NEXT JS - REACT FRAMEWORK

Next.js follows a **file-system-based routing** convention, where the file structure inside the `pages/` directory determines the application's routes. Here are the key conventions:

1. Basic Page Routing

Each file inside the `pages/` directory automatically becomes a route.
Example:

```
pages/
 ├── index.js        → "/"
 ├── about.js        → "/about"
 ├── contact.js      → "/contact"
```

- `index.js` in any folder maps to `/` for that folder.
- `about.js` maps to `/about`.

2. Nested Routes (Folders)

Folders inside `pages/` create nested routes.
Example:

```
pages/
 ├── blog/
 |    ├── index.js     → "/blog"
 |    ├── post.js      → "/blog/post"
```

3. Dynamic Routes (Parameterized URLs)

Files and folders with square brackets `[param]` define dynamic routes.
Example:

```
pages/
 ├── product/
 |    ├── [id].js     → "/product/:id"
```

- `/product/123` → `id = 123`
- `/product/xyz` → `id = xyz`

For multiple parameters:

```
pages/
 ├── blog/
 │    ├── [category]/
 |    |    ├──── [id].js   → "/blog/:category/:id"
```

- `/blog/tech/101` → `{ category: 'tech', id: '101' }`

4. Catch-All Routes (`[...param].js`)

For handling multiple segments dynamically:

```
pages/
 ├── docs/
 |    ├──── [...slug].js  → "/docs/*"
```

- `/docs/nextjs/routing` → `{ slug: ['nextjs', 'routing'] }`
- `/docs/setup` → `{ slug: ['setup'] }`

5. Optional Catch-All Routes (`[[...param]].js`)

Similar to catch-all but allows the route to be optional.

```
pages/
 ├── docs/
 |    ├──── [[...slug]].js  → "/docs" or "/docs/*"
```

- `/docs` → `{ slug: undefined }`
- `/docs/getting-started` → `{ slug: ['getting-started'] }`

6. API Routes (`pages/api/`)

Files inside `pages/api/` create API endpoints instead of pages.

```
pages/
 ├── api/
 |    ├──── hello.js    → "/api/hello"
 |    ├──── users.js    → "/api/users"
```

- `hello.js` exports a handler function (`req, res`) to process API requests.

7. Custom 404 and 500 Pages

Custom error pages can be defined as:

```
pages/
 ├──── 404.js   → Custom 404 page
 ├──── 500.js   → Custom 500 page
```

8. Middleware and Route Handlers (App Router)

For Next.js **App Router** (`app/` directory with React Server Components):

- **`app/page.js`** → Home route (`/`)
- **`app/about/page.js`** → `/about`
- **`app/blog/[id]/page.js`** → `/blog/:id`
- **Middleware (`middleware.js`)** for intercepting requests.

A **layout** is a shared UI component that wraps around multiple pages. It persists across route changes, preventing full re-renders.

10. Basic Layout Example

Create a `layout.js` file inside a folder to apply it to all pages in that route.

File Structure:

```
app/
 ├── layout.js       → Root layout (applies globally)
 ├── page.js         → Home page
 ├── about/
 │    ├── page.js    → "/about"
 │    ├── layout.js  → Layout for /about/*
 ├── contact/
 │    ├── page.js    → "/contact"
```

11. Global `layout.js` (applies to all pages)

```
// app/layout.js
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <header>Header Section</header>
        {children} {/* This renders the page content */}
        <footer>Footer Section</footer>
      </body>
    </html>
  );
}
```

12. Nested `layout.js` (for `/about` pages only)

```
// app/about/layout.js
export default function AboutLayout({ children }) {
  return (
    <div>
      <nav>About Page Navigation</nav>
      <main>{children}</main>
    </div>
  );
}
```

 Effect:

- `/about` pages will have their own navigation inside the `AboutLayout`.
- The `RootLayout` will still apply.

---

13. Nested Layouts (Multiple Levels)

You can create **nested layouts** for sections inside your app.

