

Be part of a better internet. [Get 20% off membership for a limited time](#)

5 Advanced Data-Fetching Techniques in React for Enhanced User Experience



Juntao Qiu  · [Follow](#)

Published in ITNEXT · 10 min read · Jan 16, 2024



458



6



Data fetching is a critical yet challenging aspect of modern application development. The unpredictable nature of network conditions — slow connections, system errors, downtimes — along with the inherent complexity of asynchronous programming can significantly impact the functionality and reliability of your application. Moreover, inefficient data fetching strategies can lead to performance bottlenecks, prolong loading times, and detract from user experience.

In this article, I'll delve into five data fetching patterns that can optimize your React application's performance and enhance user experience. If you would like some in-depth material and practical examples on this topic, have a look at my free tutorial here: [Advanced Data-Fetching Patterns in React](#).

This tutorial is designed to complement the concepts we'll discuss here and provide you with a hands-on learning experience.

I have also published it as a book. If you prefer to read it in [PDF or EPUB](#), you can grab [a copy on leanpub](#).



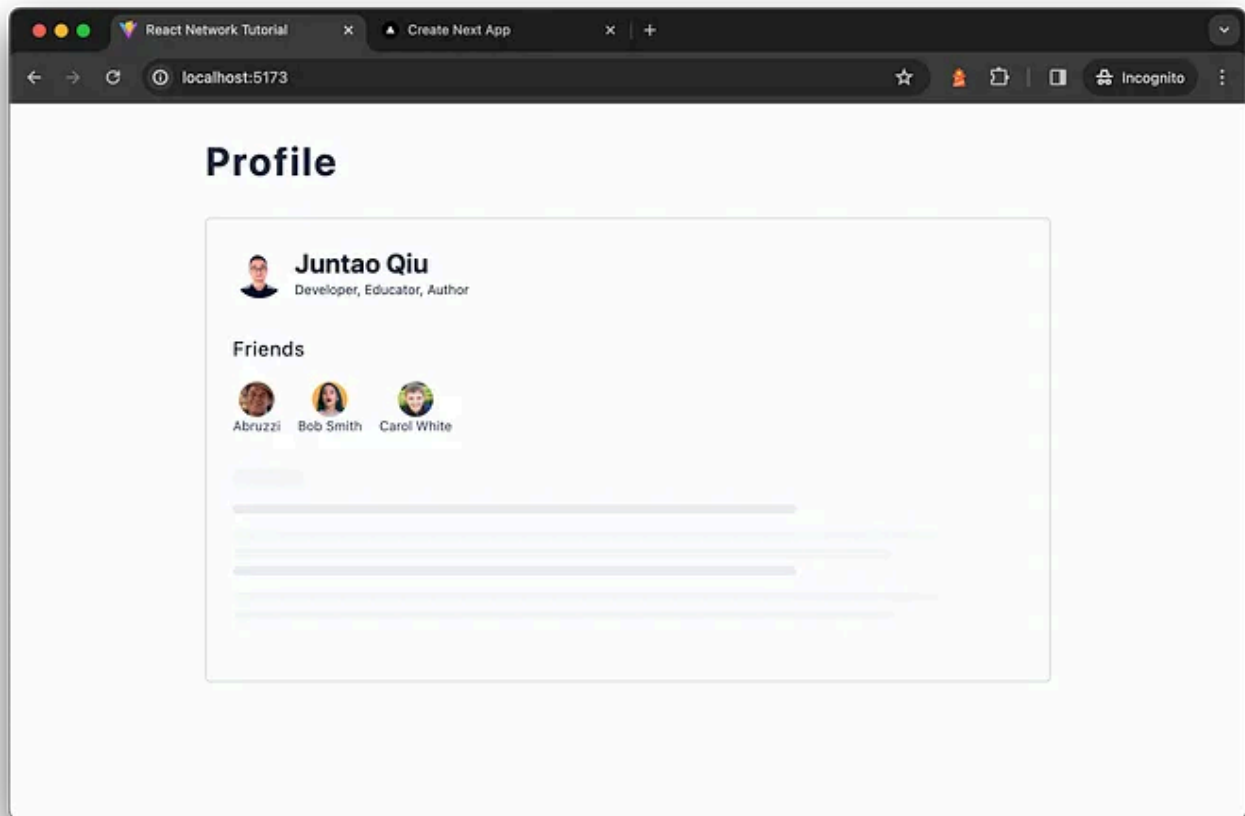
Photo by [Jon Robinson](#) on [Unsplash](#)

