ChatGPT

Migration to TurboRepo with Supabase and NextForge Patterns

Migrating your messy monorepo to a production-grade TurboRepo (using Bun and Supabase) can be tackled in a structured way. Below are step-by-step plans to **upgrade the database setup**, **streamline local development & deployments**, and **organize authenticated vs. public routes** in your Next.js app. The goal is to leverage ideas from the NextForge template while preserving your existing Supabase setup (database and auth).

1. Upgrade Database Setup with a Declarative Schema & Dedicated Package

a. Adopt Supabase's Declarative Schema for Migrations: Instead of hand-writing SQL in `supabase/migrations`, use Supabase's *declarative schemas* feature. This lets you define the desired end-state of your database in one place, and the CLI will generate the diff migrations for you [1]. For example, you can create SQL files under a new `supabase/schemas/` directory representing your tables (one file per domain or table). E.g. `supabase/schemas/stories.sql` might contain the `CREATE TABLE "stories" (...)` statement for your stories table. Once these files are in place, run `supabase db diff -f "<migration_name>"` to auto-generate a migration by comparing the schema files to the current database [2] [3]. This workflow ensures all schema changes are centralized (declarative) and then compiled into versioned migration files for consistency. After generating, apply migrations with `supabase start` (to spin up the local Postgres) and `supabase migration up` [4] [5]. This way, you preserve a **local testing environment** (Supabase's local Docker services) while managing schema changes more cleanly. In summary: maintain table definitions in `supabase/schemas/`, use `supabase db diff` to produce migrations, and continue to use `supabase migration up` (or `supabase db push` for remote) to apply changes.

b. Introduce a `database` Package (Modular DB Access): Emulate NextForge's approach by refactoring your database logic into a dedicated Turborepo package (e.g. `packages/database`). This package will encapsulate all DB-related code – connections, queries, and types – making it accessible to both your Next app and Express worker. For instance, NextForge uses a `@repo/database` package with Prisma as an ORM, so that in any server-side code one can do: `import { database } from '@repo/database'; const users = await database.user.findMany();` for a type-safe query [6]. You can achieve similar "magic" in your project: - **If you stick with Supabase JS client:** Create a singleton Supabase client in this package (using `createClient()` with the URL and service role key for admin capabilities). Then export helper functions for your common queries. For example, `listStories()` in your `supabase/queries/stories/list-stories.ts` can be moved here and use the client internally. After this, your Next pages or Express routes can simply call `database.listStories()` instead of inlining the query logic. This improves readability and modularity. - **If you want Prisma or another ORM:** You could integrate Prisma with Supabase's Postgres for truly type-safe DB operations. Define a Prisma schema for your tables (aligning with Supabase schema, including auth tables if needed) and let Prisma generate a client. In the `database` package, instantiate `new PrismaClient()` and export it (as NextForge does) [7]. This would let you do `database.story.findMany()` etc., with full TypeScript types. It adds an extra layer (and you'd need to manage the Prisma migration vs Supabase migrations carefully), but Supabase's docs show it's possible to use Prisma against Supabase by pointing the client at the Supabase Postgres URL [8]. This is optional – if your existing SQL queries are working, using the Supabase JS SDK in a centralized package might be sufficient for now.

c. Preserve Local Dev and Testing: Ensure the local environment continues to work with the new setup: - Keep your Supabase CLI config. In fact, consider versioning the `supabase` folder in the repo (which contains the docker-compose, config, etc.). You might treat `supabase/` as part of the monorepo workspace. For example, include a minimal `supabase/package.json` so that pnpm/turbo recognizes it [9] [10]. This is not strictly required, but it helps integrate Supabase into your TurboRepo workflow. - Test the new declarative migration by running `supabase db diff` and `supabase start`. The CLI should start all 13 containers as before; then apply migrations so that your local DB reflects the schemas. The declarative approach ultimately generates standard SQL migration files, so you're not losing any low-level control – it's just a more maintainable way to get there [1]. - Continue to use your Supabase local for testing and development. You can still write raw SQL or use psql for advanced checks if needed, but day-to-day schema changes can go through the new workflow. Also, if you have test cases or a test database, note that Supabase CLI offers `supabase db reset` and `supabase db remote` commands. You might, for instance, use `supabase db reset` to wipe and rebuild the local DB from migrations (helpful for integration tests or a clean slate) [11] [12].