Example Structure:

```
app/
 ├── layout.js        → Root Layout
 ├── dashboard/
 │    ├── layout.js   → Dashboard-specific layout
 │    ├── page.js     → "/dashboard"
 │    ├── settings/
 │    │    ├── page.js  → "/dashboard/settings"
```

Dashboard Layout (`app/dashboard/layout.js`)

```
export default function DashboardLayout({ children }) {
  return (
    <div>
      <aside>Sidebar</aside>
      <section>{children}</section>
    </div>
  );
}
```

 Effect:

- `/dashboard` and `/dashboard/settings` will have a **sidebar**.
- They are still wrapped inside `RootLayout`.

---

14. Group Layouts (`(group)/` Folders)

Next.js allows you to organize routes without affecting the URL by using **parentheses** in folder names.

Example Structure:

```
app/
 ├── (marketing)/
 │    ├── layout.js   → Shared layout for marketing pages
```

```
│   ├── home/
│   │   ├── page.js → "/"
│   ├── about/
│   │   ├── page.js → "/about"
├── dashboard/
│   ├── layout.js  → Dashboard-specific layout
```

 Effect:

- The **(marketing)** group applies a shared layout to `home` and `about` without affecting the URL.
- `/dashboard` uses a separate layout.

Marketing Layout (`app/(marketing)/layout.js`)

```
export default function MarketingLayout({ children }) {
  return (
    <div>
      <header>Marketing Header</header>
      {children}
    </div>
  );
}
```

---

15. Routing Metadata (SEO & Accessibility)

You can define **metadata** for each route using the **metadata.js** convention.

Title & Metadata Example

```
// app/about/page.js
export const metadata = {
  title: "About Us | MyApp",
  description: "Learn more about our company.",
};

export default function AboutPage() {
  return <h1>About Us</h1>;
}
```

 Effect:

- The page will have `<title>About Us | MyApp</title>`.
- The `<meta name="description" content="Learn more about our company." />` will be in `<head>`.

Dynamic Metadata (based on params)

```
// app/product/[id]/page.js
export async function generateMetadata({ params }) {
  return {
    title: `Product ${params.id}`,
    description: `Details about product ${params.id}.`,
  };
}

export default function ProductPage({ params }) {
  return <h1>Product ID: {params.id}</h1>;
}
```

 Effect:

- `/product/123` → `<title>Product 123</title>`

Navigation in Next.js

Next.js provides various ways to handle **navigation** between pages, including:

1. **Component Navigation** using the `Link` component.
2. **Active Links** for highlighting the current route.
3. **Programmatic Navigation** using the `useRouter` hook.

---

16. Component Navigation (`next/link`)

In Next.js, we use the `next/link` component for client-side navigation instead of `<a>` tags to improve performance.

Basic Example

```
// app/components/Navbar.js
import Link from "next/link";

export default function Navbar() {
  return (
    <nav>
      <ul>
        <li><Link href="/">Home</Link></li>
        <li><Link href="/about">About</Link></li>
        <li><Link href="/contact">Contact</Link></li>
      </ul>
    </nav>
  );
}
```

✅**Why use `Link`?**

- **Pre-fetching:** Next.js **preloads** the linked page for fast navigation.
- **No full page reload:** The transition is **instant**.

---

17. Active Links (Highlight Current Page)

When a link matches the current route, we can style it differently.

□ Example: Styling Active Links

```
// app/components/Navbar.js
"use client"; // Required for usePathname()
import Link from "next/link";
import { usePathname } from "next/navigation";

export default function Navbar() {
  const pathname = usePathname(); // Get the current route

  return (
    <nav>
      <ul>
        <li>
          <Link href="/" className={pathname === "/" ? "active" : ""}>
            Home
          </Link>
        </li>
        <li>
```

```
        <Link href="/about" className={pathname === "/about" ? "active" : ""}>
          About
        </Link>
      </li>
    </ul>
  </nav>
 );
}
```

□ Adding CSS for Active Link

```
.active {
 font-weight: bold;
 color: red;
}
```

□ **Effect:** The current page link will be highlighted in **red**.

---

18. Navigating Programmatically (`useRouter`)

Instead of clicking a link, sometimes we need to **navigate programmatically** (e.g., after a form submission or button click).

□ Using `useRouter()` to Navigate

```
"use client";
import { useRouter } from "next/navigation";

export default function Dashboard() {
 const router = useRouter();

 return (
  <div>
   <h1>Dashboard</h1>
   <button onClick={() => router.push("/profile")}>Go to Profile</button>
  </div>
 );
}
```

□ **Effect:** Clicking the button navigates to `/profile`.