1. Parallel Data Requests

Introduction to Parallel Data Requests

Parallel data requests involve fetching multiple data sources simultaneously rather than sequentially. This technique is particularly beneficial when your application needs to retrieve data from independent sources or APIs, and none of these requests depend on the results of the others. By executing these requests in parallel, you can significantly reduce the overall loading time of your application.

Think of we have a page below. The user About section and Friends section need to fetch data separately on `/users/<id>` and `/users/<id>/friends`. If we fetch data in each component itself, it will cause the request waterfall issue.



The user profile page

```
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 (  

```

```

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

```

Above we fetch data for the basic user information, and in `Friends`:

```

const Friends = ({ id }: { id: string }) => {
  const [users, setUsers] = useState<User[]>([]);

  useEffect(() => {
    const fetchFriends = async () => {
      const data = await get<User[]>(`/users/${id}/friends`);
      setUsers(data);
    };
    fetchFriends();
  }, [id]);

  return (
    <div>
      <h2>Friends</h2>
      <div>
        {users.map((user) => (
          <Friend key={user.id} user={user} />
        ))}
      </div>
    </div>
  );
};

```

We could simply modify the code to request data all together in `Profile`:

```

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

```

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

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

This way, we can demolish the waterfall issue, and the application renders faster (still depends on the slowest request to return, but faster than before).

When to Use

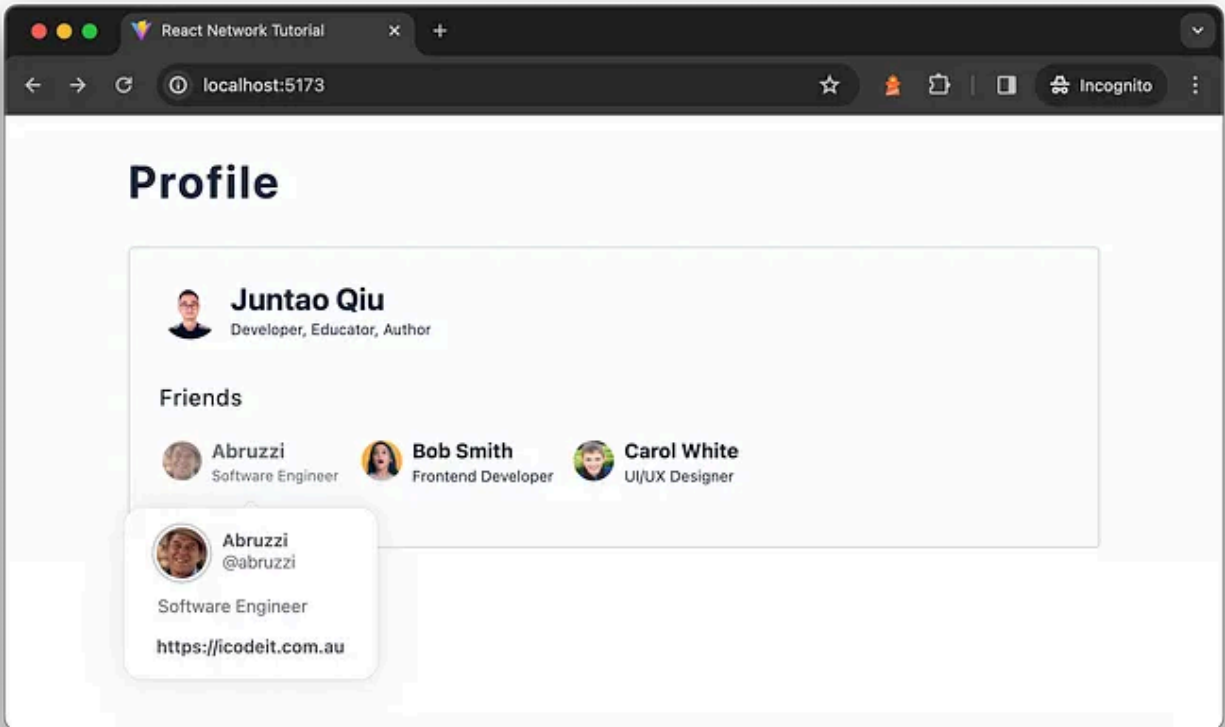
Use parallel data requests in scenarios where your application requires data from multiple endpoints that aren't interdependent. This is common in dashboards, complex forms, or pages that display aggregated data from various sources. Implementing parallel requests ensures that the user doesn't have to wait unnecessarily for sequential network calls to complete, thus enhancing the responsiveness and efficiency of your app.

