Supabase Course

Flavio Copes

Updated August 3, 2026

Contents

Build with Supabase using PostgreSQL, local migrations, generated APIs, Auth, Row Level Security, Storage, Realtime, Edge Functions, and tested recovery.

What you'll learn: Build and release a Supabase application with reviewed migrations, ownership-aware RLS, private files, secure realtime updates, and a recovery plan.

Supabase foundations

PostgreSQL, projects, local development, APIs, keys, and trust boundaries.

Understand the Supabase platform

See Supabase as PostgreSQL plus Auth, APIs, Storage, Realtime, Functions, and operational services rather than a new database engine.

Supabase is a hosted platform built around one idea: every Supabase project contains a full PostgreSQL database. Not a compatible clone, not a proprietary store behind an API — actual Postgres you can reach with any Postgres client.

You can prove that in one command:

```
psql "postgresql://postgres:s3cretPass@db.abcdefghijkl.supabase.co:5432/postgres" \
  -c "select version();"
# PostgreSQL 15.8 on aarch64-unknown-linux-gnu ...
```

Around that database, Supabase adds the services most applications need anyway: generated data APIs, authentication, object storage, realtime services, functions, pooling, backups, and a dashboard. Each one saves you from running a separate piece of infrastructure.

How a request actually flows

The piece that confuses beginners most is the **Data API**. When you write this:

```
import { createClient } from '@supabase/supabase-js'

const supabase = createClient(
  'https://abcdefghijkl.supabase.co',
  'sb_publishable_x3PLxbGmiQ04GJ5uOOZBww_T5xRcLhV'
)

const { data, error } = await supabase.from('notes').select()
```

no application server of yours is involved. The request hits Supabase's API gateway, which translates it into SQL and runs it against your Postgres database. Supabase Auth is the layer that **authenticates**: it checks the user's token and attaches their identity. PostgreSQL is the layer that **authorizes**: Row Level Security policies decide which rows that identity may see.

Draw that path once on paper — browser, Auth, Data API, Postgres — and mark the two jobs. Authentication happens before the database. Authorization happens inside it.

It is still Postgres

SQL, constraints, indexes, transactions, roles, and query plans still matter. The platform removes setup work; it does not remove database design or authorization decisions.

The classic first-week surprise makes the point. You enable Row Level Security on a table, write no policies, and every `select` starts returning an empty array — no error, just `data: []`. Nothing is broken and nothing is lost. Postgres is doing exactly what you told it: no policy allows any rows. The fix is a policy, and writing one is a database skill, not a platform setting.

Treat Supabase as Postgres with the boring parts handled, and everything you know about relational databases keeps paying off.

Create a local Supabase project

Use the CLI and a Docker-compatible runtime to start a reproducible local stack without experimenting directly in production.

Run `supabase init` to create the project configuration, then `supabase start` to launch the local services. The generated `supabase` directory describes configuration, migrations, seeds, and functions.

Commit reproducible configuration and migrations. Keep secrets and CLI temporary state out of Git. The local stack is close to the platform, but official documentation notes that it is not identical in every feature.

Start the stack, open Studio, record the local API URL and keys, then stop and restart it to verify the project state is reproducible.

Start from an empty directory and prove it can be rebuilt:

```
supabase init
supabase start
supabase status
supabase db reset
```

Commit configuration, migrations, and intentional seed data. Do not commit generated credentials or temporary container state. Stop the stack, start it again, and run the same reset before depending on dashboard edits that have no migration.

Separate public and secret keys

Understand publishable or anonymous client keys, server-only secret keys, JWT context, and why RLS remains mandatory.

Every Supabase project ships with two kinds of API credentials, and mixing them up is the fastest way to turn a small bug into a data breach.

The **publishable key** (named `sb_publishable_...`; older projects call it the `anon` key) is safe to ship in a browser or mobile app. It identifies your Supabase project; it does not grant unrestricted database authority by itself. Requests made with it run through Row Level Security, and the user session adds identity claims on top.

The **secret key** (`sb_secret_...`, or the legacy `service_role` JWT) is a different animal. Clients created with it bypass Row Level Security entirely:

```
import { createClient } from '@supabase/supabase-js'

// server-side code only
const admin = createClient(
  process.env.SUPABASE_URL,
  process.env.SUPABASE_SECRET_KEY
)

const { data } = await admin.from('notes').select()
console.log(data.length)
// every note from every user - no policy applied
```

That power is why secret keys must never reach a browser, mobile bundle, log line, or public repository. A leaked privileged key is an authorization incident: every RLS policy you wrote stops protecting anything until you rotate the key.