---

19. Other Navigation Methods (`useRouter()`)

The `router` object provides various navigation functions:

Method	Description
`router.push(url)`	Navigates to `url` (like clicking a link).
`router.replace(url)`	Navigates to `url` without adding to history.
`router.back()`	Goes back to the previous page.
`router.forward()`	Goes forward in history.
`router.refresh()`	Reloads the current page.

□ Example: Redirecting After Login

```
"use client";
import { useRouter } from "next/navigation";

export default function Login() {
  const router = useRouter();

  function handleLogin() {
    // Perform login logic...
    router.push("/dashboard"); // Redirect to dashboard after login
  }

  return <button onClick={handleLogin}>Login</button>;
}
```

□ **Effect:** After clicking **Login**, the user is redirected to `/dashboard`.

---

20. Loading UI (`loading.js`)

□ What is `loading.js`?

- It is a special file that **displays a loading indicator** while a page or layout is fetching data.
- Works **automatically** when using **React Suspense** (e.g., in server components).

□ Basic Example (`app/loading.js`)

```
export default function Loading() {
  return <p>Loading...</p>;
}
```

✅**Effect:**

- This component is displayed whenever a page is loading.
- Once the page is ready, it is replaced with the actual content.

□ Component-Level Loading (`loading.js` inside a folder)

You can create `loading.js` inside a specific **route folder** to show a loader for that route.

```
app/
├── layout.js
├── loading.js      # Global loading UI
├── dashboard/
│   ├── page.js
│   ├── loading.js  # Dashboard-specific loader
```

```
// app/dashboard/loading.js
export default function DashboardLoading() {
  return <p>Loading Dashboard...</p>;
}
```

✅**Effect:**

- This loading UI will only appear when the **dashboard page is loading**.

---

21. Templates (`template.js`)

🔹 What is a `template.js`?

- Unlike `layout.js`, **templates reset state on navigation**.
- Useful for components like **modals** or interactive sections that need a fresh state on each route change.

🔹 Example: `template.js` vs. `layout.js`

`layout.js` (State is Persisted)

```
// app/dashboard/layout.js

export default function DashboardLayout({ children }) {
  return (
    <div>
      <nav>Dashboard Menu</nav>
      {children}
    </div>
  );
}
```

✅**Effect:**

- When navigating inside the dashboard, state **remains the same**.

`template.js` (State Resets)

```
// app/dashboard/template.js

export default function DashboardTemplate({ children }) {
  return (
    <div>
      <nav>Dashboard Menu</nav>
      {children}
    </div>
  );
}
```

✅**Effect:**

- Each time a user navigates to `/dashboard`, the component **re-renders with a fresh state**.

---

22. Error Handling (`error.js`)

□ What is `error.js`?

- **Next.js automatically catches errors** in server components and displays an error UI.
- You can customize error messages using an `error.js` file.

□ Basic Error Page

```
// app/error.js

"use client"; // Required for error boundaries


import { useEffect } from "react";


export default function ErrorPage({ error, reset }) {
  useEffect(() => {
    console.error("Error occurred:", error);
  }, [error]);


  return (
    <div>
      <h1>Something went wrong!</h1>
      <p>{error.message}</p>
      <button onClick={() => reset()}>Try Again</button>
    </div>
  );
}
```

✅**Effect:**

- **Catches errors** in the app and displays a custom error message.
- Clicking **"Try Again"** resets the state and retries loading the page.

□ Component-Specific Error Handling

You can also create `error.js` inside a specific **route folder** to handle errors only for that route.

```
app/
 ├── error.js      # Global error UI
 ├── dashboard/
 │    ├── error.js  # Dashboard-specific error UI
```

```
// app/dashboard/error.js
"use client";

export default function DashboardError({ error, reset }) {
  return (
    <div>
      <h1>Dashboard Error!</h1>
      <p>{error.message}</p>
      <button onClick={() => reset()}>Reload Dashboard</button>
    </div>
  );
}
```

✅**Effect:**

- Errors **inside the dashboard** will show this error UI.
- The rest of the app **remains unaffected**.

