* **Installing Supabase Library**
  + Install the Supabase package:

bash

CopyEdit

**Step 1**:npm install @supabase/supabase-js

**Step2:**Create a new file supabase.js inside the **lib** folder.

* + Use createClient from Supabase to establish a connection.
* **Using Environment Variables**
  + Store **Supabase URL** and **SUPABASE\_KEY** in environment variables for security.
  + Create a .env.local file and define:

env

CopyEdit

SUPABASE\_URL=<your\_supabase\_url>

SUPABASE\_KEY=<your\_supabase\_key>

* + Next.js supports environment variables, keeping them secure in the server environment.
  + **Public vs. Private Variables:**
    - Variables prefixed with NEXT\_PUBLIC\_ are exposed to the client.
    - Secret keys should **not** have this prefix to prevent security risks.
* **Row-Level Security (RLS) and Service Role Key**
  + Supabase enforces **Row-Level Security (RLS**), **restricting access to certain data.**
  + In the previous project, only **authenticated users** could access cabins, bookings, etc.
  + For this project, cabin information needs to be **public** (viewable without authentication).
  + Instead of the **public key**, use the **service role key** to bypass RLS.
  + Store the **service role key** securely in .env.local:

env

CopyEdit

SUPABASE\_KEY=<your\_service\_role\_key>

* + **Warning:** This key grants **full access** to all data, so it must remain **server-side only**.
* **Finalizing Supabase Setup**
  + Import the **Supabase client** in the data service file.
  + Pre-written functions for fetching, creating, and updating data can now be used.
  + Supabase is now fully set up, allowing data operations in the app.

[End of Notes]

Code:

**1. Install Supabase**

Run this command in your project folder to install the Supabase client:

sh

CopyEdit

npm install @supabase/supabase-js

**2. Create supabase.js inside the lib folder**

This file initializes the Supabase client.

**lib/supabase.js**

javascript

CopyEdit

import { createClient } from "@supabase/supabase-js";

const supabaseUrl = process.env.NEXT\_PUBLIC\_SUPABASE\_URL;

const supabaseKey = process.env.SUPABASE\_SERVICE\_ROLE\_KEY;

export const supabase = createClient(supabaseUrl, supabaseKey);

**Note:**

* NEXT\_PUBLIC\_SUPABASE\_URL is public and safe to expose.
* SUPABASE\_SERVICE\_ROLE\_KEY is **secret** and should never be exposed to the client.

**3. Set Up Environment Variables**

Create a .env.local file at the root of your project and add the following:

sh

CopyEdit

NEXT\_PUBLIC\_SUPABASE\_URL=your\_supabase\_url

SUPABASE\_SERVICE\_ROLE\_KEY=your\_service\_role\_key

Replace your\_supabase\_url and your\_service\_role\_key with actual values from **Supabase Dashboard → Project Settings → API**.

**4. Fetch Data from Supabase**

Now, use the Supabase client to fetch data from the database.

**lib/api.js (Example Data Fetching File)**

javascript

CopyEdit

import { supabase } from "./supabase";

// Fetch all cabins

export async function getCabins() {

const { data, error } = await supabase.from("cabins").select("\*");

if (error) throw new Error(error.message);

return data;

}

// Fetch all bookings

export async function getBookings() {

const { data, error } = await supabase.from("bookings").select("\*");

if (error) throw new Error(error.message);

return data;

}

**5. Use API in Components**

Now, use the API in your React components to display data.

**Example: components/CabinsList.js**

javascript

CopyEdit

import { useEffect, useState } from "react";

import { getCabins } from "../lib/api";

export default function CabinsList() {

const [cabins, setCabins] = useState([]);

useEffect(() => {

async function fetchData() {

try {

const data = await getCabins();

setCabins(data);

} catch (error) {

console.error("Error fetching cabins:", error);

}

}

fetchData();

}, []);

return (

<div>

<h2>Cabins</h2>

<ul>

{cabins.map((cabin) => (

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

))}

</ul>

</div>

);

}

**Notes on Implementing a Cabin Listing Page in Next.js with Supabase**

* **Fetching Cabins from Supabase:**
  + Use an **async function** in a **server component** to fetch data.
  + Call getCabins() to retrieve cabins, including **ID, name, max capacity, and images**.
  + Data is fetched **on the server**, eliminating the need for useEffect or state management.
* **Rendering Cabins in a Server Component:**
  + Convert the component into an **async function**.
  + Use await getCabins() to fetch data before rendering.
  + React waits for data retrieval before sending **fully-rendered HTML** to the browser.
* **Server-Side Benefits:**
  + Close to the **data source**, reducing API calls.
  + No need for a client-side fetching library.
  + Works well with **external services** like Supabase

Notes on Loading indicators

**Global vs. Route-Specific Loading Indicators**

* Global **loading.js** provides a default loading indicator for all routes.
* A specific **loading.js** file can be created inside the cabins route to override the global one for that route and its subroutes.