Verify what your build ships

Do not trust the mental model — check the artifact. After building your frontend:

```
grep -R "sb_secret" dist/
# (no output - this must find nothing)

grep -R "service_role" dist/
# (no output)
```

If either command prints a match, a privileged key made it into client code. Rotate it in the dashboard under Settings → API Keys immediately, then fix the import that leaked it. Rotation is not optional after exposure; the key does not expire on its own.

The inventory habit

Take a small app and inventory every environment variable it uses. For each one, write three facts: can it enter client code, where does the server-only copy live, and how do you rotate it. The project URL and the publishable key can be public. The database password, the secret key, and SMTP credentials stay server-side, in your deployment platform's secret store.

One warning to keep intact: the publishable key is only safe to expose while RLS is enabled and correct on every exposed table. A public key plus an unprotected table is a public table.

Check your understanding: supabase foundations

Check the key models, decisions, commands, security boundaries, and failure cases from supabase foundations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Database and migrations

Schemas, constraints, local migrations, generated types, connections, pooling, and release safety.

Design the PostgreSQL schema

Use tables, foreign keys, constraints, indexes, and private schemas before relying on generated APIs or client types.

Before touching the dashboard's table editor, design the schema like the ordinary PostgreSQL schema it is. Put required rules in constraints, connect ownership with foreign keys, and add indexes for real query paths.

Here is the pair of tables this course keeps returning to — profiles and notes, with a clear owner relationship:

```
create table profiles (
  id uuid primary key references auth.users (id) on delete cascade,
  username text unique not null
    check (char_length(username) between 3 and 30)
);

create table notes (
  id bigint generated always as identity primary key,
  user_id uuid not null references profiles (id) on delete cascade,
  title text not null,
  body text not null default '',
  created_at timestamptz not null default now()
);
```

Every line earns its place. The foreign keys make ownership a fact the database enforces, not a convention the app remembers. The `check` constraint rejects garbage usernames at the door. `on delete cascade` answers "what happens when a user leaves" before it becomes a support ticket.

Notice that `profiles.id` references `auth.users` — the table Supabase Auth manages. That link is how a row in your schema gets tied to a real authenticated identity, and later lessons build every ownership rule on top of it.

Indexes come from queries, not habit. The app lists a user's notes, newest first, so:

```
create index notes_user_id_created_at_idx
  on notes (user_id, created_at desc);
```

Verify it is actually used:

```
explain select * from notes
where user_id = 'b7f0c2ae-1d44-4f0a-9c31-58a2d90f5e11'
order by created_at desc limit 20;
-- Index Scan using notes_user_id_created_at_idx on notes ...
```

If the plan says `Seq Scan` instead, Postgres is reading the whole table. On a small dev dataset you will not feel it; at a million rows that same query becomes your slowest endpoint. Read plans early, while they are cheap to fix.

What the Data API exposes

One Supabase-specific decision: tables in exposed schemas (by default, `public`) can become available through the Data API. Keep internal-only structures in a private schema:

```
create schema internal;

create table internal.audit_log (
  id bigint generated always as identity primary key,
  event text not null,
  at timestamptz not null default now()
);
```

Tables in `internal` are unreachable through the API no matter what a client requests.

Generated TypeScript types (`supabase gen types typescript`) describe the schema; they do not

secure it. Types keep your code honest. Only constraints and policies keep your data honest.

Version schema changes

Create, review, reset, test, and push migration files rather than relying on undocumented dashboard state.

Dashboard edits feel fast until the day you need a second environment and cannot say what production's schema actually is. Migration files fix that: every schema change is a SQL file, committed to Git, applied in order.

Create one with the Supabase CLI:

```
supabase migration new add_notes_pinned
# Created new migration at supabase/migrations/20260803141530_add_notes_pinned.sql
```

Write the change in that file:

```
alter table notes
  add column pinned boolean not null default false;
```

Then prove the whole history still builds from zero:

```
supabase db reset
# Resetting local database...
# Applying migration 20260803141530_add_notes_pinned.sql...
# Seeding data from supabase/seed.sql...
```

`supabase db reset` rebuilds the local database from nothing but your migration files and seed data. If it fails, your history is broken and you found out locally, which is the point. Run it after every migration you write.

