

PERSONAL MULTI-TECHNOLOGY PROJECTS

Premier League Predictor: Astro

A real, working weekly Premier League prediction tracker — four scored prediction sources (user, BBC expert, guest(s) averaged, BBC AI), Astro's own native server endpoints standing in as the entire backend with a single-file SQLite database via better-sqlite3, two genuinely different league tables, promotion and relegation, a real dynamic-route gameweek/ season selector, and a real deployment mounted onto an existing Astro site — the most direct "bolt this onto my real live site" story of the whole four-course set.

12 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & ASTRO + A THIN API BACKEND SETUP

Premier League Predictor: Astro

Chapter 1 · Project Overview & Astro + a Thin API Backend Setup

This is the fourth course built in the Premier League Predictor set — after **Premier League Predictor: FastAPI & PostgreSQL** (the full relational reference implementation) and **Premier League Predictor: Django & MySQL**, both complete, with **Premier League Predictor: FastAPI & Redis** still outlined. This variant takes a genuinely different angle from all three: rather than standing up a separate backend framework and API service, it builds the entire app — pages and data layer alike — inside a single real Astro project, using Astro's own native server endpoints instead of a separate service. That makes this the most direct "how would this actually live on my real site" story of the whole set, since the site this predictor is meant to eventually join is itself already built in Astro.

What the App Actually Does

The same shared spec every course in this set builds toward — a personal, weekly Premier League prediction tracker, not a generic sports app:

- **Track every fixture** across a real 38-gameweek Premier League season, 20 teams, 10 fixtures per gameweek.
- **Record four real prediction sources per fixture:**
 - the user's own prediction,
 - the BBC's expert prediction (currently Chris Sutton),
 - that week's guest predictor(s) — genuinely variable; some weeks have more than one guest, whose scores get averaged into a single "guest" figure for that week,
 - and the BBC's own published AI-generated prediction, manually transcribed each week rather than scraped or generated by this app.
- **Score two different things per prediction** once a result is in — a **correct score** (the exact scoreline matched) and a **correct result** (the predicted win/draw/loss outcome matched, even if the scoreline itself was wrong).
- **Maintain the real league table** — points (3/1/0), goal difference, wins/draws/losses — alongside a second, parallel **prediction league table** ranking how each of the four real predictors is actually doing across the season.

- **Handle promotion and relegation** between seasons — the bottom three teams are auto-calculated and removed at season end; the three promoted teams are entered manually, since which teams come up from the Championship isn't something this app tracks on its own.

Fixture entry is deliberately fast to use, not a generic form. Ten fixtures a week, every single week, for 38 weeks — a dropdown-heavy "select home team, select away team" form for every one of them would be genuinely tedious. Chapter 4 builds a real click-to-pair interface instead: 20 clickable team buttons, paired directly into home/away boxes.

THE POINT VALUES ARE ALREADY SETTLED

Unlike this course's own FastAPI sibling, which had to leave the real scoring numbers open in its own first chapter, this course starts with them already confirmed: a **correct score** is worth **40 points**, and a **correct result** is worth **10 points** — resolved directly with the user during the FastAPI course's own Chapter 6, and reused unchanged here so every variant in this set scores identically.

Why Astro for This One

Of the four variants in this set, this is the one built specifically to answer "could this genuinely just be a page on my real site" — and Astro's own real feature set makes that possible without reaching for a second framework at all:

- **Astro's own server endpoints are the backend.** A file under `src/pages/api/` exporting a `GET` or `POST` function is a real, working API route — there's no separate Express server, no separate FastAPI process, nothing extra to deploy alongside the site itself.
- **A single SQLite file, not a database server.** A personal hobby project run by one person doesn't need a standalone PostgreSQL or MySQL server sitting on standby — `better-sqlite3` reads and writes one plain file on disk, which is exactly the amount of database this app actually needs.
- **Per-page rendering control, decided route by route.** Astro can serve some pages statically and others on demand from the very same project, which turns out to matter a great deal for a data-driven app like this one — covered in real depth in Chapter 10.

