
ADVANCED
DATA-FETCHING
PATTERNS
IN REACT
JUNTAO QIU

Advanced Data Fetching Patterns in React

Fast, User-Friendly Data Fetching for Developers

Juntao Qiu

This book is for sale at
http://leanpub.com/react-data-fetching-patterns

This version was published on 2024-01-27

Contents

Preface

As your React applications grow richer with API integrations, do you find them increasingly bogged down and sluggish? You're wrestling with a common dilemma in the modern web development landscape: the more features and data connections you add, the more complex and slower your app becomes. Add to this the often baffling world of asynchronous programming, where debugging and troubleshooting can feel like navigating a labyrinth in the dark.

And it doesn't stop there. The React ecosystem is rapidly evolving, bombarding you with a flurry of new terms and concepts. **React Server Components**, **SSR (Server-Side Rendering)**, **Suspense API** – these aren't just fancy buzzwords; they're game-changers in how we think about building and optimizing our applications. But understanding and implementing them can feel overwhelming.

Embark on this journey with me, and together we'll take these intricate concepts and break them down into digestible, actionable insights. From unraveling the challenges of parallel requests and mastering lazy loading to demystifying SSR and exploring the cutting-edge realms of Server Components and Suspense, I've got you covered. You'll not only learn how to keep your applications lightning-fast but also discover strategies for efficient debugging and problem-solving in the asynchronous realm of React.

Join me, and transform the way you develop, optimize, and debug your React applications. It's time to turn those pain points into your strongest assets.

Chapter 1: Introduction

> Kicking off the tutorial 'Advanced Network Patterns in React', this chapter sets the stage for exploring diverse network request patterns in React applications. It establishes the foundational knowledge needed for handling complex network scenarios in frontend development.

In this chapter, you will learn the following content:

- Setting up the development environment with Vite and Tailwind CSS.
- Ensuring all necessary tools and configurations are in place for the tutorial.
- Introduction to the mock server for backend simulation.

This tutorial is designed to explore a range of patterns for executing network requests in React applications. Although the focus is on React, the principles and challenges discussed are relevant to other frontend libraries and frameworks.

Starting with a basic user profile, the tutorial progressively introduces more complex scenarios. These scenarios are intended to illuminate common problems and solutions encountered in large-scale applications, with a primary aim of demystifying the often-overlooked performance pitfalls in frontend development.

While React serves as the primary example, the patterns and strategies discussed are broadly applicable across various frontend technologies. They offer universal insights that can be valuable in diverse development contexts.

The tutorial assumes you have a foundational understanding of React, including familiarity with JSX and common hooks such as `useState` and `useEffect`.

Key learning outcomes of this tutorial include:

- Gaining a deep understanding of the challenges inherent in network programming and why it can be difficult to get right.
- Unraveling and demystifying the most commonly misunderstood yet widely used patterns.
- Exploring ways to make asynchronous service calls more manageable and less error-prone.
- Investigating alternative approaches for enhancing user experience.
- Learning how to apply different strategies in both frontend and backend development.
- Looking ahead to the future of server-side work and its implications for frontend development.

This tutorial aims to equip you with the knowledge and tools to navigate the complexities of network requests in React and other frontend frameworks, enhancing both your understanding and practical skills.

The page we're going to build is a `Home` page in an imaginary social media website. It doesn't do much but showing a user their home page when then log in.

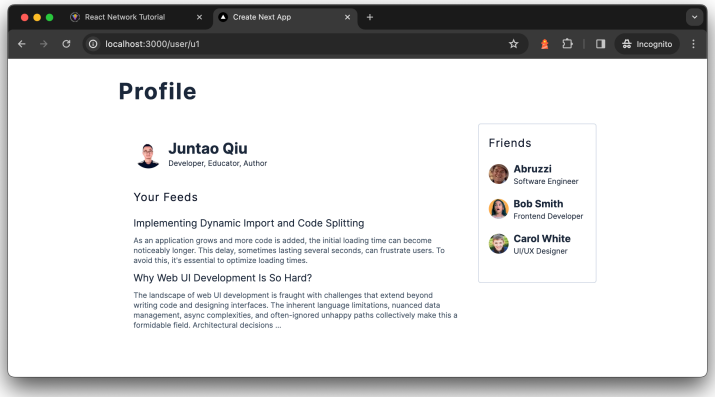

Figure 1. The home page we will build in the tutorial

Setting up the environment

We're going to use vite as the scaffolding tool to generate the structure of the application, use the following command to create a React with TypeScript enabled.

```
npm create vite@latest react-network-advanced -- --templa\
te react-ts
cd react-network-advanced
```

Let's clean up the generated template file, open up the `src/App.tsx`, and put the following code:

```
function App() {
  return (
       <div className="max-w-3xl m-auto my-4 text-slate-80\
0">
         <h1 className="text-4xl py-4 mb-4 tracking-wider \
font-bold">Profile</h1>
       </div>
  );
}

export default App;
```

We are going to use `Tailwind Css` for styling in this tutorial.

> Tailwind is a utility-first CSS framework packed with classes like flex, pt-4, text-center and rotate-90 that can be composed to build any design, directly in your markup.
>
> Utility-first CSS frameworks provide a comprehensive set of CSS utility classes for common styling tasks. Instead of writing custom CSS, developers can construct designs by combining these utility classes directly in their markup. This approach promotes rapid UI development, consistency across pages, and can lead to more maintainable codebases.

To install Tailwind, go to the `react-network-advanced` folder we created above, and execute the following command in Terminal (if you're on Mac OS)

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

And then you will need to config `Tailwind` to allow it scan `index.html` and all `tsx` files under `src` folder.

