

DATABASES

Prisma Intermediate/Advanced

The second Prisma 7 course — transactions and concurrency, aggregation, raw SQL and TypedSQL, seeding, testing, client extensions, performance, safe schema evolution and deployment — closing with a zero-downtime schema change on a live blog API.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: TRANSACTIONS

Prisma Intermediate/Advanced

Chapter 1 · Transactions

This course picks up where Prisma Fundamentals left off, using its capstone blog API as the starting project. The first topic is the one that separates a working demo from a trustworthy application: **transactions**, which make several database changes succeed or fail together, and **concurrency control**, which stops two people's changes from silently overwriting each other.

Why Transactions Matter

In Prisma Fundamentals, Chapter 7, deleting a user who had posts meant two steps: reassign or delete their posts, then delete the user. Run them as two separate queries, and a crash or error between them leaves the data half-changed.

```
// Two separate operations: NOT safe
await prisma.post.updateMany({ where: { authorId: 7 }, data: { authorId: 1 } });
// <-- if the server crashes here, the posts have moved but the user still exists
await prisma.user.delete({ where: { id: 7 } });
```

A **transaction** groups operations so the database treats them as one unit. Either every change is saved (*committed*), or none is (*rolled back*). Other queries never see a half-finished state.

PROPERTY	MEANING
Atomic	All the changes happen, or none do
Consistent	The database's rules (unique values, foreign keys) hold before and after
Isolated	Transactions running at the same time don't see each other's unfinished work
Durable	Once committed, changes survive a crash

Together these are known as **ACID**. Prisma gives you three ways to use transactions.

1. Nested Writes (Already Transactions)

Every nested write from Prisma Fundamentals, Chapter 8 already runs as a transaction. Creating a user with a profile and posts in one `create` call either saves all of them or none. When the operations are about one record and its relations, a nested write is the simplest choice.

2. Batch Transactions: `$transaction([...])`

```
// Reassign a user's posts, then delete the user -- together
const [moved, deleted] = await prisma.$transaction([
  prisma.post.updateMany({ where: { authorId: 7 }, data: { authorId: 1 } }),
  prisma.user.delete({ where: { id: 7 } }),
]);
console.log(`Moved ${moved.count} posts`);
```

You pass a list of queries, *without* `await` on each one. Prisma runs them in order inside one transaction and returns their results in the same order. If any fails, all are rolled back.

DON'T AWAIT THE QUERIES INSIDE THE LIST

Writing `await prisma.post.updateMany(...)` inside the array runs that query immediately, on its own, before the transaction starts. Build the queries without `await`, and await only `$transaction`.

The limitation: every query is decided in advance. You can't look at the result of the first query and decide what the second should do.

3. Interactive Transactions: `$transaction(async (tx) => ...)`

When later steps depend on earlier results, pass a function instead of a list:

```
async function deleteUserKeepingPosts(userId: number) {
  return prisma.$transaction(async (tx) => {
    // Find (or create) a placeholder account to own orphaned posts
    const ghost = await tx.user.upsert({
      where: { email: "deleted@blog.local" },
      update: {},
      create: { email: "deleted@blog.local", name: "Deleted user" },
    });
  });
}
```

```

    if (ghost.id === userId) throw new Error("Can't delete the placeholder account");

    const moved = await tx.post.updateMany({
      where: { authorId: userId },
      data: { authorId: ghost.id },
    });
    await tx.user.delete({ where: { id: userId } });

    return moved.count;
  });
}

```

- Inside the function, use `tx`, not `prisma`. Only queries made through `tx` are part of the transaction.
- If the function throws, everything is rolled back. If it returns, everything is committed, and `$transaction` returns the value.

USING PRISMA INSTEAD OF TX IS A SILENT BUG

commits on its own and isn't rolled back if the transaction fails later. TypeScript won't warn you, so check every line.

Timeouts: keep transactions short

```

await prisma.$transaction(
  async (tx) => { /* ... */ },
  {
    maxWait: 5000, // max time to wait to start (default 2000 ms)
    timeout: 10000, // max time the transaction may run (default 5000 ms)
  },
);

```

An open transaction holds locks that can block other requests. Prisma cancels and rolls back a transaction that runs past its `timeout`. Never do slow work inside one — calling an external API, sending an email, waiting for a user. Gather what you need first, keep the transaction to database work, and do the rest after it commits.

Isolation Levels

Isolation controls how much concurrent transactions can see of each other. Stricter levels prevent more anomalies but allow less work to happen in parallel.

DATABASE	LEVELS PRISMA SUPPORTS
PostgreSQL, MySQL	<code>ReadUncommitted</code> , <code>ReadCommitted</code> , <code>RepeatableRead</code> , <code>Serializable</code>
SQL Server	All of those, plus <code>Snapshot</code>
SQLite, CockroachDB	<code>Serializable</code> only

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

await prisma.$transaction(async (tx) => { /* ... */ }, {
  isolationLevel: Prisma.TransactionIsolationLevel.Serializable,
});
```

With stricter levels, the database may abort a transaction that conflicts with another. Prisma reports this as error `P2034`, and the right response is simply to try the transaction again:

```
async function withRetry<T>(fn: () => Promise<T>, attempts = 3): Promise<T> {
  for (let i = 1; ; i++) {
    try {
      return await fn();
    } catch (e) {
      const conflict = e instanceof Prisma.PrismaClientKnownRequestError && e.code ===
        if (!conflict || i >= attempts) throw e;
    }
  }
}
```

Optimistic Concurrency: Stopping Lost Updates

Two editors open the same post. Ada fixes a typo and saves; a minute later Grace, still looking at the old version, rewrites a paragraph and saves. Grace's save silently wipes out Ada's fix. A

transaction doesn't help here, because the two saves happen minutes apart.

The standard fix is a **version number**. Each save says "update this post only if it's still the version I loaded," and increases the version:

```
// schema.prisma: add to Post
version Int @default(0)

// Saving an edit
async function savePost(id: number, loadedVersion: number, content: string) {
  const result = await prisma.post.updateMany({
    where: { id, version: loadedVersion },           // only if nobody saved in between
    data: { content, version: { increment: 1 } },
  });
  if (result.count === 0) {
    throw new Error("Someone else changed this post. Reload and try again.");
  }
}
```

Ada loads version 3 and saves: the post becomes version 4. Grace, who also loaded version 3, then tries to save — but no post matches `version: 3` any more, so nothing is changed and she's told to reload. No locks are held while people are editing; conflicts are simply detected when they happen. That's why it's called *optimistic*.

Choosing the Right Tool

SITUATION	USE
Creating or updating a record with its related records	A nested write
Several independent changes that must succeed together	<code>\$transaction([...])</code>
Later steps depend on earlier results	An interactive transaction
A single counter update	<code>{ increment: 1 }</code> — no transaction needed
People editing the same data over minutes	A version field (optimistic concurrency)

Hands-On Exercises

EXERCISE 1

Using a batch transaction, publish every draft post by one author and set their role to `AUTHOR`, all or nothing. Then deliberately make the second query fail and prove the first was rolled back.

 [View solution](#)

EXERCISE 2

Write an interactive transaction `mergeTags(fromName, intoName)` that moves every post from one tag to another and then deletes the old tag. It should throw, leaving everything unchanged, if either tag doesn't exist.

 [View solution](#)

EXERCISE 3

Add a `version` field to `Post` and a `PATCH /posts/:slug` route to the Fundamentals capstone API that uses it. Simulate two editors saving from the same loaded version and show that the second gets a 409.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- A transaction makes several changes commit or roll back together (ACID)
- Nested writes are already transactions
- `$transaction([q1, q2])` runs pre-built queries in order, atomically — don't `await` them individually
- `$transaction(async (tx) => ...)` for logic between steps; use `tx` for every query; throw to roll back
- Options: `maxWait` (default 2000 ms), `timeout` (default 5000 ms), `isolationLevel`; keep transactions short
- SQLite and CockroachDB support only `Serializable`; retry on `P2034` conflicts

- Optimistic concurrency: a `version` field checked in `where` and incremented on save

CHAPTER 2: AGGREGATION & GROUPING

Prisma Intermediate/Advanced

Chapter 2 · Aggregation & Grouping

So far every query has returned records. Real applications also need **numbers about records**: how many posts are published, the average view count, which author wrote the most. Fetching every row and adding it up in JavaScript works for ten posts and falls over at ten thousand. This chapter shows how to make the database do that arithmetic, using the same blog schema as before.

Counting: count()

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

// Count all rows AND the rows where a nullable field is filled in
const users = await prisma.user.count({
  select: { _all: true, name: true },
});
// { _all: 42, name: 37 } -> 5 users have no name
```

`count()` accepts the same `where` filters as `findMany`, so everything from Prisma Fundamentals, Chapter 6 carries over.

Summary Statistics: aggregate()

```
const stats = await prisma.post.aggregate({
  where: { published: true },
  _count: { _all: true },
  _sum: { viewCount: true },
  _avg: { viewCount: true },
  _max: { viewCount: true, createdAt: true },
  _min: { createdAt: true },
});
```

```
// {
//   _count: { _all: 18 },
//   _sum:   { viewCount: 2140 },
//   _avg:   { viewCount: 118.9 },
//   _max:   { viewCount: 610, createdAt: 2026-09-20T... },
//   _min:   { createdAt: 2026-03-02T... }
// }
```

OPERATOR	WORKS ON	NOTES
<code>_count</code>	Any field, or <code>_all</code>	Counts non-null values; always a number, <code>0</code> when nothing matches
<code>_sum</code> , <code>_avg</code>	Numeric fields	<code>_avg</code> of an <code>Int</code> field can be a fraction
<code>_min</code> , <code>_max</code>	Numbers, dates, strings	Handy for "oldest" and "newest"

NO ROWS MEANS NULL, NOT ZERO

If the `where` matches nothing, `_sum`, `_avg`, `_min` and `_max` come back as `null`. Only `_count` is guaranteed to be a number. Handle it explicitly — `stats._sum.viewCount ?? 0` — rather than letting `null` reach a template or a calculation.