 **Implementing a Custom Loader**

* The custom loader uses a **spinner** from starter files (both **Spinner** and **SpinnerMini** are available).
* The spinner is rendered immediately in the cabins route.
* Additional text ("loading cabin data") is added below the spinner with custom styling (grid container with **items-center** and **justify-center** for centering).

 **Activation and Behavior of Streaming**

* Including a **loading.js** file **activates streaming**.
* Streaming **requires JavaScript to be enabled in the browser**.
* Disabling JavaScript (via browser dev tools) will stop the dynamic behavior, causing the page to only display the initial loading state.

 **Granular Control Over Content Loading**

* Only the data fetched from the Supabase API is delayed; static content (like headings and text) could be rendered immediately.
* The current implementation shows a loading spinner for the entire cabins page, not just for the data fetching part.
* **Achieving more granular control over which parts display a loading indicator requires** using **suspense**, a React feature that will be explored further.
* **Suspense Overview**
  + **A built-in React component that we can use to catch components that are not**

Ready to be rendered due async work and display a fall back UI

**Causes for Suspension**

* + **Data fetching** using libraries that support Suspense (e.g., React Query, Next.js).
  + **Lazy loading** additional code with React's **lazy** feature.
* **Implementation in a Component Tree**
  + Wrap the asynchronous component or sub tree (e.g., a **products** component in an e-commerce app) with a **Suspense** boundary.
  + The boundary displays a **fallback** (commonly a loading spinner) while the asynchronous work is in progress.
  + Eliminates the need for manual **isLoading** states and conditional rendering.

 **Internal Mechanics and the Role of Activity**

* When a component suspends (by throwing a **Promise**), React:
  + Traverses up to the nearest **Suspense boundary**.
  + Moves the suspending sub tree into an internal component called **activity** within the fiber tree.
* **Activity Component:**
  + **Activity** is a built-in internal component (not visible in the component tree via React DevTools).
  + Manages the suspended sub tree by storing it, preserving its state while it’s hidden.
  + The **fallback** element is placed as a sibling of the activity.
  + Initially, activity's mode is set to **visible** (showing children). When suspension occurs, it switches to **hidden**, causing the fallback to render.
  + Once asynchronous work completes, the mode resets to **visible**, and the final sub tree is rendered.
  + This process preserves state because the suspended elements remain in the fiber tree instead of being destroyed.

 **Handling Suspense with Transitions**

* During transitions (e.g., Next.js page navigations), the fallback may not render if the data-fetching promise is wrapped in a **startTransition**.
* Using a unique **key prop** can force the Suspense boundary to reset and display the fallback as needed.

**Notes on Using Suspense for Streaming UI in Next.js**

* **Suspense** in React allows **streaming individual UI parts** instead of blocking the entire page.
* The **current issue**: A loading.js file is blocking the entire UI, though only the **cabin list** depends on data.

**Wrap CabinList in a Suspense Boundary**

* Import Suspense from React.
* Wrap <CabinList /> inside <Suspense>.
* Use a **fallback** (e.g., <Spinner />) to show a loading indicator while fetching data.

 **Creating Dynamic Routes:**

* Instead of predefining routes for each cabin, you're using a folder with **square brackets**, e.g., [cabinId], inside the cabins directory.
* This allows for **dynamic URL segments**, such as /cabins/91 or /cabins/4.

