NEXT.JS

Building scalable server-sided React apps.

BY ZAVAAR SHAH

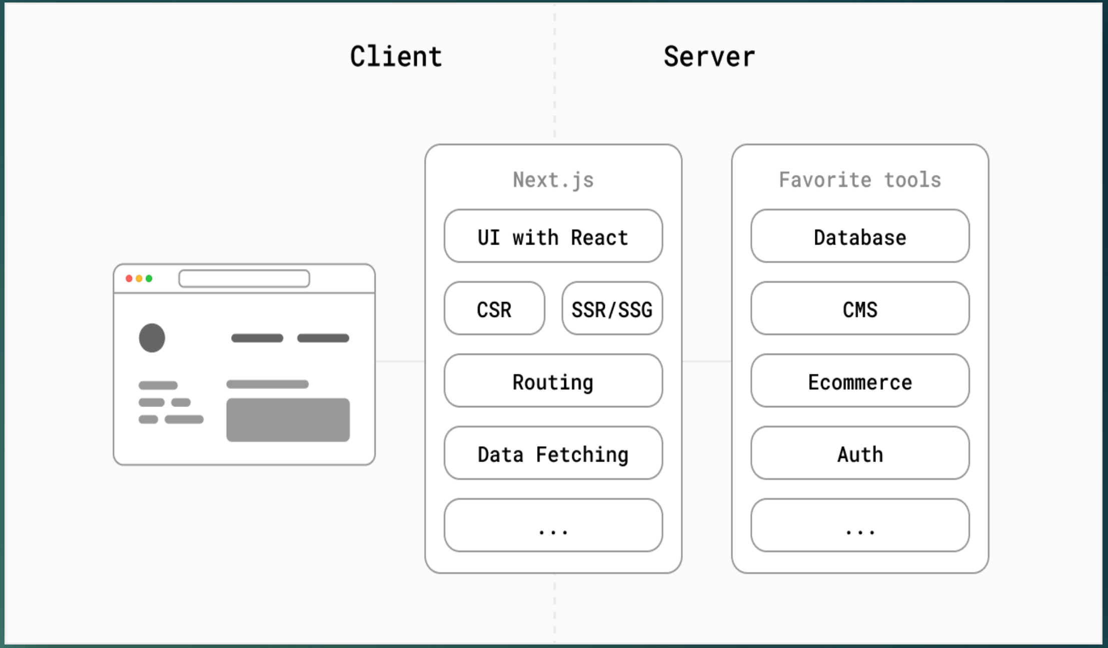
Compiled by Zavaar Shah https://github.com/thatziv

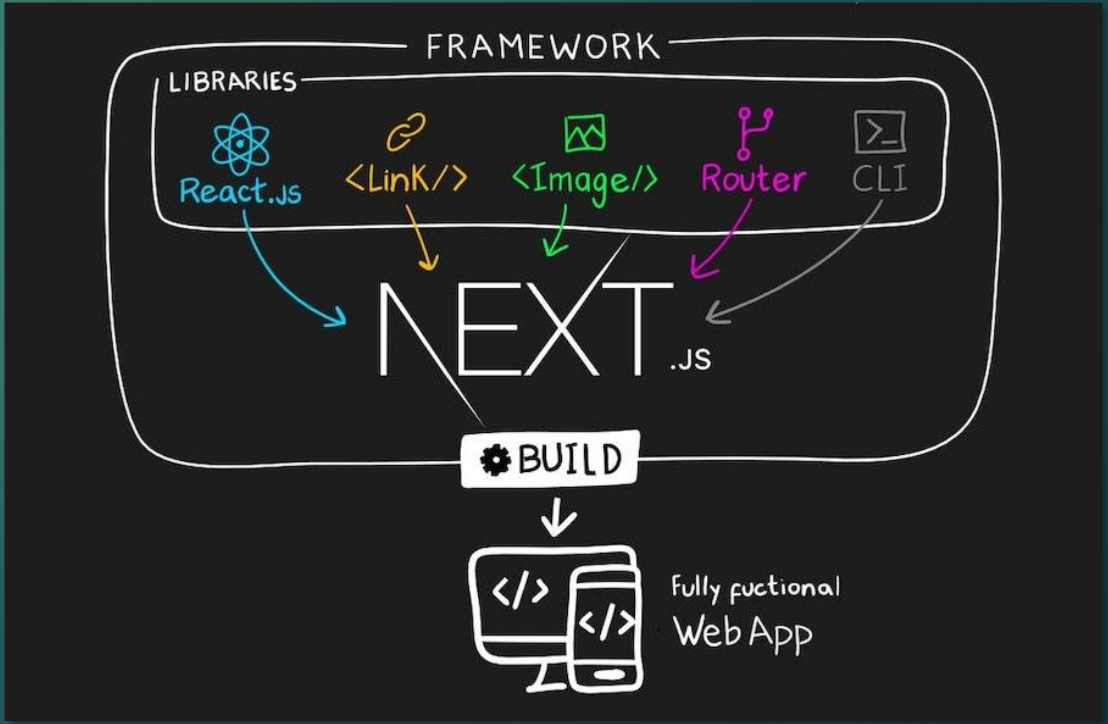
What is **Next.js**?

*Next.js is a React **framework** that gives you building blocks to create web applications.*

You can use React to build your UI, then incrementally adopt Next.js features to solve common application requirements such as routing, data fetching, integrations - all while improving the developer and end-user experience.

Whether you're an individual developer or part of a larger team, you can leverage React and Next.js to build fully interactive, highly dynamic, and performant web applications.





Why **Next.js**?

PROS

1. **Improved performance**: Next.js optimizes page loading and enables server-side rendering, which can lead to faster page loads and improved SEO.
2. **Automatic code splitting**: Next.js automatically splits your code into small, optimized chunks, which can improve performance and reduce load times.
3. **Improved developer experience**: Next.js provides features like automatic reloads, error reporting, and hot module replacement, making it easier for developers to build and debug applications.
4. **Static site generation**: Next.js allows you to generate static sites, which can be deployed easily and have excellent performance.
5. **Built-in routing**: Next.js provides a built-in routing system that simplifies client-side navigation and enables easy creation of nested routes.