If you did make a change through the local Studio UI, capture it instead of losing it: `supabase db diff -f change_name` writes the difference between your migration history and the live local schema into a new migration file.

Treat generated diffs as drafts. Review permissions, extensions, destructive statements, and data migrations before pushing — the diff tool records what changed, not whether the change was wise. A generated `drop column` deserves particular suspicion.

Shipping to the remote project

Link once, then push:

```
supabase link --project-ref abcdefghijkl
supabase db push --dry-run
# Would push these migrations: 20260803141530_add_notes_pinned.sql
supabase db push
```

The dry run previews exactly which migrations would apply. Read it before the real push.

The classic disaster in this workflow is drift: someone edits production through the dashboard, local history no longer matches remote state, and the next `db push` fails or half-applies. When that happens, capture the remote change with `supabase db diff --linked` and fold it into a proper migration file.

And never reset a production database. `db reset` is a local rebuilding tool; on anything shared it

is data loss with a progress bar.

Finish the loop the way the exercise describes: add one column through a migration, rebuild locally, apply it to a disposable remote project, then run one old application path against it to prove the change was additive.

Choose direct or pooled connections

Use direct, session-pooled, or transaction-pooled PostgreSQL connections according to runtime lifetime and feature requirements.

Supabase exposes three ways to reach the same Postgres database, and picking one by copy-paste is how production incidents start. The right choice depends on what runs your code and how long it lives.

```
direct        db.abcdefghijkl.supabase.co:5432
session pool  aws-0-eu-central-1.pooler.supabase.com:5432
transaction   aws-0-eu-central-1.pooler.supabase.com:6543
```

A **direct connection** is plain Postgres. Every client costs the database a real connection. It fits migrations, `pg_dump`, and admin work — tools that need full session behavior and run one at a time.

A **session pooler** hands each client a server connection for its whole session. It behaves like direct Postgres while letting the platform manage the connection supply.

A **transaction pooler** (port `6543`) is the interesting one. Transaction pooling reuses a server connection between transactions and cannot preserve every session feature: prepared statements, session-level `SET`, advisory locks. In exchange it absorbs the pattern serverless creates — hundreds of short-lived function invocations that would each hold a direct connection and exhaust the database limit.

Now classify the three workloads from the exercise. A migration command: direct. A persistent API server: direct or session pooling, with a bounded pool. A burst of serverless functions: transaction pooler, no exceptions.

A long-lived server should also bound itself:

```
import pg from 'pg'

const pool = new pg.Pool({
  connectionString: process.env.DATABASE_URL,
  max: 10,
  connectionTimeoutMillis: 5000,
})
```

Every replica creates its own pool, so `max` times replicas must stay under the project's connection limit — leave headroom for migrations and emergency admin access.

Match the driver to the mode

The transaction pooler needs the driver to cooperate. Prisma, for example, wants `?pgbouncer=true` appended to the pooled connection string so it disables prepared statements — and still needs the direct string for `prisma migrate`, which the pooler cannot serve.

The failure smells like this: everything works locally, then production intermittently throws `prepared statement "s0" already exists`. That is a driver using prepared statements through transaction pooling, where consecutive transactions can land on different server connections. Fix the configuration, not the retry logic. Match the driver and ORM configuration to the chosen mode instead of copying a connection string blindly.

Check your understanding: database and migrations

Check the key models, decisions, commands, security boundaries, and failure cases from database and migrations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Auth and Row Level Security

Sessions, ownership, RLS policies, policy composition, server boundaries, and denied-path tests.

Authenticate users

Create a sign-in flow, understand sessions and JWTs, and configure redirects and email delivery without confusing identity with permission.

Supabase Auth verifies identity and issues a session. What the rest of the platform cares about is what that session carries: a JWT access token whose claims — the user ID above all — PostgreSQL policies can inspect later.

Sign a user up with the client library:

```
const { data, error } = await supabase.auth.signUp({
  email: 'ada@example.com',
  password: 'correct-horse-battery-staple',
})
```

New signups need email confirmation by default. When you test against the local CLI stack, no real email is sent — messages are captured by a local Mailpit inbox instead. Run `supabase status` to get its URL, open the confirmation email there, click the link. No rate limit, no real delivery.