Figure 2. tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Lastly, you will need to use `@tailwind` directives in `src/index.css` to actually enable it:

Figure 3. src/index.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

I prefer to make the background a bit gray, so I'll add the following line in `src/index.css`

Figure 4. src/index.css

```
body {
  background-color: #fefefe;
}
```

With these changes in place, let's launch the application now in command line:

```
npm run dev
```

It by default will launch your React application on "http://localhost:5173/", And in your browser, you should be able to see the text `Profile`

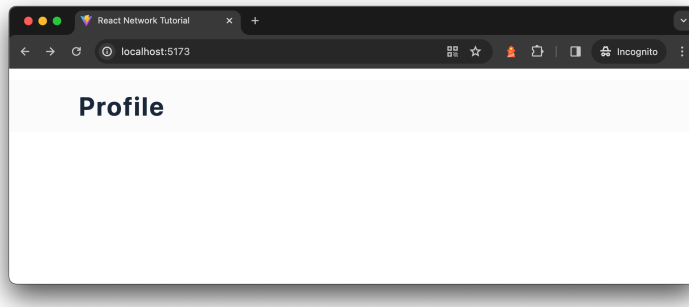

Figure 5. Check Vite and Tailwind CSS are working together

Setting up the backend service

We are going to call some API endpoints in the course of the tutorial, I have published them into a Github repo[1], go ahead and clone the repo to your local (assume it's in folder: `mock-server-network-react-tutorial`.

Run the following command to launch the mock server:

```
cd mock-server-network-react-tutorial
yarn start
```

You would be able to see something like this in your console:

[1] https://github.com/abruzzi/mock-server-network-react-tutorial

```
yarn run v1.22.19
$ node index.js
Mock server listening at http://localhost:1573
```

And if you try to access the one of the following API endpoint:

```
curl http://localhost:1573/users/u1
```

Or if you prefer, you could use `jq` to format the output, which is a bit easier to read (`curl http://localhost:1573/users/u1 | jq .`). And you should be able to see response like the following:

```
{
  "id": "u1",
  "name": "Juntao Qiu",
  "bio": "Developer, Educator, Author",
  "interests": [
    "Technology",
    "Outdoors",
    "Travel"
  ]
}
```

This introductory chapter equips learners with the essential setup and context for tackling advanced network patterns in React, paving the way for more complex concepts and practical applications in subsequent chapters. The next chapter will dive into using useEffect for building a basic profile page, demonstrating sequential network requests.

Chapter 2: Basics of Data Fetching in React

> This chapter dives into the essentials of data fetching in React applications, starting from a simple user profile page. It highlights the initial steps of making API calls, dealing with network delays, and managing state with React's useEffect hook.

In this chapter, you will learn the following content:

- Understanding and implementing basic data fetching using useEffect in React.
- Exploring the impact of network delays on frontend performance.
- Practical implementation of a user profile and friends list component with API integration.

Imagine a simple React application: a profile page where a logged-in user can view their profile. To achieve this, we need to fetch the user's information from an API using their ID.

The API endpoint we'll use returns basic user information. It's designed to include a delay, allowing us to later examine how slow API responses can impact frontend performance.

Consider this API call:

```
curl http://localhost:1573/users/u1
```

And the expected response:

```
{
  "id": "u1",
  "name": "Juntao Qiu",
  "bio": "Developer, Educator, Author",
  "interests": [
    "Technology",
    "Outdoors",
    "Travel"
  ]
}
```

In a typical React setup, we would handle this data fetching within a `useEffect` call. React triggers this effect after completing the initial render.

Here's how you might implement this in `src/profile.tsx`:

Figure 6. src/profile.tsx

```
const Profile = ({ id }: { id: string }) => {
  const [user, setUser] = useState<User | undefined>();

  useEffect(() => {
    const fetchUser = async () => {
      const data = await get<User>(`/users/${id}`);
      setUser(data);
    };

    fetchUser();
  }, [id]);

  return (
    <div>
```

```
      {user && user.name}
    </div>
  );
};
```

The `get` function is a straightforward wrapper around the native `fetch`. You can replace it with `axios.get` or any other preferred method.

Here's the utility function in `utils.ts`:

Figure 7. utils.ts

```
const baseurl = "http://localhost:1573";

async function get<T>(url: string): Promise<T> {
  const response = await fetch(`${baseurl}${url}`);

  if (!response.ok) {
    throw new Error("Network response was not ok");
  }

  return await response.json() as Promise<T>;
}

export { get };
```

To render the `Profile` component, update `App.tsx` as follows:

Figure 8. App.tsx

```tsx
import { Profile } from "./src/profile.tsx";

function App() {
  return (
    <div>
      <h1>Profile</h1>
      <div>
        <Profile id="u1" />
      </div>
    </div>
  );
}

export default App;
```

Visualizing the rendering sequence over time, it would look something like this diagram:

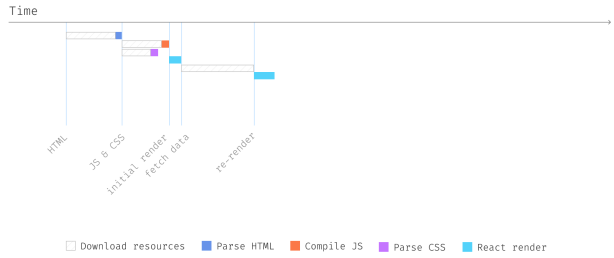

Figure 9. Fetch and then render

When a user accesses a React application, a typical flow begins with the browser downloading the initial HTML. As it parses the

HTML, it encounters links to resources like JS and CSS. This process involves downloading, parsing, and executing JS bundles, building the CSSOM, and so on.

> Note: HTML parsing is usually done in a streaming manner, meaning it starts as soon as some bytes are received rather than waiting for the entire HTML to download. For simplicity, our illustration assumes the entire HTML is downloaded before DOM construction. More details can be found in Rendering on the Web[1], Client-side rendering of HTML and interactivity[2], and Populating the page: how browsers work[3].

As JavaScript executes, React begins rendering and manipulating the DOM, then triggers a network request through `useEffect`. It waits until data returns from the server before re-rendering with the new data.

Adding loading and error handling

To enhance user experience, we can introduce a `Spinner` component during data loading and an `Error` component for handling issues like unresponsive backends or nonexistent users.

With these additions, the `Profile` component now includes additional states:

[1] https://web.dev/articles/rendering-on-the-web
[2] https://web.dev/articles/client-side-rendering-of-html-and-interactivity
[3] https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work

Figure 10. src/profile.tsx

```tsx
const Profile = ({ id }: { id: string }) => {
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<Error | undefined>();

  const [user, setUser] = useState<User | undefined>();

  useEffect(() => {
    const fetchUser = async () => {
      try {
        setLoading(true);
        const data = await get<User>(`/users/${id}`);
        setUser(data);
      } catch (e) {
        setError(e as Error);
      } finally {
        setLoading(false);
      }
    };

    fetchUser();
  }, [id]);

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Something went wrong...</div>;
  }

  return (
    <>
      {user && user.name}
    </>
```

```
  );
};
```

This structure should be familiar if you've worked with React before.