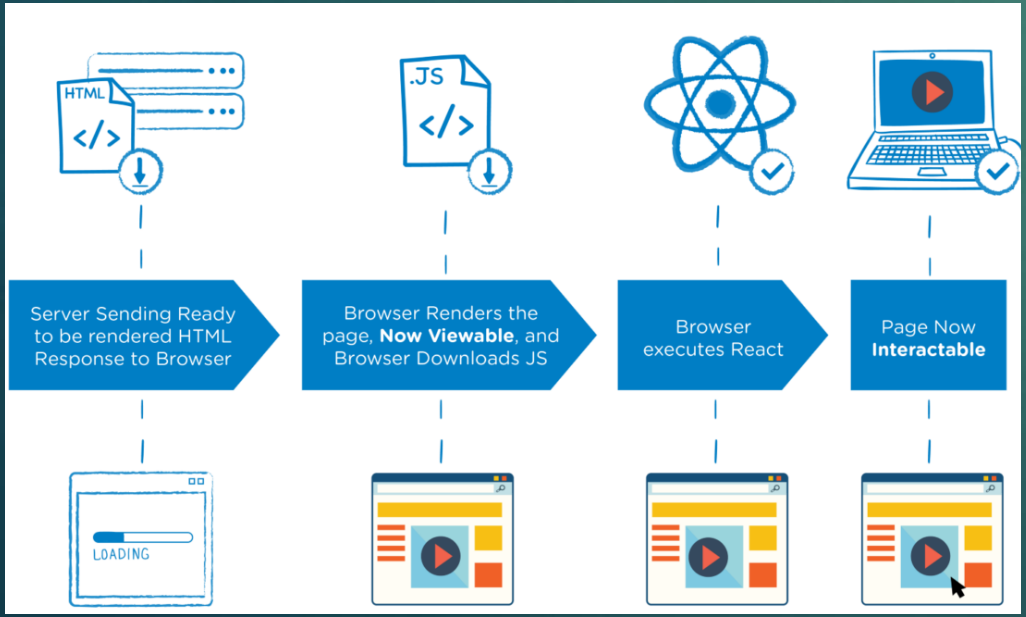
CONS

1. **Learning curve:** Next.js has a bit of a learning curve, particularly if you are not familiar with React, Node.js, or server-side rendering patterns.
2. **Limited customization**: While Next.js provides a lot of helpful tools and features out of the box, it may not be as customizable as some other frameworks *(like **Remix.js** or **Solid.js**).* This can limit your ability to tailor your application to your specific needs.
3. **Server-side rendering**
   - SSR sometimes can be slower as it can add overhead on an already busy server.
   - Requires a server to host (unlike CSR which can be hosted through GitHub Pages and others).
   - Integrating other libraries and modules that only operate in the browser, or if your component requires accessibility to client-exclusive properties like `window` or localStorage`.
4. **Just another JavaScript framework to learn...**

SSR vs. CSR

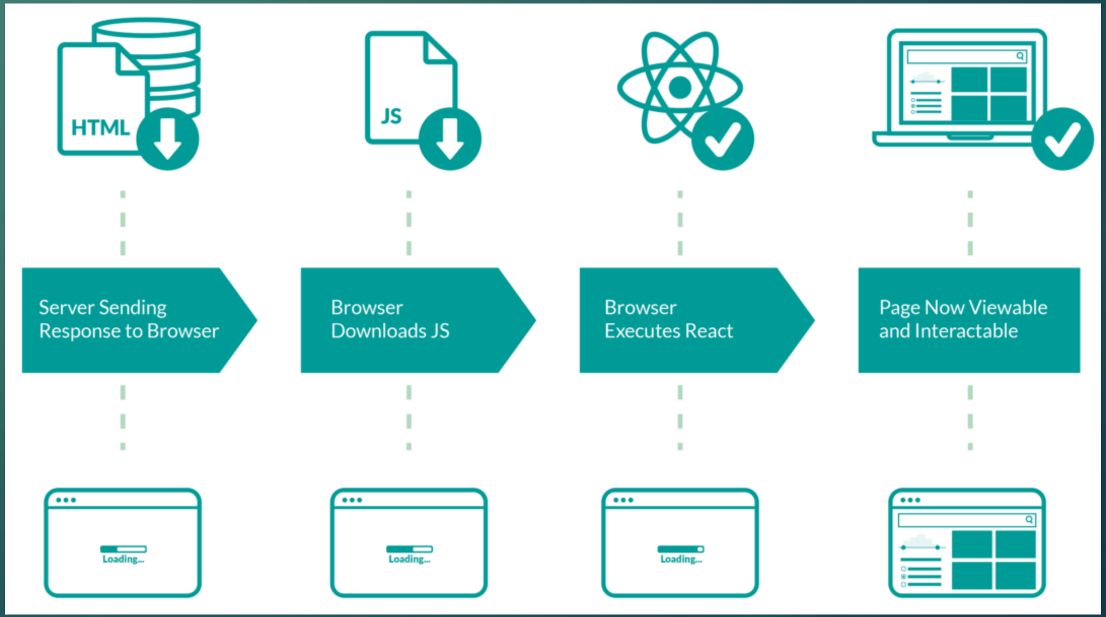
Server-side rendering

- Server-side rendering (SSR) is a technique used to render web pages on the server and send the fully-rendered HTML to the client. In this approach, the server processes the initial request, fetches any data required for the page, and generates the HTML response, which is then sent to the client's browser. The client's browser does not have to wait for any further processing or data fetching, which can improve the initial load time and provide a better user experience.