 **Fetching Cabin Data Dynamically:**

* You're using the params prop inside the [cabinId]/page.js file.
* Extracting params.cabinId lets you fetch the correct data using the getCabin(cabinId) function.

implementing **dynamic metadata generation in Next.js** for individual cabin pages. Here's a refined version of how you can achieve this:

**Steps to Fix the Issue:**

1. **Use generateMetadata instead of a static metadata export.**
2. **Fetch cabin details dynamically using params.**
3. **Ensure the metadata includes the specific cabin name.**
4. **Handle any potential errors gracefully.**

**Updated Code for CabinPage.js**

jsx

export async function generateMetadata({

  params,

}) {

  const { cabinId } = params;

  const cabin = await getCabin(cabinId);

  return {

    title: `Cabin ${cabin.name}`,

  };

}

**Notes on Global Error Boundary in Next.js**

* **Purpose of Error Handling**
  + Ensures better user experience by preventing raw error messages.
  + Handles unexpected errors and prevents application crashes.
* **Example Scenarios for Errors**
  + **Invalid URL ID:** A user accesses a bookmarked URL with an outdated or nonexistent ID.
  + **Incorrect Property Access:** Developer mistakenly assumes capacity is an object and tries to access capacity.max, causing an error.
  + **Development vs. Production Errors:**
    - In development, detailed error screens are shown.
    - In production, a standard error page is displayed, but it should still be user-friendly.
* **Setting Up a Global Error Boundary**
  + Create an **Error.js** file in the root app folder.
  + **Must be a client component**, so include use client at the top.
  + The error component receives:
    - **error:** The actual error object.
    - **reset:** A function to reset the error boundary.
* **Error Handling Behavior**
  + Displays a custom error message (error.message).
  + Provides a button for resetting the error boundary.
  + Can be reused in any Next.js app.
* **Nested Error Boundaries**
  + Possible to define multiple error boundaries at different levels of the app.
  + The closest error boundary will handle errors in its scope.
  + **Limitations:**
    - Works only for **rendering errors** (errors during component rendering).
    - Does **not** catch errors inside **callback functions** or event handlers.
* **Errors in the Root Layout**
  + **Error.js does not catch errors in the root layout** (e.g., data fetching errors).
  + To handle root layout errors, create a **global-error.js** file.
  + **global-error.js replaces the entire layout**, so it must define its own HTML and body.

**Handling Not Found Errors in Next.js**

* **Default 404 Error:**
  + When a user navigates to an undefined route, Next.js automatically displays a default 404 error page.
  + Developers may want a custom 404 page to match the application's style.
* **Creating a Custom 404 Page:**
  + Convention: Create a file named **not-found.js** in the root of the pages directory.
  + Export a React component with a custom error message and a link to navigate back.
  + Import **Link** from **next/link** for navigation.
  + Next.js automatically renders this custom 404 page for unknown routes.
* **Manually Triggering a Not Found Error:**
  + Some errors should also be treated as "not found," such as when a requested **cabin ID** does not exist.
  + Next.js provides **a function called** **notFound()** to manually trigger the 404 page.
  + Import **notFound** from **next/navigation** and call it inside the data-fetching function if an error occurs.
* **Example Implementation:**
  + the **getCabin** function:
    - If the cabin ID does not exist, call **notFound()** instead of displaying an error message.
* **Customizing Not Found Pages for Specific Routes:**
  + Place a **not-found.js** inside a dynamic route folder (e.g., **/cabins/[cabinId]**) to customize error messages for that route.
  + Example:
    - Instead of a generic "Page Not Found" message, display **"This cabin cannot be found."**
    - Adjust navigation links accordingly (e.g., redirect users back to **/cabins**).

**Static vs. Dynamic Rendering in Next.js**

* **Next.js decides** whether each route is **static** or **dynamic** based on its content.
* **Static rendering** (pre-rendered content) and **dynamic rendering** (server-side rendered on demand) are handled differently.
* To analyze how routes are rendered, **build the site** using:
  + npm run build (as defined in package.json).
* **Running the build command** shows:
  + **ESLint warnings** (e.g., suggestions to use the Image component).
  + A **list of routes** with their rendering method:
    - **Small circle (●)** → **Statically rendered routes**.
    - **Lambda (λ) symbol** → **Dynamically rendered routes** (server-side rendered).

**Why Some Routes Are Dynamic?**

* Dynamic Routes (e.g., /cabin/[id]) **cannot be pre-rendered** at build time.
* **Next.js doesn’t know** all possible IDs in advance, so it must **generate the page on demand**.
* **Possible workaround:**
  + **Manually specify known IDs**

**Notes on generateStaticParams in Next.js**

* **generateStaticParams**: A special Next.js function used **to pre-render dynamic routes as static pages.**
* **Purpose**: Improves performance by pre-generating pages for a known set of dynamic values.
* **Use Case**: When a finite number of **dynamic URL segments** (e.g., cabinId) exist.

**How It Works**

1. **Next.js normally renders dynamic routes at request time** since it doesn’t know the possible values.
2. **Developers can specify known values** (e.g., fetching all cabinIds from a database).
3. **Next.js generates static pages for each value** at build time.

