# Building a Digital Coupon Book Web App with Next.js 14, Supabase, and Stripe (using Cursor on macOS)

*High-level architecture of a Next.js app integrating Supabase (Postgres + Auth) and Stripe payments*[*[1]*](https://github.com/vercel/nextjs-subscription-payments#:~:text=,subscription%20statuses%20via%20Stripe%20webhooks)*.*

This step-by-step guide will walk you through creating a **digital coupon book web app** using Next.js 14 (App Router) with Tailwind CSS, Supabase (Postgres database with authentication, storage, and RLS security), and Stripe Checkout for payments. We will use **Cursor Premium** (an AI-enabled code editor) on macOS to accelerate development, with tips on how to leverage Cursor’s AI prompts at each stage. By the end, you’ll have a full-stack web app where users can browse coupon books, purchase them via Stripe, redeem coupons, and where admins can manage schools, coupon books, and track sales – plus the app will be installable as a PWA.

**Table of Contents:**

1. [Project Setup on macOS with Cursor](#project-setup)
2. [Bootstrapping Next.js 14 + Tailwind CSS](#bootstrapping-nextjs)
3. [Setting Up Supabase (Database, Auth & Storage)](#setup-supabase)
4. [Integrating Stripe Checkout (Payments & Webhooks)](#integrating-stripe)
5. [Implementing User Flows: Catalog, Purchase, Redeem, Verify](#user-flows)
6. [Implementing Admin Views: School, Book, Coupon & Sales Management](#admin-views)
7. [Adding PWA Support (Manifest, Service Worker, A2HS)](#pwa-support)
8. [Environment Variables & Deployment (e.g. Vercel)](#env-deploy)
9. [Security Considerations (RLS, Webhooks, Token Validation)](#security)

Throughout the guide, **Cursor prompts** will be suggested – you can enter these in Cursor’s AI chat or instruction panel to generate code or perform actions automatically, speeding up development.

<a id="project-setup"></a>

## 1. Project Setup on macOS with Cursor

**Prerequisites:** Ensure you have **Node.js** (preferably LTS version 18+ or 20) installed on your Mac. You can install it via Homebrew (brew install node) or from the Node.js website. Confirm by running node -v in Terminal. Also, install **Git** if not already present (most macOS have it, or install via Xcode Command Line Tools).

**Step 1: Create a Project Folder.** On your Mac desktop (or any directory of your choice), create a new folder for the project. For example, open **Terminal** and run:

mkdir ~/Desktop/coupon-book

This will create a folder named “coupon-book” on your Desktop. You can also do this in Finder (Right-click Desktop → New Folder). This folder will hold all your project files.

**Step 2: Open the Folder in Cursor.** Launch the **Cursor Premium** app on your Mac. Use *File → Open Folder* and select the coupon-book folder you created. Cursor will now treat this as the project workspace. (If Cursor has a command palette, you could also use that to open a folder.)

**Step 3: Initialize a Next.js App via Cursor’s Terminal.** Cursor includes an integrated terminal or you can open your macOS Terminal. In Cursor, open a Terminal panel (for example, via the “Terminal” option or a keyboard shortcut in Cursor). In the terminal, run the Next.js boilerplate creation command using **Create Next App**:

npx create-next-app@latest --typescript --tailwind coupon-book

This uses Next.js’s official CLI to bootstrap a new project named “coupon-book” in the current directory, with TypeScript and Tailwind CSS configured by default[[2]](https://nextjs.org/docs/app/api-reference/cli/create-next-app#:~:text=%60,linter%20%60Skip%20linter%20configuration). (The --tailwind flag automatically sets up Tailwind CSS configuration for us[[2]](https://nextjs.org/docs/app/api-reference/cli/create-next-app#:~:text=%60,linter%20%60Skip%20linter%20configuration), and --typescript ensures the project is in TypeScript.) If prompted with options (for example, to use src/ directory or import alias), you can accept defaults or choose as preferred.

**Cursor Prompt:** *“Run `npx create-next-app@latest --typescript --tailwind coupon-book` in the project folder.”* – You can instruct Cursor to execute this command if it supports running terminal commands via AI instructions. Otherwise, run it manually in the terminal.

After the script finishes, you should see a success message. In Cursor’s file explorer, the folder will now contain a Next.js project (pages, app directory, package.json, etc.).

**Step 4: Verify the Dev Server.** Still in the terminal, run npm run dev (or yarn dev) to start the Next.js development server on localhost:3000. Open http://localhost:3000 in your browser; you should see the default Next.js welcome page. This confirms the setup is working. Stop the server with Ctrl+C for now.

**Tip:** In Cursor, you can split the view to see code and terminal together. As you proceed, keep the dev server running to live-preview changes, restarting it when needed (e.g., after installing new packages or changing config).

We now have a Next.js 14 project ready, with the App Router structure (using the /app directory) and Tailwind CSS integrated. Next, we will start customizing this project.

<a id="bootstrapping-nextjs"></a>

## 2. Bootstrapping Next.js 14 with Tailwind CSS

Our project was created with Tailwind CSS configuration thanks to the --tailwind flag. Let’s verify and tweak the initial setup:

* **Tailwind Config:** Open tailwind.config.js in Cursor. It should already include the content paths for your Next.js app (like ./app/\*\*/\*.{js,ts,jsx,tsx}) and the default theme. The Create-Next-App CLI adds this for you. For example, it initializes a Tailwind config, PostCSS config, and includes Tailwind’s base styles in the globals CSS[[2]](https://nextjs.org/docs/app/api-reference/cli/create-next-app#:~:text=%60,linter%20%60Skip%20linter%20configuration). No additional changes should be needed unless you want to customize the theme or add plugins.
* **Global Styles:** Open app/globals.css. You should see Tailwind’s directives (@tailwind base; @tailwind components; @tailwind utilities;) already present, meaning Tailwind is wired up. You can customize this file to add any global CSS or overrides. Tailwind’s utility classes will be available in your JSX.
* **App Router Structure:** The project likely contains an app/page.tsx (the homepage) and maybe an app/layout.tsx. Next.js 14 uses the App Router by default, which means **server components** and nested layouts are available. We will use this new architecture for our pages (e.g., catalog, checkout, admin, etc.) instead of the legacy pages/ directory.

Now, let’s start building out the main parts of our app’s frontend:

**Homepage and Catalog:** In our coupon book app, the homepage could simply redirect to or show the **catalog** of available coupon books. The catalog is a list of coupon books that users can browse. We’ll create a new route for this:

* Create a file at app/catalog/page.tsx. This will be the catalog page listing all coupon books available for purchase. Initially, you can populate it with dummy content or a placeholder message. Later, we will fetch real data from Supabase.

**Cursor Prompt:** *“Create a new Next.js page at app/catalog/page.tsx that displays a heading 'Available Coupon Books' and a placeholder list.”* – Cursor can generate a basic React component for the catalog page.

For example, a simple version of app/catalog/page.tsx might look like:

export default function CatalogPage() {  
 return (  
 <main className="p-6">  
 <h1 className="text-2xl font-bold mb-4">Available Coupon Books</h1>  
 {/\* TODO: List coupon books here \*/}  
 <p>Loading books...</p>  
 </main>  
 );  
}

We’ll enhance this to list real books once Supabase is connected.

**Navigating to Catalog:** Decide how users reach the catalog. You could make the catalog page the home page, or have the home page link to it. To keep things simple, let’s use the home page to redirect to /catalog. In app/page.tsx (the root page), replace the default content with a navigation:

import { redirect } from 'next/navigation';  
export default function Home() {  
 // Redirect homepage to /catalog  
 redirect('/catalog');  
}

This uses Next.js 13+ redirect to send users to the catalog page automatically. (Alternatively, you could remove this and simply build the home page as the catalog itself.)

**Note:** Next.js 14 App Router supports **Server Actions** and form data handling, but our use case can stick to standard fetches and API routes for clarity. We will use API Route handlers for Stripe webhooks and perhaps some protected actions, while using Supabase client for direct database queries in most pages.

**Tailwind UI Components:** As you build UI, utilize Tailwind classes to style elements (Cursor’s autocomplete can help with class suggestions). For complex components like forms or modals, you might consider using pre-built components or a library like **shadcn/ui** (an open-source collection of accessible components styled with Tailwind). The Next.js SaaS starter kit uses shadcn/ui for a quick UI setup[[3]](https://github.com/nextjs/saas-starter#:~:text=Get%20started%20quickly%20with%20Next,Postgres%2C%20Stripe%2C%20and%20shadcn%2Fui), which could be helpful for admin forms, but you can also create your own.

**Cursor Prompt:** *“Using Tailwind CSS, style the catalog page with a grid of cards for each coupon book (title, price, and image placeholder).”* – Cursor can generate a styled JSX structure for a card component. You’ll fill it with real data later.

**Routing Structure:** Plan the routes you’ll need for user flows: - /catalog – lists all books. - /books/[id] – a detail page for a specific coupon book, with purchase option. - /success or /purchase-success – a page Stripe will redirect to after a successful purchase. - /account (optional) – user account page listing their purchased books. - /redeem/[couponId] or similar – page to redeem a specific coupon (or we might handle redemption within the book detail page). - /verify – a page for merchants/admins to verify a coupon code (could be a simple form). - /admin/\* – routes for admin interface (e.g., /admin/books, /admin/schools, etc.).

We will create these pages (mostly as placeholders initially) so we have the structure in place: - For dynamic routes like [id], create a folder and page.tsx inside (e.g., app/books/[id]/page.tsx). - For grouping admin routes, you can use an app/(admin)/... route group with its own layout if needed, or simply an /admin folder.

Go ahead and scaffold these with basic content:

**Cursor Prompt:** *“Generate placeholder pages for book detail (app/books/[id]/page.tsx), purchase success (app/success/page.tsx), and admin dashboard (app/admin/page.tsx).”*

For example: - app/books/[id]/page.tsx might render a heading "Book Details for [id]" for now. - app/success/page.tsx can say "Thank you for your purchase!". - app/admin/page.tsx can have links to sub-pages like "Manage Schools", "Manage Books", etc., which we will implement later.

Using Cursor to generate these boilerplate pages saves time. We will fill them in future steps.

**Verify Tailwind is Working:** Add a distinctive Tailwind class to an element (for example, give the success page heading a text-green-600 class). Run npm run dev and navigate to that page to ensure the style applies. If you see the color, Tailwind is functioning correctly. If something isn’t styled, make sure the class names are correct and the content paths in tailwind.config.js include the directories you’re using (the default config covers app/\*\*/\*.{js,ts,jsx,tsx}, which is correct for App Router).

With our Next.js app structure in place and Tailwind ready, it’s time to set up the backend resources on **Supabase** for our database, authentication, and storage.

<a id="setup-supabase"></a>

## 3. Setting Up Supabase (Database, Auth & Storage)

Supabase will provide our app with a Postgres database (for storing coupon books, coupons, purchases, etc.), authentication (user sign-up/login), and storage (for any images or files, if needed). We’ll also configure **Row Level Security (RLS)** policies to protect data.

**Step 1: Create a Supabase Project.** Go to [supabase.com](https://app.supabase.com) and sign up or log in. Create a new project (choose a name, region, and database password). Once created, in your project dashboard, find the **API** settings (usually under Settings -> API). Note the **Project URL** and **Anon API Key** – we’ll need these for our app to connect to Supabase.

**Step 2: Add Supabase Credentials to the App.** In the Next.js project, create a file named .env.local at the root (if not already present). Add the following environment variables (from your Supabase project):

NEXT\_PUBLIC\_SUPABASE\_URL=<your-supabase-project-url>  
NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY=<your-supabase-anon-key>

These will be used by the Supabase client on the frontend. (The NEXT\_PUBLIC\_ prefix means they will be exposed to the browser, which is fine for the anon key and URL.)