THIS COURSE ASSUMES REAL ASTRO GROUNDWORK ALREADY COVERED ELSEWHERE

This site's own `astro1` (Astro) course already covers Astro's own component model, file-based routing, and static-first rendering philosophy in real depth, and `website-rebuild-with-astro1` covers a full real Astro site rebuild. This course doesn't re-teach any of that — it assumes it, and moves straight into this predictor's own real schema, endpoints, and features.

Setting Up: Astro, Server Endpoints & SQLite

A local Node.js install is assumed from here on. Scaffold a fresh Astro project first:

```
# Scaffold a new Astro project (choose the empty template, TypeScript "Strict")
npm create astro@latest -- pl-predictor-astro
cd pl-predictor-astro

# Add the Node adapter, needed for real server-rendered routes
npx astro add node

# Install a lightweight, file-based SQLite driver
npm install better-sqlite3
```

Astro renders every page statically by default. Since this app needs real, on-demand server logic — API routes, an admin area that writes to the database — `astro.config.mjs` needs to switch the whole project over to server output first:

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import node from '@astrojs/node';

export default defineConfig({
  output: 'server',
  adapter: node({ mode: 'standalone' }),
});
```

A minimal project structure, following this site's own established Astro project convention rather than inventing a new one:

```

pl-predictor-astro/
├─ astro.config.mjs
├─ src/
│   └─ lib/
│       └─ db.ts          # better-sqlite3 connection + schema setup (Chapter 2)
│   └─ pages/
│       └─ index.astro    # the real league table (Chapter 7)
│       └─ admin/        # team/fixture management (Chapter 3-4)
│           └─ api/
│               └─ health.ts
│               └─ teams.ts
│               └─ fixtures.ts
│               └─ predictions.ts
│   └─ components/
└─ data/
    └─ pl_predictor.db    # created on first run, gitignored

```

A minimal health-check endpoint, confirming the setup works before any real route exists:

```

// src/pages/api/health.ts
import type { APIRoute } from 'astro';

export const GET: APIRoute = async () => {
  return new Response(JSON.stringify({ status: 'ok' }), {
    status: 200,
    headers: { 'Content-Type': 'application/json' },
  });
};

```

