How to Use Next.js 14's Server Components Effectively

The SaaS Enthusiast · Follow

14 min read · Mar 7, 2024

Imagine a scene where a person is trying to fix a sleek, futuristic device — a metaphor for modern web applications — using old, rusty tools. This contrasts with a toolbox open beside them, filled with shiny, advanced tools they've overlooked. It's a humorous take on using outdated methods (like over-relying on useState for everything) when newer, more efficient solutions (like Server Components) are available.

It's not uncommon to discover a snippet of wisdom that challenges our established practices and nudges us towards introspection. This was precisely my experience when a seemingly innocuous comment on Reddit sparked a profound reevaluation of my approach to using Next.js 14. The comment, simple yet profound, suggested: "with Next.js 14, you don't need to use useState as much." Initially, it appeared as a straightforward observation about state management. Yet, it led me down a rabbit hole of exploration into the depths of Next.js's latest features. I realized I had been oblivious to the full potential of the framework's advancements in Server-Side Rendering (SSR) and beyond. This revelation was not just about using state less frequently; it was a wake-up call to the transformative shift in how data fetching, rendering, and overall application architecture could be optimized in the era of Next.js 14. Through this article, I aim to unravel the layers of this discovery, shedding light on how the new features redefine the development experience and why embracing them could mark a significant leap forward in our web development journey.

> *"with Next.js 14 you don't need to use useState as much"*

That comment made me reflect and research a bit more about Next.js, its new features, and how I was using it all wrong. The comment was right, I don't need to use state as often with Server-Side Rendering (SSR) in a Next.js application because of the way data fetching and rendering are managed in SSR as compared to Client-Side Rendering (CSR) or Static Site Generation (SSG).

Before going to the implementation, I needed to understand the basics:

Client-Side Rendering (CSR)

In CSR, most of the rendering happens in the browser. When a user accesses your application, they first download a minimal HTML page along with the JavaScript required to run your React application. The page is then rendered in the browser, and any required data is fetched from the server or an API after the initial load. This approach heavily relies on managing state on the client side to control the rendering process and to handle the data fetched asynchronously.

Server-Side Rendering (SSR)

SSR, on the other hand, pre-renders each page on the server at request time. When a user requests a page, the server fetches all necessary data, renders the HTML for the page with this data, and then sends the fully rendered page to the client. Since the data fetching and initial rendering are done on the server, there is less immediate need for client-side state management to handle data fetching and rendering. The page arrives at the client ready to be displayed.

Impact on State Management

- **Initial Load:** With SSR, the initial page load can display data without the need to fetch it client-side, reducing the need for initial state setup and client-side data fetching logic.
- **Interactivity:** After the initial load, to make the page interactive or to update the content based on user actions (like submitting a form or filtering data), you still might need to manage state on the client side. However, the amount and complexity of state management can be reduced because the initial state is more complete from the server.
- **Reduced Client-Side Complexity:** By handling data fetching and rendering on the server, you can simplify your client-side logic. You might rely less on complex state management solutions or use simpler state management patterns since the bulk of data handling is performed server-side.

The claim about using state less often with SSR refers to the reduced need for complex client-side state management for initial data fetching and rendering. However, state is still crucial for managing interactivity and updates based on user actions after the initial page load.

Understanding Next.js V13

In Next.js V13, `getStaticProps` and `getServerSideProps` are two data fetching methods that cater to different rendering strategies: Static Site Generation (SSG) and Server-Side Rendering (SSR), respectively. Understanding the difference between these two is key to leveraging Next.js's capabilities for building optimized web applications using V13. Here's a breakdown of each:

`getStaticProps`

- Use Case: `getStaticProps` is used with Static Site Generation (SSG). It fetches data at build time, meaning the data is fetched when you export your application or when you build your site.

- Behavior: Pages are pre-rendered at build time. The HTML for each page is generated with the data fetched by `getStaticProps`. This results in static pages that can be cached by a CDN for performance benefits.
- When to Use: It's ideal for pages that can be pre-rendered with data that doesn't change often, such as blog posts, documentation, product listings, etc.
- Limitations: Since the data is fetched at build time, any changes in data after the build won't be reflected until the next build. It's not suitable for real-time data.

`getServerSideProps`

- Use Case: `getServerSideProps` is used with Server-Side Rendering (SSR). It fetches data on each request to the page.
- Behavior: Pages are rendered on the server at request time, with fresh data fetched for every page load. The server renders the page with the data, and sends the rendered page back to the client.
- When to Use: It's ideal for pages that require real-time data, user-specific data, or for pages with content that changes often, where the latest data is critical.
- Limitations: Since the page is rendered at request time, it might result in slower page loads compared to static pages served from a CDN. However, this allows for dynamic, real-time content.

Key Differences Summarized

- Data Fetching Time: `getStaticProps` fetches data at build time, whereas `getServerSideProps` fetches data at request time.
- Page Regeneration: With `getStaticProps`, pages are pre-generated and can be instantly served from a CDN. `getServerSideProps` generates pages on-the-fly, which may lead to a slight delay but ensures up-to-date content.
- Use Cases: Use `getStaticProps` for static content that doesn't change often, and `getServerSideProps` for dynamic content that changes regularly or depends on the user session.