**Implementation**

* Define generateStaticParams in a dynamic page file (e.g., /cabins/[cabinId]/page.js).
* Fetch all necessary values dynamically and return them as an array.
* Each object in the array should match the **dynamic segment name** in the folder structure.

javascript

CopyEdit

export async function generateStaticParams() {

const cabins = await getCabins(); // Fetch all cabins

return cabins.map((cabin) => ({

cabinId: cabin.id.toString(), // Convert ID to a string

}));

}

**Benefits**

* **Faster performance**: Pages are served as static files.
* **Better SEO**: Pre-rendered content is indexed efficiently.
* **Reduced server load**: No need for API calls at request time.

**Key Considerations**

* Only use if the **set of possible values is known** at build time.
* If values change frequently, consider **ISR (Incremental Static Regeneration)**.
* Ensure **string format for dynamic parameters**, as Next.js expects them as strings.

**Notes on Static Site Generation (SSG) in Next.js**

* **Static Site Generation (SSG)**: The process of exporting a Next.js app as static files for deployment on any static hosting provider.
* **Why SSG?**
  + Simplifies deployment since no server-side code is needed.
  + Compatible with hosting providers like **GitHub Pages, Netlify, and Render**.
  + Improves performance and SEO by pre-rendering pages.

**Steps to Export a Next.js Site as Static**

1. **Modify next.config.js**
   * Add output: 'export' to enable full static export.

javascript

CopyEdit

module.exports = {

output: 'export',

};

1. **Run the Build Command**
   * Execute:

bash

CopyEdit

npm run build

* + This generates an out/ folder (or rename it to dist/).

1. **Deploy the out/ Folder**
   * Upload to any static hosting service by dragging and dropping.

**Handling Dynamic Routes in SSG**

* If dynamic routes exist (/cabins/[cabinId]), **use generateStaticParams** to pre-render them.
* If generateStaticParams is missing or uses **server-side logic (e.g., cookies, API calls)**, an error occurs during static export.

**Handling Images in Static Export**

* **Next.js Image Component (next/image) does not work in static export** because it relies on Vercel’s image optimization service.
* **Solutions:**
  1. **Use standard <img> tags** instead of next/image.
  2. **Use a custom image loader** (e.g., Cloudinary) by configuring an alternative optimization service.

[End of Notes]

**Next.js code** implementing generateStaticParams to pre-render static pages for known cabin IDs:

javascript

CopyEdit

// app/cabins/[cabinId]/page.js

import { getCabins } from "@/lib/api"; // Assume this function fetches all cabins

//the function should return an array of cabinId as it is the name used for dynamic route

export async function generateStaticParams() {

const cabins = await getCabins(); // Fetch all cabins

return cabins.map((cabin) => ({

cabinId: cabin.id.toString(), // Convert ID to a string

}));

}

export default async function CabinPage({ params }) {

const { cabinId } = params;

const cabins = await getCabins();

const cabin = cabins.find((c) => c.id.toString() === cabinId);

if (!cabin) {

return <h1>Cabin not found</h1>;

}

return (

<div>

<h1>{cabin.name}</h1>

<p>{cabin.description}</p>

</div>

);