Per-Group Figures: `groupBy()`

`aggregate` gives one set of numbers for everything matched. `groupBy` gives one set *per group* — the Prisma equivalent of SQL's `GROUP BY`.

```
// Published post count and total views, per author
const perAuthor = await prisma.post.groupBy({
  by:    ["authorId"],
  where: { published: true },
  _count: { _all: true },
  _sum:   { viewCount: true },
  orderBy: { _sum: { viewCount: "desc" } },
});
// [
//   { authorId: 3, _count: { _all: 7 }, _sum: { viewCount: 1200 } },
//   { authorId: 1, _count: { _all: 9 }, _sum: { viewCount: 820 } },
```

```
// ...
// ]
```

where vs. having

OPTION	FILTERS	EXAMPLE
where	Individual rows, <i>before</i> grouping	Only published posts
having	Whole groups, <i>after</i> grouping	Only authors with at least 5 posts

```
// Authors with 5 or more published posts
const prolific = await prisma.post.groupBy({
  by:    ["authorId"],
  where: { published: true },
  _count: { _all: true },
  having: { authorId: { _count: { gte: 5 } } },
});
```

Inside `having` you can only filter on aggregate values, or on fields listed in `by`.

PAGING GROUPS NEEDS ORDERBY

If you use `skip` or `take` with `groupBy`, you must also give an `orderBy`. Without a defined order, "the first ten groups" doesn't mean anything, so Prisma refuses the query.

groupBy returns IDs, not names

Groups contain only the `by` fields and the aggregates — `groupBy` can't `include` relations. To show author names, fetch them in a second query and join in code:

```
const authors = await prisma.user.findMany({
  where: { id: { in: perAuthor.map((g) => g.authorId) } },
  select: { id: true, name: true },
});
const nameById = new Map(authors.map((a) => [a.id, a.name]));

const report = perAuthor.map((g) => ({
  author: nameById.get(g.authorId) ?? "(unknown)",
  posts: g._count._all,
```

```
views: g._sum.viewCount ?? 0,
});
```

Two queries, whatever the number of authors — not one query per author.

Counting Relations: `_count`

Often the simplest option is to ask for a count alongside each record, using `_count` inside `select` or `include`:

```
const authors = await prisma.user.findMany({
  select: {
    name: true,
    _count: {
      select: {
        posts: { where: { published: true } }, // count only published posts
      },
    },
  },
  orderBy: { posts: { _count: "desc" } }, // most posts first
});
// [{ name: "Ada", _count: { posts: 9 } }, ...]
```

WHICH ONE SHOULD I USE?

Use relation `_count` when you're listing records anyway and want a number next to each. Use `groupBy` when the numbers *are* the result. Note that `orderBy` on a relation count sorts by *all* related posts; the `where` inside `_count` only affects the number returned.

distinct

```
// Which roles are actually in use?
const roles = await prisma.user.findMany({
  distinct: ["role"],
  select: { role: true },
});
```

Prisma applies `distinct` by filtering results after they're fetched rather than with SQL's `SELECT DISTINCT`. It's convenient, but on a large table it still reads every matching row — for "how many of each," prefer `groupBy`.

A Stats Endpoint for the Blog API

```
app.get("/stats", async (req, res) => {
  const [posts, drafts, views, topTags] = await prisma.$transaction([
    prisma.post.count({ where: { published: true } }),
    prisma.post.count({ where: { published: false } }),
    prisma.post.aggregate({ where: { published: true }, _sum: { viewCount: true } }),
    prisma.tag.findMany({
      select: { name: true, _count: { select: { posts: true } } },
      orderBy: { posts: { _count: "desc" } },
      take: 5,
    }),
  ]);

  res.json({
    published: posts,
    drafts,
    totalViews: views._sum.viewCount ?? 0,
    topTags: topTags.map((t) => ({ name: t.name, posts: t._count.posts })),
  });
});
```

Wrapping the four reads in `$transaction([...])` from Chapter 1 means they all see the same snapshot of the data, so the numbers are consistent with each other.

Hands-On Exercises

EXERCISE 1

Write one query that returns the number of published posts, the total and average view count, and the date of the most recent published post. Then run it with a `where` that matches nothing and show safe handling of the `null` values.

[View solution](#)

EXERCISE 2

Using `groupBy`, list authors with at least 3 published posts and an average view count above 50, most-viewed first, showing each author's name.

[View solution](#)

EXERCISE 3

Add `GET /tags` to the blog API, returning every tag with its number of *published* posts, and add `?unused=true` support that returns only tags with no posts at all. Explain which technique you used for each and why.

[View solution](#)

CHAPTER 2 QUICK REFERENCE

- `count({ where })`; `select: { _all: true, field: true }` counts rows and non-null values
- `aggregate` with `_count`, `_sum`, `_avg`, `_min`, `_max`
- Empty results: `_count` is `0`, everything else is `null` — use `?? 0`
- `groupBy({ by, where, having, orderBy })`; `where` filters rows, `having` filters groups
- `skip` / `take` on `groupBy` require `orderBy`; groups hold IDs, so look names up separately
- Relation counts: `_count: { select: { posts: { where } } }`; sort with `orderBy: { posts: { _count: "desc" } }`
- `distinct` filters in memory; prefer `groupBy` for counts on big tables

CHAPTER 3: RAW SQL

Prisma Intermediate/Advanced

Chapter 3 · Raw SQL

Prisma Client covers most everyday queries, but not all of them. Window functions, full-text search, common table expressions, database-specific features and some bulk updates are easier — or only possible — in plain SQL. Prisma lets you drop down to SQL for those queries without giving up its connection handling or, if you write the SQL carefully, its protection against SQL injection.

Examples use PostgreSQL. Prisma names each table after its model, so the `Post` model is the table `"Post"`. PostgreSQL needs the double quotes because the name has a capital letter. (If your schema uses `@@map`, use the mapped name instead.)

\$queryRaw and \$executeRaw

METHOD	USE FOR	RETURNS
<code>\$queryRaw</code>	SELECT (anything that returns rows)	An array of row objects
<code>\$executeRaw</code>	UPDATE, DELETE, INSERT without RETURNING	The number of rows affected

```
// Rank each author's posts by views, using a window function
type RankedPost = { title: string; authorId: number; viewCount: number; rank: bigint };

const ranked = await prisma.$queryRaw<RankedPost[]>`
  SELECT "title", "authorId", "viewCount",
         RANK() OVER (PARTITION BY "authorId" ORDER BY "viewCount" DESC) AS "rank"
  FROM "Post"
  WHERE "published" = true
`;

// Archive old drafts in one statement
const cutoff = new Date("2026-01-01");
const deleted = await prisma.$executeRaw`
  DELETE FROM "Post" WHERE "published" = false AND "createdAt" < ${cutoff}
```

```
`;  
console.log(`Removed ${deleted} old drafts`);
```

Notice these are **tagged templates**: backticks directly after the method name, with no parentheses. That detail is what keeps them safe.

Why Tagged Templates Stop SQL Injection

When you write `${cutoff}` inside a tagged template, Prisma does not paste the value into the SQL text. It sends the SQL with a placeholder and passes the value separately, as a **parameter**. The database treats a parameter as data, never as SQL, whatever characters it contains.

```
const search = "x' OR '1'='1"; // hostile input  
  
// SAFE: the whole string is one parameter; matches nothing  
await prisma.$queryRaw`SELECT * FROM "Post" WHERE "title" = ${search}`;  
  
// DANGEROUS: the string becomes part of the SQL; matches EVERY post  
await prisma.$queryRawUnsafe(`SELECT * FROM "Post" WHERE "title" = '${search}'`);
```

THE UNSAFE METHODS MEAN EXACTLY THAT

`$queryRawUnsafe` and `$executeRawUnsafe` take an ordinary string. If you build that string with `+` or a normal template literal, user input becomes SQL. Prisma's docs describe this as a significant risk of SQL injection. If you must use them, pass values as extra arguments with placeholders (`$queryRawUnsafe('... WHERE "title" = $1', search)`) — never inside the string.

Building Queries in Pieces: Prisma.sql

Search endpoints often need optional conditions. `Prisma.sql` builds a safe SQL fragment you can combine with others, and `Prisma.empty` stands for "nothing here":

```
import { Prisma } from "../generated/prisma/client";  
  
function searchPosts(opts: { text?: string; authorIds?: number[] }) {  
  const textFilter = opts.text
```

```
    ? Prisma.sql`AND "title" ILIKE ${"%" + opts.text + "%`} `
    : Prisma.empty;

const authorFilter = opts.authorIds?.length
  ? Prisma.sql`AND "authorId" IN (${Prisma.join(opts.authorIds)})`
  : Prisma.empty;

return prisma.$queryRaw`
  SELECT "id", "title", "slug" FROM "Post"
  WHERE "published" = true ${textFilter} ${authorFilter}
  ORDER BY "createdAt" DESC
  LIMIT 20
`;
}
```

HELPER	DOES	SAFE WITH USER INPUT?
Prisma.sql`...`	A reusable, parameterised fragment	Yes
Prisma.join(list)	A comma-separated list of parameters, for <code>IN (...)</code>	Yes
Prisma.empty	An empty fragment, for optional parts	Yes
Prisma.raw(text)	Inserts text directly into the SQL	No — trusted text only

Column and table names can't be parameters

Parameters stand for *values*. Column names, table names and keywords like `ASC` / `DESC` can't be passed as parameters. If a user chooses the sort column, check it against a fixed list first:

```
const SORTABLE = { newest: '"createdAt" DESC', popular: '"viewCount" DESC' } as const;

const key = req.query.sort in SORTABLE ? (req.query.sort as keyof typeof SORTABLE) : "newest";
const orderBy = Prisma.raw(SORTABLE[key]); // safe: only our own fixed strings reach the DB

await prisma.$queryRaw`SELECT "title" FROM "Post" ORDER BY ${orderBy} LIMIT 10`;
```

What Comes Back: Types

Raw results skip Prisma's model types. The generic in `$queryRaw<T>` is a *promise you make* to TypeScript — Prisma doesn't check it. And the database's own types come through:

DATABASE TYPE	JAVASCRIPT VALUE
64-bit integers, including <code>COUNT(*)</code> on PostgreSQL	<code>bigint</code>
Numeric / decimal	<code>Decimal</code>
Dates and timestamps	<code>Date</code>
Bytes	<code>Uint8Array</code>

RES.JSON() CAN'T SEND A BIGINT

`JSON.stringify` throws `TypeError: Do not know how to serialize a BigInt`. The `rank` column above and any `COUNT(*)` will trigger it. Convert with `Number(row.rank)` before sending, or cast in SQL: `COUNT(*)::int`.

