Next.js Notes

Chapter 1 —> Birth

JavaScript Evolution

- Created by Brendan Eich (1995) at Netscape.
- Next.js was created in **2016** by **Vercel** (led by Guillermo Rauch) to address React's limitations.
- Framework progression: jQuery → Angular → Node.js → React.js → Next.js

Hello World Example

- **Vanilla JS**: Verbose DOM manipulation.
- **jQuery**: Simplified syntax.
- **Angular/React**: More code for this example but scalable in "bigger picture" (component-based).

Why Modern Frameworks?

- **Component Architecture**: Reusable UI pieces (e.g., buttons).
- **Virtual DOM**: Efficient UI updates (only changes rendered).
- **Ecosystem**: Strong community, documentation, and tools.
- Modern frameworks improve efficiency, scalability, and performance.

Chapter 2 —> Introduction

Next.js is a **full-stack web framework** built on top of **React.js** or simply we can say it's a React framework. While React is a **UI library** that focuses on building components, Next.js extends it into a complete framework for building **production-grade web applications**.

What is a Framework?

- A framework serves as a tool equipped with predefined rules and conventions that offer a structured approach for building applications.
- Handles database integration, routing, authentication, etc.
- Helps developers focus on writing application logic rather than low-level setups.

Key features of Next.js:

1. Solves React limitations (SEO, routing, performance)
2. Built-in features:
    - File-based routing
    - Efficient code splitting
    - Hybrid rendering (SSR/SSG)
    - Built-in optimizations (images, fonts, SEO)
    - HMR (Hot Module Replacement)
    - API Routes (backend)
    - Built-in support for Sass
    - CSS modules
    - Data fetching choice (SSG, SSR, ISR)
    - Error handling
    - Metadata API (For SEO)
    - Internationalization(support for any spoken language), etc.

Why Use a React Framework like Next.js?

1. Less Tooling Time
    - No need to configure bundlers, compilers, formatters, etc.
    - Built-in support for routing, rendering, auth, and more.
    - Focus more on business logic and React code.

2. Easy Learning Curve
    - Easier to learn if you're already familiar with React.
    - Includes backend features but without complex setup (no routing config needed).
3. Improved Performance
    - Built-in SSR (Server-Side Rendering) & SSG (Static Site Generation).
    - Automatic code splitting for faster page loads and better UX.
    - React has introduced React Server Components for SSR, but Next.js automates the setup.

    > Follows "Convention over Configuration" = less boilerplate code.

4. SEO Advantage
    - React.js renders everything on the client side, sending a minimal initial HTML response from the server. The server sends a minimal HTML file code and a JavaScript file that the browser executes to generate the HTML —hard for search engines to crawl.
    - Next.js sends **full HTML file** and minimal JavaScript code to render only the content requiring client-side interaction.
    - This improves:
        - Visibility
        - Ranking
        - Traffic
        - User trust

When to Use Next.js over React

Choose **Next.js** when:

- You care about **SEO**

- You want **fast page loads** (via SSR/SSG)
- You don’t want to configure everything yourself
- You want an all-in-one full-stack React framework
- You need **routing, data fetching**, and **backend API** in one codebase

Choose **React (only)** when:

- You're building a **simple SPA or PWA**
- You need complete control over the setup
- You’re integrating into an existing app (e.g., with a non-React backend)

Chapter 3 —> Prerequisites

Web Development Fundamentals

1. HTML -
    a. Structure
    <!DOCTYPE>,<html>,<head>,<body>
    b. Elements
    headings, paragraph, lists, <a>, <img>, <input>, <textarea>, <button>,
    c. Semantics
    header, nav, main, section, aside, footer

```
<header>Site Logo/Navigation</header>
<nav>
  <a href="/">Home</a> | <a href="/about">About</a>
</nav>
<main>
  <section id="intro">
    <h2>Welcome</h2>
    <p>Introduction text...</p>
  </section>
  <aside>Related links (Content indirectly related to main content)</asi
```

```
de>
</main>
<footer>Copyright © 2024</footer>
```

d. Forms
handling user input, perform form validations by using form element and onSubmit event listener

```
<form onsubmit="validateForm()">
  <label for="name">Name:</label>
  <input type="text" id="name" required>

  <label for="email">Email:</label>
  <input type="email" id="email" required>

  <button type="submit">Submit</button>
</form>
```