---

24. Recovering from Errors in Next.js

When an error occurs, Next.js **automatically renders an error boundary** (`error.js`).
You can **recover from errors** using the **`reset`** function.

□ Example: Recovering from Errors

```
// app/error.js

"use client"; // Required for error boundaries


import { useEffect } from "react";


export default function ErrorPage({ error, reset }) {
  useEffect(() => {
    console.error("Error occurred:", error);
  }, [error]);


  return (
    <div>
      <h1>Something went wrong!</h1>
      <p>{error.message}</p>
      <button onClick={() => reset()}>Try Again</button>
    </div>
  );
}
```

✅**Effect:**

- The `reset()` function **retries the page** instead of showing an error screen.
- **Useful for API errors** or temporary failures.

---

25. Handling Errors in Nested Routes

Each nested route **can have its own `error.js` file**, so errors **don't affect the entire app**.

□ Example: Nested Error Handling

□ Project Structure

app/

├── error.js # Global error UI

├── dashboard/

│ ├── error.js # Dashboard-specific error UI

│ ├── analytics/

│ │ ├── error.js # Analytics-specific error UI

□ `app/dashboard/error.js` (Dashboard-Only Error Handling)

```
"use client";

export default function DashboardError({ error, reset }) {
  return (
    <div>
      <h1>Dashboard Error!</h1>
      <p>{error.message}</p>
      <button onClick={() => reset()}>Reload Dashboard</button>
    </div>
  );
}
```

✅**Effect:**

- If an error occurs **only in `/dashboard`**, the error UI **appears only in that section**.
- The rest of the app **remains functional**.

---

27. Handling Errors in Layouts

Layouts allow us to **catch errors at different levels** without breaking the entire app.

□ Example: `layout.js` with Error Handling

```
// app/layout.js

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <h1>My App</h1>
        {children}
      </body>
    </html>
  );
}
```

□ `app/dashboard/layout.js` (Error Handling for Dashboard Layout)

```
export default function DashboardLayout({ children }) {
  return (
    <div>
      <nav>Dashboard Menu</nav>
      {children}
```

```
    </div>
  );
}
```

✅**Effect:**

- If an error occurs in **any dashboard page**, it is caught in `dashboard/error.js`, and the layout remains intact.

---

28. Parallel Routes in Next.js

Parallel Routes allow **independent rendering** of multiple sections.

 Example: Loading Sidebar and Main Content Separately Project Structure

```
app/
 ├── layout.js
 ├── dashboard/
 │   ├── layout.js
 │   ├── sidebar.js
 │   ├── content.js
```

 Using Parallel Routes

```
// app/dashboard/layout.js
export default function DashboardLayout({ sidebar, content }) {
  return (
    <div style={{ display: "flex" }}>
      <aside>{sidebar}</aside>
      <main>{content}</main>
```

```
    </div>
  );
}
```

✅**Effect:**

- Sidebar and content **load separately**.
- If one fails, the other still **renders properly**.

---

29. Handling Unmatched Routes (404 Pages)

If a route **doesn't exist**, Next.js automatically shows a **404 page**.
To customize it, create `not-found.js`.

 Example: Custom `not-found.js`

```
// app/not-found.js

export default function NotFound() {
  return (
    <div>
      <h1>404 - Page Not Found</h1>
      <p>Oops! The page you're looking for does not exist.</p>
      <a href="/">Go Home</a>
    </div>
  );
}
```

✅**Effect:**

- This **custom 404 page** appears for **unmatched routes**.

---

30. Conditional Routes

Sometimes, we need to **conditionally render pages** based on authentication or user roles.

 Example: Redirecting Unauthorized Users

```
// app/dashboard/page.js

"use client";

import { useEffect } from "react";

import { useRouter } from "next/navigation";


export default function Dashboard() {

  const router = useRouter();

  const isAuthenticated = false; // Simulate user authentication


  useEffect(() => {

    if (!isAuthenticated) {

      router.replace("/login");

    }

  }, []);


  return <h1>Dashboard</h1>;

}
```

✅**Effect:**

- If the user **is not authenticated**, they are **redirected** to `/login`.

---

31. Intercepting Routes (`(..)`)

□ What is Intercepting Routing?

