Next.js Notes + TailwindCSS

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);
}
```

```css
/* 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 (Syntactically Awesome Style Sheets)**: a helpful tool that lets you write **variables**, **nest CSS rules**, and reuse code pieces. Then it *magically* turns into normal CSS. For using CSS with superpowers, i.e., Sass, we'll have to install the dedicated package for it, `npm install --save-dev sass`.
- **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 />} /> {/* Dynamic 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 Routes
    - Intercepting Routes
    - 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.

Chapter 5 —> Create Next.js Application

Creating Next.js Application Options

1. **Manual Installation**
    - Configure packages, files, and folder structure manually.
    - Full control but time-consuming.
    - **npm**: Installs/manages packages (e.g., Axios, Redux).
2. **Automatic Installation (Recommended)**
    - Uses `create-next-app` CLI tool.
    - Quick setup with pre-configured templates (TypeScript, Tailwind CSS, etc.).

- Zero dependency; runs via `npx` (no global install needed).
- **npx**: Used to run command-line tools and execute commands from packages without global installation (e.g., `create-next-app`).

Project Structure

1. `app/`
   - It's the root of the application. Root directory for frontend routes and backend code.
   - Key files:
     - `favicon.ico`: Browser tab icon (replaceable).
     - `globals.css`: Global CSS (variables, fonts).
     - `layout.js`: Root layout (shared across all routes).
       - Wrap with providers (Redux), add metadata, or Navbar here.
     - `page.js`: Home route (`/`). Renders only on the homepage.
     - `page.module.css`: Scoped CSS for `page.js`.
2. `node_modules/`
   - Stores all dependencies (e.g., React, Next.js). Managed by npm.
3. `public/`
   - Static assets (images, fonts). Automatically optimized.
4. Configuration Files
   - `.gitignore`: Excludes files (e.g., `node_modules`) from Git.
   - `jsconfig.json`: Configures import aliases (e.g., `@/*` for `./`).
   - `package-lock.json`: Locks dependency versions (critical for consistency).
   - `package.json`: Project metadata + scripts (e.g., `dev`, `build`).
5. `README.md`
   - Project documentation (setup, usage, contributions).

```
// module css import
import styles from './page.module.css';

// using @/* for any files and folders located in this location ./* i.e., the root
// We can change @/* to @* or even #/*
import something from '../../components';
import something from '@/components';
```

Some Next.js Concepts

1. **Fast Refresh**

   Real-time UI updates during development (no manual reload).

2. Children components (routes) are injected via `{children}` prop in `layout.js`.

Chapter 6 —> Client vs. Server

Definitions

- **Client**: The user's device (browser, mobile) that requests and displays the UI.
- **Server**: A powerful remote machine hosting your app. Handles computations, data fetching, and pre-renders components.

Evolution: Pages Router → App Router

- **Pre-Next.js 13**: Only **pages** (routes like `/`, `/about`) could be server-rendered.
  - Led to prop drilling and duplicate API calls.
- **Next.js 13+ (App Router)**: **Component-level SSR** – now individual components can be server/client-rendered.

Key Improvement: Fetch data *inside* components (no prop drilling).

- Pages Router required `getServerSideProps` / `getStaticProps` at page level.
- App Router allows data fetching in *any* component (via `async/await` in Server Components).

Server vs. Client Components

Server Components	Client Components
Render on the server.	Render in the browser.
No interactivity (no hooks, events).	Handle clicks, inputs, hooks (e.g., `useState`).
Default in Next.js.	Opt-in with `"use client"` directive.
Smaller JS bundle, better SEO.	Larger JS, but enables interactivity.
Directly fetch data (`async/await`).	Use `useEffect`/SWR for client-side fetching.

```
// Server Component (default)
export default function Page() {
  return <h1>Hello, Server!</h1>;
}

// Client Component (opt-in)
"use client";
export default function Button() {
  return <button onClick={() => alert("Hi")}>Click Me</button>;
}
```

When to Use Which?

- **Server Component**: Static content, data fetching (e.g., blog post, product listing).

> Next.js mirrors server logs in the browser console in development mode, **so developers can see server-side behavior without checking terminal logs**.

- **Client Component**: Interactive elements (e.g., buttons, forms, animations).

What do you need to do?	Server Component	Client Component
Fetch data.	✓	✗
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server / Reduce client-side JavaScript	✓	✗
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	✗	✓
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use React Class components	✗	✓

Key Benefits of Server Components

1. **Faster Loads for UX**: Pre-rendered HTML reduces browser workload.
2. **Smaller JS Bundle**: Less client-side JavaScript.
3. **Better SEO**: Content is ready for crawlers.
4. **Efficient Utilization of Server Resources**: Fetching data closer to the server, the time required to retrieve data is reduced, resulting in improved performance.

Key Concepts

1. **"use client"**:
    - Any component/file importing a client component **must** be a client component.

- Don’t include Server Component inside Client Component as all imports inside a Client Component become part of the client bundle.
- In simple terms, when we use "use client" in a file, all the other modules imported into that file, including child server components, are treated as part of the client module.
- In case we encounter such scenario to use SC in CC *(Solution: Lift server components up or pass as* `children` *)*. Detailed explanation provided further.

2. **Pre-rendering**:
    - Next.js generates HTML for both Server/Client Components upfront (faster load).
    - Client components are **pre-rendered** on the server (for SEO/UX) but hydrate on the client.
3. **Static Rendering**:
    - Next.js, by default, performs static rendering, which means it pre-renders the necessary content on the server before sending it to the client. This pre-rendering process includes server and client components that can be pre-rendered without compromising functionality.
    - Server Components are pre-rendered at build time (non-dynamic data).
4. So basically, two things happen:
    a. Server Components are guaranteed to be only rendered on the server.
    b. On the other hand, client components are primarily rendered on the client side.

Best Practices

- **Minimize Client Components**: Use only for interactivity.
- **Fetch Data in Server Components**: Avoid passing data through props.
- **Avoid Nesting Server in Client**: Breaks server rendering.

Chapter 7 —> Routing

Next.js uses the file based router system to define routes. No external library needed.

- **Folders** define routes.
- **Files** define the UI for that route segment (e.g., `page.js`).

Simple Routes

Create a folder (use **kebab-case when writing route names**) and add a `page.js` file inside it.

- `app/about/page.js` → `/about` route.

Nested Routes

Create folders inside other folders.

- `app/projects/list/page.js` → `/projects/list` route.

Dynamic Routes

For routes that change based on data (e.g., blog posts, product IDs).