Both methods are powerful tools in Next.js that allow developers to choose the most appropriate rendering strategy based on the needs of their application, optimizing performance while delivering dynamic, user-centric experiences.

Understanding Next.js V14

The traditional data fetching methods like `getStaticProps` and `getServerSideProps` were indeed reimagined with a new approach known as Server Components. This change represents a shift towards a more unified and flexible way of building web applications with Next.js, aiming to combine the best of both static generation and server-side rendering.

Server Components

Server Components in Next.js allow you to write components that render exclusively on the server. This mechanism doesn't send the component code to the client, leading to smaller bundle sizes and potentially faster load times. Server Components can fetch data directly within the component itself, removing the strict separation between data fetching and UI rendering that existed with `getStaticProps` and `getServerSideProps`.

Fetching Data with Server Components

- **Direct Fetching:** You can directly use data fetching libraries or native fetch within your Server Components. This data is fetched during render time on the server and passed down to client components as props.
- **Streaming:** Server Components also enable streaming capabilities, allowing content to be streamed to the client as it's being generated, which can improve perceived performance for end-users.

Benefits

- **Flexibility:** Server Components provide a more flexible architecture by allowing you to mix server-side and client-side components seamlessly. You can optimize your application by deciding which components need to be server-side based on data requirements and which can be client-side.
- **Performance:** Since Server Components do not add to the JavaScript bundle sent to the browser, they can significantly reduce the amount of code the client needs to download, parse, and execute. This can lead to faster load times and improved performance.
- **Incremental Adoption:** Server Components can be adopted incrementally, meaning you can start using them in parts of your application without needing to refactor everything at once.

Migration from `getStaticProps` and `getServerSideProps`

For pages and components that relied on `getStaticProps` for static generation or `getServerSideProps` for server-side rendering, you can now directly incorporate data fetching logic within Server Components. This approach simplifies the data fetching process, making it more intuitive to integrate with the component lifecycle and potentially streamlining your application's architecture.

It's important to note that while Server Components offer a powerful new model for building Next.js applications, understanding when and how to use them effectively will be key to leveraging their benefits without sacrificing the user experience.

Can you give me an example?

Let's walk through a simple example of migrating from using `getServerSideProps` to the new Server Components model in Next.js.

Original Code with `getServerSideProps`

Imagine we have a page that displays a list of posts, fetched from an API at request time using `getServerSideProps`.

```
import axios from 'axios';

export async function getServerSideProps() {
  const response = await axios.get('https://api.example.com/posts');
  const posts = response.data;

  return {
    props: {
      posts,
    },
  };
}

export default function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

Migrated Code to Server Component

With Server Components, you can directly fetch data within the component. Note that in the Server Components model, files that contain Server Components should have a `.server.js` extension to be distinguished from client-side components. Here's how you might migrate the previous example:

pages/posts.server.js

```
import axios from 'axios';

export default async function Posts() {
  const response = await axios.get('https://api.example.com/posts');
  const posts = response.data;

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

Key Changes

- File Extension: The file is now named `posts.server.js`, indicating it's a Server Component.
- Direct Data Fetching: Data fetching is done directly inside the component using `await`. This is possible because Server Components run on the server, allowing for server-side logic like data fetching to be incorporated directly.
- No `getServerSideProps`: The use of `getServerSideProps` is eliminated. Instead, the component itself handles the responsibility of fetching data and rendering based on that data.

To Be Noted

- **Client Components:** If you need interactivity within parts of your page (like event handlers), you would separate those into Client Components (files with `.js` or `.jsx` extension) and import them into your Server Components. Server Components are read-only and do not include interactivity.

- **Environment Variables:** Be mindful of using environment variables directly in Server Components. Since they run on the server, ensure you're not exposing sensitive information to the client.

- **API Routes:** You still can and often will use API routes for client-side data fetching, form submissions, etc. Server Components do not replace the need for API routes but provide a more efficient way to render pages and components server-side with direct data dependencies.

This example demonstrates a basic migration strategy. Depending on the complexity of your application, you might need to consider additional factors such as error handling, loading states, and integrating client-side components for interactive elements.

What about the interactions?

Expanding the example to include interactivity (updating and deleting posts) requires a mix of Server Components for fetching data and Client Components for handling interactive elements like buttons. Let's first outline how you might implement this in Next.js version 13 with `getServerSideProps`, and then migrate it to use Server Components in version 14.

pages/posts.js (V13)

```
// pages/posts.js

import axios from 'axios';

export async function getServerSideProps() {
  const res = await axios.get('https://api.example.com/posts');
  return {
    props: {
      posts: res.data,
    },
  };
}

export default function Posts({ posts }) {
  async function updatePost(id) {
    // Implementation for updating a post
    // E.g., sending a PATCH request to an API
  }

  async function deletePost(id) {
```

```
    // Implementation for deleting a post
    // E.g., sending a DELETE request to an API
  }

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          {post.title}
          <button onClick={() => updatePost(post.id)}>Update</button>
          <button onClick={() => deletePost(post.id)}>Delete</button>
        </li>
      ))}
    </ul>
  );
}
```

Migrating to Server Components with Interactions

In version 14 with Server Components, you'll separate the concerns: use a Server Component for fetching and displaying the list of posts, and Client Components for interactive elements like the update and delete buttons.

