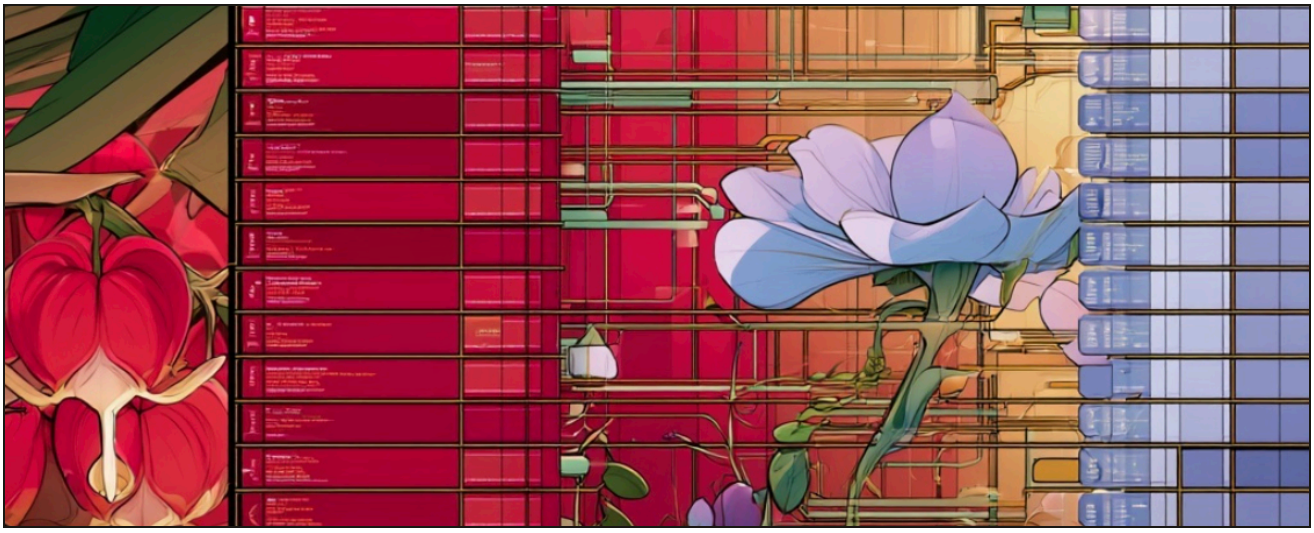
LORRAINE SAWICKI

FIVE RECIPES



01 — 05

Building a Garden App

Five short, practical descriptions documenting how What's Growing Here came together — from authentication to a restored database.

STACK
Next.js · Supabase · Claude Code

AUDIENCE
Developers

FORMAT
Recipe cards

01

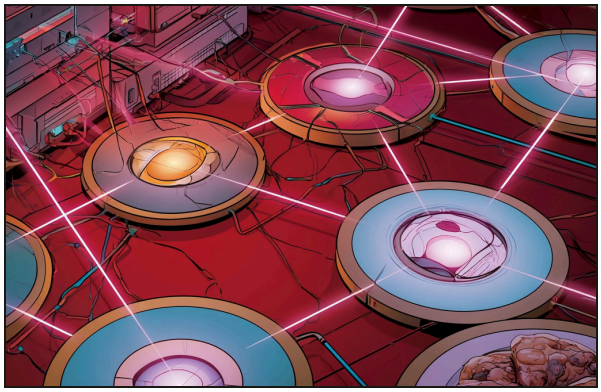
Setting Up Supabase Auth in Next.js

DIFFICULTY · MEDIUM TIME · 45 MIN STACK · APP ROUTER, @SUPABASE/SSR

The App Router splits rendering across server and client, so auth needs a client for each: one that runs in the browser, one that runs on the server and can read/write cookies. *@supabase/ssr* supplies both, and a middleware keeps the session cookie fresh on every request.