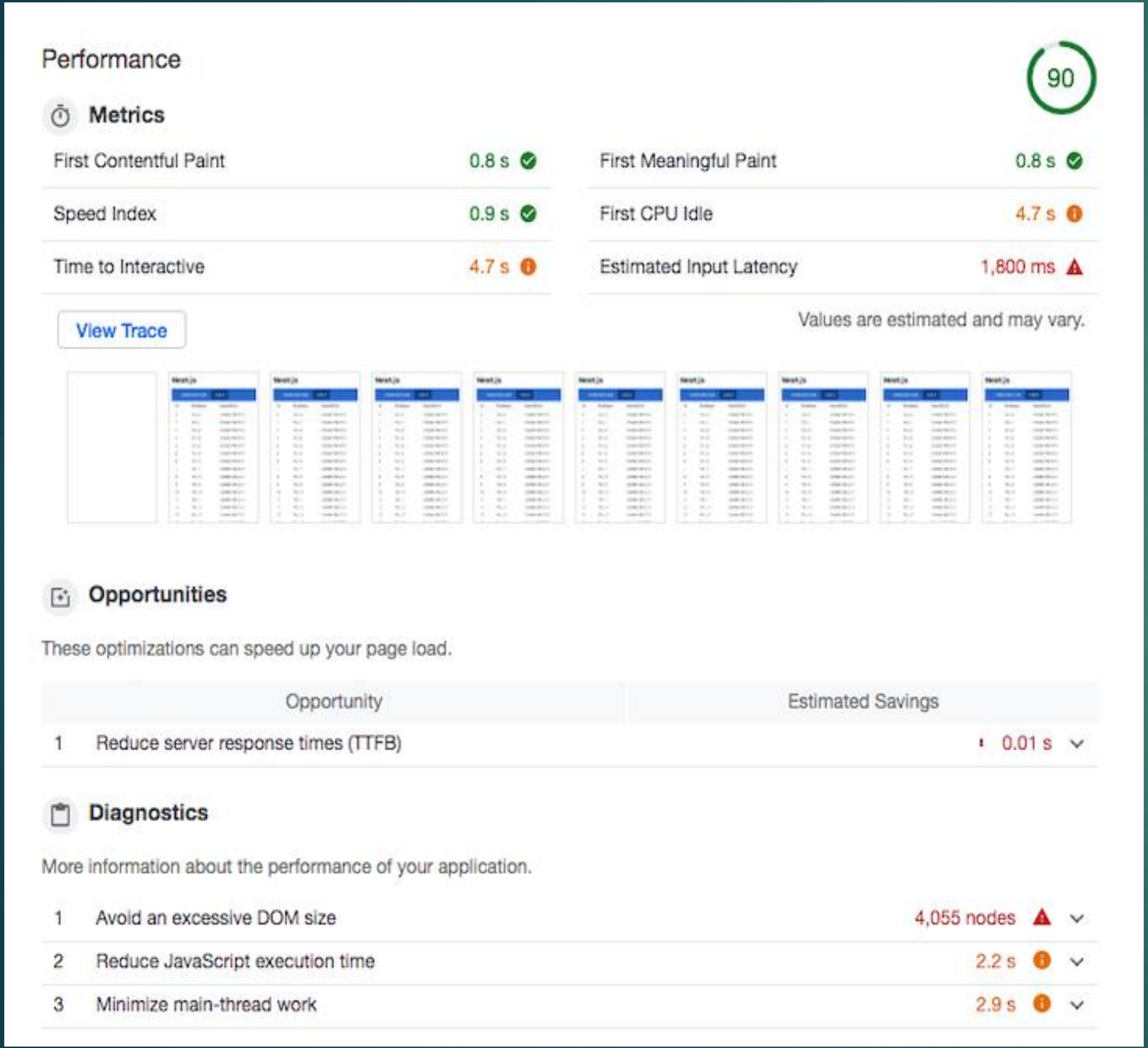

HTML
.JS
Server Sending Ready to be rendered HTML Response to Browser
Browser Renders the page, **Now Viewable**, and Browser Downloads JS
Browser executes React
Page Now **Interactable**
LOADING

Client-side rendering

- Client-side rendering (CSR) is the process of generating the HTML and displaying content in the user's browser using JavaScript after the initial page load. In client-side rendering, the server sends a minimal amount of HTML and JavaScript to the client, and then the client uses that code to fetch and display as necessary. This approach allows for more interactive and dynamic user interfaces but can also result in slower initial page load times.



Next.js (SSR) vs. React.js (CSR)



Performance

90

Metrics

First Contentful Paint	0.8 s ✓	First Meaningful Paint	0.8 s ✓
Speed Index	0.9 s ✓	First CPU Idle	4.7 s ●
Time to Interactive	4.7 s ●	Estimated Input Latency	1,800 ms ▲

View Trace

Values are estimated and may vary.

Opportunities

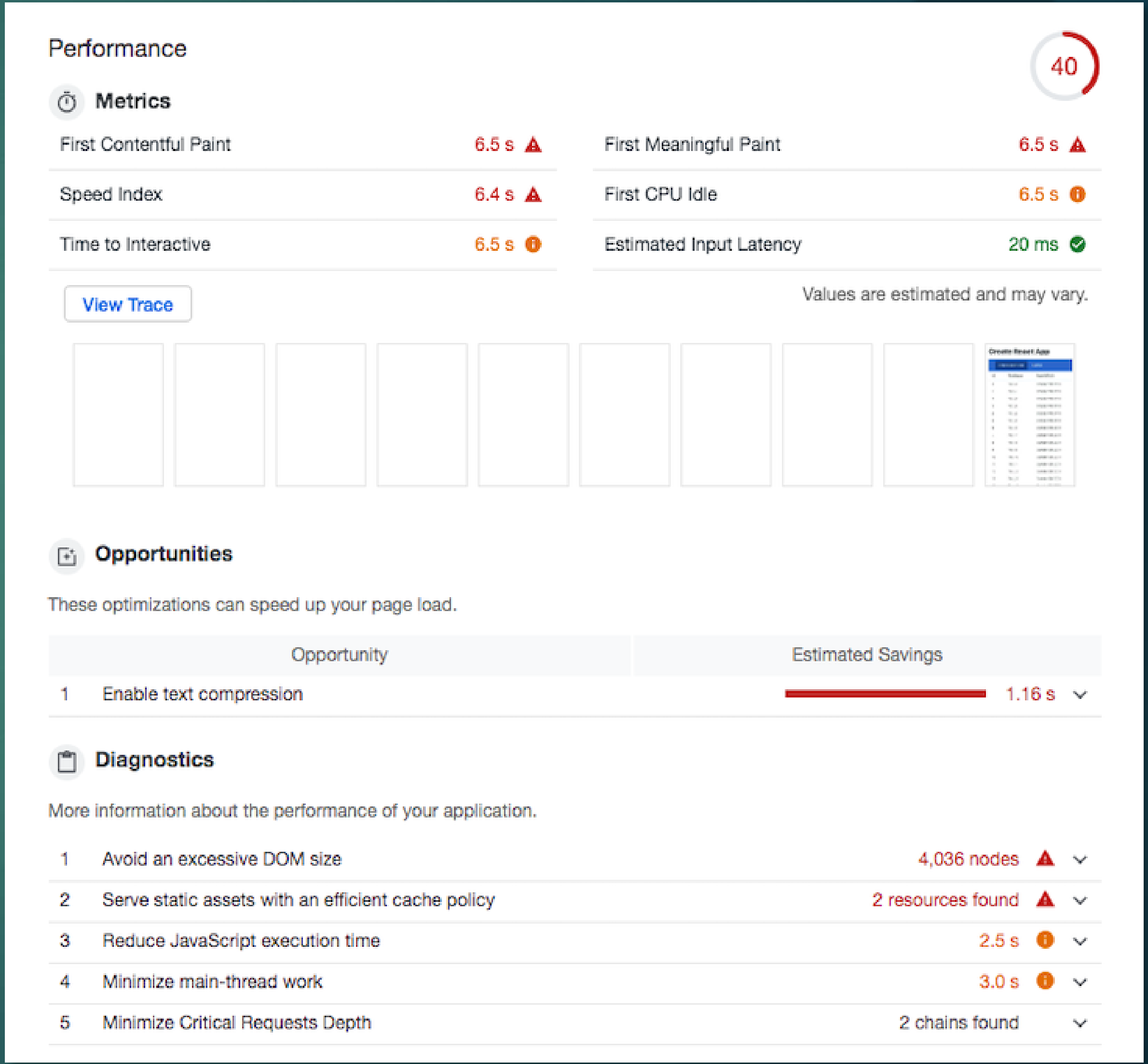
These optimizations can speed up your page load.