Also, generate a **service role key** from Supabase (in the API settings page, it’s listed as “service\_role”). Add it to the env file without the NEXT\_PUBLIC\_ prefix, since it must remain secret on the server:

SUPABASE\_SERVICE\_ROLE\_KEY=<your-supabase-service-role-key>

We will use the service role key on the server (Node.js) for sensitive operations like handling Stripe webhooks or verifying coupon codes, because the service role can bypass RLS restrictions[[4]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Bypassing%20access%20controls). **Never expose the service role key to the client** or commit it to git, as it grants full access to your database (we’ll only use it in secure server-side code)[[5]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=If%20you%20exclusively%20use%20Storage,access%20to%20all%20Storage%20APIs).

**Step 3: Install Supabase JS SDK.** In the project terminal, install the Supabase client library:

npm install @supabase/supabase-js

This library allows our Next.js app to interact with Supabase (for queries, auth, etc.)[[6]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=To%20integrate%20Supabase%2C%20install%20the,necessary%20Supabase%20client%20library).

**Step 4: Initialize Supabase Client.** In your Next.js app, create a helper for Supabase. For example, create a file lib/supabaseClient.ts (or .js if not using TS) with the following:

import { createClient } from '@supabase/supabase-js';  
  
const supabaseUrl = process.env.NEXT\_PUBLIC\_SUPABASE\_URL!;  
const supabaseAnonKey = process.env.NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY!;  
  
export const supabase = createClient(supabaseUrl, supabaseAnonKey);

This uses the URL and anon key from your env to create a Supabase client instance[[7]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=%2F%2F%20lib%2FsupabaseClient.js%20import%20,js). We export it for use in our pages and API routes. (The ! non-null assertions assume these env vars are set; ensure you have the .env.local loaded in development by restarting npm run dev after adding it.)

Now we can use supabase to make calls. For example, supabase.from('coupon\_books').select('\*') would fetch data from a table (once we create the table). We’ll use this in our React components or Route Handlers as needed.

**Step 5: Set Up Database Schema.** Next, define the database tables needed for our app. We can do this via the Supabase **Table Editor** (GUI) or via SQL. For a complex schema, writing SQL and running it in Supabase’s SQL editor (or using their migration tools) is reliable. Here are the tables we need and their key fields:

* **schools**: Represents a school or organization that issues coupon books.
* id (uuid, primary key) – unique identifier.
* name (text) – school name.
* (Add any other relevant fields like address, etc. as needed.)
* **coupon\_books**: A coupon book that users can buy.
* id (uuid, primary key).
* school\_id (uuid, foreign key to schools.id).
* title (text) – name of the coupon book (e.g. “Fall 2025 Coupon Book”).
* price (numeric or integer) – price in cents or dollars (consistent with Stripe; if using Stripe Price ID, this might be stored separately).
* year or season (optional text) – e.g. “2025” or “Fall 2025”, if multiple per school.
* stripe\_price\_id (text) – store the Stripe Price ID associated with this book’s purchase (we’ll get this when configuring Stripe).
* created\_at (timestamp).
* Perhaps cover\_image\_url (text) – a URL to an image (if we use Supabase storage for covers).
* **coupons**: Individual coupons within a book.
* id (uuid, primary key).
* book\_id (uuid, foreign key to coupon\_books.id).
* title or description (text) – e.g. “50% off at Store X”.
* expires\_at (date or timestamp, optional) – expiration date of that coupon.
* max\_redemptions (int, default 1) – how many times this coupon can be redeemed *per user or in total* (for physical coupon books, typically 1 per user).
* We might not store a static code here, since each user’s redemption will generate a unique code. But if needed, could have a base code or vendor code.
* **purchases**: Records of users purchasing coupon books.
* id (uuid, primary key).
* user\_id (uuid) – the Supabase Auth user ID of the buyer.
* book\_id (uuid) – which coupon book was purchased.
* stripe\_session\_id (text) – the Stripe Checkout Session ID for this purchase (to reconcile webhooks).
* paid (boolean) – whether the payment was completed. (We’ll set this true via Stripe webhook.)
* purchase\_at (timestamp).
* **redemptions**: Records of a user redeeming a coupon.
* id (uuid, primary key).
* user\_id (uuid) – the user who redeemed (should match the purchaser).
* coupon\_id (uuid) – the coupon used.
* redeemed\_at (timestamp).
* verify\_code (text) – a one-time random code or token generated at redemption time for verification.
* verified (boolean) – whether a staff/merchant has verified this redemption.
* verified\_at (timestamp, nullable) – when it was verified (if at all).

This schema allows us to track everything: schools can have many coupon books, books have many coupons, users buy books (purchases), and users redeem coupons (redemptions).

You can create these tables using Supabase’s interface (going to **Database → Tables → New Table** for each). Ensure each table has a primary key (Supabase can generate UUIDs by default if you set the column default to gen\_random\_uuid()). Set up foreign keys for data integrity (Supabase Table Editor lets you mark a column as foreign key referencing another table).

**Cursor Prompt (SQL):** *“Generate SQL statements to create tables for schools, coupon\_books, coupons, purchases, redemptions with appropriate foreign keys.”* – You can ask Cursor to draft the CREATE TABLE statements. Copy the output into Supabase SQL editor and run it, or use it as reference in the Table Editor.

**Step 6: Supabase Auth Configuration.** By default, Supabase comes with Email/Password authentication ready. We don’t need to do much to enable it – the supabase.auth.signUp and signIn functions (or magic links, OAuth, etc.) all work out of the box. In our app, we’ll likely have a simple email/password sign-up for users to purchase books. Supabase also provides the auth.users table where user profiles are stored.

We might want a **profiles** table for additional user info (like name, and whether they are admin). Supabase can auto-fill it via triggers. But for simplicity, we could just use the auth.user() data in our app as needed. If we need to mark some users as admins, a simple approach is to maintain a profiles table with an is\_admin boolean or a role field, and use Supabase’s **Policy** to grant admin rights. Alternatively, maintain a separate list of admin user IDs in an env var or config (not scalable, but quick). We’ll address admin determination later in the Admin section.

**Step 7: Enable Row Level Security (RLS).** By **default, Supabase enables RLS on new tables** created through their interface[[8]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=you%20enable%20RLS). Double-check each table’s settings: in Supabase dashboard, go to Database → Table → click the table, and ensure **RLS is toggled ON**. If any were created via SQL, you may need to manually enable RLS with a command like ALTER TABLE purchases ENABLE ROW LEVEL SECURITY;[[9]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=RLS%20must%20always%20be%20enabled,schema). RLS is crucial: when enabled, **no data is accessible via the anon (public) key until you create policies** to allow it[[10]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=1). This prevents accidental exposure of data.

Now, define **policies** for each table to enforce who can read or write data. Some common policies for our use case:

* **Public data (if any):** For example, the list of coupon books might be okay to read without login (so users can browse before signing up). If so, we can create a policy on coupon\_books to allow select for anon (unauthenticated) role[[11]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=11). Alternatively, we can require login to view catalog – but that’s a product decision. For now, let’s allow anyone to read books and schools info:
* Policy on schools for select: allow for role anon using TRUE (all rows visible)[[11]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=11).
* Policy on coupon\_books for select: allow for anon using TRUE. *(Later, if we have sensitive fields like cost or internal notes, we might restrict or use a view.)*
* **Purchases:** Only authenticated users should interact with purchases. A user should only be able to see **their own** purchases:
* Policy for purchases on select: to role authenticated, using user\_id = auth.uid()[[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id) (each user sees only rows where the user\_id matches their Supabase auth ID). This is a common pattern: comparing a table’s user\_id to the Supabase session’s user ID[[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id).
* Policy for purchases on insert: allow when user\_id = auth.uid() (so users can create their own purchase records – though in our flow, we might insert purchases server-side after Stripe payment, using service role bypass).
* Probably *no* update or delete by clients – purchases, once created, aren’t edited by the user (maybe an admin could refund/delete via a separate mechanism, but that would be server-side).
* **Coupons:** Users should only see coupons for books they have purchased. We can enforce this by a more complex policy or simply not exposing coupons except through a server call that checks purchase. However, to leverage RLS, one approach:
* Add a **join table or view** for “user\_coupons” where each user gets entries for each coupon of books they bought. Alternatively, make a select policy on coupons like: EXISTS (SELECT 1 FROM purchases p WHERE p.book\_id = coupons.book\_id AND p.user\_id = auth.uid() AND p.paid IS TRUE) – This would allow a user to select coupons of books they bought. We can implement that in a policy (Supabase allows subqueries in policies). This ensures users can’t fetch coupons for books they didn’t buy.
* Policy for coupons on select: to authenticated, with the condition above.
* No need for insert/update on coupons by normal users (that’s admin’s domain).
* **Redemptions:**
* For users: allow inserting a redemption when user\_id = auth.uid() *and* ensure the coupon belongs to a book the user owns (again a subquery or perhaps check via a function). This is a bit complex in policy form; alternatively, we handle redemption inserts via our Next API with service role after verifying logic.
* Allow select on redemptions for authenticated where user\_id = auth.uid() (so users can see their own redemption history if needed).
* For admin/merchants verifying codes, if they are not logged in as the user, we might handle that separately (likely via a server function that uses service role to find any redemption by code).
* **Admins:** For tables like schools, coupon\_books, coupons, ideally only admins should be able to insert or update them:
* We can create an admin Postgres role and assign certain users to it, or simpler, check against an is\_admin flag in a profiles table. Supabase JWT can include custom claims if we had set that up; without it, a straightforward way is to maintain an admins table of user IDs and write policies like auth.uid() in (select id from admins).
* For example, on coupon\_books for insert/update/delete: to role authenticated using auth.uid() = ANY(SELECT user\_id FROM admins) (if you had an admins table). Or check a is\_admin flag via a join.
* As an MVP, you might skip formal admin RLS and instead not expose those operations on the client at all except through a secure admin interface running server-side with service key. But it’s good to plan RLS for defense in depth[[13]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=RLS%20is%20a%20Postgres%20primitive,party%20tooling).
* We’ll assume one “global admin” role for simplicity in this guide.

To create policies, use Supabase UI “Authentication → Policies” or the SQL editor. For example, to add the policy that users can view their own todos, Supabase docs give:

create policy "Individuals can view their own todos."  
on todos for select  
using ((auth.uid()) = user\_id);

This concept applied to our purchases table would be: each user can select their own purchases[[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id). Similarly for other tables with user\_id.

**Cursor Prompt (Policies):** *“Write RLS policy SQL for the purchases table so that users can only select or insert their own records (matching user\_id to auth.uid()).”* – This can produce a snippet that you adjust for other tables.

Make sure to **test your RLS policies**: Supabase provides a “Test out policies” feature where you can simulate a user’s JWT to see what they can access, or you can write a quick script using the supabase JS client logged in as a user to verify that forbidden data is indeed inaccessible. RLS will ensure, for example, that even if a user manipulates the client code or API calls, the database will not return someone else’s coupons or purchases.

**Step 8: Supabase Storage (optional for images/files).** If you plan to have images (like a cover image for each coupon book, or perhaps images for coupons), Supabase Storage can be used. It’s basically an S3-like object storage integrated with the database and RLS.

To use it, go to **Storage** in Supabase dashboard: - Create a **Bucket** (e.g., named covers or public). Decide if it’s public or requires auth. For user-uploaded images that should only be seen by that user, keep it private and use RLS. For public assets (like a coupon book cover that anyone can see), you can make the bucket public or provide signed URLs. - If private, set up policies on the storage.objects table similar to other tables. For instance, Supabase by default **does not allow any uploads without a policy**[[14]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Access%20policies)[[15]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=By%20default%20Storage%20does%20not,table). You might add a policy to allow inserts for authenticated users to a specific bucket. Example from Supabase docs for storage: to allow users to read files they uploaded, a policy can match the object’s owner to auth.uid()[[16]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Allow%20a%20user%20to%20access,uploaded%20by%20the%20same%20user). Or restrict by folder naming convention (e.g., only allow uploads to a folder named with their user ID)[[17]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=8). - For simplicity, if using a public bucket for book covers, you can disable RLS on that bucket (public means no auth needed to read). If you want only admins to upload there, you could create an admin policy similarly.