Server Component for Fetching and Displaying Posts

```
components/PostsList.server.js

import axios from 'axios';
import PostActions from './PostActions.client';

export default async function PostsList() {
  const res = await axios.get('https://api.example.com/posts');
  const posts = res.data;

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          {post.title}
          <PostActions postId={post.id} />
        </li>
      ))}
    </ul>
  );
}
```

Client Component for Update and Delete Actions

```
// components/PostActions.client.js

export default function PostActions({ postId }) {
  async function updatePost() {
    // Implementation for updating a post
    // E.g., sending a PATCH request to an API
  }

  async function deletePost() {
    // Implementation for deleting a post
    // E.g., sending a DELETE request to an API
  }

  return (
    <>
      <button onClick={updatePost}>Update</button>
      <button onClick={deletePost}>Delete</button>
    </>
  );
}
```

Key Changes and Considerations

- **Server and Client Component Split:** The `PostsList.server.js` Server Component is responsible for fetching and displaying the list of posts. The `PostActions.client.js` Client Component handles the interactive elements (update and delete buttons) and must be a regular `.client.js` file to ensure it's run on the client side.
- **Direct Data Fetching in Server Components:** Similar to the previous example, data fetching is done directly within the Server Component.
- **Interactivity Managed by Client Components:** All interactivity, including updating and deleting posts, is handled by Client Components. This is because Server Components are not interactive and are intended for parts of your application that can be pre-rendered on the server.

This setup ensures that your application can leverage the efficiency and performance benefits of Server Components while still providing a dynamic and interactive user experience where necessary.

Updating State From Client Component

In a Next.js application using Server Components along with Client Components, when an item is deleted (or updated), the client-side code must inform the server-

side code to refresh the data and update the UI accordingly. Since Server Components (which handle data fetching and initial rendering) run on the server and do not re-run in response to client-side interactions directly, you'll need to employ a strategy to refresh the data shown to the user after an action like deletion.

After deleting a post, you can trigger a client-side data fetch to reload the list of posts. This approach involves using client-side state management, possibly with React hooks like `useState` and `useEffect`, in a Client Component.

Example:

Assuming you have a `PostsList.client.js` component that initially gets its data from a Server Component but also has the capability to fetch data client-side:

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import PostActions from './PostActions.client';

// use client
function PostsList() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    async function fetchPosts() {
      const response = await axios.get('https://api.example.com/posts');
      setPosts(response.data);
    }

    fetchPosts();
  }, []);

  async function refreshPosts() {
    const response = await axios.get('https://api.example.com/posts');
    setPosts(response.data);
  }

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          {post.title}
          <PostActions postId={post.id} refreshPosts={refreshPosts} />
        </li>
      ))}
    </ul>
  );
}
```

```
export default PostsList;
```

In `PostActions.client.js`, you would call `refreshPosts` after successfully deleting a post:

```
async function deletePost(postId) {
  // Delete post request
  await axios.delete(`https://api.example.com/posts/${postId}`);
  // Refresh the posts list
  props.refreshPosts();
}
```

Conclusion

Transitioning from Next.js version 13 to version 14 introduces significant architectural changes and enhancements that impact how data fetching, rendering, and component structuring are approached. These changes are geared towards improving performance, user experience, and developer efficiency. Here's a comprehensive summary of our discussion covering the evolution from `getServerSideProps` to Server Components and Client Components, and the benefits of adopting the new model in Next.js version 14.

Next.js Version 13 and Earlier

- `getServerSideProps` and `getStaticProps`: These data fetching methods were integral to how developers implemented Server-Side Rendering (SSR) and Static Site Generation (SSG) in Next.js. `getServerSideProps` fetches data at request time, making it suitable for dynamic content that changes often, while `getStaticProps` fetches data at build time for static content.
- Rendering Strategy: Components rendered on the server or statically generated at build time are sent to the client as HTML. The client then re-hydrates these components for interactivity, with client-side JavaScript taking over subsequent navigation and dynamic updates.
- Challenges: While effective, this model often led to larger JavaScript bundles being sent to the client, potentially impacting performance. It also maintained a

clear separation between data fetching and component logic, sometimes complicating the development process.

Next.js Version 14: Introduction of Server Components and Client Components

