database # guide

Prisma ORM

TCSS 460 — Client/Server Programming

You have written SQL by hand in the SQL Fundamentals guide and set up a running PostgreSQL database in the PostgreSQL & Docker Setup guide. You understand tables, columns, constraints, and CRUD queries. Now the question is: how do you connect your Express routes to that database? You could embed SQL strings in your TypeScript code, but there is a better way. Prisma is an ORM that lets you define your database schema in code, generates type-safe TypeScript from it, and gives you a clean API for querying your database — no SQL strings required.

> **About the Examples**
>
> This guide uses `User` and `Post` models throughout to illustrate Prisma concepts. These are not the models you will build in your group project — your project will have its own schema. The patterns and techniques shown here apply regardless of which models you define.

---

1 What is an ORM?

An **ORM** (Object-Relational Mapper) is a tool that sits between your application code and your database. Instead of writing raw SQL queries as strings, you work with TypeScript objects and method calls. The ORM translates those calls into SQL behind the scenes.

1.1 The Translation Layer

Consider what happens without an ORM. You write something like this to fetch a user:

```
const result = await pool.query('SELECT * FROM users WHERE id = $1',
[userId]);
const user = result.rows[0];
```

That SQL string is just a string — TypeScript does not know what columns exist, what types they are, or whether the query is even valid. You find out at runtime, when the query fails.

With Prisma, the same operation looks like this:

```
const user = await prisma.user.findUnique({ where: { id: userId } });
```

TypeScript knows that `user` has properties like `id`, `email`, and `name` — because Prisma generated those types from your schema. If you mistype a field name, the compiler catches it before you run the code.

1.2 Why Not Just Write SQL?

SQL is powerful and you should understand it. But embedding SQL strings in application code creates problems:

- **No compile-time safety.** A typo in a column name compiles fine and crashes at runtime.
- **No autocompletion.** Your editor cannot help you write queries against string-based SQL.
- **Mapping boilerplate.** You have to manually map database rows to TypeScript objects, handling type conversions and null checks yourself.
- **Migration management.** You have to track schema changes manually or build your own migration system.

An ORM solves all of these. You still need to understand the SQL that runs underneath — the SQL Fundamentals guide exists for that reason — but you do not need to write it by hand for every route.

---

2 Why Prisma?

There are several ORMs available for Node.js and TypeScript (Drizzle, TypeORM, Sequelize, Knex). This course uses Prisma because it offers the strongest combination of type safety, developer experience, and learning-friendly tooling.

2.1 Schema-as-Code

Your database schema lives in a single file — `prisma/schema.prisma`. This file is the **single source of truth** for your data model. When you change it, Prisma generates migrations (the SQL to update your database) and regenerates TypeScript types — automatically.

No more wondering whether the database matches the code. The schema file *is* the code.

2.2 Auto-Generated TypeScript Types

When you run `npx prisma generate`, Prisma reads your schema and generates TypeScript interfaces for every model. If your schema defines a `User` model with `id`, `email`, and `name` fields, you get a `User` type with exactly those properties and their correct TypeScript types. Your editor autocompletes field names, and the compiler rejects invalid queries.

2.3 Migrations Built In

Prisma includes a migration system. When you change your schema (add a field, rename a table, add a relation), you run one command and Prisma generates the SQL migration and applies it. You never write migration SQL by hand unless you want to.

2.4 Developer Experience

Prisma provides tooling that makes database work less painful:

- **Prisma Studio** — A visual database browser that launches in your browser
- **Clear error messages** — Prisma errors tell you what went wrong and often suggest how to fix it
- **Rich documentation** — The Prisma docs are among the best in the ecosystem

---

3 Setting Up Prisma

> **Lecture Code vs. Group Project**
>
> The lecture demo repository will already have Prisma configured — you do not need to set it up yourself for lecture examples. However, your group project starter template does **not** include Prisma. You will follow the steps in this section to add Prisma to your group project in a future sprint.

