

DATABASES

Prisma Fundamentals

A practical introduction to the Prisma ORM for Node.js and TypeScript, written for Prisma 7 — setup with driver adapters, the schema language, migrations, CRUD, filtering and pagination, relations, nested writes and reading related data — closing with a complete Express blog API.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: WHAT AN ORM IS & WHERE PRISMA FITS

Prisma Fundamentals

Chapter 1 · What an ORM Is & Where Prisma Fits

Almost every application stores data in a database, and almost every application is written in a language that isn't SQL. Something has to sit between the two. This course is about one popular choice for Node.js and TypeScript, **Prisma**. This first chapter explains the problem Prisma solves, the other ways of solving it, and how Prisma is put together, before any code is written in Chapter 2.

The Problem: Two Different Worlds

A relational database stores data in **tables** of rows and columns, linked by foreign keys. Your application works with **objects**: a `user` with a list of `posts`, each with its own `author`. Converting between the two by hand is repetitive and error-prone:

```
// Using a plain database driver: SQL as a string, results as untyped rows
const result = await pool.query(
  'SELECT id, email, name FROM users WHERE id = $1', [userId]
);
const user = result.rows[0]; // what shape is this? the compiler doesn't know
console.log(user.emial);      // typo: no error until it prints "undefined"
```

The SQL is just text, so typos in column names only show up when the code runs, and the results carry no type information. This gap between the table model and the object model is sometimes called the **object-relational impedance mismatch**.

Three Ways to Bridge the Gap

APPROACH	EXAMPLES (NODE.JS)	HOW YOU WRITE QUERIES	TRADE-OFF
Database driver	<code>pg</code> , <code>mysql2</code> , <code>better-sqlite3</code>	Raw SQL strings	Full control, but no type safety and a lot of repetitive code

APPROACH	EXAMPLES (NODE.JS)	HOW YOU WRITE QUERIES	TRADE-OFF
Query builder	Knex, Kysely	JavaScript methods that build SQL	Close to SQL, with safer composition; you still map results yourself
ORM	Sequelize, TypeORM, Prisma, Drizzle	Methods on models or a generated client	Much less code and better types, at the cost of some control and an extra layer to learn

ORM stands for *object-relational mapper*: a library that maps database tables to objects in your code, so you can read and write data without writing most of the SQL yourself.

AN ORM DOESN'T REPLACE KNOWING SQL

Every query an ORM runs is still SQL underneath. When something is slow or behaves oddly, you need to understand the SQL it produced — Chapter 9's N+1 problem is the classic example. Prisma also lets you write raw SQL when you need it (Prisma Intermediate/Advanced, Chapter 3).

How Prisma Is Different

Many ORMs have you describe your tables with classes in your application code. Prisma instead starts from a single **schema file**, written in its own small language, and **generates** a type-safe client from it:

```
// schema.prisma: the single source of truth (Chapter 3)
model User {
  id    Int    @id @default(autoincrement())
  email String @unique
  name  String?
}

// Your code: using the generated client (Chapter 5)
const user = await prisma.user.findUnique({ where: { id: userId } });
console.log(user?.emial); // compile error: 'emial' does not exist on type User
```

The typo that slipped past the plain driver is now caught by TypeScript before the code runs, and your editor can autocomplete every field and query option.

ORM	WHERE THE DATA MODEL LIVES	STYLE
Sequelize	JavaScript model definitions	A long-established ORM; types added on top
TypeORM	TypeScript classes with decorators	Close to ORMs in Java and C#
Drizzle	TypeScript table definitions	Deliberately SQL-like queries, very thin layer
Prisma	A separate <code>schema.prisma</code> file	Generated client with an object-style query API

None of these is simply "best." Prisma's strengths are its readable schema, strong typing, and polished migration tooling. Its costs are an extra generation step, its own schema language to learn, and a query API that's further from SQL than Drizzle or a query builder.

The Parts of Prisma

PART	WHAT IT DOES	COVERED IN
Prisma schema	Describes your database connection, data models and relations	Chapters 3, 7, 8
Prisma Client	The generated, type-safe library your code uses to query the database	Chapters 5, 6, 9
Prisma Migrate	Turns changes to the schema into SQL migration files and applies them	Chapter 4
Prisma Studio	A browser-based viewer and editor for your data	Chapter 5
Prisma CLI	The <code>prisma</code> command that runs generation, migrations and Studio	Throughout

The company behind Prisma also sells hosted products, such as a managed Postgres database and a connection pooling and caching service. They're optional; everything in this course works with a database you run yourself.

Which Databases, and Which Version?

This course uses **Prisma ORM 7**, the current stable release line. It supports PostgreSQL, MySQL, MariaDB, SQLite, Microsoft SQL Server, CockroachDB and MongoDB. Examples use SQLite for quick experiments and PostgreSQL for anything closer to production.

WHAT CHANGED IN PRISMA 7

Earlier versions ran every query through a separate engine written in Rust. Prisma 7's client is written in TypeScript and talks to your database through an ordinary JavaScript database driver, plugged in with a **driver adapter**. It also added a `prisma.config.ts` settings file and moved to ES modules. Many older tutorials show the previous setup, so if a guide doesn't match what you see, check which version it was written for. Chapter 2 sets up a project the Prisma 7 way.

PRISMA 8 IS ON THE WAY

At the time of writing, Prisma ORM 8 is available as a release candidate, but only for some databases. Prisma ORM 7 remains the stable, fully supported version and will keep receiving fixes for 18 months after version 8's full release, which is why this course uses it.

Hands-On Exercises

EXERCISE 1

In your own words, explain the "object-relational impedance mismatch," and give two concrete problems with the plain-driver example at the start of this chapter.

 [View solution](#)

EXERCISE 2

For each project, suggest whether a plain driver, a query builder, or an ORM such as Prisma would suit best, and why: (a) a one-off script that copies data between two tables; (b) a medium-sized web app written in TypeScript by a small team; (c) a reporting tool made almost entirely of complex hand-tuned SQL.

 [View solution](#)

EXERCISE 3

Find a Prisma tutorial online and work out which Prisma version it was written for. What clues in its code or setup instructions tell you, based on this chapter?

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- An **ORM** (object-relational mapper) maps database tables to objects in code
- Options range from raw drivers (`pg`) through query builders (Knex, Kysely) to ORMs (Sequelize, TypeORM, Drizzle, Prisma)
- Prisma is **schema-first**: `schema.prisma` describes the data, and a type-safe client is generated from it
- Its parts: the schema, Prisma Client, Prisma Migrate, Prisma Studio and the `prisma` CLI
- This course uses **Prisma ORM 7**: a TypeScript client with driver adapters, `prisma.config.ts` and ES modules
- Prisma 8 is a release candidate for some databases; version 7 stays the stable choice
- You still need SQL to understand what an ORM is doing

CHAPTER 2: PROJECT SETUP

Prisma Fundamentals

Chapter 2 · Project Setup

This chapter builds a working Prisma 7 project from an empty folder: a TypeScript project, the Prisma packages, the schema, the configuration file, a first table, and a small script that writes and reads a row. It uses SQLite, so there's no database server to install. Every later chapter starts from this project.

PIN THE VERSION: PLAIN "NPM INSTALL PRISMA" NOW GETS PRISMA 8

On the npm registry, the `latest` tag for the `prisma` package currently points to a Prisma 8 release candidate (Chapter 1). Installing it without a version gives you Prisma 8, which is set up differently. This course pins every Prisma package to version 7 with `@7`.

Step 1: Create a TypeScript Project

You'll need Node.js 20.19 or later (22 is recommended).

```
mkdir prisma-blog && cd prisma-blog
npm init -y
npm install --save-dev typescript tsx @types/node
```

`tsx` runs TypeScript files directly, without a separate compile step. Prisma 7 is published as ES modules, so tell Node your project uses them too, in `package.json`:

```
{
  "name": "prisma-blog",
  "type": "module",
  ...
}
```

And create `tsconfig.json`:

```
{
  "compilerOptions": {
    "module": "ESNext",
    "moduleResolution": "bundler",
    "target": "ES2023",
    "strict": true,
    "esModuleInterop": true
  }
}
```

Step 2: Install Prisma

```
# The CLI, used only during development
npm install --save-dev prisma@7 @types/better-sqlite3

# What your application uses at run time
npm install @prisma/client@7 @prisma/adapter-better-sqlite3@7 dotenv
```

PACKAGE	ROLE
<code>prisma</code>	The command-line tool: <code>init</code> , <code>migrate</code> , <code>generate</code> , <code>studio</code>
<code>@prisma/client</code>	The runtime library the generated client is built on
<code>@prisma/adapter-better-sqlite3</code>	The driver adapter that connects Prisma to SQLite through the <code>better-sqlite3</code> driver
<code>dotenv</code>	Loads settings such as the database URL from a <code>.env</code> file

Step 3: Initialise Prisma

```
npx prisma init --datasource-provider sqlite --output ../generated/prisma
```

This creates three things:

```

prisma-blog/
├─ prisma/
│   └─ schema.prisma      # your data model (Chapter 3)
├─ prisma7.config.ts      # Prisma's settings
├─ .env                   # the database URL
├─ package.json
└─ tsconfig.json

```

WHY "PRISMA7.CONFIG.TS"?

