database # deployment # guide

Deploying a Web API with Database

TCSS 460 — Client/Server Programming

Your API has a database now. In Sprint 0, you deployed a stateless Express server — it responded to requests, but it did not store anything. Since then, you have added PostgreSQL, Prisma, and real data. Deploying that version means your cloud environment needs a database too, not just a running server. This guide walks you through provisioning a managed PostgreSQL instance, connecting it to your deployed API, and running Prisma migrations in production.

> **Prerequisite: You Have Already Deployed**
>
> This guide assumes you have already deployed a stateless API to Render (or another platform) using the Deploying a Simple Web API guide. If you have not done that yet, start there. Everything here builds on that foundation.

---

1 What Changes When You Add a Database

Your Sprint 0 deployment had three moving parts: your code, a build command, and a `PORT` environment variable. Adding a database introduces three more:

What	Why
A managed PostgreSQL instance	Your local PostgreSQL server does not exist in the cloud — you need a database hosted somewhere accessible
A `DATABASE_URL` environment variable	Your Prisma configuration reads the connection string from `process.env.DATABASE_URL` — the cloud version needs this set to the production database
A migration step	Your database tables do not exist yet in production — Prisma migrations need to run before your API can serve requests

Everything else — your route handlers, your Prisma schema, your middleware — is identical. The code does not know or care whether PostgreSQL is running on your laptop or on a server in Virginia. It reads from `DATABASE_URL`, connects, and queries. That abstraction is the entire point.

1.1 Where Does the Database Live?

You have two broad options:

- **Platform-managed database:** Your hosting platform (e.g., Render) provisions a PostgreSQL instance alongside your web service. Convenient — everything is in one dashboard. But often limited by free-tier restrictions.
- **External database service:** A separate service (e.g., Supabase, Neon) hosts your database. Your API connects to it over the internet using a connection string. More flexible — you can switch hosting platforms without touching your database.

Either way, your API connects using a `DATABASE_URL`. The difference is where you get that URL from.

---

2 Updating Your Build for Prisma

Before you provision a database, update your deployment's build configuration. This applies regardless of which database provider you choose.

2.1 The Build Command

In Sprint 0, your build command was:

```
npm install && npm run build
```

That installed dependencies and compiled TypeScript. Now Prisma needs an additional step: generating the Prisma Client. The Prisma Client is the TypeScript code that your application imports to query the database — it is generated from your `schema.prisma` file and must exist before your TypeScript compiles.

Update your build command to:

```
npm install && npx prisma generate && npm run build
```

The new step — `npx prisma generate` — reads your Prisma schema and generates the client code. Without it, your TypeScript compilation will fail with import errors because the generated Prisma Client module does not exist yet.

2.2 The Migration Step

Your local development workflow uses `npx prisma migrate dev`, which creates and applies migrations interactively. In production, you use a different command:

```
npx prisma migrate deploy
```

This command applies all pending migrations to the production database without prompting. It does not create new migrations — it only runs the ones already committed to your `prisma/migrations/` directory.

Where does this command run? You have two options depending on your platform:

Approach	How	When to Use
Pre-deploy command	A separate command that runs after the build but before the server starts	Render supports this in the service settings
Start command	Chain it before your server: `npx prisma migrate deploy && npm start`	Works on any platform

For Render, the cleanest approach is to use the **Pre-Deploy Command** field in your service settings. Set it to `npx prisma migrate deploy`. This runs once per deployment, applies any new migrations, and then Render starts your server.

If your platform does not have a pre-deploy step, chain the commands in your start script:

```
npx prisma migrate deploy && npm start
```

This works but runs migrations on every server restart, not just on deploys. For a course project, that is fine — the `migrate deploy` command is a no-op when there are no pending migrations.

2.3 Check Your Dependencies

Make sure `prisma` is in your `dependencies` in `package.json` — not just in `devDependencies`. Some platforms prune dev dependencies after the build step, which would remove the `prisma` CLI before your migration command runs.

```
npm install prisma --save
```

If `prisma` is already in your `dependencies`, you are fine. If it is only in `devDependencies`, move it. The `@prisma/client` package should also be in `dependencies` — the starter project already does this.