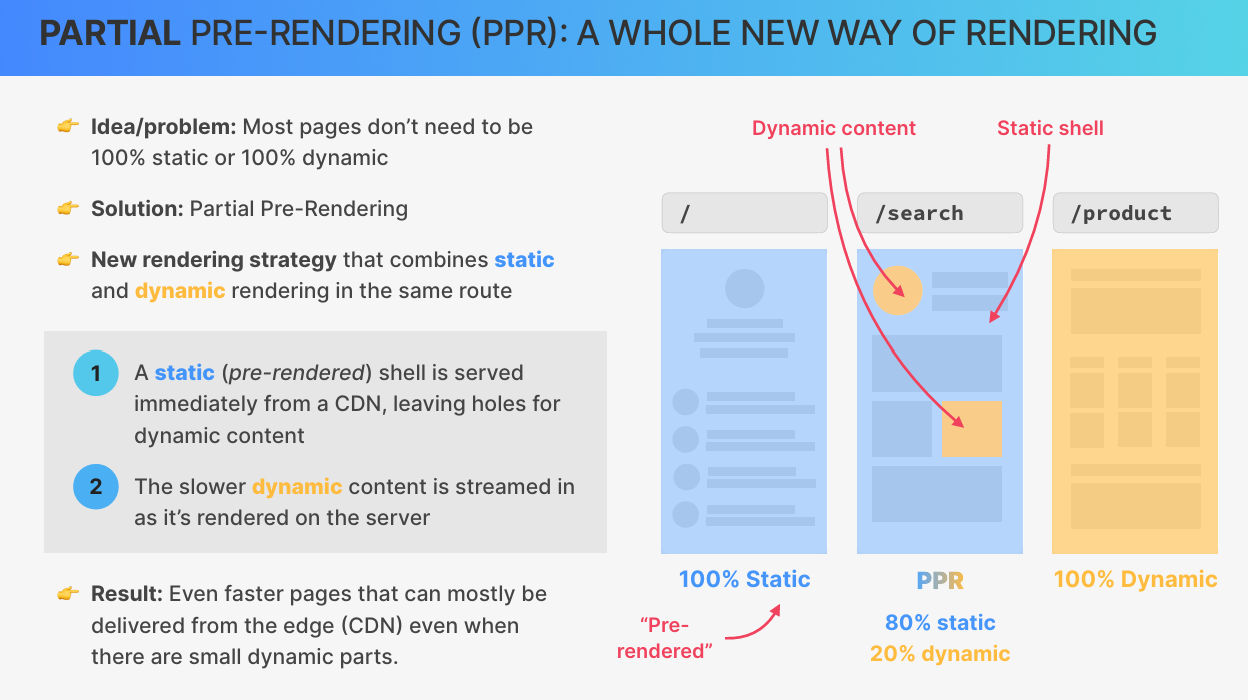
}

**🔹 Explanation:**

1. **generateStaticParams Function**:
   * Fetches all cabins using getCabins().
   * Maps over them to return an array of objects { cabinId: "89" }, ensuring IDs are **strings** (required for dynamic routes).
   * This tells **Next.js at build time** which pages should be pre-generated.

**Notes on Partial Pre-Rendering in Next.js**

* **Problem Addressed**:
  + Most websites don’t need to be fully static or fully dynamic but a mix of both.
  + Current Next.js routes are either **static** (html pre-rendered with data at build time) or **dynamic** (html rendered at request time).
  + Example: A website with static content except for a dynamic username in the navigation.
    - Currently, the entire page becomes dynamic due to one dynamic element, which is inefficient.
* **Solution: Partial Pre-Rendering (PPR)**:
  + Combines **static** and **dynamic rendering** in the same route.
  + **Pre-rendering**: **Static parts are generated at build time** and **served via a CDN for fast delivery**.
  + **Partial**: Only specific parts of the page are dynamically rendered at request time.
* **How PPR Works**:



* 1. A **static shell** (pre-rendered page) is served immediately from a CDN.
     + **Contains "holes" for dynamic content**.
  2. **The server starts rendering the dynamic content in the background**.
  3. **Dynamic content is streamed to the client as soon as it’s ready, filling the "holes" in the static shell**.
  4. Result: Faster page loads with mostly static content and small dynamic parts.
* **Benefits**:
  + Pages are no longer forced to be fully dynamic due to a single dynamic element.
  + Improves performance by leveraging CDN delivery for static parts.
* **Implementation in Next.js**:
  + **Not yet available** in Next.js 14 (as of the recording).
  + Future implementation:
    - Opt-in feature enabled in the **Next.js config file**.

npm install next@canary

/\*\* @type {import('next').NextConfig} \*/

const nextConfig = {

experimental: {

ppr: 'incremental',

},

}

module.exports = nextConfig

* + - Static rendering remains the default; PPR allows dynamic parts to coexist.
    - Dynamic components (e.g., using **cookies function**) are wrapped in a **Suspense boundary**.
      * **Suspense**: A React API used to isolate dynamic components.
      * Prevents dynamic rendering from spreading to the entire route.
    - A **static fallback** is shown while the dynamic part is rendering.
    - Once rendering is complete, the dynamic content replaces the fallback.
  + Eg code: import { Suspense } from "react"
  + import { StaticComponent, DynamicComponent, Fallback } from "@/app/ui"
  + export const experimental\_ppr = true
  + export default function Page() {
  + return {
  + <>
  + <StaticComponent />
  + <Suspense fallback={<Fallback />}>
  + <DynamicComponent />
  + </Suspense>
  + </>
  + };
  + }

Dynamic contens like these will be rendered at request time

import { cookies } from 'next/headers'