This section walks through adding Prisma to an existing Express project. You should already have a Node.js project with TypeScript configured and a running PostgreSQL database.

3.1 Install Dependencies

Prisma requires three packages:

```
npm install prisma --save-dev
npm install @prisma/client @prisma/adapter-pg pg
```

- `prisma` — The CLI tool (dev dependency). You use this to generate code, run migrations, and launch Studio.
- `@prisma/client` — The runtime library that your application code imports.
- `@prisma/adapter-pg` — The driver adapter that connects Prisma Client to PostgreSQL.
- `pg` — The underlying PostgreSQL driver for Node.js.

You also need the TypeScript type definitions for `pg`:

```
npm install @types/pg --save-dev
```

3.2 Initialize Prisma

Run the init command to scaffold the Prisma files:

```
npx prisma init
```

This creates several files:

File	Purpose
`prisma/schema.prisma`	Your data model — models, fields, relations
`prisma.config.ts`	CLI configuration — database URL, migration settings
`.env`	Environment variables (database connection string)

3.3 The Configuration File

The generated `prisma.config.ts` file tells the Prisma CLI where to find your schema and how to connect to your database:

```
import 'dotenv/config';
import { defineConfig } from 'prisma/config';

export default defineConfig({
    schema: 'prisma/schema.prisma',
    migrations: {
```

```
        path: 'prisma/migrations',
    },
    datasource: {
        url: process.env.DATABASE_URL!,
    },
});
```

This file is used by CLI commands like `prisma migrate dev` and `prisma studio`. It reads the `DATABASE_URL` from your `.env` file.

3.4 The Connection String

Open `.env` and set your database connection string:

```
DATABASE_URL="postgresql://username:password@localhost:5432/mydb"
```

The format is: `postgresql://USER:PASSWORD@HOST:PORT/DATABASE`. If you followed the PostgreSQL & Docker Setup guide, you already have this.

> **Never Commit `.env` Files**
>
> Your `.gitignore` should include `.env`. The connection string contains your database password. If you commit it to git, the password is in your repository history forever — even if you delete the file later.

---

4 The Prisma Schema

The `prisma/schema.prisma` file is where you define your data model. Every table in your database corresponds to a `model` block in this file. Every column corresponds to a field.

4.1 Generator and Datasource Blocks

At the top of your schema, you will see two configuration blocks:

```
generator client {
    provider = "prisma-client"
    output   = "../src/generated/prisma"
}

datasource db {
```

```
    provider = "postgresql"
}
```

The **generator** block tells Prisma to generate the TypeScript client. The `output` path specifies where the generated code goes — this must be set explicitly. The **datasource** block declares that you are using PostgreSQL. The actual connection URL lives in `prisma.config.ts`, not here.

> **The `output` Path Matters**
>
> You must set the `output` field in the generator block. Prisma generates TypeScript types and runtime code at this location. Throughout this guide, we assume the output path is `../src/generated/prisma`. When you import from Prisma Client, you import from this path.

4.2 Defining a Model

A model maps to a database table. Each line inside the model defines a column:

```
model User {
    id        Int      @id @default(autoincrement())
    email     String   @unique
    name      String
    active    Boolean  @default(true)
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt
}
```

Reading this model: there is a `users` table (Prisma pluralizes and lowercases the model name by default) with six columns. Let's look at each field.

4.3 Field Types

Prisma field types map to both database column types and TypeScript types:

Prisma Type	PostgreSQL Type	TypeScript Type
`String`	`text`	`string`
`Int`	`integer`	`number`
`Float`	`double precision`	`number`

Prisma Type	PostgreSQL Type	TypeScript Type
`Boolean`	`boolean`	`boolean`
`DateTime`	`timestamp(3)`	`Date`
`BigInt`	`bigint`	`bigint`
`Json`	`jsonb`	`JsonValue`

A `?` after the type makes the field optional (nullable): `name String?` means the `name` column can be `NULL` in the database and `string | null` in TypeScript.

