ChatGPT

NextMegaWebDev

Table of Contents

Next.js 15 Core Concepts

File-based Routing (app/ directory)

Next.js uses a **file-system based router**: each file or folder in the `app/` directory becomes a route. For example, `app/dashboard/page.tsx` maps to `/dashboard`. Folders can be nested to form nested URL segments; each subfolder defines a sub-route [1]. This means simply organizing files and folders in `app/` creates the site's route structure. Unlike traditional React routers, you do not manually configure routes – Next.js infers them from the filesystem.

Dynamic & Nested Routes

You can create **dynamic routes** by naming folders/files with brackets. For instance, `app/blog/[slug]/page.tsx` will match `/blog/my-post` and provide `params.slug` to the component [2] [3]. You can also define **catch-all** routes using `...`, like `[...paths]`, or **optional catch-all** with `[[...paths]]`. Nested routes are achieved by nesting folders: e.g. `app/dashboard/settings/page.tsx` yields `/dashboard/settings`. Each folder can also contain its own `layout.tsx` and `page.tsx` (see below) to build complex nested layouts [1] [4]. In Next.js 15, be aware that dynamic route `params` are now promises by default, so you should use `await` or React's `use` to access them (older synchronous behavior is still supported but deprecated) [3].

Client vs Server Components

Next.js 15 fully embraces React Server Components. By default, components are **Server Components**, rendering on the server. To create a **Client Component** (for use of state, effects, event handlers, or browser APIs), you add the directive `"use client"` at the top of the file [5]. Only mark a component as a client component if it needs interactivity (state, `useEffect`, `onClick`, etc.) [5] [6]. This ensures that only necessary code is shipped to the browser. Server Components can fetch data and use secrets safely (e.g. database calls) with minimal client-side JS, improving performance [7]. When using `"use client"`, remember that props passed from server to client must be serializable [6]. A good pattern is to keep most UI as Server Components and only opt into client-side rendering for small interactive parts [5] [8].

Page vs Layout Components

Within each route folder, you create a `page.tsx` (or `.js/.jsx/.tsx`) and an optional `layout.tsx`. A **Page Component** renders the content for that specific route segment [9]. A **Layout Component** wraps pages in that folder (and nested routes), providing shared UI (e.g. navigation, header) and preserving state during navigation [10]. For example, a root `app/layout.tsx` defines the overall HTML `<body>` and wraps all pages; a `app/blog/layout.tsx` could wrap all blog-related pages. Layouts render their children via a `children` prop, and you can nest layouts by placing a `layout.tsx` in each subfolder [4]. In summary: **Page** = specific route UI; **Layout** = shared wrapper UI for its folder's pages [9] [10]. Next.js enforces one layout and one page per folder to keep the structure clear.

Rendering Methods

Next.js supports multiple rendering strategies:

- **Server-Side Rendering (SSR)**: Page is rendered on each request on the server. In the Pages Router, this uses `getServerSideProps`. SSR ensures fresh data on each request [11]. However, Next.js 15's App Router does not use `getServerSideProps`; instead, you can opt into dynamic rendering (no caching) or use Server Actions.

- **Static Site Generation (SSG)**: Page is rendered at build time. Next.js calls `getStaticProps` (Pages Router) during build, producing static HTML. This is fast on load but requires rebuilds to update content [12] [13].

- **Incremental Static Regeneration (ISR)**: A hybrid. You statically generate a page at build, but specify a `revalidate` interval so Next.js re-generates the page in the background on the server (and updates the static cache) when requests come in [14] [15]. This lets you update static pages without a full rebuild. In App Router, you can also set `revalidate` in fetch options (as shown later in Sanity example).

- **Client-Side Rendering (CSR)**: Data fetching and rendering happen entirely in the browser. For example, using `useEffect` or SWR to fetch data after the page loads [16] [17]. CSR delays initial content (affecting SEO/first paint), but is useful for interactive pages or real-time updates on the client.

- **Streaming (Server Components + Suspense)**: With React 18's Suspense, Next.js can stream HTML to the client in chunks as it's generated [18]. This means parts of the page (e.g. above-the-fold) can render immediately before all data is ready [18]. Streaming is fully server-rendered (so SEO is fine) and improves metrics like TTFB and FCP by letting high-priority parts load first [18] [19]. In App Router, you can create loading states (via `loading.js` or `<Suspense>`) to enable streaming behavior. In summary, streaming breaks pages into smaller pieces on the server and progressively renders them on the client [18].

API Routes, Route Handlers, and Middleware

Next.js provides built-in support for server-side routes and middleware:

- **API Routes (Pages Router)**: In the Pages Router, placing a file in `pages/api/*.js` (or `.ts`) creates an API endpoint. You export a default function `(req, res) => { … }`, and Next.js handles HTTP methods for you (e.g. GET/POST).

- **Route Handlers (App Router)**: In the App Router, API-like routes are created with **Route Handlers**. Instead of `pages/api`, you create a folder like `app/api/hello/route.ts`. Inside, you export `GET`, `POST`, etc., functions using the Web Fetch API (Request/Response) [20] [21]. For example:

```ts
export async function GET(request: Request) {
  return new Response("Hello, world!");
}
```

This handles `/api/hello`. Route Handlers behave similarly to API Routes but live alongside App Router pages. They support fetch methods and you can use `NextResponse` for redirects, cookies, etc [20] [21].