2. CSS -

a. Structure
Box model - padding, margin, border
Selectors - type, class, id, child, sibling
Typography - font, size, weight, alignment
Colors & Background - colors, gradients, background images

```
/* Box model */
div {
  width: 300px;
  padding: 20px;    /* Inner space */
  border: 2px solid black;
  margin: 30px;     /* Outer space */
}

/* Type */ h1 { color: blue; }
/* Class */ .btn { background: red; }
```

```css
/* ID */ #header { height: 80px; }
/* Child */ ul > li { list-style: none; }
/* Sibling */ h2 + p { margin-top: 0; }

body {
  font-family: 'Arial', sans-serif;
  font-size: 16px;
  line-height: 1.5;
  font-weight: 400/bold;
  text-align: center;
}

.element {
  color: #ffffff; /* Text color */
  background-color: rgba(0,0,0,0.5);
  /* A gradient is like a smooth blend of two or more colors. Instead of
  one solid color, the colors gradually change. */
  background: linear-gradient(to right/135deg, red, yellow);
  background-image: url('image.jpg');
}
```

b. Layout and Positioning (Refer NotesFS)
Display - block, inline, inline-block
Position - relative, absolute, sticky, fixed
Flexbox & Grid

c. Effects
Transition - Learn to create smooth transitions using different CSS properties like delay, duration, property, timing-function
Think of a transition like a magic trick: when you change something—like the color or size of a box —the change doesn't happen instantly; it slides or fades smoothly. You control how long it takes, and how it moves.
Key properties:

- `transition-property`: what you want to change (e.g., `background-color`, `transform`, `width`, `opacity`)

- `transition-duration`: how long the change takes (e.g., `2s` for two seconds)
- `transition-delay`: wait this long before starting (e.g., `0.5s`)
- `transition-timing-function`: how the speed of the change feels like "slow at start," "fast in the middle".

`linear`	Same speed from start to finish
`ease`	Starts slow, speeds up, then slows down
`ease-in`	Starts slow, then speeds up
`ease-out`	Starts fast, then slows down
`ease-in-out`	Slow → Fast → Slow
`cubic-bezier(...)`	Custom timing with control points

Transformations - Explore 2D and 3D transformations like scaling, rotating, translating elements

Think of a piece of paper. You can **rotate it**, **scale it**, or **move it**. CSS lets you do this to elements on a web page.

Types of Transforms:

2D Transforms:

Transform	What it does
`translate(x, y)`	Moves element left/right (x) or up/down (y)
`rotate(deg)`	Rotates the element (like a clock hand)
`scale(x, y)`	Grows or shrinks the element
`skew(x, y)`	Tilts the element

3D Transforms:

Transform	What it does
`rotateX(deg)`	Rotates around X-axis (up/down flip)
`rotateY(deg)`	Rotates around Y-axis (sideways flip)
`translateZ(px)`	Moves closer/farther away (depth)

Animations - Learn how to create animations using keyframes

Think of a cartoon—it's made of **frames**. In CSS, **keyframes** tell the

browser how an element should change over time.
How It Works:
Define `@keyframes name { ... }` with percentages (from 0% to 100%).
Apply that animation with:

- `animation-name`
- `animation-duration`
- `animation-timing-function`, etc.

Shadows and Gradients - Explore with box shadows and linear or radial gradients

Shadows

- **Box-shadow**: gives an element a shadow, like a floating box.

  Syntax: `box-shadow: offsetX offsetY blur spread color;`

Gradients

- **Linear-gradient**: colors fade in a straight line.
- **Radial-gradient**: colors fade in a circle (like a spotlight).

```
/* Transition */
.button {
   transition: <property> <duration> <timing-function> <delay>;
  transition: background-color 0.3s ease 2s;

   /* comma-separate transitions for multiple properties */
  transition: background-color 0.5s ease, transform 0.3s linear;
}
.button:hover {
  background-color: blue;
}

/* Transformation */
.element {
  transform: rotate(15deg) scale(1.1);
}
```

```
/* Animation */
@keyframes slide {
  from { transform: translateX(-100%); }
  to { transform: translateX(0); }
}
.slide-in {
  animation: slide 0.5s forwards;
}

/* Shadows and Gradient */
.card {
  box-shadow: 2px 2px 10px rgba(0,0,0,0.1);
  background: linear-gradient(45deg, red, blue);
}
```

d. Advanced (Plus)
Learn how to use CSS processors like sass or frameworks like TailwindCSS for more powerful and efficient styling
What are they?