	Opportunity	Estimated Savings
1	Reduce server response times (TTFB)	0.01 s

Diagnostics

More information about the performance of your application.

1	Avoid an excessive DOM size	4,055 nodes ▲
2	Reduce JavaScript execution time	2.2 s ●
3	Minimize main-thread work	2.9 s ●



Performance

40

Metrics

First Contentful Paint	6.5 s ▲	First Meaningful Paint	6.5 s ▲
Speed Index	6.4 s ▲	First CPU Idle	6.5 s ●
Time to Interactive	6.5 s ●	Estimated Input Latency	20 ms ✓

View Trace

Values are estimated and may vary.

Opportunities

These optimizations can speed up your page load.

	Opportunity	Estimated Savings
1	Enable text compression	1.16 s

Diagnostics

More information about the performance of your application.

1	Avoid an excessive DOM size	4,036 nodes ▲
2	Serve static assets with an efficient cache policy	2 resources found ▲
3	Reduce JavaScript execution time	2.5 s ●
4	Minimize main-thread work	3.0 s ●
5	Minimize Critical Requests Depth	2 chains found

Compiled by Zavaar Shah https://github.com/thatziv

https://blog.logrocket.com/create-react-app-vs-next-js-performance-differences/

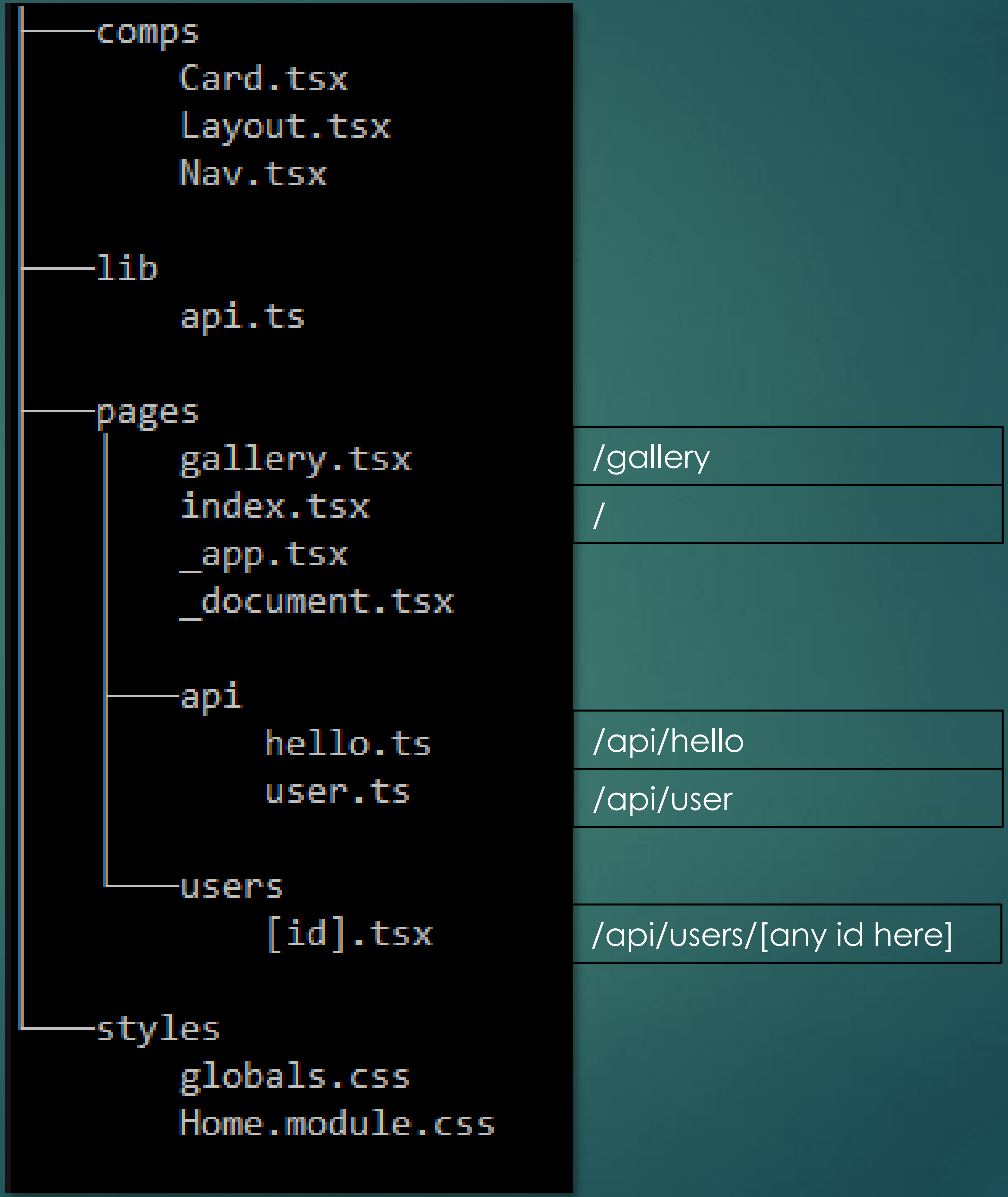
Create Next App

- `npx create-next-app@latest`
- `# automatic setup`
- `cd my-app/`
- `# go to ‘my-app’ directory`
- `npm run dev`
- `# start dev server @ http://localhost:3000`

Routing - How is **Next** organized?