- **Middleware**: Next.js allows a single `middleware.ts` at the project root to intercept all requests before they hit routes [22] [23]. Middleware runs (usually at the Edge runtime) and can inspect/modify the request: e.g. perform redirects, rewrites, or header changes based on conditions [22]. It is ideal for tasks like authentication redirects or A/B experiments. For example:

```ts
import { NextResponse } from 'next/server';
export function middleware(request) {
  if (!request.cookies.get('token')) {
    return NextResponse.redirect(new URL('/login', request.url));
  }
  return NextResponse.next();
}
```

This checks a cookie on every request and redirects if not present. Note: heavy data fetching in middleware is discouraged; use it for quick checks and rewrites [22].

Authentication with next-auth

Setting up Providers (Credentials, GitHub, Google)

next-auth (Auth.js) simplifies auth in Next.js. After installing `next-auth`, you create an auth route. In the Pages Router you'd add `pages/api/auth/[...nextauth].js` (or `.ts`) containing:

```ts
import NextAuth from "next-auth"
import GitHubProvider from "next-auth/providers/github"
// import GoogleProvider from "next-auth/providers/google"
export default NextAuth({
  providers: [
    GitHubProvider({
      clientId: process.env.GITHUB_ID,
      clientSecret: process.env.GITHUB_SECRET
    }),
    // GoogleProvider({ clientId, clientSecret }),
    // CredentialsProvider({...})
```

```
  ],
})
```

All requests to `/api/auth/*` (signIn, callback, signOut, etc.) are then handled automatically [24]. For **Credentials Provider** (username/password), next-auth lets you define an `authorize()` function that checks credentials and returns a user object (or `null` on failure) [25] [26]. For OAuth like GitHub or Google, you add `Providers.GitHub()` or `Google()` with your app's client ID/secret [24] [27]. For example:

```
import GoogleProvider from "next-auth/providers/google"
export default NextAuth({
  providers: [
    GoogleProvider({ clientId: process.env.GOOGLE_ID, clientSecret:
process.env.GOOGLE_SECRET })
  ]
})
```

[27]. NextAuth handles the OAuth flow; you must configure redirect URIs in the provider's developer console (e.g. `http://localhost:3000/api/auth/callback/google`).

Protecting Pages and Sessions

To protect pages, next-auth provides hooks and middleware. On the client-side, you use the `useSession()` hook in a component: it returns `session` and `status`. For example, you can render a loading state while `status==="loading"`, and show an "Access Denied" message if `status==="unauthenticated"`, or the protected content if `session` is valid [28]. Example:

```
import { useSession } from "next-auth/react"
export default function Page() {
  const { data: session, status } = useSession();
  if (status==="loading") return <p>Loading…</p>;
  if (!session) return <p>Access Denied</p>;
  return <p>Protected Content for {session.user.email}</p>;
}
```

[28].

For server-side protection, NextAuth v4 offers middleware. You can add `export { default } from "next-auth/middleware"` in a `/middleware.js`, which protects all pages by default or by using a `matcher` config for specific paths [29]. This makes route protection declarative, redirecting unauthenticated users to sign-in. (Note: the middleware uses JWT sessions behind the scenes [30].)

Also, wrap your app with `<SessionProvider>` (from `next-auth/react`) in `_app.js` (or the root layout) to make session data available in client components [31]. Example:

```
// pages/_app.js
import { SessionProvider } from "next-auth/react"
export default function App({ Component, pageProps: {session, ...pageProps} }) {
  return (
    <SessionProvider session={session}>
      <Component {...pageProps} />
    </SessionProvider>
  );
}
```[31]. This ensures `useSession` works correctly throughout the app.

Client-side vs Server-side Session Handling

NextAuth provides different session APIs: On the client, use the `useSession()` hook or `getSession()` to access the current session (asynchronous). On the server, use
`getServerSession()` inside page handlers or route handlers. The key difference is that `getSession()` is only for client (browser) usage, whereas `getServerSession()` should be used in server components or APIs[32]. In Next.js 13 App Router, you can call `getServerSession(authOptions)` in a server component or route to read the session. Session callbacks (in `[...nextauth]` config) allow customizing JWT contents or persisting sessions. Keep in mind that `useSession()` has a
`status` you should handle (loading/unauthenticated) on the client[28], while on the server you check if the session object is non-null.

Search Forms and Real-Time Features

Debounced Search

For a responsive search bar, you should debounce the user's input to avoid flooding the server with requests. A common pattern in React is to use `useState` for the query and `useEffect` with `setTimeout`. For example:
```jsx
function SearchBar() {
  const [query, setQuery] = useState("");
  const [results, setResults] = useState([]);
  useEffect(() => {
    const handler = setTimeout(() => {
      if (query) fetch(`/api/search?q=${query}`)
        .then(res => res.json())
        .then(data => setResults(data));
    }, 500); // wait 500ms after last keystroke
    return () => clearTimeout(handler);
  }, [query]);
  // ...
}
```

This waits 500ms after the user stops typing before calling the API. You can use `lodash.debounce` or write a custom hook for debounce. For example, a custom hook might set a delayed "key" for SWR to use, as shown in one tutorial [33]:

```
useEffect(() => {
  const handle = debounce(() => setKey(query), 300);
  handle();
  return () => handle.cancel();
}, [query]);
```

[33]. This way, the API is hit only after the user has paused typing, improving performance and UX.