From Prisma 7.10, the configuration file is named `prisma7.config.ts`, so that it doesn't clash with Prisma 8's `prisma.config.ts`, which uses a different format. Earlier Prisma 7 releases, and many guides, call it `prisma.config.ts`. Use whichever name your `prisma init` creates.

The schema

```

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

datasource db {
  provider = "sqlite"
}

```

- The **generator** block says what to generate. `prisma-client` is Prisma 7's new client generator; `output` is where the generated code goes, relative to the schema file. In Prisma 7 the output path is required.
- The **datasource** block says which kind of database you're using. The connection details don't go here any more.

The configuration file and .env

```

// prisma7.config.ts
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"],
  },
});
```

```
# .env
DATABASE_URL="file:./dev.db"
```

The config file tells the Prisma CLI where the schema and migrations live and how to reach the database. The `import "dotenv/config"` line matters: Prisma 7 doesn't load `.env` by itself, so without it `DATABASE_URL` would be empty.

KEEP .ENV OUT OF VERSION CONTROL

`.env` will hold real database passwords once you move beyond SQLite. Add `.env`, the generated client folder, and your SQLite database file to `.gitignore`.

Step 4: Add a Model and Create the Table

Add a first model to the schema (Chapter 3 explains every part):

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

```
# Create the database table (Chapter 4 explains migrations)
npx prisma migrate dev --name init
```

```
# Generate the type-safe client into generated/prisma
npx prisma generate
```

GENERATE AFTER EVERY SCHEMA CHANGE

The client is generated code. Whenever you change `schema.prisma`, run `prisma generate` again, or your TypeScript types will describe the old schema.

Step 5: Create the Client and Use It

Put the client in one shared file, so the whole application uses a single instance:

```
// lib/prisma.ts
import "dotenv/config";
import { PrismaBetterSqlite3 } from "@prisma/adapters-better-sqlite3";
import { PrismaClient } from "../generated/prisma/client";

const adapter = new PrismaBetterSqlite3({ url: process.env.DATABASE_URL! });
export const prisma = new PrismaClient({ adapter });
```

Notice the import path: the client comes from your own `generated/prisma` folder, not from `@prisma/client` as in older versions. Now a first script:

```
// script.ts
import { prisma } from "../lib/prisma";

const user = await prisma.user.create({
  data: { email: "ada@example.com", name: "Ada" },
});
console.log("Created:", user);

const all = await prisma.user.findMany();
console.log("All users:", all);

await prisma.$disconnect();
```

```
$ npx tsx script.ts
Created: { id: 1, email: 'ada@example.com', name: 'Ada' }
All users: [ { id: 1, email: 'ada@example.com', name: 'Ada' } ]
```

Run it a second time and it fails, because `email` is `@unique` and Ada already exists. That's the database protecting your data, and Chapter 5 shows how to handle it.

Using PostgreSQL Instead

The structure is the same for other databases; only the provider, the URL, and the adapter change. For PostgreSQL, install the `pg` driver and its adapter, and change three things:

```
npm install @prisma/adapter-pg@7 pg

// schema.prisma
datasource db { provider = "postgresql" }

# .env
DATABASE_URL="postgresql://user:password@localhost:5432/blog"

// lib/prisma.ts
import { PrismaPg } from "@prisma/adapter-pg";
const adapter = new PrismaPg({ connectionString: process.env.DATABASE_URL! });
```

Hands-On Exercises

EXERCISE 1

Build the project in this chapter from scratch and run `script.ts` successfully. Then check which Prisma versions were actually installed, and explain why that check matters right now.

 [View solution](#)

EXERCISE 2

Remove the `import "dotenv/config"` line from the config file and run `npx prisma migrate dev`. What happens, and why? Put it back afterwards.

 [View solution](#)

EXERCISE 3

Write a `.gitignore` for this project, and explain why each entry is there. Then modify `script.ts` so that running it twice doesn't fail.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- Node 20.19+; `"type": "module"` in `package.json`; TypeScript run with `tsx`
- Pin Prisma 7: `prisma@7`, `@prisma/client@7`, and the adapter `@7` — plain `prisma` installs a Prisma 8 release candidate
- `npx prisma init --datasource-provider sqlite --output ../generated/prisma`
- Generator `prisma-client` with a required `output`; the datasource has only the provider
- `prisma7.config.ts` (`prisma.config.ts` before 7.10) holds the schema path, migrations path and database URL; it must `import "dotenv/config"`
- `npx prisma migrate dev --name init`, then `npx prisma generate` after every schema change
- Create one client in `lib/prisma.ts` with a driver adapter; import it from `generated/prisma/client`

CHAPTER 3: THE PRISMA SCHEMA

Prisma Fundamentals

Chapter 3 · The Prisma Schema

Chapter 2 added a one-line `User` model without explaining it. This chapter covers the schema language properly: models, field types, optional and list fields, the attributes that control defaults, keys and uniqueness, enums, and how to control the names Prisma uses in the database. By the end you'll have a realistic starting schema for a blog. Relations between models come in Chapters 7 and 8.

Models and Fields

Each **model** describes one table (or, in MongoDB, one collection). Each line inside it is a **field**: a name, a type, and optionally some attributes.

```
model User {
  //name      type      attributes
  id          Int        @id @default(autoincrement())
  email       String     @unique
  name        String?
  createdAt   DateTime   @default(now())
}
```

By convention, model names are singular and in PascalCase (`User` , `BlogPost`), and field names are camelCase (`createdAt`). Both must start with a letter. Prisma Client uses them directly: this model becomes `prisma.user` , with a TypeScript type `User` .

Scalar Types

PRISMA TYPE	TYPESCRIPT TYPE	USE FOR
<code>String</code>	<code>string</code>	Text
<code>Boolean</code>	<code>boolean</code>	True/false flags

PRISMA TYPE	TYPESCRIPT TYPE	USE FOR
Int	number	Whole numbers (32-bit)
BigInt	bigint	Very large whole numbers
Float	number	Approximate decimals, such as measurements
Decimal	Decimal (a library type)	Exact decimals, especially money
DateTime	Date	Dates and times
Json	JSON values	Flexible structured data
Bytes	binary (Uint8Array)	Raw binary data

NEVER USE FLOAT FOR MONEY

Float stores approximate binary values, so sums like 0.1 + 0.2 don't come out exactly. Use Decimal for prices and balances, or store whole pennies in an Int .

Optional and List Fields

SYNTAX	MEANING	TYPESCRIPT
name String	Required: every row must have a value	string
name String?	Optional: may be NULL in the database	string null
tags String[]	A list of values in one column	string[]

LISTS NEED DATABASE SUPPORT

Lists of plain values (scalar lists) only work on databases with a native array type — PostgreSQL and CockroachDB, plus MongoDB. On SQLite and MySQL, model a list as a separate table with a relation instead (Chapter 8).

Field Attributes

ATTRIBUTE	WHAT IT DOES
@id	Marks the primary key: each row's unique identifier
@default(...)	Sets a value when you don't provide one
@unique	No two rows may have the same value
@updatedAt	Prisma sets it to the current time whenever the row is updated
@map("...")	Uses a different column name in the database
@db.X	Chooses a specific database column type, such as @db.VarChar(255)

Defaults and ID strategies

DEFAULT	PRODUCES	GOOD FOR
autoincrement()	1, 2, 3, ... (relational databases)	Simple internal IDs
uuid() / uuid(7)	A random UUID; version 7 is ordered by time	IDs that are hard to guess, or created on several servers
cuid() , ulid() , nanoid()	Other kinds of unique string IDs	Shorter or URL-friendly IDs
now()	The current date and time	createdAt timestamps
dbgenerated("...")	A default calculated by the database itself	Database-specific defaults
A literal: @default(false)	That fixed value	Flags, counters, statuses

SEQUENTIAL IDS CAN LEAK INFORMATION

With `autoincrement()`, a public URL like `/orders/1042` tells anyone roughly how many orders you have, and invites them to try `/orders/1043`. For IDs that appear in URLs, a UUID is often the safer choice. (Real security still needs proper permission checks.)

Enums

```
enum Role {
  READER
  AUTHOR
  ADMIN
}

model User {
  id Int @id @default(autoincrement())
  role Role @default(READER)
}
```

An **enum** restricts a field to a fixed list of values, and Prisma generates a TypeScript type for it, so `role: "EDITOR"` is a compile error. Enums work on all the relational databases in this course, including SQLite since Prisma 6.2.