In our app, maybe each coupon book could have a cover image in storage. We’d store its URL (which would include the bucket path) in the coupon\_books.cover\_image\_url. Admin UI would handle uploading the file (using Supabase storage JS SDK or using Cursor to drag/drop an image into a known folder).

If you do use storage, install the storage SDK: npm install @supabase/storage-js (though supabase-js includes it internally). You can then use supabase.storage.from('bucket').upload('path', file) in your code. Ensure the user has rights or use service key for admin operations.

**Step 9: Connect Supabase Auth in the App.** We should allow users to sign up/log in. We have a few options: - Build our own simple email/password form and use supabase.auth.signUp / signIn on submit (as shown in a Medium snippet[[18]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=%2F%2F%20app%2Fsignup%2Fpage.js%20import%20,lib%2FsupabaseClient)[[19]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=if%20%28error%29%20,)). This is straightforward. We need to consider confirming email (Supabase can send confirmation emails by default). - Use Supabase’s prebuilt Auth UI (they have an @supabase/auth-ui-react package that provides <Auth/> component for a quick start). Alternatively, use NextAuth (but since we already have Supabase auth, we don’t need NextAuth). - For now, let’s implement a basic form for sign up and login.

Create pages for authentication: - app/login/page.tsx - app/signup/page.tsx

On the sign-up page, we’ll have a form that collects email and password, and on submit calls supabase.auth.signUp({ email, password }). The Medium article example for signup[[18]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=%2F%2F%20app%2Fsignup%2Fpage.js%20import%20,lib%2FsupabaseClient)[[20]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=if%20%28error%29%20,) is a good reference. Similarly for login (signIn).

Make sure to handle error and success (maybe redirect to catalog after login, etc.). Supabase will also keep the session and set a cookie by default, so subsequent supabase calls include the user’s JWT.

**Cursor Prompt:** *“Generate a Next.js server component for app/signup/page.tsx with a form that signs up a user via supabase.auth.signUp and then alerts or redirects on success.”* – It might provide a client component example (since signUp requires client-side). You could mark the component 'use client' at top to use hooks and handle form on client side.

**Testing Auth:** Start your dev server and try signing up a new user. Then check the Supabase Auth > Users table to confirm the user is created. You might need to verify the email depending on Supabase settings (you can disable email confirmations in Auth settings for testing). For now, to keep things simple, you could disable the email confirmation requirement or use the magic link that is sent to your email to confirm.

Finally, **update the app UI** to account for auth state: - Show login/signup links for non-authenticated users, and a logout button for logged-in users. You can use supabase.auth.getUser() or the provided Auth context if any. Supabase JS v2 also has a createBrowserClient for Next 13; but our basic usage is fine. - Consider using middleware.js in Next.js to protect certain routes (like /account or any /admin routes) by redirecting if not logged in. This can be done by reading the supabase.auth.session cookie or calling an endpoint. However, full middleware integration with Supabase may require a bit more setup (Supabase provides examples of Next.js middleware for auth).

At this point, our app can connect to the database. We have the schema ready and the client configured. Next, we will integrate **Stripe** to handle payments for purchasing the coupon books.

<a id="integrating-stripe"></a>

## 4. Integrating Stripe Checkout (Payments & Webhooks)

Now we’ll set up **Stripe** for handling payments when a user purchases a coupon book. We’ll use **Stripe Checkout** for a simple, secure payment flow, and set up a **webhook** to be notified of completed payments (so we can fulfill the purchase by granting access to the coupons).

**Step 1: Set Up Stripe Account & API Keys.** Create or log in to your [Stripe Dashboard](https://dashboard.stripe.com). Ensure you are in *Test Mode* (toggle in the top right corner of the dashboard – it should be orange and say “Test Mode”).

* Get your **Stripe API keys** in Developers -> API Keys. Copy the **Publishable key** (starts with pk\_test\_...) and **Secret key** (sk\_test\_...). Add them to your .env.local:

NEXT\_PUBLIC\_STRIPE\_PUBLISHABLE\_KEY=<your-publishable-key>  
STRIPE\_SECRET\_KEY=<your-secret-key>

The publishable key can be exposed to the client (for Stripe.js), the secret key must stay on server. The environment file now contains Stripe and Supabase keys (remember never to commit this file).

* Also, we will need a **Webhook Signing Secret** in a moment, but we’ll get that after creating a webhook endpoint in Stripe.

**Step 2: Install Stripe SDK.** We need Stripe’s libraries:

npm install stripe @stripe/stripe-js

This installs both the Node.js Stripe library (for server-side) and the Stripe.js library (for client-side). We already listed these in our earlier step, but if not done, do it now[[21]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=Next%2C%20install%20the%20Stripe%20package,to%20manage%20payments).

**Step 3: Create Stripe Products/Prices for Coupon Books.** There are two ways to charge for a coupon book: 1. **Pre-create Products in Stripe:** You can log in to Stripe Dashboard -> Products, and create a product for each coupon book, each with a Price (one-time price). Then you have stable price IDs to use in Checkout. This is good if the product list is small or changes infrequently (like yearly coupon books). You’d store the Stripe price\_id in your coupon\_books table so the app knows which price to use. 2. **On-the-fly Checkout Sessions:** You can create a Checkout Session with line items that specify amount and description directly. This is simpler if you don’t want to pre-manage products on Stripe, but it is less integrated with Stripe’s product catalog. Given that our books are distinct products, it’s reasonable to pre-create them in Stripe or create them via API at admin time.

For this guide, let’s assume we’ll **store a Stripe Price ID in each coupon\_book**. So do the following for each coupon book currently available (if you have some dummy data): - In Stripe Dashboard -> Products: “Add product”. Name it e.g. “Fall 2025 Coupon Book – [School Name]”. Set “One time” price and the amount (e.g. $20.00 or whatever). Save it. - Copy the Price ID (it will look like price\_XXXXXXXXXXXXXXXX) and update that coupon\_book’s stripe\_price\_id field in the database. (If you haven’t inserted any books yet, you can do so now via Supabase Table Editor or SQL, and include the price ID.)

If you prefer to do this in code later, you could use Stripe API to create products, but manual is fine for now.

**Step 4: Checkout Session API Route.** In Next.js App Router, we create an API route to initiate the Stripe Checkout. Create a file: app/api/checkout/route.ts. This will handle POST requests when a user wants to purchase a book.

Inside, we’ll use our Stripe secret key to create a checkout session. For example:

import { NextResponse } from 'next/server';  
import Stripe from 'stripe';  
  
const stripe = new Stripe(process.env.STRIPE\_SECRET\_KEY!, {  
 apiVersion: '2022-11-15'  
});  
  
export async function POST(request: Request) {  
 const { bookId } = await request.json();  
 // Look up the coupon book in DB to get price and title (optional)  
 // For security, you might fetch the stripe\_price\_id from Supabase instead of trusting client.  
 // But we'll assume the request sends bookId which we use to find price.  
  
 // (Pseudo-code: fetch from supabase)  
 // const { data: book } = await supabase.from('coupon\_books').select('stripe\_price\_id, title').eq('id', bookId).single();  
 // if (!book) return NextResponse.json({ error: 'Book not found' }, { status: 404 });  
  
 try {  
 const session = await stripe.checkout.sessions.create({  
 payment\_method\_types: ['card'],  
 line\_items: [  
 { price: /\*book.stripe\_price\_id\*/ 'price\_xxx', quantity: 1 }  
 ],  
 mode: 'payment',  
 success\_url: `${process.env.NEXT\_PUBLIC\_SITE\_URL}/success?session\_id={CHECKOUT\_SESSION\_ID}`,  
 cancel\_url: `${process.env.NEXT\_PUBLIC\_SITE\_URL}/catalog`  
 });  
 return NextResponse.json({ sessionId: session.id });  
 } catch (err: any) {  
 console.error(err);  
 return NextResponse.json({ error: err.message }, { status: 500 });  
 }  
}

This is a rough idea. Key points: - We initialize Stripe with our secret key. - We parse the request JSON to get a bookId. - (Optionally, we fetch the book from our DB to retrieve the stripe\_price\_id and maybe verify that the book is available, etc. We could use our Supabase client here, but note: this code runs on the server side in a Route Handler. We have access to environment variables and can even use the service key if needed. For a simple check, using the anon key to fetch the public info is fine.) - We call stripe.checkout.sessions.create({...}) with payment\_method\_types, the line item (using the Stripe Price ID and quantity 1)[[22]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=try%20,process.env.NEXT_PUBLIC_DOMAIN%7D%2Fcancel%60%2C), and set mode: 'payment' for one-time charge[[22]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=try%20,process.env.NEXT_PUBLIC_DOMAIN%7D%2Fcancel%60%2C). - success\_url and cancel\_url: We direct to our app. success\_url includes a query parameter session\_id={CHECKOUT\_SESSION\_ID} so Stripe will replace it with the actual session ID. We’ll handle that on the success page. We use an env NEXT\_PUBLIC\_SITE\_URL for our domain (in dev this might be http://localhost:3000 – make sure to set that in your .env.local as well). Alternatively, Next 13+ can use headers().get('host') to build the URL dynamically for dev vs production. - We return the session.id to the client in JSON.

Make sure to add NEXT\_PUBLIC\_SITE\_URL=http://localhost:3000 to your env for development (and update it to your prod URL when deploying)[[23]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=In%20the%20,set%20up%20for%20your%20URLs).

**Security:** We trust the bookId from client here, but we do a DB lookup. This prevents a malicious user from providing a fake price ID or manipulating amount, since we only use the ID we have on record for that book. Always validate on the server side rather than using a client-supplied price directly.

**Cursor Prompt:** *“Implement the POST handler in app/api/checkout/route.ts to create a Stripe Checkout Session for a given book ID.”* – Cursor can write this, just remember to adjust the price fetching logic and env names appropriately. Also consider that NextResponse is used in App Router instead of the older res.status().json() style.

**Step 5: Trigger Checkout from Client.** On the book detail page (or wherever the user clicks “Buy”), we need to call this API and then redirect to Stripe. We’ll use **Stripe.js** on the client.

In app/books/[id]/page.tsx, or better, in a client component within it (because we will handle a click event), do something like:

'use client';  
import { loadStripe } from '@stripe/stripe-js';  
  
const stripePromise = loadStripe(process.env.NEXT\_PUBLIC\_STRIPE\_PUBLISHABLE\_KEY!);  
  
export default function BookDetail({ params }) {  
 const bookId = params.id;  
 const handlePurchase = async () => {  
 const stripe = await stripePromise;  
 const res = await fetch('/api/checkout', {  
 method: 'POST',  
 headers: { 'Content-Type': 'application/json' },  
 body: JSON.stringify({ bookId })  
 });  
 const data = await res.json();  
 if (data.sessionId) {  
 await stripe!.redirectToCheckout({ sessionId: data.sessionId });  
 } else {  
 alert(`Error: ${data.error || 'Unable to create checkout session'}`);  
 }  
 };  
  
 return (  
 <main>  
 {/\* ... display book info ... \*/}  
 <button onClick={handlePurchase} className="btn">Buy This Book</button>  
 </main>  
 );  
}

This code uses Stripe.js’s redirectToCheckout with the session ID returned from our API[[24]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=method%3A%20%27POST%27%2C%20headers%3A%20%7B%20%27Content,). It will redirect the user to Stripe’s checkout page. The user completes payment there, and Stripe will then redirect back to our success\_url (which we set to our-site/success?session\_id=...).