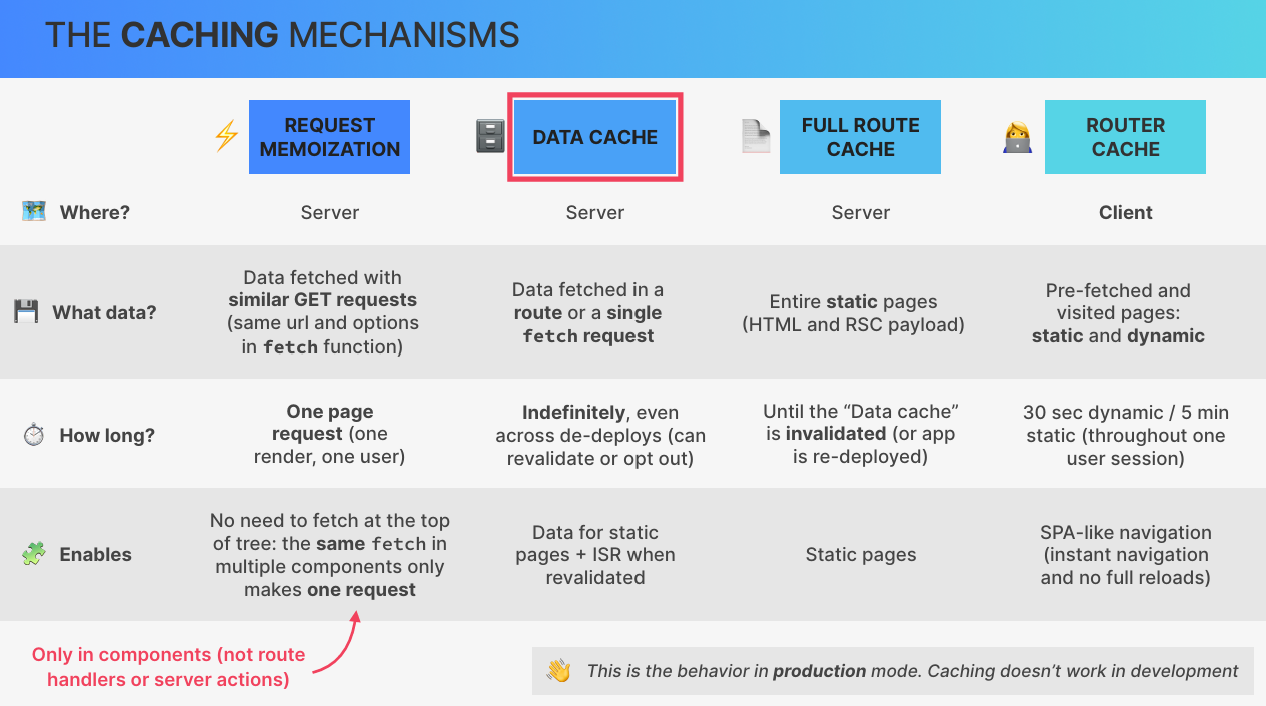
export async function User() {

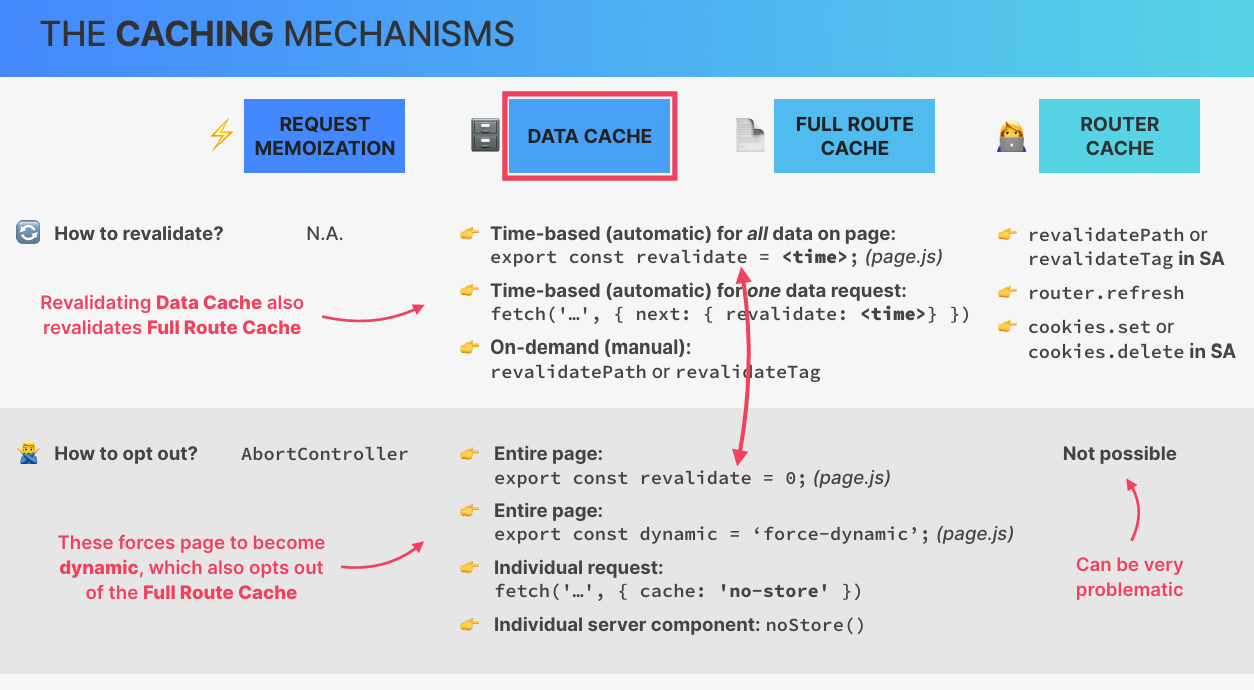
const session = (await cookies()).get('session')?.value

return '...'

}

* **Key Concepts**:
  + **Static Shell**: Pre-rendered static content with placeholders for dynamic parts.
  + **Suspense Boundary**: Isolates dynamic components and defines "holes" in the static shell.
  + **Static Fallback**: Temporary content shown while dynamic parts are rendering.
* **Advantages of PPR**:
  + No new APIs to learn (uses existing React Suspense).
  + Improves performance by minimizing dynamic rendering.
  + Straightforward to implement once available.





**Notes on Next.js Caching Mechanisms**

**What is Caching?**

* **Caching** is the process of storing fetched or computed data in a temporary location for future access.
* Purpose: Avoids re-fetching or re-computing data, improving performance and reducing costs.
* In **Next.js**, caching is aggressive by default, affecting both server and client-side data.

**Benefits of Caching**

* Improves app performance with faster page loads.
* Reduces computing and data access costs (e.g., fewer API calls to a CMS).

**Challenges with Next.js Caching**

* Caching is always **on by default** in the **app router**, leading to potential issues:
  + Displaying **stale data** (outdated data) on the client.
  + Some caches **cannot be turned off**, causing frustration for developers.
* Caching behavior is one of the most criticized aspects of Next.js.

**Four Caching Mechanisms in Next.js**

1. **Request Memoization** (Server-side):
   * Caches data fetched during a single page render for one user.
   * Prevents duplicate network requests for the same data within the same render.
   * Example: Fetching products in multiple components only results in one API call.