Model Attributes: Rules Across Several Fields

Attributes that start with `@@` apply to the whole model rather than one field.

```
model Subscription {
  userId Int
  topic String
  created DateTime @default(now())

  @@id([userId, topic]) // composite primary key: one row per user per topic
}

model Post {
  id Int @id @default(autoincrement())
  blogId Int
  slug String

  @@unique([blogId, slug]) // the same slug can appear in different blogs, but not in the same blog
  @@index([blogId]) // speed up lookups by blog (Prisma Intermediate/Advanced)
}
```

Controlling Database Names: @map and @@map

Many databases use lower-case, underscore-separated names (`created_at` , `blog_posts`), while TypeScript code prefers `createdAt` and `Post` . You don't have to choose:

```
model Post {
  id          Int          @id @default(autoincrement())
  createdAt   DateTime     @default(now()) @map("created_at")

  @@map("blog_posts")
}
```

Your code uses `prisma.post` and `createdAt` , while the database table is `blog_posts` with a column `created_at` . This is especially useful when you connect Prisma to an existing database whose names you can't change.

A Starting Schema for the Blog

This is the schema later chapters build on. Relations between users and posts are added in Chapter 7.

```
enum Role {
  READER
  AUTHOR
  ADMIN
}

model User {
  id          Int          @id @default(autoincrement())
  email       String       @unique
  name        String?
  role        Role         @default(READER)
  createdAt   DateTime     @default(now())
  updatedAt   DateTime     @updatedAt
}

model Post {
  id          Int          @id @default(autoincrement())
  title       String
  slug        String       @unique
  content     String?
  published   Boolean      @default(false)
```

```

viewCount    Int          @default(0)
publishedAt  DateTime?
createdAt    DateTime      @default(now())
updatedAt    DateTime      @updatedAt
}

```

```

# Check the schema for mistakes and tidy its layout
npx prisma validate
npx prisma format

```

EDITOR SUPPORT

The official Prisma extension for VS Code adds syntax highlighting, error checking and autocompletion for `.prisma` files. `prisma format` lines the columns up the way the examples in this chapter are laid out.

Hands-On Exercises

EXERCISE 1

Replace your Chapter 2 schema with this chapter's blog schema, run `prisma validate` and `prisma format`, then apply it (`npx prisma migrate dev --name blog-schema`) and regenerate the client. Look at the generated SQL in `prisma/migrations` and find where each attribute ended up.

 [View solution](#)

EXERCISE 2

Design a `Product` model for a shop: a UUID ID, a name, a unique SKU, a price, a stock count starting at zero, an optional description, a status that can only be DRAFT, ACTIVE or DISCONTINUED, and created/updated timestamps. Store it in a table called `products`. Explain your choice of type for the price.

 [View solution](#)

EXERCISE 3

Find and fix the mistakes in this model: `model users { ID int @id; Email String @unique @unique; tags String[]; price Float }`, assuming it's for a SQLite database that stores prices.

 View solution

CHAPTER 3 QUICK REFERENCE

- A **model** is a table; each **field** has a name, type and optional attributes; models are singular PascalCase, fields camelCase
- Types: `String`, `Boolean`, `Int`, `BigInt`, `Float`, `Decimal`, `DateTime`, `Json`, `Bytes`; use `Decimal` for money
- `?` makes a field optional; `[]` makes a list (PostgreSQL, CockroachDB, MongoDB only)
- `@id`, `@default(...)`, `@unique`, `@updatedAt`, `@map`, `@db.X`
- Defaults: `autoincrement()`, `uuid()`, `cuid()`, `now()`, `dbgenerated()`, or a literal
- `enum` limits a field to fixed values (on SQLite since Prisma 6.2)
- `@@id`, `@@unique`, `@@index` and `@@map` apply to the whole model
- `npx prisma validate` checks the schema; `npx prisma format` tidies it

CHAPTER 4: PRISMA MIGRATE

Prisma Fundamentals

Chapter 4 · Prisma Migrate

Your schema describes what the database *should* look like. **Prisma Migrate** gets it there: it compares the schema with the database, writes the SQL needed to close the gap, and keeps a history of every change so that other developers, test servers and production can all be brought to exactly the same state.

What a Migration Is

A **migration** is a file of SQL that changes the database structure — creating a table, adding a column, adding an index. Prisma stores each one in its own folder, in the order they were made:

```
prisma/
├─ schema.prisma
└─ migrations/
    ├─ 20260927093000_init/
    │   └─ migration.sql
    ├─ 20260927101500_blog_schema/
    │   └─ migration.sql
    └─ migration_lock.toml    # records which database type the migrations are for
```

Prisma also keeps a table in your database, `_prisma_migrations`, listing which migrations have been applied there. Comparing that table with the folders tells it what still needs to run.

COMMIT YOUR MIGRATIONS

The `migrations` folder is part of your project's source code. Commit it, along with the schema, so that everyone applies the same changes in the same order. Never edit a migration that has already been applied somewhere else.

migrate dev: The Development Workflow

```
# 1. Edit schema.prisma, e.g. add a field to Post:
#     excerpt String?

# 2. Create and apply a migration
npx prisma migrate dev --name add-post-excerpt

# 3. Regenerate the client (Prisma 7 no longer does this for you)
npx prisma generate
```

`migrate dev` does several things in one command:

1. Replays your existing migrations in a temporary **shadow database**, to check that the history still works and that your real database hasn't been changed behind Prisma's back (**drift**).
2. Compares the result with your schema, and writes a new migration for any differences.
3. Applies any unapplied migrations to your development database.

PRISMA 7: GENERATE AND SEED ARE NOW SEPARATE

In earlier versions `migrate dev` also regenerated the client and ran your seed script. Prisma 7 removed both, so you run `npx prisma generate` yourself, and `npx prisma db seed` when you want seed data (Prisma Intermediate/Advanced, Chapter 4). Some pages in Prisma's own documentation still describe the old behaviour, so trust the Prisma 7 upgrade guide if they disagree.

The shadow database

The shadow database is created and deleted automatically each time. With SQLite this needs no setup. With PostgreSQL or MySQL, the database user needs permission to create databases. If it can't have that (common on hosted databases), create an empty database yourself and point Prisma at it in the config file:

```
import "dotenv/config";
import { defineConfig, env } from "prisma/config";

export default defineConfig({
  schema: "prisma/schema.prisma",
  migrations: { path: "prisma/migrations" },
  datasource: {
    url: env("DATABASE_URL"),
    shadowDatabaseUrl: env("SHADOW_DATABASE_URL"),
  },
});
```

```
},
});
```

When a Change Would Lose Data

Some schema changes can't be applied to existing rows safely. Adding a required column with no default to a table that already has rows, for example, would leave those rows without a value. Dropping a column deletes its data. In these cases `migrate dev` warns you, and may offer to **reset** the development database.

CHANGE	WHY IT'S RISKY	SAFER APPROACH
Add a required column	Existing rows have no value	Give it a <code>@default</code> , or add it as optional first
Remove a column or table	Its data is deleted	Make sure nothing needs the data first; back it up
Rename a field	Prisma sees "drop the old, add a new" and would lose the data	Edit the migration SQL to rename instead (below)
Add <code>@unique</code>	Fails if duplicates already exist	Clean up duplicates first

Editing a Migration Before It Runs: --create-only

Prisma can't tell a rename from a delete-and-add. When you rename `content` to `body`, the generated SQL drops `content` and adds an empty `body`. To keep the data, create the migration without applying it, fix the SQL, then apply it:

```
# 1. Create the migration file, but don't run it
npx prisma migrate dev --create-only --name rename-content-to-body

# 2. Edit prisma/migrations/<timestamp>_rename-content-to-body/migration.sql,
#    replacing the generated DROP/ADD lines with:
ALTER TABLE "Post" RENAME COLUMN "content" TO "body";

# 3. Apply it
npx prisma migrate dev
```

OR AVOID THE RENAME IN THE DATABASE

If only your code needs the new name, use `@map` (Chapter 3): rename the field in the schema to `body` and add `@map("content")`. The column keeps its old name and no migration is needed.

Other Migrate Commands

COMMAND	WHAT IT DOES	WHERE TO USE IT
<code>migrate dev</code>	Creates and applies migrations; checks for drift	Development only
<code>migrate reset</code>	Deletes the database, recreates it, and reapplies every migration	Development only — it destroys data
<code>migrate deploy</code>	Applies pending migrations and nothing else: no new migrations, no drift checks, no resets, no shadow database	Production, staging and CI
<code>migrate status</code>	Shows which migrations have and haven't been applied	Anywhere
<code>db push</code>	Makes the database match the schema directly, with no migration files	Quick prototyping only

NEVER RUN MIGRATE DEV OR RESET ON PRODUCTION

Both are designed for disposable development databases and can delete data. Production only ever gets `migrate deploy`, which applies the reviewed, committed migrations and refuses to do anything else. Prisma Intermediate/Advanced, Chapter 8, covers production migrations in detail.

DB PUSH VS. MIGRATE

`db push` is handy while you're sketching a schema and don't care about history. But it leaves no record of what changed, so it can't be replayed on another machine. Switch to migrations as soon as anyone else, or any real data, depends on the database.

Hands-On Exercises

EXERCISE 1

Add an optional `excerpt` field to `Post` and create a migration for it. Then run `npx prisma migrate status` and explain its output. What would happen if you forgot to run `prisma generate` afterwards?