- **Syntax**: Wrap a folder name in square brackets: `[folder-name]`.
- **Example**: `app/projects/[slug]/page.js` handles routes like `/projects/jobit`, `/projects/carrent`, etc.
- **Accessing Data**: The dynamic segment is passed as a `params` prop to the page.js file. So we can de-structure it like ProjectDetails ({ params }) function in page.js file.
    - `app/projects/[slug]/page.js` → `params.slug` contains the value (e.g., "jobit").

Route Groups

Organize related routes into logical groups **without affecting the URL path**.

- **Syntax**: Wrap the group folder name in parentheses: `(group-name)`.
- **Use Case**: To avoid clutter in the `app` directory.
- **Example**:
    - Create `app/(auth)/sign-in/page.js` → URL is `/sign-in` (not `/auth/sign-in`).

- Create `app/(auth)/sign-up/page.js` → URL is `/sign-up`.

Parallel Routes

Core Concept: Allows simultaneous, independent rendering of multiple pages within a single layout. Each section (called a "slot") is defined as a separate page component and can have its own loading state, error handling, and navigation.

Mechanism: Implemented using named **slots** (folders prefixed with `@`, e.g., `@analytics`, `@team`). These slots are automatically passed as props to the parent layout and can be rendered alongside the main `children` content.

Key Benefits:

- **Modular Layouts:** Construct complex dashboards or interfaces with independent sections.
- **Conditional UI:** Slots can be rendered or hidden based on application state (e.g., user authentication).
- **Independent State:** Each slot manages its own data fetching and state without affecting others.

Use Cases: Ideal for dashboards, split-view layouts, and conditionally rendered sections like login modals or sidebars.

Intercepting Routes

Core Concept: A technique to "catch" a navigation event and display the target content within the current layout (e.g., as a modal or overlay) instead of performing a full page transition. This preserves the user's context and creates a seamless, app-like experience.

Mechanism: Uses special folder naming conventions to define the interception level:

- `(.)` - Intercept from the same level
- `(..)` - Intercept from the parent level
- `(..)(..)` - Intercept from two levels up
- `(...)` - Intercept from the root level

Key Benefit: Enables modal-like behavior where content appears over the current page. The original page remains intact in the background, and the URL still reflects the intended destination.

Key Takeaways

- **Simplicity**: File-based routing is intuitive and eliminates complex setup.
- **Flexibility**: Dynamic routes and route groups provide powerful organization without compromising URL structure.
- **Power**: Advanced features like Parallel and Intercepting routes enable complex, modern UX patterns.
- **Shared Layout**: Import a component (like a Navbar) into the root/parent component of the routes i.e. `app/layout.js` to display it on every page. We can also import Navbar in each route page but that is not feasible or good practice.

Chapter 8 —> Rendering

Core Concepts

- **Rendering**: The process of generating the UI from code.
- **Environments**:
  - **Client (CSR)**: User's browser. Best for pure interactivity (e.g., B2B dashboards). Sacrifices SEO.
  - **Server (SSR)**: Your deployment server. Best for SEO, performance, and security.

	Client	Server
Rendering Process	Occurs on the user's browser	Happens on the server before sending the page to the client's browser
Interactivity & Load Time	Provides a dynamic and interactive user experience	Provides a fully rendered HTML page to the client resulting in faster initial page load time
Fetching & SEO	Smoother transition between the pages and real-time data fetching	Fully rendered content enhancing search engine rankings and social media sharing previews
Load & Performance	Reduced server load and potentially lower hosting costs as the client's browser is responsible for handling the rendering.	Performs well on any slower device as rendering is done on the server
Consistent Rendering	Compatibility and performance depend on the user's device configuration.	Consistent rendering across any devices regardless of the configuration reducing the risk of compatibility issues
Security	Potential risk of security vulnerabilities such as Cross-Site Scripting (XSS), Code Injection, Data Exposure, etc.	Reduces the amount of client-side JavaScript code sent to user's browser thus enhancing security by limiting potential vulnerabilities

- **Time Periods**:
  - **Build Time**: It's a series of steps where we prepare our application code for production involving the steps of code compilation, bundling, optimization, etc.

    In short, build time or compile time is the time period in which we, the developer, is compiling the code.

    Remember the `npm run dev` script? It's that command that generated the build of our application containing all the necessary static files, bundling, optimization, dependency resolution, etc.

- **Run Time**: When the app is executing and responding to user requests. It's about handling user interaction, such as user input, responding to events, to data processing, such as manipulating/accessing data and interacting with external services or APIs.

Runtime Environments (RTE)

Next.js provides two RTEs to execute code:

- **Node.js Runtime (Default)**: Full access to Node.js APIs.
- **Edge Runtime**: Lightweight, based on Web APIs. Limited Node.js API support.
- **Switch Runtime**: Define in any component or page:

```
export const runtime = 'edge'; // or 'nodejs'
```

Server Rendering Strategies

Next.js allows mixing these strategies in the same app.

Static Site Generation (SSG)

- **When**: At **Build Time**.
- **How**: Pages are pre-rendered to HTML/CSS/JS during `npm run build`.
- **Pros**: Blazing fast, easily cached on a CDN, great SEO.
- **Cons**: Content is static. Requires a full rebuild to update.
- **Use Case**: Blogs, documentation, marketing sites (content rarely changes).

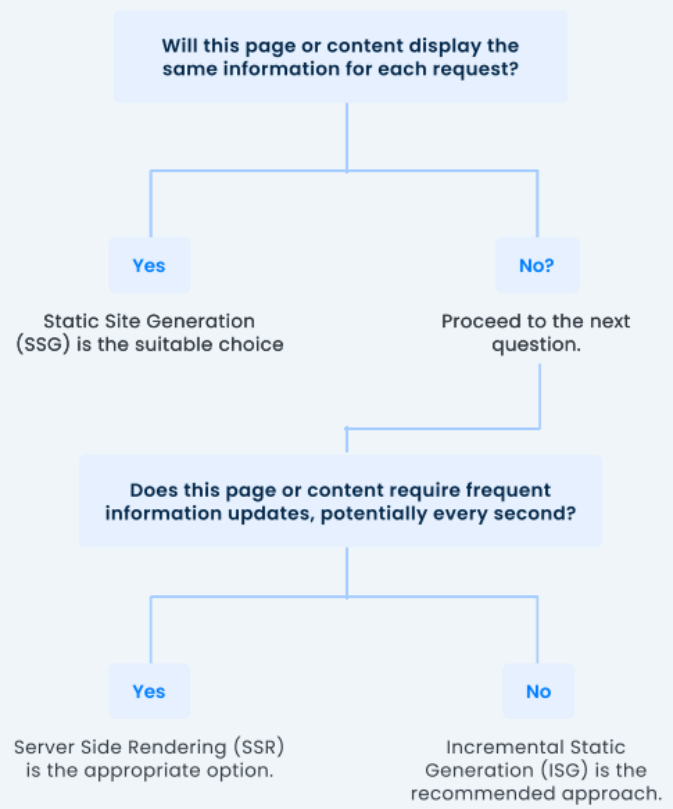
Incremental Static Regeneration (ISR)

- **What**: Enhanced SSG.
- **How**: Pages are generated at build time **or on-demand** after build. You can set a time to revalidate (update) pages in the background.
- **Pros**: Get SSG benefits + can update content without a full rebuild.
- **Use Case**: Product listings, news sites (mix of static and occasionally updated content).