- **Sass**: a helpful tool that lets you write **variables**, **nest CSS rules**, and reuse code pieces. Then it *magically* turns into normal CSS.
- **Tailwind CSS**: a toolkit with lots of tiny building blocks (classes) you can combine quickly to style your page. No writing long CSS—just use class names!

Since these require setup and not pure HTML+CSS, here's a simple illustration to show how they make styling easier:

```
/* Imagine this is Sass — it doesn't work directly in HTML */
/* Pretend file: style.scss */
$main-color: tomato;

.nav {
  background: $main-color;
```

```
  ul {
    list-style: none;
    li {
      display: inline-block;
      margin-right: 10px;
    }
  }
}

/* This compiles to regular CSS like: */
.nav { background: tomato; }
.nav ul { list-style: none; }
.nav ul li { display: inline-block; margin-right: 10px; }
```

3. JS -
    a. Variables and Data Types
    b. Operators
    c. Control Flow
    d. Functions
    e. DOM Manipulation

Modern JavaScript

1. ES6 Features
    a. Arrow Functions
    b. Destructuring
    c. Spread Syntax
    d. Template Literals
    e. Modules

```
// Arrow function
const add = (a, b) => a + b;
```

```
// With single parameter
const square = x => x * x;


// Array destructuring
const [first, second] = [10, 20];
// Object destructuring
const { name, age } = { name: 'John', age: 30 };


// Array spreading
const nums1 = [1, 2, 3];
const nums2 = [...nums1, 4, 5]; // [1, 2, 3, 4, 5]
// Object spreading
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a:1, b:2, c:3 }


// Template Literals
const name = 'John';
const greeting = `Hello ${name}!`;
// Multiline strings
const message = `
  This is a
  multi-line
  string
`;


// Exporting (math.js)
export const add = (a, b) => a + b;
export const PI = 3.14;
// Importing (app.js)
import { add, PI } from './math.js';
```

2. Asynchronous Programming
    a. Promises
    b. Async/Await
    c. Fetch API
    d. Axios

```
// Promises
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve('Data received');
    } else {
      reject('Error fetching data');
    }
  }, 1000);
});
fetchData
  .then(data => console.log(data))
  .catch(error => console.error(error));


// Async/Await
async function getData() {
  try {
    const response = await fetch('api/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

```
// Fetch API
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));


// Axios
axios.get('https://api.example.com/data')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

3. Additional JS concepts
   a. Array Methods - `map`, `filter`, `reduce`, `slice`, `splice`, `forEach`, `includes`, `join`, `reverse`
   b. Error Handling

```
const numbers = [1, 2, 3, 4, 5, 6];

// 1. map - double each number
const doubled = numbers.map(num => num * 2);
console.log('map:', doubled); // [2, 4, 6, 8, 10, 12]

// 2. filter - get even numbers
const evens = numbers.filter(num => num % 2 === 0);
console.log('filter:', evens); // [2, 4, 6]

// 3. reduce - sum all numbers
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log('reduce:', sum); // 21

// 4. slice - get elements from index 1 to 3 (not inclusive)
const sliced = numbers.slice(1, 4);
console.log('slice:', sliced); // [2, 3, 4]

// 5. splice - remove 2 elements starting from index 2 and insert 99, 100
```

```
const spliced = [...numbers]; // make a copy to avoid modifying the origina
l
spliced.splice(2, 2, 99, 100);
console.log('splice:', spliced); // [1, 2, 99, 100, 5, 6]

// 6. forEach - log each element
console.log('forEach:');
numbers.forEach(num => console.log(num)); // 1 2 3 4 5 6

// 7. includes - check if 4 is in the array
const hasFour = numbers.includes(4);
console.log('includes:', hasFour); // true

// 8. join - join elements into a string with "-"
const joined = numbers.join('-');
console.log('join:', joined); // "1-2-3-4-5-6"

// 9. reverse - reverse the array
const reversed = [...numbers].reverse(); // copy to avoid mutating original
console.log('reverse:', reversed); // [6, 5, 4, 3, 2, 1]


// Error handling
try {
  // Code that might throw an error
  const data = JSON.parse(invalidJson);
} catch (error) {
  // Handle error gracefully
  console.error('Failed to parse JSON:', error.message);
  showUserMessage('Invalid data format. Please try again.');
} finally {
  // Cleanup code
  console.log('Operation attempted');
}
```