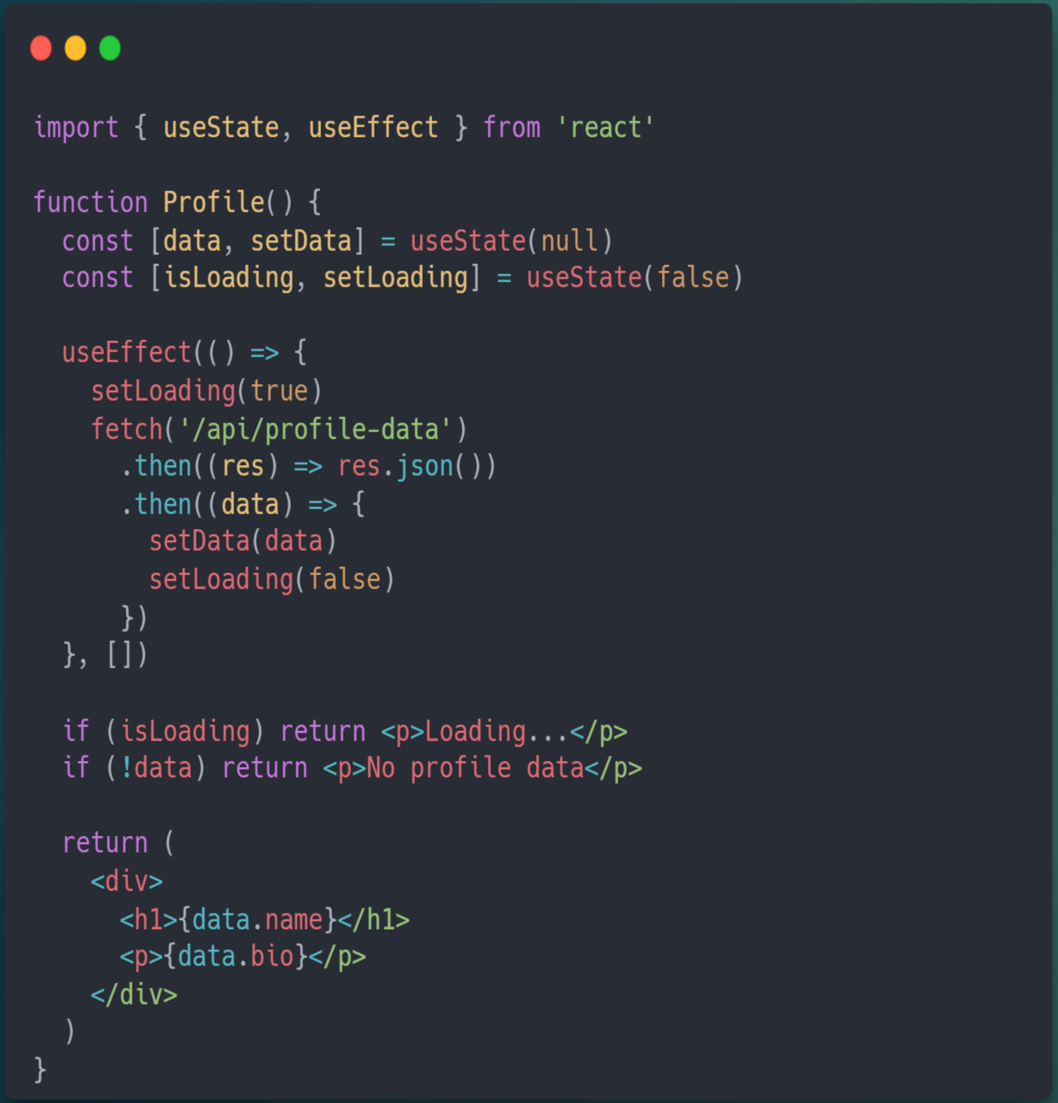
In Next.js, a page is a React Component exported from a **.js, .jsx, .ts, or .tsx** file in the **pages** directory. Each page is associated with a route based on its file name.

Directory of src/



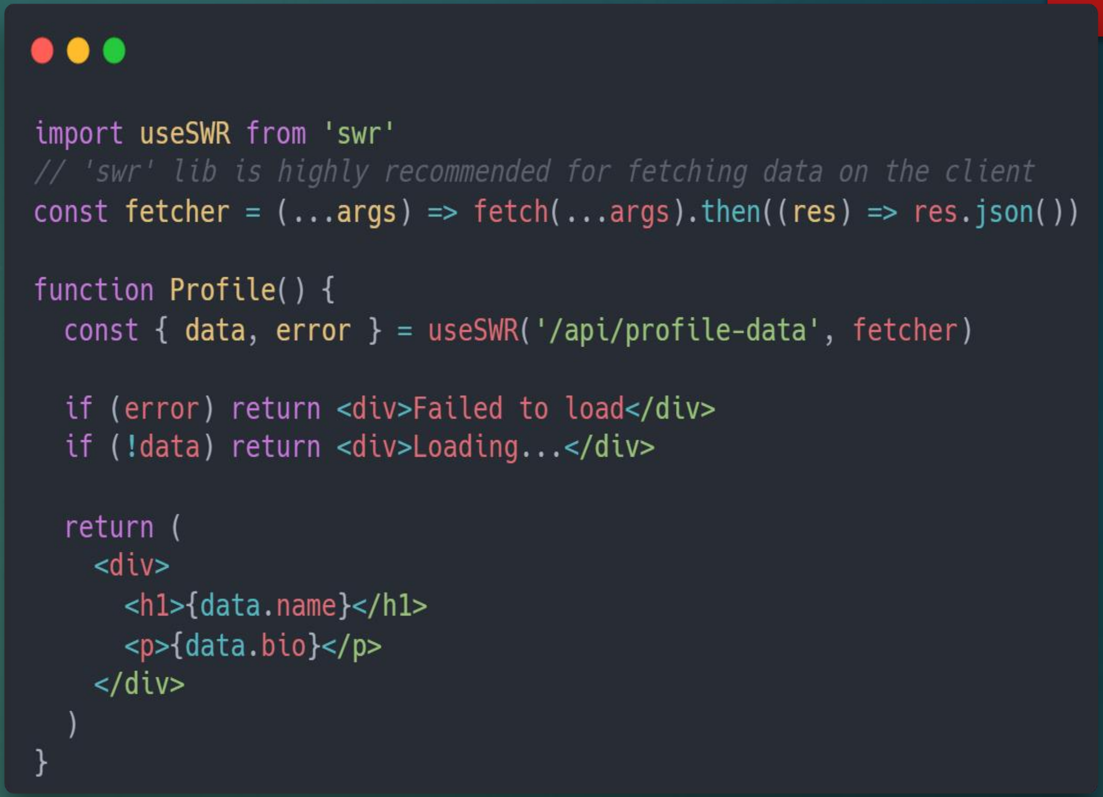
```
├───comps
│       Card.tsx
│       Layout.tsx
│       Nav.tsx
│
├───lib
│       api.ts
│
├───pages
│   │   gallery.tsx              /gallery
│   │   index.tsx                /
│   │   _app.tsx
│   │   _document.tsx
│   │
│   ├───api
│   │       hello.ts             /api/hello
│   │       user.ts              /api/user
│   │
│   └───users
│           [id].tsx             /api/users/[any id here]
│
└───styles
        globals.css
        Home.module.css
```