```
npm run dev
```

Visiting `http://localhost:4321/api/health` should return `{"status":"ok"}` — a real, live server response coming straight out of the Astro project itself, with no second process running anywhere.

EVERY PAGE DEFAULTS TO ON-DEMAND NOW — THAT'S A DELIBERATE, TEMPORARY TRADEOFF

Setting `output: 'server'` makes every page in the project server-rendered on each request by default, not just the API routes — including pages that don't actually need it, like a mostly-static "About this tracker" page, if one ever gets added. That's the right tradeoff for this chapter, since almost every page this app needs really is dynamic. Chapter 10 comes back to this directly: Astro lets an individual page opt back into static prerendering with a single `export const prerender = true` line, and by then it'll be clear exactly which of this app's own pages can genuinely take advantage of that.

Where This Course Is Headed

A real SQLite schema for teams, seasons, gameweeks, and fixtures (Chapter 2); managing the 20 competing teams each season (Chapter 3); the fast click-to-pair fixture-entry UI (Chapter 4); recording all four prediction sources per fixture (Chapter 5); entering results and applying the confirmed 40/10 scoring (Chapter 6); the real league table (Chapter 7); the parallel prediction league table (Chapter 8); promotion and relegation between seasons (Chapter 9); Astro islands and exactly where client-side JavaScript actually earns its place in a static-first framework (Chapter 10); deployment (Chapter 11); and a capstone on mounting this predictor directly onto the real, live Astro site (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain the real difference between a "correct score" and a "correct result" prediction, and describe one real scenario where a prediction earns the 10-point correct-result score but not the 40-point correct-score bonus.

 [View solution](#)

EXERCISE 2

Explain why this app averages multiple guest predictors into a single "guest" figure for weeks with more than one guest, rather than tracking each guest as a permanently separate predictor across the season.

[View solution](#)

EXERCISE 3

Set up the Astro project and the `/api/health` route yourself, confirm it responds with a real server request, then explain — in your own words — the real difference between a page that's server-rendered on every request under `output: 'server'` and one that opts back into static prerendering with `export const prerender = true`.

[View solution](#)

CHAPTER 1 QUICK REFERENCE

- **The shared app** — a weekly Premier League prediction tracker: 4 real prediction sources per fixture (user, BBC expert, guest(s) averaged, BBC AI), scored 40 points for a correct score and 10 for a correct result, across a real 38-gameweek season
- **Two league tables** — the real Premier League table, plus a parallel prediction league table ranking predictor performance
- **Promotion/relegation** — bottom 3 auto-calculated and removed; the 3 promoted teams entered manually
- **Why this variant** — Astro's own native server endpoints as the entire backend, plus a single-file SQLite database — the most direct "bolt this onto my real site" story of the whole set
- **Config** — `output: 'server'` with the `@astrojs/node` adapter, so every route is on-demand by default
- **Database** — SQLite via `better-sqlite3`, a single gitignored file, no separate database server to run
- **Assumed groundwork** — Astro fundamentals, covered in this site's own dedicated `astro1` and `website-rebuild-with-astro1` courses
- **Next chapter:** Data Modeling — teams, seasons, gameweeks & fixtures, as a lightweight SQLite-backed API

CHAPTER 2: DATA MODELING: TEAMS, SEASONS, GAMEWEEKS & FIXTURES

Premier League Predictor: Astro

Chapter 2 · Data Modeling: Teams, Seasons, Gameweeks & Fixtures

Everything else in this course — the admin tools, the fixture-entry UI, predictions, results, both league tables, promotion and relegation — sits on top of the schema this chapter builds. Five real tables: `teams`, `seasons`, `season_teams`, `gameweeks`, and `fixtures`, written as raw SQL rather than through an ORM, since `better-sqlite3` is a thin, direct SQLite driver with no object-relational layer sitting on top of it. Predictions themselves are deliberately left out of this chapter — that's a genuinely separate concern, covered on its own in Chapter 5.

Team & Season: What Persists vs. What's Real Per Season

A real football club doesn't stop existing the season it gets relegated. `teams` holds every team this app has ever tracked — promoted, relegated, or currently playing, it doesn't matter — because a team's own identity, name, and history are genuinely permanent facts, not something tied to a single season.

`seasons` represents one real Premier League season — "2026/27," with a start date, an end date, and an `is_current` flag so the app always knows which season is the active one.

season_teams: The 20 Teams Actually Competing

Which 20 teams are actually in the Premier League changes every season — three go down, three come up. That's a genuinely different fact from "this team exists," so it gets its own table rather than a column on `teams` itself: `season_teams`, a join table recording which teams are competing in which season.

THIS IS EXACTLY WHAT CHAPTER 3 AND CHAPTER 9 OPERATE ON

Chapter 3's own admin tooling for managing "the 20 competing teams each season" is really just CRUD against `season_teams` rows for the current season. Chapter 9's promotion/relegation logic, at the end of a season, is the same table on the other end: remove the bottom three teams' own `season_teams` rows for next season, add three new rows for the promoted teams. `teams` itself

never changes in either operation — only which teams are marked as competing in a given season does.

Gameweek & Fixture: The Real Schema

A real season has 38 gameweeks; each gameweek has (normally) 10 fixtures, one per pair of the 20 competing teams. The whole schema, written once as plain SQL and run at startup:

```
-- src/lib/schema.sql
CREATE TABLE IF NOT EXISTS teams (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL UNIQUE,
  short_name TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS seasons (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL UNIQUE,           -- e.g. "2026/27"
  start_date TEXT NOT NULL,           -- ISO 8601, e.g. "2026-08-15"
  end_date TEXT NOT NULL,
  is_current INTEGER NOT NULL DEFAULT 0 -- SQLite has no BOOLEAN type; 0 or 1
);