4.4 Field Attributes

Attributes modify field behavior. They start with `@`:

Attribute	Purpose	Example
`@id`	Marks the primary key	`id Int @id`
`@default(...)`	Sets a default value	`@default(autoincrement())`, `@default(now())`, `@default(true)`
`@unique`	Adds a unique constraint	`email String @unique`
`@updatedAt`	Auto-updates to current timestamp on every update	`updatedAt DateTime @updatedAt`
`@map("...")`	Maps to a different column name in the database	`createdAt DateTime @map("created_at")`

4.5 Model Attributes

Model-level attributes start with `@@` and apply to the entire table:

```
model User {
    id    Int    @id @default(autoincrement())
    email String @unique
    name  String
```

```
    @@map("users")
}
```

`@@map("users")` tells Prisma that this model maps to a table called `users` in the database. This is useful when your database uses `snake_case` table names but you want `PascalCase` model names in TypeScript.

4.6 Relations Between Models

Relational databases are relational — tables reference each other through foreign keys. Prisma expresses this with relation fields:

```
model User {
    id    Int    @id @default(autoincrement())
    email String @unique
    name  String
    posts Post[]
}

model Post {
    id        Int      @id @default(autoincrement())
    title     String
    content   String?
    published Boolean  @default(false)
    authorId  Int
    author    User     @relation(fields: [authorId], references: [id])
}
```

Reading this: a `User` has many `Post` records (the `Post[]` field). Each `Post` belongs to one `User` through the `authorId` foreign key. The `@relation` attribute tells Prisma which field holds the foreign key (`authorId`) and which field it references (`id` on `User`).

In the database, only `Post` gets a real column (`author_id`). The `posts` field on `User` is virtual — it does not exist as a column, but Prisma uses it to let you query related data.

Try It Yourself

1. Open `prisma/schema.prisma` in your project
2. Define a `User` model with `id`, `email`, `name`, `createdAt`, and `updatedAt` fields
3. Add a `Post` model with `id`, `title`, `content`, `published`, `createdAt`, and a relation to `User`
4. Does your editor give you syntax highlighting and autocompletion? If you have the Prisma VS Code extension, it should.

---

5 Migrations

A **migration** is a set of SQL statements that changes your database schema — creating tables, adding columns, modifying constraints. Prisma generates these migrations automatically from your schema file.

5.1 Creating Your First Migration

After defining your models in `schema.prisma`, run:

```
npx prisma migrate dev --name init
```

This command does three things:

1. **Generates a migration file** — SQL that creates the tables defined in your schema
2. **Applies the migration** — Runs that SQL against your database
3. **Marks the migration as applied** — Records it in a `_prisma_migrations` table so it is not run again

> **Generate Separately**
>
> In Prisma 7, `migrate dev` no longer runs `prisma generate` automatically. After running your migration, you need to run `npx prisma generate` to regenerate the TypeScript client.

5.2 What a Migration File Looks Like

Prisma creates a timestamped folder inside `prisma/migrations/`:

```
prisma/
  migrations/
    20260410120000_init/
      migration.sql
```

The `migration.sql` file contains the SQL that Prisma generated:

```
-- CreateTable
CREATE TABLE "User" (
    "id" SERIAL NOT NULL,
    "email" TEXT NOT NULL,
    "name" TEXT NOT NULL,
    "active" BOOLEAN NOT NULL DEFAULT true,
    "createdAt" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
    "updatedAt" TIMESTAMP(3) NOT NULL,

    CONSTRAINT "User_pkey" PRIMARY KEY ("id")
);

-- CreateIndex
CREATE UNIQUE INDEX "User_email_key" ON "User"("email");
```

This is regular SQL — the same kind you wrote in the SQL Fundamentals guide. Prisma wrote it for you by reading your schema.