   * Works only with the native fetch function and identical requests (same URL and options).
   * **Limitations**:
     + Only works in React components (not in route handlers or server actions).
     + No revalidation possible (data is not persisted).
2. **Data Cache** (Server-side):
   * Stores **fetched data for a specific route** or **fetch request**.
   * Data **persists across multiple requests and users**, **even after app redeploys**.
   * Enables **Incremental Static Regeneration (ISR)**:
     + **Static pages are regenerated when the data cache is revalidated**.
   * **Revalidation**:
     + Time-based: Set a revalidate constant (in seconds) to automatically refresh data.
     + On-demand: Use revalidatePath or revalidateTag to manually refresh data.
   * **Opting Out**:
     + Set revalidate to 0 (always revalidate) or use dynamic constant to force dynamic rendering.
     + Use noStore() in server components or specific fetch APIs to disable caching.
3. **Full Route Cache** (Server-side):
   * Stores entire static pages as **HTML** and **RSC payload** at build time.
   * Enables static pages to be served to multiple users.
   * Persists until the **data cache** is invalidated.
   * Cleared on app redeploy.
   * **Opting Out**:
     + Disabling the data cache also disables the full route cache.
4. **Router Cache** (Client-side):
   * Stores **pre-fetched** and **visited pages** in the browser.
   * Enables instant navigation, creating a single-page application (SPA) feel.
   * **Issues**:
     + **Pages are not re-fetched from the server**, leading to stale data.
     + Cache duration:
       - Dynamic pages: 30 seconds.
       - Static pages: 5 minutes.
     + No way to opt out or manually revalidate.
   * **Revalidation**:
     + **Triggered by revalidating the data cache in a server action**.
     + Use Router.refresh() or modify cookies in server actions.