- Server Components: These components run exclusively on the server, with the capability to fetch data directly within the component. They do not send their code to the client, significantly reducing the size of the JavaScript bundle that needs to be downloaded and parsed by the browser. Server Components use a `.server.js` extension.
- Client Components: Designed for handling client-side interactivity, Client Components run in the browser and use a `.client.js` extension. They are crucial for dynamic functionalities like handling user inputs, clicks, and other interactions.
- Streaming SSR: Next.js 14 enhances SSR with streaming capabilities, allowing parts of a page to be sent to the client as soon as they're ready. This approach improves the time-to-first-byte (TTFB) and the overall user experience by progressively rendering the page.
- Migration Benefits: Migrating from the traditional `getServerSideProps` approach to using Server Components and Client Components offers several advantages:
- Reduced JavaScript Bundle Size: By eliminating the need to send server component code to the client, the initial load time is decreased, and performance is enhanced.
- Improved Flexibility: Developers gain more control and flexibility by mixing server and client components, enabling optimized data fetching strategies and rendering paths.
- Enhanced User Experience: Streaming SSR and selective hydration ensure that users see content faster, even on complex pages with significant dynamic content.
- Simplified Development Process: Directly fetching data in Server Components can simplify the data fetching process, making the codebase more intuitive and easier to manage.

Adopting Next.js 14's Server Components and Client Components represents a paradigm shift in building web applications with Next.js. This approach not only