TypedSQL: Real Types for Raw Queries

TypedSQL fixes the "trust me" typing problem. You write each query in its own `.sql` file, and `prisma generate --sql` connects to your database, works out the real parameter and column types, and generates a typed function. It's a **preview feature**, so enable it first:

```
// schema.prisma
generator client {
  provider      = "prisma-client"
  output        = "../generated/prisma"
  previewFeatures = ["typedSql"]
}
```

```
-- prisma/sql/topPostsByAuthor.sql
-- @param {Int} $1:authorId
-- @param {Int} $2:limit
SELECT "title", "slug", "viewCount"
FROM "Post"
WHERE "authorId" = $1 AND "published" = true
```

```
ORDER BY "viewCount" DESC
LIMIT $2;
```

```
npx prisma generate --sql
```

```
import { topPostsByAuthor } from "../generated/prisma/sql";

const rows = await prisma.$queryRawTyped(topPostsByAuthor(3, 5));
// rows is typed from the real columns: title, slug, viewCount
```

- The file name becomes the function name, so it must be a valid JavaScript identifier.
- Placeholders follow your database: `$1` for PostgreSQL, `?` for MySQL; SQLite accepts `$1`, `?` or named parameters.
- Generating needs a live database connection, which matters in CI.
- It can't handle dynamic column lists or optional clauses — use `$queryRaw` with `Prisma.sql` for those.
- Type detection is weaker on SQLite and MySQL before 8.0, where you may need `@param` annotations for every parameter.

When to Reach for Raw SQL

SITUATION	RECOMMENDATION
Normal CRUD, filtering, relations	Prisma Client
A fixed complex report or window function	TypedSQL
A query with optional parts	<code>\$queryRaw</code> with <code>Prisma.sql</code> and <code>Prisma.empty</code>
Dynamic sort column	Allow-list, then <code>Prisma.raw</code> on your own strings
Anything built from user input by string concatenation	Never

RAW SQL CAN STILL BE TRANSACTIONAL

`tx.$queryRaw` and `tx.$executeRaw` work inside an interactive transaction, and raw queries can go in a `$transaction([...])` list too, so Chapter 1's guarantees still apply.

Hands-On Exercises

EXERCISE 1

Using `$queryRaw`, return the number of published posts per month of the current year, then send it from an Express route without the BigInt error.

 [View solution](#)

EXERCISE 2

This code is vulnerable. Explain the attack, then rewrite it safely:

```
prisma.$queryRawUnsafe(`SELECT * FROM "User" WHERE "email" = '${req.query.email}'`)
```

 [View solution](#)

EXERCISE 3

Build `GET /search?q=&tag=&sort=` with raw SQL where every part is optional: a title search, a tag name, and `sort` of `newest` or `popular`.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- `$queryRaw`...`` returns rows; `$executeRaw`...`` returns the number of affected rows
- Tagged template values become parameters — safe from SQL injection
- `*Unsafe` methods take a plain string — never put user input in it
- `Prisma.sql`, `Prisma.join`, `Prisma.empty` are safe; `Prisma.raw` is for trusted text only
- Identifiers and keywords can't be parameters — use an allow-list
- 64-bit integers come back as `bigint`; convert before `res.json()`

- TypedSQL (preview): `prisma/sql/*.sql`, `prisma generate --sql`, `$queryRawTyped(fn(args))`

CHAPTER 4: SEEDING & TEST DATA

Prisma Intermediate/Advanced

Chapter 4 · Seeding & Test Data

A fresh database is empty. Before you can click around the blog, try a new feature, or run the tests in the next chapter, you need data in it — ideally the *same* data every time, created by one command. That's what a **seed script** is for.

Two Kinds of Seed Data

KIND	EXAMPLES	WHERE IT RUNS
Reference data	Required categories, an admin account, default settings	Every environment, including production
Sample data	Fifty fake users, two hundred posts	Development and testing only

Keep them separate. Fake users in production are a real, and embarrassing, mistake.

Configuring the Seed Command

In Prisma 7 the seed command lives in the Prisma config file (`prisma.config.ts`, or `prisma7.config.ts` from 7.10 — see Prisma Fundamentals, Chapter 2), under `migrations`:

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

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

```
npm install -D tsx
npx prisma db seed
```

SEEDING IS NO LONGER AUTOMATIC

Up to Prisma 6, `migrate dev` and `migrate reset` ran the seed script for you. Prisma 7 removed that. After resetting a database, run `npx prisma db seed` yourself — or add an npm script that does both, such as `"db:fresh": "prisma migrate reset --force && prisma db seed"`.

A First Seed Script

```
// prisma/seed.ts
import { prisma } from "../lib/prisma"; // the shared client + adapter from Prisma Fullstack

async function main() {
  await prisma.user.create({
    data: {
      email: "ada@example.com",
      name: "Ada",
      role: "ADMIN",
      posts: { create: [{ title: "Hello", slug: "hello", content: "First post", published: true }] },
    },
  });
}

main()
  .catch((e) => { console.error(e); process.exitCode = 1; })
  .finally(() => prisma.$disconnect());
```

It works once. Run it a second time and it fails with `P2002`: `ada@example.com` already exists. A seed script that can only run on an empty database is fragile.

Idempotent Seeds

An **idempotent** script gives the same end result however many times you run it. There are two common ways to get there.

Option 1: upsert by a unique field

```
const tagNames = ["prisma", "orm", "typescript"];
for (const name of tagNames) {
  await prisma.tag.upsert({ where: { name }, update: {}, create: { name } });
}

await prisma.user.upsert({
  where: { email: "ada@example.com" },
  update: { name: "Ada", role: "ADMIN" }, // keep it in the state we want
  create: { email: "ada@example.com", name: "Ada", role: "ADMIN" },
});
```

Best for **reference data**: it's safe to run against a database that already holds real data, because it only touches the rows it names.

Option 2: clear, then create

```
await prisma.$transaction([
  prisma.post.deleteMany(), // children first...
  prisma.profile.deleteMany(),
  prisma.tag.deleteMany(),
  prisma.user.deleteMany(), // ...parents last
]);
```

Best for **sample data** in development. Delete children before parents, following each relation's `onDelete` rule. In the blog schema from Prisma Fundamentals, `Post.author` is required and uses the default rule, so deleting a user who still has posts fails with `P2003`; `Profile` and `Comment` are handled automatically (a profile is deleted with its user, comments are deleted with their post, and a comment's author is set to `null` when the user goes). Never run this against production — guard it:

```
if (process.env.NODE_ENV === "production") {
  throw new Error("Refusing to wipe data in production");
}
```

Realistic Fake Data With Faker

```
npm install -D @faker-js/faker
```

```
import { faker } from "@faker-js/faker";

faker.seed(42); // same "random" data on every run

const tags = await prisma.tag.findMany();

for (let i = 0; i < 20; i++) {
  await prisma.user.create({
    data: {
      email: faker.internet.email().toLowerCase(),
      name: faker.person.fullName(),
      posts: {
        create: Array.from({ length: faker.number.int({ min: 0, max: 6 }) }, (_, j) => {
          title: faker.lorem.sentence({ min: 3, max: 7 }),
          slug: `post-${i}-${j}`,
          content: faker.lorem.paragraphs(3),
          published: faker.datatype.boolean({ probability: 0.7 }),
          viewCount: faker.number.int({ max: 500 }),
          tags: { connect: faker.helpers.arrayElements(tags, { min: 0, max: 3 }).map((t) => ({
            id: t.id
          })),
        },
      },
    },
  });
}
```

WHY SEED THE RANDOM GENERATOR?

`faker.seed(42)` makes Faker produce the same sequence every run. A bug you found in "the seeded data" will still be there tomorrow, and a teammate running the same script gets the same database. Change the number when you want a different set.

RANDOM VALUES CAN STILL COLLIDE

Faker can generate the same email twice. With a `@unique` column that throws `P2002`. The slugs above avoid it by building from the loop counters; for emails, add the index

(`user\${i}@example.com`) or use Faker's `unique` helpers where your version provides them.

Speed: createMany

One `create` per row means one database round trip per row. For flat data without nested relations, `createMany` inserts everything in one statement:

```
await prisma.tag.createMany({
  data: ["prisma", "orm", "typescript", "sql"].map((name) => ({ name })),
  skipDuplicates: true, // ignore rows that would break a unique constraint
});
```

CREATEMANY LIMITS

`createMany` can't create nested relations, and `skipDuplicates` isn't supported on every database (SQLite, MongoDB and SQL Server don't support it). Check before relying on it; for those databases, use `upsert` or filter out existing rows first.

Organising a Bigger Seed

```
prisma/
├─ seed.ts          <- decides what to run
└─ seeds/
  ├─ reference.ts   <- tags, admin user (upserts; safe anywhere)
  └─ sample.ts      <- fake users and posts (dev/test only)
```

```
// prisma/seed.ts
import { seedReference } from "../seeds/reference";
import { seedSample } from "../seeds/sample";

await seedReference();
if (process.env.NODE_ENV !== "production") {
  await seedSample();
}
```

Hands-On Exercises

EXERCISE 1

Write `seeds/reference.ts`: an admin user and five fixed tags, fully idempotent. Run it three times and show the row counts don't change.

 [View solution](#)

EXERCISE 2

Write `seeds/sample.ts`: clear the sample data, then create 10 users with 0–5 posts each using a seeded Faker. Make sure it can't run in production and never deletes the admin user or the reference tags.