We should ensure only logged-in users can buy. If supabase.auth.session() is not present, you might redirect to login instead of calling the API. For brevity, assume the user is logged in (we can enhance with a check and perhaps an enforced auth via Next middleware or a check at handlePurchase to redirect to login if not).

**Step 6: Create a Success Page Handler.** When the user is redirected back from Stripe, they land on, say, /success?session\_id=cs\_test\_a1b2c3.... We need to use that session\_id to confirm the payment and show the user a confirmation (and maybe details like “you’ve purchased X book, you can now access it”).

We have two ways to confirm the payment: - Use Stripe’s client or server API to retrieve the session by ID and check payment\_status. But this requires our secret key on server or publishable key with the appropriate permission (Stripe.js can’t retrieve a CheckoutSession by ID on the client, as far as I know, so we’d do it on server via another API route). - Use our own database/webhook as the source of truth: The webhook (coming next) will mark the purchase as paid in our DB. The success page can simply read from our DB to ensure the purchase exists. This requires that the webhook has fired by the time the user is on the success page (which is likely but not guaranteed if the redirect is very fast). - Simpler: On success page, just show a generic “Payment succeeded!” and instruct user to go to their account or book page to see content. Then rely on the fact that by the time they navigate there, our DB has the purchase.

A hybrid approach: success page could call a Next API route that uses the Stripe API to retrieve the session (using secret key on server) to verify it, and possibly use the session metadata to create the purchase. But doing logic at webhook is preferred to decouple from user being online.

For this guide, we will primarily handle fulfillment via webhook. So the success page just thanks the user and maybe asks them to check their “My Books” section.

So, implement app/success/page.tsx:

'use client';  
import { useSearchParams } from 'next/navigation';  
  
export default function SuccessPage() {  
 const searchParams = useSearchParams();  
 const sessionId = searchParams.get('session\_id');  
  
 return (  
 <main className="p-6">  
 <h1 className="text-xl font-bold text-green-600">Payment Successful!</h1>  
 <p>Thank you for your purchase. Your coupon book has been unlocked.</p>  
 {/\* Optionally, show details if you want to fetch using sessionId \*/}  
 <a href="/account" className="underline">Go to My Books</a>  
 </main>  
 );  
}

This is a basic page. If you wanted, you could trigger a fetch to our DB to load the book details just purchased (maybe using sessionId to identify it). But since we can also get that from state or Supabase by querying purchases of the current user, it might be easier to show it on their account page.

**Step 7: Stripe Webhook for Purchase Fulfillment.** This is critical. We need Stripe to notify our app when a payment is completed (or failed, etc.). We’ll set up a webhook endpoint in Next.js to receive events like checkout.session.completed. In those events, we’ll: - Verify the event is legitimate (using Stripe’s signature header and our webhook secret). - Parse the event and if it’s a successful payment, mark the corresponding purchase in our DB as paid (or create the purchase record if we hadn’t already).

Create a route: app/api/stripe/webhook/route.ts. This will handle Stripe webhook requests. **Note:** Next.js (since v12) requires special handling to get raw request body (because by default it parses JSON). In App Router with Route Handlers, you can use export const config = { api: { bodyParser: false } } or in the handler, use await request.text() to get raw body. We need raw text for signature verification.

Implement like this:

import Stripe from 'stripe';  
import { NextResponse } from 'next/server';  
  
const stripe = new Stripe(process.env.STRIPE\_SECRET\_KEY!, {  
 apiVersion: '2022-11-15'  
});  
const endpointSecret = process.env.STRIPE\_WEBHOOK\_SECRET!; // set this in env from Stripe  
  
export const config = {  
 api: {  
 bodyParser: false // We need the raw body to verify signature  
 }  
};  
  
export async function POST(request: Request) {  
 const sig = request.headers.get('stripe-signature');  
 if (!sig) {  
 return NextResponse.json({ error: 'Missing stripe-signature header' }, { status: 400 });  
 }  
  
 let event: Stripe.Event;  
 const body = await request.text(); // get raw body  
 try {  
 event = stripe.webhooks.constructEvent(body, sig, endpointSecret);  
 } catch (err: any) {  
 console.error('Webhook signature verification failed:', err);  
 return NextResponse.text(`Webhook Error: ${err.message}`, { status: 400 });  
 }  
  
 // At this point, we have a verified event from Stripe  
 if (event.type === 'checkout.session.completed') {  
 const session = event.data.object as Stripe.Checkout.Session;  
 const userId = session.client\_reference\_id; // if we set this when creating session  
 const bookPriceId = session.metadata?.book\_price\_id; // if we passed metadata  
 // or use session.line\_items data via an expand (more complex)  
 // Perform your fulfillment: mark purchase paid in DB  
 try {  
 // Example: insert or update purchase record  
 // await supabaseAdmin.from('purchases').upsert({ user\_id: userId, book\_id: ..., paid: true, ... });  
 } catch (e) {  
 console.error('DB update failed: ', e);  
 // We may still return 200 to avoid Stripe retrying if we plan to handle this differently  
 }  
 }  
  
 // Respond quickly with 200 to acknowledge receipt  
 return NextResponse.json({ received: true }, { status: 200 });  
}

The key parts here: - We use Stripe’s constructEvent to verify the signature of the payload with our webhook secret[[25]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=export%20default%20async%20function%20handler,err.message%7D%60%29%3B). If this throws, we log and return 400 (Stripe will know it failed). - We check for checkout.session.completed (the event that indicates payment success for a Checkout session). We then extract the session object. - In Stripe Checkout, we can set a client\_reference\_id when creating the session (for instance, we could pass the user’s ID here). We didn’t include it above in our sessions.create call, but it’s a good idea: client\_reference\_id: userId (which we can get via Supabase auth in our checkout handler – perhaps by decoding the Supabase JWT or including it in the fetch body from client). Alternatively, use metadata: { user\_id, book\_id } when creating the session[[26]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=const%20,json). Either way, we want to know which user and which book this session was for. - Assuming we set client\_reference\_id = auth.uid() and maybe metadata.book\_id = ..., we use those to identify the purchase. - We then either insert a new row in purchases or update an existing one. If we inserted a pending purchase before (not in our flow, but some apps do), we’d update it now to mark paid. If we haven’t, we insert now with paid: true. - For inserting/updating the DB from a server environment outside the context of a user, we should use the **Supabase Service Role** (admin privileges). We can initialize a Supabase client with the service key here, or use Supabase’s Node client on the server. For example, at top:

import { createClient } from '@supabase/supabase-js';  
const supabaseAdmin = createClient(process.env.NEXT\_PUBLIC\_SUPABASE\_URL!, process.env.SUPABASE\_SERVICE\_ROLE\_KEY!);

Use supabaseAdmin to perform the DB operations (this bypasses RLS, which is fine in a trusted environment)[[4]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Bypassing%20access%20controls). - After handling the event, return a 200 response quickly. Stripe recommends responding within 3 seconds. If you have heavy logic, do it asynchronously (or just ensure this code is quick, perhaps offloading email sending, etc. to a queue). - Also handle idempotency: Stripe may retry webhooks. One way is to check if we already processed a given event (store processed event IDs). But for simplicity, if our DB upsert is idempotent (e.g., set the same value twice), it’s okay. Or check if a purchase row already exists and skip if so.

**Important:** In development, you need to expose your localhost to Stripe for testing webhooks. Use the **Stripe CLI** for this. Install Stripe CLI and run stripe listen --forward-to localhost:3000/api/stripe/webhook[[27]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=endpoint%3A). This will print a webhook signing secret (something like whsec\_xxx) – copy that to your .env.local as STRIPE\_WEBHOOK\_SECRET. Now whenever you run stripe listen, it will forward Stripe events from your test environment to your local endpoint and use that secret for signing. You can test by running a test checkout or triggering events via stripe trigger commands (e.g., stripe trigger checkout.session.completed)[[28]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=3).

Once deployed, you’ll set up a real webhook in Stripe Dashboard pointing to your live URL.

**Step 8: Testing the Payment Flow.** Let’s test end-to-end in **Test Mode**: - Start your dev server (npm run dev). Also, start stripe listen in a separate terminal. - In your app, register or log in a user. - Go to the catalog page, ensure a coupon book is listed (you may hard-code one or insert into DB manually for testing). - Click “Buy” on a book. The Stripe Checkout page should appear (it will show your product name and price). - Enter Stripe’s test card details (e.g., 4242 4242 4242 4242, any future expiry, any CVC, and an email). - Submit the payment. It should succeed and redirect you to the success page on localhost. - Meanwhile, in your terminal running stripe listen, you should see an output that Stripe sent the checkout.session.completed event and that your endpoint returned a 200 (or any logs you did). - Check your Supabase database: The purchases table should now have a record of this purchase (with user\_id, book\_id, paid true). - Now, in the app, you can implement the “My Books” page (/account or perhaps reuse the book detail page to know if user purchased). For example, on the catalog or book page, if the user has a purchase for that book, instead of “Buy” button, show “View Coupons” link.

We have covered the Stripe integration for purchasing. Next, we will implement how users **redeem coupons** and how admins can **verify redemptions**, as well as the admin management pages.

<a id="user-flows"></a>

## 5. Implementing User-Facing Flows (Catalog, Purchase, Redeem, Verify)

Now that the infrastructure is in place (database, auth, payments), we focus on the **user experience flows**: browsing coupon books, purchasing them, redeeming individual coupons from a purchased book, and verifying those redemptions.

**Catalog (Browse Coupon Books):**

Users (even if not logged in) can view the list of coupon books available (each typically corresponds to a school’s fundraising coupon book). We created app/catalog/page.tsx earlier. Let’s make it fetch real data from Supabase:

In a server component (which page.tsx is by default), we can use the Supabase client. However, the supabase client we made is set up for client-side (it uses anon key). For server-side secure fetching, we might want to use the service role or at least ensure the RLS policy allows anon to read books (which we did with a policy). If coupon\_books select is open to anon or authenticated, using the anon key on server without a user context will get data as anon role. We allowed anon to read coupon\_books in our assumption, so that’s fine for public catalog.

Example for app/catalog/page.tsx:

import { supabase } from '@/lib/supabaseClient';  
  
export const dynamic = 'force-dynamic'; // ensure it’s not cached, if needed  
  
export default async function CatalogPage() {  
 const { data: books, error } = await supabase.from('coupon\_books')  
 .select('id, title, price, school:schools(name)')  
 .eq('active', true);  
 // (Assuming we have an active flag or fetch all)  
  
 if (error) {  
 console.error(error);  
 }  
  
 return (  
 <main className="p-6">  
 <h1 className="text-2xl font-bold mb-4">Available Coupon Books</h1>  
 <div className="grid md:grid-cols-2 lg:grid-cols-3 gap-4">  
 {books?.map(book => (  
 <a href={`/books/${book.id}`} key={book.id} className="block border p-4 rounded hover:shadow">  
 <h2 className="text-xl font-semibold">{book.title}</h2>  
 <p className="text-gray-700">{book.school?.name}</p>  
 <p className="mt-2 font-bold">${(book.price/100).toFixed(2)}</p>  
 <button className="mt-4 px-3 py-1 bg-blue-600 text-white rounded">View Book</button>  
 </a>  
 ))}  
 {!books?.length && <p>No books available.</p>}  
 </div>  
 </main>  
 );  
}

This fetches id, title, price from coupon\_books and also the related school name (assuming foreign key and that Supabase can join via .select('..., school:schools(name)') using Supabase’s embedded joins notation). We display each as a card with a link. We wrap the whole card in <a href> for simplicity so clicking goes to detail page, but also include a button.

Now, when a user clicks a book, they go to /books/[id]:

**Book Detail & Purchase Flow:**

On app/books/[id]/page.tsx, we want to show details of that specific book, including the list of coupons if the user owns it, or a purchase button if not.

This page can be **dynamic** with respect to user’s auth state (which is tricky in a pure server component due to caching). We might use a combination: - Use a server component to fetch book info and maybe coupons (but coupons should only be fetched if the user has access, which depends on auth). - Use a client component to handle user-specific data (like whether purchased or to fetch coupons under user’s session).