offers a more efficient and performance-optimized architecture but also aligns with modern web development practices that emphasize user experience and developer productivity. While the transition involves rethinking how components are structured and data is fetched, the benefits of reduced bundle sizes, faster load times, and improved scalability make it a compelling upgrade for Next.js applications.

For teams and projects currently using `getServerSideProps` extensively, evaluating the potential impacts and planning a phased migration to Server Components and Client Components can help leverage these advancements, ensuring that applications remain fast, modern, and maintainable.

Empower Your Tech Journey:

Explore a wealth of knowledge designed to elevate your tech projects and understanding. From safeguarding your applications to mastering serverless architecture, discover articles that resonate with your ambition.

New Projects or Consultancy

For new project collaborations or bespoke consultancy services, reach out directly and let's transform your ideas into reality. Ready to take your project to the next level?

- Email me at one@upskyrocket.com
- Visit My Partner In Tech for custom solutions

Front End

- How to Use Next.js 14's Server Components Effectively
- Contact Form: Elevating Web UX with React-Hook-Form, Yup, and MUI
- Setting Up An Agency's Website in Just 24 Hour swith Next.js and AWS Amplify
- Embracing Zustand: A Game-Changer for React State Management
- Improving Side Navigation UX in Next.js with MUI 5: A Practical Approach
- Twice the Trouble: Unraveling Next.js's Repeated API Requests with ChatGPT's Help
- Creating a Lasting Impression with Modern Backgrounds

- Blending Motion Magic: Integrating Tailwind CSS Animations into MUI V5 for an Engaging Login Experience

Protecting Routes

- How to Create Protected Routes Using React, Next.js, and AWS Amplify
- How to Protect Routes for Admins in React Next.js Using HOC