 [View solution](#)

EXERCISE 3

Add npm scripts so that one command resets the database, applies every migration and seeds it. Explain why this matters more in Prisma 7 than it did in Prisma 6.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- Configure `migrations.seed` (e.g. `"tsx prisma/seed.ts"`) in the Prisma config file
- Run with `npx prisma db seed` — Prisma 7 no longer seeds automatically after `migrate dev` / `reset`
- Reference data: `upsert` by a unique field; safe in every environment
- Sample data: delete children before parents, then create; guard against production
- `faker.seed(n)` gives repeatable fake data; build unique fields from counters

- `createMany` for flat bulk inserts; no nested relations; `skipDuplicates` isn't available on every database

CHAPTER 5: TESTING CODE THAT USES PRISMA

Prisma Intermediate/Advanced

Chapter 5 · Testing Code That Uses Prisma

Code that talks to a database is awkward to test: tests need data, they can interfere with each other, and a real database is slower than pure functions. There are two main approaches — replace Prisma Client with a **mock**, or run the tests against a **real test database**. They catch different kinds of bug, and most projects end up using both.

Examples use **Vitest**, which works smoothly with the ES modules Prisma 7 uses. Jest works too; Prisma's own documentation uses Jest with `jest-mock-extended`.

The Code Under Test

```
// src/posts.ts
import type { PrismaClient } from "../generated/prisma/client";

export function slugify(title: string) {
  return title.toLowerCase().trim().replace(/^[a-z0-9]+/g, "-").replace(/^-|-$/g, "");
}

export async function createPost(
  db: PrismaClient,
  input: { title: string; content: string; authorId: number },
) {
  if (input.title.trim().length < 3) throw new Error("Title too short");
  return db.post.create({
    data: { ...input, slug: slugify(input.title), published: false },
  });
}
```

Notice `createPost` receives the client as a parameter (`db`) rather than importing it. This is **dependency injection**: the app passes the real client, a test passes a fake one. It's the simplest way to make database code testable.

SEPARATE PURE LOGIC FIRST

`slugify` doesn't touch the database at all, so it needs neither a mock nor a test database — just ordinary tests. The more logic you move into plain functions like this, the less database testing you need.

Approach 1: Mocking Prisma Client

```
npm install -D vitest vitest-mock-extended
```

```
// test/posts.unit.test.ts
import { describe, it, expect, beforeEach } from "vitest";
import { mockDeep, mockReset, type DeepMockProxy } from "vitest-mock-extended";
import type { PrismaClient } from "../generated/prisma/client";
import { createPost } from "../src/posts";

const db: DeepMockProxy<PrismaClient> = mockDeep<PrismaClient>();
beforeEach(() => mockReset(db)); // no leftovers between tests

describe("createPost", () => {
  it("saves a draft with a slug built from the title", async () => {
    db.post.create.mockResolvedValue({ id: 1, slug: "hello-world" } as any);

    await createPost(db, { title: "Hello, World!", content: "...", authorId: 3 });

    expect(db.post.create).toHaveBeenCalledWith({
      data: expect.objectContaining({ slug: "hello-world", published: false, authorId: 3 }),
    });
  });

  it("rejects a short title without touching the database", async () => {
    await expect(createPost(db, { title: "Hi", content: "", authorId: 3 }))
      .rejects.toThrow("Title too short");
    expect(db.post.create).not.toHaveBeenCalled();
  });
});
```

`mockDeep` builds a fake client where every model and method exists and is typed. You decide what each method returns with `mockResolvedValue`, and check how it was called with

`toHaveBeenCalledWith`. No database is involved, so these tests run in milliseconds.

If your code imports a shared client

Code written as in Prisma Fundamentals imports `prisma` from `lib/prisma` directly. Vitest can swap that module for a mock:

```
import { vi } from "vitest";
import { mockDeep } from "vitest-mock-extended";
import type { PrismaClient } from "../generated/prisma/client";

vi.mock("../lib/prisma", () => ({ prisma: mockDeep<PrismaClient>() }));

import { prisma } from "../lib/prisma"; // now the mock
```

It works, but passing the client in is usually easier to read and doesn't depend on module-mocking rules.

What mocks can't tell you

A MOCK ONLY KNOWS WHAT YOU TOLD IT

The mocked `create` returns whatever you configured. It won't reject a duplicate slug, won't enforce a foreign key, won't apply a default, and won't notice a typo in a `where` clause that the real database would treat very differently. A test can pass against the mock while the real query fails. Mock tests check *your logic*; they don't check *your queries*.

Approach 2: Integration Tests Against a Real Database

Prisma's documentation recommends running a separate database, usually in Docker, with its own connection string, and applying migrations before the tests run.

```
# docker-compose.test.yml
services:
  db-test:
    image: postgres:17
    environment:
      POSTGRES_USER: test
      POSTGRES_PASSWORD: test
```

```

    POSTGRES_DB: blog_test
  ports:
    - "5433:5432"      # a different port from the development database

```

```

# .env.test
DATABASE_URL="postgresql://test:test@localhost:5433/blog_test"

```

```

// package.json
"scripts": {
  "test:unit": "vitest run test/*.unit.test.ts",
  "test:int": "docker compose -f docker-compose.test.yml up -d && dotenv -e .env.test
}

```

`dotenv -e .env.test` (from the `dotenv-cli` package) makes every step use the test database. `migrate deploy` applies existing migrations without generating new ones, exactly as in production. `--no-file-parallelism` runs test files one after another so they don't fight over the same tables.

MAKE SURE YOU'RE POINTING AT THE TEST DATABASE

Integration tests delete data. Add a check at the top of your setup file — for example, refuse to run unless `DATABASE_URL` contains `_test` — so a mistake in your environment can never wipe your development data.

A clean slate for every test

Each test should start from a known state. Deleting table by table (in child-to-parent order, as in Chapter 4) works for small schemas; Prisma's docs suggest truncating every table for anything larger:

```

// test/reset.ts (PostgreSQL)
import { prisma } from "../lib/prisma";

export async function resetDatabase() {
  if (!process.env.DATABASE_URL?.includes("_test")) throw new Error("Not a test database");

  const tables = await prisma.$queryRaw<{ tablename: string }[]>`

```

```

SELECT tablename FROM pg_tables
WHERE schemaname = 'public' AND tablename <> '_prisma_migrations'
`;
const list = tables.map((t) => `${t.tablename}`).join(", ");
await prisma.$executeRawUnsafe(`TRUNCATE TABLE ${list} RESTART IDENTITY CASCADE`);
}

```

This uses `$executeRawUnsafe` from Chapter 3 — acceptable here because the table names come from the database's own catalogue, never from user input. `CASCADE` removes the need for a deletion order, `RESTART IDENTITY` resets auto-increment IDs, and `_prisma_migrations` is kept so the migration history survives.

```

// test/posts.int.test.ts
import { describe, it, expect, beforeEach, afterAll } from "vitest";
import { prisma } from "../lib/prisma";
import { createPost } from "../src/posts";
import { resetDatabase } from "../reset";

beforeEach(resetDatabase);
afterAll(() => prisma.$disconnect());

it("refuses a second post with the same slug", async () => {
  const author = await prisma.user.create({ data: { email: "a@test.local", name: "A" } });
  const input = { title: "Hello World", content: "...", authorId: author.id };

  await createPost(prisma, input);
  await expect(createPost(prisma, input)).rejects.toMatchObject({ code: "P2002" });
});

```

This is exactly the bug a mock can't find: `createPost` has no handling for duplicate slugs, and only a real database with a real `@unique` constraint reveals it.

Choosing Between Them

	MOCKED CLIENT	REAL TEST DATABASE
Speed	Milliseconds	Seconds

	MOCKED CLIENT	REAL TEST DATABASE
Setup	A library	Docker, a second database, migrations
Checks	Your branching, validation, which queries you call	That the queries really work: constraints, relations, defaults, transactions
Risk	Passing tests for broken queries	Slow or flaky tests if isolation is sloppy

A PRACTICAL SPLIT

Test pure logic with plain unit tests, use mocks for code with lots of branches, and give every query that matters — especially anything with transactions, constraints or raw SQL — at least one integration test against a real database.

Hands-On Exercises

EXERCISE 1

Using a mocked client, test Chapter 1's `savePost` (optimistic concurrency): one test where `updateMany` reports `count: 1`, and one where it reports `count: 0` and the function throws.

 [View solution](#)

EXERCISE 2

Fix `createPost` so a duplicate slug gets a numeric suffix (`hello-world-2`), and prove it with an integration test. Explain why the mocked test from the chapter would still pass on the unfixed version.

 [View solution](#)

EXERCISE 3

Write integration tests for Chapter 1's `deleteUserKeepingPosts`: prove that a plain `user.delete` on a user with posts fails with `P2003`, and that the function succeeds — moving the posts to the placeholder account and removing the user's profile along with them. Why couldn't a mock test catch either behaviour?

[View solution](#)

CHAPTER 5 QUICK REFERENCE

- Pass the client in as a parameter (dependency injection) to make code easy to test
- Mocks: `mockDeep<PrismaClient>()`, `mockReset` in `beforeEach`, `mockResolvedValue`, `toHaveBeenCalledWith`
- Mocks test your logic, not your queries — no constraints, relations or defaults
- Integration: a separate Docker database, `.env.test`, `migrate deploy`, then tests
- Reset between tests with `TRUNCATE ... RESTART IDENTITY CASCADE`, keeping `_prisma_migrations`
- Guard against running destructive setup on a non-test database; run test files one at a time

CHAPTER 6: CLIENT EXTENSIONS

Prisma Intermediate/Advanced

Chapter 6 · Client Extensions

Some behaviour belongs everywhere: logging slow queries, hiding soft-deleted posts, adding a computed `url` field. Copying that into every route invites mistakes. **Client extensions** let you build it into Prisma Client itself, with `$extends`.

MIDDLEWARE IS GONE IN PRISMA 7

Older tutorials use `prisma.$use(...)` middleware. It was deprecated in Prisma 5 and removed in Prisma 7. The `query` component of `$extends`, covered below, replaces it.