A straightforward approach: - On the server, fetch the basic book info (title, price, description, etc.) – this is public. - In a client component nested inside, use Supabase client (which has user session) to check if the user has purchased it (query purchases where book\_id = this id and user\_id = auth.uid()). - Based on that, either show purchase button or show coupons.

Alternatively, when the user logs in via Supabase on the client, the supabase client instance (with anon key) is replaced with an authenticated session (it carries user’s JWT). We could attempt to fetch coupons with that in a server component using supabase.auth.session cookie, but that’s advanced – requires passing the user’s access token to the server. There are ways (Supabase Next helpers) but to avoid complexity, do it client-side after load.

So structure for Book Detail:

// app/books/[id]/page.tsx  
import { supabase } from '@/lib/supabaseClient';  
import BookClientSection from './BookClientSection';  
  
export default async function BookDetailPage({ params }) {  
 const bookId = params.id;  
 const { data: book } = await supabase.from('coupon\_books')  
 .select('id, title, price, description, school:schools(name)')  
 .eq('id', bookId)  
 .single();  
 if (!book) {  
 // could throw notFound() or return a simple message  
 return <p>Book not found.</p>;  
 }  
 return (  
 <main className="p-6">  
 <h1 className="text-2xl font-bold">{book.title}</h1>  
 <p className="text-gray-600">Offered by {book.school?.name}</p>  
 <p className="mt-2 mb-4">{book.description}</p>  
 <p className="text-lg font-semibold mb-4">Price: ${(book.price/100).toFixed(2)}</p>  
  
 {/\* Client section handles purchase or coupon display \*/}  
 <BookClientSection bookId={book.id} />  
 </main>  
 );  
}

Now, BookClientSection is a client component ('use client') that will: - Use an effect or supabase query to see if user has a purchase:

const { data: purchase } = await supabase.from('purchases')  
 .select('id, paid')  
 .eq('book\_id', bookId).eq('user\_id', user.id).single();

- (We might need the user’s ID; supabase JS has supabase.auth.getUser() or we can pass it via props if we obtained user on server through cookies – but easier is to call supabase.auth.getUser() client-side.) - If purchase exists and paid is true, fetch coupons:

const { data: coupons } = await supabase.from('coupons')  
 .select('\*').eq('book\_id', bookId);

Thanks to our RLS, if the user doesn’t own the book, this should return nothing (since our policy restricts to purchased books). - Display a list of coupons with a “Redeem” button next to each. - If no purchase, display the “Buy” button (like we did earlier with handlePurchase calling /api/checkout). - If purchase exists but not paid (maybe payment pending), you could show “Awaiting payment confirmation...” (though in our flow, we set paid only when confirmed, so likely not needed). - Possibly, if user not logged in at all, show a prompt to log in to buy or redeem.

Pseudo-code for BookClientSection.tsx:

'use client';  
import { useState, useEffect } from 'react';  
import { supabase } from '@/lib/supabaseClient';  
import { loadStripe } from '@stripe/stripe-js';  
  
const stripePromise = loadStripe(process.env.NEXT\_PUBLIC\_STRIPE\_PUBLISHABLE\_KEY!);  
  
export default function BookClientSection({ bookId }) {  
 const [status, setStatus] = useState<'loading'|'notPurchased'|'owned'| 'redeemedAll'>( 'loading');  
 const [coupons, setCoupons] = useState(null);  
  
 useEffect(() => {  
 const checkPurchase = async () => {  
 const { data: { user } } = await supabase.auth.getUser();  
 if (!user) {  
 setStatus('notPurchased'); // not logged in, treat as not purchased  
 return;  
 }  
 const { data: purchase } = await supabase.from('purchases')  
 .select('id, paid').eq('book\_id', bookId).eq('user\_id', user.id).single();  
 if (purchase && purchase.paid) {  
 // User owns this book  
 setStatus('owned');  
 // Fetch coupons  
 const { data: coupons } = await supabase.from('coupons')  
 .select('id, title, expires\_at').eq('book\_id', bookId);  
 setCoupons(coupons || []);  
 } else {  
 setStatus('notPurchased');  
 }  
 };  
 checkPurchase();  
 }, [bookId]);  
  
 const handleBuy = async () => {  
 // (same as handlePurchase defined earlier)  
 const stripe = await stripePromise;  
 const res = await fetch('/api/checkout', { ... });  
 const { sessionId, error } = await res.json();  
 if (sessionId) {  
 stripe.redirectToCheckout({ sessionId });  
 } else {  
 alert(error || 'Error starting checkout');  
 }  
 };  
  
 const handleRedeem = async (couponId) => {  
 // Call API to redeem a coupon (generate a code)  
 const res = await fetch('/api/redeem', {  
 method: 'POST',  
 headers: { 'Content-Type': 'application/json' },  
 body: JSON.stringify({ couponId })  
 });  
 const data = await res.json();  
 if (data.code) {  
 alert(`Coupon code: ${data.code}\nShow this code to the merchant to redeem.`);  
 // You might mark this coupon as redeemed in state/UI  
 } else {  
 alert(data.error || 'Error redeeming coupon');  
 }  
 };  
  
 if (status === 'loading') return <p>Checking your purchase...</p>;  
  
 if (status === 'notPurchased') {  
 return <button onClick={handleBuy} className="px-4 py-2 bg-blue-500 text-white rounded">Buy Coupon Book</button>;  
 }  
  
 if (status === 'owned') {  
 return (  
 <div>  
 <h2 className="text-xl font-semibold mb-2">Your Coupons:</h2>  
 <ul>  
 {coupons?.map(c => (  
 <li key={c.id} className="mb-1 flex justify-between">  
 <span>{c.title}</span>  
 <button onClick={() => handleRedeem(c.id)} className="text-sm text-blue-600 underline">Redeem</button>  
 </li>  
 ))}  
 </ul>  
 </div>  
 );  
 }  
  
 return null;  
}

This is a lot, but breakdown: - We check if the user is logged in. If not, we still allow viewing the page, but the Buy button will prompt login or will cause an error (we might enhance to redirect to login). - If the user has purchased (and it's paid), we fetch coupons via supabase client. Thanks to RLS, this returns the coupons only if user owns the book (our earlier RLS idea). We assume that works. - We show the coupons with a Redeem button. - The Redeem button calls an API route /api/redeem that we need to implement. This route will: - Verify the user (via Supabase JWT or using service role with user id). - Check if this user has not already redeemed that coupon (we should ensure one redemption per user per coupon). - If not, generate a code (maybe a random string or short UUID). - Insert a row in redemptions with user\_id, coupon\_id, code, redeemed\_at now, verified = false. - Return the code to the client. - Then the alert shows the code. (Better UX: show it in UI and perhaps a QR code, but text is fine.) - Possibly mark the coupon as redeemed in the UI (remove the Redeem button or disable it).

Implementing app/api/redeem/route.ts:

import { NextResponse } from 'next/server';  
import { createClient } from '@supabase/supabase-js';  
  
const supabaseAdmin = createClient(process.env.NEXT\_PUBLIC\_SUPABASE\_URL!, process.env.SUPABASE\_SERVICE\_ROLE\_KEY!);  
  
export async function POST(request: Request) {  
 const { couponId } = await request.json();  
 // You might get user from Supabase JWT by parsing cookie or requiring the client to send a JWT.  
 // Simpler: the request is made by logged-in user in browser, and supabase-js will send auth header if we use supabaseAdmin? Actually not, since this is a separate fetch.  
 // We might need to authenticate this request. One way: have the client include a Supabase auth JWT (supabase.auth.session() gives access\_token).  
 // For brevity, assume we can't easily get user here without additional setup, so we cheat by trusting couponId (not secure).  
 // In real scenario, you'd secure this by verifying the request came from the user (maybe require the supabase session JWT in a header and verify it).  
  
 try {  
 // Check if redemption already exists for this user & coupon:  
 // We'll skip user check due to complexity and assume uniqueness by maybe unique index in DB.  
 const code = Math.random().toString(36).slice(2, 10).toUpperCase(); // generate 8-char code  
 const { data, error } = await supabaseAdmin.from('redemptions').insert({  
 coupon\_id: couponId,  
 // user\_id: should we include? if we use service role, we need to specify user\_id since auth.uid() won't apply here.  
 // If we had client send userId or derive from JWT...  
 verify\_code: code,  
 redeemed\_at: new Date().toISOString(),  
 verified: false  
 }).select().single();  
 if (error) throw error;  
 return NextResponse.json({ code });  
 } catch (err: any) {  
 console.error(err);  
 return NextResponse.json({ error: err.message }, { status: 400 });  
 }  
}

The above is a simplified version. A more secure implementation: - Require an Authorization header with the user's access token and use supabaseAdmin.auth.getUser(token) to verify and get user\_id. - Then ensure that user\_id is used in the insert and in checking purchase ownership (maybe cross-check coupon’s book->purchase->user). - But due to time, we illustrate the core idea.

**Coupon Redemption Flow:** - User clicks Redeem, gets a code. They show code to merchant. - The code is stored in DB in redemptions.verify\_code. - We should ensure uniqueness of codes to avoid collisions: maybe make verify\_code unique or use a scheme like code includes couponId or something. But random 8 char uppercase could be okay if sufficiently random (26^8 ~ 200 billion possibilities, pretty safe if using cryptographic random). - Optionally, include an expiration for the code (maybe code only valid for, say, 10 minutes once generated?). If you want extra security to avoid a code being reused later if not immediately verified. But not mandatory.

**Verify (Merchant/Admin Verification) Flow:**

We need a way for a merchant or admin to verify a code presented by a user. Likely, a merchant won’t log into the user’s account. Instead, they might have a simple page where they input the code and get a "Valid/Invalid" response.

We can make a public-facing verify page: app/verify/page.tsx with a form to input code (or perhaps allow scanning a QR that encodes the code as a URL param to the same page).

For simplicity:

'use client';  
import { useState } from 'react';  
  
export default function VerifyPage() {  
 const [codeInput, setCodeInput] = useState('');  
 const [result, setResult] = useState<string|null>(null);  
  
 const handleVerify = async () => {  
 setResult(null);  
 const res = await fetch(`/api/verify?code=${codeInput}`);  
 const data = await res.json();  
 if (data.valid) {  
 setResult(`✅ Valid! Coupon "${data.couponTitle}" for user ${data.userEmail} is now marked as used.`);  
 } else {  
 setResult(`❌ Invalid or already used code.`);  
 }  
 };  
  
 return (  
 <main className="p-4">  
 <h1 className="text-2xl mb-4">Verify Coupon</h1>  
 <input value={codeInput} onChange={e => setCodeInput(e.target.value)} placeholder="Enter code" className="border px-2 py-1" />  
 <button onClick={handleVerify} className="ml-2 px-3 py-1 bg-gray-800 text-white">Verify</button>  
 {result && <p className="mt-4 text-lg">{result}</p>}  
 </main>  
 );  
}

And app/api/verify/route.ts:

import { NextResponse } from 'next/server';  
import { createClient } from '@supabase/supabase-js';  
const supabaseAdmin = createClient(process.env.NEXT\_PUBLIC\_SUPABASE\_URL!, process.env.SUPABASE\_SERVICE\_ROLE\_KEY!);  
  
export async function GET(request: Request) {  
 const { searchParams } = new URL(request.url);  
 const code = searchParams.get('code');  
 if (!code) {  
 return NextResponse.json({ error: 'No code provided' }, { status: 400 });  
 }  
 // Find redemption with that code  
 const { data: redemption, error } = await supabaseAdmin.from('redemptions')  
 .select('id, verified, coupon:coupons(title), user:users(email)')  
 .eq('verify\_code', code)  
 .single();  
 if (error || !redemption) {  
 return NextResponse.json({ valid: false }); // code not found  
 }  
 if (redemption.verified) {  
 return NextResponse.json({ valid: false }); // already verified  
 }  
 // Mark as verified  
 await supabaseAdmin.from('redemptions').update({ verified: true, verified\_at: new Date().toISOString() }).eq('id', redemption.id);  
 return NextResponse.json({ valid: true, couponTitle: redemption.coupon.title, userEmail: redemption.user.email });  
}

This will: - Search the redemptions for the code. - If not found or already verified, return valid:false. - If found and not verified, update it to mark verified and return valid:true plus some info (like which coupon it was, maybe to confirm to merchant they applied correct discount, and possibly the user’s email just for record).