INGREDIENTS

- @supabase/ssr + @supabase/supabase-js
- Project URL & anon key (env vars)
- lib/supabase/client.ts & server.ts
- middleware.ts for session refresh
- A server action for sign-in / sign-out



METHOD

1. Install `@supabase/ssr` and `@supabase/supabase-js`, and drop the project URL and anon key into `.env.local`.
2. Write two client factories — one using `createBrowserClient` for Client Components, one using `createServerClient` for Server Components and actions.
3. Add a middleware that calls `supabase.auth.getUser()` on every request, so expired tokens get refreshed before a page ever renders:

```
export async function middleware(request: NextRequest) {
  const { supabase, response } = createMiddlewareClient(request)
  await supabase.auth.getUser()   // refreshes cookie if needed
  return response
}
```

4. In protected Server Components, read the user and redirect with `next/navigation`'s `redirect()` if there's no session.
5. Handle sign-in / sign-out as server actions so cookies are set on the response, not just in client state.

NOTE Forgetting the middleware is the single most common cause of "logged out on refresh" bugs — the browser client alone never talks to the server.

02

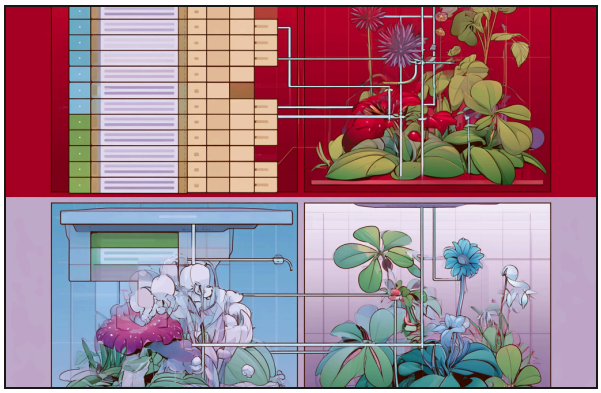
Building the Plant Catalog Schema

DIFFICULTY · MEDIUM TIME · 1 HR STACK · POSTGRES, ROW LEVEL SECURITY

Two tables carry the app: *species*, a shared reference list, and *plants*, each user's own collection pointing back at it. RLS does the access control, so the API layer never has to check ownership by hand.

INGREDIENTS

- species (public, read-only reference)
- plants (owner_id → auth.uid())
- Foreign key plants.species_id
- ALTER TABLE … ENABLE ROW LEVEL SECURITY
- One policy per operation (select/insert/update/delete)



METHOD

1. Create `species` first — it has no owner and is readable by anyone.
2. Create `plants` with an `owner_id uuid references auth.users` column, defaulted to `auth.uid()`.
3. Turn on RLS, then write one policy per operation rather than one broad policy — it's easier to reason about later:

```
alter table plants enable row level security;

create policy "read own plants"
  on plants for select
  using (auth.uid() = owner_id);

create policy "insert own plants"
  on plants for insert
  with check (auth.uid() = owner_id);
```

4. Repeat for `update` and `delete`, then test with both the anon key and a signed-in user's JWT to confirm the boundary actually holds.
5. Add a Postgres index on `owner_id` — every RLS-filtered query runs through it.

NOTE A missing policy doesn't error — it just returns zero rows. If a query comes back empty, check RLS before you check your code.

03

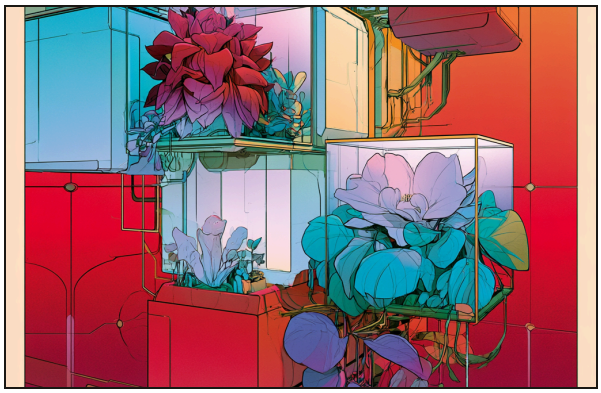
Adding Image Upload to Storage

DIFFICULTY · EASY–MEDIUM TIME · 30 MIN STACK · SUPABASE STORAGE

Photos live in a Storage bucket, not the database — the *plants* row just stores a path. Bucket policies mirror the table policies from Recipe 02, so a user can only write into their own folder.