Server-Side Rendering (SSR)

- **When**: At **Run Time** (on every user request).
- **How**: The server generates HTML dynamically for each request.
- **Pros**: Fully dynamic & interactive, real-time data, user-specific content.
- **Cons**: Higher server load, slower than SSG/ISR, harder to cache.
- **Use Case**: Authenticated dashboards, real-time apps (chat, live feeds), e-commerce pages with personalization.

When to use which?



Key Takeaways

Next.js gives you the flexibility to choose the best rendering strategy **per page or even per component**:

- **Default**: SSG for maximum performance.
- **Need updates?**: Use ISR.
- **Fully dynamic?**: Use SSR.

Chapter 9 —> Data Fetching

The Shift: From Client to Server

- **Traditional React (Client-Side):**

```
// In a Client Component ("use client")
const [data, setData] = useState(null);
useEffect(() => {
  fetch('/api/data').then(res => res.json()).then(setData);
}, []);
```

  - Requires `useEffect`, state management. Runs in the browser.
  - **Process:** Component mounts → `useEffect` triggers → API call made → response stored in state → UI updates.

- **Next.js (Server-Side with RSCs):**

```
// In a Server Component (Default)
async function DataComponent() {
  const data = await fetch('/api/data').then(res => res.json());
  return <div>{data.message}</div>;
}
```

  - Direct `async/await` in the component. Runs on the server. Simpler, less code, and better for SEO and performance.
  - **Process:** Server executes component → fetches data → renders final HTML → sends it to the client.

Controlling Rendering Strategies with Caching (`cache` & `revalidate`)

St**atic Site Generation (SSG) - Default**

- **Behavior:** Data is fetched and cached at **build time**. The generated page is served from a CDN for every request.
- **How to:**

- Do nothing (it's the default), or explicitly set a long revalidation time.

```
// This page will be statically generated at build time
export const revalidate = false; // or 3600 (seconds)
```

- **Use Case:** Content that rarely changes (e.g., blog posts, documentation).

Server-Side Rendering (SSR)

- **Behavior:** Data is fetched on **every request** at runtime. Fresh data always.
- **How to:**
  1. **On-Demand (No Cache):**

     cache has two values:

     - `cache: 'force-cache'`: (Default for SSG) Next.js will look in its cache first. If a valid cached response exists and it's still up-to-date, it uses it. If not, it fetches from the network and caches the result.
     - `cache: 'no-store'`: (For SSR) Next.js skips the cache entirely, fetches from the network on every request, and does not save the response.

     ```
     fetch('https://api.com/data', { cache: 'no-store' });
     ```

     > There are additional methods, such as revalidatePath or revalidateTag, for on-demand validation, but we'll dive into those later.

  2. **Time-Based (Revalidate):**

     ```
     // Revalidate(update cache) fetch API
     fetch('https://api.com/data', { next: { revalidate: false | 0 | number } });
     // Or for the entire page/route:
     export const revalidate = false | 0 | number;
     ```

- **Use Case:** Highly dynamic data (e.g., user-specific dashboards, real-time analytics).

Incremental Static Regeneration (ISR)

- **Behavior:** Hybrid approach. Pages are statically generated but can be **revalidated** in the background after a specified time, ensuring content is never too stale.
- **How to:**

```
// This page is static but will revalidate every hour
export const revalidate = 3600; // 1 hour in seconds

async function Page() {
  const data = await fetch('https://api.com/products', { next: { revalidate:3600 } });
  // ...
}
```

- **Use Case:** Content that updates periodically but doesn't need to be real-time (e.g., product catalog, news feed).

`revalidate` Values

Choose your strategy per page/component:

- `false` / `Infinity`: Cache forever (Pure SSG). For static content.
- `0`: Never cache, fetch on every request (Pure SSR). For fully dynamic, real-time data.
- `number`: Revalidate after `number` seconds (ISR). For content that updates periodically.

Key Takeaways

- **Use RSCs and** `fetch` by default for simpler, more efficient server-side data fetching.

- Next.js extends the native `fetch` API to automatically handle caching and revalidation, making it the preferred choice over Axios for server-side data fetching.
- Control your app's behavior by configuring **caching and revalidation**.

Chapter 10 —> SEO and Metadata

What is SEO?

- **SEO (Search Engine Optimization)** is the process of improving your website's visibility in organic (non-paid) search engine results.
- It's like making your website the most attractive and easy-to-find "cat toy" in a room full of distractions for the "cat" (the search engine).

Best Practices to Improve SEO

- **Keywords:** Use relevant words/phrases that users search for.
- **Content Quality:** Create valuable, engaging content that keeps users on your site.
- **Meta Tags:** Provide a concise summary (title, description) for search engines.
- **Website Structure:** Organize your site logically with clear headings and URLs.
- **Site Speed:** Faster sites are ranked higher and provide a better user experience (a key Next.js strength).
- **Backlinks:** Get links from other reputable sites to build authority.
- **Clear URLs:** Use human-readable URLs (e.g., `/about`, not `/page?id=123`).

Next.js Metadata API

Next.js provides a powerful API to manage metadata, improving how your site appears in search results and on social media.

Static vs. Dynamic Metadata

- **Static Metadata:** Fixed information that doesn't change. This includes things like the page title, meta description, and meta keywords. Once set, these

elements remain the same unless intentionally updated by a website owner or developer.

- **Dynamic Metadata:** Information that changes based on the page or content (e.g., the title of a blog post). Generated for each page.
  For instance, the meta description might change depending on the specific search term a user uses or based on the content of the page.
  It's like a label that updates itself depending on what's inside the box or who's looking at it.

There are two ways through which we can add metadata to our website using Next.js's Metadata API:

1. **Config-Based Metadata**

   Export a `metadata` object from a `layout.js` or `page.js` file.