- Secure Your Next.js App: Advanced User Management with AWS Cognito Groups

Mastering Serverless Series

- Mastering Serverless (Part I): Enhancing DynamoDB Interactions with Document Client
- Mastering Serverless (Part II): Mastering AWS DynamoDB Batch Write Failures for a Smoother Experience.
- Mastering Serverless (Part III): Enhancing AWS Lambda and DynamoDB Interactions with Dependency Injection
- Mastering Serverless IV: Unit Testing DynamoDB Dependency Injection With Jest
- Mastering Serverless (Part V): Advanced Logging Techniques for AWS Lambda

Advanced Serverless Techniques

- Advanced Serverless Techinques I: Do Not Repeat Yourself
- Advanced Serverless Techniques II: Streamlining Data Access with DAL
- Advanced Serverless Techniques III: Simplifying Lambda Functions with Custom DynamoDB Middleware
- Advanced Serverless Techniques IV: AWS Athena for Serverless Data Analysis
- Advanced Serverless Techniques V: DynamoDB Streams vs. SQS/SNS to Lambda

Front End Development | React | Reactjs | Software Development | Nextjs

Follow

Written by The SaaS Enthusiast

351 Followers

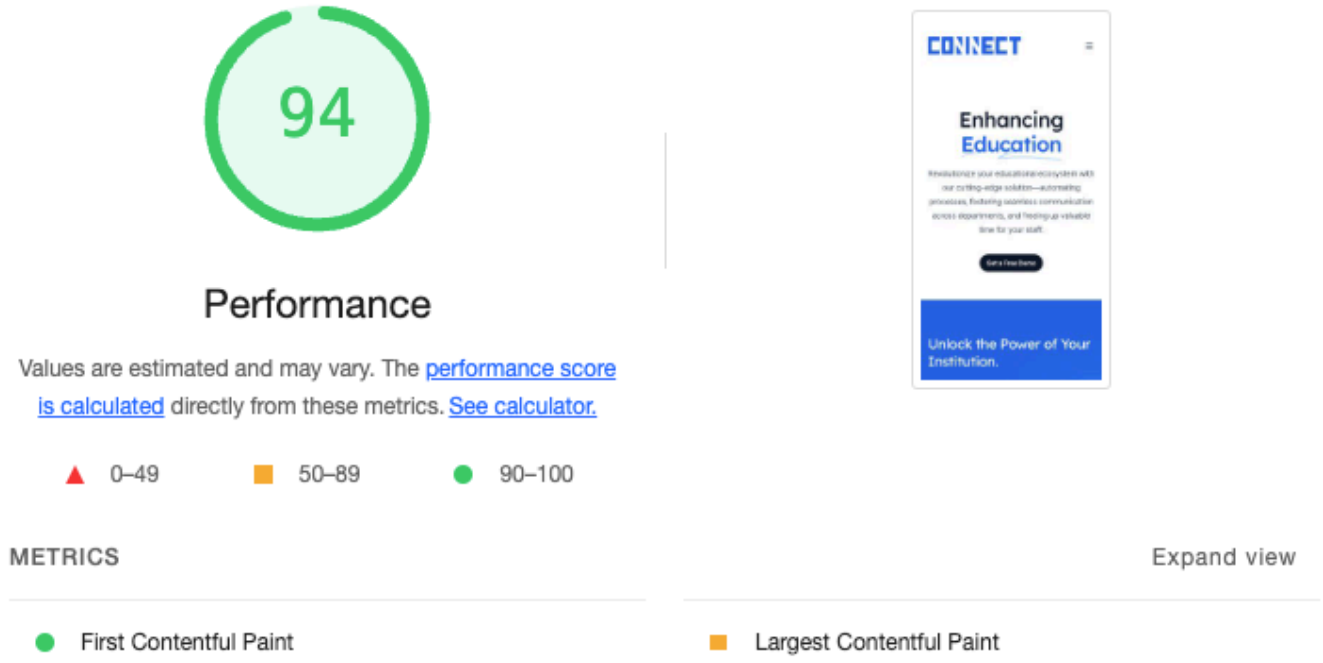
More from The SaaS Enthusiast

The SaaS Enthusiast

I Thought I Knew useEffect, But I Was Wrong: useEffect Misconceptions

The Problem: Automatic Saving of Answers

May 22 79 2

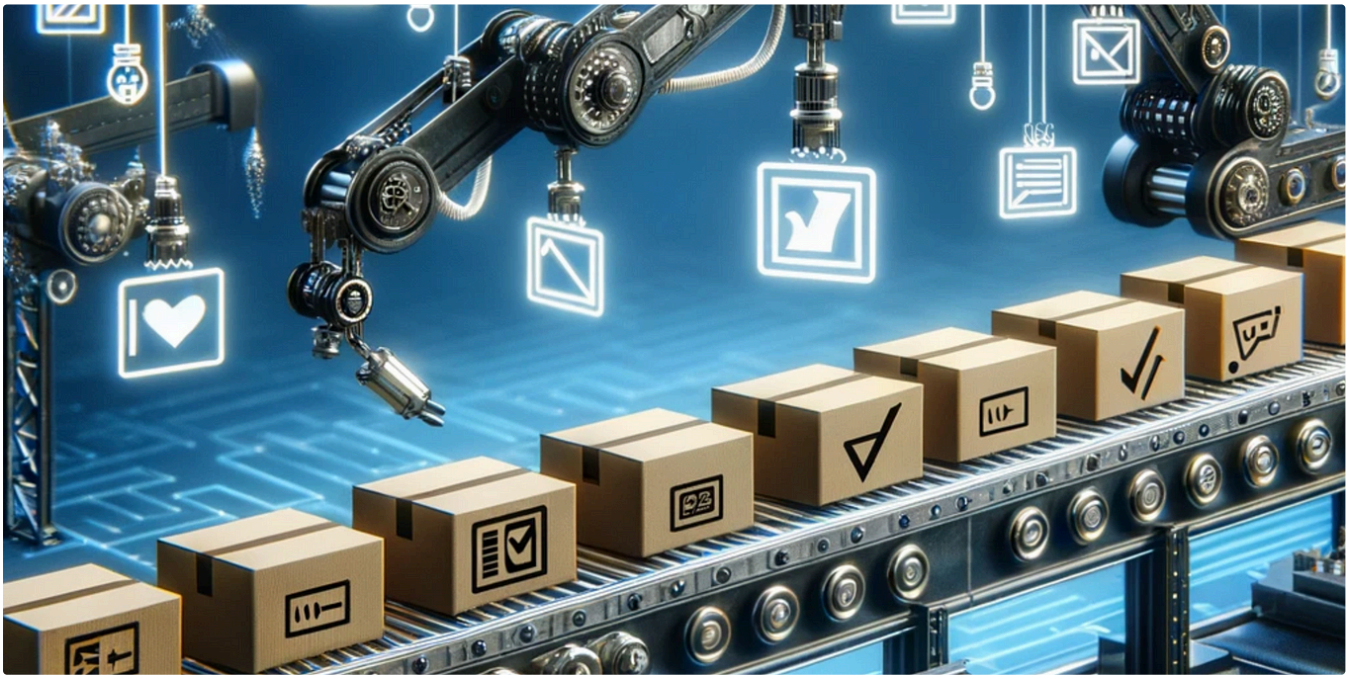


The SaaS Enthusiast

Next.js SEO: Best Practices for Higher Rankings

In addition to leveraging Next.js features for SEO, there are several other best practices you should consider to optimize your site's...

Apr 18 7



The SaaS Enthusiast

Dynamic Forms in React: A Guide to Implementing Reusable Components and Factory Patterns

The Challenge of Managing Forms in Modern Web Applications

Mar 13 5 1

The SaaS Enthusiast

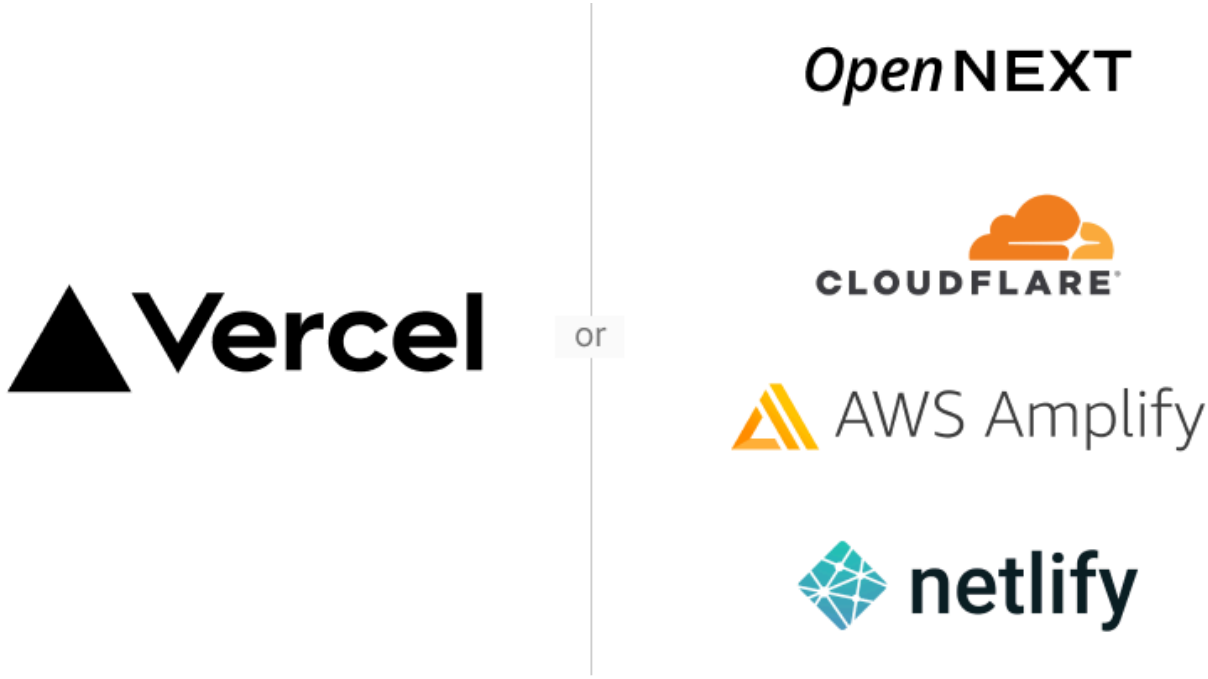
Serverless Simplified: Integrating Docker Containers into AWS Lambda via serverless.yml