 [View solution](#)

EXERCISE 2

Create a post with some content. Rename the `content` field to `body` in a way that keeps the post's text, and prove the data survived.

 [View solution](#)

EXERCISE 3

A teammate suggests running `npx prisma migrate dev` on the production server "to be safe." Explain what could go wrong, and what should happen instead.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- A migration is a SQL file in `prisma/migrations/`; the `_prisma_migrations` table records which have run; commit the folder
- `migrate dev --name x` checks for drift in a shadow database, creates a migration from schema changes, and applies it
- In Prisma 7, run `prisma generate` (and `prisma db seed`) yourself afterwards
- Set `shadowDatabaseUrl` in the config file when the database user can't create databases
- Watch for data-losing changes: required columns, drops, renames, new unique constraints
- `--create-only` lets you edit the SQL (for example to rename a column) before applying it
- `migrate reset` and `migrate dev` are for development only; production uses `migrate deploy`
- `migrate status` shows what's applied; `db push` is for throwaway prototyping

CHAPTER 5: CRUD WITH PRISMA CLIENT

Prisma Fundamentals

Chapter 5 · CRUD With Prisma Client

With a schema and a database in place, it's time to use them. This chapter covers the four basic operations — **C**reate, **R**ead, **U**psert and **D**elese — using the blog schema from Chapter 3, plus how to handle the errors Prisma throws and how to look at your data with Prisma Studio.

THE PATTERN BEHIND EVERY QUERY

Every model gets the same set of methods: `prisma.user.create(...)`, `prisma.post.findMany(...)`, and so on. Each takes one object of options, such as `where`, `data` and `select`, and every one returns a Promise, so use `await`. Your editor's autocompletion will show exactly which options each method accepts.

Create

```
import { prisma } from "../lib/prisma";

// One row: returns the created record, including defaults like id and createdAt
const post = await prisma.post.create({
  data: { title: "Hello Prisma", slug: "hello-prisma", content: "My first post" },
});

// Many rows: returns only a count
const result = await prisma.post.createMany({
  data: [
    { title: "Second post", slug: "second-post" },
    { title: "Third post", slug: "third-post" },
  ],
});
console.log(result.count); // 2

// Many rows, returning the records (PostgreSQL, CockroachDB, SQLite)
const posts = await prisma.post.createManyAndReturn({
  data: [{ title: "Fourth", slug: "fourth" }],
});
```

TypeScript checks the `data` object against the schema: leave out a required field such as `slug`, or misspell one, and it won't compile. Fields with defaults (`id`, `published`, `createdAt`) can be left out.

SKIPDUPLICATES ISN'T EVERYWHERE

`createMany` accepts `skipDuplicates: true` to quietly skip rows that would break a unique rule, but SQLite, SQL Server and MongoDB don't support it. On those, check first or insert one at a time.

Read

METHOD	RETURNS	USE WHEN
<code>findUnique</code>	One record or <code>null</code>	Looking up by the ID or another <code>@unique</code> field
<code>findUniqueOrThrow</code>	One record, or throws an error	The record must exist
<code>findFirst</code>	The first match or <code>null</code>	Searching by any field, where several might match
<code>findFirstOrThrow</code>	The first match, or throws	As above, when a match must exist
<code>findMany</code>	An array (possibly empty)	Lists of records
<code>count</code>	A number	How many records match

```
const bySlug = await prisma.post.findUnique({ where: { slug: "hello-prisma" } });

const firstDraft = await prisma.post.findFirst({ where: { published: false } });

const all = await prisma.post.findMany();

const drafts = await prisma.post.count({ where: { published: false } });
```

FINDUNIQUE ONLY ACCEPTS UNIQUE FIELDS

`prisma.post.findUnique({ where: { title: "Hello" } })` won't compile, because `title` isn't unique — the database could return several rows. Use `findFirst` or `findMany` for non-unique fields. Chapter 6 covers `where` in depth.

Choosing fields: select

```
// Only fetch what you need; the result type shrinks to match
const titles = await prisma.post.findMany({
  select: { id: true, title: true },
});
// titles: { id: number; title: string }[]
```

Update

```
// One record, found by a unique field
const published = await prisma.post.update({
  where: { slug: "hello-prisma" },
  data: { published: true, publishedAt: new Date() },
});

// Many records: returns a count
await prisma.post.updateMany({
  where: { published: false },
  data: { content: "Coming soon" },
});

// Atomic number operations: safe even if two requests arrive at once
await prisma.post.update({
  where: { slug: "hello-prisma" },
  data: { viewCount: { increment: 1 } },
});
```

DON'T READ, ADD ONE, THEN WRITE

Reading `viewCount`, adding 1 in your code, and saving it back can lose counts: if two visitors arrive at the same moment, both read 5 and both save 6. `{ increment: 1 }` asks the database to do the addition itself, so both visits are counted. There are also `decrement`, `multiply`, `divide` and `set`.

Upsert: update or create

```
const user = await prisma.user.upsert({
  where: { email: "ada@example.com" },
```

```
update: { name: "Ada Lovelace" }, // if she exists
create: { email: "ada@example.com", name: "Ada Lovelace" }, // if not
});
```

Delete

```
// One record, by a unique field; returns the deleted record
await prisma.post.delete({ where: { slug: "fourth" } });

// Many records; returns a count
await prisma.post.deleteMany({ where: { published: false } });
```

AN EMPTY WHERE DELETES EVERYTHING

`prisma.post.deleteMany({})` — or `updateMany` with no `where` — affects every row in the table. It's occasionally what you want (clearing test data), and a disaster when it isn't.

Handling Errors

When the database rejects an operation, Prisma throws a `PrismaClientKnownRequestError` with a **code** that tells you why. Two you'll meet constantly:

CODE	MEANING	TYPICAL CAUSE
P2002	Unique constraint failed	Creating a user with an email that already exists
P2025	Record not found	<code>update</code> or <code>delete</code> on a record that doesn't exist; the <code>...OrThrow</code> methods

```
import { Prisma } from "../generated/prisma/client";

try {
  await prisma.user.create({ data: { email: "ada@example.com" } });
} catch (e) {
  if (e instanceof Prisma.PrismaClientKnownRequestError && e.code === "P2002") {
    console.log("That email is already registered.");
  }
}
```

```

    } else {
      throw e; // anything else is unexpected: let it surface
    }
  }
}

```

Looking at Your Data: Prisma Studio

```
npx prisma studio
```

This opens a browser-based table viewer on your own machine, where you can browse, filter, add, edit and delete rows without writing any code. It's ideal for checking what your scripts actually did.

Hands-On Exercises

EXERCISE 1

Write a script that creates three posts with `createMany`, publishes one of them, counts the drafts, and then prints just the titles of all posts. Check the results in Prisma Studio.

 [View solution](#)

EXERCISE 2

Write a function `registerUser(email, name)` that returns a friendly message instead of crashing when the email is already taken, and a function `recordView(slug)` that increments a post's view count and returns `null` if the post doesn't exist.

 [View solution](#)

EXERCISE 3

For each task, pick the best method and explain why: (a) show one post from its URL slug; (b) find the most recent unpublished post; (c) make sure an "about" page exists, creating it only if missing; (d) clear all posts before a test run.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- Every model has the same methods; each takes an options object and returns a Promise
- **Create:** `create`, `createMany` (returns a count), `createManyAndReturn` (PostgreSQL, CockroachDB, SQLite)
- **Read:** `findUnique` (unique fields only), `findFirst`, `findMany`, `count`, and `...OrThrow` variants
- `select` fetches only the fields you name, and narrows the result type
- **Update:** `update`, `updateMany`, `upsert`; use `{ increment: 1 }` and friends for counters
- **Delete:** `delete`, `deleteMany` — an empty `where` affects every row
- Errors: `Prisma.PrismaClientKnownRequestError`; `P2002` duplicate unique value, `P2025` record not found
- `npx prisma studio` opens a browser view of your data

CHAPTER 6: FILTERING, SORTING & PAGINATION

Prisma Fundamentals

Chapter 6 · Filtering, Sorting & Pagination

Chapter 5's queries used simple conditions such as `{ published: false }`. Real applications need more: posts with "prisma" in the title, published this month, newest first, twenty to a page. This chapter covers the filtering, sorting and paging options, and one trap — case sensitivity — that behaves differently on every database.

Filter Operators

Writing `{ published: true }` is shorthand for `{ published: { equals: true } }`. Instead of a plain value, a field can take an object of operators:

OPERATOR	EXAMPLE	MATCHES
<code>equals</code> , <code>not</code>	<code>{ title: { not: "Draft" } }</code>	Equal / not equal
<code>in</code> , <code>notIn</code>	<code>{ id: { in: [1, 2, 3] } }</code>	Any value in the list / none of them
<code>lt</code> , <code>lte</code> , <code>gt</code> , <code>gte</code>	<code>{ viewCount: { gte: 100 } }</code>	Less than, less or equal, greater than, greater or equal
<code>contains</code>	<code>{ title: { contains: "prisma" } }</code>	Text containing the value
<code>startsWith</code> , <code>endsWith</code>	<code>{ slug: { startsWith: "how-to" } }</code>	Text beginning / ending with the value

```
// Popular posts published since the start of September
const popular = await prisma.post.findMany({
  where: {
    published: true,
    viewCount: { gte: 100 },
    publishedAt: { gte: new Date("2026-09-01") },
  },
});
```

Several conditions at the top level of `where` are combined with AND: a post must match all of them.