Then sign in and look at the identity you received:

```
const { data, error } = await supabase.auth.signInWithPassword({
  email: 'ada@example.com',
  password: 'correct-horse-battery-staple',
})

console.log(data.user.id)
// 8f7a2c1e-4b9d-4e0f-a1c2-3d4e5f6a7b8c
```

Create two local users and record both IDs. Every authorization test in the lessons ahead compares what those two identities can and cannot do, so keep those identities separate.

Identity is not permission

Authentication answers ”who is this”. It does not decide which note, file, or channel a user may access — that is authorization, and it lives in Row Level Security policies, not in Auth settings.

The production checklist

Configure allowed redirect URLs, token handling, and email delivery deliberately. Redirect URLs are an allowlist: a confirmation link pointing anywhere unlisted fails, which is a protection, not a bug.

Email is where hosted projects bite. Every project ships with a built-in mail service so auth flows work immediately, but it is rate limited to a handful of messages per hour and meant for trying the flow, not production delivery. Iterate on a signup form against a hosted project and you will hit:

```
Error: email rate limit exceeded
```

The frustrating part: signups appear to succeed, confirmation emails just stop arriving, and only the error response says why. The fix is custom SMTP under the Authentication settings — any transactional email provider works, with a sender address on a domain you verified there. After that, the rate limit is yours to raise in the dashboard.

Write owner-based RLS policies

Enable RLS and express select, insert, update, and delete ownership rules with auth.uid and matching checks.

Enable Row Level Security on every exposed application table. A policy should express who may perform one operation and which rows qualify.

Use `auth.uid()` against an indexed owner column. Read policies use `USING`; inserted or changed rows need `WITH CHECK` so a user cannot assign ownership to someone else.

Create policies for private notes. Prove user A can manage one note and user B cannot read, modify, or delete it.

Enable RLS and express ownership for one operation at a time:

```
alter table notes enable row level security;

create policy "read own notes"
on notes for select
using (user_id = auth.uid());
```

Add separate insert, update, and delete policies with the correct USING and WITH CHECK expressions. Test no session, the owner, and another authenticated user. Never use the service-role key for these tests because it bypasses the boundary you are trying to prove.

Test the denied paths

Run authorization tests as anonymous, authenticated, cross-user, and privileged actors instead of checking only the happy path.

An RLS test that only checks the happy path proves nothing. The policy's entire job is denial, so denial is what you test: no session, the owner, another user, malformed input, and a trusted server action.

Build the matrix for the notes table. Two users, Ada and Grace, each owning one note. Five actors, four operations. Every cell holds an expectation: allowed or denied.

Then automate it against a fresh local database, signed in with the publishable key like a real client:

```
import { createClient } from '@supabase/supabase-js'
import test from 'node:test'
import assert from 'node:assert/strict'

const ada = createClient(url, publishableKey)
await ada.auth.signInWithPassword({
  email: 'ada@example.com',
  password: 'correct-horse-battery-staple',
})

test('ada cannot read grace notes', async () => {
  const { data, error } = await ada.from('notes')
    .select()
    .eq('user_id', graceId)

  assert.equal(error, null)
  assert.deepEqual(data, [])
})
```

Note the shape of that denial. A blocked `select` does not raise an error — RLS filters the rows out and you get `data: []`, exactly like an empty table. This is the failure mode that fools people: the app ”works”, every query succeeds, and only asserting on the actual rows reveals that users see nothing, or that a broken policy lets them see everything. Assert on contents, never on the absence of errors.

Writes deny more loudly:

```
const { error } = await ada.from('notes')
  .insert({ user_id: graceId, title: 'spoofed' })

console.log(error.code, error.message)
// 42501 new row violates row-level security policy for table "notes"
```

That is the `WITH CHECK` clause rejecting an ownership spoof. Add the anonymous actor too: a client with no session must read zero rows and fail every write.

Keep the privileged path out of user tests

Do not use a service-role client to test ordinary user behavior, because it can bypass RLS — it passes every check and validates nothing. Test privileged server actions separately, as their own actor with their own expectations. Keep those privileged jobs narrow, and make each one perform its own authorization before accepting a user-controlled identifier.

Run the suite against a database rebuilt with `supabase db reset`, so leftover rows from a previous run never mask a hole.

Check your understanding: auth and row level security

Check the key models, decisions, commands, security boundaries, and failure cases from auth and row level security.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Storage and Realtime

Private buckets, object ownership, signed access, Broadcast, Presence, and channel authorization.

