**Adding New Routes in Next.js**

* In **Next.js**, routing is **file-based**. You create folders inside the app/ directory, and each folder represents a route.
* Inside each route folder, you must create a page.js file. This file contains a **React component** that is rendered for that route.

**Steps to Create Routes**

1. **Create a new folder in app/**
   * Example: /cabins, /about, /account
2. **Inside each folder, create a page.js file**

jsx

CopyEdit

export default function Page() {

return <h1>This is the Cabins page</h1>;

}

* + This component is **server-side rendered by default** in Next.js.

1. **Nested Routes**
   * You can create deeper routes by adding subfolders.
   * Example: /cabins/test
     + Create a cabins/test/ folder
     + Add a page.js file inside it.

**Customizing VS Code for Better Navigation**

* Since all route files are named page.js, VS Code can display them with custom labels.
* You can set **folder-based labels** to differentiate between multiple page.js files.

**Next.js Navigation and Link Component**

* **Navigation in Next.js**
  + Allows users to move between different pages efficiently.
  + A regular <a> **tag can be used but causes a full page reload, leading to performance issues**.
* **Using <Link> Component**
  + Provided by Next.js for **client-side navigation** **without full page reloads.**
  + Imported using:

js

CopyEdit

**import Link from 'next/link';**

* + Uses href instead of to (unlike React Router).
  + Example:

js

CopyEdit

<Link href="/cabins">Explore Luxury Cabins</Link>

* + Provides a **Single Page Application (SPA) feel**, even with server-rendered pages.
* **Optimizations with <Link>**
  + **Prefetching:** **Pages linked on a page are preloaded in production**.
  + **Code Splitting:** **Each page is downloaded as a separate chunk, improving performance.**
  + **Caching:** Previously visited pages are **stored in the browser**, reducing reloads.
* **Creating a Reusable Navigation Component**
  + Stored in components/navigation.js to organize the project.
  + Example Navigation Component:

js

CopyEdit

import Link from 'next/link';

export default function Navigation() {

return (

<ul>

<li><Link href="/">Home</Link></li>

<li><Link href="/cabins">Cabins</Link></li>

<li><Link href="/about">About</Link></li>

<li><Link href="/account">Your Account</Link></li>

</ul>

);

}

* + Imported and used in page.js for **consistent navigation** across pages.
* **Project Folder Structure Consideration**
  + Placing components in /app/components **automatically creates a new route**, unless structured properly.
  + **Solution:** Improve project architecture later to prevent unwanted routing.
* **Reusable Layouts in Next.js**
  + Instead of adding the navigation manually to each page, a **layout component** can be used.
  + **Next topic:** Implementing layouts for a structured and reusable UI.

**Global Layout in Next.js**

* **Global Layout in Next.js**
  + Every Next.js app has a **root layout** (layout.js), **which wraps the entire application(page.js)**.
  + Next.js enforces the presence of layout.js by regenerating it if deleted.
* **Creating the Root Layout**
  + The **root layout should export a component named** **RootLayout** (or any other name, but conventionally this).
  + It must include <html> and <body> tags.
  + Example:

jsx

CopyEdit

export default function RootLayout({ children }) {

return (

<html lang="en">

<body>

{/\* Navigation Component \*/}

{children} {/\* Dynamic page content \*/}

</body>

</html>

);

}

* **The children Prop**
  + Essential for rendering page-specific content within the layout.
  + Works similarly to the **React Router Outlet**.
  + Every page's content replaces children dynamically when navigating.
* **Navigation and Shared UI Components**
  + The navigation bar is placed inside layout.js to persist across all pages.
  + Additional global elements (e.g., footer, logo) are added inside the layout.
* **Metadata in Next.js**
  + Instead of manually defining <head> content, Next.js allows exporting **metadata**.
  + Example:

jsx

CopyEdit

export const **metadata** = {

title: "The Wild Oasis",

};

* + The page title is automatically updated without directly modifying <head>.
* **Next.js Conventions**
  + Special filenames like layout.js and page.js define the structure.
  + Routing is **folder-based**—creating a new folder automatically generates a route.
  + Static assets (e.g., images) are placed inside the **public** folder and referenced directly (/icon.png).
* **Adding a Logo**
  + Used a prebuilt **Logo** component.
  + Imported the logo image from public/ folder.
  + Next.js recommends using its built-in **Image component** for optimization.

**Advanced Notes: Data Fetching with React Server Components (RSC)**

**Key Concepts**

1. **Pages in Next.js**:
   * By default, pages in Next.js are **server components**.
   * This is the default behavior in the **RSC model**.
2. **Data Fetching in Server Components**:
   * Server components can **fetch data directly** using async/await.
   * This is a **new capability** in React, as traditional React components cannot be async.