Next, let's add more than just the username. We'll create an `About` component to display user information, adding simple styles for visual appeal.

> For brevity, I'm omitting Tailwind CSS from the code snippets. You can view the full styled components in the corresponding code repository.

Figure 11. src/about.tsx

```tsx
const About = ({ user }: { user: User }) => {
  return (
    <div>
      <div>
        <img
          src={user.avatar}
          alt={`User ${user.name} Avatar`}
        />
      </div>
      <div>
        <div>{user.name}</div>
        <p>{user.bio}</p>
      </div>
    </div>
  );
};
```

When data is correctly fetched, it renders as shown:

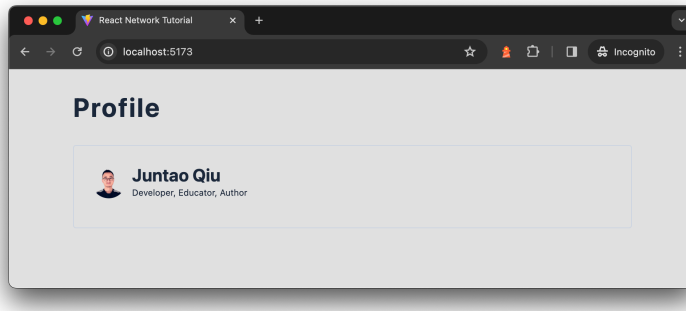

Figure 12. User basic information fetched and rendered

We now have a basic Profile page, retrieving data from a backend API intentionally delayed by 1.5 seconds.

Implementing the User's Friends List

Consider the user's friends list, typically stored in a separate table and accessed via a different API endpoint. For example, `/users/<id>/friends`. We'll fetch this data in a new component, `Friends`.

Figure 13. src/friends.tsx

```
1  const Friends = ({ id }: { id: string }) => {
2    const [loading, setLoading] = useState<boolean>(false);
3    const [users, setUsers] = useState<User[]>([]);
4
5    useEffect(() => {
6      const fetchFriends = async () => {
7        setLoading(true);
8        const data = await get<User>(`/users/${id}/friends`\
```

```
);
      setLoading(false);
      setUsers(data);
    };

    fetchFriends();
  }, [id]);

  if(loading) {
    return <div>Loading...</div>
  }

  return (
    <div>
      <h2>Friends</h2>
      <div>
        {users.map((user) => (
          <div>
            <img
              src={`https://i.pravatar.cc/150?u=${user.id\
}`}
              alt={`User ${user.name} avatar`}
            />
            <span>{user.name}</span>
          </div>
        ))}
      </div>
    </div>
  );
};

export { Friends };
```

The structure of the `Friends` component mirrors that of `Profile`: managing state, fetching data in `useEffect`, and rendering based

on loading, error, and successful data retrieval states.

We can incorporate `Friends` into the `Profile` component like any regular React component:

Figure 14. src/profile.tsx

```
const Profile = () => {
    //...

    return (
    <>
      {user && <About user={user} />}
      <Friends id={id} />
    </>
  );
}
```

At first glance, this implementation seems fine. However, if you consider the time taken for each API call, you might spot a potential issue. What if the `/friends` API also takes 1.5 seconds to respond? The total time to display the full page would be 3 seconds.

This chapter lays the groundwork for mastering network requests in React. It provides a practical example of fetching and rendering data, setting the stage for more advanced patterns and performance considerations in subsequent chapters. The next chapter will further explore complex data fetching scenarios, addressing performance challenges in frontend applications.

Chapter 3: Fetching Resources in Parallel

> Chapter 3 tackles the challenge of optimizing network requests in React applications. It focuses on implementing parallel data fetching strategies to minimize the impact of the network waterfall effect, enhancing application performance and user experience.

In this chapter, you will learn the following content:

- Understanding and mitigating the request waterfall effect.
- Implementing parallel requests for efficiency.
- Handling dependencies in network requests.

The first issue we encounter is with the rendering order. Initially, in the `Profile` component, `useEffect` triggers a network request. However, since data takes 1.5 seconds to return, we display a loading indicator in the meantime.

Once the data arrives, we render the `About` section, and a similar process occurs in the `Friends` component. Here, `useEffect` initiates another network request, waiting for the data to return.

Visualizing the request timeline, it looks like this:

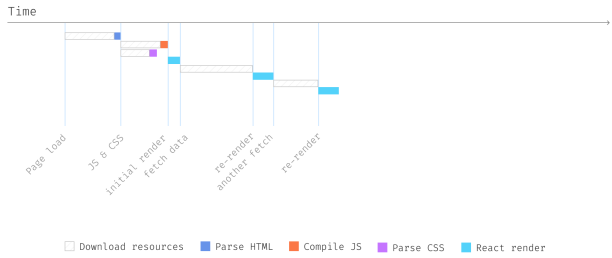

Figure 15. The request waterfall issue

The process involves three renderings. After the first render, the page displays a `loading...` message while initiating the `/users/u1` request. When the server responds, the `About` section is displayed. As `Friends` renders, lacking available data, it shows a `loading...` message in its section and sends out the `/users/u1/friends` request. Upon receiving this data, the third rendering occurs.

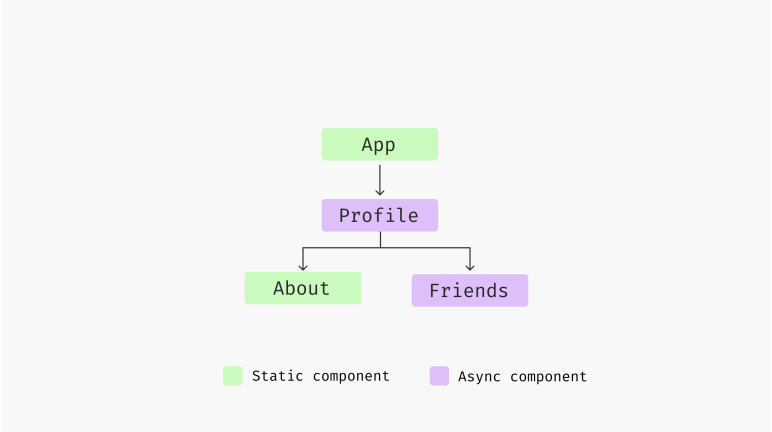

Figure 16. The component tree of About + Friends

This sequence might be obvious in our current setup, but consider more complex scenarios. Imagine the `Friends` component nested deeper in the component tree or used in different pages or sections. In such cases, identifying the problem by statically reading the code becomes challenging.