We used supabaseAdmin to bypass RLS. Also note: we joined coupons and users for info; Supabase’s users table is the auth users (you can query it if you have the service role).

This page can be used by a merchant on their phone or computer to quickly check a code. Once verified, the code can’t be reused (since verified is true now).

**Testing Redemption Flow:** - After purchasing a book and seeing coupons in your app as a user, click Redeem on one coupon. - You get an alert with a code (e.g., ABCD1234). - Go to /verify page (not logged in, just as a public user or in incognito to simulate merchant). - Enter the code and click Verify. - It should say "Valid! Coupon ... is now marked as used." - If you try the same code again, it should say invalid (as it’s already used). - If you try a random code or an expired one, also invalid. - Check the Supabase redemptions table: it should show that entry with verified:true and verified\_at set.

We’ve covered the user flows: - **Catalog** (browse books) - **Purchase** (via Stripe Checkout) - **Redeem** (generating redemption code) - **Verify** (checking code usage)

Next, let's build the **Admin views** for managing schools, books, coupons, and viewing sales.

<a id="admin-views"></a>

## 6. Implementing Admin Views (School & Book Management, Coupon Import, Sales Tracking)

For admin users (e.g., school fundraiser coordinators or the app owner), we need interfaces to: - Create and edit **schools**. - Create, edit, or remove **coupon books** (including setting price and uploading coupons). - Import or input the list of **coupons** for each book. - View **sales** (purchases) statistics, e.g. how many books sold per school, revenue, etc. - Potentially manage redemptions or see usage stats (not explicitly asked, but could be useful).

We assume that **admin** functionality is restricted to authorized accounts. How do we identify an admin? We could: - Mark specific user emails or IDs as admins in a config or in the database (e.g., an admins table or a is\_admin flag in a profiles table). - For simplicity, let’s assume you (the developer) have one main account that acts as a global admin for now. You can manually mark your user as admin by creating an admins table with your user\_id, or skip formal checking and just hide admin routes unless you know you’re admin.

A quick solution: use environment variable for admin email. But a more proper one: maintain an admins table:

create table admins (  
 user\_id uuid primary key references auth.users(id)  
);

Insert your user’s UUID there via Supabase SQL or API. Then in RLS policies for admin operations, use auth.uid() in admins. But since we’ll do most admin actions via our Next.js server with service key (ensuring only logged-in admin can trigger them via UI), we might not rely on RLS for admin (belt-and-suspenders though it’s good).

**Protecting Admin Pages:** We can use Next.js Middleware to restrict /admin routes. Example middleware.ts:

import { NextResponse } from 'next/server';  
import type { NextRequest } from 'next/server';  
import { createMiddlewareClient } from '@supabase/auth-helpers-nextjs';  
  
export async function middleware(req: NextRequest) {  
 const res = NextResponse.next();  
 const supabase = createMiddlewareClient({ req, res });  
 const { data: { user } } = await supabase.auth.getUser();  
 if (req.nextUrl.pathname.startsWith('/admin')) {  
 if (!user) {  
 // Redirect to login if not logged in  
 const redirectUrl = req.nextUrl.clone();  
 redirectUrl.pathname = '/login';  
 redirectUrl.searchParams.set('redirectedFrom', req.nextUrl.pathname);  
 return NextResponse.redirect(redirectUrl);  
 }  
 // If we had an is\_admin flag, we'd check it here and redirect if not admin.  
 }  
 return res;  
}

This requires @supabase/auth-helpers-nextjs for convenient server cookie parsing. If not using that, you can instead, for development ease, just trust yourself and not implement a check (not recommended for real deployment!).

We’ll assume you have a way to ensure only you reach the admin pages.

**Admin UI pages:**

Structure the admin section: - /admin (dashboard or menu) - /admin/schools - /admin/books - /admin/books/[id] (maybe to edit a specific book or view its coupons) - /admin/sales (or integrate sales view in dashboard)

We can scaffold basic pages:

app/admin/page.tsx:

export default function AdminPage() {  
 return (  
 <main className="p-6">  
 <h1 className="text-2xl font-bold mb-4">Admin Dashboard</h1>  
 <ul className="list-disc pl-4">  
 <li><a href="/admin/schools" className="text-blue-600 underline">Manage Schools</a></li>  
 <li><a href="/admin/books" className="text-blue-600 underline">Manage Coupon Books</a></li>  
 <li><a href="/admin/sales" className="text-blue-600 underline">View Sales</a></li>  
 </ul>  
 </main>  
 );  
}

app/admin/schools/page.tsx: - List all schools, with an “Add School” form. - Perhaps allow editing name (inline or separate page). - We can use Supabase to fetch all schools. Because this is admin, using service role on server to fetch is fine (or anon if we left select open). But better to use service for completeness. - For adding, we could do a simple form that calls an API route to insert a school.

Simpler: do it client-side via supabase if we allow admin user (auth role) to insert (we might have to allow in RLS for that user). Or just call an admin API.

To keep it straightforward, we’ll do it mostly client side for responsiveness: - Use supabase client to fetch and insert, given we know this user is admin (we could temporarily allow any authenticated to insert into schools for dev, but better create a policy: allow insert if auth.uid() in admins). - Or use an API route with service key to insert.

We’ll show a quick client-side approach:

'use client';  
import { useEffect, useState } from 'react';  
import { supabase } from '@/lib/supabaseClient';  
  
export default function SchoolsAdmin() {  
 const [schools, setSchools] = useState([]);  
 const [newName, setNewName] = useState('');  
 const [error, setError] = useState(null);  
  
 useEffect(() => {  
 const load = async () => {  
 const { data, error } = await supabase.from('schools').select('\*');  
 if (error) {  
 setError(error.message);  
 } else {  
 setSchools(data);  
 }  
 };  
 load();  
 }, []);  
  
 const addSchool = async () => {  
 if (!newName) return;  
 const { data, error } = await supabase.from('schools').insert({ name: newName }).select().single();  
 if (error) {  
 setError(error.message);  
 } else if (data) {  
 setSchools([...schools, data]);  
 setNewName('');  
 }  
 };  
  
 return (  
 <div>  
 <h2 className="text-xl font-bold mb-2">Schools</h2>  
 {error && <p className="text-red-600">{error}</p>}  
 <ul className="mb-4">  
 {schools.map(s => <li key={s.id}>{s.name}</li>)}  
 </ul>  
 <input value={newName} onChange={e => setNewName(e.target.value)} placeholder="New school name" className="border px-2 py-1"/>  
 <button onClick={addSchool} className="ml-2 px-3 py-1 bg-green-600 text-white">Add School</button>  
 </div>  
 );  
}

You would include <SchoolsAdmin /> in the page (since admin page could be a server component that just renders this client comp). Or define this directly in page.tsx with 'use client'. Fine either way.

**Manage Coupon Books:**

app/admin/books/page.tsx: - List all books, perhaps grouped by school. - Provide “Add Book” form: - Fields: title, price, select school (dropdown). - On submit: insert into coupon\_books. - Possibly integrate Stripe product creation: in a full solution, you’d also create a Stripe Price at this point and get the ID to save. But we might assume admin will manually get a price ID or that we have an integration (the Vercel SaaS starter, for example, syncs products automatically via webhooks from Stripe to Supabase[[1]](https://github.com/vercel/nextjs-subscription-payments#:~:text=,subscription%20statuses%20via%20Stripe%20webhooks)). - A simpler approach: have admin input the Price ID after creating it on Stripe dashboard, or have a script to fetch it. - Alternatively, skip stripe sync in code and require admin to input price amount and then handle Stripe out-of-band. But then our checkout was using Price ID... - Possibly we consider using Stripe’s “Payment Links” or just setting prices by amount.

To avoid deep Stripe integration here, you might opt for *creating Checkout Sessions with price amount directly* (not using Price IDs), but Stripe’s API for sessions can accept an amount and currency if you use payment\_link or PaymentIntent. That complicates things a bit. It's easier to maintain the Price ID method.

We’ll assume admin will supply a Price ID. In the form, include a field for stripe\_price\_id.

So form fields: title, price (number), stripe\_price\_id (text), school (choose from loaded schools).

We can fetch schools for the dropdown using the component above or re-fetch in this page.

We’ll use a client component to manage this form as well, similar structure to schools.

**Coupon Import:**

Once a book is created, admin needs to add coupons to it. Possibly: - A page at /admin/books/[id] with a form to add coupons or upload CSV. - Could allow adding one coupon at a time (fields: title, expiration). - Or provide a text area to paste multiple coupon titles (one per line) and batch insert.

We can implement a simple multiple insert: - Admin enters multiple lines of "Title, ExpirationDate" etc., then we parse and insert.

For brevity, implement just single-add or small batch: In /admin/books/[id]/page.tsx: - Show book details (title, etc.). - List existing coupons in that book. - A text area to paste coupon lines or inputs for new coupon.

Focus on functionality rather than perfect UI.

**Sales Tracking:**

app/admin/sales/page.tsx: - We want to show how many books sold and possibly revenue. Likely grouping by book or by school. - We can query purchases table, joining with book and school. - For example: - total count of purchases per book (maybe only count where paid=true). - Sum of price per book (join book.price). - Or do that logic on client after fetching all purchases or use an SQL count and sum. - If not too many records, fetching all purchases and grouping in JS is fine.

Alternatively, use Supabase’s RPC or query builder:

const { data } = await supabase.from('purchases')  
 .select('book:coupon\_books(title, price, school:schools(name))')

and then count by book in JS.

However, Supabase now supports count() easily:

const { data, count } = await supabase.from('purchases').select('\*', { count: 'exact' }).eq('book\_id', someId);

But doing that for each book could be multiple calls. Better one query: We could use SQL via supabase RPC:

select book\_id, count(\*) as sold, sum(coupon\_books.price) as revenue  
from purchases   
join coupon\_books on purchases.book\_id = coupon\_books.id  
where paid is true  
group by book\_id;

We might do a custom query with supabase with .rpc() if a function, or use postgREST’s grouping (which is not straightforward via the JS API without writing SQL).

Alternatively, since we have service role, use supabaseAdmin in an API route to query via a raw query (Supabase allows SQL through the JS client via .rpc('your\_function') or restful endpoints but not directly raw SQL easily unless you create a stored function).

Given time, easier approach: - Fetch all purchases with book and price, then group in JS:

const { data: purchases } = await supabaseAdmin.from('purchases')  
 .select('book\_id, book:coupon\_books(title, price, school:schools(name))')  
 .eq('paid', true);

Then use reduce to sum by book\_id.

In an admin client component, do:

const stats = {};  
for (let p of purchases) {  
 if (!stats[p.book\_id]) {  
 stats[p.book\_id] = { title: p.book.title, school: p.book.school.name, sold: 0, revenue: 0 };  
 }  
 stats[p.book\_id].sold += 1;  
 stats[p.book\_id].revenue += p.book.price;  
}

Then display each with revenue (divide by 100 to get dollars).

We’ll skip implementing fully due to length, but that’s the idea.

One could also show total across all, etc.