5.3 When to Create a New Migration

Create a new migration whenever you change `schema.prisma`:

- Added a new model? → `npx prisma migrate dev --name add-post-model`
- Added a field to an existing model? → `npx prisma migrate dev --name add-user-bio`
- Changed a field type or constraint? → `npx prisma migrate dev --name make-email-unique`

The `--name` flag is a human-readable label. Use it to describe what changed — future-you will thank you when reading the migration history.

5.4 The Nuclear Option: `migrate reset`

If your development database gets into a bad state (schema mismatches, corrupted data, conflicting migrations), you can start over:

```
npx prisma migrate reset
```

> **This Drops Everything**
>
> `migrate reset` drops the entire database and recreates it from scratch by replaying all migrations. All data is lost. This is fine in development — you should be able to recreate test data easily. Never run this against a production database.

---

6 Prisma Client — CRUD Operations

With your schema defined and migrations applied, you interact with the database through **Prisma Client** — the auto-generated TypeScript API. Every model in your schema becomes a property on the client with methods for creating, reading, updating, and deleting records.

6.1 Setting Up the Client

Create a file that initializes and exports the Prisma Client instance. This file will be imported by your route handlers:

```
// src/prisma.ts
import { PrismaClient } from './generated/prisma';
import { PrismaPg } from '@prisma/adapter-pg';

const connectionString = process.env.DATABASE_URL!;
const adapter = new PrismaPg({ connectionString });

export const prisma = new PrismaClient({ adapter });
```

The `PrismaPg` adapter connects Prisma Client to your PostgreSQL database using the `pg` driver. You create one client instance and reuse it across your entire application — do not create a new client for every request.

> ⚠ **One Client Instance**
>
> Creating multiple Prisma Client instances opens multiple database connection pools. This exhausts your database's connection limit quickly. Always create one instance in a shared module and import it everywhere.

6.2 Create — `create()`

Insert a new record:

```
const user = await prisma.user.create({
    data: {
        email: 'alice@example.com',
        name: 'Alice',
    },
});
```

The `data` object must include all required fields (fields without `@default` or `?`). The return value is the created record, including auto-generated fields like `id` and `createdAt`.

SQL equivalent:

```
INSERT INTO "User" (email, name, "createdAt", "updatedAt")
VALUES ('alice@example.com', 'Alice', NOW(), NOW())
RETURNING *;
```

6.3 Read — `findMany()` and `findUnique()`

Retrieve multiple records:

```
const users = await prisma.user.findMany();
```

SQL equivalent: `SELECT * FROM "User";`

Retrieve a single record by a unique field:

```
const user = await prisma.user.findUnique({
    where: { id: 1 },
});
```

SQL equivalent: `SELECT * FROM "User" WHERE id = 1;`

`findUnique` only works with fields marked `@id` or `@unique`. If the record does not exist, it returns `null`.

6.4 Update — `update()`

Update an existing record:

```
const updated = await prisma.user.update({
    where: { id: 1 },
    data: { name: 'Alice Updated' },
});
```

The `where` clause identifies which record to update (must be a unique field). The `data` object contains only the fields you want to change — omitted fields keep their current values.

SQL equivalent:

```
UPDATE "User" SET name = 'Alice Updated', "updatedAt" = NOW()
WHERE id = 1
RETURNING *;
```

Notice that Prisma automatically updates the `updatedAt` field (because of the `@updatedAt` attribute in the schema).

6.5 Delete — `delete()`

Remove a record:

```
const deleted = await prisma.user.delete({
    where: { id: 1 },
});
```

SQL equivalent: `DELETE FROM "User" WHERE id = 1 RETURNING *;`

The return value is the deleted record — useful for confirming what was removed.

6.6 The Pattern

Every Prisma Client method follows the same structure:

```
const result = await prisma.<model>.<operation>({
    where: { /* identify which record(s) */ },
    data:  { /* provide new values (create/update only) */ },
});
```

Once you learn this shape, every CRUD operation is a variation of the same pattern.

