda/

[65]: https://aws.amazon.com/

[66]: https://azure.microsoft.com/

[67]: https://www.digitalocean.com/

[68]: https://react.dev/reference/react-dom/server

[69]: https://react.dev/reference/react-dom/server/renderToString

[70]: https://react.dev/reference/react-dom/client/hydrateRoot

[71]: https://vite.dev/guide/ssr

[72]: https://react.dev/reference/react-dom/server/renderToString

[73]: https://react.dev/reference/react-dom/server/renderToPipeableStream

[74]: https://github.com/reactwg/react-18/discussions/22

[75]: https://github.com/reactwg/react-18/discussions/37

[76]: https://gist.github.com/gaearon/e7d97cdf38a2907924ea12e4ebdf3c85

[77]: https://github.com/facebook/react/issues/14927

[78]: https://www.joshwcomeau.com/react/the-perils-of-rehydration/

[79]: https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation

[80]: https://www.gatsbyjs.com/

[81]: https://docusaurus.io/

[82]: https://astro.build/

[83]: https://developer.mozilla.org/en-US/docs/Glossary/gzip_compression

[84]: https://developer.mozilla.org/en-US/docs/Glossary/Brotli_compression

[85]: https://web.dev/articles/script-evaluation-and-long-tasks

[86]: https://web.dev/articles/tti

[87]: https://vite.dev/guide/build.html#chunking-strategy

[88]: https://webpack.js.org/guides/code-splitting/

[89]: https://en.wikipedia.org/wiki/Hash_function

[90]: https://rollupjs.org/configuration-options/#output-manualchunks

[91]: https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/rel/modulepreload

[92]: https://blog.bytebytego.com/p/http1-vs-http2-vs-http3-a-deep-dive

[93]: https://blog.bytebytego.com/p/http1-vs-http2-vs-http3-a-deep-dive

[94]: https://blog.cloudflare.com/http3-usage-one-year-on/#overall-request-distribution-by-http-version

[95]: https://hono.dev/docs/getting-started/nodejs#http2

[96]: https://github.com/expressjs/express/issues/5462

[97]: https://github.com/vercel/next.js/discussions/10842

[98]: https://web.dev/learn/performance/code-split-javascript

[99]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import

[100]: https://v8.dev/blog/cost-of-javascript-2019

[101]: https://github.com/btd/rollup-plugin-visualizer

[102]: https://docs.npmjs.com/about-scopes

[103]: https://mui.com/material-ui/getting-started/installation/

[104]: https://mui.com/material-ui/material-icons/

[105]: https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking

[106]: https://lodash.com/

[107]: https://lodash.com/docs/4.17.15

[108]: https://web.dev/articles/commonjs-larger-bundles

[109]: https://github.com/mgechev/is-esm

[110]: https://mui.com/material-ui/material-icons/?srsltid=AfmBOorYA_l6BMHjk-

Fz7brkwJeklR2j5EvtCB8h0pgQJiF2HOWK7uCU&query=star#search-material-icons

[111]: https://lodash.com/

[112]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/trim

[113]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase

[114]: https://date-fns.org/

[115]: https://momentjs.com/

[116]: https://moment.github.io/luxon/#/

[117]: https://floating-ui.com/

[118]: https://tailwindcss.com/

[119]: https://github.com/dcastil/tailwind-merge

[120]: https://emotion.sh/docs/introduction

[121]: https://blog.codinghorror.com/the-broken-window-theory/

[122]: https://www.npmjs.com/package/npm-why

[123]: https://mui.com/material-ui/react-snackbar/

[124]: https://www.radix-ui.com/primitives/docs/components/toast

[125]: https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/Lazy_loading

[126]: https://react.dev/reference/react/lazy

[127]: https://vite.dev/

[128]: https://react.dev/reference/react/lazy

[129]: https://react.dev/reference/react/Suspense

[130]: https://react.dev/reference/react/lazy

[131]: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Async_JS/Promises#async_and_await

[132]: https://github.com/developerway/web-perf-fundamentals

[133]: https://react.dev/learn/creating-a-react-app#full-stack-frameworks

[134]: https://react.dev/learn/creating-a-react-app#other-frameworks

[135]: https://tanstack.com/start/latest

[136]: https://tanstack.com/start/latest/docs/framework/react/quick-start

[137]: https://tanstack.com/start/latest/docs/framework/react/examples/quickstart-file-based

[138]: https://tanstack.com/start/latest/docs/framework/react/learn-the-basics#the-root-of-your-application

[139]: https://tanstack.com/start/latest/docs/framework/react/examples/quickstart-file-based

[140]: https://tanstack.com/router/latest/docs/framework/react/api/router

[141]: https://tanstack.com/router/latest/docs/framework/react/api/router/useLocationHoo

[142]: https://tanstack.com/router/latest/docs/framework/react/api/router/linkComponent

[143]: https://tanstack.com/router/latest/docs/framework/react/routing/installation-with-vite

[144]: https://tanstack.com/router/latest/docs/framework/react/routing/installation-with-webpack

[145]: https://tanstack.com/router/latest/docs/framework/react/guide/preloading

[146]: https://reactrouter.com/start/framework/installation

[147]: https://nextjs.org/docs/app/getting-started/installation

[148]: https://react.dev/reference/rsc/server-components

[149]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[150]: https://tanstack.com/query/latest/docs/framework/react/guides/prefetching

[151]: https://swr.vercel.app/docs/getting-started

[152]: https://2024.stateofreact.com/en-US/libraries/data-loading/

[153]: https://tanstack.com/query/latest/docs/framework/react/guides/queries

[154]: https://swr.vercel.app/docs/data-fetching

[155]: https://tanstack.com/query/latest/docs/framework/react/quick-start

[156]: https://swr.vercel.app/docs/getting-started

[157]: https://tanstack.com/start/latest/docs/framework/react/ssr

[158]: https://reactrouter.com/start/modes

[159]: https://nextjs.org/

[160]: https://nextjs.org/docs/app/getting-started/installation

[161]: https://nextjs.org/docs/pages/getting-started/installation

[162]: https://nextjs.org/docs/pages/getting-started/installation

[163]: https://nextjs.org/docs/pages/building-your-application/data-fetching

[164]: https://nextjs.org/docs/app

[165]: https://react.dev/reference/rsc/use-server

[166]: https://www.advanced-react.com/

[167]: https://nodejs.org/en/learn/modules/how-to-use-streams

[168]: https://www.corewebvitals.io/pagespeed/yield-to-main-thread

[169]: https://developer.mozilla.org/en-US/docs/Web/API/Scheduler/yield

[170]: https://chromewebstore.google.com/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

[171]: https://www.advanced-react.com/

[172]: https://github.com/pmndrs/zustand

[173]: https://redux.js.org/

[174]: https://react.dev/reference/react/createContext

[175]: https://react.dev/learn/react-compiler

[176]: https://react.dev/learn/react-compiler

[177]: https://react.dev/learn/react-compiler

[178]: https://babeljs.io/

[179]: https://www.youtube.com/watch?v=lGEMwh32soc

[180]: https://react.dev/blog/2024/10/21/react-compiler-beta-release

[181]: https://react.dev/blog/2025/04/21/react-compiler-rc