**Key Points on Cache Control**

* **Request Memoization**:
  + No revalidation; opt-out using an abort controller with fetch.
* **Data Cache**:
  + Highly configurable with time-based and on-demand revalidation.
  + Opt-out by setting revalidate to 0 or using dynamic.
* **Full Route Cache**:
  + Tied to the data cache; revalidated when data cache is cleared.
* **Router Cache**:
  + No opt-out option; revalidated indirectly via data cache or manual refresh.

**Production vs. Development Caching**

* **Production**:
  + All caching mechanisms are active.
* **Development**:
  + Almost no caching occurs to ensure developers always see fresh data.

**Criticism of Next.js Caching**

* **Router Cache** is problematic:
  + No opt-out option.
  + Stale data can persist for 30 seconds (dynamic) or 5 minutes (static).
  + Hard to guarantee up-to-date data for users.

**Conclusion**

* Next.js caching is powerful but complex, with aggressive defaults.
* Understanding and controlling caching requires familiarity with multiple APIs and mechanisms.
* Developers must balance performance benefits with the risk of stale data.

[End of Notes]

**Next.js Caching and Incremental Static Regeneration (ISR)**

* **Development Mode & Caching**
  + Caching does **not work** in development mode.
  + To simulate a **production environment**, use:
    - npm run build → Builds the application.
    - npm run start → Runs the production server.
    - To simplify, create a script:
    - "scripts": {
    - "prod": "next build && next start"
    - }

Run with npm run prod.

* **Static Site Generation (SSG) & Data Caching**
  + Pages are **prerendered as static content** (SSG).
  + Data is cached **at build time** and does not change until rebuild.
  + Example:
    - Supabase data (e.g., cabin price) **does not update** on reload.
    - The issue is **not browser caching**, but **Next.js full-route caching**.
* **Making a Route Dynamic (Disabling Caching)**
  + To **disable caching** and fetch fresh data on each request:
  + export const revalidate = 0;
  + This forces the page to regenerate **dynamically**.
  + Downside: Every request queries the database, increasing load.
* **Incremental Static Regeneration (ISR)**
  + **Middle ground** between SSG (static) and SSR (dynamic).
  + Re-generates a static page **at intervals** without full rebuild.
  + Implement by setting revalidate to a time interval (in seconds):
  + export const revalidate = 3600; // 1 hour
  + **How it works**:
    - Users **see the cached page**.
    - After the interval (e.g., 15 sec), Next.js **fetches fresh data**.
    - The updated page is **cached** for future users.
  + **Best practice:**
    - Use **longer intervals** for infrequent updates (e.g., blog pages).
    - Use **shorter intervals** for frequently changing data (e.g., stock prices).
* **Testing ISR**
  + Set revalidate = 15 for a **15-second cache update interval**.
  + Change data in Supabase → Refresh page → See update **after 15 seconds**.
  + Increase interval (e.g., 60 for 1 min) to **observe the effect better**.

[End of Notes]

**Incremental Static Regeneration (ISR)**

* **Purpose:** Middle ground between fully static and fully dynamic pages.
* **How It Works:**
  + Static pages are **regenerated** at specified intervals.
  + Fresh data is **fetched and cached** periodically.
  + Users get updated content **without** rebuilding the app.
* **Implementation:**
  + Set revalidate to a specific interval (in seconds):

js

CopyEdit

export const revalidate = 3600; // Revalidate every 1 hour

* + In the example, testing with revalidate = 15 updates data every **15 seconds**.

**How to Use no-store in Next.js**

**1. Using fetch with cache: 'no-store'**

When fetching data, you can pass { cache: 'no-store' } as an option to force fresh data retrieval.

js

CopyEdit

export async function getCabins() {

const response = await fetch('https://your-api-url.com/cabins', {

cache: 'no-store',

});

const data = await response.json();

return data;

}

* This ensures that the data **never gets cached** and is always fetched from the server.

**2. Using revalidate = 0 (Page-Level No Cache)**

If you want **the entire page to always fetch fresh data**, set revalidate = 0:

js

CopyEdit

export const revalidate = 0;

* This tells Next.js that the page **should not use ISR or caching**, forcing it to fetch new data on every request.