2. Lazy Load + Suspense (Code Split)

Introduction to Lazy Load and Suspense

Lazy loading, combined with React's Suspense, allows you to split your application into multiple chunks that are loaded only when they are needed.

This method drastically reduces the initial load time by ensuring that users download only the essential code at first. Suspense acts as a placeholder for your components or data until the required code chunk or data is loaded.



User Detail Card when hover

We could use `React.lazy` and `Suspense` API to make it faster only requires a JavaScript bundle when needed.

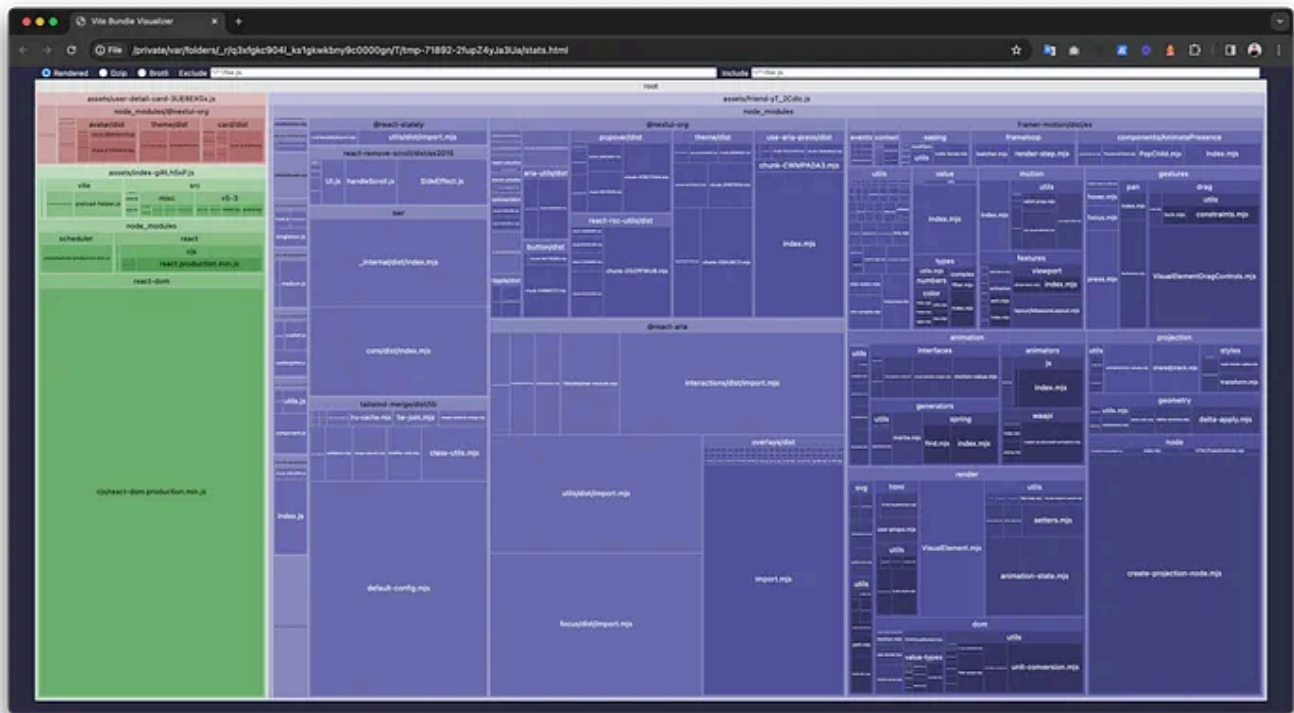
```
const UserDetailCard = React.lazy(() => import("./user-detail-card.tsx"));

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

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

`Friend` uses React's lazy loading feature to dynamically load a `UserDetailCard` component within a popover. The `UserDetailCard` is not loaded until it is actually rendered, which is managed by the `React.lazy` function. This enhances performance by reducing the initial load time of the application. In the `Friend` component, a popover is created with a button as its trigger. When this button is clicked, the popover displays, and the `UserDetailCard` component, encapsulated within a `Suspense` component, loads. The `Suspense` component provides a loading fallback (`<div>Loading...</div>`) until `UserDetailCard` is ready to be displayed. This setup optimizes resource loading and improves user experience by showing detailed user information in a popover on demand.

The bundler (web pack or vite, for example) will pack the lazy load into a separate JavaScript file (along with their dependencies), and that could reduce the JavaScript downloaded when the page loads initially.



Code splitting with React.lazy

The image above has three separate JavaScript bundles which are loaded separately when required.

When to Use

This approach is ideal for large applications with many routes and components, particularly when certain parts of the app are not immediately necessary for the initial user interaction. Use lazy load and Suspense for heavy components, large libraries, or additional routes that are not critical for the initial rendering.

3. Preload Data Before Interaction

Introduction to Preloading Data

Preloading data refers to fetching data in advance of the user's interaction with a component that requires this data. This proactive approach ensures

that the necessary data is already available by the time the user interacts with the component, providing a smoother and more responsive experience.