The situation worsens with more nested components following the same `useEffect + loading + error` pattern, potentially leading to cumulative performance issues:

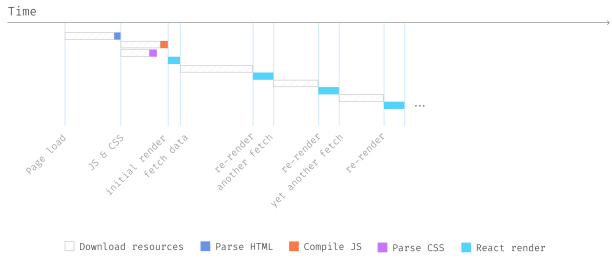

Figure 17. Request waterfall could be even worse

Over time, as the component tree grows, the page becomes increasingly slower.

However, one might wonder if initiating data fetching simultaneously could mitigate this wait time.

Sending Requests in Parallel

We can address this issue by sending parallel requests. In the `Profile` component, we can start both requests simultaneously using `Promise.all`, passing the fetched `friends` list to the `Friends` component:

Figure 18. src/profile.tsx

```tsx
const Profile = ({ id }: { id: string }) => {
    //...
  const [user, setUser] = useState<User | undefined>();
  const [friends, setFriends] = useState<User[]>([]);

  useEffect(() => {
    const fetchUserAndFriends = async () => {
      try {
        setLoading(true);

        const [user, friends] = await Promise.all([
          get<User>(`/users/${id}`),
          get<User[]>(`/users/${id}/friends`),
        ]);

        setUser(user);
        setFriends(friends);
      } catch (e) {
        setError(e as Error);
      } finally {
        setLoading(false);
      }
    };

    fetchUserAndFriends();
  }, [id]);

    //...
};

export { Profile };
```

> The `Promise.all()` static method accepts an iterable of promises, returning a single Promise. This promise resolves when all input promises fulfill (including for an empty iterable), resulting in an array of fulfillment values. If any input promises reject, the returned promise rejects with the first rejection reason.

Consequently, we modify `Friends` into a presentational component, responding only to the passed `users` list, rather than making its own requests:

Figure 19. src/friends.tsx

```
const Friends = ({ users }: { users: User[] }) => {
  return (
    <div>
      <h2>Friends</h2>
      <div>
        {users.map((user) => (
          <div>
            <img
              src={`https://i.pravatar.cc/150?u=${user.id\
}`}
              alt={`User ${user.name} avatar`}
            />
            <span>{user.name}</span>
          </div>
        ))}
      </div>
    </div>
  );
};

export { Friends };
```

Now, the total wait time is reduced to `max(1.5, 1.5) = 1.5` seconds, a significant improvement:

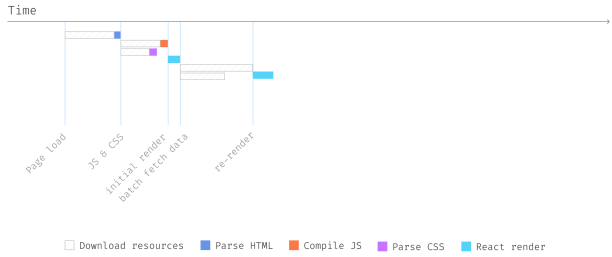

Figure 20. Send requests in parallel

The only remaining issue is the potential wait for the slower request in extreme cases. We'll accept this limitation for now and explore solutions in subsequent chapters.

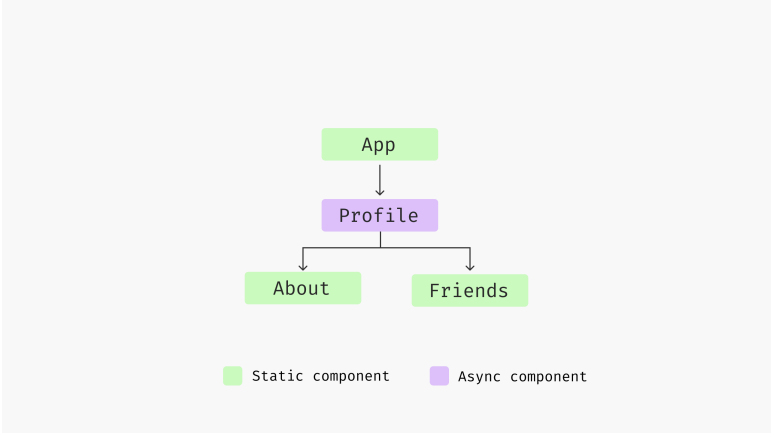

Figure 21. The only async component now is Profile

Request Dependency

Parallel requests expedite the loading of independent data. However, some requests depend on others. For example, we might need to fetch user information first and use the `interests` array from the response to retrieve recommended feeds for the user. This sequential dependency necessitates a return to the initial approach.

In the `Feeds` component, we define loading, error, and data states, and `useEffect` initiates network fetching after the initial render:

Figure 22. src/feeds.tsx

```tsx
const Feeds = ({ category }: { category: string }) => {
  const [loading, setLoading] = useState<boolean>(false);
  const [feeds, setFeeds] = useState<Feed[]>([]);

  useEffect(() => {
    const fetchFeeds = async () => {
      setLoading(true);
      const data = await get(`/articles/${category}`);

      setLoading(false);
      setFeeds(data);
    };

    fetchFeeds();
  }, [category]);

  if (loading) {
    return <div>Loading...</div>;
  }

  return (
    <div>
      <h2>Your Feeds</h2>
      <div>
        {feeds.map((feed) => (
          <>
            <h3>{feed.title}</h3>
            <p>{feed.description}</p>
          </>
        ))}
      </div>
    </div>
  );
};
```

In the `Profile` component, we include `Feeds` as follows:

Figure 23. src/profile.tsx

```
return (
    <>
      {user && <About user={user} />}
      <Friends users={friends} />
      {user && <Feeds category={user.interests[0]} />}
    </>
  );
```

Initially, `About` and `Friends` load, and as soon as the `user` data is available, we use `interests[0]` to fetch feeds, potentially taking another second. The overall wait time amounts to `max(1.5, 1.5) + 1 = 2.5` seconds.