How \$extends Works

```
const base = new PrismaClient({ adapter });
const prisma = base.$extends({ /* ... */ });
```

`$extends` returns a *new* client and leaves the original untouched. You can chain several extensions — `base.$extends(a).$extends(b)` — and if two define the same thing, the later one wins. An extension object can have four parts:

COMPONENT	ADDS OR CHANGES	EXAMPLE
query	What happens when a query runs	Timing, filtering, default arguments
model	New methods on a model	<code>prisma.post.findPublishedBySlug()</code>
result	Computed fields on returned records	<code>post.url</code>
client	New top-level methods	<code>prisma.\$healthCheck()</code>

query: Wrapping Every Query

```
const prisma = base.$extends({
  query: {
    $allModels: {
      async $allOperations({ model, operation, args, query }) {
        const start = performance.now();
        const result = await query(args); // run the real query
        const ms = performance.now() - start;
        if (ms > 200) console.warn(`Slow query: ${model}.${operation} took ${ms.toFixed(2)}ms`);
        return result;
      },
    },
  },
});
```

- `query(args)` runs the original operation. Forget to call it, or to return its result, and the query silently does nothing useful.
- Target one model and operation (`post: { findMany(...) }`), all operations on a model (`post: { $allOperations(...) }`), or everything (`$allModels`).
- Raw queries have their own top-level hooks: `$queryRaw`, `$executeRaw` and the `Unsafe` variants.

Changing arguments: a soft-delete filter

Soft deletion marks a row as deleted instead of removing it, so it can be restored. Add `deletedAt DateTime?` to `Post` and migrate, then hide deleted posts by default:

```
const prisma = base.$extends({
  query: {
    post: {
      async findMany({ args, query }) {
        args.where = { ...args.where, deletedAt: null };
        return query(args);
      },
      async findFirst({ args, query }) {
        args.where = { ...args.where, deletedAt: null };
        return query(args);
      },
      async count({ args, query }) {
        args.where = { ...args.where, deletedAt: null };
        return query(args);
      },
    },
  },
});
```

```

    },
  },
},
});

```

SOFT DELETE BY EXTENSION HAS REAL GAPS

- **Nested reads aren't covered.** Prisma's docs state that query extensions don't apply to nested read and write operations. `prisma.user.findMany({ include: { posts: true } })` still returns deleted posts.
- **Every operation needs thought.** `findUnique`, `aggregate`, `groupBy`, raw SQL and updates each behave differently; the example above covers only three.
- **You can't change `select` or `include`** in a query extension, because that would change the result's type.

For data that must never leak, filter explicitly, use a database view, or use row-level security in the database.

model: Adding Methods

```

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

const prisma = base.$extends({
  model: {
    post: {
      async findPublishedBySlug(slug: string) {
        return base.post.findFirst({ where: { slug, published: true, deletedAt: null } });
      },
      async softDelete(id: number) {
        return base.post.update({ where: { id }, data: { deletedAt: new Date() } });
      },
    },
  },
  $allModels: {
    async exists<T>(this: T, where: Prisma.Args<T, "findFirst">["where"]) {
      const ctx = Prisma.getExtensionContext(this);
      const found = await (ctx as any).findFirst({ where, select: { id: true } });
      return found !== null;
    },
  },
},
});

```

```
});

await prisma.post.softDelete(12);
await prisma.user.exists({ email: "ada@example.com" }); // true / false, on every mod
```

Model methods keep business rules in one place: routes call `findPublishedBySlug` instead of remembering all three conditions. The `$allModels` version uses `Prisma.getExtensionContext(this)` to find out which model it was called on.

result: Computed Fields

```
const prisma = base.$extends({
  result: {
    post: {
      url: {
        needs: { slug: true },
        compute(post) {
          return `/posts/${post.slug}`;
        },
      },
    },
    readingMinutes: {
      needs: { content: true },
      compute(post) {
        const words = (post.content ?? "").split(/\s+/).filter(Boolean).length;
        return Math.max(1, Math.round(words / 200));
      },
    },
  },
});

const post = await prisma.post.findFirst({ select: { title: true, url: true } });
// { title: "Hello", url: "/posts/hello" }
```

`needs` lists the real fields the computation uses, so Prisma fetches them even when you only `select` the computed field. Computed fields are typed, can be selected like real ones, and are calculated only when read. They can't be used in `where` or `orderBy` — the database doesn't know they exist.

client: Top-Level Methods

```
const prisma = base.$extends({
  client: {
    async $healthCheck() {
      await base.$queryRaw`SELECT 1`;
      return { ok: true, at: new Date().toISOString() };
    },
  },
});

app.get("/health", async (req, res) => res.json(await prisma.$healthCheck()));
```

Packaging and Sharing Extensions

```
// lib/extensions/slowQueryLog.ts
export const slowQueryLog = (thresholdMs = 200) =>
  Prisma.defineExtension({
    name: "slowQueryLog",
    query: { $allModels: { async $allOperations({ model, operation, args, query }) { /*
    */
  }});

// lib/prisma.ts
const base = new PrismaClient({ adapter });
export const prisma = base
  .$extends(slowQueryLog(200))
  .$extends(postHelpers)
  .$extends(postComputedFields);

export type AppPrisma = typeof prisma;
```

UPDATE YOUR TYPES AFTER EXTENDING

An extended client has a different type from `PrismaClient`. Functions that receive the client (as in Chapter 5) should use `AppPrisma`; otherwise TypeScript won't know about `softDelete` or `url`. In unit tests, you can still mock it with `mockDeep<AppPrisma>()`.

Hands-On Exercises

EXERCISE 1

Write a `query` extension that makes `post.findMany` return at most 50 rows unless the caller passes `take`, and never more than 100 even if they do.

 [View solution](#)

EXERCISE 2

Add a `result` extension giving users a `displayName` (their name, or the part of their email before the `@` if they have none), and a `model` method `user.findByEmail` that ignores letter case in the email.

 [View solution](#)

EXERCISE 3

Using the soft-delete extension from this chapter, write an integration test (Chapter 5 style) that shows exactly which queries hide a soft-deleted post and which still return it. Explain what you'd change before relying on it.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- `client.$extends({...})` returns a new client; the original is unchanged; chain several; later wins
- `query`: wrap operations; always `return query(args)`; `$allModels` / `$allOperations`; replaces removed `$use` middleware
- Query extensions don't apply to nested reads/writes and can't change `select` / `include`
- `model`: custom methods; `Prisma.getExtensionContext(this)` in `$allModels`
- `result`: computed fields with `needs` and `compute`; not usable in `where` / `orderBy`
- `client`: top-level `$methods`; package reusable ones with `Prisma.defineExtension`

- Export `type AppPrisma = typeof prisma` and use it wherever the client is passed around

CHAPTER 7: PERFORMANCE

Prisma Intermediate/Advanced

Chapter 7 · Performance

Prisma makes queries easy to write, which also makes slow ones easy to write. Almost every Prisma performance problem is one of a few familiar mistakes: too many queries, too much data, a missing index, or too few (or too many) database connections. This chapter shows how to spot each one and what to do about it. The rule throughout: **measure first**, then fix what the measurements show.

Step 1: See the SQL

```
const base = new PrismaClient({
  adapter,
  log: [{ emit: "event", level: "query" }, "warn", "error"],
});

base.$on("query", (e) => {
  console.log(`${e.duration} ms  ${e.query}`);
});
```

Every SQL statement Prisma sends is now printed with its duration. Chapter 6's slow-query extension times whole Prisma operations; this log shows the individual statements underneath, which is what you need to spot the problems below. Keep it on in development and off (or sampled) in production, because it's noisy and the `params` field can contain personal data.

Problem 1: Too Many Queries (N+1)

```
// SLOW: 1 query for posts + 1 query per post for its author
const posts = await prisma.post.findMany({ take: 50 });
for (const p of posts) {
  const author = await prisma.user.findUnique({ where: { id: p.authorId } });
  console.log(p.title, author?.name);
}
```

```
// FAST: 2 queries in total, however many posts there are
const posts2 = await prisma.post.findMany({
  take: 50,
  include: { author: { select: { name: true } } },
});
```

The first version sends 51 queries for 50 posts. In the query log it's unmistakable: the same statement repeated over and over with a different ID. The fix is to load related data with `include` or a nested `select`, as in Prisma Fundamentals, Chapter 9.

Joins instead of extra queries

By default, `include` fetches each level of relations with a separate query using `WHERE id IN (...)`. On PostgreSQL and MySQL, the `relationJoins` preview feature can load them in a single query with database joins instead:

```
// schema.prisma: previewFeatures = ["relationJoins"]
await prisma.post.findMany({
  relationLoadStrategy: "join", // or "query" for the default behaviour
  include: { author: true, tags: true },
});
```

Neither strategy is always faster — one round trip versus several simpler queries. If a page is slow and loads several levels of relations, try both and compare the timings in your log.

Problem 2: Too Much Data

HABIT	COST	BETTER
<code>findMany()</code> with no <code>select</code>	Every column, including long <code>content</code> text	<code>select</code> only the fields the page shows
<code>include: { posts: true }</code>	Every related row, unbounded	Add <code>take</code> , <code>where</code> and <code>orderBy</code> inside the include
No take on a list endpoint	The whole table as it grows	Always paginate

HABIT	COST	BETTER
Large skip values	The database still reads and discards skipped rows	Cursor pagination for deep pages
Counting by fetching	Rows sent just to call <code>.length</code>	<code>count()</code> or relation <code>_count</code> (Chapter 2)

```
// A list page needs a summary, not the article bodies
const page = await prisma.post.findMany({
  where: { published: true },
  orderBy: { id: "desc" },
  take: 20,
  ...(cursor ? { cursor: { id: cursor }, skip: 1 } : {}),
  select: { id: true, title: true, slug: true, author: { select: { name: true } } },
});
```

Problem 3: Missing Indexes

An index lets the database find matching rows without reading the whole table. Prisma creates indexes for `@id` and `@unique` fields. For other columns you filter or sort on often, add one:

```
model Post {
  // ...
  @@index([authorId]) // already in the blog schema
  @@index([published, createdAt(sort: Desc)]) // "latest published posts"
}
```

Then run `prisma migrate dev` to create it. To check whether a query actually uses an index, ask the database directly with `EXPLAIN ANALYZE` (PostgreSQL), using Chapter 3's raw SQL:

```
const plan = await prisma.$queryRaw`
  EXPLAIN ANALYZE
  SELECT "id", "title" FROM "Post"
  WHERE "published" = true ORDER BY "createdAt" DESC LIMIT 20
`;
console.log(plan);
```

```
// "Seq Scan on Post" -> reading every row: consider an index
// "Index Scan using ..." -> the index is being used
```

INDEXES AREN'T FREE

Every index is updated on every insert and update, and takes disk space. Add indexes for queries you've measured as slow and that run often, not for every column "just in case". On a small table, a full scan is often faster anyway, so test with realistic data volumes — Chapter 4's seed scripts help here.