Real-Time Updates (WebSockets, SWR, Server Actions)

For live data, you have several approaches: - **WebSockets (e.g. Socket.io)**: Establish a WebSocket connection between client and server for push updates. For example, a Next.js page can connect via `socket.io-client` and listen for "notification" events from the server. When the server emits data, the client updates immediately [34]. The GeeksforGeeks example shows a Next.js app using Socket.io for real-time notifications via websockets [34]. - **SWR / React Query**: On the client, libraries like SWR or React Query can be used to poll or revalidate data. SWR's revalidation can be triggered on intervals, focus, or via a manual `.mutate()`. You could combine SWR with webhooks or browser events to refresh. For example, SWR automatically retries on failure and revalidates stale data [35] [36]. It provides a `useSWR` hook for fetching data, which you can trigger after some event. - **Next.js Server Actions (Experimental)**: In Next.js 15+, you can define server actions (`'use server'` functions) and invoke them from client components (e.g. via `<form action={...}>` or onClick) [37]. This sends a server request behind the scenes and can re-render components without a full page reload. While not a real-time push, server actions let you mutate data on the server easily and return updated UI. - **SSE or Pub/Sub**: Alternatively, server-sent events (EventSource) or a pub/sub service can push updates that clients subscribe to.

In practice, for a user notification feature you might have the server emit updates on a socket, and all connected clients listening will update their UI immediately. Meanwhile, data caches (SWR) can also poll or be invalidated after a mutation. The combination of websockets for pushes and optimistic UI (see below) is common for a real-time feel [34].

Sanity CMS Integration

Setting Up Sanity Studio

Sanity is a headless CMS with its own "Studio" app. To set it up, install Sanity CLI and run (from your terminal):

```
npm create sanity@latest -- --dataset production --template clean --typescript
--output-path studio
```

```
cd studio
npm run dev
```

This creates a new Sanity Studio project (using TypeScript template) and starts it on http://localhost:3333 [38]. The Sanity Studio is a React app where you define your schemas (e.g. blog posts, authors) and manage content. For Next.js integration, you typically run the Studio locally (or deploy it separately) and use Sanity's API to fetch content. The above steps come from Sanity's official quickstart [38].

Fetching & Rendering Sanity Content

In your Next.js frontend, use the Sanity client to fetch content via GROQ queries. Install utilities:

```
npm install next-sanity @sanity/image-url @portabletext/react
```

Configure the Sanity client (e.g. in `lib/sanity.js`) with your project ID and dataset. Then in a Server Component, you can do:

```
import { client } from '@/sanity/client';
export default async function PostPage({ params }: { params: { slug:
string } }) {
  const query = '*[_type=="post" && slug.current==$slug][0]';
  const post = await client.fetch(query, { slug: params.slug }, { next: {
revalidate: 30 } });
  return (
    <article>
      <h1>{post.title}</h1>
      {/* Render content */}
    </article>
  );
}
```

This uses `client.fetch()` with a GROQ query and passes the `slug` parameter. Note the `{ next: { revalidate: 30 } }` option which enables ISR for this page [39]. To render rich text (Portable Text) from Sanity, use the `<PortableText>` component:

```
import { PortableText } from "next-sanity";
{/* ... */}
<div>
  {Array.isArray(post.body) && <PortableText value={post.body} />}
</div>
```

This converts Sanity's block content into React elements [40]. For images, you can use `@sanity/image-url` to build a URL: the example does `imageUrlBuilder(...).image(source).width(550).height(310).url()` [41]. In summary, fetch

Sanity content with GROQ queries via the client in your Next.js pages, then display fields (text, images) in JSX.

Live Previews with Sanity + Next.js

Sanity provides a real-time **preview** feature so that content edits in the Studio can immediately reflect in the Next.js site (without a full rebuild). The idea is to use a live listener (via Sanity's real-time APIs) in your preview mode. In practice, you set up a preview session that subscribes to changes for a given query. Sanity's toolkit includes a React hook that uses the same GROQ query but also listens for updates when you're logged into Sanity [42]. As one Sanity blog explains, "we are able to mimic a part of our real-time datastore in the browser...exposed as a React Hook that you use in your page template to update the page data coming in from the Next.js data" [42]. In other words, when the author edits a document in Sanity Studio, the hook fetches the new content and updates the page instantly. Setting this up typically involves configuring the `withPreview` mode and using `previewSubscription` from `@sanity/preview-kit` or similar. The result is a "what you see is what you get" editing flow: draft changes appear live on the site.

Rendering Based on Cookies and Sessions

Reading Cookies on Server/Client

Cookies in Next.js are handled differently on server and client. In a **Server Component** or API Route, you can use Next.js's `cookies()` function (from `next/headers`) to read cookies. This is asynchronous in Next.js 15. Example:

```
import { cookies } from 'next/headers';
export default async function Page() {
  const cookieStore = await cookies();
  const theme = cookieStore.get('theme');
  // theme.value is the cookie value if it exists
  return <body className={theme ? theme.value : 'light'}>...</body>;
}
```

[43]. This reads the `theme` cookie on each request (since the browser sends all cookies to the server). In contrast, on the **client-side**, cookies can be accessed via `document.cookie` (if not `HttpOnly`) or via libraries like `js-cookie`. Remember that **HttpOnly** cookies (common for sessions) cannot be read by client JS – they must be handled on the server.