By the end of this step, you'll have a cleaner database setup: **declarative schema files** to track the whole DB, automatically generated migration SQL, and a **database package** to house your Supabase client and query logic. This mirrors NextForge's structured approach and makes your code more maintainable.

2. Set Up Local Dev Scripts and Deployment Workflow

a. Unified Dev Startup: Simplify running Next, Express, and Supabase together for development. You mentioned using `concurrently` – that's a good quick solution. For example, you could add a root script:

```
// package.json
"dev": "concurrently \\\"pnpm --filter=apps/next-app dev\\\" \\\"pnpm --
filter=apps/express-worker dev\\\" \\\"pnpm --filter=@yourorg/supabase start\\
\""
```

This would concurrently run the Next.js app, the Express server, and `supabase start`. A more **TurboRepo-integrated** approach is to leverage turborepo pipelines: - Include the Supabase workspace in your `pnpm-workspace.yaml` (as noted, e.g. add `'supabase'` alongside apps/packages paths) [10]. - In `supabase/package.json`, define scripts for common tasks: e.g. `"start": "supabase status || supabase start"` to start the local DB (with a fallback if it's already running), and maybe shortcuts like `"reset": "supabase db reset || supabase start"` [11] [12]. This lets you run `pnpm start --filter=supabase` from the root to ensure the DB is up. - Configure `turbo.json` so that running your apps depends on the DB starting. For instance, you can declare that the Next app's dev task depends on `supabase#start` (meaning the Supabase "start" script) to guarantee the database is running before Next tries to make any DB calls. TurboRepo supports workspace-to-workspace dependencies via `<workspace>#<task>` syntax [13] [14]. This is an advanced but powerful setup – it ensures when you run `turbo run dev`, it will first run `@yourorg/supabase#start` then parallelize the Next and Express dev servers. If this feels like over-engineering, the simpler `concurrently` approach is fine for now, but keep this in mind as the project grows.

b. Environment Configuration: Make sure all services know how to connect to Supabase. Both your Next app and Express server will need the Supabase credentials: - **Supabase URL and anon/service keys:** In local dev, these are provided by the CLI (likely `http://localhost:54321` and the anon key in your `.env`). In production, you'll use the project's API URL and the anon key (for client-side) or service key (for server-side admin usage). Add these to your environment variables in each context. For example, NextForge's turborepo sets `SUPABASE_URL`, `SUPABASE_ANON_KEY`, and `SUPABASE_SERVICE_KEY` as shared env variables [15]. In Vercel, you'd add them to the project's Environment Variables. On Fly.io, add them to the app's config (Fly secrets). - **Database connection string:** If you decide to connect directly to Postgres (e.g., using Prisma or pg library in Express), you'll need the connection string. Supabase provides a `postgres://` URL for the database (with username, password, host, port). In local dev, this is in the supabase docker config (by default `postgres:postgres@localhost:54322/postgres`). In production Supabase, get it from the Supabase dashboard (likely in the Settings > Database section). Ensure your Express server has this if needed. (If you stick to the Supabase JS client, it uses the URL+key instead of a raw connection string, so just ensure those are set.)