> **Try It Yourself**
>
> 1. Set up the Prisma Client in your project using the pattern in Section 6.1
> 2. Create a simple script that creates a user, reads it back, updates the name, and deletes it
> 3. After each operation, log the result — notice how Prisma always returns the affected record
> 4. Try creating a user with a duplicate email — what error do you get?

---

7 Filtering and Sorting

The `findMany` examples so far return all records. In practice, you need to filter, sort, and limit results. Prisma provides a rich query API for this.

7.1 Basic Filtering with `where`

Filter records using the `where` option:

```
// Find all active users
const activeUsers = await prisma.user.findMany({
    where: { active: true },
});

// Find a user by email
const user = await prisma.user.findUnique({
    where: { email: 'alice@example.com' },
});
```

7.2 Filter Operators

For more complex filtering, Prisma provides operators inside the `where` clause:

```
const users = await prisma.user.findMany({
    where: {
        email: { contains: 'example.com' },
    },
});
```

Common operators:

Operator	SQL Equivalent	Example
`equals`	`=`	`{ name: { equals: 'Alice' } }`
`contains`	`LIKE '%...%'`	`{ email: { contains: 'gmail' } }`
`startsWith`	`LIKE '...%'`	`{ name: { startsWith: 'A' } }`
`endsWith`	`LIKE '%...'`	`{ email: { endsWith: '.edu' } }`
`gt`	`>`	`{ age: { gt: 18 } }`
`gte`	`>=`	`{ age: { gte: 21 } }`
`lt`	`<`	`{ age: { lt: 65 } }`
`lte`	`<=`	`{ age: { lte: 30 } }`
`in`	`IN (...)`	`{ role: { in: ['ADMIN', 'EDITOR'] } }`
`not`	`!=`	`{ name: { not: 'Bob' } }`

You can combine multiple conditions — they are joined with `AND` by default:

```
const users = await prisma.user.findMany({
    where: {
        active: true,
        email: { contains: 'example.com' },
    },
});
```

SQL equivalent: `SELECT * FROM "User" WHERE active = true AND email LIKE '%example.com%';`

7.3 Sorting with `orderBy`

Sort results using `orderBy`:

```
const users = await prisma.user.findMany({
    orderBy: { createdAt: 'desc' },
});
```

SQL equivalent: `SELECT * FROM "User" ORDER BY "createdAt" DESC;`

You can sort by multiple fields by passing an array:

```
const users = await prisma.user.findMany({
    orderBy: [
        { active: 'desc' },
        { name: 'asc' },
    ],
});
```

7.4 Pagination with `skip` and `take`

Prisma supports offset-based pagination using `skip` and `take`:

```
const users = await prisma.user.findMany({
    skip: 20,   // skip the first 20 records
    take: 10,   // return the next 10
    orderBy: { id: 'asc' },
});
```

SQL equivalent: `SELECT * FROM "User" ORDER BY id ASC LIMIT 10 OFFSET 20;`

Full Pagination Coverage

This section covers only the basics. For a complete treatment — including response metadata, composable filtering with the spread operator pattern, cursor-based pagination, and building production-ready paginated endpoints — see the Pagination & Filtering guide.

---

8 Relations and Includes

The real power of a relational database is in its *relations*. Prisma makes querying related data straightforward with `include` and `select`.

8.1 Including Related Data

By default, Prisma queries return only the fields of the model you query — not related models. To include related data, use `include`:

```
const user = await prisma.user.findUnique({
    where: { id: 1 },
    include: { posts: true },
});
```

The result includes the user's data *and* all of their posts:

```
{
    "id": 1,
    "email": "alice@example.com",
    "name": "Alice",
    "posts": [
        { "id": 1, "title": "First Post", "published": true },
        { "id": 2, "title": "Draft", "published": false }
    ]
}
```

Without `include: { posts: true }`, the `posts` field would not exist in the result at all.

SQL equivalent: Prisma runs a `JOIN` or a second query behind the scenes — you do not need to write the join yourself.