Setting Cookies (Server vs Client)

To **set cookies**, you must send a `Set-Cookie` header from the server. In Next.js App Router, use a **Route Handler** or **Server Action** to modify cookies. For example:

```
// app/api/login/route.ts
import { NextResponse } from 'next/server';
export async function POST(request) {
```

```
  const response = NextResponse.redirect(new URL('/', request.url));
  response.cookies.set('token', 'abc123', { httpOnly: true, secure: true });
  return response;
}
```

This sends the cookie to the browser on login. You cannot set cookies from inside a normal Server Component; it must be done in a response context (Route Handler or action) [44] [45] . In the Route Handler example above, the cookie is `HttpOnly` (so JavaScript can't read it) and `secure` (HTTPS only). For **client-side setting** (less secure), you could use `document.cookie = "name=value; path=/;"` in an effect, but this is rarely needed if you handle auth on the server. In short: use `response.cookies.set()` on the server [45] , and use `cookies().get()` in Server Components to read. Also note that Next.js will manage cookie deletion when you e.g. do `response.cookies.delete('name')` in a handler [46] .

Conditional Rendering by Cookie/Session

Once you have cookies or session info, you can conditionally render UI. For example, if a `token` cookie indicates the user is signed in, you might render a user menu; otherwise show "Sign In". In a Server Component:

```
const token = (await cookies()).get('token');
if (!token) {
  return <p>Please <Link href="/login">log in</Link>.</p>;
}
```

Similarly, you could read a `theme` cookie and add a class to `<body>` to toggle dark mode. In a Client Component, you might check `useSession().status`. The key is to branch your JSX based on the cookie or session value. For instance, you may do:

```
export default async function Header() {
  const session = await getServerSession(authOptions);
  return session ? <UserMenu user={session.user} /> : <SignInButton />;
}
```

Or in a client component:

```
const { data: session } = useSession();
return session ? <Profile /> : <LoginPrompt />;
```

No matter the method, the pattern is to **read the cookie/session first**, then use simple `if` checks in your render. The examples above use functions from Next.js or next-auth to retrieve the values (and we cite reading cookies [43] and using `useSession()` [28] accordingly).

PostgreSQL Guide

Introduction and Basics

PostgreSQL is a powerful open-source **relational database** (RDBMS) with over 20 years of development [47]. It supports standard SQL for structured data and also offers JSON capabilities for semi-structured data. Core concepts in PostgreSQL include **databases**, **schemas**, **tables**, and **rows**. A **table** has named columns (fields) and stores records; common SQL commands include `CREATE TABLE`, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and so on (you can find full tutorials online [48] [47]). Tables can relate to each other via **primary keys** and **foreign keys** – for example, a `users` table with a `user_id` primary key and a `posts` table where each row has a `user_id` foreign key referring to `users`. This enforces relational integrity.

Basic SQL in Postgres looks like:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  email TEXT UNIQUE NOT NULL
);
INSERT INTO users (name, email) VALUES ('Alice', 'alice@example.com');
SELECT * FROM users WHERE name = 'Alice';
```

You can also define **schemas** (namespaces) to organize tables. Indexing is important for performance: you create an index on a column to speed up searches (e.g. `CREATE INDEX ON users(email)`), which PostgreSQL heavily supports (see advanced features below).

Postgres also supports ACID transactions (guaranteeing reliability of multi-step operations) [49]. You can wrap multiple statements in `BEGIN ... COMMIT;` to make them atomic. This means either all statements succeed, or if one fails, the entire transaction rolls back (ensuring data consistency). Transaction support makes Postgres reliable for critical applications. In short, PostgreSQL is a full-featured SQL database ideal for most use cases [47] [49].

Advanced SQL Features

PostgreSQL includes many advanced database features. Some key ones:

- **Indexes**: Structures to speed up queries. Beyond simple indexes, Postgres has multicolumn indexes, unique indexes, partial indexes (index only some rows), expression indexes, etc. [49]. Use `CREATE INDEX` or declare a column `UNIQUE` to index it. Proper indexing is crucial for large tables to keep queries fast.

- **Views**: Virtual tables defined by queries. A **view** is created with `CREATE VIEW v AS SELECT ...`. You can query a view like a table; it represents a saved query, useful for abstraction or simplified reporting.

- **Triggers and Functions**: You can define triggers that run on table events (INSERT, UPDATE, DELETE). A trigger usually calls a user-defined function. For example, a trigger could automatically update a timestamp or enforce a rule. Postgres's procedural language (PL/pgSQL) lets you write functions: e.g. `CREATE FUNCTION log_insert() RETURNS trigger AS $$ BEGIN ... END; $$ LANGUAGE plpgsql;`. Then `CREATE TRIGGER ...`. Triggers are powerful for audit trails or derived column updates [49].

- **Transactions & Concurrency**: PostgreSQL offers full ACID transactions and MVCC (multi-version concurrency control). This means many users can read/write concurrently without interfering. The `SERIALIZABLE` isolation level provides true serial execution guarantees, while `READ COMMITTED` or `REPEATABLE READ` handle most cases.

- **Joins**: SQL joins combine tables in queries. PostgreSQL supports INNER, LEFT, RIGHT, FULL, CROSS, and even advanced joins like LATERAL. For example: `SELECT orders.id, users.name FROM orders JOIN users ON orders.user_id = users.id;`. Proper joins and foreign keys let you navigate relational data effectively.

In summary, Postgres goes beyond basic SQL: it has extensive data types, indexing options, views, triggers, and transaction support [49]. It is highly extensible and suitable for both simple and complex applications.

Using Prisma with PostgreSQL

Prisma is a popular ORM (Object-Relational Mapping) for Node.js and TypeScript, often used with Next.js. To connect Prisma to Postgres, first set the `DATABASE_URL` in your `.env` (e.g. `postgresql://user:password@localhost:5432/mydb`). In `schema.prisma`, define your models, e.g.:

```
model User {
  id    Int    @id @default(autoincrement())
  name  String
  email String @unique
}
```

Then run `npx prisma migrate dev` to apply the schema to your database, and `npx prisma generate` to generate the client. In your code, import and use the PrismaClient. Typically you create a singleton instance (e.g. in `lib/prisma.js`) to avoid too many connections [50]. Example setup:

```
// src/lib/prisma.js
import { PrismaClient } from '@prisma/client';
let prisma;
if (!global.prisma) {
  global.prisma = new PrismaClient();
}
export const prisma = global.prisma;
```

Now you can use `prisma` in Next.js API routes or Server Components.

CRUD Example with Next.js & Prisma

With Prisma set up, performing CRUD operations is straightforward. For example, to **fetch** users:

```
// app/api/users/route.ts (Server Component or API route)
import { prisma } from "@/lib/prisma";
export async function GET() {
  const users = await prisma.user.findMany();
  return new Response(JSON.stringify(users));
}
```

This returns all users. To **create** a new user:

```
export async function POST(request: Request) {
  const { name, email } = await request.json();
  const user = await prisma.user.create({ data: { name, email } });
  return new Response(JSON.stringify(user), { status: 201 });
}
```

To **update** a user:

```
export async function PATCH(request: Request) {
  const { id, name, email } = await request.json();
  const user = await prisma.user.update({
    where: { id },
    data: { name, email },
  });
  return new Response(JSON.stringify(user));
}
```

And to **delete**:

```
export async function DELETE(request: Request) {
  const { id } = await request.json();
  await prisma.user.delete({ where: { id } });
  return new Response(null, { status: 204 });
}
```

These examples follow the Next.js 14+ App Router pattern (Route Handlers) with Prisma. They mirror each other and follow best practice: data is parsed from the request body, then Prisma methods like `findMany`, `create`, `update`, `delete` are called [51] [52]. Indeed, one tutorial shows exactly these Prisma calls in Next.js:

```
const users = await prisma.user.findMany();
const user = await prisma.user.create({ data: { name, email } });
const user = await prisma.user.update({ where: { id }, data: { name, email } });
await prisma.user.delete({ where: { id } });
```

[51] [52]. Integrate these into your Next.js API or server components for full CRUD functionality.

Top 20 Common Mistakes & Fixes

1. **Missing** `"use client"`: Forgetting to add `"use client"` at the top of an interactive component causes errors (e.g. using `useState` without it). Fix: add `'use client'` to any component file that uses hooks or browser-only code [5].

2. **Incorrect Session Setup**: Not wrapping your app with `<SessionProvider>` means `useSession()` won't work. Always include the provider in `_app.js` or root layout [31].

3. **Async in Client Components**: Trying to `await` inside a client component (without marking it async) or calling server APIs directly. Fix: use server actions or route handlers for server code, and keep client components plain functions.

4. **Not Handling Loading State**: When using `useSession()` or `useSWR()`, forgetting to handle the loading state can flash incorrect UI. Always check `status === 'loading'` or loading flag before rendering sensitive content [28].

5. **Wrong Data Fetching Method**: Using `getStaticProps` in the App Router (it only works in Pages Router) or forgetting to add `async`/`await` in server components fetch. Use `client.fetch()` in Server Components or SWR in client components instead.

6. **Caching Mistakes**: Not setting `revalidate` or caching incorrectly (all pages static vs all dynamic). By default, App Router pages are dynamic (no cache) unless you configure caching. To use ISR, explicitly set a `revalidate` option in the fetch or use the `export const revalidate = N` const.

7. **Dynamic Route Parameters**: Forgetting to convert `params` promise to value in a dynamic route. In Next 15, `params` is a promise, so you may need `const { slug } = await params;` or wrap with React's `use` hook [3].

8. **Context Misuse**: Trying to use React context in a Server Component or placing a provider in `layout.tsx` incorrectly. Next.js requires providers to be client components that wrap children [53]. Fix by making a separate client-side provider component.

9. **Request vs Router**: Attempting to read the Next.js `Request` object directly in a server component (it's not available) [54]. Instead, use `headers()`, `cookies()`, or the `params`/`searchParams` props provided to the component [54].

10. **Key Prop in Lists**: In JSX loops (`.map`), forgetting the `key` prop leads to subtle bugs. Always give a stable `key` (e.g. `post.id`) to each rendered item.

11. **Improper Routing File Structure**: Misplacing `page.js` outside of an `app/` folder or naming it incorrectly. Ensure each route folder has a `page.tsx` (or `.js`) to define the page UI.

12. **Cookie Misconfiguration**: Forgetting security flags on cookies. Always set `httpOnly: true` for session cookies and `secure: true` in production. Without `httpOnly`, scripts can steal cookies; without `secure`, cookies might send over HTTP.

13. **CSR/SSR SEO Neglect**: Not considering SEO – for SEO-critical pages, use SSR/SSG so crawlers see full HTML. Pure CSR can hurt SEO unless handled carefully.

14. **Unoptimized Images**: Serving large images without Next's `<Image>` optimization or missing `width`/`height` can bloat pages. Use `next/image` for optimal delivery.

15. **Missing Dependencies**: React hooks like `useEffect` or `useCallback` need correct dependency arrays. Forgetting dependencies causes stale closures or infinite loops.

16. **Large Client Bundles**: Importing heavy libraries in client components unnecessarily (e.g. putting an icon library in a layout). Use dynamic imports or lighter libs (see next section on patterns).

17. **State on Server**: Trying to use React state in a Server Component. State must live in a Client Component (`"use client"`). Don't call hooks in server components.

18. **Auth Callback Errors**: For next-auth, not configuring callback URLs correctly with GitHub/Google leads to errors. Ensure the OAuth settings (redirect URIs) match NextAuth routes [55].

19. **Wrong Environment Variables**: Putting secrets in `NEXT_PUBLIC_` prefix – these become public. Use `NEXT_PUBLIC_` only for safe variables, others should stay server-only [56]. Also remember to restart the server after changing `.env` files.

20. **Ignoring Error Boundaries**: Not using an error boundary around critical UI. A thrown error can crash the whole app. Wrap parts of your app with an `<ErrorBoundary>` component (see below) to display a fallback UI instead of a blank page [57].

Top 20 Useful Patterns for Real-Time Apps

1. **WebSocket Notifications**: As mentioned, use Socket.io or native WebSockets. The server can emit events and the React client listens. For example, connect in a layout with `useEffect` and update state on message. This is ideal for chat, notifications, live feeds [34].

2. **Optimistic UI Updates**: When mutating data (e.g. sending a message), update the UI first and then send the request. If it fails, revert or show error. Libraries like SWR or React Query support optimistic updates (e.g. `mutate()`).

3. **Error Boundaries**: Wrap parts of your app in an error boundary component so one component's crash doesn't unmount everything [57]. Example:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError() { return { hasError: true }; }
  componentDidCatch(error, info) { logError(error, info); }
  render() { return this.state.hasError ? <FallbackUI /> :
this.props.children; }
}
```

   Place `<ErrorBoundary>` around heavy or third-party components [57].

4. **Toast Notifications/Alerts**: Use a notification system (e.g. react-toastify) to show real-time alerts. Trigger toasts on events like message received or errors. This pattern is common for informing users of background events.

5. **Dynamic Imports**: Lazily load large or rarely used components with `next/dynamic` [58]. For instance, a map or chart library can be loaded only when needed:

```
const Map = dynamic(() => import('../components/Map'), { ssr: false });
```

   Then render it behind a condition (e.g. modal open) [58]. This speeds up initial load.

6. **React Suspense for Data**: Use React's `<Suspense>` and `loading.js` (Next.js) to show fallback UIs while data or components load. For example, a `<Suspense fallback={<Spinner/>}>` around a data-driven part of the page defers its rendering until ready, creating smoother transitions.

7. **Pop-up Modals with Portals**: For in-app dialogs, use React portals so modals render at the document root and overlay properly. This avoids CSS issues with z-index and stacking contexts.

8. **Context/State Management**: Use React Context or Zustand for global state (e.g. user session, theme). For example, a ThemeContext can store dark/light mode, or a SocketContext can provide a shared WebSocket instance.

9. **Server-Sent Events (SSE)**: As an alternative to WebSockets, SSE lets the server push text-stream events. A client can `new EventSource('/api/stream')` to receive updates.

10. **Stale-While-Revalidate (SWR)**: Use SWR (Stale-While-Revalidate) pattern for data: display cached data immediately and revalidate in background. SWR library helps implement this automatically.

11. **Polling with React Query**: For real-time list updates, set a short refetch interval (e.g. every 5 seconds) to poll the server with React Query or SWR.

12. **Intersection Observer Lazy Load**: Use `react-intersection-observer` to load content or images only when in view, reducing initial load. Good for long feeds or grids.

13. **Mutation Queues**: Queue client actions when offline. For example, store actions in IndexedDB and replay when connection is back.

14. **Optimistic UI with Cache Invalidation**: After a successful mutation via route handler, revalidate relevant SWR/React Query caches so UI updates with new data automatically.

15. **Dynamic Styling via Session/Cookie**: Tailor HTML/CSS based on user state. E.g., apply a CSS class from a theme cookie: `<body className={themeCookie || 'light'}>` to switch site theme without reload.

16. **Progressive Web App (PWA)**: Use service workers to cache assets and enable offline functionality, making the app behave like a native app.

17. **Batching Server Actions**: In Next.js forms with `action={serverAction}`, you can batch multiple mutations in one request and return updated props.