Matching NULL

```
// Posts with no content yet
await prisma.post.findMany({ where: { content: null } });

// Posts that do have content
await prisma.post.findMany({ where: { content: { not: null } } });
```

UNDEFINED MEANS "IGNORE THIS CONDITION"

`null` and `undefined` are very different in a `where`. `{ content: null }` finds rows where content is empty; `{ content: undefined }` is ignored completely. So if a search function builds `where: { title: searchTerm }` and `searchTerm` is undefined, the filter vanishes and you get *every* post. Check inputs before using them.

Combining Conditions: AND, OR and NOT

```
// Published posts mentioning "prisma" in the title OR the content
const results = await prisma.post.findMany({
  where: {
    published: true,
    OR: [
      { title: { contains: "prisma" } },
      { content: { contains: "prisma" } },
    ],
    NOT: { slug: { startsWith: "test-" } },
  },
});
```

`OR` takes a list, and a row matches if *any* item matches. `AND` takes a list too, and is useful when you need the same field twice. `NOT` excludes rows that match. They can be nested as deeply as you need.

Case Sensitivity: Different on Every Database

Does `contains: "prisma"` match a title containing "Prisma"? It depends on the database.

DATABASE	DEFAULT TEXT MATCHING	CASE-INSENSITIVE OPTION
PostgreSQL	Case-sensitive	Add <code>mode: "insensitive"</code>
MongoDB	Case-sensitive	Add <code>mode: "insensitive"</code>
MySQL	Usually case-insensitive (depends on the column's collation)	Already insensitive by default
SQL Server	Usually case-insensitive	Already insensitive by default
SQLite	Not case-insensitive for columns Prisma creates	Edit the migration to add <code>COLLATE NOCASE</code> to the column

```
// PostgreSQL: match "Prisma", "PRISMA" and "prisma"
await prisma.post.findMany({
  where: { title: { contains: "prisma", mode: "insensitive" } },
});
```

A SEARCH THAT WORKS IN DEVELOPMENT CAN FAIL IN PRODUCTION

If you develop on SQLite and deploy on PostgreSQL (or the reverse), the same search can return different results. Develop against the same kind of database you run in production, and test searches with mixed-case text.

Sorting

```
// Newest first
await prisma.post.findMany({ orderBy: { createdAt: "desc" } });

// Several fields: most viewed first, then alphabetical among equal counts
await prisma.post.findMany({
  orderBy: [{ viewCount: "desc" }, { title: "asc" }],
});
```

```
// Choose where empty values go
await prisma.post.findMany({
  orderBy: { publishedAt: { sort: "desc", nulls: "last" } },
});
```

ALWAYS SORT WHEN PAGING

Without `orderBy`, the database may return rows in any order it likes, and that order can change between queries. Paging without a fixed sort can show the same post twice or skip one entirely.

Pagination

Offset pagination: take and skip

```
const pageSize = 20;
const page = 3; // pages numbered from 1

const posts = await prisma.post.findMany({
  where: { published: true },
  orderBy: { createdAt: "desc" },
  skip: (page - 1) * pageSize,
  take: pageSize,
});
const total = await prisma.post.count({ where: { published: true } });
```

Simple, and it lets users jump to any page number. But the database still has to count past every skipped row, so very deep pages get slower, and if posts are added while someone is paging, items shift between pages.

Cursor pagination

```
// First page
const firstPage = await prisma.post.findMany({
  take: 20,
  orderBy: { id: "asc" },
});
const lastId = firstPage[firstPage.length - 1]?.id;
```

```
// Next page: start at the last post seen, and skip that post itself
const nextPage = await prisma.post.findMany({
  take: 20,
  skip: 1,
  cursor: { id: lastId },
  orderBy: { id: "asc" },
});
```

	OFFSET (SKIP / TAKE)	CURSOR
Jump to page 7	Yes	No — only next/previous
Speed on deep pages	Gets slower	Stays fast
New rows while paging	Items can shift or repeat	Stable
Good for	Admin tables, numbered pages	Infinite scroll, feeds, large tables

Removing Duplicates: distinct

```
// One row per distinct role
const roles = await prisma.user.findMany({
  distinct: ["role"],
  select: { role: true },
});
```

Hands-On Exercises

EXERCISE 1

Write a query for published posts that either have at least 50 views or were published in the last 7 days, excluding any whose slug starts with `draft-`, sorted by views (most first) then title.

 View solution

EXERCISE 2

Write a `searchPosts(term?: string, page = 1)` function that returns 10 published posts per page, newest first, filtered by title if a term is given, along with the total number of pages. Make sure a missing term can't accidentally return drafts.

 [View solution](#)

EXERCISE 3

On your SQLite project, create posts titled "Prisma Tips" and "prisma tricks", then search with `contains: "prisma"`. Explain the result, and what you'd change to make the search case-insensitive on SQLite and on PostgreSQL.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- Operators: `equals`, `not`, `in`, `notIn`, `lt`/`lte`/`gt`/`gte`, `contains`, `startsWith`, `endsWith`
- Top-level conditions are ANDed; combine with `AND`, `OR` (lists) and `NOT`
- `null` matches empty values; `undefined` removes the condition entirely
- Case sensitivity varies: `mode: "insensitive"` on PostgreSQL and MongoDB; MySQL and SQL Server are usually insensitive; SQLite needs `COLLATE NOCASE`
- `orderBy` takes one object or a list; `{ sort, nulls: "first" | "last" }` controls empty values
- Offset paging: `skip` + `take` (plus `count` for totals); cursor paging: `cursor`, `skip: 1`, `take`
- Always `orderBy` when paging; `distinct` removes duplicate rows

CHAPTER 7: RELATIONS I: ONE-TO-ONE & ONE-TO-MANY

Prisma Fundamentals

Chapter 7 · Relations I: One-to-One & One-to-Many

So far the blog's `User` and `Post` models have no connection: a post doesn't know who wrote it. Real data is full of these links — posts have authors, users have profiles, comments belong to posts. This chapter adds them, explains the two kinds of field every relation needs, and covers what happens to related rows when something is deleted.

Three Kinds of Relation

KIND	EXAMPLE	COVERED IN
One-to-many	One user writes many posts; each post has one author	This chapter
One-to-one	One user has at most one profile; each profile belongs to one user	This chapter
Many-to-many	A post has many tags; a tag is on many posts	Chapter 8

One-to-Many: Users and Posts

```
model User {
  id    Int    @id @default(autoincrement())
  email String @unique
  // ...other fields from Chapter 3
  posts Post[]                          // relation field: a user's posts
}

model Post {
  id      Int    @id @default(autoincrement())
  title   String
  // ...other fields from Chapter 3
  authorId Int    // relation scalar field: the foreign key
  author   User   @relation(fields: [authorId], references: [id])
}
```

```
@@index([authorId])
}
```

Every relation involves two different kinds of field:

FIELD	KIND	IN THE DATABASE?	PURPOSE
<code>authorId</code>	Relation scalar field (foreign key)	Yes: a real column	Stores the ID of the related user
<code>author</code>	Relation field	No	Lets Prisma Client load or connect the related user
<code>posts</code>	Relation field (the other side)	No	Lets you reach a user's posts from the user

The `@relation(fields: [authorId], references: [id])` attribute goes on the side that holds the foreign key. It says: "`authorId` in this table refers to `id` in the `User` table." Both sides of a relation must be declared, and `prisma format` will add a missing side for you.

INDEX YOUR FOREIGN KEYS

You'll often look up "all posts by this author," which searches the `authorId` column. MySQL adds an index for a foreign key automatically, but PostgreSQL doesn't, so add `@@index([authorId])` to keep those lookups fast.

Required or optional?

As written, every post must have an author: `authorId` is `Int` and `author` is `User`. To allow posts with no author (for example, imported posts), make both optional:

```
authorId Int?
author   User? @relation(fields: [authorId], references: [id])
```

ADDING A REQUIRED RELATION TO A TABLE WITH DATA

If your `Post` table already has rows, adding a required `authorId` leaves them without an author, so the migration can't be applied as it is (Chapter 4). In development, resetting is fine. With real data,

add the column as optional, fill in an author for every existing post, then make it required in a second migration.

One-to-One: Users and Profiles

```
model User {
  id      Int      @id @default(autoincrement())
  profile Profile? // at most one profile
  posts   Post[]
}

model Profile {
  id      Int      @id @default(autoincrement())
  bio     String?
  avatar  String?
  userId  Int      @unique // @unique makes it one-to-one
  user    User     @relation(fields: [userId], references: [id])
}
```

The only structural difference from one-to-many is `@unique` on the foreign key. Without it, several profiles could point at the same user; with it, each user can have at most one. The user's side is `Profile?` because a user may not have created a profile yet.