Protect Storage objects

Use private buckets, predictable object paths, storage RLS policies, upload limits, and signed URLs without trusting a caller-provided path.

Supabase Storage keeps object metadata in PostgreSQL: every file in a bucket has a row in `storage.objects`, and RLS policies on that table control API access. You protect files with the same tool you protect rows.

Start with private buckets unless every file is intentionally public:

```
insert into storage.buckets (id, name, public, file_size_limit, allowed_mime_types)
values ('avatars', 'avatars', false, 1048576, array['image/png', 'image/jpeg']);
```

The size and MIME limits reject a 2 GB "avatar" before it costs you anything. Restrict size and content type at the bucket, but still inspect untrusted uploads where application safety depends on the file contents — a content type header is a claim, not proof.

Ownership lives in the path

Never trust a caller-provided path. Put the authenticated user ID in a controlled path and verify ownership in policy:

```
create policy "users manage own avatar"
on storage.objects for all
to authenticated
using (
  bucket_id = 'avatars'
  and (storage.foldername(name))[1] = auth.uid()::text
)
with check (
  bucket_id = 'avatars'
  and (storage.foldername(name))[1] = auth.uid()::text
);
```

`storage.foldername(name)` splits the object path into its folders; the policy demands the first folder equal the caller's user ID. Uploads then look like:

```
const { error } = await supabase.storage
  .from('avatars')
  .upload(`${user.id}/avatar.png`, file, { upsert: true })
```

Run the proof with your two test users. Ada uploads to `<ada-id>/avatar.png`: success. Ada replaces her own object with `upsert: true`: success. Ada uploads to `<grace-id>/avatar.png`: a 403 with "new row violates row-level security policy". Ada requests Grace's private object: denied. If any of those four checks surprises you, fix the policy, not the test.

To share a private object without opening the bucket, mint a **signed URL** on demand:

```
const { data } = await supabase.storage
  .from('avatars')
  .createSignedUrl(`${user.id}/avatar.png`, 3600)
// data.signedUrl works for one hour, then expires
```

The common mistake is the shortcut: making the bucket public "for now" because signed URLs felt like work. Public means every object is readable by anyone who has or guesses the path, with no policy consulted. Undoing it later means assuming every path already leaked.

Choose a Realtime feature

Select Broadcast, Presence, or Postgres Changes according to whether the application sends events, tracks participants, or observes database rows.

Supabase Realtime is one websocket connection carrying three different tools, and they are not interchangeable. **Broadcast** sends application events between connected clients. **Presence** tracks shared client state, like who is online. **Postgres Changes** observes changes from database replication and streams them to subscribers.

Broadcast is the workhorse:

```
const channel = supabase.channel('room:42')

channel.on('broadcast', { event: 'message' }, ({ payload }) => {
  console.log(payload.text)
})

await channel.subscribe()

channel.send({
  type: 'broadcast',
  event: 'message',
  payload: { text: 'hello from Ada' },
})
```

The event goes from one client through Realtime to the other subscribers. The database is not involved, which is exactly why it stays cheap under chatty traffic. Current Supabase guidance prefers Broadcast for scalable fan-out in many cases.

Presence answers "who is here right now":

```
channel.on('presence', { event: 'sync' }, () => {
  console.log(Object.keys(channel.presenceState()).length, 'online')
})
```

Presence is useful but can become noisy — every join, leave, and state update fans out to every subscriber, so keep the tracked state tiny.

Postgres Changes turns the database into the event source:

```
supabase.channel('settings-watch')
  .on('postgres_changes',
    { event: 'UPDATE', schema: 'public', table: 'settings' },
    payload => console.log('settings changed', payload.new))
  .subscribe()
```

It shines when the write already happens for its own sake and a few clients want to observe it. Database subscriptions still need filters and authorization, and the server checks each change against each subscriber — cost grows with writes and with listeners.

Classify before you build

Run the exercise's three cases through the decision. Chat messages: high volume, clients talking to clients — Broadcast, persisting messages separately on your own terms. Online cursors: ephemeral shared state — Presence, throttled hard. A low-volume admin table update that a dashboard should reflect: Postgres Changes fits.

The mistake to avoid is routing every UI action through database changes because one mechanism feels tidier. Do not send every UI action through Postgres Changes: you pay for each event twice — once as a table write, once as replication fan-out — and busy channels start lagging behind the conversation they carry. Choose the smallest Realtime feature for each job.