8.2 Nested Reads — Filtering Related Data

You can filter, sort, and limit included relations:

```
const user = await prisma.user.findUnique({
    where: { id: 1 },
    include: {
        posts: {
            where: { published: true },
            orderBy: { createdAt: 'desc' },
            take: 5,
        },
    },
});
```

This returns the user with only their 5 most recent published posts. The `where`, `orderBy`, and `take` options inside `include` work the same as they do in a top-level `findMany`.

8.3 Nested Writes — Creating Related Data Together

You can create a parent and its children in a single operation:

```
const user = await prisma.user.create({
    data: {
        email: 'bob@example.com',
        name: 'Bob',
        posts: {
            create: [
                { title: 'Hello World', published: true },
                { title: 'Work in Progress', published: false },
            ],
        },
    },
    include: { posts: true },
});
```

This creates one `User` record and two `Post` records, all in a single database transaction. If any part fails, everything rolls back — you never end up with a user but no posts or posts without a user.

8.4 `select` vs `include`

Both `select` and `include` control which fields appear in the result, but they work differently:

`include` adds related models to the result. You get *all* fields of the main model plus the included relations:

```
const user = await prisma.user.findUnique({
    where: { id: 1 },
    include: { posts: true },
});
// Returns: { id, email, name, active, createdAt, updatedAt, posts: [...] }
```

`select` picks specific fields. You get *only* the fields you select — nothing else:

```
const user = await prisma.user.findUnique({
    where: { id: 1 },
    select: {
        name: true,
        email: true,
        posts: {
            select: { title: true, published: true },
        },
    },
});
// Returns: { name, email, posts: [{ title, published }] }
```

Use `select` when you want to minimize the data returned — for example, when building a list view that only needs names and emails, not every field on the model.

You Cannot Use Both

`select` and `include` are mutually exclusive on the same query level. If you need to pick specific fields *and* include relations, use `select` and include the relation inside the `select` block (as shown above).

Try It Yourself

1. Add a `Post` model to your schema with a relation to `User` (see Section 4.6)
2. Run `npx prisma migrate dev --name add-posts` then `npx prisma generate`
3. Create a user with two posts using the nested write pattern from Section 8.3
4. Query the user with `include: { posts: true }` — verify the posts appear
5. Try the same query with `select` — pick only `name` and `posts.title`

---

9 Prisma in Express Routes

Now that you know how to query with Prisma Client, let's connect it to your Express routes. The pattern is consistent: the route handler receives a request, runs a Prisma query, and sends the result as JSON.

9.1 The Basic Pattern

Every database-backed route handler follows the same shape:

```
import { Request, Response } from 'express';
import { prisma } from '../prisma';

export const getUsers = async (request: Request, response: Response) => {
    try {
        const users = await prisma.user.findMany();
        response.json({ data: users });
    } catch (error) {
        response.status(500).json({ error: 'Failed to retrieve users' });
    }
};
```

Three things to notice:

1. `async` — Prisma queries return Promises, so the handler must be async.
2. `try/catch` — Database operations can fail (connection lost, constraint violated, etc.). Always wrap Prisma calls in a try/catch.
3. **Shared client** — The `prisma` instance is imported from a shared module (Section 6.1), not created inside the handler.