c. Smooth Deployments: You plan to host the Next.js app on Vercel and the Express worker on Fly.io, with a custom domain for the API. Here's how to achieve an easy deployment: - **Next.js on Vercel:** In a turborepo, you'll typically create a separate Vercel project for each app. In your case, that likely means one Vercel project pointed at the `apps/app` directory (assuming that's your Next.js app). Vercel allows configuring the root directory for the project; set it to your Next app folder so it builds that app only [16]. Do the usual Vercel setup: set env vars (the Supabase URL and anon key, plus any other needed keys), and Vercel will auto-deploy on git push. If you also spin up a separate marketing site app in the repo later, that could be another Vercel project – but since you might keep marketing pages within the same Next app (see Step 3), you may not need a second "web" project. - **Express API on Fly.io:** Containerize or configure your Express server for Fly. Typically this means writing a Dockerfile for the Express app (or using the Fly Launch command which can auto-detect a Node app). Ensure the Express app also has the needed env vars (likely the Supabase service key and URL so it can interact with the DB or Supabase API). Set the Fly app's internal port to whatever your Express listens on (e.g. 3000). Once deployed, set up a custom domain on Fly to `api.yourdomain.com` (Fly has guides for adding certificates for custom domains). This gives you a stable API base URL for your Next app to call. - **Supabase in Production:** You will most likely use the hosted Supabase service (so you don't have to run 13 containers in production!). That means your production DB and auth live in Supabase Cloud. Use the Supabase CLI to **promote your local changes to the cloud**. For example, after testing migrations locally, do `supabase login && supabase link --project-ref YOUR_PROJECT_REF && supabase db push` to push your migrations to the remote project [17]. This will apply any new migrations to the cloud database. Going forward, you can integrate this into deployment routines or CI (there's a GitHub Action for Supabase). The CLI approach ensures your prod DB is in sync with your local schema. - **Domain and CORS considerations:** If your Next app is on a different domain (say, `myapp.vercel.app` or `www.yourdomain.com`) and your Express API is on `api.yourdomain.com`, be sure to handle CORS on the Express server so that the browser can call it. Also update your Supabase settings if needed: Supabase allows configuring "Allowed URLs" for auth callbacks and such – add your Vercel and custom domains there.

By scripting the above, you get a **one-command local dev** experience and a predictable deploy process. For example, you might end up with a README instructing: “Run `pnpm dev` to start everything (Next, Express, Supabase). Deploy Next via Vercel, Express via Fly (after building), and push DB migrations via Supabase CLI.” Each piece is separate but now well-organized.

3. Organize Protected vs Public Routes with Supabase Auth in Next.js

With NextForge, authentication was handled by Clerk, but in your case we’ll use **Supabase Auth** and achieve a similar clear separation between authenticated and unauthenticated pages.

a. Route Grouping in Next.js App Router: Leverage Next.js 13+ route groups to separate pages logically without affecting URLs. For example, you could create an `app/(authenticated)/` folder for all pages that require login, and an `app/(public)/` (or simply keep public pages at root or in a `(marketing)` group) for pages accessible to everyone. Next.js treats folders in parentheses as organizational groups that do not add to the URL path [18]. This means you can have `app/(authenticated)/dashboard/page.tsx` serve `/dashboard` and `app/(public)/pricing/page.tsx` serve `/pricing`, etc., but the grouping allows you to apply different layouts or middleware. It’s not strictly required to use two groups — you could also just protect specific pages individually — but grouping makes it obvious which parts of the app are gated behind auth.

b. Protected Layout with Supabase Auth Check: In the authenticated route group, implement a layout or middleware that checks for a Supabase user session **server-side**, redirecting if not present. Supabase provides helpers for Next.js App Router to facilitate this. For example, install `@supabase/auth-helpers-nextjs` (the newer `@supabase/ssr` package) and use its `createServerComponentClient`. In your `(authenticated)/layout.tsx` (or in each protected page if you prefer), do something like:

```
// app/(authenticated)/layout.tsx (or page.tsx)
import { createServerComponentClient } from "@supabase/auth-helpers-nextjs";
import { cookies } from "next/headers";
import { redirect } from "next/navigation";

export default async function AuthenticatedLayout({ children }) {
  const supabase = createServerComponentClient({ cookies });
  const { data } = await supabase.auth.getSession();
  if (!data.session?.user) {
    redirect("/login");  // or redirect to the public homepage
  }
  return <SomeAuthLayoutUI>{children}</SomeAuthLayoutUI>;
}
```

This code runs on the server for each request, ensuring a user is logged in before rendering the protected content. (The `cookies` import gives the function access to the user's Supabase auth cookies.) If no user session is found, it immediately redirects to a login page [19]. The snippet above demonstrates protecting the root (`/`) page by redirecting unauthenticated users to `/login` [19]. You can apply the same logic in a layout to cover a whole group of routes, or even use Next.js Middleware for a global approach. The benefit of doing it in the App Router layer is you can easily tailor the redirect target and layout.