Intercepting routes **render a different page in place of another page without changing the URL**.

□ Example Folder Structure

```
app/
 ├── dashboard/
 │    ├── page.js   # Normal Dashboard Page
 │    ├── (..)modal/page.js  # Intercepting Route for Modal
```

□ Example: Intercepting Route (`(..)`)

```
// app/dashboard/(..)modal/page.js
export default function Modal() {
  return (
    <div className="modal">
      <h1>This is a modal</h1>
    </div>
  );
}
```

✅**Effect:**

- The **modal is rendered over the dashboard page** without changing the URL.

---

32. Parallel Intercepting Routes

□ What is Parallel Intercepting?

Parallel routes allow rendering multiple layouts **simultaneously**, while intercepting routes let you **replace a page temporarily**.

 Example Folder Structure

```
app/
 ├── layout.js
 ├── dashboard/
 │    ├── layout.js
 │    ├── sidebar.js
 │    ├── content.js
 │    ├── (..)modal/page.js  # Intercepted Modal
```

 Example: Parallel Route with Interception

```
// app/dashboard/layout.js
export default function DashboardLayout({ sidebar, content, modal }) {
  return (
    <div>
      <aside>{sidebar}</aside>
      <main>{content}</main>
      {modal} {/* This is the intercepted modal */}
    </div>
  );
}
```

✅**Effect:**

- **Both sidebar & content load separately**.
- **Modal is intercepted** and rendered **over content**.

---

33. Route Handlers (`route.js`)

What are Route Handlers?

- Next.js **handles API routes** in the **app router (`app/api`)**.
- Route handlers replace the **traditional `pages/api` approach**.

 Example Folder Structure

```
app/
```

```
├── api/
│   ├── users/
│   │   ├── route.js  # Handles API requests
```

---

34. Handling GET Requests

 Example: `GET` Route Handler

```
// app/api/users/route.js
export async function GET() {
  const users = [{ id: 1, name: "John" }, { id: 2, name: "Jane" }];
  return Response.json(users);
}
```

✅**Effect:**

- Visiting `/api/users` **returns a JSON response**:

```
[
  { "id": 1, "name": "John" },
  { "id": 2, "name": "Jane" }
]
```

---

35. Handling POST Requests

 Example: `POST` Route Handler

```
export async function POST(request) {
  const body = await request.json(); // Read request body
  return Response.json({ message: `User ${body.name} added` });
}
```

✅**Effect:**

- Sending a `POST` request with `{ "name": "Alex" }` **returns**:

{ "message": "User Alex added" }

---

36. Handling PATCH Requests

 Example: `PATCH` Route Handler

```
export async function PATCH(request) {
  const { id, name } = await request.json();
  return Response.json({ message: `User ${id} updated to ${name}` });
}
```

✅**Effect:**

- Sending `{ "id": 1, "name": "Mike" }` updates user **id 1 to Mike**.

---

37. Handling DELETE Requests

 Example: `DELETE` Route Handler

```
export async function DELETE(request) {
  const { id } = await request.json();
  return Response.json({ message: `User ${id} deleted` });
}
```

✅**Effect:**

- Deleting `{ "id": 2 }` **returns**:

{ "message": "User 2 deleted" }

---

38. Dynamic Route Handlers (`[id]`)

□ What is a Dynamic Route Handler?

- Used for **fetching specific data** like `/api/users/1`.

□ Example Folder Structure

```
app/
 ├── api/
 │   ├── users/
 │   │   ├── [id]/
 │   │   │   ├── route.js  # Dynamic API Route
```

□ Example: Dynamic `GET` Handler

```
export async function GET(request, { params }) {
  const { id } = params;
  return Response.json({ id, name: `User ${id}` });
}
```

✅**Effect:**

- Visiting `/api/users/3` returns:

```
{ "id": "3", "name": "User 3" }
```

---

39. URL Query Parameters

□ How to Access Query Params?

```
export async function GET(request) {
  const { searchParams } = new URL(request.url);
  const name = searchParams.get("name");
  return Response.json({ message: `Hello, ${name}` });
}
```

✅**Effect:**

- Visiting `/api/users?name=Alex` returns:

{ "message": "Hello, Alex" }

---

40. Redirects in Route Handlers

 Example: Redirecting Unauthorized Users

```
export async function GET() {
  return Response.redirect("/login");
}
```

✅**Effect:**

- Visiting `/api/protected` redirects users to `/login`.