Authorize Realtime channels

Use private channels, topic design, RLS-backed authorization, cleanup, and reconnect behavior without leaking events across tenants.

A channel topic is part of the authorization boundary. `room:42` is not a decorative label — it is the value your policies inspect to decide who may join. Use stable tenant or room identifiers, never guessable convenience names built from emails or usernames.

By default, channels are open to any client that knows the topic. Mark them **private** so Realtime enforces authorization at join time:

```
const channel = supabase.channel('room:42', {
  config: { private: true },
})
```

Private channels are authorized with RLS policies on the `realtime.messages` table. A `select` policy controls who may join and receive; an `insert` policy controls who may send. Verify membership against your own data:

```
create policy "members receive room events"
on realtime.messages for select
to authenticated
using (
  exists (
    select 1 from room_members
    where room_members.room_id =
      split_part(realtime.topic(), ':', 2)::bigint
      and room_members.user_id = auth.uid()
  )
```

```
);
```

`realtime.topic()` returns the topic the client asked to join. The policy checks it against the `room_members` table, so membership — durable, queryable, revocable — is the source of truth for who hears what.

Test it with two private rooms and two users. Ada is a member of room 42, Grace is not. Ada's `subscribe()` callback reports `SUBSCRIBED`. Grace's reports an error status instead of silently joining. If Grace gets in anyway, check that the client really passed `private: true` — forgetting that flag is the common hole, and nothing looks wrong because members still connect fine.

Clean up and expect messiness

Remove subscriptions when a client leaves:

```
await supabase.removeChannel(channel)
```

A leaked subscription keeps receiving events and keeps consuming connection quota, and in UI components it double-handles every message after a remount.

Then design for the transport's honesty: expect reconnects, duplicates, gaps, and out-of-order application effects where the transport does not promise otherwise. A client that was offline for ten seconds missed whatever was broadcast during them — Realtime does not replay history. Keep durable truth in the right database table and treat channel events as hints: on reconnect, refetch state from the table, then resume listening.

Check your understanding: storage and realtime

Check the key models, decisions, commands, security boundaries, and failure cases from storage and realtime.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Functions and production

Edge Functions, secrets, environments, observability, backups, limits, and recovery.

Add a trusted Edge Function

Use a server-side function for secrets and privileged integrations while still validating identity, authorization, input, and external responses.

An Edge Function can keep provider secrets and trusted integration code out of the browser. It does not become safe merely because it runs on the server — a function that trusts its input is just a new attack surface with lower latency.

The checklist for every function is the same: verify the caller, authorize the requested resource, validate input, set timeouts, and handle retries deliberately. Skip any one of them and the function undoes protection the rest of the platform provides.

Build one that performs one narrow operation on the caller's note: emailing it to the caller.

```
supabase functions new send-note
```

```
supabase functions serve send-note
```

The function establishes the caller's identity first, using the request's own authorization header:

```
import { createClient } from 'npm:@supabase/supabase-js@2'

Deno.serve(async (req) => {
  const supabase = createClient(
    Deno.env.get('SUPABASE_URL')!,
    Deno.env.get('SUPABASE_ANON_KEY')!,
    { global: { headers: { Authorization: req.headers.get('Authorization')! } } },
  )

  const { data: { user } } = await supabase.auth.getUser()
  if (!user) return new Response('Unauthorized', { status: 401 })

  const { noteId } = await req.json()
  const { data: note } = await supabase
    .from('notes').select().eq('id', noteId).single()
  if (!note) return new Response('Not found', { status: 404 })

  // call the email provider here, with a timeout
  return new Response('Sent', { status: 200 })
})
```

Because this client carries the caller's token, RLS still applies: asking for another user's `noteId` returns nothing, and the function answers 404. Prefer this pattern — keep the user-scoped client for reads and writes so the database keeps enforcing the boundary for you.

Sometimes a step genuinely needs privilege. A service-role client inside the function bypasses RLS, so use it only after the function establishes the same boundary itself: verify the caller, authorize the requested resource, validate input, then perform the one privileged statement and nothing broader.

Secrets live in function configuration, never in code:

```
supabase secrets set RESEND_API_KEY=re_8fKm2Vw...
```

Read it with `Deno.env.get('RESEND_API_KEY')`. For the outbound call, set timeouts (`AbortSignal.timeout(5000)` on `fetch`), check the provider's response status, and decide what a retry means before adding one — an email sent twice is a bug you cannot unsend.