Initiating with the development and local testing of Docker containers tailored for AWS Lambda, this workflow delineates a streamlined...

Feb 20 4

See all from The SaaS Enthusiast

Recommended from Medium

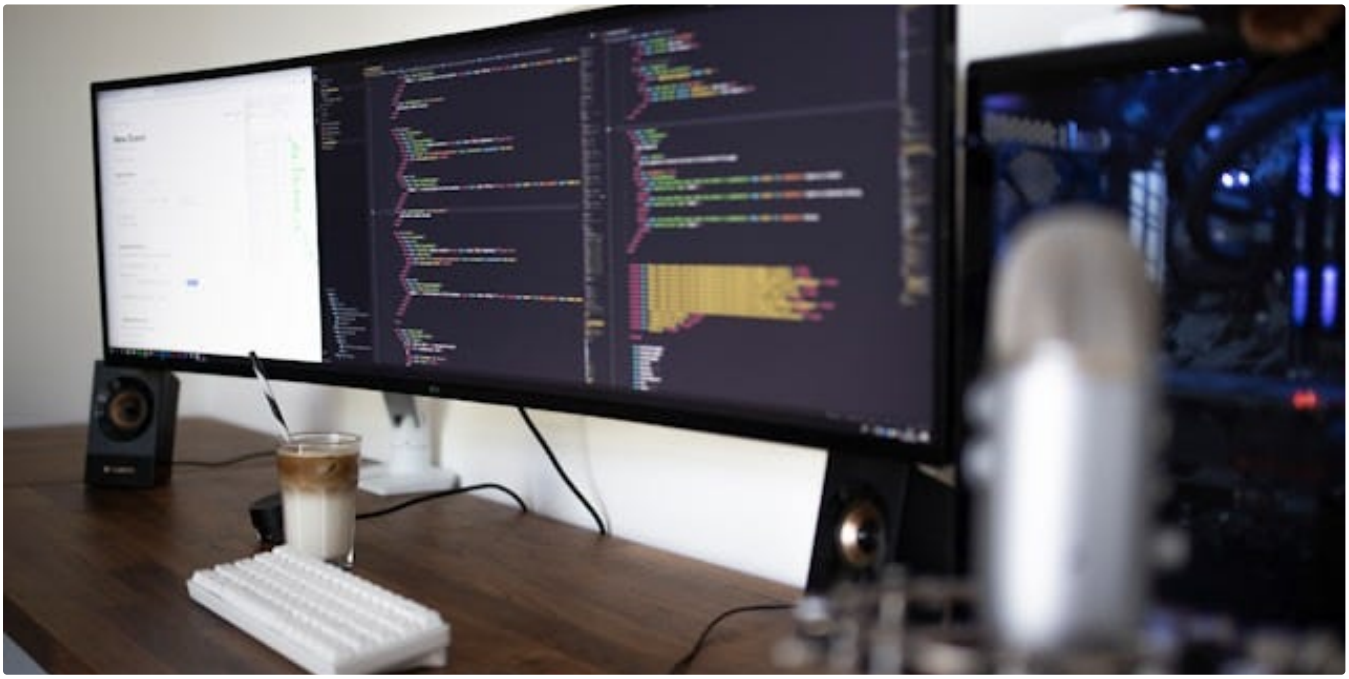


Mitchell Kossoris

Vercel Just Changed its Pricing —How Does it Compare?

The popular NextJS frontend infrastructure service has historically been criticized for its high cost.

Apr 5 104 2



Mark Harris

Maximizing Next.js Potential: Advanced Caching Strategies for Peak Performance

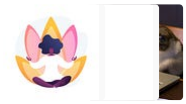
In the realm of web development, site performance isn't merely an aspect to be optimized; it's a fundamental pillar that supports a smooth...

Jul 3 8

Lists

General Coding Knowledge

20 stories · 1430 saves

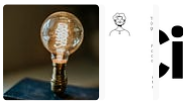


Stories to Help You Grow as a Software Developer

19 stories · 1232 saves

Coding & Development

11 stories · 714 saves



Good Product Thinking

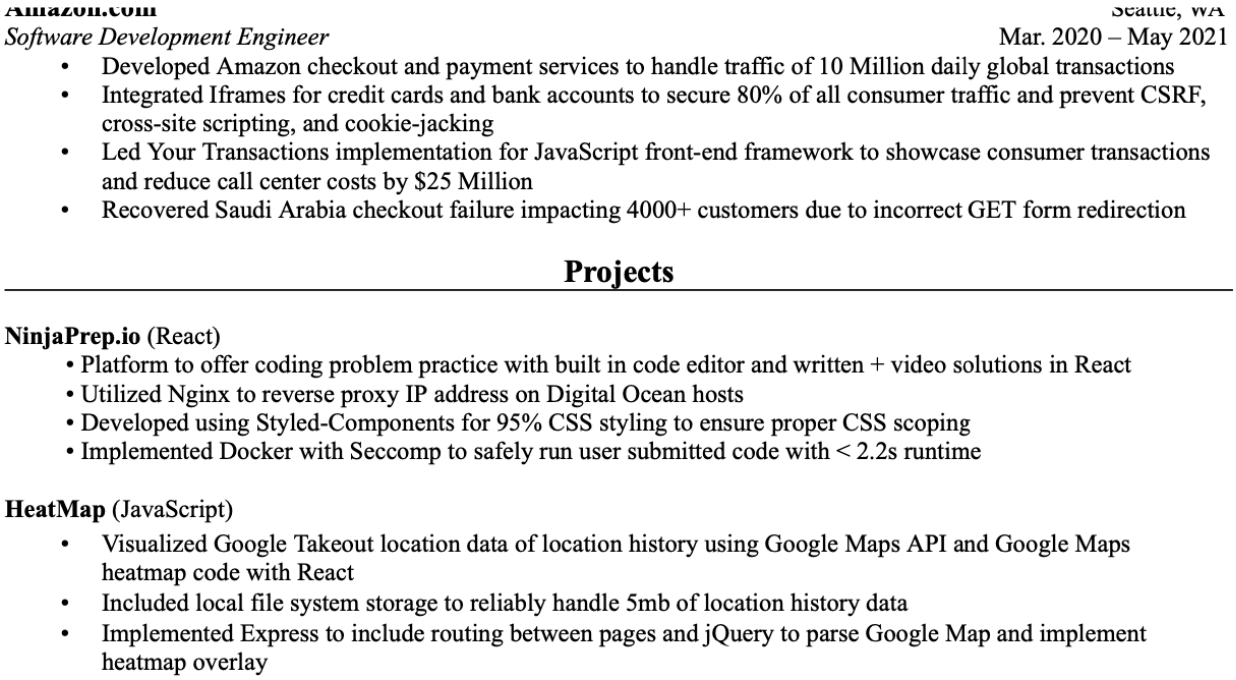
11 stories · 649 saves

Sviat Kuzhelev

Mastering NextJS Architecture with TypeScript in Mind | Design Abstractions 2024

You'll never know how everything works, but you should understand the system. — by Sviat Kuzhelev.

Jun 17 6.8K



Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

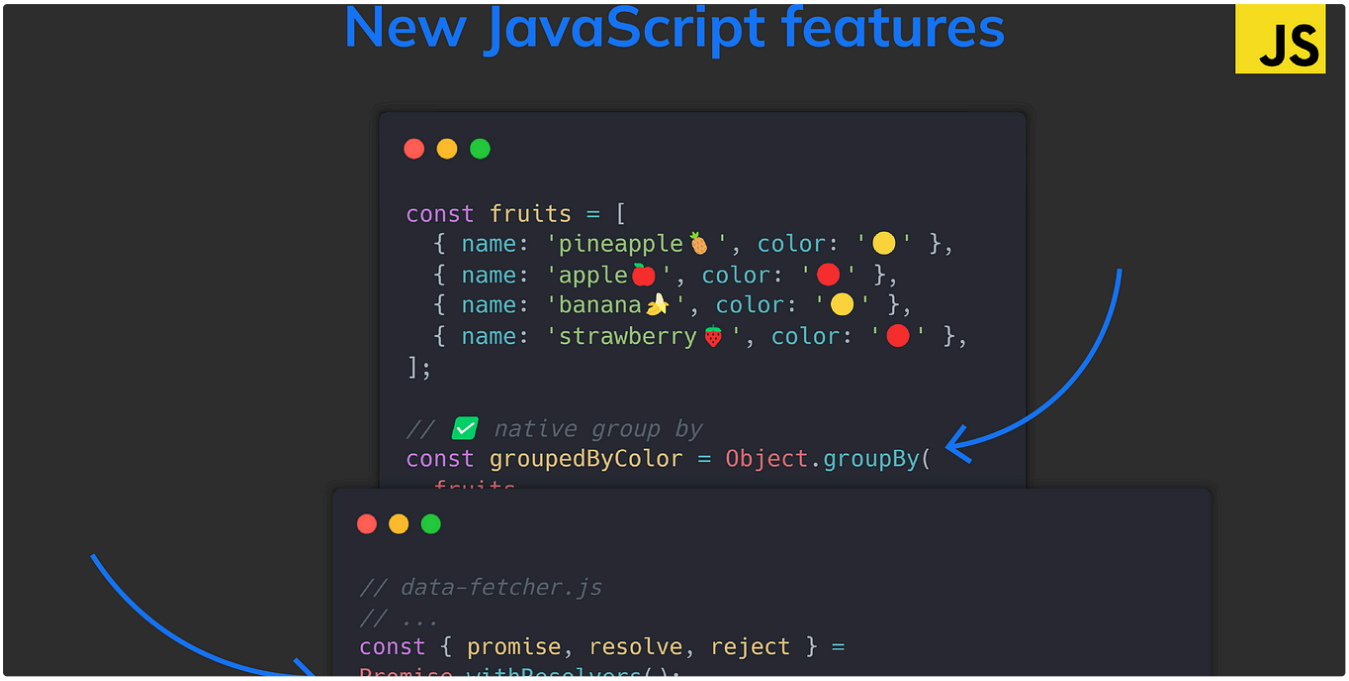
HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 15.1K 231

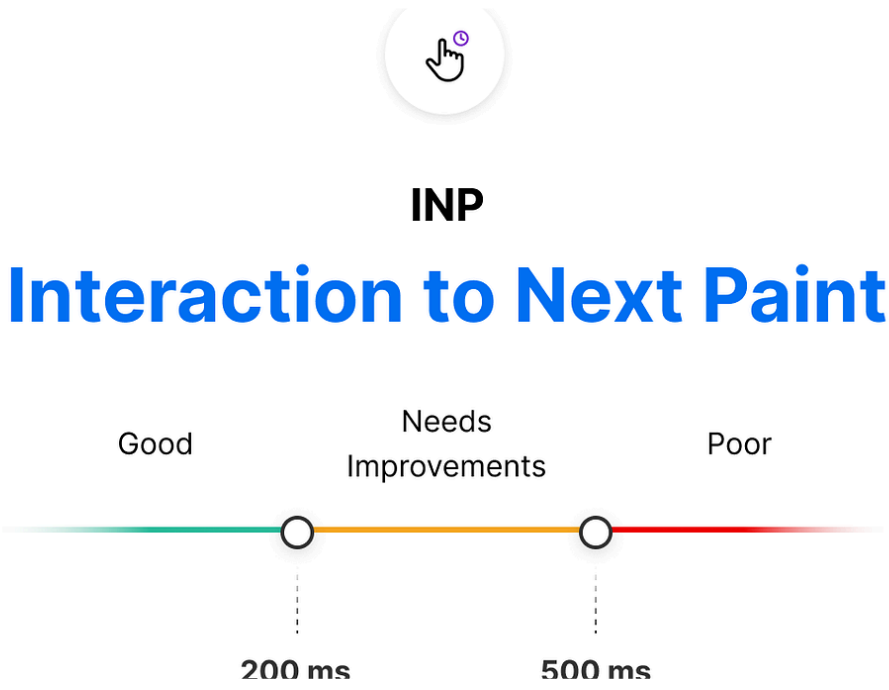


Tari Ibaba in Coding Beauty

5 amazing new JavaScript features in ES15 (2024)

5 juicy ES15 features with new functionality for cleaner and shorter JavaScript code in 2024.

Jun 2 1.93K 10



imran farooq in JavaScript in Plain English

How To Improve Interaction With Next Paint In Next.js

How to prepare for INP in React and Next.js?

May 13 8

See more recommendations