---

41. Cookies in Route Handlers

Setting Cookies

```
import { cookies } from "next/headers";

export async function POST() {
  cookies().set("auth", "token123", { httpOnly: true });
  return Response.json({ message: "Cookie Set" });
}
```

Reading Cookies

```
export async function GET() {
  const auth = cookies().get("auth");
  return Response.json({ token: auth?.value });
}
```

✅**Effect:**

- **Stores & retrieves cookies securely.**

---

42. Caching in Route Handlers

□ How to Enable Caching?

```
export async function GET() {
  return new Response(JSON.stringify({ message: "Hello" }), {
    headers: { "Cache-Control": "s-maxage=60" }, // Cache for 60 seconds
  });
}
```

✅**Effect:**

- Caches API responses for **60 seconds**.

---

43. Middleware in Next.js

□ What is Middleware?

Middleware runs **before a request reaches a route** and can be used for: ✅Authentication

✅Redirects/Rewrites

✅Logging

✅Rate Limiting

□ Middleware File Structure

```
app/
 ├── middleware.js
```

□ Example: Middleware for Authentication

```
import { NextResponse } from "next/server";

export function middleware(request) {
  const token = request.cookies.get("auth");

  if (!token) {
    return NextResponse.redirect(new URL("/login", request.url));
  }
```

```
  return NextResponse.next(); // Continue to the requested page
}

export const config = { matcher: ["/dashboard/:path*"] }; // Apply to all dashboard pages
```

✅**Effect:**

- Blocks users without authentication from accessing `/dashboard`.

---

44. Rendering Strategies in Next.js

Next.js supports **multiple rendering methods**, each with different performance optimizations.

Rendering Strategy	Description	Use Case
CSR (Client-Side Rendering)	Loads a blank page first, then fetches data in the browser.	Single-page applications, dashboards.
SSR (Server-Side Rendering)	Fetches data on the server for every request.	Dynamic content like personalized feeds.
Static Rendering (SSG)	Pre-generates pages at build time.	Blogs, marketing pages, documentation.
Dynamic Rendering	Generates pages **only when needed**, based on user input.	Search results, filtered data.
Streaming	Loads and updates UI progressively.	Large pages with real-time updates.

---

45. Client-Side Rendering (CSR)

□ How does CSR work?

1 The browser loads a blank page.

2 JavaScript (React) runs to **fetch data from an API**.

3 The UI updates **after data is fetched**.

 Example: CSR using `useEffect`

```
"use client"; // This is a client component
import { useState, useEffect } from "react";

export default function CSRPage() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("/api/data")
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  return <div>{data ? data.message : "Loading..."}</div>;
}
```

✅**Effect:**

- The page loads **without data first**, then updates.

---

46. Server-Side Rendering (SSR)

How does SSR work?

1 The **server fetches data before sending HTML** to the browser.

2 The page loads **immediately with data**.

 Example: SSR using `getServerSideProps`

```
export async function getServerSideProps() {
  const res = await fetch("https://api.example.com/data");
  const data = await res.json();
```

```
  return { props: { data } };
}

export default function SSRPage({ data }) {
  return <div>{data.message}</div>;
}
```

✅**Effect:**

- The page **loads with data pre-rendered**.

---

47. Suspense for SSR

□ What is Suspense in SSR?

Suspense allows streaming SSR, meaning **part of the page loads while waiting for data**.

□ **Example: Streaming with Suspense**

```
import { Suspense } from "react";
import SlowComponent from "@/components/SlowComponent";

export default function Page() {
  return (
    <div>
      <h1>Fast Content</h1>
      <Suspense fallback={<p>Loading...</p>}>
        <SlowComponent />
      </Suspense>
    </div>
  );
}
```

✅**Effect:**

- The **page loads instantly**, but `SlowComponent` loads **after data is ready**.

---

48. React Server Components (RSC)

□ What are React Server Components?

RSC **runs only on the server**, reducing JavaScript sent to the browser.