```
import { preload } from "swr";

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

const Friend = ({ user }: { user: User }) => {

  const handleMouseEnter = () => {
    preload(`/user/${user.id}/details`, () => getUserDetail(user.id));
  };

  return (
    <NextUIProvider>
      <Popover placement="bottom" showArrow offset={10}>
        <PopoverTrigger>
          <button
            onMouseEnter={handleMouseEnter}
          >
            <Brief user={user} />
          </button>
        </PopoverTrigger>
        <PopoverContent>
          { /* UserDetailCard */ }
        </PopoverContent>
      </NextUIProvider>
    </Popover>
  );
};
```

[Open in app ↗](#)**Medium** Search Write

Library like TanStack Query, SWR all supports such preload mechanism, the code example above using swr for preload and in the actual component when we do fetch, the data should already be there - so the user experience would be much better.

When to Use

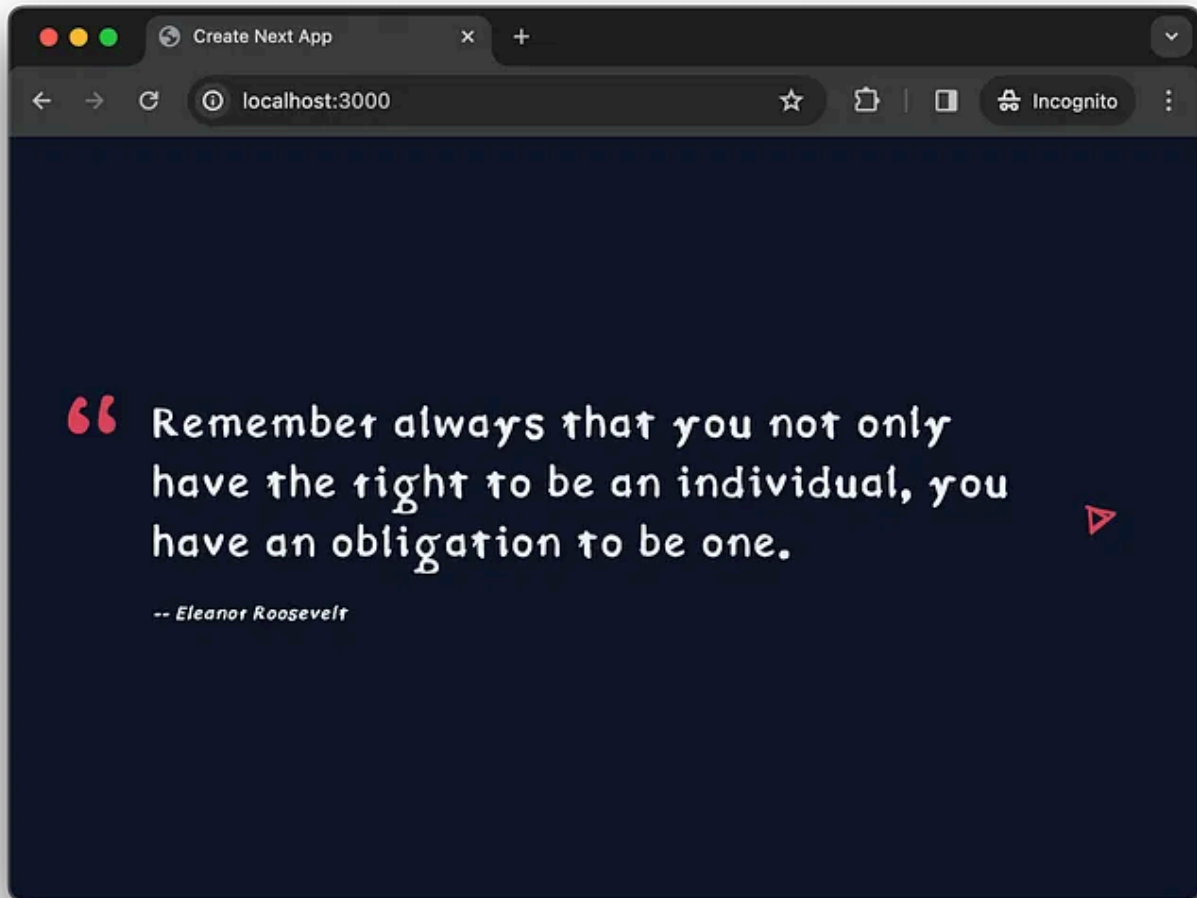
Preload data when user interactions are predictable, and you can anticipate the data they need next. This is particularly effective in user interfaces where certain actions, like hovering over or clicking a specific element, are expected to lead to data-dependent interactions.

4. Static Site Generation (SSG)

Introduction to Static Site Generation

Static Site Generation is a technique where the HTML pages of a website are generated at build time. This method often incorporates data that doesn't change frequently, allowing for pages to be served quickly without the need for server-side computations or database queries at runtime.

Let's say I have a quote of the day application, at the build time we could pre-render a few quotes for the users to get some thing to see when they firstly landed on the application.



Quote of the day application

In the following example, I'm using Next.js to populate data into Home page.

```
async function getQuotes(): Promise<QuoteType[]> {
  const res = await fetch(
    "https://api.quotable.io/quotes/random?tags=happiness,famous-quotes&limit=3"
  );

  return res.json();
}

export default async function Home() {
  const quotes = await getQuotes();
  return (
    <Quote initQuotes={quotes} />
  );
}
```

```
);  
}
```

In Next.js, the above request will be sent at build time, and when the user accesses the application in the browser, the quotes will already be part of the HTML.

When to Use

Static Site Generation is best used for content that changes infrequently, such as blog posts, documentation, e-commerce product listings, or landing pages. It's ideal for improving performance in scenarios where the content can be pre-rendered and served directly from a CDN, thereby reducing server load and speeding up content delivery.

5. React Server Component (Streaming)

Introduction to React Server Components

React Server Components allow for rendering components on the server and streaming them to the client. This method enables sending only the necessary data and minimal JavaScript to the client, reducing the amount of code that needs to be downloaded, parsed, and executed, thereby speeding up the interaction time.

Let's rethink the `Profile` example above. With the React Server Component, we can define the following components first:

```
import { Suspense } from "react";  
  
async function UserInfo({ id }: { id: string }) {
```

```
const user = await getUser(id);

return (
  <>
    <About user={user} />
    <Suspense fallback={<FeedsSkeleton />}>
      <Feeds category={user.interests[0]} />
    </Suspense>
  </>
);
}

async function Friends({ id }: { id: string }) {
  const friends = await getFriends(id);

  return (
    <div>
      <h2>Friends</h2>
      <div>
        {friends.map((user) => (
          <Friend user={user} key={user.id} />
        ))}
      </div>
    </div>
  );
}
```

The code snippet presents two asynchronous React components, `UserInfo` and `Friends`, which fetch and display user-related information.

In `UserInfo`, an asynchronous function fetches user details using `getUser(id)`. The component renders an `<About>` component with the fetched user data and a `<Feeds>` component inside a `Suspense` block. The `Suspense` component uses `<FeedsSkeleton>` as a fallback, displaying it until `<Feeds>` has finished loading the user's interests.

The `Friends` component, also asynchronous, retrieves a list of friends for a given user ID through `getFriends(id)`. It renders a list of friends, where each friend is displayed using the `<Friend>` component, passed with individual

user data. The `map` function iterates over the `friends` array, rendering a `<Friend>` component for each item with a unique `key` prop.