Client-side fetching



```
import { useState, useEffect } from 'react'

function Profile() {
  const [data, setData] = useState(null)
  const [isLoading, setLoading] = useState(false)

  useEffect(() => {
    setLoading(true)
    fetch('/api/profile-data')
      .then((res) => res.json())
      .then((data) => {
        setData(data)
        setLoading(false)
      })
  }, [])

  if (isLoading) return <p>Loading...</p>
  if (!data) return <p>No profile data</p>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Traditional React implementation (works in Next as well)

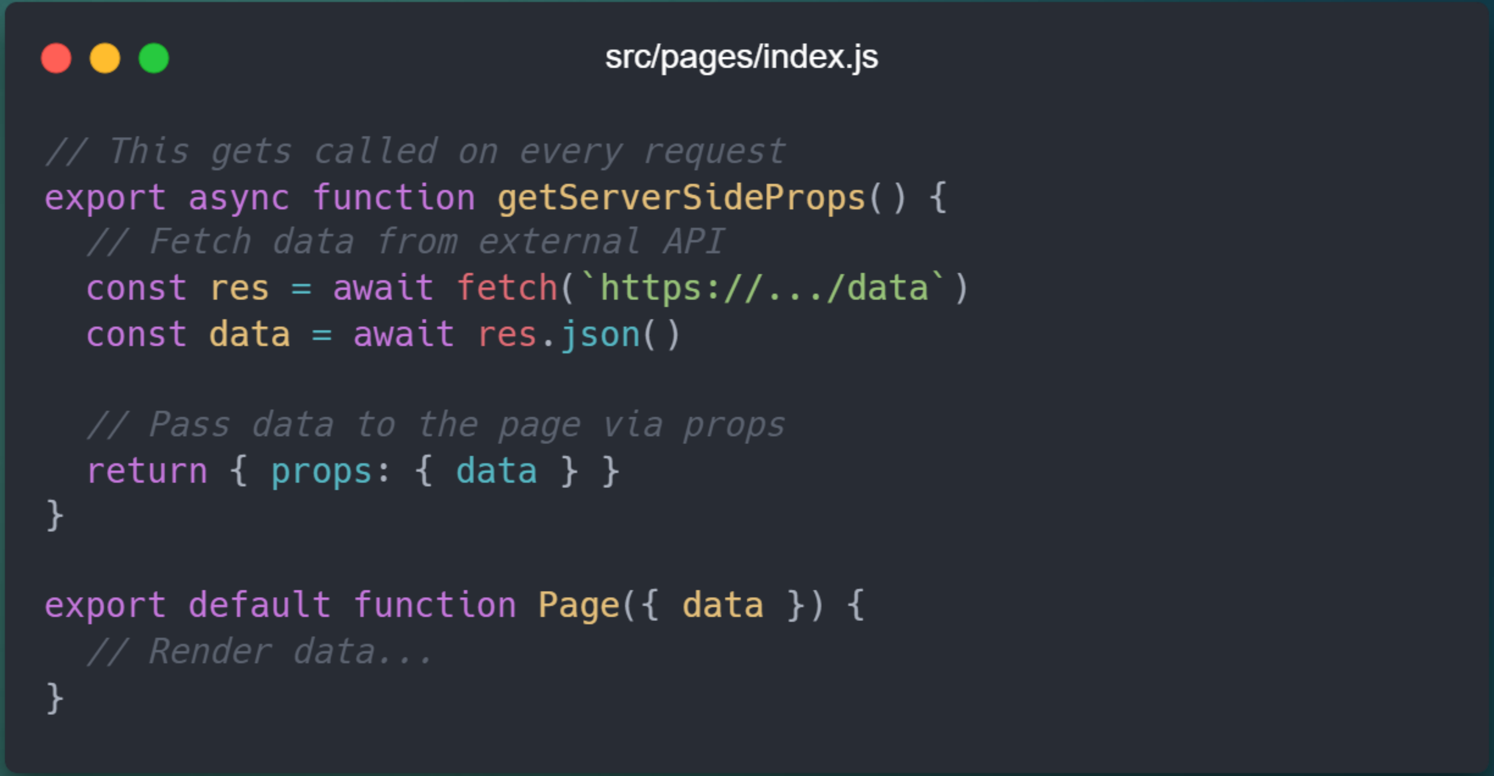


```
import useSWR from 'swr'
// 'swr' lib is highly recommended for fetching data on the client
const fetcher = (...args) => fetch(...args).then((res) => res.json())

function Profile() {
  const { data, error } = useSWR('/api/profile-data', fetcher)

  if (error) return <div>Failed to load</div>
  if (!data) return <div>Loading...</div>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Using SWR

SSR - getServerSideProps

Next.js will pre-render this page on ***each request*** using the data returned by `getServerSideProps`.