Test the denied paths before the happy one. A request with a missing session must get 401. A valid session sending another user's note ID must get 404. Only when both hold does the success case mean anything.

Separate environments and secrets

Keep local, preview, staging, and production resources distinct, link the CLI explicitly, and prevent destructive commands from targeting the wrong project.

The Supabase CLI will happily run a destructive command against the wrong project. Environment separation is what makes that mistake hard instead of easy.

Use separate projects, or the platform's supported branching workflows, for environments that

must not share users or data. Local development runs on the CLI stack; staging and production are distinct hosted projects with their own project references. Record each project reference and its purpose somewhere visible — without recording credentials next to it:

```
abcdefghijkl  production  (real users, never experiment here)
mnopqrstuvwx  staging     (disposable data, safe to break)
```

The CLI acts on whatever project the directory is linked to:

```
supabase link --project-ref mnopqrstuvwx
supabase projects list
#   LINKED   REFERENCE ID    NAME
#
#            abcdefghijkl    production
#           mnopqrstuvwx    staging
```

Check the marker before anything irreversible. CLI defaults differ between commands — some act on the local stack, some on the linked project — so pass local or linked targets explicitly for destructive or production-sensitive work:

```
supabase db reset --local            # rebuilds the local stack, always safe
supabase db push --linked --dry-run  # preview what would hit the linked project
```

The nightmare command is `supabase db reset --linked`. It rebuilds the linked project's database from your local migration files — on production that is total data loss, and the confirmation prompt is the only thing between you and it. Review the linked project before pushing migrations, every time, even when you are sure.

Secrets follow the same split. Each environment gets its own values, set where that environment runs:

```
supabase secrets set RESEND_API_KEY=re_8fKm2Vw... --project-ref mnopqrstuvwx
supabase secrets list --project-ref mnopqrstuvwx
```

Never reuse production credentials in staging. The whole point of the split is that a staging mistake cannot touch production data — a shared API key quietly reconnects the two.

Close by writing the release checklist the exercise asks for: the source environment, the target project, the migration preview output, the smoke test to run after deploy, and who owns the rollback. Five lines in a markdown file beat memory under pressure.

Observe, back up, and recover

Monitor database and platform signals, understand backup coverage, rehearse restore, check current limits, and keep provider-independent exports.

A production project you cannot observe or restore is a countdown. Set up both before the first real user, while mistakes are still free.

Watch what predicts trouble

Monitor errors, slow queries, connections, Auth failures, function logs, Storage failures, Realtime behavior, and usage limits. The dashboard's Reports and Logs pages cover most of it; for query time, `pg_stat_statements` is already available:

```
select calls, round(mean_exec_time) as avg_ms, query
```

```
from pg_stat_statements
order by mean_exec_time desc
limit 5;
```

Tie alerts to user-visible outcomes. ”Connections near the limit” and ”Auth error rate climbing” deserve attention; a dashboard where every moving number alerts trains you to ignore all of it.

Know what a backup covers

Hosted projects on paid plans get scheduled database backups, with point-in-time recovery available when your acceptable data loss is minutes rather than a day. Read the current plan's coverage and retention instead of assuming — limits change, and the moment to discover yours is not during an incident.

Database backups do not automatically mean every Storage object or external service is recoverable. Uploaded files live in object storage — the database only holds their metadata. If losing user uploads is unacceptable, keep exports or replication for those buckets too, wherever the recovery goal requires them.

Keep one provider-independent export in the rotation:

```
supabase db dump -f prod-2026-08-03.sql
# Dumping schemas from remote database...
```

That file restores into any PostgreSQL anywhere, which is the exit you hope never to need.

The drill is the proof

Restore a disposable backup or dump into a separate project — never over production. Then reconnect a copy of the application safely and verify the things that break quietly: Auth-linked ownership (do `notes.user_id` values still match users in the restored auth schema?), Storage files, and one complete workflow end to end, sign-in included.

The failure you are hunting: a restore that brings back your tables but not the auth data they reference, leaving every `user_id` pointing at nobody. Counts match, rows look fine, and no user can see their own data. Finding that in a drill costs an afternoon. Finding it in production costs you users.

Check your understanding: functions and production

Check the key models, decisions, commands, security boundaries, and failure cases from functions and production.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/supabase/.