   Static Example:

```
// app/about/page.js
export const metadata = {
  title: 'About Us | My Website',
  description: 'Learn more about our company and mission.',
};

export default function AboutPage() {
  return <div>...</div>;
}
```

   Resulting HTML:

```
<head>
    <title>About Us | My Website</title>
    <meta name="description" content="Learn more about our company an
d mission.">
</head>
```

   Dynamic Example (for dynamic routes):

Use the `generateMetadata` function to fetch data and generate metadata.

```
// app/blog/[slug]/page.js
export async function generateMetadata({ params }) {
  // Fetch data for this specific blog post
  const post = await fetch(`https://api.com/posts/${params.slug}`)
                      .then(res => res.json());

    const seoDescription = "You can post Blogs in this.";

  return {
    title: `${post.title} | My Blog`,
    description: seoDescription,
       other: {
      "og:title": post.title,
      "og:description": seoDescription,
      "og:image": resource.image,
      "twitter:title": title,
      "twitter:description": seoDescription,
      "twitter:image": resource.image,
    },
  };
}

export default function BlogPost({ params, searchParams }) {
  // The SAME fetch call for the post data is AUTOMATICALLY DEDUPLICA
TED.
  // Next.js won't fetch it twice. It's cached.
  // ... render the post
};
```

Key Points:

- **Server Components Only:** The `metadata` object and `generateMetadata` function **only work in Server Components**. They cannot be used in Client Components ("use client").

- `generateMetadata` runs on the server and accepts `params` and `searchParams`.
- **Automatic Deduplication:** `fetch` requests inside `generateMetadata` and the Page component for the same URL are automatically memoized (cached). You are not making two network calls.
- So, these fetch requests are automatically memoized for the same data across generateMetadata, generateStaticParams, Layouts, Pages, and Server Components.
- **Social Media Tags:** Use `og/openGraph` (for Facebook, LinkedIn) and `twitter` meta tags to control how links look when shared on social media platform. Like the twitter one will allow you to control how your content is displayed when shared on Twitter.
- In simpler terms, when someone shares a link to your website (like a blog post) on social media platforms like **Facebook**, **Twitter**, or **LinkedIn**, those platforms **automatically try to create a preview** of that link. That preview usually includes:

  A **title,** A **description,** An **image**

2. **File-Based Metadata**

   Place specially named files in your `/app` directory. Next.js will automatically use them. This **overrides Config-Based metadata** for the same property**.**

   `app` folder -

   - `favicon.ico` - Website icon in browser tab.
   - `icon.png` / `apple-icon.png` - Icons for different devices.
   - `opengraph-image.png` - Image shown when link is shared on social media.
   - `twitter-image.png` - Image specifically for Twitter shares.
   - `opengraph-image.alt.txt` - Alt text for the Open Graph image.
   - `twitter-image.alt.txt`
   - `sitemap.xml` / `robots.txt` - Files to guide search engine crawlers.

Key Takeaways

- Use the **Metadata API** to control how your site looks in search results and on social media.
- For **static** data, use the `metadata` export.
- For **dynamic** data (e.g., blog posts, products), use `generateMetadata`.
- Use **file-based metadata** for images and special files. It's simple and has high priority.
- Next.js automatically **deduplicates** `fetch` calls between `generateMetadata` and the page component.

Chapter 11 —> Backend

The Next.js Backend Revolution

Next.js simplifies backend development by eliminating complex setup (like Express). You can create powerful APIs directly within your project using a **file-based routing system** similar to frontend pages. No separate server management is needed.

Creating API Routes

- **Location:** Place API rout es inside the `app/api/` directory.
- **The Special File:** Create a `route.js` (or `route.ts`) file inside a folder. The folder's name becomes the API endpoint.
- **HTTP Methods:** Export functions named after HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, etc.).

```
// File: app/api/hello/route.js

// GET: HTTP method handler / Route handler
export async function GET() {
  return Response.json({ message: 'Hello from Server!' });
}
```

- **API URL:** `/api/hello`

Nested & Dynamic Routes

- **Nested Routes:** Create subfolders. `app/api/users/route.js` → `/api/users`
- **Dynamic Routes:** Use square brackets `[paramName]` for dynamic segments.

```
// File: app/api/users/[userId]/route.js

// request: Contains incoming request data (headers, URL, body, etc.)
// { params }: Destructured object containing dynamic route parameters
export async function GET(request, { params }) {
  // Access the dynamic segment (e.g., 123 from /api/users/123)
  const userId = params.userId;
  return Response.json({ user: `User ${userId}` });
}
```

- **API URL:** `/api/users/123`
- **Key Point:** The dynamic segment's name in the folder (`[userId]`) must match the property name used to access it (`params.userId`).

Understanding API Parameters

- **Route Parameters:** Essential parts of the URL path (`/api/users/123`). Accessed via `params`.
- **Query Parameters:** Optional filters after `?` (`/api/products?category=electronics`). We define URL query parameters in Next.js by using the router or the Link component to include specific query parameters.

```
// 1. Using Next.js Router
router.push({
   pathname: '/products',
   query: { category: 'electronics', page: 1 },
});

// 2. Using Next.js Link
<Link
   href={{ pathname: '/products', query: { category: 'electronics', page: 1 }
```

```
}}
> Show Electronics Products </Link>

// 3. String Interpolation
// Both of the above can then be used with String Interpolation to achieve
// similar results of adding Query parameters to the URL
const category = 'electronics';
const page = 1;
router.push(`/products?category=${category}&page=${page}`);
<Link href={`/products?category=${category}&page=${page}`}>
   Show Electronics Products
</Link>
```

They are accessed via the request.

```
// request.nextUrl is a Next.js-specific helper that gives easy access to par
sed URL info like pathname, hostname, origin and searchParams etc. So,
we don't need to construct new URL() object.
export function GET(request) {
   const searchParams = request.nextUrl.searchParams;
   const query = searchParams.get('query');
}
```

Middleware

Middleware acts as a gatekeeper between incoming requests and your route handlers. It intercepts requests before they reach their destination, allowing you to perform operations like authentication, logging, or request modification.