□ Example: Server Component (default in `app/`)

```
export default async function ServerComponent() {
  const data = await fetch("https://api.example.com/data").then((res) =>
    res.json()
  );

  return <div>{data.message}</div>;
}
```

✅**Effect:**

- No client-side JavaScript required! □

---

49. Server and Client Components

Component Type	Where it Runs	Use Case
Server Components	Runs on the **server**, never in the browser.	Data fetching, heavy processing.
Client Components	Runs on the **client (browser)**.	User interactions, state management.

□ Example: Using Client and Server Components Together

```
// Server Component
import ClientComponent from "./ClientComponent";

export default function Page() {
  return (
    <div>
      <h1>Server Component</h1>
      <ClientComponent />
```

```
    </div>
  );
}

// Client Component
"use client"; // Marks it as a client component
import { useState } from "react";

export default function ClientComponent() {
  const [count, setCount] = useState(0);

  return <button onClick={() => setCount(count + 1)}>Click {count}</button>;
}
```

✅**Effect:**

- Server loads the page, but the button **handles clicks in the client**.

---

50. Static vs. Dynamic Rendering

Rendering Type	When it Runs	Use Case
Static Rendering	Pre-built at **build time**.	Blog posts, landing pages.
Dynamic Rendering	Generates pages **at request time**.	Dashboards, user profiles.

 Example: Static Rendering

```
export async function generateStaticParams() {
  return [{ id: "1" }, { id: "2" }];
}
```

 Example: Dynamic Rendering

```
export async function GET(request) {
  const id = request.nextUrl.searchParams.get("id");
  return Response.json({ message: `User ID: ${id}` });
}
```

---

51. Streaming in Next.js

□ What is Streaming?

Streaming **progressively loads page content** instead of waiting for everything to be ready.

□ Example: Streaming a Component

```
export async function GET() {
  return new Response("Loading content...", {
    status: 200,
    headers: { "Content-Type": "text/html", "Transfer-Encoding": "chunked" },
  });
}
```

✅**Effect:**

- Users **see content appear in chunks** instead of waiting for the whole page.

---

52. Server-Only Code & Third-Party Packages

□ Server-Only Code

```
import fs from "fs"; // Only available on the server

export default function ServerOnly() {
  const data = fs.readFileSync("file.txt", "utf-8");
  return <div>{data}</div>;
}
```

□ Third-Party Packages in Next.js

You can install **server-only** or **client-only** packages:

```
npm install axios moment
```

---

53. Context Providers & Client Component Placement

 Example: Context Provider

```
"use client";
import { createContext, useContext } from "react";

const ThemeContext = createContext("light");

export function ThemeProvider({ children }) {
  return <ThemeContext.Provider value="dark">{children}</ThemeContext.Provider>;
}

export function useTheme() {
  return useContext(ThemeContext);
}
```

✅**Effect:**

- The **context provider manages global state**.

---

54. Interleaving Server and Client Components

 What is Interleaving?
Next.js allows **mixing server and client components** within a page, enabling **optimal data fetching and UI interactivity.**

 Example: Server Component with a Client Component Inside

```
// Server Component (fetches data)
import ClientComponent from "./ClientComponent";

export default async function ServerComponent() {
  const data = await fetch("https://api.example.com/data").then((res) =>
```

```
    res.json()
  );

  return (
    <div>
      <h1>Server Component Data: {data.message}</h1>
      <ClientComponent />
    </div>
  );
}

// Client Component (handles user interactions)
"use client";
import { useState } from "react";

export default function ClientComponent() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

✅**Effect:**

- The **server fetches data before rendering**.
- The **button handles state changes on the client** without server interaction.

---

55. Data Fetching in Next.js

Next.js supports **various data-fetching strategies** based on the rendering method.

Fetching Method	Where it Runs	Best For
`fetch()` inside Server Component	**Server**	Pre-rendered content
`useEffect` inside Client Component	**Client**	Dynamic user interactions
API Routes (`app/api/`)	**Server**	Custom APIs

 Example: Fetching Data in a Server Component