> ⚠ **The Most Common Build Failure**
>
> If your deploy fails with an error about `prisma` not being found or Prisma Client imports failing, check two things: (1) `prisma` is in `dependencies`, and (2) `npx prisma generate` is in your build command. These account for the majority of first-time database deployment failures.

---

3 Option A: Render PostgreSQL

If you deployed your API to Render in Sprint 0, adding a database is straightforward — Render offers managed PostgreSQL instances that live alongside your web service in the same dashboard.

> ❗ **These Instructions Are a Reference, Not a Single Source of Truth**
>
> Cloud platform UIs change frequently. The steps below were probably accurate at the time of writing, but buttons may have moved, options may have been renamed, and workflows may have changed. Use these instructions as a general guide — if something does not match what you see, consult Render's official documentation for the current process.
>
> If you are using an AI coding assistant to help with deployment, tell it to read the platform's most recent documentation. AI models have a training data cutoff and may give you outdated instructions otherwise.

> ⚠ **Render Free Tier: 30-Day Database Limit**
>
> Render's free PostgreSQL tier expires after **30 days**, with a 14-day grace period to upgrade or export before the database is deleted. This is enough time to get your deployment working and verify everything connects, but it is not a permanent solution for the quarter. Plan to migrate to an external database (see Section 6) before the timer runs out.

3.1 Create a PostgreSQL Instance

1. From the Render dashboard, click **New** → **PostgreSQL**
2. Configure the database:

Setting	Value
Name	Your team's database name (e.g., `tcss460-team-5-db`)
Database	Leave as default or set a name (e.g., `tcss460`)
User	Leave as default
Region	Same region as your web service (e.g., Oregon)
Instance Type	Free

1. Click **Create Database**

Render provisions the database and displays its connection details. You need the **Internal Database URL** if your web service is also on Render (same region), or the **External Database URL** if connecting from outside Render.

3.2 Connect Your Web Service

1. Go to your existing web service in the Render dashboard
2. Click the **Environment** tab
3. Add a new environment variable:

Key	Value
`DATABASE_URL`	The **Internal Database URL** from your PostgreSQL instance

The internal URL is faster and free of data transfer charges because traffic stays within Render's network. It looks something like:

```
postgresql://user:password@dpg-abc123-a/tcss460
```

1. Update your **Build Command** to include Prisma generation:

```
npm install && npx prisma generate && npm run build
```

1. Add a **Pre-Deploy Command** for migrations:

```
npx prisma migrate deploy
```

1. Click **Save Changes** — Render will redeploy your service automatically

3.3 Verify the Connection

After the deploy completes, check the build and runtime logs:

1. **Build log:** Look for `prisma generate` output — you should see it generating the Prisma Client successfully
2. **Pre-deploy log:** Look for migration output — you should see your migrations being applied (e.g., `Migration 20240401_init applied`)
3. **Runtime log:** Your server should start without database connection errors

Then hit an endpoint that reads from the database. If it returns data (or an empty array for a fresh database), the connection is working.

> **Try It Yourself**
>
> 1. Open your deployed API's `/api-docs` in a browser
> 2. Use Scalar to send a GET request to an endpoint that queries the database
> 3. Send a POST request to create a record, then GET it back — does the data persist?

---

4 Option B: Supabase

Supabase is a hosted PostgreSQL service with a free tier that lasts beyond 30 days — making it a better fit for a quarter-long project. Your API does not run on Supabase; only the database lives there. Your Express server stays on Render (or wherever you deployed it) and connects to Supabase using a standard PostgreSQL connection string.

> **These Instructions Are a Reference, Not a Single Source of Truth**
>
> Cloud platform UIs change frequently. The steps below were probably accurate at the time of writing. If something does not match what you see, consult Supabase's official documentation for the current process.

4.1 Supabase Free Tier Limits

Before you start, know what you are getting:

Resource	Free Tier Limit
Database storage	500 MB
Projects	2 active projects
Inactivity pause	Projects pause after 1 week of inactivity
Bandwidth	5 GB database egress per month

For a course project, 500 MB is more than enough. The inactivity pause is the main limitation — if nobody hits your API for a week, Supabase pauses the database and the next request takes a few seconds while it wakes up. You can unpause manually from the dashboard. During active development, this rarely triggers.