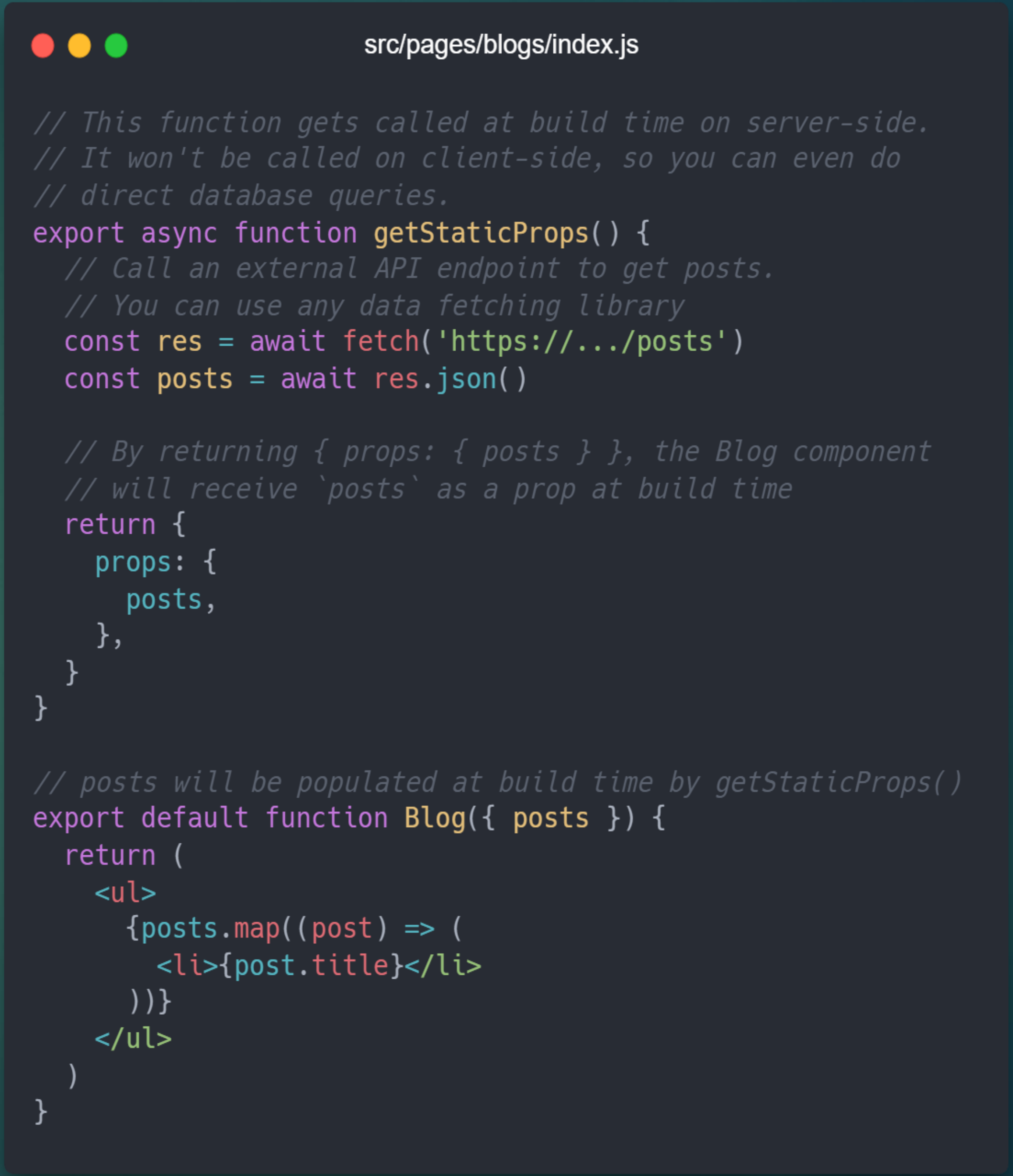


src/pages/index.js

```
// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  // Pass data to the page via props
  return { props: { data } }
}

export default function Page({ data }) {
  // Render data...
}
```

SSG - getStaticProps

SSG (Static Site Generation) will pre-render the page at ***build time*** using props returned by `getStaticProps`.



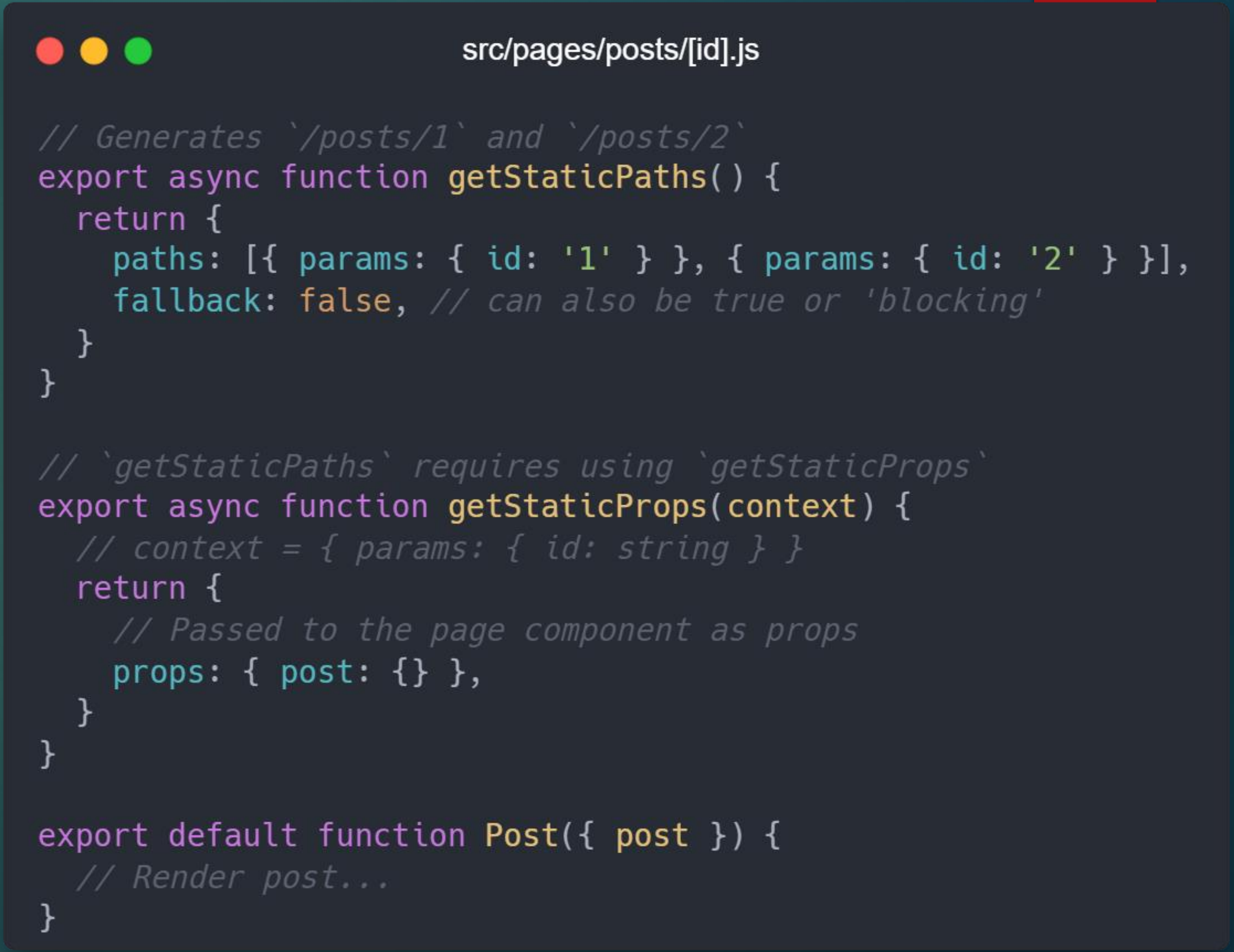
src/pages/blogs/index.js

```
// This function gets called at build time on server-side.
// It won't be called on client-side, so you can even do
// direct database queries.
export async function getStaticProps() {
  // Call an external API endpoint to get posts.
  // You can use any data fetching library
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}

// posts will be populated at build time by getStaticProps()
export default function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}
```

*NOTE: Runs **every request** in development mode.