WHICH SIDE HOLDS THE FOREIGN KEY?

Put it on the side that can't exist without the other. A profile makes no sense without a user, but a user can exist without a profile, so `Profile` holds `userId`.

Using Relations in Queries (a Preview)

```
// Create a post for an existing user, by setting the foreign key
await prisma.post.create({
  data: { title: "Relations", slug: "relations", authorId: 1 },
});

// Load a post together with its author
const post = await prisma.post.findUnique({
```

```
    where: { slug: "relations" },
    include: { author: true },
  });
  console.log(post?.author.email);
```

Chapter 8 shows more ways to create related records together, and Chapter 9 covers reading related data properly with `include` and `select`.

Referential Actions: What Happens on Delete?

If you delete a user who has written posts, what should happen to the posts? The database enforces an answer, set with `onDelete` (and `onUpdate`, for when a referenced ID changes):

ACTION	ON DELETING THE USER...	
<code>Restrict</code>	The delete is refused while the user still has posts	
<code>NoAction</code>	Similar to Restrict; the exact timing of the check varies by database	
<code>Cascade</code>	The user's posts are deleted too	
<code>SetNull</code>	The posts stay, with <code>authorId</code> set to NULL (the relation must be optional)	
<code>SetDefault</code>	The posts' <code>authorId</code> is set to its default value	
IF YOU DON'T CHOOSE	<code>ONDELETE</code>	<code>ONUPDATE</code>
Required relation	<code>Restrict</code>	<code>Cascade</code>
Optional relation	<code>SetNull</code>	<code>Cascade</code>

```
// Deleting a user also deletes their profile
user User @relation(fields: [userId], references: [id], onDelete: Cascade)
```

CHOOSE CASCADE DELIBERATELY

`Cascade` is convenient for data that's meaningless on its own, like a profile. For valuable data, it's dangerous: deleting one user could silently delete years of their posts, and their posts' comments,

and so on down the chain. The default `Restrict` makes you decide explicitly what happens to the posts first.

DATABASE DIFFERENCES

SQL Server doesn't offer `Restrict` (use `NoAction`) and rejects some cascade chains. On MySQL, `SetDefault` behaves like `NoAction`. MongoDB has no foreign keys, so Prisma enforces relations itself.

Hands-On Exercises

EXERCISE 1

Add the `User` – `Post` and `User` – `Profile` relations from this chapter to your schema, with profiles deleted along with their user. Migrate, generate, then create a user with a profile and two posts using foreign keys.

 [View solution](#)

EXERCISE 2

With the schema from Exercise 1, try to delete the user. What happens, and why? Then describe two different ways to make deleting a user possible, and when each is appropriate.

 [View solution](#)

EXERCISE 3

Add a `Comment` model: each comment belongs to one post and, optionally, to one user (guests can comment). Choose sensible `onDelete` behaviour for both relations and explain your choices.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- A relation has a **relation scalar field** (the foreign key, a real column) and **relation fields** on both sides (not stored)
- `@relation(fields: [authorId], references: [id])` goes on the side holding the foreign key
- One-to-many: `Post[]` on one side, a single `User` on the other
- One-to-one: the same, but with `@unique` on the foreign key and an optional side (`Profile?`)
- Make a relation optional with `Int?` and `User?`; add `@@index` on foreign keys
- `onDelete` / `onUpdate`: `Restrict`, `NoAction`, `Cascade`, `SetNull`, `SetDefault`
- Defaults: required relations `Restrict` on delete; optional `SetNull`; both `Cascade` on update

CHAPTER 8: RELATIONS II: MANY-TO-MANY & NESTED WRITES

Prisma Fundamentals

Chapter 8 · Relations II: Many-to-Many & Nested Writes

A post can have several tags, and each tag can be on many posts. Neither side can hold a single foreign key, so this needs a third table in between. This chapter shows Prisma's two ways of modelling that, and then covers **nested writes**: creating or linking related records in the same call, which works for every kind of relation from Chapter 7 too.

How Many-to-Many Works in a Database

Post

id	title
1	Intro
2	Queries

join table

postId	tagId
1	10
1	11
2	10

Tag

id	name
10	prisma
11	sql

postId → Post.id

tagId → Tag.id

Each row of the join table links one post to one tag. Post 1 is tagged "prisma" and "sql"; post 2 is tagged "prisma".

Implicit Many-to-Many

The simplest way: put a list field on both sides and let Prisma manage the join table.

```
model Post {
  id    Int    @id @default(autoincrement())
  // ...
  tags  Tag[]
}

model Tag {
```

```

id    Int    @id @default(autoincrement())
name  String @unique
posts Post[]
}

```

Prisma creates a hidden join table, named from the two model names in alphabetical order — here `_PostToTag`, with columns `A` and `B`. You never refer to it in your code.

IMPLICIT MANY-TO-MANY HAS LIMITS

- Both models must have a single-field `@id` (no composite IDs).
- You can't store anything else about the link, such as when a tag was added or by whom.
- You can't set `onDelete` or `onUpdate` on it.
- It doesn't work on MongoDB.

Explicit Many-to-Many

When the link itself carries information, make the join table a real model:

```

model Post {
  id    Int    @id @default(autoincrement())
  tags  PostTag[]
}

model Tag {
  id    Int    @id @default(autoincrement())
  name  String @unique
  posts PostTag[]
}

model PostTag {
  postId    Int
  tagId     Int
  assignedAt DateTime @default(now())
  post      Post      @relation(fields: [postId], references: [id], onDelete: Cascade)
  tag       Tag        @relation(fields: [tagId], references: [id], onDelete: Cascade)

  @@id([postId, tagId]) // each post-tag pair only once
}

```

```
@@index([tagId])
}
```

This is really two one-to-many relations (Chapter 7): a post has many `PostTag` rows, and so does a tag. The composite key stops the same tag being added to a post twice.

	IMPLICIT	EXPLICIT
Schema	Two list fields	A third model with two relations
Extra data on the link	No	Yes (<code>assignedAt</code> , <code>order</code> , ...)
Queries	Simpler: <code>post.tags</code> are tags	One step more: <code>post.tags</code> are links, each with a <code>tag</code>
Delete behaviour	Managed by Prisma	You choose it
Use when	A plain link is enough	The link needs its own data or rules

START IMPLICIT, SWITCH IF NEEDED

explicit model, but it means a migration that copies the existing links into the new table, so it's worth thinking ahead for links you suspect will grow.

Nested Writes

A **nested write** creates, links or unlinks related records inside the same `create` or `update` call. Prisma runs the whole thing as one transaction: either everything succeeds, or nothing is saved.

OPERATION	WHAT IT DOES
<code>create</code>	Create the related record(s) and link them
<code>connect</code>	Link existing records, found by a unique field
<code>connectOrCreate</code>	Link a record if it exists, create it if not
<code>disconnect</code>	Remove the link, keeping both records

OPERATION	WHAT IT DOES
<code>set</code>	Replace all links with exactly this list
<code>delete</code> , <code>update</code> , <code>upsert</code>	Change the related records themselves

Creating a user with posts and a profile in one call

```
const user = await prisma.user.create({
  data: {
    email: "alan@example.com",
    name: "Alan",
    profile: { create: { bio: "Codebreaker" } },
    posts: {
      create: [
        { title: "Computing Machinery", slug: "computing-machinery" },
        { title: "Morphogenesis", slug: "morphogenesis" },
      ],
    },
  },
});
```

You never set `userId` or `authorId` yourself: Prisma fills in the foreign keys because it knows how the records are related.

Tagging a post (implicit many-to-many)

```
await prisma.post.create({
  data: {
    title: "Nested Writes",
    slug: "nested-writes",
    author: { connect: { email: "alan@example.com" } }, // existing user
    tags: {
      connectOrCreate: [
        { where: { name: "prisma" }, create: { name: "prisma" } },
        { where: { name: "orm" }, create: { name: "orm" } },
      ],
    },
  },
});

// Later: replace the post's tags with just "prisma"
```

```
await prisma.post.update({
  where: { slug: "nested-writes" },
  data: { tags: { set: [{ name: "prisma" }] } },
});
```

Tagging a post (explicit many-to-many)

```
await prisma.post.update({
  where: { slug: "nested-writes" },
  data: {
    tags: {
      create: {
        tag: { connectOrCreate: { where: { name: "sql" }, create: { name: "sql" } } },
      },
    },
  },
});
```

With an explicit join model you create a `PostTag` row, and inside it connect or create the `Tag`. It's one level deeper than the implicit version.

USE EITHER THE FOREIGN KEY OR THE RELATION, NOT BOTH

In one `data` object, set a relation either by its foreign key (`authorId: 1`) or through the relation field (`author: { connect: { id: 1 } }`). Mixing the two styles in the same call gives a type error. Nested writes need the relation-field style.

Hands-On Exercises

EXERCISE 1

Add an implicit many-to-many relation between `Post` and `Tag`. In a single `create` call, create a post for an existing user, tagged with "prisma" (which already exists) and "typescript" (which doesn't). Then look at the join table in Prisma Studio.