4.2 Create a Supabase Project

1. Go to supabase.com and create a free account
2. Click **New Project**
3. Configure the project:

Setting	Value
Project name	Your team's project name (e.g., `tcss460-team-5`)
Database password	A strong password — save this somewhere, you will need it for the connection string
Region	US West (or closest to your hosting platform)
Plan	Free

1. Click **Create new project** and wait for provisioning (takes about a minute)

4.3 Get the Connection String

Once your project is ready:

1. Go to **Project Settings** → **Database** (or click **Connect** in the top navigation)
2. Find the **Connection string** section
3. Select the **URI** tab
4. Copy the connection string — it looks something like:

```
postgresql://postgres.[project-ref]:[password]@aws-0-us-west-
1.pooler.supabase.com:5432/postgres
```

Replace `[password]` with the database password you set when creating the project.

> **Direct Connection vs. Connection Pooling**
>
> Supabase provides two connection modes: **direct** (port 5432) and **transaction pooling** via Supavisor (port 6543). For a course project with low traffic, the direct connection is simpler and works with Prisma migrations out of the box. Use the direct connection string unless you have a reason not to.

4.4 Connect Your Web Service

1. Go to your web service on Render (or your hosting platform)
2. Open the **Environment** tab
3. Add (or update) the environment variable:

Key	Value
`DATABASE_URL`	The connection string from Supabase

1. Make sure your build command and pre-deploy command are set (see Section 2):
   - **Build Command:** `npm install && npx prisma generate && npm run build`
   - **Pre-Deploy Command:** `npx prisma migrate deploy`
2. Save and redeploy

4.5 Verify the Connection

Same verification as with Render PostgreSQL — check the logs, then hit a database-backed endpoint:

1. **Build log:** `prisma generate` succeeds
2. **Pre-deploy log:** Migrations apply successfully
3. **Runtime:** Server starts, database endpoints respond

You can also verify from the Supabase side: go to the **Table Editor** in your Supabase dashboard. After migrations run, you should see your tables listed there. You can browse rows directly — useful for debugging.

> **Try It Yourself**
>
> 1. Deploy with the Supabase `DATABASE_URL` set
> 2. Hit a POST endpoint to create a record
> 3. Open the Supabase Table Editor — do you see the new row?
> 4. Hit a GET endpoint — does it return the data you just created?

---

5 The Key Insight: Only the URL Changes

Compare what you did in Option A and Option B. The Prisma schema is identical. The build command is identical. The migration command is identical. Your route handlers, your middleware, your error handling — all identical. The only difference between Render PostgreSQL and Supabase is the value of one environment variable: `DATABASE_URL`.

```
# Render PostgreSQL (internal)
DATABASE_URL="postgresql://user:pass@dpg-abc123-a/tcss460"

# Supabase
DATABASE_URL="postgresql://postgres.xyzabc:pass@aws-0-us-west-
1.pooler.supabase.com:5432/postgres"
```

This is not a coincidence. It is the result of two design decisions you have been using all quarter:

1. **Prisma abstracts the database.** Your code calls `prisma.user.findMany()`, not raw SQL against a specific server. Prisma handles the connection details.
2. **Environment variables separate configuration from code.** Your code reads `process.env.DATABASE_URL`. Where that URL points — your laptop, Render, Supabase, anywhere — is a deployment decision, not a code decision.

This means switching databases is a configuration change, not a code change. No new branches, no pull requests, no rewriting queries. Change the URL, redeploy, done.

> **Gen AI & Learning: Why Abstraction Layers Matter**
>
> When you ask an AI coding assistant to "connect my API to a database," it will generate Prisma queries or raw SQL — code that works against any PostgreSQL instance. The AI does not need to know whether you are using Render, Supabase, or a Raspberry Pi under your desk. That is the power of abstraction: the code is portable because it depends on a standard interface (a PostgreSQL connection string), not a specific vendor.
>
> This same principle applies at every layer of your stack. Express routes do not care whether the front-end is React or plain HTML. Your front-end does not care whether the API is written in TypeScript or Go. Each layer communicates through a standard interface — HTTP, SQL, environment variables — and everything behind that interface is swappable. Understanding this is more valuable than memorizing any one platform's setup steps.

---

6 Migrating from Render to Supabase

If you started with Render PostgreSQL to get up and running quickly, you will need to migrate before the 30-day free tier expires. The good news: you already understand that this is a one-variable change.