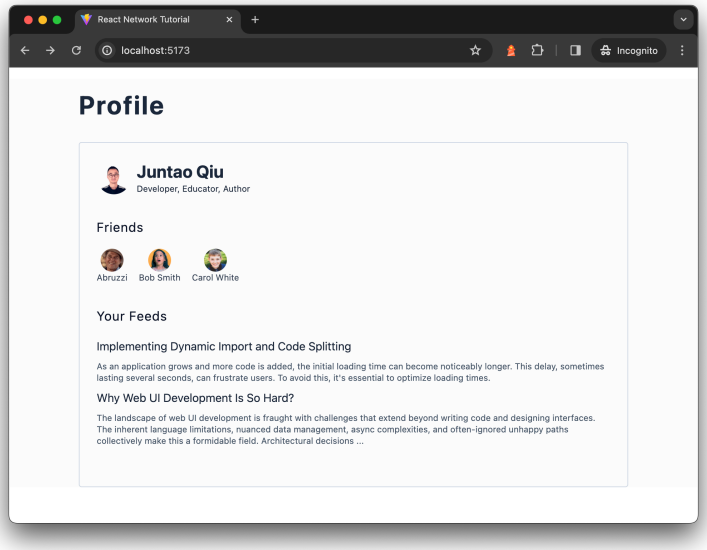

Figure 24. The UI after all components are rendered

This approach combines parallel and sequential requests, yielding better performance than the initial method.

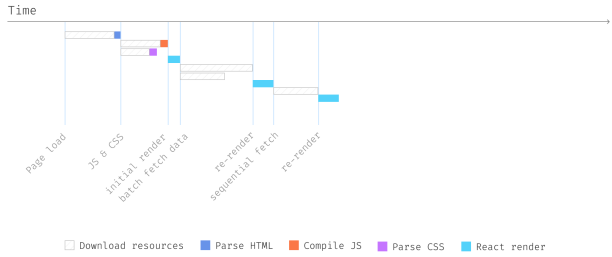

Figure 25. The mix of parallel and sequential requests

The `feeds` request must wait for the completion of the previous two requests, displaying a large `spinner` in the interim. While this solution is functional, we must consider the runtime data requirements for each specific user.

By mastering parallel requests and managing dependencies in network calls, this chapter sets the foundation for building faster and more responsive React applications. Join us as we continue to navigate the intricate world of advanced network patterns in React. In the next chapter, we explore further optimization strategies and delve into more complex scenarios of network fetching in React.

Chapter 4: Optimizing Friend List Interactions

> This chapter focuses on enhancing the user experience in React applications by implementing a detailed user profile popover. It explores the integration of external UI libraries like NextUI for building interactive features and discusses efficient data fetching strategies.

In this chapter, you will learn the following content:

- Leveraging NextUI for dynamic UI components.
- Efficient data fetching for enhanced features.
- Balancing performance with interactive design.

Let's delve further into typical frontend application scenarios, uncovering new patterns and potential avenues for performance improvement.

Consider enhancing the current Profile page. Suppose a user clicks on a friend's avatar, and we display a popover with additional details fetched from a `/users/1/details` endpoint. This feature, common in platforms like Twitter or LinkedIn, adds depth to user interaction.

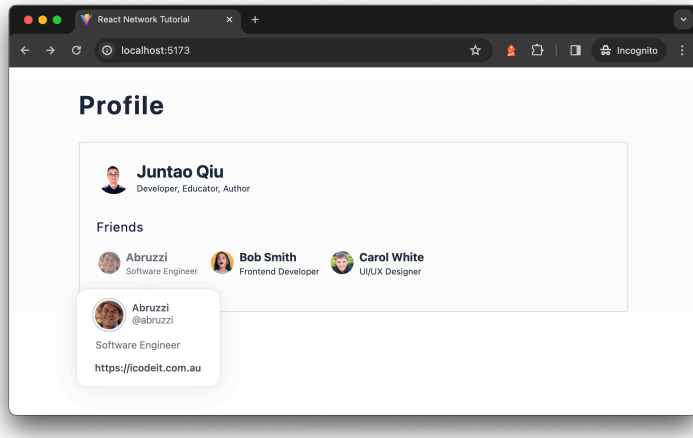

Figure 26. User Detail Popover

To maintain focus on our main topic, I'll skip the detailed implementation of the popover itself. Instead, we'll utilize components from `nextui` for the popover behavior and `UserDetailCard`.

> NextUI is a React UI library built on top of Tailwind CSS and React Aria, offering beautiful and accessible user interfaces. Despite its name similarity and website design, it's an independent community project and is not affiliated with Vercel or Next.js.

Install and config NextUI

Firstly, let's install NextUI into our project:

```
yarn add @nextui-org/react framer-motion
```

And the we will need to edit `tailwind.config.cjs`

Figure 27. tailwind.config.cjs

```
const {nextui} = require("@nextui-org/react");

/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    "./index.html",
    "./src/**/*.{jsx,tsx}",
    "./node_modules/@nextui-org/theme/dist/**/*.{js,ts,js\
x,tsx}",
  ],
  theme: {
    extend: {},
  },
  darkMode: "class",
  plugins: [nextui()],
}
```

And finally we'll need to wrap the Application with a `NextUIProvider`:

```
import { NextUIProvider } from "@nextui-org/react";

function App() {
  return (
    <NextUIProvider>
      <div>
      <h1>Profile</h1>
        <div>
          <Profile id="u1" />
        </div>
```

```
      </div>
    </NextUIProvider>
  );
}
```

Next let's implement the popover component with `Friend`.

Implementing a Popover Component

A popover is a non-modal dialog that appears adjacent to its trigger element. It's often used to display additional rich content.

Here's a basic implementation using `@nextui-org/react`:

```
import React from "react";
import {Popover, PopoverTrigger, PopoverContent, Button} \
from "@nextui-org/react";

export default function App() {
  return (
    <Popover placement="right">
      <PopoverTrigger>
        <Button>Open Popover</Button>
      </PopoverTrigger>
      <PopoverContent>
        <div>
          <div>Popover Content</div>
          <div>This is the popover content</div>
        </div>
      </PopoverContent>
    </Popover>
  );
}
```

Clicking the “Open Popover” button reveals a popover box to the right. This box contains a header “Popover Content” in bold, followed by a descriptive text. It’s styled with padding and font adjustments for better presentation.