 [View solution](#)

EXERCISE 2

Write a function `setPostTags(slug, tagNames)` that makes a post's tags exactly match the given list, creating any tags that don't exist yet.

 View solution

EXERCISE 3

A course platform links students to courses. For each link it must record when the student enrolled and whether they've completed the course. Design the models, and explain why an implicit relation won't do.

 View solution

CHAPTER 8 QUICK REFERENCE

- Many-to-many needs a join table linking the two models
- **Implicit:** list fields on both sides; Prisma manages a hidden `_AToB` table; no extra data, single-field IDs only, not on MongoDB
- **Explicit:** a join model with two relations and `@@id([aId, bId])`; can store extra data and set delete rules
- Nested writes: `create`, `connect`, `connectOrCreate`, `disconnect`, `set`, `delete`, `update`, `upsert`
- A nested write runs as one transaction; Prisma fills in the foreign keys
- Set a relation by foreign key or by relation field — not both in one call

CHAPTER 9: READING RELATED DATA

Prisma Fundamentals

Chapter 9 · Reading Related Data

Chapters 7 and 8 connected users, profiles, posts and tags. This chapter is about reading them back: loading a record together with its related records, choosing exactly which fields come back, filtering records by their relations, and avoiding the most common performance mistake in any ORM, the **N+1 query problem**.

include: Add Related Records

By default a query returns only the model's own fields. `include` adds relations on top:

```
const post = await prisma.post.findUnique({
  where: { slug: "nested-writes" },
  include: {
    author: true,    // the whole User record
    tags:   true,    // all its tags
  },
});
// post.title, post.author.email, post.tags[0].name ... all typed
```

Includes can be nested, to follow relations further:

```
const user = await prisma.user.findUnique({
  where: { email: "alan@example.com" },
  include: {
    profile: true,
    posts: { include: { tags: true } }, // each post with its tags
  },
});
```

Filtering, Sorting and Limiting Included Lists

```
// A user with only their 5 latest published posts
const user = await prisma.user.findUnique({
  where: { email: "alan@example.com" },
  include: {
    posts: {
      where: { published: true },
      orderBy: { publishedAt: "desc" },
      take: 5,
    },
  },
});
```

UNBOUNDED INCLUDES GROW WITH YOUR DATA

`include: { posts: true }` is fine with ten posts and a problem with ten thousand: every one is loaded into memory. For lists that can grow, add `take`, or load them separately with pagination (Chapter 6).

select: Choose Exactly What Comes Back

`select` returns *only* the fields you list, including relations. It's the tool for keeping responses small — and for keeping private fields private.

```
// For a post list: title, slug, and just the author's name
const list = await prisma.post.findMany({
  where: { published: true },
  select: {
    title: true,
    slug: true,
    author: { select: { name: true } },
    tags: { select: { name: true } },
  },
});
// { title: string; slug: string; author: { name: string | null }; tags: { name: string
```

INCLUDE

SELECT

Model's own fields

All of them

Only the ones you list

	INCLUDE	SELECT
Relations	The ones you list, added on	The ones you list
Use for	"This record, plus its related data"	"Exactly these fields," such as for an API response

ONE OR THE OTHER AT EACH LEVEL

You can't use `include` and `select` side by side at the same level of a query. You can mix them at different levels, though: `include: { author: { select: { name: true } } }` returns all the post's fields plus only the author's name.

INCLUDE CAN LEAK DATA

`include: { author: true }` returns the whole user record. If your `User` model later gains a `passwordHash` or other private field, every response built this way now contains it. For data sent to browsers or other systems, `select` the fields you mean to share.

Filtering by Relations

You can also filter records by what their related records look like:

FILTER	FOR	MATCHES RECORDS WHERE...
<code>some</code>	Lists	At least one related record matches
<code>every</code>	Lists	All related records match
<code>none</code>	Lists	No related record matches
<code>is</code> / <code>isNot</code>	Single relations	The related record matches / doesn't match

```
// Users who have published at least one post
await prisma.user.findMany({ where: { posts: { some: { published: true } } } });

// Posts tagged "prisma"
await prisma.post.findMany({ where: { tags: { some: { name: "prisma" } } } });

// Users who have written nothing at all
await prisma.user.findMany({ where: { posts: { none: {} } } });
```

```
// Posts written by admins
await prisma.post.findMany({ where: { author: { is: { role: "ADMIN" } } } });
```

EVERY IS TRUE FOR AN EMPTY LIST

"Users where every post is published" also returns users with *no posts at all*, because there's no post that breaks the rule. If you mean "users who have posts, and all of them are published," combine

`every` with `some: {}`.

The N+1 Query Problem

This code works, and it's a classic mistake:

```
// Listing 50 posts with their authors' names -- the slow way
const posts = await prisma.post.findMany({ take: 50 }); // 1 query

for (const post of posts) {
  const author = await prisma.user.findUnique({ // +1 query per post
    where: { id: post.authorId },
  });
  console.log(post.title, "by", author?.name);
}
```

One query for the list, then one more for every row: **N+1** queries, 51 in this case. Each is quick, but they add up, especially when the database is on another machine and every query is a network round trip. The fix is to ask for the related data as part of the list query:

```
const posts = await prisma.post.findMany({
  take: 50,
  include: { author: { select: { name: true } } },
});
for (const post of posts) console.log(post.title, "by", post.author.name);
```

Prisma now loads the authors for all 50 posts together. By default it does this with one extra query for the relation, however many posts there are, so the total stays fixed. On PostgreSQL,

CockroachDB and MySQL, a preview feature (`relationJoins`) can load everything in a single joined query instead.

Seeing the queries

```
// lib/prisma.ts: print every SQL query Prisma runs
export const prisma = new PrismaClient({ adapter, log: ["query"] });
```

COUNT THE QUERIES

With query logging on, run both versions of the loop and count the lines printed. N+1 problems are easy to miss in development with five rows and very visible in production with five thousand.

Hands-On Exercises

EXERCISE 1

Write a query for a blog's home page: the 10 newest published posts, each with its title, slug, publication date, author's name and tag names — and nothing else. Explain why you chose `select` or `include`.

 [View solution](#)

EXERCISE 2

Write queries for: (a) tags that aren't used on any post; (b) authors who have written at least one post tagged "prisma"; (c) users who have written posts and whose posts are all published.

 [View solution](#)

EXERCISE 3

Turn on query logging, create at least 10 posts, and run this chapter's slow loop and fast version. Count the queries each makes, then explain how the counts would change with 1,000 posts.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- `include` adds relations to all of a model's fields; nest it to go deeper
- Included lists accept `where`, `orderBy` and `take` — bound lists that can grow
- `select` returns only the fields you list; prefer it for data you send elsewhere
- Don't use `include` and `select` at the same level; nesting them is fine
- Relation filters: `some`, `every`, `none` for lists; `is`, `isNot` for single relations; `every` matches empty lists
- N+1: one query per row in a loop; fix it by loading the relation in the list query
- `log: ["query"]` on the client shows every query Prisma runs

CHAPTER 10: CAPSTONE — THE DATA LAYER FOR A SMALL BLOG API

Prisma Fundamentals

Chapter 10 · Capstone: The Data Layer for a Small Blog API

This chapter brings the whole course together in one small working application: a JSON API for the blog, built with Express and Prisma. It uses the final schema, every kind of query from Chapters 5–9, and proper handling of Prisma's errors. Each section names the chapter it comes from.

Step 1: The Final Schema (Chapters 3, 7, 8)

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

datasource db {
  provider = "sqlite"
}

enum Role {
  READER
  AUTHOR
  ADMIN
}

model User {
  id          Int           @id @default(autoincrement())
  email       String        @unique
  name        String?
  role        Role          @default(READER)
  profile     Profile?
  posts       Post[]
  comments    Comment[]
  createdAt   DateTime      @default(now())
  updatedAt   DateTime      @updatedAt
}

model Profile {
```

```

    id      Int      @id @default(autoincrement())
    bio     String?
    userId  Int       @unique
    user    User      @relation(fields: [userId], references: [id], onDelete: Cascade)
  }

  model Post {
    id      Int      @id @default(autoincrement())
    title   String
    slug    String   @unique
    content  String?
    published Boolean @default(false)
    viewCount Int     @default(0)
    publishedAt DateTime?
    authorId Int
    author   User      @relation(fields: [authorId], references: [id])
    tags     Tag[]
    comments Comment[]
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt

    @@index([authorId])
  }

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

  model Comment {
    id      Int      @id @default(autoincrement())
    body     String
    createdAt DateTime @default(now())
    postId   Int
    post     Post     @relation(fields: [postId], references: [id], onDelete: Cascade)
    authorId Int?
    author   User?    @relation(fields: [authorId], references: [id], onDelete: SetNull)

    @@index([postId])
    @@index([authorId])
  }

```

```
npx prisma migrate dev --name capstone      # Chapter 4
npx prisma generate
npm install express@5
npm install --save-dev @types/express
```

Step 2: One Shared Client (Chapters 2, 9)