6.1 When to Migrate

Render shows the expiration date on your PostgreSQL instance's dashboard. You will also receive email warnings. Do not wait until the last day — give yourself time to verify the new connection.

A reasonable timeline:

1. **Day 1–7:** Use Render PostgreSQL. Get your deployment working, verify everything connects, fix any migration issues.
2. **Before Day 25:** Set up Supabase (Section 4), swap the `DATABASE_URL`, redeploy, verify.
3. **Day 30:** Render deletes the free database. You are already on Supabase. Nothing breaks.

6.2 The Migration Steps

1. **Create a Supabase project** if you have not already (Section 4.2)

2. **Copy the Supabase connection string** (Section 4.3)
3. **Update** `DATABASE_URL` on your Render web service's Environment tab — replace the Render internal URL with the Supabase URL
4. **Redeploy** — Render will rebuild and run your pre-deploy migration command against the new database
5. **Verify** — your migrations create the tables fresh on Supabase; hit your endpoints to confirm

> ⚠ **Data Does Not Transfer Automatically**
>
> Swapping the `DATABASE_URL` points your API at an empty Supabase database. Your Prisma migrations will recreate all the tables, but any data in the old Render database is gone. For a course project, this is usually fine — your data is test data. If you do need to preserve data, you would need to export from Render and import into Supabase, which is beyond the scope of this guide.

6.3 Verify the Cutover

After redeploying with the new `DATABASE_URL`:

- Build log shows `prisma generate` succeeding
- Pre-deploy log shows migrations applying to the new database
- Server starts without connection errors
- A GET endpoint returns data (or an empty array for a fresh database)
- A POST endpoint creates a record that persists across requests
- Supabase Table Editor shows your tables and data

Once everything checks out, you can let the Render PostgreSQL instance expire without concern.

---

7 Verifying Your Database Deployment

Whether you use Render PostgreSQL, Supabase, or another provider, run through this checklist after every deployment that touches the database:

7.1 Post-Deploy Checklist

- **Build succeeded** — `prisma generate` ran without errors
- **Migrations applied** — check the pre-deploy or start logs for migration output
- **Server started** — your startup message appears in the runtime logs (e.g., `Server running on port 10000`)
- **Health check passes** — `GET /health` returns a response
- **Database reads work** — a GET endpoint that queries the database returns data or an empty array
- **Database writes work** — a POST endpoint creates a record; a subsequent GET returns it
- **API docs load** — `/api-docs` renders your OpenAPI spec (Scalar should display database-backed endpoints)
- **No connection errors in logs** — check the runtime logs for Prisma connection failures or timeout messages

7.2 Quick Smoke Test

The fastest way to verify: open your deployed `/api-docs`, use Scalar to POST a new record, then GET it back. If the round trip works, your database connection, migrations, and Prisma Client are all functioning correctly.

If the GET returns the record you just created, you are done. If it returns an empty array or an error, check the logs.

---

8 Common Database Deployment Issues

When a database-backed deployment fails, the error is almost always in one of these categories. Check them in order.

8.1 `DATABASE_URL` Not Set

Symptom: Server crashes on startup with a Prisma connection error, or `prisma migrate deploy` fails with "environment variable not found."

Fix: Check your hosting platform's environment variables. Make sure `DATABASE_URL` is set and the value is correct — no extra spaces, no missing password, no typos.

8.2 Prisma Client Not Generated

Symptom: Build fails with TypeScript import errors referencing Prisma Client modules that do not exist.

Fix: Add `npx prisma generate` to your build command, before `npm run build`:

```
npm install && npx prisma generate && npm run build
```

8.3 Migrations Did Not Run

Symptom: Server starts, but database queries fail with errors like `relation "users" does not exist` or `table not found`.

Fix: Your migrations have not been applied to the production database. Check that your pre-deploy command (`npx prisma migrate deploy`) is set, or that your start command includes the migration step. Check the deploy logs — if the migration command ran, you should see output listing which migrations were applied.

Also verify that your `prisma/migrations/` directory is committed to git. The `migrate deploy` command reads migration files from your repository — if they are in `.gitignore`, production has nothing to apply.

8.4 Connection Refused or Timed Out

Symptom: `prisma migrate deploy` or your server hangs, then fails with a connection timeout or "connection refused" error.