Problem 4: Connections

Each `PrismaClient` keeps a **pool** of open database connections and reuses them. In Prisma 7, the pool belongs to the database driver behind your adapter, and so do its settings:

```
import { PrismaPg } from "@prisma/adapter-pg";

const adapter = new PrismaPg({
  connectionString: process.env.DATABASE_URL,
  max: 10, // pool size (pg's default is 10)
  connectionTimeoutMillis: 5_000, // pg's default is 0: wait forever
  idleTimeoutMillis: 300_000,
});
```

PRISMA 6 URL SETTINGS NO LONGER APPLY

Tutorials that add `?connection_limit=5&pool_timeout=10` to the database URL describe Prisma 6's built-in pool. With driver adapters, set pool options on the adapter instead. Prisma's upgrade guide also notes that `pg` has no connection timeout by default, where Prisma 6 used 5 seconds; the values above restore Prisma 6's timeouts.

The most common connection mistake

```
// WRONG: a new client, and a new pool, on every request
app.get("/posts", async (req, res) => {
  const prisma = new PrismaClient({ adapter: new PrismaPg({ connectionString }) });
```

```
res.json(await prisma.post.findMany());
});
```

This opens fresh connections for every request and quickly exhausts the database's connection limit. Create one client when the app starts and share it — exactly what `lib/prisma.ts` from Prisma Fundamentals does. Pool size is a balance: too small and requests queue waiting for a connection; too large, multiplied by every running copy of your app, and you exceed what the database allows. Serverless platforms make this harder, which Chapter 9 covers.

Problem 5: Doing Work One Row at a Time

INSTEAD OF	USE
A loop of <code>create</code> calls	<code>createMany</code> (Chapter 4)
A loop of <code>update</code> calls with the same change	<code>updateMany</code> with a <code>where</code>
Read a value, add 1, write it back	<code>{ increment: 1 }</code> , one atomic statement
Independent queries awaited one after another	<code>Promise.all</code> or <code>\$transaction(...)</code>
Complex bulk changes Prisma can't express	One raw SQL statement (Chapter 3)

```
// Three independent reads: run them together, not one after another
const [latest, popular, tagCount] = await Promise.all([
  prisma.post.findMany({ orderBy: { createdAt: "desc" }, take: 5, select: { title: true } },
  prisma.post.findMany({ orderBy: { viewCount: "desc" }, take: 5, select: { title: true } },
  prisma.tag.count(),
]);
```

A Performance Checklist

1. Turn on the query log and reproduce the slow request.
2. Many near-identical statements? Fix the N+1 with `include` / `select`.
3. Large rows or unbounded lists? Add `select`, `take` and pagination.
4. One slow statement? `EXPLAIN ANALYZE` it and consider an index.

5. Fast queries but slow requests under load? Look at the pool size and timeouts.

6. Measure again to confirm the fix actually helped.

Hands-On Exercises

EXERCISE 1

An endpoint returns each tag with its three most-viewed published posts, but it loops over tags and runs a query per tag. Use the query log to count its queries, then rewrite it to use a fixed number of queries.

 [View solution](#)

EXERCISE 2

Seed 100,000 posts, then use `EXPLAIN ANALYZE` to compare "latest 20 published posts" before and after adding `@@index([published, createdAt(sort: Desc)])`. Describe what changed in the plan.

 [View solution](#)

EXERCISE 3

Your API runs as 4 copies, each with a pool of 10, and the database allows 50 connections. Is that safe? What changes when you add a migration job, an admin script and a fifth copy? Configure the adapter accordingly.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- Measure first: `log: [{ emit: "event", level: "query" }]` and `$on("query", e => e.duration)`
- N+1: repeated identical statements; fix with `include` / nested `select`
- `relationLoadStrategy: "join"` (preview `relationJoins`) vs. the default `"query"` — measure both
- Fetch less: `select`, bounded includes, `take`, cursor pagination, `count()`
- `@@index` for frequent filters/sorts; confirm with `EXPLAIN ANALYZE`

- Prisma 7 pools live in the driver: set `max` and timeouts on the adapter; one shared client per app
- Batch: `createMany`, `updateMany`, `increment`, `Promise.all`

CHAPTER 8: SCHEMA EVOLUTION IN PRODUCTION

Prisma Intermediate/Advanced

Chapter 8 · Schema Evolution in Production

In development, a bad migration costs a `migrate reset`. In production, the database holds real data and the app is running while you change it. This chapter covers changing a live schema safely: which changes are risky, how to rename without losing data, how to change a column's shape without downtime, how to move data, how migrations run in CI, and how to bring an existing database under Prisma Migrate.

Development vs. Production Commands

COMMAND	WHERE	WHAT IT DOES
<code>migrate dev</code>	Development only	Creates new migrations from schema changes and applies them; may offer to reset the database
<code>migrate dev --create-only</code>	Development	Writes the migration SQL without applying it, so you can edit it first
<code>migrate deploy</code>	CI, staging, production	Applies pending migrations from the <code>migrations</code> folder; never creates new ones or resets
<code>migrate status</code>	Anywhere	Shows which migrations have and haven't been applied
<code>migrate resolve</code>	Recovery	Marks a migration as applied or rolled back after you fix a problem by hand

NEVER RUN MIGRATE DEV AGAINST PRODUCTION

`migrate dev` is built for throwaway databases. If it detects drift, it offers to reset — which deletes everything. Production only ever gets `migrate deploy`, running migration files that have already been reviewed and tested.

Which Changes Are Risky?

CHANGE	RISK	SAFE APPROACH
Add a table, or an optional column	Low	Just migrate
Add a required column	Fails on existing rows unless it has a default	Add with <code>@default</code> , or optional → backfill → required
Add an index	Can lock a big table while building	On PostgreSQL, edit the SQL to <code>CREATE INDEX CONCURRENTLY</code> for large tables
Rename a column or table	Prisma generates drop + add: data loss	Edit the SQL to a rename, or use <code>@map</code>
Change a column's type or meaning	Old code breaks during deploy	Expand and contract
Drop a column or table	Irreversible; running code may still use it	Stop using it first, deploy, then drop

Renaming Without Losing Data

Rename `viewCount` to `views` in the schema and Prisma sees one field disappear and another appear. The migration it writes drops the old column — and every view count with it — and adds an empty new one. The fix, from Prisma's own docs, is to create the migration without applying it and edit the SQL:

```
npx prisma migrate dev --name rename-viewcount --create-only
```

```
-- Generated (destroys data):
-- ALTER TABLE "Post" DROP COLUMN "viewCount",
-- ADD COLUMN "views" INTEGER NOT NULL DEFAULT 0;

-- Edited (keeps data):
ALTER TABLE "Post" RENAME COLUMN "viewCount" TO "views";
```

```
npx prisma migrate dev # applies the edited migration
```

OR DON'T RENAME THE COLUMN AT ALL

If you only want a nicer name in your code, keep the database column and map to it: `views Int @default(0) @map("viewCount")`. The TypeScript field is `views`; the column is untouched and no migration SQL is needed. `@@map` does the same for table names.

A RENAME STILL BREAKS RUNNING CODE

Even a correct rename is a problem mid-deploy: old copies of the app, still running, query `viewCount`, which no longer exists. For a live app with no downtime, use expand and contract instead.

Expand and Contract

The core technique for zero-downtime changes: never make a change that the currently running code can't handle. Instead, go in small steps, each one deployable on its own. Example: replacing `published Boolean` with a richer `status` (`DRAFT`, `PUBLISHED`, `ARCHIVED`).

STEP	SCHEMA	CODE
1. Expand	Add <code>status PostStatus?</code> (optional)	Unchanged
2. Dual write	—	Every write sets both <code>published</code> and <code>status</code>
3. Backfill	A data migration fills <code>status</code> for old rows	—
4. Switch reads	Make <code>status</code> required	Read <code>status</code> ; still write both
5. Stop old writes	—	Only write <code>status</code>
6. Contract	Drop <code>published</code>	—

It's slower than one big change, and each step is a separate deploy. In exchange, at every moment both the old and the new version of the app work against the database, and any step can be paused or rolled back. The capstone in Chapter 10 walks through this exact change.

Data Migrations

Step 3 above changes *data*, not structure. Prisma Migrate only generates schema changes, but a migration is just SQL, so you can write the data change yourself in an empty migration:

```
npx prisma migrate dev --name backfill-post-status --create-only
```

```
-- prisma/migrations/2026..._backfill-post-status/migration.sql
UPDATE "Post"
SET "status" = CASE WHEN "published" THEN 'PUBLISHED'::"PostStatus"
                    ELSE 'DRAFT'::"PostStatus" END
WHERE "status" IS NULL;
```

- Write data migrations so they're safe to re-run (`WHERE "status" IS NULL`).
- On a very large table, one `UPDATE` can lock rows for a long time. Backfill in batches from a script (using Chapter 3's raw SQL or `updateMany` with a `take`-style ID range) instead of one migration.
- Test the migration on a copy of production data before running it for real.