**Practical Example: Fetching Data in a Server Component**

1. **Using a Dummy API**:
   * Example: JSONPlaceholder (e.g., https://jsonplaceholder.typicode.com/users).
   * Fetch data directly in the component using the fetch function.
   * Example code:

javascript

Copy

async function CabinsPage() {

const res = await fetch('https://jsonplaceholder.typicode.com/users');

const data = await res.json();

console.log(data); // Logs data to the server terminal

return (

<ul>

{data.map(user => (

<li key={user.id}>{user.name}</li>

))}

</ul>

);

}

1. **Server-Side Logging**:
   * Logs from server components appear in the **terminal**, not the browser console.
   * Confirms that the component is running on the server.
2. **Rendering Data**:
   * Data is rendered directly into the HTML on the server.
   * Example: Rendering a list of user names.
   * **View Page Source**: Confirms that data is pre-rendered in the HTML.
3. **Caching**:
   * Data is cached by Next.js after the first fetch.
   * Subsequent navigations to the page use the cached data, improving performance.

**Adding Interactivity with Client Components and Server-Client Data Flow**

**Key Concepts**

1. **Server Components**:
   * Cannot use **React hooks** (e.g., useState, useEffect).
   * Cannot import or render client components without the use client directive.
2. **Client Components**:
   * Handle interactivity (e.g., buttons, toggles).
   * Require the use client directive to mark them as client-side components.
3. **Server-Client Boundary**:
   * Data can be passed from server components to client components via **props**.
   * Props must be **serializable** (no functions or classes).

**Practical Example: Building a Counter (Client Component)**

1. **Creating a Counter**:
   * Example code:

javascript

Copy

'use client'; // Marks this as a client component

import { useState } from 'react';

export default function Counter() {

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

return (

<button onClick={() => setCount(count + 1)}>

Current count: {count}

</button>

);

}

* + **Interactivity**: The button updates the count on click.

1. **Hydration**:
   * On slow networks, the **static HTML** is loaded first.
   * Once the React bundle is downloaded, the page is **hydrated**, adding interactivity.
   * Users see content immediately, even before interactivity is enabled.

**Crossing the Server-Client Boundary with Data**

1. **Passing Data from Server to Client**:
   * Fetch data in a **server component**.
   * Pass the data as **props** to a **client component**.
   * Example:

javascript

Copy

// Server Component (CabinsPage.js)

async function CabinsPage() {

const res = await fetch('https://jsonplaceholder.typicode.com/users');

const users = await res.json();

return <Counter users={users} />;

}

// Client Component (Counter.js)

'use client';

export default function Counter({ users }) {

console.log(users); // Logs data in the browser console

return (

<div>

<button onClick={() => setCount(count + 1)}>

Current count: {count}

</button>

<p>There are {users.length} users.</p>

</div>

);

}

1. **Initial Render**:
   * On the **initial render**, both server and client components are rendered on the server.
   * The rendered HTML is sent to the client, allowing users to see content immediately.
   * Once the React bundle is downloaded, the client components are **hydrated**, enabling interactivity.

**Notes on Loading Indicators in Next.js**

* **Problem:** Pages with data fetching have a slight delay before rendering, leading to a poor user experience.
* **Solution:** Use a **loading indicator** (e.g., a spinner or text) to show users that data is being loaded.

**Using loading.js in Next.js**

* Next.js provides a built-in convention for loading states using a **loading.js** file.
* This file should be created in the **app** folder at the root level.
* The **loading.js** file applies globally to all pages, even deeply nested routes (e.g., /cabins/test/23).

**Implementing loading.js**

* Create a new file: **loading.js**
* Define a React component:

javascript

CopyEdit

export default function Loading() {

return <p>LOADING DATA</p>;

}

* This will show the text **"LOADING DATA"** while the page content loads.

**How loading.js Works**

* **Instant Loading State:** The loading message is rendered on the server immediately.
* **Content Streaming:** The actual page content is streamed from the server to the client **gradually** instead of all at once.
* **Next.js Mechanism:** Uses **renderToReadableStream** instead of **renderToString** (which React normally uses).
* **Progressive Hydration:** Parts of the layout (e.g., navbar, footer) load first, while the content takes a moment to appear.

**JavaScript Requirement & Limitations**

* **JavaScript must be enabled** for streaming to work.
* If JavaScript is disabled, streaming won’t function, and **loading.js should not be used** in such cases.

**Granular Loading Control with Suspense**

* The **loading.js** file applies to **entire pages**.
* If only certain components need a loading state, **Suspense** can be used for finer control.
* Example use case: If a page has **20 components** and only one fetches data, Suspense can target just that component instead of replacing the whole page.

**Conclusion**

* loading.js provides an **easy** and **built-in** way to handle loading states globally.
* Works for **all sub-routes** automatically.
* For **more control**, use Suspense instead of loading.js.