```
export default async function ServerComponent() {
  const res = await fetch("https://api.example.com/data");
  const data = await res.json();

  return <div>{data.message}</div>;
}
```

✅**Effect:**

- Data **fetching happens on the server**, reducing client load.

---

56. Fetching Data with Server Components

 Why Use Server Components for Fetching?

- **No JavaScript sent to the client** (better performance).
- **SEO-friendly** (data is pre-rendered).
- **Improved security** (data stays on the server).

 Example: Fetching and Passing Data to a Client Component

```
import ClientComponent from "./ClientComponent";

export default async function ServerComponent() {
  const data = await fetch("https://api.example.com/data").then((res) =>
    res.json()
  );

  return <ClientComponent data={data} />;
}

// Client Component (renders data)
"use client";
export default function ClientComponent({ data }) {
  return <div>Client Receives: {data.message}</div>;
}
```

✅**Effect:**

- The **server fetches data and sends it to the client**.

---

57. Loading and Error States

 How to Handle Loading & Errors in Next.js?

- **Use Suspense for loading states** in Server Components.
- **Use `error.js`** files to catch errors in Server Components.
- **Handle errors manually** in API calls.

 Example: Handling Loading and Errors with Suspense

```
import { Suspense } from "react";
import FetchData from "./FetchData";

export default function Page() {
  return (
    <Suspense fallback={<p>Loading...</p>}>
      <FetchData />
    </Suspense>
  );
}
```

✅**Effect:**

- The page **shows "Loading..." until data is fetched**.

 Example: Handling Errors with `error.js`

```
// error.js
export default function ErrorComponent({ error }) {
  return <p>Error: {error.message}</p>;
}
```

✅**Effect:**

- If a server component fails, **Next.js automatically renders `error.js`**.

---

58. JSON Server Setup for Mock Data

 Why Use JSON Server?

- Simulates a **REST API** for local development.
- Helps **test data fetching** without an actual backend.

 Step 1: Install JSON Server

npm install -g json-server

 Step 2: Create `db.json`

```
{
  "users": [
    { "id": 1, "name": "Alice" },
    { "id": 2, "name": "Bob" }
  ]
}
```

 Step 3: Start JSON Server

json-server --watch db.json --port 3001

✅**Effect:**

- JSON Server runs at `http://localhost:3001/users`.

 Example: Fetching from JSON Server

```
export default async function FetchUsers() {
  const users = await fetch("http://localhost:3001/users").then((res) =>
    res.json()
  );
  return <pre>{JSON.stringify(users, null, 2)}</pre>;
}
```

---

59. Caching Data & Opting Out

 How Does Next.js Cache API Requests?

- **By default, data is cached indefinitely**.
- You can **disable caching** if data should always be fresh.

 Example: Opting Out of Caching

```
export async function GET() {
  const response = await fetch("https://api.example.com/data", {
    cache: "no-store",
  });
  return Response.json(await response.json());
}
```

✅**Effect:**

- The API **fetches fresh data every request**.

---

60. Request Memoization in Next.js

 What is Memoization?

- Prevents **multiple fetch calls for the same request**.
- Reduces **unnecessary API requests**.

 Example: Memoized Fetching

```
import { unstable_cache } from "next/cache";

const getData = unstable_cache(async () => {
  const res = await fetch("https://api.example.com/data");
  return res.json();
}, ["data-key"]);
```

```
export default async function Page() {
  const data = await getData();
  return <div>{data.message}</div>;
}
```

✅**Effect:**

- **Reduces API calls** by caching the response.

---

61. Time-Based Data Revalidation

 Revalidate Data at Regular Intervals
Next.js can **fetch new data automatically** after a set time.

 Example: Revalidating Data Every 60 Seconds

```
export async function GET() {
  const response = await fetch("https://api.example.com/data", {
    next: { revalidate: 60 },
  });
  return Response.json(await response.json());
}
```

✅**Effect:**

- Data is **revalidated every 60 seconds**.

---

62. Client-Side Data Fetching

 When to Fetch Data on the Client?

- **User-triggered actions** (e.g., search, filters).
- **Real-time updates** (e.g., live notifications).

 Example: Client-Side Fetching with `useEffect`

```
"use client";
import { useState, useEffect } from "react";

export default function ClientFetching() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("/api/data")
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  return <div>{data ? data.message : "Loading..."}</div>;
}
```

✅**Effect:**

- **Data is fetched dynamically on the client**.