9.2 A Full CRUD Example

Here is a complete set of route handlers for a `User` resource, defined in a controllers file (e.g., `src/controllers/users.ts`):

```
// src/controllers/users.ts
import { Request, Response } from 'express';
import { prisma } from '../prisma';

// GET /users
export const getUsers = async (request: Request, response: Response) => {
    try {
        const users = await prisma.user.findMany({
            orderBy: { id: 'asc' },
        });
        response.json({ data: users });
    } catch (error) {
        response.status(500).json({ error: 'Failed to retrieve users' });
    }
};

// GET /users/:id
export const getUserById = async (request: Request, response: Response) => {
    const id = Number(request.params.id);
    try {
        const user = await prisma.user.findUnique({ where: { id } });
        if (!user) {
            response.status(404).json({ error: 'User not found' });
            return;
```

```
        }
        response.json({ data: user });
    } catch (error) {
        response.status(500).json({ error: 'Failed to retrieve user' });
    }
};

// POST /users
export const createUser = async (request: Request, response: Response) => {
    const { email, name } = request.body;
    try {
        const user = await prisma.user.create({
            data: { email, name },
        });
        response.status(201).json({ data: user });
    } catch (error) {
        response.status(500).json({ error: 'Failed to create user' });
    }
};

// PUT /users/:id
export const updateUser = async (request: Request, response: Response) => {
    const id = Number(request.params.id);
    const { email, name } = request.body;
    try {
        const user = await prisma.user.update({
            where: { id },
            data: { email, name },
        });
        response.json({ data: user });
    } catch (error) {
        response.status(500).json({ error: 'Failed to update user' });
    }
};

// DELETE /users/:id
export const deleteUser = async (request: Request, response: Response) => {
    const id = Number(request.params.id);
    try {
        const user = await prisma.user.delete({ where: { id } });
        response.json({ data: user });
    } catch (error) {
        response.status(500).json({ error: 'Failed to delete user' });
    }
};
```

Each handler maps directly to a CRUD operation. The route file wires them together:

```
import { Router } from 'express';
import { getUsers, getUserById, createUser, updateUser, deleteUser } from
'../controllers/users';

const userRouter = Router();

userRouter.get('/', getUsers);
userRouter.get('/:id', getUserById);
```

```
userRouter.post('/', createUser);
userRouter.put('/:id', updateUser);
userRouter.delete('/:id', deleteUser);

export { userRouter };
```

9.3 Error Handling for Prisma Operations

The generic `catch` blocks above hide valuable information. Prisma throws specific, typed errors that you can handle individually. The key class is `PrismaClientKnownRequestError` — errors that Prisma recognizes and tags with a code:

```
import { Prisma } from './generated/prisma';

export const createUser = async (request: Request, response: Response) => {
    const { email, name } = request.body;
    try {
        const user = await prisma.user.create({
            data: { email, name },
        });
        response.status(201).json({ data: user });
    } catch (error) {
        if (error instanceof Prisma.PrismaClientKnownRequestError) {
            if (error.code === 'P2002') {
                response.status(409).json({
                    error: 'A user with this email already exists',
                });
                return;
            }
        }
        response.status(500).json({ error: 'Failed to create user' });
    }
};
```

9.4 Common Prisma Error Codes

Code	Meaning	Typical HTTP Status	When It Happens
`P2002`	Unique constraint violation	`409 Conflict`	Creating a record with a duplicate value in a `@unique` field
`P2025`	Record not found	`404 Not Found`	Updating or deleting a record that does not exist

Code	Meaning	Typical HTTP Status	When It Happens
`P2003`	Foreign key constraint violation	`400 Bad Request`	Creating a record that references a non-existent parent
`P2014`	Required relation violation	`400 Bad Request`	Deleting a record that other records depend on

Handling these errors explicitly lets you return meaningful error messages to your API consumers instead of a generic "500 Internal Server Error."

Gen AI & Learning: Prisma Error Handling

Once you understand the pattern — check `instanceof PrismaClientKnownRequestError`, switch on `error.code`, return the right HTTP status — this is repetitive work that an AI coding assistant handles well. Describe the error codes you want to handle and the HTTP responses you want to return, and let the agent generate the error-handling logic. The design decision (which errors to handle, what messages to return) is still yours.

---

10 Prisma Studio

Prisma Studio is a visual database browser that runs in your web browser. It lets you view, create, edit, and delete records without writing queries — useful for debugging and verifying that your routes are storing data correctly.

10.1 Launching Studio

```
npx prisma studio
```

This opens a browser tab (default port 5555) showing all your models. Click on a model to see its data in a spreadsheet-like view.