18. **Error Fallback UI**: Show friendly UI when data fetch fails. Use `try/catch` around fetch or SWR error handling to display an error message and retry button.

19. **Multi-Tab Synchronization**: If a user logs in/out or changes setting in one tab, sync it to others via `broadcast-channel` or localStorage events.

20. **Scroll/Route State Preservation**: Use Next.js's new `useRouter` hooks to save scroll position or form state during navigation, improving perceived performance. For example, disable scroll on navigation to a detail view and restore on back.

In each pattern, think about how React and Next.js features work: use Suspense and error boundaries for robustness, dynamic imports for performance, and client/server data strategies (SWR, websockets, server actions) for interactivity. For altering UI based on cookies/session, leverage both server logic (to read cookies at request time) and client hooks to reflect changes instantly.

Tooling and Deployment

Linting, Formatting, TypeScript

Next.js comes with built-in ESLint support. It bundles `eslint-config-next`, a set of recommended rules that catch common mistakes (unused vars, missing dependencies, etc.) [59]. To enable linting, Next.js includes an `.eslintrc.json` by default when you initialize a project with `create-next-app`. Use `npm run lint` to check your code. For formatting, integrate Prettier. You can extend ESLint to enforce Prettier rules (or use a separate Prettier config). In `package.json`, you might have:

```
"scripts": {
  "lint": "next lint",
  "format": "prettier --write ."
}
```

For TypeScript, Next.js auto-generates a `tsconfig.json` if you add a `.ts` file. Set strict flags in `tsconfig` (e.g. `"strict": true`) for type safety. The official ESLint docs note `eslint-config-next` includes React and Next-specific rules [59]. Configure TypeScript in `tsconfig.json` for JSX and include paths as needed.

Turbopack vs Webpack

Next.js traditionally used Webpack. Newer versions offer Turbopack – a Rust-based bundler optimized for incremental rebuilds and performance [60]. Turbopack can significantly speed up dev builds for large projects. To use it, run Next with the `--turbopack` flag:

```
"scripts": {
  "dev": "next dev --turbopack",
  "build": "next build --turbopack"
}
```

[60] [61]. Turbopack is **stable for development**, though production builds are still being finalized. The Next.js docs say: "Turbopack is an incremental bundler... you can use it for a much faster local development experience" [60]. You can switch back to Webpack by omitting the flag (`next dev` uses Webpack). In summary: Turbopack = faster dev builds (with some feature limitations), Webpack = fully stable, battle-tested.

Deploying to Vercel or Self-Hosting

Next.js is made by Vercel, and **Vercel** is the recommended host. Deploying to Vercel is zero-config: after pushing your code to GitHub/GitLab, Vercel auto-detects the Next.js app and builds it [62]. The Vercel dashboard walks you through linking your repo, and you can use default settings (project name, root directory). As their docs state, *"the easiest way to deploy Next.js is to use the Vercel platform (the creators of Next.js)"* [62]. On Vercel you get instant previews for each pull request and automatic SSL.

For self-hosting, you have options: - **Node server**: Build the app (`next build`) and run `next start` on a Node.js server (e.g. AWS EC2 or Heroku). You may export a [Standalone build] or [serverless functions] per Next's guides. - **Docker**: Build a Docker image containing your app. For example, your Dockerfile might install dependencies, run `next build`, and use `CMD ["next", "start"]`. The Self-Hosting docs show how to configure environment variables and use Docker images for consistent environments [63]. - **Static Export**: For fully static sites (no API), you can do `next export` and host on any static host.

If self-hosting, ensure you set environment variables (`DATABASE_URL`, API keys, etc.) on your server or via Docker. Also handle HTTPS and scaling manually. The Self-Hosting guide provides details on these options. In all cases, Next.js handles routing, so your server just serves the Next.js output.

Environment Variables and Security

Next.js supports environment variables out of the box. Use `.env.local` (for local development) and `.env.production` or your hosting platform's env settings for production. By default, only variables prefixed with `NEXT_PUBLIC_` are **exposed to the browser** [56]. For example, `NEXT_PUBLIC_API_BASE` can be read in client code. Other variables (e.g. database passwords) should **NOT** be prefixed – they stay server-only. The docs note: *"To expose a variable to the browser, prefix it with* `NEXT_PUBLIC_`*... it will be inlined into the JS bundle at build time."* [56]. For runtime configuration (Docker or multiple envs), Next.js can use un-prefixed vars in server code via `process.env.VAR`. Remember that build-time inlining means if you build once and deploy same build everywhere, the `NEXT_PUBLIC_` values are fixed at build. Plan envs accordingly.

For security, always use `httpOnly` and `secure` flags on cookies, validate user input to prevent SQL injection (use ORM like Prisma to parameterize queries), and enable HTTPS. Also, sanitize any HTML if you render it. Use Content Security Policy (CSP) headers to mitigate XSS. Next.js has guides on Security and CSP.

Observability (Logging, Error Tracking)

Monitoring and error tracking are critical in production. Integrate tools like **Sentry** or **LogRocket**. For example, Sentry has a dedicated Next.js setup: their docs provide an installation wizard and instructions to capture frontend and backend errors [64]. You typically install `@sentry/nextjs`, configure it (often by running `npx @sentry/wizard`) and then Sentry automatically captures unhandled exceptions and performance traces. In your code, use `try/catch` with `Sentry.captureException(error)` to log errors [65].