CREATE TABLE IF NOT EXISTS season_teams (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  season_id INTEGER NOT NULL REFERENCES seasons(id),
  team_id INTEGER NOT NULL REFERENCES teams(id),
  UNIQUE (season_id, team_id)
);

CREATE TABLE IF NOT EXISTS gameweeks (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  season_id INTEGER NOT NULL REFERENCES seasons(id),
  number INTEGER NOT NULL,           -- 1 through 38
  UNIQUE (season_id, number)
);

CREATE TABLE IF NOT EXISTS fixtures (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```

gameweek_id INTEGER NOT NULL REFERENCES gameweeks(id),
home_team_id INTEGER NOT NULL REFERENCES teams(id),
away_team_id INTEGER NOT NULL REFERENCES teams(id),
kickoff_time TEXT,                -- ISO 8601, nullable
home_score INTEGER,               -- nullable until played
away_score INTEGER,
status TEXT NOT NULL DEFAULT 'scheduled',
CHECK (home_team_id != away_team_id)
);

```

WHY THIS STACK NEVER RUNS INTO THE FASTAPI SIBLING'S OWN FOREIGN-KEY PROBLEM

The FastAPI & PostgreSQL course had to resolve a real ambiguity: SQLAlchemy needs an explicit `foreign_keys=[...]` argument on `home_team` and `away_team`, because its ORM layer tries to *infer* which relationship follows which foreign key, and two foreign keys pointing at the same table make that inference genuinely ambiguous. That whole problem simply doesn't exist here. `home_team_id` and `away_team_id` are just two plain columns, each with its own explicit `REFERENCES teams(id)` clause, read directly off the row with no relationship-inference step sitting in between. Nothing is being guessed, so there's nothing to disambiguate — a genuinely different consequence of a genuinely different kind of database layer, not a coincidence.

SQLITE IGNORES FOREIGN KEYS UNLESS A PRAGMA TURNS THEM ON

Every `REFERENCES` clause above is declared correctly, but SQLite — unlike PostgreSQL, which always enforces foreign keys by default — genuinely ignores foreign key constraints on every connection until `PRAGMA foreign_keys = ON` is explicitly run against that connection. Skip it, and inserting a fixture with a `home_team_id` that doesn't exist in `teams` at all will silently succeed, with the broken reference only surfacing later, whenever something tries to join against the missing row. This is set once, in the connection module below, before the schema is even created.

The Database Connection Module

One shared module opens the database, turns on foreign key enforcement, and creates every table above if it doesn't already exist — imported by every API route that needs it, rather than each route managing its own connection:

```

// src/lib/db.ts
import Database from 'better-sqlite3';

```

```
import { readFileSync } from 'node:fs';
import path from 'node:path';
import { fileURLToPath } from 'node:url';

const __dirname = path.dirname(fileURLToPath(import.meta.url));
const dbPath = path.join(process.cwd(), 'data', 'pl_predictor.db');

export const db = new Database(dbPath);

// Off by default in SQLite — must be set per connection, before anything else runs
db.pragma('foreign_keys = ON');