Key Characteristics:

- **Single File Limitation:** Only one `middleware.js/ts` file is allowed at the project root
- **Runs on Every Request:** Executes for all routes unless configured otherwise
- **Can Short-Circuit Requests:** Can return responses directly without reaching routes

Basic Middleware Structure:

```
import { NextResponse } from 'next/server'

export function middleware(request) {
   console.log('do your stuff');
}

// other example in middleware.ts
import { NextResponse } from 'next/server';
import type { NextRequest } from 'next/server';

export function middleware(req: NextRequest) {
  // e.g., check auth
  return NextResponse.next();
}
```

Organizing Middleware Logic:

Despite the single file constraint, you can modularize your middleware logic in seperate files:

```
// lib/authMiddleware.js
import { NextResponse } from 'next/server'

export function authenticate(request) {
   console.log('Authentication middleware');
}
```

```
// lib/logMiddleware.js
import { NextResponse } from 'next/server'

export function logRequest(request) {
   console.log('Logging middleware');
}
```

Main Middleware File:

```
// middleware.js
import { NextResponse } from 'next/server'
import { authenticate } from './lib/authMiddleware'
import { logRequest } from './lib/logMiddleware'

export default function middleware(request) {
    authenticate(request, () => {
            logRequest(request);
    })
}
```

Middleware Chaining

Next.js doesn't natively support *chaining multiple middleware functions* like Express (`app.use(middleware1, middleware2)`), but you can **simulate chaining** using function composition or higher-order functions.

Caching, Revalidation & Runtimes

- **Cache:** `GET` API routes are **cached** by default for performance in Next.js.
- **Time-Based Revalidation:** Use `revalidate` to set a cache duration (in seconds).
- **Runtimes:** Default is Node.js.

```
// To disable default caching. fetch fresh data on every request
export const dynamic = 'force-dynamic';

// or we can revalidate API route
export const revalidate = 3600; // Revalidate data every hour

// change runtime
export const runtime = "edge";

export async function GET() { }
```

Examples

1. CRUD API Application in Next.js App Directory

With **Next.js App Router**, you can build a full-stack application. API routes can be placed in `app/api/`, and you can define handlers for different HTTP methods.

Folder structure: `app/api/products/route.ts`

```
import { NextRequest, NextResponse } from 'next/server';

let products = [{ id: 1, name: 'Laptop' }];

// GET - Retrieve all products
export async function GET() {
  return NextResponse.json(products);
}

// POST - Add a new product
export async function POST(req: NextRequest) {
  const data = await req.json();
  const newProduct = { id: Date.now(), ...data };
  products.push(newProduct);
  return NextResponse.json(newProduct, { status: 201 });
}
```

To handle specific product operations (Update, Delete), add a **dynamic route**:

Folder: `app/api/products/[id]/route.ts`

```
export async function PUT(req: NextRequest, { params }: { params: { id: string
} }) {
  const id = Number(params.id);
  const updated = await req.json();
  products = products.map(p => p.id === id ? { ...p, ...updated } : p);
  return NextResponse.json({ message: 'Product updated' });
}

export async function DELETE(req: NextRequest, { params }: { params: { id: st
ring } }) {
```

```
  const id = Number(params.id);
  products = products.filter(p => p.id !== id);
  return NextResponse.json({ message: 'Product deleted' });
}
```

2. Nested Dynamic Routes with Multiple Dynamic Segments in Next.js

You can nest dynamic segments using brackets (`[param]`), and combine them for advanced routing.

Example folder structure: `app/products/[category]/[productId]/page.tsx`

```
// Accessing Params in `page.tsx`:
import { useParams } from 'next/navigation';

export default function ProductPage({ params }: { params: { category: string;
productId: string } }) {
  return (
    <div>
      <h1>Category: {params.category}</h1>
      <h2>Product ID: {params.productId}</h2>
    </div>
  );
}
```

This route would match: `/products/electronics/12345`

3. Next.js vs Express.js

Feature	Express.js	Next.js (App Router)
Architecture	Minimalist backend-only	Full-stack (frontend + backend)
Routing	Imperative (`app.get(...)`)	File-system based routing
Middleware	Chainable via `app.use()`	Runs in `middleware.ts` with custom logic for chaining
API Endpoints	Declared in JS files	Placed in `app/api/` with HTTP method exports
Rendering	No rendering (API only)	Supports SSR, ISR, SSG

Feature	Express.js	Next.js (App Router)
Use case	APIs, microservices	Full-stack apps with UI and APIs
Customization	Highly customizable	Convention-over-configuration

4. Which to Choose for a Large-Scale E-Commerce Platform?

Considerations

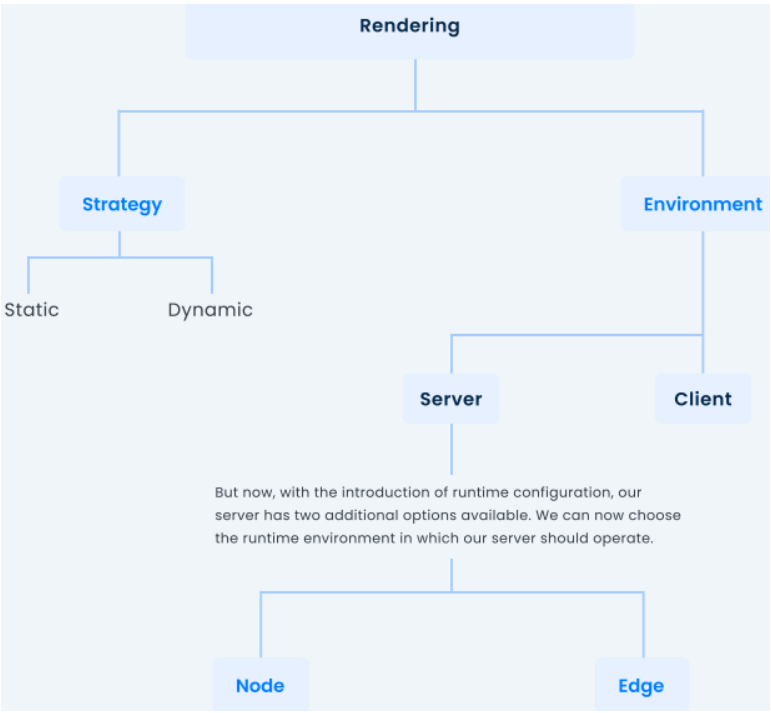
Criteria	Recommendation
Need for full-stack (UI + API)	✅ Next.js
SEO, SSR for product pages	✅ Next.js
Highly customized backend (queues, microservices, etc.)	✅ Express.js
Headless architecture with React frontend	✅ Express.js
Tight integration between frontend and backend	✅ Next.js
Heavy API workload (no frontend)	✅ Express.js

Key Takeaways

- **Simplicity:** Next.js backend development is incredibly straightforward with file-based routing.
- **Power:** It supports all essential backend features: dynamic routes, query params, middleware, caching, and multiple runtimes.
- **Integration:** Your backend and frontend live in the same project, simplifying development and deployment.