Defining a Trigger Component

The `Friend` component can act as a trigger for the popover. We wrap it with `PopoverTrigger` as follows:

Figure 28. src/friend.tsx

```tsx
import { User } from "../types";
import { Popover, PopoverContent, PopoverTrigger } from "\
@nextui-org/react";
import { Brief } from "./brief.tsx";

import UserDetailCard from "./user-detail-card.tsx";

export const Friend = ({ user }: { user: User }) => {
  return (
    <Popover placement="bottom" showArrow offset={10}>
      <PopoverTrigger>
        <button>
          <Brief user={user} />
        </button>
      </PopoverTrigger>
      <PopoverContent>
        <UserDetailCard id={user.id} />
      </PopoverContent>
    </Popover>
  );
};
```

The `Brief` component accepts a `User` object and renders its details:

Figure 29. src/brief.tsx

```
export function Brief({user}: { user: User }) {
  return (
    <div>
      <div>
        <img
          src={`https://i.pravatar.cc/150?u=${user.id}`}
          alt={`User ${user.name} avatar`}
          width={32}
          height={32}
        />
      </div>
      <div>
        <div>{user.name}</div>
        <p>{user.bio}</p>
      </div>
    </div>
  );
}
```

A click on the `Brief` component activates the popover.

Implementing UserDetailCard Component (Fetching Data)

`UserDetailCard` is designed to fetch and display user details. The user detail includes:

Figure 30. types.ts

```ts
export type UserDetail = {
  id: string;
  name: string;
  bio: string;
  twitter: string;
  homepage: string;
};
```

We use our reusable `get` function to fetch these details from the `/users/<id>/details` endpoint:

Figure 31. src/user-detail-card.tsx

```tsx
export function UserDetailCard({ id }: { id: string }) {
  const [loading, setLoading] = useState<boolean>(false);
  const [detail, setDetail] = useState<UserDetail | undef\
ined>();

  useEffect(() => {
    const fetchFeeds = async () => {
      setLoading(true);
      const data = await get<UserDetail>(`/users/${id}/de\
tails`);

      setLoading(false);
      setDetail(data);
    };

    fetchFeeds();
  }, [id]);

  if (loading || !detail) {
    return <div>Loading...</div>;
  }

```

```
  return (
    <Card shadow="none">
      <CardHeader>
        <div>
          <Avatar
            isBordered
            radius="full"
            size="md"
            src={`https://i.pravatar.cc/150?u=${detail.id\
}`}
          />
          <div>
            <h4>{detail.name}</h4>
            <p>{detail.twitter}</p>
          </div>
        </div>
      </CardHeader>
      <CardBody>
        <p>{detail.bio}</p>
      </CardBody>
      <CardFooter>
        <div>
          <p>
            <a href={detail.homepage}>{detail.homepage}</\
a>
          </p>
        </div>
      </CardFooter>
    </Card>
  );
}

export default UserDetailCard;
```

If we could visualise the current component tree

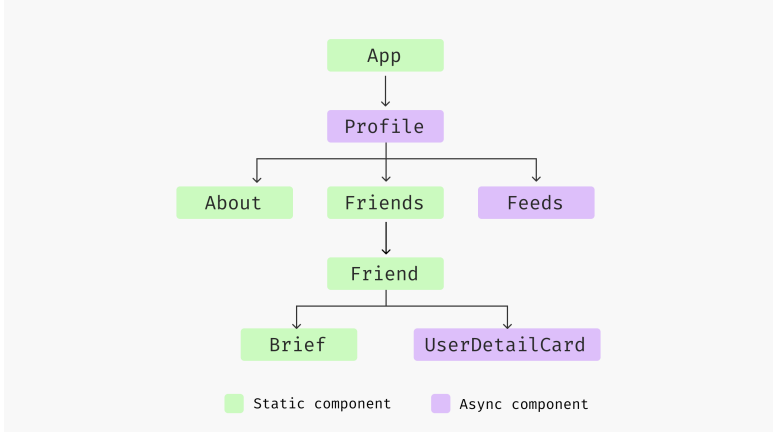

Figure 32. Component Tree with User Detail Card

This implementation appears efficient. However, a network inspection reveals increasing data transfer to the client as more third-party libraries are integrated. The additional JavaScript and CSS for the popover and `UserDetailCard` could be unnecessary for users who don't interact with these features.

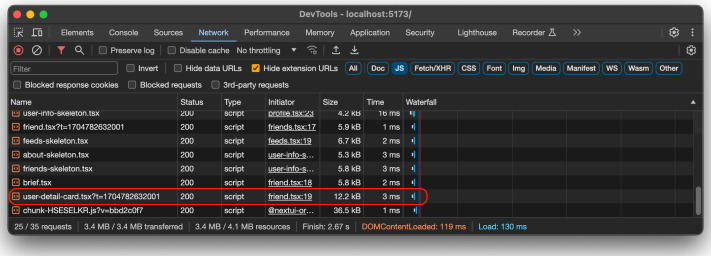

Figure 33. More data transferred through network

Is it possible to delay loading these resources until needed? For instance, only loading the `UserDetailCard` JS bundle when a user clicks on a `Friend` avatar, followed by a request to `/users/1/details` for detailed information. Let's find out in the

next chapter.

With the introduction of advanced UI elements and thoughtful data fetching strategies, this chapter elevates the user experience in React applications, paving the way for more engaging and efficient frontend designs. In the next chapter, we'll dive into code splitting and lazy load to reduce the initial load, that also the foundation of React concurrent we'll learn later.

Chapter 5: Leveraging Lazy Load and Suspense in React

> Chapter 5 of the 'Advanced Network Patterns in React' tutorial explores the concepts of Lazy Loading and React Suspense for optimizing performance. It demonstrates how to dynamically load components only when they are required, reducing initial load times and improving user experience.

In this chapter, you will learn the following content:

- Introduction to Lazy Loading in React
- Utilizing React Suspense for better loading handling
- Practical implementation in a user profile application

At the end of the previous chapter, we noticed that the page now has more bytes to load initially, which might not fair for user who don't hover on a Friend component - they still need to pay for the extra network requests and JavaScript bundles.