10.2 What You Can Do

- **Browse data** — View all records in any table, with pagination

- **Filter and sort** — Search for specific records
- **Edit records** — Click a cell to modify a value, then save
- **Add records** — Create new rows directly in the UI
- **Delete records** — Remove rows you no longer need
- **View relations** — Click on a relation field to navigate to the related record

10.3 When to Use Studio

Studio is a development tool, not a production tool. Common use cases:

- **After running a migration** — Verify that tables were created correctly
- **After seeding data** — Check that seed records exist
- **Debugging a route** — Did the `POST /users` route actually create the record? Open Studio and look
- **Quick data fixes** — Need to change one field on one record? Faster than writing a script

You can also connect to any database directly using a connection URL:

```
npx prisma studio --url "postgresql://user:password@localhost:5432/mydb"
```

Try It Yourself

1. Run `npx prisma studio` in your project
2. Open the browser tab and click on one of your models
3. Create a new record using the Studio UI
4. Now query that record from your Express API — it should appear
5. Delete the record in Studio and query again — it should be gone

---

11 Summary

Concept	Key Point
ORM	Maps database tables to TypeScript objects — write TS code, not SQL strings

Concept	Key Point
Why Prisma	Schema-as-code, auto-generated types, built-in migrations, great DX
`prisma init`	Creates `schema.prisma`, `prisma.config.ts`, and `.env`
Schema models	Each `model` block maps to a database table; fields map to columns
Field attributes	`@id`, `@default`, `@unique`, `@updatedAt`, `@map` — modify field behavior
`@@map`	Maps a model name to a different table name in the database
Relations	`@relation` links models; `Post[]` on the parent, `authorId` + `@relation` on the child
`migrate dev`	Generates and applies migration SQL from schema changes
`migrate reset`	Drops and recreates the database — development only
`prisma generate`	Regenerates the TypeScript client from the schema
Prisma Client	`prisma.<model>.create/findMany/findUnique/update/delete`
Driver adapter	`PrismaPg` adapter connects Prisma Client to PostgreSQL via the `pg` driver
Filtering	`where` with operators: `equals`, `contains`, `gt`, `gte`, `lt`, `lte`, `in`
Sorting	`orderBy: { field: 'asc' | 'desc' }`
Pagination	`skip` and `take` — see the Pagination & Filtering guide for full coverage
`include`	Fetches related data in the same query
`select`	Picks specific fields — mutually exclusive with `include` at the same level
Nested writes	Create parent and children in a single transaction

Concept	Key Point
Error handling	`PrismaClientKnownRequestError` with codes: P2002 (unique), P2025 (not found)
Prisma Studio	Visual database browser at `localhost:5555` — `npx prisma studio`

---

12 References

Official Documentation:

- Prisma — Getting Started — Quickstart guides for PostgreSQL, MySQL, SQLite, and Prisma Postgres
- Prisma Schema Reference — Complete reference for field types, attributes, and model definitions
- Prisma Client — CRUD Operations — `create`, `findMany`, `findUnique`, `update`, `delete`, and more
- Prisma Client — Filtering and Sorting — `where` clause operators and `orderBy`
- Prisma Client — Relation Queries — `include`, `select`, nested reads and writes
- Prisma Migrate — Creating and applying migrations
- Prisma CLI Reference — All CLI commands and options
- Prisma Error Reference — Error codes for `PrismaClientKnownRequestError`
- Prisma Config Reference — `prisma.config.ts` options and configuration
- Prisma Studio — Visual database browser

---

13 Further Reading

External Resources

- SQL Fundamentals Guide — Prerequisite guide covering the SQL that Prisma generates under the hood
- PostgreSQL & Docker Setup Guide — Setting up the database that Prisma connects to
- Pagination & Filtering Guide — Full coverage of offset/cursor pagination, composable filtering with the spread operator pattern, and response metadata
- Prisma — Upgrade to Prisma ORM 7 — Migration guide for projects upgrading from Prisma 6 to 7
- Handling Exceptions and Errors (Prisma Docs) — In-depth guide to error handling with Prisma Client

---

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