INGREDIENTS

- Bucket: plant-photos (private)
- Path convention: {user_id}/{plant_id}.jpg
- Storage RLS policy per operation
- Client-side resize before upload
- Signed URL for display



METHOD

1. Create a private bucket and mirror the table policies: a user may `insert` / `select` only inside a folder named after their own `auth.uid()`.
2. Upload straight from the browser — no need to route the file through your own server:

```
const path = `${user.id}/${plantId}.jpg`
await supabase.storage
  .from('plant-photos')
  .upload(path, file, { upsert: true })

await supabase.from('plants')
  .update({ photo_path: path })
  .eq('id', plantId)
```

3. Resize on the client before upload (a canvas downscale to ~1600px is plenty) — it keeps the bucket small and uploads fast on mobile.
4. When rendering, request a signed URL scoped to a short expiry rather than making the bucket public.

NOTE Storage policies are written in the same policy language as table RLS, but they run against `storage.objects` — easy to forget, since the SQL editor puts them in a different tab.

04

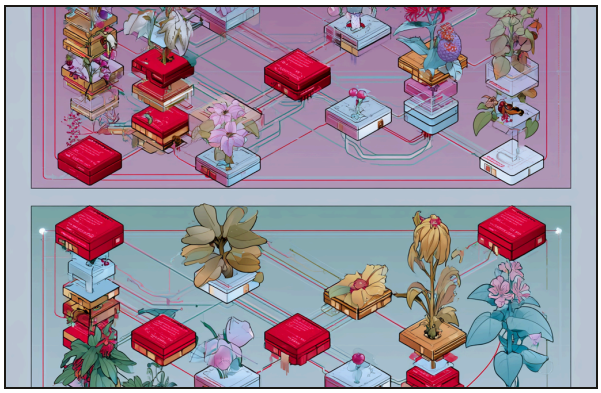
Restoring a Supabase Database

DIFFICULTY · MEDIUM TIME · 20 MIN STACK · PG_DUMP, PSQL

The scariest recipe in the book, and the one worth rehearsing before you need it. Supabase keeps daily backups, but restoring is a manual *psql* job — there's no one-click button for a full point-in-time recovery on smaller plans.

INGREDIENTS

- A backup .sql/.dump file
- Direct (non-pooled) connection string
- psql installed locally
- A scratch project to restore into first
- Your migrations folder, for reconciliation



METHOD

1. Download the backup from Dashboard → Database → Backups, or from wherever your own `pg_dump` cron drops it.
2. Grab the *direct* connection string (port 5432), not the pgbouncer pooler one (6543) — restores need a real session, not a pooled one.
3. Restore into a throwaway project first, so a bad backup never touches production:

```
psql "postgresql://postgres:[PASSWORD]@db.[REF].supabase.co:5432/postgres" \
  -f backup.sql
```

4. Check the restored row counts against what you expect, then re-run any migrations that landed after the backup was taken.
5. Only once that scratch project looks right, repeat the restore against the real one — during a maintenance window, with the app pointed elsewhere.

NOTE Restoring resets sequences and can drop RLS policies defined outside the dump — re-check policies immediately after, before anyone signs back in.

05

Scaffolding a Feature with Claude Code

DIFFICULTY · EASY TIME · VARIES STACK · CLAUDE CODE, CLAUDE.MD

Most of this app's plumbing — the recipes above included — was scaffolded this way: describe the feature in plain language, let Claude Code draft a plan and touch the files, then review in small batches instead of accepting one giant diff.

INGREDIENTS

- A CLAUDE.md with project conventions
- One feature request, written in plain language
- Git, for cheap checkpoints
- Patience to review diffs, not just accept them

METHOD

1. Keep a short `CLAUDE.md` at the repo root — stack, folder conventions, anything you'd tell a new teammate on day one.
2. Describe the feature as an outcome, not a file list: "users should be able to upload a photo when adding a plant" rather than naming components up front.
3. Let it propose a plan before writing code, and read that plan — it's the cheapest place to redirect.
4. Review the resulting diff in small chunks; ask directly for edge cases and error handling it skipped.
5. Commit once it's reviewed, then iterate with a follow-up prompt rather than restarting the request from scratch.

NOTE The Supabase recipes above are a good test of this loop — RLS policies are exactly the kind of detail worth reading closely rather than skimming past.