The Ecosystem

1. Foundations
    a. Node.js
        - JavaScript runtime built on Chrome's V8 engine
        - JavaScript runtime outside the browser. Allows running JavaScript on the server
        - Includes npm (Node Package Manager)
    b. NPM
        - Package manager for JavaScript
        - Install packages: `npm install package-name`
        - Initialize project: `npm init`
2. Bundlers and Compilers
    a. Webpack: Bundles all JS/CSS/images into a single optimized file.
    b. Babel: Transpiles modern JavaScript to ensure compatibility with all/older browsers or we can say it transforms JS into backwards-compatible code.

> Next.js handles this for you automatically under the hood!

3. Version Control
    a. Git - Version control system.
    b. GitHub - Cloud-based Git repository management and for collaboration.

```
feat: add user authentication
|    |
|    +-> Summary in present tense
|
+----——→ Type: chore, docs, feat, fix, refactor, style, test

// Example commit message:
fix(login): validate email format before submission
```

Added email validation regex to prevent invalid email submissions.
The validation now checks for basic email format before allowing
form submission.
Fixes #123

React JS

1. Fundamentals

    a. Components
    Components are the building blocks of React applications. They split the UI into reusable, isolated pieces.

    - **Functional Components**: JavaScript functions returning JSX
    - **Class Components**: ES6 classes extending `React.Component`

```
// Functional Component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

// Class Component
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

    JSX & Component Lifecycle
    JSX (JavaScript XML) is a syntax extension that lets you write HTML inside JavaScript.

    Component Lifecycle (Class Components)
    Key methods:

    - `componentDidMount()`: called after the component is rendered to the DOM.

- `componentDidUpdate()`: called after the component updates.
- `componentWillUnmount()`: called before the component is removed from the DOM.

```
const element = <h1 className="greeting">Hello, world!</h1>;
// Compiled to:
React.createElement('h1', { className: 'greeting' }, 'Hello, world!');
```

b. State and Props

- **Props**: Read-only, passed from parent to child
- **State**: Mutable data managed within a component

```
function Greeting(props) {
  const [name, setName] = useState(props.defaultName);
  return (
    <div>
      <h1>Hello, {name}</h1>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
    </div>
  );
}
```

> P.S., Don't forget to learn about the special "Key" prop when rendering the dynamic list with map method.

The `key` Prop
Purpose - Helps React identify dynamic list items for efficient updates:

```
const todoItems = todos.map(todo ⇒ (
  <li key={todo.id}>{todo.text}</li>));
```

Why Required?
Without `key`, React may re-render entire lists inefficiently.
Rules:

- Must be unique among siblings
- Avoid using array indices (unless list is static)

```
// ✅ Good (Unique ID)
{todos.map(todo ⇒ <Todo key={todo.id} {...todo} />)}

// ⚠️ Avoid (Index causes issues with reordering)
{todos.map((todo, index) ⇒ <Todo key={index} {...todo} />)}
```

c. Events
Synthetic event handlers (camelCase naming):

```
<button onClick={(e) ⇒ console.log("Clicked", e)}>Click</button>
```

d. Conditional Rendering
Render content based on conditions:

```
{isLoggedIn ? <LogoutButton /> : <LoginButton />}
{unreadMessages.length > 0 && <h2>You have messages!</h2>}
```

2. Hooks & Router

a. Hooks

`useState`
Lets you add state to function components.

```
const [count, setCount] = useState(0);
```

```
<button onClick={() => setCount(count + 1)}>Count: {count}</button>
```

useEffect

Lets you handle side effects (data fetching, subscriptions) in function components. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React classes.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Re-run when count changes
```

useRef

It returns a mutable ref object whose `.current` property is initialized to the passed argument. It can be used to access a DOM element directly.

```
const inputRef = useRef();
<input ref={inputRef} />
<button onClick={() => inputRef.current.focus()}>Focus</button>
```

useContext

Access context without prop drilling:

```
const ThemeContext = React.createContext('light');
const theme = useContext(ThemeContext); // 'light'
```

useMemo

It returns a memoized value. It only recomputes the memoized value when one of the dependencies has changed.
Memoizes expensive calculations:

```
const expensiveValue = useMemo(() => computeValue(a, b), [a, b]);
```

useCallback

It returns a memoized callback. It is useful when passing callbacks to optimized child components that rely on reference equality to prevent

unnecessary renders.
Memoizes functions to prevent re-renders:

```
const memoizedCallback = useCallback(() => doSomething(a, b), [a,
b]);
```

b. Router
React Router is a standard library for routing in React. It enables the navigation among views of various components.