Chapter 12 —> Node vs Edge Runtime

By far, we learned different rendering strategies and environments in which we can render our application. A quick map of that would be something like this:



Edge Runtime

Think of Edge Runtime as having mini-computers distributed globally, close to where your users are. Instead of processing requests on a central server far away, it handles tasks on servers near the user's location. Edge doesn't support ISR.

Key Characteristics:

- **Proximity**: Runs code geographically closer to users
- **Speed**: Faster response times due to reduced distance
- **Lightweight**: Optimized for quick, simple operations

Comparison Table

Aspect	Node.js Runtime	Edge Runtime
Location	Centralized servers	Distributed globally
Latency	Higher (distance to server)	Lower (closer to users)
Execution Time	No hard timeout	25-30 second limit
Memory	Higher capacity	~4MB limit
APIs	Full Node.js API support	Limited Web APIs only

Aspect	Node.js Runtime	Edge Runtime
Cold Starts	Possible in serverless	Minimal to none

Edge Functions

Edge Functions are lightweight JavaScript functions that run on Vercel's edge network across 30+ regions worldwide, closest to your users.

Serverless Functions

Serverless Functions are Node.js functions that run on-demand in isolated environments, automatically scaling with traffic.

Edge Runtime Limitations

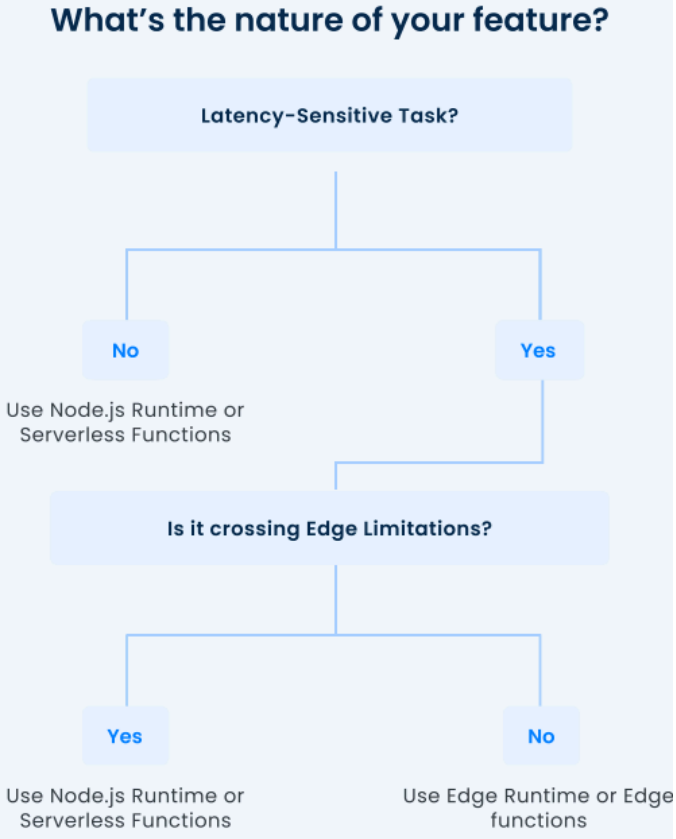
1. **Time Constraint**: Maximum 25-30 second execution time
2. **Size Limit**: 4MB package size (including dependencies)
3. **API Restrictions**: Limited to Web APIs (no full Node.js APIs)
4. **Network Dependency**: Performance depends on network conditions

Node.js Runtime Limitations

1. **Cold Starts**: Delay when functions start initially
2. **Resource Usage**: Higher memory/CPU consumption
3. **Latency**: Higher response times for distant users
4. **Scaling**: Can be slower to scale across regions

Practical Tips

- **Start with Node.js**: Default choice for most applications
- **Optimize with Edge**: Move performance-critical parts to Edge
- **Check Compatibility**: Verify required APIs work in Edge runtime
- **Monitor Performance**: Test both runtimes for your specific use case
- Here is a small mindmap. Ask yourself —



Real-World Use Cases

- **Edge**: Authentication/authorization checks, Geolocation services, Real-time notifications, API route optimization, Content personalization, A/B testing, Simple transformations
- **Node.js**: Database operations, File processing, Heavy computation tasks, Long-running processes, Complex business logic

Chapter 13 —> Server Actions

Server Actions are asynchronous JavaScript functions that execute **exclusively on the server**. They are a React/Next.js feature designed to perform data mutations and other server-side operations directly from your React components.

The "use server" Directive: This directive is the key. Placing `'use server'` at the top of a file or inside a function tells Next.js to treat that code as a server-side function.

> Analogy: Think of them like "API endpoints" that you define as simple functions, seamlessly integrated into your components, rather than as separate route handlers.

Why Do We Need Server Actions?

Purpose: They are not a full replacement for traditional API Routes but rather a powerful **alternative for mutations** (CREATE, UPDATE, DELETE operations). Their main goal is to **streamline backend operations** and drastically improve the Developer Experience (DX).

Key Advantages:

- **Simplified Architecture:** You can often avoid creating separate API route files (e.g., `pages/api/` or `app/api/`) for simple operations.
- **Full-Component Flexibility:** They can be used in **both Server and Client Components**, making them incredibly versatile.
- **Progressive Enhancement:** They can power HTML forms that work **even if JavaScript is disabled** in the user's browser.

How to Create & Use Server Actions

A. In Server Components

You can define the Server Action **inline** within your Server Component file.

Example: A form in a Server Component (`app/page.jsx` or `app/page.tsx`)

```
// This is a Server Component by default
export default function HomePage() {
  // Define the Server Action inline
  async function createItem(formData) {

    'use server'; // This makes it a Server Action

    // Extract data from the form
    const name = formData.get('name');
    // Perform server-side logic (e.g., database insert)
```

```
  await db.item.create({ data: { name } });
  // Revalidate a cache or redirect
  revalidatePath('/');
 }

 // Use the action in a plain HTML form
 return (
  <form action={createItem}>
   <input type="text" name="name" required />
   <button type="submit">Create Item</button>
  </form>);
}
```

B. In Client Components

You **cannot** define Server Actions inline in Client Components. You must define them in a **separate file** and then import them in Client or Server Component.