Migrations in CI and Deployment

```
# In CI, on every pull request
npx prisma validate                                # schema is valid
npx prisma migrate diff \
  --from-migrations prisma/migrations \
  --to-schema prisma/schema.prisma \
  --exit-code                                       # fails if the schema has changes with no i

# At deploy time, before starting the new app version
npx prisma migrate deploy
```

The `migrate diff` check catches the classic mistake of editing `schema.prisma` and forgetting to commit a migration. `migrate deploy` should run once per deployment, from one place — a

release step or job — not from every copy of the app as it starts up, where several could try to migrate at once.

COMMIT MIGRATIONS; NEVER EDIT APPLIED ONES

The `prisma/migrations` folder is part of your code. Once a migration has run anywhere shared, treat it as permanent. Prisma records a checksum of each applied migration, and editing one afterwards causes errors about modified migrations. To fix a mistake, write a new migration.

Adopting Prisma on an Existing Database

Joining a project with a database that was never managed by Prisma takes two steps: generate a schema from it, then tell Migrate that this is the starting point.

1. Introspect with db pull

```
npx prisma db pull
```

Prisma reads the tables, columns, keys and indexes and writes matching models into `schema.prisma`. Tidy the result — rename models and fields to your conventions using `@map` and `@@map`, so the database itself doesn't change.

2. Baseline

Migrate would otherwise try to create every table again. A **baseline** migration describes the existing database and is marked as already applied:

```
mkdir -p prisma/migrations/0_init

npx prisma migrate diff \
  --from-empty \
  --to-schema prisma/schema.prisma \
  --script > prisma/migrations/0_init/migration.sql

npx prisma migrate resolve --applied 0_init # run against each existing database
```

From then on, new schema changes become ordinary migrations on top of `0_init`. A brand-new database (a new developer, or a test database) gets `0_init` applied for real, so every environment ends up with the same structure.

Hands-On Exercises

EXERCISE 1

Rename `Post.content` to `body` twice: once with an edited migration, once with `@map`. Show the SQL for each, and say which you'd choose for an app with several copies running.

 [View solution](#)

EXERCISE 2

Add a required `excerpt String` to `Post` on a database with existing posts, without downtime and without a meaningless default. Plan the migrations and deploys, with the SQL.

 [View solution](#)

EXERCISE 3

Write a CI workflow (GitHub Actions or similar) that fails a pull request with a missing migration, and a deploy step that applies migrations exactly once before the new version starts.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- `migrate dev` in development only; `migrate deploy` everywhere else
- Renames: `--create-only`, edit to `RENAME COLUMN`; or keep the column and use `@map` / `@@map`
- Expand → dual write → backfill → switch reads → stop old writes → contract
- Data migrations: SQL in an empty `--create-only` migration; re-runnable; batch big tables
- CI: `migrate diff --from-migrations ... --to-schema ... --exit-code`; deploy: `migrate deploy` once

- Never edit an applied migration — write a new one
- Existing database: `db pull`, then baseline with `migrate diff --from-empty` and `migrate resolve -`
`-applied 0_init`

CHAPTER 9: DEPLOYING PRISMA

Prisma Intermediate/Advanced

Chapter 9 · Deploying Prisma

Everything so far has run on a laptop. Deploying a Prisma app raises a few specific questions: when the client gets generated, how the running app and the migration step find the database, how many connections each environment opens, and what changes on serverless and edge platforms. This chapter works through them, then takes an honest look at Prisma's own hosted products.

Environments and Their Settings

ENVIRONMENT	DATABASE	MIGRATIONS RUN BY
Development	Local (often Docker)	You, with <code>migrate dev</code>
Test / CI	Throwaway (Chapter 5)	CI, with <code>migrate deploy</code>
Staging	Production-like copy	The deploy pipeline
Production	Real data	The deploy pipeline, once per release (Chapter 8)

Keep connection strings in environment variables set by the platform, never in the repository. Commit a `.env.example` listing the variable names with placeholder values, and add `.env` to `.gitignore`.

Two connection strings

In Prisma 7 the running app and the CLI connect separately: the app through its driver adapter, the CLI through the `datasource.url` in the config file. That makes it natural to give each its own URL — useful as soon as a connection pooler sits in front of the database, because migrations need a *direct* connection:

```
# .env.example
DATABASE_URL="postgresql://app:***@pooler.example.com:6432/blog" # app traffic, via t
DIRECT_URL="postgresql://app:***@db.example.com:5432/blog" # migrations, straight
```

```
// prisma config file: used by the CLI (migrate deploy, db pull...)
datasource: { url: env("DIRECT_URL") },

// lib/prisma.ts: used by the running app
const adapter = new PrismaPg({ connectionString: process.env.DATABASE_URL });
```

Building: Generate the Client Explicitly

Since Prisma Fundamentals, the client has been generated into your own folder (`generated/prisma`). That folder is build output: add it to `.gitignore` and generate it during every build. Don't rely on it appearing as a side effect of installing packages or running migrations — in Prisma 7, migrate commands don't generate it for you.

```
// package.json
"scripts": {
  "build": "prisma generate && tsc",
  "release": "prisma migrate deploy",
  "start": "node dist/server.js"
}
```

NO MORE ENGINE BINARIES

Older Prisma versions shipped a Rust "query engine" binary that had to match the server's operating system, leading to `binaryTargets` settings and "engine not found" errors in Docker. Prisma 7's `prisma-client` generator is written in TypeScript and talks to the database through your driver adapter, so that whole class of deployment problem is gone.

A Docker image

```
# Dockerfile
FROM node:22-slim AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build # prisma generate + tsc
```

```
FROM node:22-slim
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY --from=build /app/dist ./dist
COPY --from=build /app/generated ./generated
COPY prisma ./prisma
COPY prisma.config.ts ./
CMD ["npm", "start"]
```

The migrations folder and config are copied so the same image can run the `release` step. Note that `npm ci --omit=dev` skips dev dependencies: if `prisma` (the CLI) is a dev dependency, run migrations from the build image or a separate job instead. Adjust `prisma.config.ts` if your project uses the `prisma7.config.ts` name.

Shutting down cleanly

```
const server = app.listen(port);

process.on("SIGTERM", () => {
  server.close(async () => { // stop taking requests, finish current ones
    await prisma.$disconnect(); // then release database connections
    process.exit(0);
  });
});
```

Platforms send `SIGTERM` before stopping a container. Disconnecting *once*, at shutdown, is good practice; disconnecting after each request is not — Prisma's docs warn it slows every request down.

Long-Running Servers vs. Serverless

	LONG-RUNNING SERVER	SERVERLESS FUNCTIONS
Client instances	One per process, for its whole life	One per function instance; many can exist at once
Where to create the client	At startup (<code>lib/prisma.ts</code>)	Outside the handler, so warm instances reuse it

	LONG-RUNNING SERVER	SERVERLESS FUNCTIONS
Pool size	Chapter 7's budget	Start at 1 per instance without an external pooler
Main risk	Too many processes × pool size	A traffic spike creates hundreds of instances and exhausts the database
Usual fix	Tune pool sizes	An external pooler (PgBouncer or a managed equivalent)

```
// A serverless handler
import { PrismaClient } from "../generated/prisma/client";
import { PrismaPg } from "@prisma/adapter-pg";

// Created once per instance, reused while the instance stays warm
const prisma = new PrismaClient({
  adapter: new PrismaPg({ connectionString: process.env.DATABASE_URL, max: 1 }),
});

export async function handler() {
  const posts = await prisma.post.findMany({ where: { published: true }, take: 10 });
  return { statusCode: 200, body: JSON.stringify(posts) };
}
```

THE DEVELOPMENT HOT-RELOAD PROBLEM

Frameworks that reload code on every save (Next.js in development, for example) can create a new `PrismaClient` on each reload and leak connections. Prisma's documented fix is to keep the client on `globalThis` in development, since global variables survive reloads.

Edge Runtimes

Edge platforms such as Cloudflare Workers and Vercel Edge Functions run JavaScript close to users, but they aren't Node.js and often can't open ordinary TCP database connections. What works depends on the driver:

DRIVER	HOW IT CONNECTS	WORKS ON
Neon serverless, PlanetScale serverless	HTTP	Cloudflare Workers and Vercel Edge
libSQL / Turso	HTTP	Cloudflare Workers and Vercel Edge
Cloudflare D1	Cloudflare binding	Cloudflare Workers only
node-postgres (<code>pg</code>)	TCP via Cloudflare's <code>connect()</code>	Cloudflare Workers only, not Vercel Edge

Because Prisma 7 always uses a driver adapter, running at the edge is mostly a matter of choosing an edge-compatible driver and its adapter. Check your platform's and Prisma's current docs before committing: this area changes often.

THE EDGE ISN'T AUTOMATICALLY FASTER

A function running next to the user but talking to a database on another continent pays that distance on every query. For a database-heavy page, running near the *database* is usually faster. Edge makes most sense for work that needs little or no database access, or with a globally distributed or cached data layer.

Prisma's Hosted Products: an Honest Look

Prisma (the company) sells hosted services alongside the free, open-source ORM. You don't need any of them to use Prisma ORM.

PRODUCT	WHAT IT IS	CONSIDER IT WHEN	TRADE-OFFS
Prisma Postgres	A managed PostgreSQL database	You want a database with pooling built in and little setup	Compare price, region and backup options with other managed Postgres providers
Prisma Accelerate	A hosted connection pool with optional query caching	Serverless or edge apps exhausting connections; read-heavy pages	An extra network hop and dependency; cached data can be stale

Weigh them the way you would any vendor: what problem it solves for *your* app, what it costs at your traffic, and how hard it would be to leave. A self-managed PgBouncer, or your cloud

provider's own pooler, solves the connection problem without tying you to one vendor.

Product names, features and pricing change — check Prisma's current documentation rather than relying on this summary.

Hands-On Exercises

EXERCISE 1

Set up the blog API's configuration for two URLs (pooled and direct), a `.env.example`, `.gitignore` entries and build/release/start scripts. Explain which process uses which URL.

 [View solution](#)

EXERCISE 2

A serverless version of the API works in testing but fails with "too many connections" during a traffic spike. Diagnose it and list the fixes in order of preference.