Routes

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Home from './Home';
import UserProfile from './UserProfile';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="users/:id" element={<UserProfile />} /> {/* Dynami
c           User Profile */}
      </Routes>
    </BrowserRouter>
  );
}
```

Route Parameters
Route parameters are placeholders in the URL that can capture values at their position. For example: /users/5, /users/abc, /users/99

```
import { useParams } from 'react-router-dom';

function UserProfile() {
  const { id } = useParams(); // id will be 5, abc, or 99
```

```
  return <h2>User Profile ID: {id}</h2>;
}
```

Nested Routes

Nested routes allow you to define routes inside other components. Useful for apps where some pages share layout or structure.

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Users from './Users';
import UserDetail from './UserDetail';
import NewUser from './NewUser';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="users" element={<Users />}>
          <Route path=":id" element={<UserDetail />} /> {/* /users/123
*/}
          <Route path="new" element={<NewUser />} /> {/* /users/new
*/}
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

// Inside the Users component, we must add <Outlet /> to display the
nested content. Users.jsx file -
import { Outlet } from 'react-router-dom';

function Users() {
  return (
    <div>
      <h2>All Users</h2>
```

```
      {/* This is where the child components like UserDetail or NewUser
will be rendered */}
      <Outlet />
    </div>
  );
}
```

c. State Management

Context API: The Context API provides a way to pass data through the component tree without having to pass props down manually at every level.

Redux: Redux is a predictable state container for JavaScript apps. It helps you manage global state.

Zustand: Zustand is a small, fast and scaleable barebones state-management solution.

d. Style

Inline styles - Inline styles are written as objects in React.
CSS Modules - CSS Modules allow you to write CSS that is scoped to a component.
Sass - Sass is a CSS preprocessor that adds features like variables, nesting, and mixins.
TailwindCSS is a utility-first CSS framework.
Material UI

```
// Inline styles
const divStyle = {
  color: 'blue',
  fontSize: '20px',
};
function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}

// CSS Modules
```

```
import styles from './Button.module.css';
function Button() {
  return <button className={styles.error}>Delete</button>;
}

// TailwindCSS
function Button() {
  return <button className="bg-blue-500 hover:bg-blue-700 text-whi
te py-2           px-4 rounded">Button</button>;
}
```

What is CSS-in-JS?
CSS-in-JS is a styling technique where CSS is composed using JavaScript. Instead of writing traditional `.css` files, you define your styles within JavaScript files (typically in React apps).

Use CSS-in-JS when:

- You want tightly coupled styles and components.
- You need dynamic styles based on props/state.
- You want automatic vendor prefixing.
- No class name collisions.
- You're building large, maintainable React apps.

Styled Components
Styled Components is a CSS-in-JS library for React and React Native that lets you use tagged template literals to style your components.

```
// npm install styled-components

import styled from 'styled-components';

const Button = styled.button`
  background-color: ${(props) => (props.primary ? 'blue' : 'gray')};
  color: white;
  padding: 10px 20px;
```

```
  border: none;
  border-radius: 5px;
`;

function App() {
  return (
  {/* In React, a Fragment <> is a way to group multiple elements witho
ut     adding an extra node (like a <div>) to the DOM. */}
    <>
      <Button primary>Primary Button</Button>
     <Button>Default Button</Button>
    </>
  );
}
```

Emotion
Emotion is another powerful CSS-in-JS library similar to Styled Components, but with more flexibility and performance optimizations.

3. Forms & HTTP Requests

   Learn to create form validation, handling form submission with or without using third party libraries like Formik , React Hook Form.
   Formik is a popular library for building forms in React.
   React Hook Form is another library that simplifies form handling.
   HTTP requests in React can be made using `fetch` or libraries like `axios`.