Step 1: Create a dedicated file for actions (`app/actions.js`)

```
'use server'; // Everything exported from this file is a Server Action

import { db } from '@/lib/db';
import { revalidatePath } from 'next/cache';
import { redirect } from 'next/navigation';

export async function createItem(formData) {
 const name = formData.get('name');
 await db.item.create({ data: { name } });
 revalidatePath('/');
 // redirect('/some-page'); // You could also redirect
}

export async function getItem(id) {
 // Yes, you can use Server Actions for data fetching too!
 const item = await db.item.findUnique({ where: { id } });
```

```
  return item;
}
```

Step 2: Import and use them in a Client Component (`app/client-form.jsx`)

```
'use client'; // This is a Client Component

import { createItem } from '@/app/actions'; // Import the Server Action
import { useActionState } from 'react'; // For state and errors

export default function ClientForm() {
  const [state, formAction, isPending] = useActionState(createItem, null);

  return (
    <form action={formAction}>
      <input type="text" name="name" required />
      <button type="submit" disabled={isPending}>
        {isPending ? 'Creating...' : 'Create Item'}
      </button>
      {state?.error && <p>Error: {state.error}</p>}
    </form>);
}
```

How Do They Work? (The Magic)

The mechanism differs based on where they are called from:

- **In Server Components:** The component and the action are both on the server. When the form is submitted, the action function is executed directly on the server, and the resulting HTML is streamed back to the client. No network fetch is needed in the traditional sense.
- **In Client Components:** Next.js automatically **transforms the function call into a secure POST request** to your server. This is why you don't need to manually set up an API route—Next.js handles the endpoint creation for you behind the scenes.

Enhanced UX with React Hooks

Server Actions work seamlessly with special React hooks to build modern, interactive forms.

`useFormStatus` **- For Pending States**

This hook provides status information about the **last form submission**. It must be used within a component that is a **child of a** `<form>` **element**.

`useActionState` **(formerly** `useFormState`**) - For Error Handling**

This hook allows you to **manage state based on the result** of a form action. It's perfect for handling errors returned by your Server Action.

Limitations

Server Actions are powerful but not a silver bullet for every use case.

- ❌ **Not for Cross-Platform Apps:** The endpoints are not exposed as public REST/GraphQL APIs. You cannot use them directly from a native mobile or desktop app.
- ❌ **Not for Real-Time/Persistent Connections:** They are request/response based. They cannot handle WebSockets, Server-Sent Events (SSE), or webhooks that require a persistently open connection.
- ❌ **Testing Challenges:** You cannot easily test them with external tools like **Postman** because the endpoint is hidden and managed by Next.js. Testing must be done through your application's UI or with internal testing utilities. On the positive side, it enhances security since your API isn't exposed.
- ❌ **HTTP Semantics:** They use POST requests for all operations (GET, POST, PUT, DELETE), which goes against standard HTTP method conventions.

When to Use Them: They shine in **monolithic Next.js applications** where the frontend and backend are tightly coupled, and the primary goal is development speed and simplicity.

Tailwind CSS

What is Tailwind CSS?

Tailwind CSS is not merely a styling framework, but as a **utility-first ideology** that accelerates and simplifies the creation of modern, responsive websites.

- **Utility-First Approach**: Unlike frameworks such as Bootstrap, Ant Design, or Material UI that come with predefined components, Tailwind CSS provides **small, reusable utility classes**. Developers combine these classes directly in their HTML to style elements exactly as desired, eliminating the need to write custom CSS from scratch.
- **Benefits**:
  - **Faster and Easier Development**: Speeds up the process of building modern, responsive websites.
  - **Complete Customization**: Offers full control over styling, allowing for unique designs rather than "cookie-cutter" looks.
  - **Clean and Maintainable Codebase**: Helps keep CSS lean and efficient by generating styles only for classes actually used. It avoids issues like worrying about class names, BEM methodology, or managing gigantic CSS files.
  - **Solves Common UI Struggles**: Addresses challenges with responsive layouts, custom styling, dark mode, theming, pseudo-classes, media queries, and dynamic utilities.
- **Distinction from Inline Styles**: A common misconception is that Tailwind classes are just inline styles. However, Tailwind utility classes are **reusable** and support advanced CSS properties like **pseudo-classes** and **media queries**, which inline styles do not.

Core Mechanics and Utility Classes

- **How it Works**: Every Tailwind class is a predefined CSS rule. When a class like `flex` is used, Tailwind generates the corresponding `display: flex;` property behind the scenes.
- **Basic Styling Examples**:
  - **Text Styling**: `text-center`, `text-lg`, `text-blue-400` for alignment, size, and color. `font-mono`, `font-extraBold` for font family and weight.
  - **Layout and Spacing**:

    - `bg-violet-200` for background color.
    - `h-10`, `w-full` for height and width (`h-10` is approx 2.5rem or 40 pixels, `w-full` is 100%).
    - `border-2`, `border-violet-600`, `rounded-md` for border width, color, and radius.
    - **Margin and Padding**: **Margin** is external spacing, pushing elements away from others. **Padding** is internal spacing, adding space inside an element.
    - `margin-top-2` (`mt-2`) for top margin.
    - `margin-y-4` (`my-4`) for vertical (top and bottom) margin.
    - `padding-2` (`p-2`) for padding on all sides.
    - Flexible notation: `m[top|right|bottom|left|y|x]-[size]` and `p[t|r|b|l|y|x]-[size]`.
- **Tailwind Play**: An interactive online playground for testing Tailwind CSS styles. It allows users to see the **generated CSS** for each utility class used, including default styles, theme variables, global styles, and specific utility rules.

Just-In-Time (JIT) Compiler

- **Generates Styles On Demand**: The JIT compiler is a core flexibility feature that generates styles only for the exact classes a project uses. This keeps the final CSS small and efficient.
- **Arbitrary Values**: The JIT compiler allows developers to use **any custom value** for properties that are not predefined in Tailwind's utility classes. This is done by enclosing the value in square brackets, e.g., `text-[13px]`, `bg-[#abcdef]`.
- **Benefits of JIT**:
    - **Optimizes Performance**: Generates only necessary styles.
    - **Faster Build Times**: No need to pre-compile thousands of classes.
    - **Supports Arbitrary Values**: Allows for unique customisation.
    - **Seamless Operation**: Works in both development and production environments.
    - **Built-in by Default**: Requires no extra setup.

Layouts: Flexbox and Grid

Tailwind provides powerful utilities for structuring layouts using positioning, display properties, Flexbox, and Grid.

Positioning: Defines where an element appears.

- `relative`: Moves an element relative to its normal position.
- `absolute`: Moves an element to its nearest parent.
- `fixed`: Sticks an element to the viewport, preventing scrolling.
- `sticky`: Behaves normally until scrolled to a certain point.