SSG - getStaticPaths

Static Site Generation with dynamic routes using getStaticPaths allows for the static pre-rendering of all paths programmatically.



src/pages/posts/[id].js

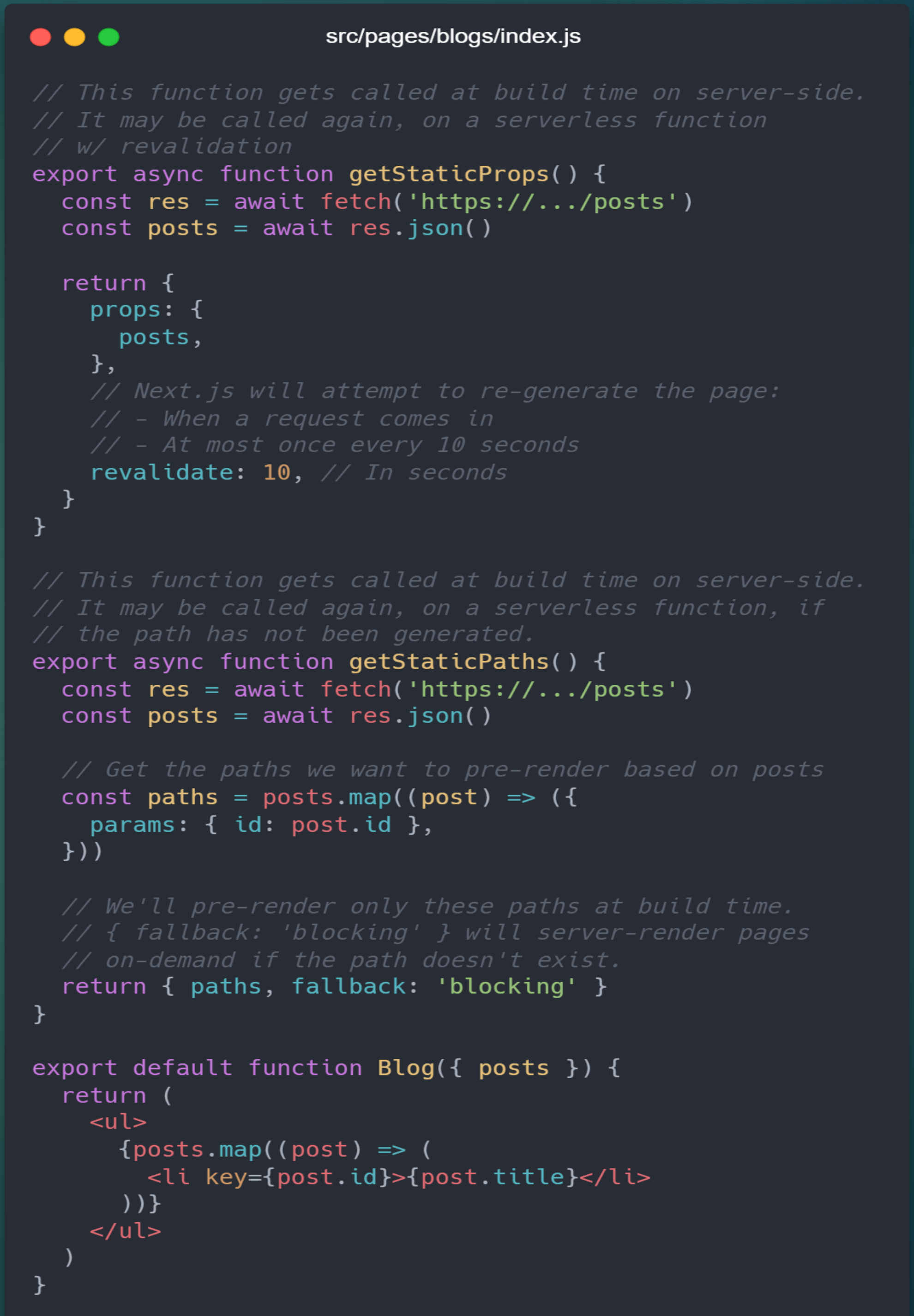
```
// Generates `/posts/1` and `/posts/2`
export async function getStaticPaths() {
  return {
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
    fallback: false, // can also be true or 'blocking'
  }
}

// `getStaticPaths` requires using `getStaticProps`
export async function getStaticProps(context) {
  // context = { params: { id: string } }
  return {
    // Passed to the page component as props
    props: { post: {} },
  }
}

export default function Post({ post }) {
  // Render post...
}
```

Incremental Static Regeneration (ISR) through revalidation

Incremental Static Regeneration (ISR) allows you to use static-generation on a per-page basis, without needing to rebuild the entire site. With ISR, you can retain the benefits of static while scaling to millions of pages.

Here's how it works:

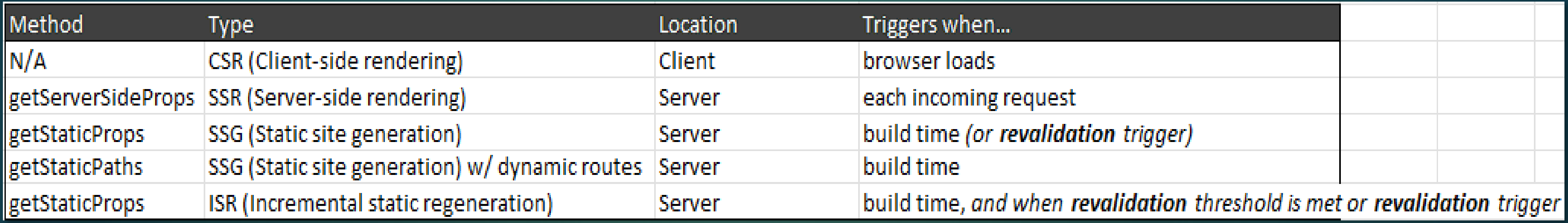
1. When a user requests a page that was generated using `getStaticProps`, Next.js serves the static HTML file from the cache (if available).
2. At the same time, Next.js initiates a revalidation request in the background.
3. If the data has changed since the page was generated, Next.js generates a new HTML file and replaces the cache with the new version.
4. The next time a user requests the page, they will see the new version with updated data.



src/pages/blogs/index.js

```
// This function gets called at build time on server-side.
// It may be called again, on a serverless function
// w/ revalidation
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every 10 seconds
    revalidate: 10, // In seconds
  }
}

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// the path has not been generated.
export async function getStaticPaths() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll pre-render only these paths at build time.
  // { fallback: 'blocking' } will server-render pages
  // on-demand if the path doesn't exist.
  return { paths, fallback: 'blocking' }
}

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

Quick summary



Method	Type	Location	Triggers when...
N/A	CSR (Client-side rendering)	Client	browser loads
getServerSideProps	SSR (Server-side rendering)	Server	each incoming request
getStaticProps	SSG (Static site generation)	Server	build time *(or **revalidation** trigger)*
getStaticPaths	SSG (Static site generation) w/ dynamic routes	Server	build time
getStaticProps	ISR (Incremental static regeneration)	Server	build time, *and when **revalidation** threshold is met or **revalidation** trigger*

Compiled by Zavaar Shah https://github.com/thatziv

https://nextjs.org/docs/basic-features/data-fetching/overview