We could delay such extra (not immediate useful) content into another request as late as possible (maybe never if the users don't ask). For example, we could split `UserDetailCard` (and its dependency) into a separate JavaScript bundle and load it whenever the user hover on it.

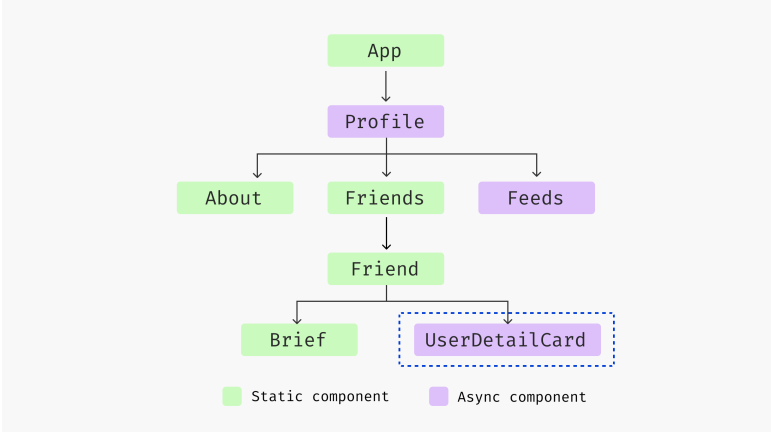

Figure 34. Separate UserDetailCard as another bundle

Let's see how to implement it in React with lazy load and suspense.

Introducing Lazy Loading

Lazy loading in React is a strategy to dynamically load components on-demand, improving performance for larger applications by minimizing the initial code load. React's `lazy` function, used alongside `Suspense`, renders dynamic imports as regular components, enhancing user experience and resource efficiency.

Consider this implementation:

```
import React, { Suspense, lazy } from 'react';

// Lazy load the component
const LazyComponent = lazy(() => import('./LazyComponent'\
));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

Here, `LazyComponent` is dynamically imported using `lazy()`. The `Suspense` component wraps around `LazyComponent`, providing a fallback UI during its loading phase. When `LazyComponent` is needed, it loads on demand, improving performance by splitting the code into smaller chunks.

> Code splitting is a technique in React that enables splitting your code into various bundles, which can then be loaded on demand or in parallel. This is particularly beneficial for improving the performance of large applications. When a user navigates to a part of your application that requires a component or library, only then does the necessary bundle get loaded, significantly reducing the initial load time of the application. This is especially useful for users with slower internet connections or on mobile devices. React's `lazy` func-

tion, coupled with `Suspense`, provides a straightforward way to implement code splitting, leading to more efficient resource usage and enhanced user experience.

Implementing UserDetailCard with lazy loading

Applying this concept, we've updated the `Friend` component in `src/friend.tsx` to use `React.lazy` for importing `UserDetailCard`. This change ensures that `UserDetailCard` loads only when necessary:

Figure 35. src/friend.tsx

```
import { User } from "../types";
import { Popover, PopoverContent, PopoverTrigger } from "\
@nextui-org/react";
import React, { Suspense } from "react";
import { Brief } from "./brief.tsx";

const UserDetailCard = React.lazy(() => import("./user-de\
tail-card.tsx"));

export const Friend = ({ user }: { user: User }) => {
  return (
    <Popover placement="bottom" showArrow offset={10}>
      <PopoverTrigger>
        <button>
          <Brief user={user} />
        </button>
      </PopoverTrigger>
      <PopoverContent>
```

```
        <Suspense fallback={<div>Loading...</div>}>
          <UserDetailCard id={user.id} />
        </Suspense>
      </PopoverContent>
    </Popover>
  );
};
```

The code defines a `Friend` component in React that displays user information using a popover. It imports user-related types and components from `@nextui-org/react` and a local `Brief` component for displaying a summary of the user.

The `UserDetailCard` component is dynamically imported using `React.lazy` for performance efficiency, loading only when needed.

```
const UserDetailCard = React.lazy(() => import("./user-de\
tail-card.tsx"));
```

Within the `Friend` component, a `Popover` is set up with its trigger wrapping a button that shows the `Brief` user summary. When clicked, the popover displays, containing the `UserDetailCard` within a `Suspense` component. This setup ensures a loading fallback is shown while the `UserDetailCard` loads, providing detailed user information based on the user's ID. This approach optimizes loading performance by fetching detailed user data only when the popover is activated.

```
<Suspense fallback={<div>Loading...</div>}>
  <UserDetailCard id={user.id} />
</Suspense>
```

> `Suspense` is a React component that lets your components "wait" for something before rendering. Initially introduced for React.lazy (lazy loading of components), its use has expanded to include data fetching and other asynchronous operations (we'll see how to do that in later Chapter with Next.js). `Suspense` provides a way to specify a fallback UI – for example, a loading indicator – that shows up while waiting for the component to load or data to be fetched. This helps in creating smoother user experiences in React applications, as it allows for more control over what gets displayed during the waiting period of an asynchronous operation. The integration of `Suspense` with lazy loading and other React features reflects the framework's ongoing evolution to meet modern web development needs.

This lazy loading approach is evident in the network panel of devtools, where two requests are made upon clicking `Friend`: one for the JavaScript bundle of `UserDetailCard`, and another for the user's details from the API.

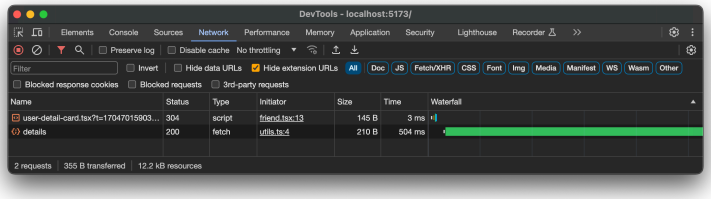

Figure 36. Waterfall with lazy load

Analysis the current bundles, it doesn't seem help a lot:

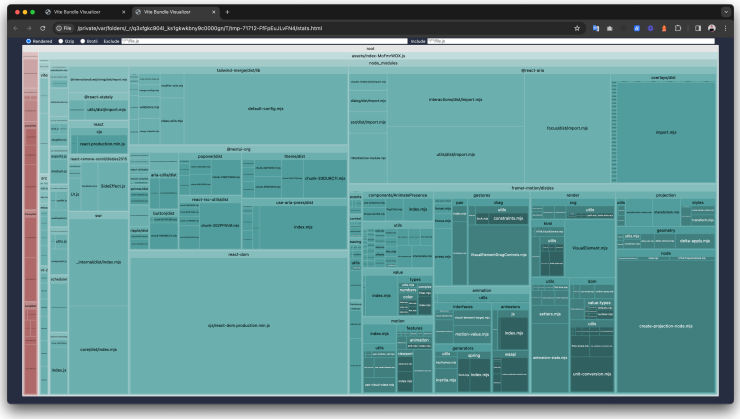

Figure 37. Bundle size analysis

Note in the chart above, the thin slice on the left hand side it the `UserDetailCard`, while the big one on the right is everything else. And if we look closely we'll find the biggest one is `framer-motion` - a package that adding the animation in React - shipped within NextUI. Obviously we don't really need animation for everything, it only used when the popover shows up.

We could further split the `Friend` into a separate bundle with NextUI components, and leave the index lightweight.

So firstly we don't import `Friend` in `Friends`, instead we lazy load it with suspense:

```
import { User } from "../types.ts";
import React, { Suspense } from "react";
import { FriendSkeleton } from "../misc/friend-skeleton.t\
sx";

const Friend = React.lazy(() => import("./friend.tsx"));

const Friends = ({ users }: { users: User[] }) => {
  return (
    <div>
      <h2>Friends</h2>
      <div>
        {users.map((user) => (
          <Suspense fallback={<FriendSkeleton />}>
            <Friend user={user} key={user.id} />
          </Suspense>
        ))}
      </div>
    </div>
  );
};

export { Friends };
```

And in the `Profile.tsx`, we will remove `NextUIProvider` and add it as a wrapper to `Friend`, because we don't need NextUI for the whole application but the popover in `Friend`.

Figure 38. src/friend.tsx

```
//...
const UserDetailCard = React.lazy(() => import("./user-de\
tail-card.tsx"));

const Friend = ({ user }: { user: User }) => {
  return (
    <NextUIProvider>
      <Popover placement="bottom" showArrow offset={10}>
        <PopoverTrigger>
          <button>
            <Brief user={user} />
          </button>
        </PopoverTrigger>
        <PopoverContent>
          <Suspense fallback={<div>Loading...</div>}>
            <UserDetailCard id={user.id} />
          </Suspense>
        </PopoverContent>
      </Popover>
    </NextUIProvider>
  );
};

export default Friend;
```

With these updates, our new analysis reveals that we have three distinct bundles: `UserDetail`, `Friend`, and `Profile`.

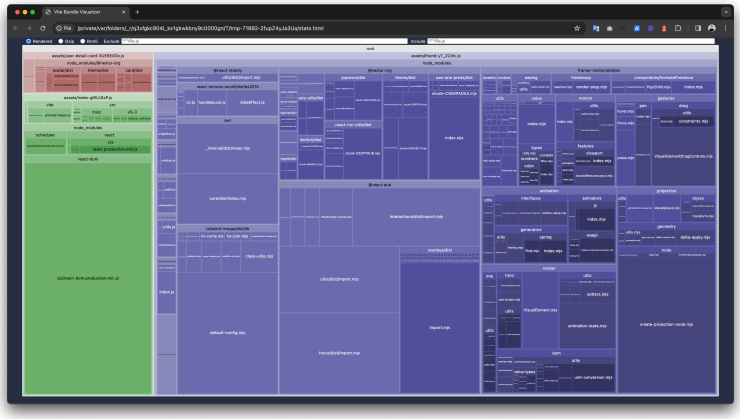

Figure 39. Bundle size analysis after splitting Friend

The largest bundle, shown in blue, corresponds to the `Friend` component. The smaller, red-tinted one at the top-left is the `UserDetailCard`, and the green-tinted one represents the `Profile` component.

This is a significant improvement. Now, the loading process works as follows: When the `Profile` component initiates parallel requests for `/users/u1` and `/users/u1/friends`, we display skeleton screens for the About section and the Friends list. As the friends data arrives and the `Friends` component starts rendering, the browser concurrently downloads the related bundle. During this time, a `FriendSkeleton` is displayed, transitioning to the `Friend` component once the bundle is fully downloaded.

Moreover, if the user doesn't hover over a `Friend`, there's no additional action. The `UserDetail` data is fetched only when the user hovers over a `Friend`, optimizing resource loading and enhancing performance.

The potential issue

Visualizing the request process, we see a sequence similar to the network waterfall discussed in Chapter 2:

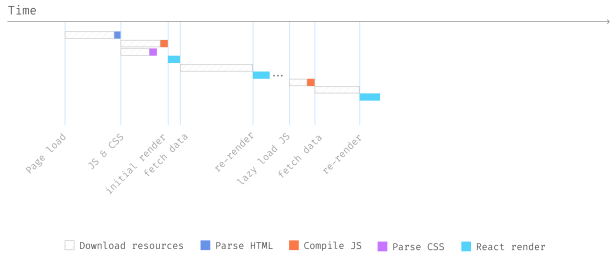

Figure 40. Waterfall with lazy load visualized

This observation leads to a question: is it possible to parallelize these requests to further reduce delays? Exploring this possibility could unlock more performance enhancements, especially in complex application structures.

This chapter is a deep dive into optimizing React applications using Lazy Load and Suspense. It provides practical examples and insights into improving load times and overall application efficiency. In the next chapter, we'll explore more advanced networking patterns and continue enhancing the performance and user experience of React applications.

Chapter 6: Prefetching Techniques in React Applications

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Introducing SWR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Implementing SWR for Preloading

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Integrating `preload` with SWR

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Chapter 7: Introducing Server Side Rendering

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Introducing Next.js

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

React Server Components

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Chapter 8: Introducing Static Site Generation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Mixing Server-Side Rendering and Static Site Generation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Chapter 9: Optimizing User Experience with Skeleton Loading in React

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Chapter 10: The New Suspense API in React

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Re-Introducing Suspense & Fallback

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Using skeletons in different layers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Streaming in Next.js

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Optimizing UI by Grouping Related Data Components

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Chapter 11: Lazy Load, Dynamic Import, and Preload in Next.js

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Dynamic Load in Next.js

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Implementing the UserDetailCard

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Preload in Next.js

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Client Component Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

First Version - Entire Component as Client-Side

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns

Second Version - Client Logic at Leaf Node

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/react-data-fetching-patterns