 [View solution](#)

EXERCISE 3

Write a short decision note choosing where to deploy the blog API (a single container, serverless, or edge) for a site with readers mostly in one country and a PostgreSQL database in the same region. Justify it using this chapter.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- Secrets in environment variables; commit `.env.example`, ignore `.env` and `generated/`
- App uses the adapter's URL (pooled); CLI uses the config's `datasource.url` (direct)
- Build runs `prisma generate`; release runs `migrate deploy` once; start runs the app
- Prisma 7: no engine binaries, no `binaryTargets` headaches
- `$disconnect()` once on `SIGTERM`, never per request

- Serverless: client outside the handler, pool of 1, external pooler for spikes
- Edge: pick an edge-compatible driver + adapter; mind the distance to the database
- Hosted products (Prisma Postgres, Accelerate) are optional; compare them like any vendor

CHAPTER 10: CAPSTONE — EVOLVING A LIVE SCHEMA WITHOUT DOWNTIME

Prisma Intermediate/Advanced

Chapter 10 · Capstone: Evolving a Live Schema Without Downtime

The blog API from Prisma Fundamentals is live. Posts are either published or not, stored in a `published Boolean`. The editors now want a third state, **archived**: hidden from the front page but still reachable by link. The job is to replace `published` with a `status` field — `DRAFT`, `PUBLISHED` or `ARCHIVED` — while the site keeps running, several copies of the API are serving traffic, and no reader sees an error.

This capstone follows Chapter 8's expand-and-contract plan, and uses something from every chapter along the way.

RELEASE	SCHEMA	CODE	CHAPTERS USED
1. Expand	Add optional <code>status</code> , plus an index	Unchanged	7, 8, 9
2. Dual write	—	Writes set both fields, in one place	1, 5, 6
3. Backfill	Data migration fills old rows	—	3, 8
4. Switch reads	<code>status</code> becomes required	Reads use <code>status</code> ; new stats	2, 4, 5
5. Stop old writes	—	Only <code>status</code> written	6
6. Contract	Drop <code>published</code>	—	8, 9

Release 1: Expand

```
// schema.prisma
enum PostStatus {
  DRAFT
  PUBLISHED
  ARCHIVED
}
```

```

model Post {
  // ...existing fields, including published Boolean @default(false)
  status PostStatus? // optional for now

  @@index([authorId])
  @@index([status, createdAt(sort: Desc)]) // the front page will filter on status (C
}

```

```
npx prisma migrate dev --name add-post-status
```

```

-- The generated migration: purely additive
CREATE TYPE "PostStatus" AS ENUM ('DRAFT', 'PUBLISHED', 'ARCHIVED');
ALTER TABLE "Post" ADD COLUMN "status" "PostStatus";
CREATE INDEX "Post_status_createdAt_idx" ON "Post"("status", "createdAt" DESC);

```

Old code ignores the new column; nothing it does can fail. It ships through the pipeline from Chapters 8 and 9: CI checks the migration exists, the release step runs `migrate deploy` once, then the new version starts. On a large table, edit the index line to `CREATE INDEX CONCURRENTLY` before committing (Chapter 8).

Release 2: Dual Write

Every write must now keep the two fields in agreement. Scattering that across routes would be easy to get wrong, so the rule lives in one place: model methods from Chapter 6.

```

// lib/extensions/postStatus.ts
import { Prisma } from "../../generated/prisma/client";

export type Status = "DRAFT" | "PUBLISHED" | "ARCHIVED";

// During the transition, both representations are written together
export function statusFields(status: Status) {
  return { status, published: status === "PUBLISHED" };
}

```

```

export const postStatus = Prisma.defineExtension((client) =>
  client.$extends({
    name: "postStatus",
    model: {
      post: {
        async setStatus(id: number, status: Status, expectedVersion: number) {
          const result = await client.post.updateMany({
            where: { id, version: expectedVersion }, // Chapter
            data: { ...statusFields(status), version: { increment: 1 } },
          });
          if (result.count === 0) throw new Error("Post changed or not found");
        },
      },
    },
  }),
);

```

Routes that create or publish posts now call `statusFields(...)` or `setStatus(...)`. The optimistic-concurrency check from Chapter 1 stops two editors publishing and archiving the same post at once.

```

// test/postStatus.int.test.ts (Chapter 5)
it("keeps published and status in agreement", async () => {
  const author = await base.user.create({ data: { email: "a@test.local" } });
  const post = await base.post.create({
    data: { title: "T", slug: "t", authorId: author.id, ...statusFields("DRAFT") },
  });

  await prisma.post.setStatus(post.id, "PUBLISHED", post.version);
  expect(await base.post.findUnique({ where: { id: post.id } }))
    .toMatchObject({ status: "PUBLISHED", published: true, version: 1 });

  await prisma.post.setStatus(post.id, "ARCHIVED", 1);
  expect(await base.post.findUnique({ where: { id: post.id } }))
    .toMatchObject({ status: "ARCHIVED", published: false });
});

```

ARCHIVED POSTS DURING THE TRANSITION

Old code still reads `published`, and an archived post has `published: false`, so old copies of the app treat it as a draft — hidden. That's acceptable here, because archived posts should leave the

front page anyway. Always check what the *old* code will make of each new state: if it would have exposed something private, you'd hold back the new state until Release 4.

Release 3: Backfill

Posts written before Release 2 still have `status = NULL`. A data migration (Chapter 8) fills them in, using SQL (Chapter 3):

```
npx prisma migrate dev --name backfill-post-status --create-only
```

```
-- migration.sql
UPDATE "Post"
SET "status" = CASE WHEN "published" THEN 'PUBLISHED'::"PostStatus"
                    ELSE 'DRAFT'::"PostStatus" END
WHERE "status" IS NULL;
```

The `WHERE "status" IS NULL` makes it safe to re-run and leaves alone any post already written by Release 2. Before moving on, verify the result — this is the check that makes the next step safe:

```
const [missing, mismatched] = await prisma.$transaction([
  prisma.post.count({ where: { status: null } }),
  prisma.$queryRaw<{ n: number }[]>`
    SELECT COUNT(*)::int AS n FROM "Post"
    WHERE ("status" = 'PUBLISHED') <> "published"
  `,
]);
console.log({ missing, mismatched: mismatched[0].n }); // both must be 0
```

Release 4: Switch Reads

With every row filled in, `status` can become required:

```
status PostStatus @default(DRAFT) // no longer optional
```

```
-- Generated migration
ALTER TABLE "Post" ALTER COLUMN "status" SET NOT NULL,
ALTER COLUMN "status" SET DEFAULT 'DRAFT';
```

The default matters: old copies still running during this deploy create posts without setting `status`, and the default keeps those inserts working. Reads now switch over:

```
// Front page: published only (uses the new index)
prisma.post.findMany({
  where: { status: "PUBLISHED" },
  orderBy: { createdAt: "desc" },
  take: 20,
  select: { title: true, slug: true, url: true }, // url from Chapter 6's result ext
});

// A single post: published or archived are both reachable by link
prisma.post.findFirst({ where: { slug, status: { in: ["PUBLISHED", "ARCHIVED"] } } });

// Stats (Chapter 2): posts per status in one query
prisma.post.groupBy({ by: ["status"], _count: { _all: true } });
```

Update the seed scripts (Chapter 4) to set `status` too, including a few archived posts, so development data exercises the new state.

Release 5: Stop Old Writes

Once no running copy reads `published`, `statusFields` stops writing it — one line, because the rule lived in one place:

```
export function statusFields(status: Status) {
  return { status };
}
```

Before shipping, search the whole codebase — including raw SQL strings, seed scripts, tests and admin scripts — for `published`. Raw SQL is invisible to TypeScript, so the compiler won't point it out.

Release 6: Contract

Finally, after at least one full release with nothing reading or writing the old column, remove it from the schema:

```
npx prisma migrate dev --name drop-post-published
```

```
ALTER TABLE "Post" DROP COLUMN "published";
```

THE ONLY STEP YOU CAN'T UNDO

Every earlier release could be rolled back by redeploying the previous version. Dropping a column destroys data. Take a backup first, and make sure the release before this one has run long enough to be trusted. There's no prize for contracting quickly.

Looking Back

CHAPTER	ROLE IN THE CAPSTONE
1. Transactions	Version check in <code>setStatus</code> ; consistent verification reads
2. Aggregation	Posts per status with <code>groupBy</code>
3. Raw SQL	The backfill and the mismatch check
4. Seeding	Sample data with archived posts
5. Testing	Integration test proving both fields agree
6. Extensions	<code>setStatus</code> and the single source of the write rule
7. Performance	The <code>status, createdAt</code> index for the front page
8. Schema evolution	The whole expand-and-contract sequence
9. Deploying	One migration step per release, before the new code starts

Hands-On Exercises

EXERCISE 1

Midway through (after Release 2), a bug is found and Release 2 must be rolled back to Release 1's code. What happens to posts archived in the meantime, and what must you check before redeploying Release 2?

[View solution](#)

EXERCISE 2

The posts table has 20 million rows. Replace Release 3's single `UPDATE` with a batched backfill script that's safe to stop and restart, and report progress.

[View solution](#)

EXERCISE 3

Apply the same approach to a new change: split `User.name` into `firstName` and `lastName`. Write the release plan and the backfill SQL, and point out where this is harder than the status change.

[View solution](#)

CHAPTER 10 QUICK REFERENCE

- Expand → dual write → backfill → switch reads → stop old writes → contract, each a separate release
- Keep the dual-write rule in one function or extension, so stopping it is a one-line change
- Ask what the *old* code will do with each new state before introducing it
- Backfills are re-runnable and verified with counts before the next step
- Give a newly required column a default so old code's inserts keep working
- Search raw SQL and scripts, not just TypeScript, before removing a field
- Dropping the column is the only irreversible step: back up and wait

COURSE COMPLETE

Together with Prisma Fundamentals, this course has taken the blog API from its first model to a schema that can change safely while in use. The same habits — measure, keep rules in one place, change in small reversible steps — apply well beyond Prisma.