```
// lib/prisma.ts
import "dotenv/config";
import { PrismaBetterSqlite3 } from "@prisma/adapter-better-sqlite3";
import { PrismaClient } from "../generated/prisma/client";

const adapter = new PrismaBetterSqlite3({ url: process.env.DATABASE_URL! });

export const prisma = new PrismaClient({
  adapter,
  log: process.env.NODE_ENV === "development" ? ["query"] : [],
});
```

Step 3: Listing and Searching Posts (Chapters 6, 9)

```
// server.ts
import express from "express";
import { prisma } from "../lib/prisma";
import { Prisma } from "../generated/prisma/client";

const app = express();
app.use(express.json());
const PAGE_SIZE = 10;

// A reusable "public" shape for posts: never the whole author record
const postSummary = {
  title: true, slug: true, publishedAt: true, viewCount: true,
  author: { select: { name: true } },
  tags: { select: { name: true } },
} satisfies Prisma.PostSelect;

// GET /posts?q=prisma&tag=orm&page=2
```

```

app.get("/posts", async (req, res) => {
  const q    = String(req.query.q ?? "").trim();
  const tag  = String(req.query.tag ?? "").trim();
  const page = Math.max(1, Number(req.query.page) || 1);

  const where: Prisma.PostWhereInput = { published: true };
  if (q)   where.OR    = [{ title: { contains: q } }, { content: { contains: q } }];
  if (tag) where.tags = { some: { name: tag } };

  const [posts, total] = await Promise.all([
    prisma.post.findMany({
      where,
      select: postSummary,
      orderBy: [{ publishedAt: "desc" }, { id: "desc" }],
      skip: (page - 1) * PAGE_SIZE,
      take: PAGE_SIZE,
    }),
    prisma.post.count({ where }),
  ]);

  res.json({ posts, page, totalPages: Math.ceil(total / PAGE_SIZE) });
});

```

WHAT THIS ROUTE GETS RIGHT

Search and tag filters are only added when present, so an empty query can't drop the `published` rule (Chapter 6). `select` keeps author emails out of the response (Chapter 9), and the authors and tags are loaded with the list, so there's no N+1 problem. The same `where` feeds both the page and the count.

Step 4: One Post, With Comments (Chapters 5, 9)

```

app.get("/posts/:slug", async (req, res) => {
  const post = await prisma.post.findFirst({
    where: { slug: req.params.slug, published: true },
    select: {
      ...postSummary,
      content: true,
      comments: {
        orderBy: { createdAt: "asc" },
        take: 50,
      },
    },
  });
  res.json(post);
});

```

```

        select: { body: true, createdAt: true, author: { select: { name: true } } },
      },
    },
  });
  if (!post) return res.status(404).json({ error: "Post not found" });
  res.json(post);
});

```

`findFirst` is used rather than `findUnique` because the condition includes `published`, which isn't unique: a draft with this slug should look like it doesn't exist.

Step 5: Writing Data (Chapters 5, 8)

```

// POST /posts { "title", "slug", "content", "authorEmail", "tags": ["prisma"] }
app.post("/posts", async (req, res) => {
  const { title, slug, content, authorEmail, tags = [] } = req.body;
  if (!title || !slug || !authorEmail) {
    return res.status(400).json({ error: "title, slug and authorEmail are required" });
  }

  const post = await prisma.post.create({
    data: {
      title, slug, content,
      author: { connect: { email: authorEmail } },
      tags: {
        connectOrCreate: [...new Set<string>(tags)].map((name) => ({
          where: { name }, create: { name },
        })),
      },
    },
    select: postSummary,
  });
  res.status(201).json(post);
});

// POST /posts/:slug/publish
app.post("/posts/:slug/publish", async (req, res) => {
  const post = await prisma.post.update({
    where: { slug: req.params.slug },
    data: { published: true, publishedAt: new Date() },
    select: postSummary,
  });
  res.status(200).json(post);
});

```

```

});
res.json(post);
});

// POST /posts/:slug/view -- safe under concurrent requests
app.post("/posts/:slug/view", async (req, res) => {
  const { viewCount } = await prisma.post.update({
    where: { slug: req.params.slug },
    data: { viewCount: { increment: 1 } },
    select: { viewCount: true },
  });
  res.json({ viewCount });
});

```

Step 6: Turning Prisma Errors Into HTTP Responses (Chapters 5, 7)

Express 5 passes errors from `async` route handlers to the error handler automatically, so one function can translate Prisma's error codes for every route:

```

app.use((err: unknown, req: express.Request, res: express.Response, next: express.NextFunction) => {
  if (err instanceof Prisma.PrismaClientKnownRequestError) {
    if (err.code === "P2002") return res.status(409).json({ error: "That value is already taken" });
    if (err.code === "P2025") return res.status(404).json({ error: "Not found" });
    if (err.code === "P2003") return res.status(409).json({ error: "Other records still exist" });
  }
  console.error(err);
  res.status(500).json({ error: "Something went wrong" });
});

const server = app.listen(3000, () => console.log("Blog API on http://localhost:3000"));

// Close the database connection cleanly on shutdown
process.on("SIGTERM", () => server.close(() => prisma.$disconnect()));

```

REQUEST	PRISMA RESULT	HTTP RESPONSE
Create a post with a slug that exists	<code>P2002</code> unique constraint	409 Conflict
Publish or view a slug that doesn't exist	<code>P2025</code> record not found	404 Not Found

REQUEST	PRISMA RESULT	HTTP RESPONSE
Create a post for an unknown author email	<code>connect</code> finds no user (<code>P2025</code>)	404 Not Found
Delete a user who still has posts	Restricted by the foreign key (<code>P2003</code>)	409 Conflict

Trying It Out

```
npx tsx server.ts

# in another terminal
curl -X POST localhost:3000/posts -H "Content-Type: application/json" \
  -d '{"title":"Hello API","slug":"hello-api","authorEmail":"alan@example.com","tags":['
curl -X POST localhost:3000/posts/hello-api/publish
curl "localhost:3000/posts?tag=prisma"
curl localhost:3000/posts/hello-api
```

Where Each Part Came From

CHAPTER	USED IN THIS CAPSTONE
1 — Where Prisma Fits	The reason for a typed, schema-first data layer
2 — Project Setup	The Prisma 7 client with a driver adapter, in one shared file
3 — The Schema	Models, enum, defaults, <code>@unique</code> , <code>@updatedAt</code>
4 — Migrate	Applying the final schema, then <code>generate</code>
5 — CRUD	<code>create</code> , <code>update</code> , <code>increment</code> , error codes
6 — Filtering & Pagination	Search, <code>OR</code> , safe optional filters, sorted offset paging
7 — Relations I	Author, profile and comment relations; <code>Restrict</code> , <code>Cascade</code> , <code>SetNull</code>
8 — Relations II	Tags, <code>connect</code> and <code>connectOrCreate</code> nested writes
9 — Reading Related Data	Nested <code>select</code> , relation filters, no N+1, query logging

What's Deliberately Missing

- **Authentication and permissions** — anyone can create or publish posts. A real API must check who's asking.
- **Input validation** beyond basic checks — a library such as Zod would validate request bodies properly.
- **Transactions, aggregation, raw SQL, testing, performance and production migrations** — the subjects of Prisma Intermediate/Advanced, which starts where this course ends.

Hands-On Exercises

EXERCISE 1

Add `POST /posts/:slug/comments`, which lets a guest or a registered user (by optional email) add a comment to a published post. Return 404 for missing or unpublished posts.

 [View solution](#)

EXERCISE 2

Add `GET /authors/:email`, returning the author's name, bio and their five most-viewed published posts, without exposing their email in the response body.

 [View solution](#)

EXERCISE 3

Add `PUT /posts/:slug/tags`, which replaces a post's tags with a given list, and `GET /tags/unused`, which lists tags on no posts. Test both with `curl`.

 [View solution](#)

COURSE QUICK REFERENCE

- **Setup:** pin `@7`; generator `prisma-client` with `output`; driver adapter; `prisma7.config.ts` with `dotenv`
- **Schema:** models, types, `?`, `@id`, `@default`, `@unique`, `@updatedAt`, `@map`, enums, `@@id` / `@@unique` / `@@index`
- **Migrate:** `migrate dev --name` then `generate`; `--create-only` to edit; `migrate deploy` in production
- **CRUD:** `create`, `findUnique` / `findFirst` / `findMany`, `update`, `upsert`, `delete`, `*Many`, `increment`
- **Queries:** operators, `AND` / `OR` / `NOT`, case sensitivity by database, `orderBy`, offset and cursor paging
- **Relations:** `@relation(fields, references)`, one-to-one via `@unique`, implicit and explicit many-to-many, `onDelete`
- **Nested writes:** `create`, `connect`, `connectOrCreate`, `disconnect`, `set`
- **Reading:** `include` vs `select`, `some` / `every` / `none`, avoid N+1, `log: ["query"]`
- **Errors:** `P2002` duplicate, `P2025` not found, `P2003` foreign key

Course Complete

This completes **Prisma Fundamentals**, 10/10 chapters. Next: **Prisma Intermediate/Advanced**.