Backend

1. Basics - HTTP Protocol, APIs and REST, HTTP Methods, Status Codes, HTTP Headers, Request and Response, Resource URI
2. CRUD
3. Authentication and Authorization - User Sessions, JWT, Cookies, Permissions and Roles
4. Database
5. Deployment -

a. Environments (Production, Development, Staging)

b. Hosting Platforms (Vercel, Netlify, Firebase, Render, Heroku, AWS Amplify, Railway)

c. Advanced:

   CI/CD (GitHub Actions, Jenkins, AWS CodePipeline and more...)

   - **CI** = Automatically test every code change.
   - **CD** = Automatically deploy if tests pass.

   Docker

Next.js

1. Fundamentals
   - Why use Next.js? (Performance, SEO, SSR/SSG)
   - Core concepts: Components, state, modules
2. Architecture
   - App vs Pages directory
   - Client vs Server rendering
3. File-Based Routing
   - Simple Routes: `app/about/page.js` → `/about`
   - Nested Routes: `app/products/list/page.js` → `/products/list`
   - Dynamic Routes: `app/products/[id]/page.js` → `/products/123`
   - Parallel
   - Intercepting
   - Route Groups: `(admin)/dashboard/page.js` → `/dashboard`
4. Styling
   - CSS Modules, Tailwind CSS, Sass
5. Data Fetching
   - SSG: Static content (e.g., blogs)

- SSR: Dynamic content (e.g., user dashboards)
- ISR: Hybrid approach (e.g., product listings)
- CSR: Client-side fetching (e.g., dashboards)

6. SEO & Metadata
   - Learn Static, Dynamic and File Based Metadata
   - Optimize pages for search engines
   - Use `metadata` objects or file-based conventions
7. Error/Loading States
   - Learn `error.js`, `loading.js`, `not-found.js` and `layout.js` files
8. Authentication
   - NextAuth.js, Clerk
9. API Routes
   - Route Handlers
     Create custom request handlers - Static and Dynamic Route Handlers
   - Middleware
   - Supported HTTP Methods
   - NextResponse
   - CORS and Headers
10. Databases
    - Integrate MongoDB, PostgreSQL, or Prisma

Chapter 4 —> How It Works

Traditional Web Development (Vanilla HTML, CSS, JS)

- **How it works:**
  - Browser (client) requests a webpage → Server sends **HTML, CSS, JS** files.
  - Browser parses HTML, applies CSS, and executes JS for interactivity.

    - For multiple pages, the client requests each page separately, and the server sends new files.
- **Limitations:**
    - **Processing:** Mostly client-side → Can strain low-end devices.
    - **Bandwidth:** Heavy usage due to full file transfers per request.
    - **Load Time:** Slower initial load since all files must be fetched and parsed.

The React Way (CSR)

- **How it works:**
    - Server sends a **minimal HTML + bundled JS** file.
    - React renders the app **client-side** using Virtual DOM → Updates only necessary parts.
    - Navigation uses **React Router** (no full page reloads).
- **Limitations:**
    - **Complexity:** Managing state, props, and components can be challenging.
    - **Processing:** Heavy reliance on client-side JS → Slower on low-end devices.
    - **SEO:** Search engines struggle with JS-rendered content.

The Next.js Way (Hybrid Rendering - SSR + CSR)

- **How it works:**
    - Server executes React components → Generates **pre-rendered HTML + CSS + JS**.
    - Client receives fully rendered HTML → Faster initial load.
    - This HTML file includes initial content, fetched data, and React component markup, making the client render it immediately without waiting for JavaScript to download and execute.
    - The server will still send the JavaScript code as needed for the user interaction.

    - **Hydration:** JS attaches event handlers to static HTML (if mismatched → Hydration error).
    - **For Subsequent Requests:** You have control where to render your page content (SSR or CSR).
- **Advantages:**
    - **Faster Load Time:** Pre-rendered HTML improves performance.
    - **Better SEO:** Search engines can crawl server-rendered content.
    - **Flexibility:** Choose rendering method (SSR/CSR) per page.

Key Terms:

- **SSR (Server-Side Rendering):** Server generates HTML → Better SEO & performance.
- **CSR (Client-Side Rendering):** Browser renders using JS → More dynamic but slower initial load.
- **Hydration:** Attaching JS interactivity to pre-rendered HTML.