```
export async function Profile({ id }: { id: string }) {  
  return (  
    <div>  
      <h1>Profile</h1>  
      <div>  
        <div>  
          <Suspense fallback={<UserInfoSkeleton />}>  
            <UserInfo id={id} />  
          </Suspense>  
        </div>  
        <div>  
          <Suspense fallback={<FriendsSkeleton />}>  
            <Friends id={id} />  
          </Suspense>  
        </div>  
      </div>  
    </div>  
  );  
}
```

In this setup, `Profile` is an asynchronous component that renders a user's profile information. It uses two child components, `UserInfo` and `Friends`, each wrapped in a `Suspense` component. The `Suspense` components provide respective fallback skeletons (`<UserInfoSkeleton />` and `<FriendsSkeleton />`) which are displayed while the data for `UserInfo` and `Friends` is being fetched and loaded.

In a scenario where `Profile`, `UserInfo`, and `Friends` are React Server Components:

1. **Server-Side Execution:** These components would execute on the server. This means the server handles fetching the user information and friends'

data, rather than offloading these tasks to the client's browser.

2. **Streaming to Client:** After the server-side execution, only the necessary HTML and minimal JavaScript are sent to the client. React Server Components can stream rendered content to the client, which receives and displays it without needing to run the data fetching logic or heavy component logic client-side.
3. **Enhanced Performance:** This approach minimizes the amount of JavaScript sent to the browser, reducing load times and improving performance. The client browser doesn't need to fetch the data or render the components; it only displays the streamed HTML content.

Current Adoption and Availability

It's important to note that, as of now, React Server Components are not widely adopted across all frameworks. Their usage is somewhat limited and they are primarily available in a few frameworks, most notably Next.js. This limited adoption can be attributed to several factors:

- **Experimental Status:** React Server Components are still in the experimental phase. They represent a new paradigm in React development, and as with any cutting-edge technology, there's a period of exploration and refinement before widespread adoption.
- **Framework Support:** Currently, only a few frameworks like Next.js offer out-of-the-box support for React Server Components. This is partly due to the need for specific server-side rendering capabilities and infrastructure which are not universally available in all React frameworks.