c. Public Pages and Login: For the login (and any marketing/public) pages, you'll place them outside the protected group. For instance, `app/login/page.tsx` could be in the root or in `app/(public)/login/page.tsx`. On that page, you might also check the session – if a user *is* already logged in and somehow lands on `/login`, you can redirect them to the app (so logged-in users don't see the login form) [20] [21]. Use Supabase's client libraries to handle sign-in/sign-up. The Supabase Auth UI or a custom form (like the example using `createClientComponentClient` for a Login component [22] [23]) can be embedded on the login page. Since you already have Supabase auth working, continue to use those methods (just ensure the redirect URL for magic links or OAuth is configured to go to your Next app, e.g., to `/auth/callback` route if you handle provider callbacks).

d. (Optional) Separate Marketing App: NextForge separates marketing pages into a different app (`apps/web`) for clarity and performance. You can do this if your marketing site is substantial or managed by different team members. However, as you noted, it might be overkill. Using route groups as described keeps things in one codebase. You can still have a distinct layout or theme for marketing pages vs. the app's pages. For example, `app/(public)/layout.tsx` could contain a marketing site navbar, while `app/(authenticated)/layout.tsx` contains an app navbar with user info, etc. This gives a clean separation without requiring a whole new project. If down the line you find the marketing content growing or needing to be on a separate subdomain, you could spin it out to its own app, but it's not necessary if the current approach suffices.

e. Supabase Auth Integration Considerations: Since you have Supabase Auth, make sure to initialize the Supabase client with the appropriate environment (use the anon key on the client-side). Supabase Auth uses HTTP-only cookies for session by default (when using the Next.js helper, it sets a cookie). The `createServerComponentClient` approach will read those cookies. Ensure your Next app domain is listed in Supabase settings (for auth cookie domain and redirect URLs). On the client side, you might use `createClientComponentClient` (from the helpers) for any client-side needs (e.g., for a logout button or real-time listener). The heavy lifting of protecting routes, though, should be done on the server as above so that even initial page load is protected.

By implementing the above, you achieve a setup where: - All **public pages** (marketing, landing, sign-in, docs, etc.) are accessible without login and separated in code for clarity. - All **authenticated pages** are grouped and will automatically redirect to login if a user isn't signed in, using Supabase's session under the hood. This matches NextForge's idea of protected route groups, just adapted to Supabase. It will be obvious in your codebase which routes are protected, and you can confidently develop new features knowing they won't accidentally be exposed to non-users.

Sources:

- NextForge documentation on database package usage [6] – demonstrating a dedicated DB client approach for clean, type-safe queries.
- Supabase documentation on declarative database schemas [1] [2] – explaining how to maintain schemas and auto-generate migrations.

- Philipp Steinrötter's guide on integrating Supabase with Turborepo [9] [12] – covers treating `supabase/` as a workspace and adding CLI commands for streamlined dev workflow.
- Next.js route groups docs [18] – showing how to organize routes without affecting URL structure, useful for separating auth-protected pages.
- Example of protecting a Next.js 13 page with Supabase Auth (server-side) [19] and redirecting to login. This pattern ensures only authenticated users can access certain routes.

---

[1] [2] [3] [4] [5] Declarative database schemas | Supabase Docs
https://supabase.com/docs/guides/local-development/declarative-database-schemas

[6] database.mdx
https://github.com/vercel/next-forge/blob/1cffa7d05eefc7b72d35fa66c02d97dccf27a908/docs/content/docs/packages/database.mdx

[7] [8] supabase.mdx
https://github.com/vercel/next-forge/blob/1cffa7d05eefc7b72d35fa66c02d97dccf27a908/docs/content/docs/migrations/database/supabase.mdx

[9] [10] [11] [12] [13] [14] [15] [17] Set up a monorepo with Supabase and Turborepo - Philipp Steinrötter
https://philipp.steinroetter.com/posts/supabase-turborepo

[16] Deploying to Vercel - Next Forge
https://www.next-forge.com/docs/deployment/vercel

[18] File-system conventions: Route Groups - Next.js
https://nextjs.org/docs/app/api-reference/file-conventions/route-groups

[19] [20] [21] [22] [23] Implementing Supabase Auth in Next13 with Prisma - DEV Community
https://dev.to/mihaiandrei97/implementing-supabase-auth-in-next13-with-prisma-172i