Fix: Check the connection string in `DATABASE_URL`:

- **Host and port** — are they correct for your provider?
- **Credentials** — is the password right? Did you replace the `[password]` placeholder?
- **Network access** — some providers restrict connections by IP. Supabase allows connections from anywhere by default; Render internal URLs only work from within Render's network
- **SSL** — some providers require SSL. If you see SSL-related errors, try adding `?sslmode=require` to the end of your connection string

8.5 `prisma` CLI Not Found in Production

Symptom: The pre-deploy or start command fails with `prisma: command not found` or `npx: prisma not found`.

Fix: Move `prisma` from `devDependencies` to `dependencies` in your `package.json`:

```
npm install prisma --save
```

Some platforms prune `devDependencies` after the install step. If `prisma` is only a dev dependency, it gets removed before your migration command runs.

---

9 Other Options to Explore

Render PostgreSQL and Supabase are not the only choices. Your team is free to investigate and use any managed PostgreSQL provider. The principle is the same everywhere: get a connection string, set `DATABASE_URL`, run migrations, deploy.

Provider	What It Offers	Notes
Neon	Serverless PostgreSQL with a generous free tier	Scales to zero when idle — good for intermittent usage; branching feature lets you create database copies for testing
Railway	Simple UX, database + hosting in one platform	Limited free credits (usage-based); nice developer experience
Heroku Postgres	Managed PostgreSQL with Student Pack credits	Essential-0 plan ($5/month) covered by GitHub Student Developer Pack; reliable, well-documented
Aiven	Managed database platform with a free tier	PostgreSQL among several supported databases; free plan includes one service

Evaluating a Provider

When exploring a database provider, check three things: (1) Does the free tier last long enough for the quarter? (2) Does it provide a standard PostgreSQL connection string? (3) Can Prisma connect to it without special configuration? If yes to all three, it will work with your project.

Your instructor will not troubleshoot platform-specific issues — that responsibility belongs to your team. Choose a platform your group is comfortable supporting.

---

10 Summary

Concept	Key Point
What changes	A managed database instance, a `DATABASE_URL` env var, and a migration step — your code stays the same
Build command	Add `npx prisma generate` before `npm run build` to generate the Prisma Client during deployment
Migration command	`npx prisma migrate deploy` applies committed migrations to production — use a pre-deploy command or chain it with your start command
`prisma` in dependencies	Must be in `dependencies`, not `devDependencies`, or the CLI may be pruned before migrations run
Render PostgreSQL	Quick start if already on Render — 30-day free tier limit with 14-day grace period
Supabase	External PostgreSQL with a longer-lasting free tier (500 MB, pauses after 1 week inactivity)
Only the URL changes	Switching database providers means changing one environment variable — no code changes
Migrating providers	Swap `DATABASE_URL`, redeploy, re-run migrations — tables are recreated, but data does not transfer automatically
Verify after deploy	Build log → migration log → server startup → health check → database read → database write
Common failures	Missing `DATABASE_URL`, missing `prisma generate` in build, migrations not committed, wrong connection string

---

11 References

Official Documentation:

- Render — PostgreSQL Databases — Creating and managing PostgreSQL on Render
- Render — Pre-Deploy Commands — Running commands before the server starts
- Render — Environment Variables — Configuring env vars in the Render dashboard
- Supabase — Database Overview — Getting started with Supabase PostgreSQL
- Supabase — Connecting to Your Database — Connection strings and pooling options
- Prisma — Deploy with Prisma Migrate — Running migrations in production
- Prisma — Deploying to Render — Prisma-specific deployment guidance for traditional servers

Course Guides:

- Deploying a Simple Web API — Sprint 0 deployment (no database) — prerequisite for this guide
- Prisma ORM — Prisma schema, migrations, and CRUD operations
- PostgreSQL & Docker Setup — Local PostgreSQL setup and connection strings

---

12 Further Reading

External Resources

- The Twelve-Factor App — Backing Services — Why databases should be attached resources, swappable via configuration
- The Twelve-Factor App — Config — Why configuration belongs in environment variables, not code
- Prisma — Connection Management — How Prisma manages database connections (connection pooling, limits, timeouts)
- Supabase — Free Tier Limits — Current pricing and free tier details
- Neon — Free Tier — Alternative serverless PostgreSQL with a generous free plan

---

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.