For logging, you can use console or a logging library (Winston, pino) on the server, and integrate Sentry's logging. Client-side errors also show up in Sentry. For real-user session tracking or UX analytics, consider LogRocket or FullStory, which record user interactions and replay sessions. Finally, set up performance monitoring (Next.js includes Web Vitals, or use tools like OpenTelemetry) to catch slow pages.

In summary, enable linting/formatting, use TypeScript, choose an appropriate bundler (Turbopack for dev), deploy on Vercel or your platform of choice, manage secrets with env variables, and monitor with tools like Sentry. Citations above point to official docs for ESLint [59], Vercel deployment [62], environment variables [56], and Sentry setup [64].

---

[1] Getting Started: Project Structure | Next.js
https://nextjs.org/docs/app/getting-started/project-structure

[2] [3] File-system conventions: Dynamic Segments | Next.js
https://nextjs.org/docs/app/api-reference/file-conventions/dynamic-routes

[4] [9] [10] Getting Started: Layouts and Pages | Next.js
https://nextjs.org/docs/app/getting-started/layouts-and-pages

[5] [7] [8] Getting Started: Server and Client Components | Next.js
https://nextjs.org/docs/app/getting-started/server-and-client-components

6 Directives: use client | Next.js
https://nextjs.org/docs/app/api-reference/directives/use-client

11 Rendering: Server-side Rendering (SSR) | Next.js
https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering

12 13 Rendering: Static Site Generation (SSG) | Next.js
https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation

14 15 Rendering: Incremental Static Regeneration (ISR) | Next.js
https://nextjs.org/docs/13/pages/building-your-application/rendering/incremental-static-regeneration

16 17 Rendering: Client-side Rendering (CSR) | Next.js
https://nextjs.org/docs/pages/building-your-application/rendering/client-side-rendering

18 19 Routing: Loading UI and Streaming | Next.js
https://nextjs.org/docs/app/building-your-application/routing/loading-ui-and-streaming

20 21 Routing: Route Handlers | Next.js
https://nextjs.org/docs/app/building-your-application/routing/route-handlers

22 23 Routing: Middleware | Next.js
https://nextjs.org/docs/app/building-your-application/routing/middleware

24 31 Getting Started | NextAuth.js
https://next-auth.js.org/getting-started/example

25 26 Credentials | NextAuth.js
https://next-auth.js.org/providers/credentials

27 55 Google | NextAuth.js
https://next-auth.js.org/providers/google

28 29 30 Securing pages and API routes | NextAuth.js
https://next-auth.js.org/tutorials/securing-pages-and-api-routes

32 Client API | NextAuth.js
https://next-auth.js.org/getting-started/client

33 35 36 Using SWR with Debounce: A Step-by-Step Guide | by ruhi chandra | Medium
https://medium.com/@ruhi.chandra14/using-swr-with-debounce-a-step-by-step-guide-1428aed4e6be

34 Real-time Notification System using Next.js and Socket.io | GeeksforGeeks
https://www.geeksforgeeks.org/real-time-notification-system-using-next-js-and-socket-io/

37 Data Fetching: Server Actions and Mutations | Next.js
https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations

38 Setting up your studio | Sanity Docs
https://www.sanity.io/docs/next-js-quickstart/setting-up-your-studio

39 40 41 Displaying content in Next.js | Sanity Docs
https://www.sanity.io/docs/next-js-quickstart/diplaying-content-in-next-js

42 Live Preview with Next.js | Sanity
https://www.sanity.io/blog/live-preview-with-nextjs

43 44 45 46 Functions: cookies | Next.js
https://nextjs.org/docs/app/api-reference/functions/cookies

47 48 49 PostgreSQL Tutorial | GeeksforGeeks
https://www.geeksforgeeks.org/postgresql-tutorial/

50 51 52 Building a Full-Stack CRUD App with Next.js 14, Prisma, and PostgreSQL - DEV Community
https://dev.to/abdur_rakibrony_349a3f89/building-a-full-stack-crud-app-with-nextjs-14-prisma-and-postgresql-b3c

53 54 Common mistakes with the Next.js App Router and how to fix them - Vercel
https://vercel.com/blog/common-mistakes-with-the-next-js-app-router-and-how-to-fix-them

56 Guides: Environment Variables | Next.js
https://nextjs.org/docs/pages/guides/environment-variables

57 Error Boundaries – React
https://legacy.reactjs.org/docs/error-boundaries.html

58 SEO: Dynamic Imports for Components | Next.js
https://nextjs.org/learn/seo/dynamic-import-components

59 Configuration: ESLint | Next.js
https://nextjs.org/docs/app/api-reference/config/eslint

60 61 API Reference: Turbopack | Next.js
https://nextjs.org/docs/app/api-reference/turbopack

62 Pages Router: Deploy to Vercel | Next.js
https://nextjs.org/learn/pages-router/deploying-nextjs-app-deploy

63 Guides: Self-Hosting | Next.js
https://nextjs.org/docs/app/guides/self-hosting

64 65 Next.js | Sentry for Next.js
https://docs.sentry.io/platforms/javascript/guides/nextjs/