Display Properties: Determines an element's layout and visibility.

- `block`: Takes up the full width.
- `inline`: Behaves like text, without width or height.
- `flex`: Enables Flexbox.
- `grid`: Enables CSS Grid properties.

Flexbox: Essential for responsive layouts.

- **Enabling Flexbox**: `flex` class.
- **Alignment**: `justify-end` (aligns items to the right), `justify-around`, `justify-between`, `justify-center`, `justify-evenly`, `items-center`.
- **Spacing**: `space-x-6` (adds horizontal spacing between elements), `space-y-6` (adds vertical spacing).
- **Column Layout**: `flex-col` to stack elements vertically.
- **Learning Resource**: **Flexbox Froggy** is recommended for practicing Flexbox.

CSS Grid: Simplifies complex layouts.

- **Enabling Grid**: `grid` class.
- **Columns**: `grid-cols-3` (creates a three-column layout), `grid-cols-5`.
- **Gaps**: `gap-2` for spacing between grid items.
- **Learning Resource**: **Grid Garden** is recommended for practicing CSS Grid.

Responsive Design: Mobile-First Breakpoints

Tailwind CSS makes responsive design easy by using **mobile-first breakpoints**.

- **Predefined Device Widths**: `sm` (small), `md` (medium), `lg` (large), `xl` (extra large).
- **min-width (Default)**: By default, Tailwind uses `min-width` media queries, meaning styles are applied for a specific screen width **or larger**.

  ◦ **Mobile-First Methodology**: Un-prefixed utilities (e.g., `text-center`) apply to **all screen sizes**, especially mobile by default. Prefixed utilities (e.g., `md:block`) take effect at the specified breakpoint **and above**.

  ◦ **Example**: `md:block` means `display: block` will apply on medium devices (width >= 768px) and wider, overriding `hidden`. Larger screen sizes override smaller ones.
- **max-width (Optional)**: Can be used by adding a `max-` prefix (e.g., `max-sm`, `max-md`) to apply styles only if the width is **lower than** a specific size.
- **Custom Breakpoints**: Developers can fully customize breakpoints (e.g., `extra-small`, `tri-XL`) in their configuration.
- **Arbitrary Values for Breakpoints**: The JIT compiler allows using arbitrary values for screen widths, e.g., `min-[320px]`.
- **Key Takeaway**: Always design and develop mobile first, using un-prefixed utilities for mobile and then layering changes for larger screens with prefixed utilities.

Dark Mode and Theming

Tailwind CSS supports **dark mode natively**.

- **dark: Prefix**: To apply styles specifically for dark mode, simply prefix the utility class with `dark:` (e.g., `dark:bg-black`, `dark:text-white`).
- **System Preferences**: By default, Tailwind's dark mode adapts based on the user's operating system or browser preferences.
- **Manual Toggle**: To allow users to manually switch between light and dark modes:

  1. Toggling dark mode manually:
     Modify the Tailwind configuration by adding a custom variant in the CSS file: `@custom-variant dark` where `dark` turns on dark mode.
     Use JavaScript (or a framework like React) to toggle a `dark` class on the

`<html>` element, which Tailwind will then use to apply the `dark:` prefixed styles.

2. Using a data attribute in `@custom-variant dark` to automatically activate the theme.

- **Custom Themes**: Developers can introduce new colors or fonts not present in the default Tailwind theme.

Customization and Reusability

- **Inline Customization**: For small, unique tasks, arbitrary values can be used directly in classes within square brackets (e.g., `text-[pink]`, `bg-[#abcdef]`, `p-[16px]`, `text-[36px]`).
- **Tailwind Config (CSS V4)**: In Tailwind CSS v4, the configuration approach shifts from a JavaScript file to directly within CSS using **Tailwind CSS directives**.

  ◦ **Defining Custom Values**: Modify the `@theme` directive in the CSS file. For example, add a custom color `--color-chestnut: #value;` and then use `text-chestnut`.

  ◦ **Namespaces**: Colors are defined with `--color-`, fonts with `--font-`.

  ◦ **Customizable Properties**: Colors, fonts, text sizes, breakpoints, containers, blurs, animations, and more can be customized.
- Tailwind offers directives like `@import`, `@theme`, `@apply`, `@source`, `@utility`, `@variant`, `@custom-variant`, `@reference` and more... for base, components and utilities (they are parts of an approach to structuring and organizing styles using CSS configuration file method)

  Base applies styles to the entire application for your elements like H1, P tag should have a specific style across project.
  Components are used to style specific components or reusable UI elements like buttons, cards, etc.
  Utilities are atomic styles for individual properties like margins, padding, typography, colors and more...
- **Structuring Styles with Directives**: To enhance reusability and maintainability, especially for large applications, Tailwind offers directives:

- **@apply**: Inserts Tailwind CSS styles into your CSS, allowing you to use them as if they were regular CSS properties.
- **@layer**: Organizes styles into different layers.
  - **base**: Applies global styles to raw HTML elements (e.g., `H1`, `P`).
  - **components**: Styles reusable UI elements (e.g., cards, buttons, footers). This allows encapsulating multiple utility classes into a single custom component class.
  - **utilities**: Defines atomic styles for individual properties or common combinations (e.g., `FlexCenter` combining `flex justify-center items-center`).

```
/* allows using <div class="card"> instead of multiple utility classes */
@layer components {
  .card { @apply m-10 rounded-lg bg-white; }
}

/* allows using <div class="flex-center"> */
@layer utilities {
  .flex-center { @apply flex justify-center items-center; }
}
```

Tailwind CSS Tips and Tricks

- **Special Utilities**:
  - **Accent Colors**: `accent-[color]` to change the default browser accent color for elements like checkboxes and radio groups.
  - **Fluid Text**: Custom style approach `text-[min(10vw,70px)]` for text that scales smoothly with screen size, rather than just at breakpoints.
  - **File Input Styling**: The `file:` prefix simplifies styling native file inputs.
  - **Selection Highlights**: The `selection:` prefix allows overriding the default blue text highlight color, e.g., `selection:bg-green-500`.
  - **Carrot Color**: Customize the color of the caret like `caret-pink-500` (text input cursor).

- **More Advanced Utilities**: `before:`, `after:`, `landscape:`, `portrait:`, print styles, gradients, animations, and ARIA/screen reader specific styles.

- **Writing Less JavaScript**: Many tasks traditionally done with JavaScript (like accordions or state management) can often be handled directly with Tailwind CSS utilities and HTML features.
  - Use `open:` prefix selector with `<details>` and `<summary>` elements to create accordions without JavaScript.