- **Ecosystem Adjustments:** Adopting React Server Components also requires changes and adaptations in the broader React ecosystem, including tooling, libraries, and development practices.

- **Learning Curve:** There's a learning curve associated with adopting React Server Components as they introduce new concepts and patterns in React development.

When to Use

Despite these considerations, React Server Components can be extremely beneficial in scenarios where offloading compute-intensive operations to the server is advantageous. They are particularly suitable for data-heavy or dynamic applications, where reducing the client-side processing can lead to significant performance gains.

As the technology matures and becomes more integrated into the React ecosystem, we can expect broader adoption and support across more frameworks and tools.

As we conclude our journey through these key data-fetching strategies in React, remember that this is just the tip of the iceberg. For those eager to dive deeper and put these concepts into practice, visit my comprehensive tutorial, [Advanced Data-Fetching Patterns in React](https://itnext.io/5-advanced-data-fetching-techniques-in-react-for-enhanced-user-experience-881195f67692). There, you'll find extensive content that builds upon what we've covered here, with practical examples and in-depth explanations to further your learning and application of these techniques in real-world scenarios.

Summary

This article explores five key data-fetching techniques in React that significantly enhance application performance and user experience. From executing parallel requests to reduce load times, employing lazy loading with Suspense for effective code-splitting, and preloading data for smoother interactions, each method addresses specific performance challenges.

We also discussed Static Site Generation for efficiently handling static content and the emerging yet experimental React Server Components for server-side rendering. While some of these techniques are more established than others, each plays a crucial role in optimizing React applications.

Understanding and applying these strategies can lead to more responsive, faster-loading applications, highlighting the importance of staying updated with the latest React development trends for better web experiences.

If you like the reading, please [Sign up for my newsletter](#). I share Clean Code and Refactoring techniques weekly via [blogs](#), [books](#), [courses](#) and [videos](#).