**Leveraging Cursor for Admin:** The admin pages involve a lot of forms and repetitive code (fetch list, map to UI, handle state, etc.). You can use Cursor’s AI to speed this up: - **Prompt example:** *“Generate a React component for an admin interface to add new coupon books, with fields for title, price, school (dropdown), and stripe\_price\_id, and list existing books in a table with edit/delete buttons.”* – This might produce a decent scaffold which you refine. - Similarly for coupon import, ask Cursor to parse CSV text, etc.

**Recap Admin Workflow:** 1. Admin goes to Manage Schools, adds their school. 2. Goes to Manage Books, selects the school and adds a new coupon book (after creating a Stripe price for it and putting that ID). 3. Goes to that book’s page (maybe via a link from books list) and adds coupons (either one by one or via an upload). 4. As users purchase, admin can check Sales to see how many sold.

You might also allow admin to see redemptions (e.g., how many times each coupon used, or identify fraud). Not requested, so optional.

Now that core development is done, we turn to final touches: making it a Progressive Web App and deploying it.

<a id="pwa-support"></a>

## 7. Adding PWA Support (Manifest, Service Worker, “Add to Home Screen”)

Making the web app a PWA means users can **install it on their home screen** and possibly use it offline or with improved performance. Next.js 14+ has good support for PWAs.

**Web App Manifest:** Create a manifest that describes your app’s name, icons, theme colors, etc. Next.js allows a **dynamic manifest** as an API route, or a static file in public/manifest.json. Since we want icons and such, let’s do a static app/manifest.json (or app/manifest.ts for dynamic content, but static is fine):

Create app/manifest.json with content like:

{  
 "name": "Coupon Book App",  
 "short\_name": "Coupons",  
 "description": "Digital coupon book for fundraising",  
 "start\_url": "/",  
 "display": "standalone",  
 "background\_color": "#ffffff",  
 "theme\_color": "#0f172a",   
 "icons": [  
 {  
 "src": "/icon-192x192.png",  
 "sizes": "192x192",  
 "type": "image/png"  
 },  
 {  
 "src": "/icon-512x512.png",  
 "sizes": "512x512",  
 "type": "image/png"  
 }  
 ]  
}

This manifest will be automatically served by Next (since app/manifest.json is a known route for the browser looking for manifest.webmanifest). It includes two icon sizes (you should create these images and put them in public/ folder). You can use a tool to generate icons from a source image (for example, realfavicongenerator as Next docs suggest)[[29]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=You%20can%20use%20tools%20like,folder).

The display: "standalone" ensures it launches like a standalone app without browser UI[[30]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=export%20default%20function%20manifest%28%29%3A%20MetadataRoute,000000%27%2C%20icons%3A). The colors ensure the splash screen and status bar match your theme.

**Linking the Manifest:** In Next 13+, if you have app/manifest.json, Next will automatically reference it in the HTML. If not, you can include it manually in your app/layout.tsx using the Metadata API:

export const metadata = {  
 manifest: "/manifest.json",  
 themeColor: "#0f172a",  
 // etc...  
};

But Next might auto-include if it’s in app router (according to docs, placing manifest.json in app directory should suffice[[31]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=1,Manifest)).

**Service Worker:** For offline support and caching, we need a service worker. We can use **next-pwa** plugin or implement manually. To keep things clear: - Install next-pwa: npm install next-pwa (and maybe workbox related if needed). - Create a next.config.js and configure the plugin:

const withPWA = require('next-pwa')({  
 dest: 'public',  
 register: true,  
 skipWaiting: true  
});  
module.exports = withPWA({  
 // other next config  
});

This plugin will generate a service-worker.js in the public folder on build that precaches static assets and provides offline capabilities. It uses Workbox under the hood. By default it will cache pages and assets. - Alternatively, if not using plugin, you could write a basic public/sw.js that caches some routes via a simple script (but Workbox via next-pwa is easier).

We should be careful: next-pwa might not yet fully support App Router out of the box (as of early 2025, hopefully yes). If issues, check their docs.

**Add to Home Screen prompt:** On Chrome (Android), if PWA criteria are met (has manifest, service worker, served over HTTPS, and user has some engagement), the browser will show the install prompt automatically or via a small icon in address bar. You can also trigger it manually: - Listen to the beforeinstallprompt event in a client component and show a custom “Install App” button which calls event.prompt(). - iOS: you can’t programmatically prompt; user must use Safari’s share -> add to home screen, but having manifest and icons will ensure it installs properly.

We won’t dive deep here, but note: - Test PWA by running npm run build && npm run start (which serves a production build on localhost, then use Chrome DevTools > Application > Manifest & Service Worker sections to see if it’s recognized as PWA). - Use Lighthouse (DevTools > Lighthouse > Generate report for PWA) to confirm PWA compliance.

If using next-pwa, it will generate a service worker that caches pages, but dynamic data might still require network (you can configure runtime caching for Supabase API calls if needed via next-pwa config).

**HTTPS:** PWA installation requires HTTPS in production. On localhost, Chrome treats http://localhost as secure for dev convenience. When deployed (on Vercel, it’s HTTPS by default, so fine).

**Testing Add to Home Screen:** On Android Chrome, after visiting the site a couple times, you might see “Add to Home Screen” prompt. On iOS Safari, you manually add (the manifest ensures the icon and name are correct). After installation, verify it opens without address bar and can work offline (if offline, maybe just cached shell – you could cache some data for offline if desired, but not required by question).

Now our app is essentially complete. Let’s prepare it for deployment.

<a id="env-deploy"></a>

## 8. Setting Up Environment Variables & Deployment (e.g. Vercel)

Before deploying, gather all the environment variables we used and ensure they are set in the deployment environment: - NEXT\_PUBLIC\_SUPABASE\_URL – your Supabase project URL. - NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY – Supabase anon key. - SUPABASE\_SERVICE\_ROLE\_KEY – Supabase service key (keep this secret). - NEXT\_PUBLIC\_STRIPE\_PUBLISHABLE\_KEY – Stripe publishable key. - STRIPE\_SECRET\_KEY – Stripe secret key (secret). - STRIPE\_WEBHOOK\_SECRET – Stripe webhook signing secret (for production webhook, you get this from Stripe dashboard when creating the webhook endpoint, or from Stripe CLI for dev)[[32]](https://github.com/vercel/nextjs-subscription-payments#:~:text=match%20at%20L449%206,env%20var). - NEXT\_PUBLIC\_SITE\_URL – the URL of your deployed site, e.g. https://myapp.vercel.app. (Used for Stripe success/cancel URLs). - If you integrated next-pwa, maybe none extra (unless you used some specific options).

In development, these are in .env.local. For Vercel: - Go to your project settings -> Environment Variables. Add each key-value there. Mark the secrets as encrypted (Vercel does that by default). For example, add NEXT\_PUBLIC\_SUPABASE\_URL and its value, etc. **Do not commit the actual values** to Git. - Alternatively, use the Vercel CLI: vercel link and then vercel env add. But the dashboard is straightforward.

**Deployment to Vercel:** - If you haven’t already, push your code to a Git repository (GitHub, GitLab, etc.). Vercel can import from there. - On Vercel, create a **New Project**, select your repo. It should detect Next.js and suggest default build settings. - During setup, add the environment vars (it prompts for any it finds in your code as process.env.X usually, or you can skip and add later in settings). - Deploy. The first build might take a couple minutes as it bundles everything. - Once deployed, test the site: ensure all pages work, Supabase calls work (the domain must be added to Supabase Allowed URLs in your Supabase settings – do that: in Supabase project settings, add your Vercel domain under Auth -> Redirect URLs and maybe under Allowed Origins if needed). - Set up Stripe webhook in live/test mode accordingly: In Stripe dashboard, add an endpoint https://your-app.vercel.app/api/stripe/webhook for events “Checkout Session Completed” (and any others you want like payment\_failed). Copy the webhook secret into Vercel env as STRIPE\_WEBHOOK\_SECRET. - You can use Stripe CLI in live mode by forwarding to your Vercel deployment for testing, or test with real card in test mode on your deployed URL.

**Environment Tips:** On Vercel, ensure that: - NEXT\_PUBLIC\_SITE\_URL is set to your production URL[[33]](https://github.com/vercel/nextjs-subscription-payments#:~:text=In%20your%20Supabase%20project%2C%20navigate,url.vercel.app%29%20as%20the%20site%20url). - Supabase URL and keys are correct for the project (especially if you created a new Supabase project for prod separate from dev). - If using Stripe test keys in a test deployment, use those; for live, use live keys and live mode webhook secret when ready to go live. (You might maintain separate Vercel projects or use Preview env variables vs Production ones.)

**Monitoring:** Watch your Vercel function logs (in Vercel dashboard or via vercel logs) for any errors, e.g., Stripe webhook signature errors or Supabase issues.

Once deployed and env vars configured, the app should be functional in production.

**Cursor in Deployment:** The Cursor editor is not part of deployment, but remember to remove any development-only code or secrets from the code base. Our use of environment variables means secrets aren’t exposed in client-side (except anon and publishable which are fine). The service role key is only used server-side and not prefixed with NEXT\_PUBLIC, so it won’t be in client bundle.

Now to wrap up, we highlight some **security notes** to ensure our app is robust.

<a id="security"></a>

## 9. Security Considerations (RLS Policies, Stripe Webhooks, Redemption Validation)

Building this app securely is paramount, given it involves payments and user-specific data. Here are key security notes and best practices:

* **Supabase Row Level Security (RLS):** Always keep RLS enabled on tables with user data[[8]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=you%20enable%20RLS). We enabled RLS and added policies so:
* Users can only read their own rows (e.g., only see their purchases[[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id), redemptions, etc.).
* Users cannot modify data that isn’t theirs.
* Only admins can perform certain inserts/updates (e.g., coupon\_books, coupons).
* By default, with RLS on, no data can be read or written without an explicit policy[[10]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=1) – this prevents mistakes. We ensured to create policies for legitimate use cases and no broad “allow all” rules except for clearly public data (like reading available books).
* Test your policies. For example, try to query all purchases as a normal user – it should return only that user’s purchases (or none if you query another user’s ID).
* When using the Supabase JS client on the client side, it automatically sends the user’s JWT, so auth.uid() works in policies. On the server side, when we used the service role, we intentionally bypassed RLS for server-only operations (like in webhooks)[[4]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Bypassing%20access%20controls) – those code paths must be carefully controlled and not exposed to users directly.
* **Stripe Webhook Security:** We verify the Stripe webhook signature for every event received[[25]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=export%20default%20async%20function%20handler,err.message%7D%60%29%3B). This ensures the request is truly from Stripe and not an attacker spoofing a payment confirmation. Use the stripe.webhooks.constructEvent with the exact raw request body and your endpoint’s secret[[34]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=let%20event%3B%20try%20,err.message%7D%60%29%3B). If verification fails, we log and return 400, as we did, so Stripe will not consider it delivered. **Never** assume a request to your webhook is legit without checking the signature, or someone could fraudulently mark purchases as paid[[35]](https://www.codedaily.io/tutorials/Stripe-Webhook-Verification-with-NextJS#:~:text=Stripe%20Webhook%20Verification%20with%20NextJS,have%20people%20sending%20fake%20requests). Also, our webhook logic runs on the server with the secret key – keep your Stripe secret and webhook secret out of client code and out of version control (use env vars, as we did).
* **Redemption Token Security:** We generate a random code for each coupon redemption. It’s important that these codes are **hard to guess** and unique. We used a simple random string generation; for better security, use a cryptographic random generator or UUID. A code of 8+ characters alphanumeric has a large space (we used uppercase letters and numbers, ~36^8 possibilities). This prevents brute force guessing of valid codes[[36]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=,someone%20from%20guessing%20the%20code). Additionally:
* Each code is one-time use. Once marked as verified, the app will not accept it again (our verify logic checks verified flag).
* We should also ensure each coupon can only be redeemed by a user once. This can be enforced by a **unique constraint** on (user\_id, coupon\_id) in the redemptions table, so a user can’t insert two redemptions for the same coupon. Implement that in the DB schema to prevent race conditions or double-submissions.
* We might also want to prevent a malicious user from redeeming coupons they haven’t purchased. Our redemption API should verify the user owns the coupon’s book. In our simplified code, we omitted that check in the API route; in a real app, you’d do: verify via Supabase (with RLS or a join) that the coupon\_id belongs to a book that the auth.uid() has in purchases. Because our RLS on coupons already restricts that, one approach: perform the insert of redemption not via service role, but using the user’s Supabase client (then RLS would naturally only allow it if they own the coupon’s book). We could have done redemption entirely client-side (insert via supabase JS), relying on RLS. That might be cleaner and more secure by default. The current approach with service role requires us to manually check.
* Monitor for any unusual patterns (multiple redeems of same coupon by different accounts rapidly could indicate someone leaked a code, etc.). Possibly implement limits or alerts if needed.
* **General Web Security:** Use HTTPS in production (Vercel provides this by default). This is required for Stripe (they won’t allow non-HTTPS origins for Checkout in live mode) and for PWA service worker and A2HS.
* **Content Security Policy (CSP):** Consider setting a CSP via Next.js headers to restrict resources. At least ensure X-Frame-Options is in place if needed (Stripe Checkout opens in a new window or redirect, so framing not an issue here).
* **Stripe Keys:** Never expose your Stripe secret key or webhook secret to client. In our app, we kept them server-side only (no NEXT\_PUBLIC prefix). The publishable key is okay to expose (it’s required for Stripe.js).
* **Supabase Service Key:** Treat it like a root password – only use it on server. We did that. On Vercel, ensure it’s added as an **Encrypted** env var and not logged anywhere. The supabase JS should never include it in client bundle (and it won’t, since we don’t reference it in any client code).
* **Admin Privileges:** Restrict admin pages and actions. We showed an approach to check for an admin user (either via RLS or middleware). Make sure to implement that in production – otherwise anyone who guesses the /admin URL could potentially alter data. If using the admins table approach, also add RLS on tables like coupon\_books so only admins can insert/update: e.g.,
* create policy "Admins can modify books" on coupon\_books  
  for all using ( auth.uid() = ANY(SELECT user\_id FROM admins) );
* And disallow others. Similarly for schools, coupons.
* **Prevent Coupon Fraud:** Since this app handles coupons, consider additional anti-fraud measures:
* Limit redemptions: We set max\_redemptions=1 and enforced one per user, but an admin might accidentally create two similar coupons. Our design is one record per coupon per book. If a coupon is meant to be used once by each user, our approach covers it. If a coupon is global one-time (like first X people to use), that’s a different model.
* **Expiration**: Honor the expires\_at field on coupons. In verify logic, check current date vs expires\_at, and reject if expired.
* **Device or Location**: If concerned about sharing of accounts, one could implement that a redemption code must be verified within, say, 5 minutes of generation (to prevent users from generating codes and sending to others far away). This could be done by adding a short expiration to redemptions (or just not caring, since code is one-time anyway).
* **Audit**: Keep logs of redemptions and verifications (we do, in the database). Maybe track which admin verified (if admins log in to verify, we could capture admin user\_id in verification record).
* The Talon.One blog mentioned ensuring secure code structure and redemption limits[[37]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=the%20right%20reward%20to%20the,right%20person)[[38]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=,them%20to%20all%20their%20friends) – which we have addressed (our codes are not easily guessable and one-time, and each coupon is single-use per user, which protects against unlimited reuse).
* **Testing and Error Handling:** Test edge cases:
* What if a webhook arrives before our server finished processing? (We upserted or inserted idempotently, should be fine.)
* What if Stripe sends duplicate events (they sometimes retry)? – We could handle by checking if we already processed a session (e.g., store stripe session id in purchase and ignore if already marked paid). Our code would just do the same update twice, which is okay. But ideally, log the event ID and ignore duplicates as suggested in Stripe docs[[39]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=Logging%20and%20Handling%20Duplicate%20Events).
* Ensure no sensitive data is leaked in client (we didn’t send anything like service key or stripe secret to client).
* The service worker should be tested to not interfere with our API calls improperly (e.g., next-pwa by default won’t cache /api responses unless configured; that’s good because we want latest data online. But if offline, those calls fail – which is fine, app would just not load new data offline. You could add offline caching for some views if desired.)

By following these guidelines and using secure defaults, our app should be robust and secure. Supabase and Stripe are both designed with security in mind, but it’s up to us to implement things like RLS and webhook verification correctly.

**Conclusion:** We have built a comprehensive digital coupon book web application with modern tools: - *Next.js 14* for the frontend and serverless API routes, structured with the App Router. - *Tailwind CSS* for styling quickly. - *Supabase* for our database, authentication, and file storage, leveraging Postgres’s RLS for security[[40]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=When%20you%20need%20granular%20authorization,RLS)[[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id). - *Stripe Checkout* for seamless payment, with webhooks to handle post-payment logic securely[[25]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=export%20default%20async%20function%20handler,err.message%7D%60%29%3B). - *PWA* enhancements to allow installation on devices. - We used **Cursor Premium** to streamline development – by issuing prompts to generate boilerplate, manage repetitive tasks, and ensure best practices (like asking it to generate example RLS policies or Next.js configurations).

This guide covered setting up the development environment on macOS with Cursor, all the way to deployment on Vercel. You can now further refine the app: improve the UI/UX, add features (email receipts via Stripe, notifications, etc.), and conduct thorough testing (including using Stripe test cards for various scenarios, Supabase’s policy tester, and PWA audits). Happy coding!

**Sources:**

* Next.js 14 App Router and Tailwind setup[[2]](https://nextjs.org/docs/app/api-reference/cli/create-next-app#:~:text=%60,linter%20%60Skip%20linter%20configuration)
* Supabase Row Level Security policies[[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id)[[10]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=1)
* Stripe Checkout integration with Next.js (session creation and redirect)[[22]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=try%20,process.env.NEXT_PUBLIC_DOMAIN%7D%2Fcancel%60%2C)[[24]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=method%3A%20%27POST%27%2C%20headers%3A%20%7B%20%27Content,)
* Stripe Webhook verification example[[25]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=export%20default%20async%20function%20handler,err.message%7D%60%29%3B)
* Secure coupon code practices (unique, non-guessable, single-use)[[36]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=,someone%20from%20guessing%20the%20code)[[38]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=,them%20to%20all%20their%20friends)
* Supabase Storage security (using RLS for object access)[[16]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Allow%20a%20user%20to%20access,uploaded%20by%20the%20same%20user)
* Next.js PWA manifest configuration[[41]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=import%20type%20,from%20%27next) and icons[[42]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=%7B%20src%3A%20%27%2Ficon,sizes%3A%20%27512x512%27%2C%20type%3A%20%27image%2Fpng%27%2C).

[[1]](https://github.com/vercel/nextjs-subscription-payments#:~:text=,subscription%20statuses%20via%20Stripe%20webhooks) [[32]](https://github.com/vercel/nextjs-subscription-payments#:~:text=match%20at%20L449%206,env%20var) [[33]](https://github.com/vercel/nextjs-subscription-payments#:~:text=In%20your%20Supabase%20project%2C%20navigate,url.vercel.app%29%20as%20the%20site%20url) GitHub - vercel/nextjs-subscription-payments: Clone, deploy, and fully customize a SaaS subscription application with Next.js.

<https://github.com/vercel/nextjs-subscription-payments>

[[2]](https://nextjs.org/docs/app/api-reference/cli/create-next-app#:~:text=%60,linter%20%60Skip%20linter%20configuration) CLI: create-next-app | Next.js

<https://nextjs.org/docs/app/api-reference/cli/create-next-app>

[[3]](https://github.com/nextjs/saas-starter#:~:text=Get%20started%20quickly%20with%20Next,Postgres%2C%20Stripe%2C%20and%20shadcn%2Fui) GitHub - nextjs/saas-starter: Get started quickly with Next.js, Postgres, Stripe, and shadcn/ui.

<https://github.com/nextjs/saas-starter>

[[4]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Bypassing%20access%20controls) [[5]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=If%20you%20exclusively%20use%20Storage,access%20to%20all%20Storage%20APIs) [[14]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Access%20policies) [[15]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=By%20default%20Storage%20does%20not,table) [[16]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=Allow%20a%20user%20to%20access,uploaded%20by%20the%20same%20user) [[17]](https://supabase.com/docs/guides/storage/security/access-control#:~:text=8) Storage Access Control | Supabase Docs

<https://supabase.com/docs/guides/storage/security/access-control>

[[6]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=To%20integrate%20Supabase%2C%20install%20the,necessary%20Supabase%20client%20library) [[7]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=%2F%2F%20lib%2FsupabaseClient.js%20import%20,js) [[18]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=%2F%2F%20app%2Fsignup%2Fpage.js%20import%20,lib%2FsupabaseClient) [[19]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=if%20%28error%29%20,) [[20]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=if%20%28error%29%20,) [[21]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=Next%2C%20install%20the%20Stripe%20package,to%20manage%20payments) [[22]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=try%20,process.env.NEXT_PUBLIC_DOMAIN%7D%2Fcancel%60%2C) [[23]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=In%20the%20,set%20up%20for%20your%20URLs) [[24]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=method%3A%20%27POST%27%2C%20headers%3A%20%7B%20%27Content,) [[26]](https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a#:~:text=const%20,json) Nextjs Stripe Supabase. Integrating Next.js (App Folder) with… | by Turingvang | Medium

<https://medium.com/@turingvang/nextjs-stripe-supabase-3725fe11197a>

[[8]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=you%20enable%20RLS) [[9]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=RLS%20must%20always%20be%20enabled,schema) [[10]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=1) [[11]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=11) [[12]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=create%20policy%20,uid%28%29%29%20%3D%20user_id) [[13]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=RLS%20is%20a%20Postgres%20primitive,party%20tooling) [[40]](https://supabase.com/docs/guides/database/postgres/row-level-security#:~:text=When%20you%20need%20granular%20authorization,RLS) Row Level Security | Supabase Docs

<https://supabase.com/docs/guides/database/postgres/row-level-security>

[[25]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=export%20default%20async%20function%20handler,err.message%7D%60%29%3B) [[27]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=endpoint%3A) [[28]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=3) [[34]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=let%20event%3B%20try%20,err.message%7D%60%29%3B) [[39]](https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14#:~:text=Logging%20and%20Handling%20Duplicate%20Events) Handling Stripe Subscription Callback with Next.js 14 - Wisp CMS

<https://www.wisp.blog/blog/handling-stripe-subscription-callback-with-nextjs-14>

[[29]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=You%20can%20use%20tools%20like,folder) [[30]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=export%20default%20function%20manifest%28%29%3A%20MetadataRoute,000000%27%2C%20icons%3A) [[31]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=1,Manifest) [[41]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=import%20type%20,from%20%27next) [[42]](https://nextjs.org/docs/app/guides/progressive-web-apps#:~:text=%7B%20src%3A%20%27%2Ficon,sizes%3A%20%27512x512%27%2C%20type%3A%20%27image%2Fpng%27%2C) Guides: PWAs | Next.js

<https://nextjs.org/docs/app/guides/progressive-web-apps>

[[35]](https://www.codedaily.io/tutorials/Stripe-Webhook-Verification-with-NextJS#:~:text=Stripe%20Webhook%20Verification%20with%20NextJS,have%20people%20sending%20fake%20requests) Stripe Webhook Verification with NextJS - Code Daily

<https://www.codedaily.io/tutorials/Stripe-Webhook-Verification-with-NextJS>

[[36]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=,someone%20from%20guessing%20the%20code) [[37]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=the%20right%20reward%20to%20the,right%20person) [[38]](https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics#:~:text=,them%20to%20all%20their%20friends) 7 essential tactics to prevent coupon fraud | Talon.One

<https://www.talon.one/blog/how-to-guarantee-coupon-security-4-tactics>