[Software Development](#)[Programming](#)[Front End Development](#)[Software Engineering](#)[User Experience](#)

Written by Juntao Qiu 

[Follow](#)

2.1K Followers · Writer for ITNEXT

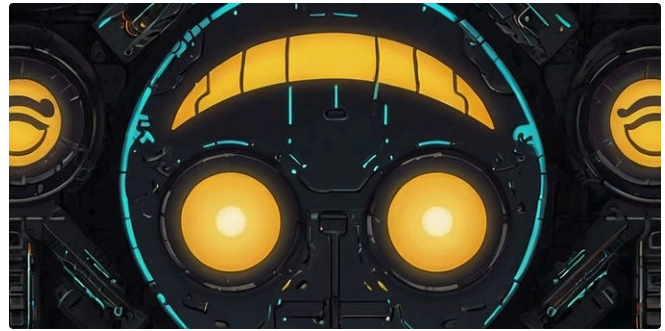

I help developers write better code. Developer | Author | Creator.

<https://juntao.substack.com/> @JuntaoQiu

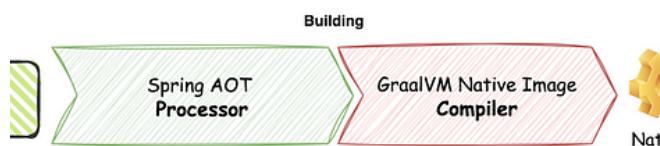
More from Juntao Qiu and ITNEXT


 Juntao Qiu in ITNEXT
My Take on Kent Beck's 'Tidy First?'

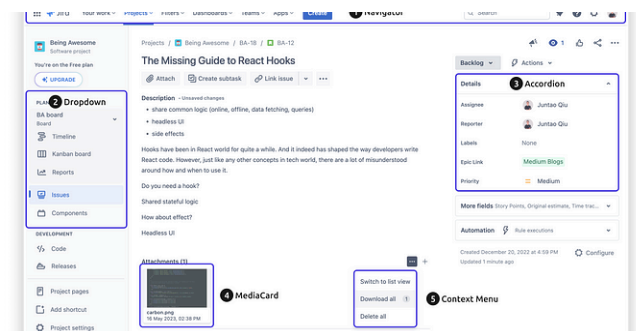
As a fan of Kent Beck, known for his influential books like Test-Driven Development by...

Jun 10  43  1
 Raphael Yoshiga in ITNEXT
Why TDD is a waste of time

Discover the Test-Driven-Development flaws and how to develop instead

 Jun 17  958  77

 Saeed Zarinfam in ITNEXT
10 Spring Boot Performance Best Practices

Making Spring Boot applications performant and resource-efficient

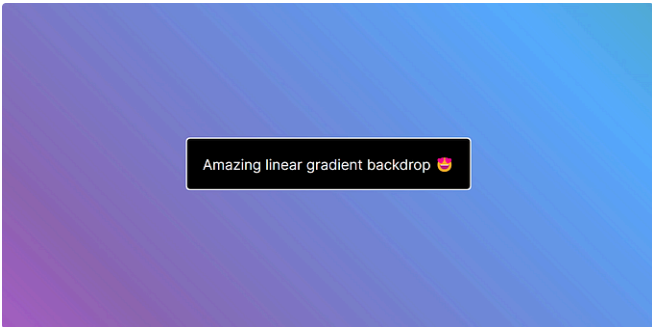

 Juntao Qiu in ITNEXT
Why Web UI Development Is So Hard?

Web UI development might appear straightforward at first glance, but delve...

See all from Juntao Qiu

See all from ITNEXT

Recommended from Medium



 Tari Ibaba in Coding Beauty

New HTML <dialog> tag: An absolute game changer

The new <dialog> tag changes everything for UI design.

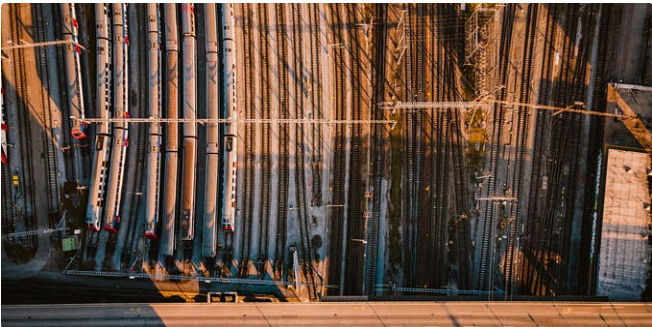
★ Jun 28

👤 694

💬 10

🔖⁺

⋮



 Andrew Zuo

Async Await Is The Worst Thing To Happen To Programming

I recently saw this meme about async and await.

★ Jun 21

👤 1.6K

💬 108

🔖⁺

⋮

Lists

**General Coding Knowledge**

20 stories · 1378 saves

**Stories to Help You Grow as a Software Developer**

19 stories · 1192 saves

**Coding & Development**

11 stories · 697 saves

**Leadership**

53 stories · 382 saves



Afan Khan in JavaScript in Plain English

Avoid These 5 Mistakes as a Web Developer

Learn from others.



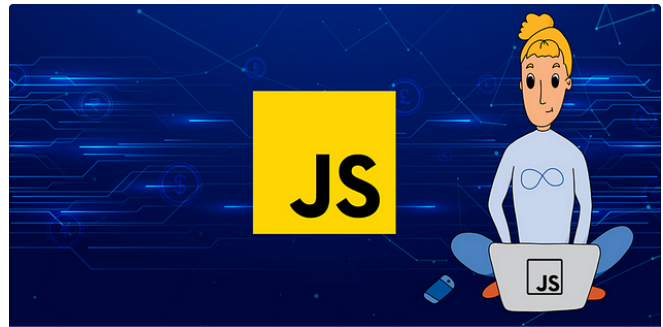
Jan 12



522



6



Codingwinner

33 Concepts Every JavaScript Developer Should Know

I have collected 50+ articles on this subject which I hope will help you learn better.

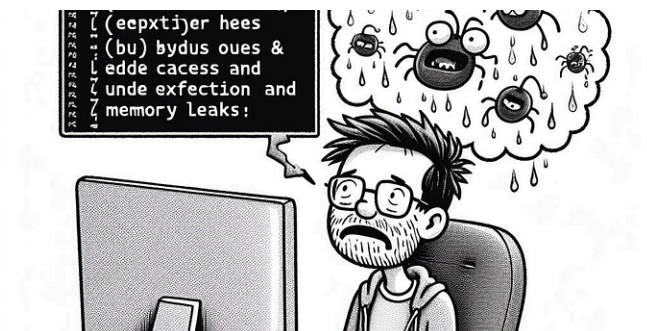
May 29



227



2



Daniel Craciun in Level Up Coding

The Dark Side of useEffect in React

One Mistake away from Disaster





Deepak Chaudhari

Advanced JavaScript Concepts: 2024

Description: Uncover the intricacies of advanced JavaScript concepts, from nested...

 Mar 3  491  8

  Jan 18  1.3K  6

See more recommendations