const schema = readFileSync(path.join(__dirname, 'schema.sql'), 'utf-8');
db.exec(schema);
```

BETTER-SQLITE3 IS GENUINELY SYNCHRONOUS — THERE'S NO AWAIT INSIDE IT

Most Node.js database drivers are asynchronous, returning promises for every query. `better-sqlite3` deliberately isn't — `db.prepare(sql).get(...)`, `.all(...)`, and `.run(...)` all execute immediately and return a real value directly, with no promise and no `await` anywhere in the database layer itself. Every API route built from Chapter 3 onward still declares its `GET` / `POST` function as `async`, since that's what Astro's own `APIRoute` type expects — but the actual database calls inside those functions run synchronously underneath, finishing before the next line of code even starts.

Nullable Scores: A Fixture Exists Before It's Played

`home_score` and `away_score` are both left off `NOT NULL`, and `status` defaults to `'scheduled'`. A fixture row is created the moment it's added to a gameweek — Chapter 4's own click-to-pair UI — long before it's actually played. Entering a result in Chapter 6 isn't an `INSERT` at all; it's an `UPDATE` against a fixture row that has existed since the gameweek was first set up, filling in the two score columns and flipping `status` to `'played'`.

ONE REAL CONSTRAINT THIS SCHEMA CAN'T ENFORCE ON ITS OWN

`CHECK (home_team_id != away_team_id)` stops a team from playing itself — that's a genuine per-row check SQLite can enforce directly, the same as it would in PostgreSQL. A team appearing *twice in the same gameweek* — once as home in one fixture, once as away in a different fixture — is a different kind of rule entirely: it spans multiple rows, comparing every fixture in a gameweek against every

other one. A single-row `CHECK` constraint can't express that in SQLite any more than it could in Postgres. This schema leaves it to the application layer instead: Chapter 4's own fixture-entry UI removes a team from the list of clickable options the moment it's already been used somewhere in that gameweek, so the rule is enforced by what the interface lets you click, not by the database refusing an insert after the fact.

REAL FIXTURES DO GET MOVED BETWEEN GAMEWEEKS — THIS SCHEMA ALREADY ALLOWS IT

Real Premier League fixtures are rescheduled sometimes — cup replays, European commitments, TV moves. Because `gameweek_id` is just a plain column on `fixtures`, moving a fixture to a different gameweek is technically nothing more than updating one column on one row — the schema itself puts up no resistance. What's still genuinely undecided is the actual workflow for doing that safely in this app (should predictions already made against the old gameweek carry over, get cleared, or something else?) — a real open question, not something this chapter needs to resolve to keep building.

TypeScript Row Types

Following the same two-layer discipline this course's own siblings use — the schema describes what's actually stored, a set of plain interfaces describes what a row looks like once it comes back out of the database:

```
// src/lib/types.ts
export interface TeamRow {
  id: number;
  name: string;
  short_name: string;
}

export interface SeasonRow {
  id: number;
  name: string;
  start_date: string;
  end_date: string;
  is_current: number; // 0 or 1 — SQLite has no real boolean column type
}

export interface FixtureRow {
```

```
id: number;
gameweek_id: number;
home_team_id: number;
away_team_id: number;
kickoff_time: string | null;
home_score: number | null;
away_score: number | null;
status: string;
}
```

`is_current` stays a raw `number` at this layer, matching exactly what SQLite actually stores and hands back. Any code that wants a real boolean converts it explicitly —

`Boolean(season.is_current)` — rather than pretending the database column was ever a true boolean type to begin with.

Where This Course Is Headed

Managing the 20 competing teams each season — real CRUD against `season_teams` (Chapter 3); the fast click-to-pair fixture-entry UI that creates `fixtures` rows against this chapter's own schema (Chapter 4); recording all four prediction sources per fixture (Chapter 5); entering results — the real `UPDATE` this chapter set up (Chapter 6); the real league table (Chapter 7); the prediction league table (Chapter 8); and promotion/relegation, operating on `season_teams` exactly as previewed above (Chapter 9).

Hands-On Exercises

EXERCISE 1

Explain why "which 20 teams are competing this season" is modeled as a separate `season_teams` table rather than a column on `teams` itself, and name the two later chapters that operate directly on `season_teams` rows.

 [View solution](#)

EXERCISE 2

Explain why fixtures' `home_team_id` and `away_team_id` columns don't need anything like SQLAlchemy's `foreign_keys=[...]` argument in this raw-SQL, better-sqlite3 schema, even though both columns reference the same teams table.

 [View solution](#)

EXERCISE 3

Demonstrate the real consequence of forgetting `PRAGMA foreign_keys = ON`: insert a fixture row referencing a team ID that doesn't exist, once against a database connection where the pragma was never set, and once against one where it was — then write down exactly what happens in each case.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **teams** — every team ever tracked, permanent regardless of promotion/relegation
- **seasons** — one real Premier League season, with an `is_current` flag stored as 0/1
- **season_teams** — the join table recording which 20 teams compete in a given season; Chapters 3 and 9 both operate directly on it
- **gameweeks** — 1 through 38 per season, unique per (season_id, number)
- **fixtures** — `home_team_id/away_team_id` both reference `teams(id)` directly, with no relationship-inference layer to disambiguate, unlike the SQLAlchemy sibling
- **Real gotcha** — SQLite ignores foreign keys entirely until `PRAGMA foreign_keys = ON` is run per connection, set once in `db.ts` before the schema is created
- **better-sqlite3 is synchronous** — `.get()/all()/run()` return real values directly, no `await` needed inside the database layer itself
- **Nullable home_score/away_score** — a fixture exists before it's played; Chapter 6 fills these in via `UPDATE`, not `INSERT`
- **Real limit** — "no team plays twice in one gameweek" is enforced by the UI (Chapter 4), not a database constraint, unlike "a team can't play itself" (`CHECK`)
- **Still open** — the real workflow for moving a fixture to a different gameweek after it's already been entered

- **Next chapter:** Admin — managing the 20 competing teams each season

CHAPTER 3: ADMIN: MANAGING THE 20 COMPETING TEAMS EACH SEASON

Premier League Predictor: Astro

Chapter 3 · Admin: Managing the 20 Competing Teams Each Season

Chapter 2 built the schema; this chapter turns `seasons` and `season_teams` into real, working Astro API routes — the actual admin tooling used to set up a new season and manage which 20 teams are competing in it.

A Small JSON Response Helper, Reused From Here On

Every route from this chapter onward needs to return JSON with the right `Content-Type` header and status code. Rather than repeating that boilerplate in every single route, one small helper handles it once:

```
// src/lib/http.ts
export function json(data: unknown, status = 200): Response {
  return new Response(JSON.stringify(data), {
    status,
    headers: { 'Content-Type': 'application/json' },
  });
}
```

Bootstrapping a Season

```
// src/pages/api/seasons/index.ts
import type { APIRoute } from 'astro';
import { db } from '../../../lib/db';
import { json } from '../../../lib/http';

export const GET: APIRoute = async () => {
  const seasons = db.prepare('SELECT * FROM seasons ORDER BY start_date DESC').all();
  return json(seasons);
};
```

```

export const POST: APIRoute = async ({ request }) => {
  const { name, start_date, end_date, is_current = false } = await request.json();

  if (is_current) {
    // Only one season is ever "current" — clear the flag on every other one first
    db.prepare('UPDATE seasons SET is_current = 0').run();
  }

  const result = db.prepare(
    'INSERT INTO seasons (name, start_date, end_date, is_current) VALUES (?, ?, ?, ?)'
  ).run(name, start_date, end_date, is_current ? 1 : 0);

  const season = db.prepare('SELECT * FROM seasons WHERE id = ?').get(result.lastInsertId);
  return json(season, 201);
};

```

ANOTHER CROSS-ROW RULE, ENFORCED THE SAME WAY CHAPTER 2 ALREADY ESTABLISHED

"At most one season is marked current" spans every row in `seasons`, not just the one being inserted — the same category of rule as Chapter 2's own "no team twice in a gameweek," which a single-row constraint can't express on its own. The plain `UPDATE seasons SET is_current = 0` above clears every other season's flag before the new one is even inserted — a deliberate, explicit app-level guarantee rather than something left to chance.

Adding a Team to a Season. Reuse or Create?

A promoted club might genuinely be a completely new name to this app, or it might be a club that was in the Premier League three seasons ago, got relegated, and is only now coming back up. Chapter 2's own design decision — `teams` rows persist forever, independent of any single season — means this route has to check which case it's actually in before deciding what to do:

```

// src/pages/api/seasons/[seasonId]/teams/index.ts
import type { APIRoute } from 'astro';
import { db } from '../../../lib/db';
import { json } from '../../../lib/http';

const MAX_TEAMS_PER_SEASON = 20;

```

```

export const POST: APIRoute = async ({ params, request }) => {
  const seasonId = Number(params.seasonId);
  const { team_name, short_name } = await request.json();

  const season = db.prepare('SELECT * FROM seasons WHERE id = ?').get(seasonId);
  if (!season) {
    return json({ error: 'Season not found' }, 404);
  }

  const addTeam = db.transaction(() => {
    const { count } = db.prepare(
      'SELECT COUNT(*) AS count FROM season_teams WHERE season_id = ?'
    ).get(seasonId) as { count: number };

    if (count >= MAX_TEAMS_PER_SEASON) {
      throw new Error('SEASON_FULL');
    }

    // Reuse the historical team row if this club has been tracked before
    let team = db.prepare('SELECT * FROM teams WHERE name = ?').get(team_name) as
      { id: number; name: string; short_name: string } | undefined;

    if (!team) {
      const result = db.prepare(
        'INSERT INTO teams (name, short_name) VALUES (?, ?)'
      ).run(team_name, short_name ?? team_name);
      team = db.prepare('SELECT * FROM teams WHERE id = ?').get(result.lastInsertRowid)
    }

    const existing = db.prepare(
      'SELECT * FROM season_teams WHERE season_id = ? AND team_id = ?'
    ).get(seasonId, team!.id);

    if (existing) {
      throw new Error('ALREADY_IN_SEASON');
    }

    db.prepare(
      'INSERT INTO season_teams (season_id, team_id) VALUES (?, ?)'
    ).run(seasonId, team!.id);

    return team;
  });

```

```

});

try {
  const team = addTeam();
  return json(team, 201);
} catch (err) {
  if ((err as Error).message === 'SEASON_FULL') {
    return json({ error: `Season already has ${MAX_TEAMS_PER_SEASON} teams` }, 400);
  }
  if ((err as Error).message === 'ALREADY_IN_SEASON') {
    return json({ error: 'Team is already in this season' }, 409);
  }
  throw err;
}
};

```

DB.TRANSACTION() REPLACES FLUSH()-THEN-COMMIT() HERE — A GENUINELY DIFFERENT MECHANISM FOR THE SAME GUARANTEE

The FastAPI & PostgreSQL course uses `db.flush()` to get a new `Team` row's id assigned without ending the SQLAlchemy session, so it can build the related `SeasonTeam` row before the final `commit()` — both rows join or fail together as one real transaction. This stack has no session lifecycle to manage at all, so it reaches the same all-or-nothing guarantee differently:

`db.transaction(fn)` wraps the entire inner function — the count check, the team lookup-or-create, the duplicate check, and the final insert — as one real SQLite transaction. If `fn` returns normally, everything inside it commits together; if it throws anywhere, SQLite automatically rolls back everything it did, and nothing partial is ever left behind. Same real atomicity, reached through wrapping a whole function rather than manually staging a flush partway through a longer-lived session.

ASTRO HAS NO AUTOMATIC EXCEPTION-TO-JSON TRANSLATION, UNLIKE FASTAPI'S HTTPEXCEPTION

FastAPI's own `HTTPException` is caught by the framework itself and turned into a proper JSON error response with the right status code — raising it is enough. Astro's API routes have no equivalent built in: an uncaught thrown error inside a route handler becomes a generic, unstyled 500 response with no useful JSON body at all. That's exactly why the errors above are thrown as plain `Error` objects carrying a short string code (`'SEASON_FULL'` , `'ALREADY_IN_SEASON'`) *inside* the transaction, then caught and explicitly translated into a real `json({error: ...}, status)` response *outside* it. Every route in this course follows the same shape: known failure cases are caught and turned into a

deliberate response; anything genuinely unexpected is re-thrown and allowed to become a 500, since there's nothing sensible to say about it anyway.

NO REFRESH() NEEDED — THE JUST-INSERTED ROW IS SIMPLY READ BACK

SQLAlchemy's `db.refresh(db_season)` re-populates a Python object's fields from the database after an insert. `better-sqlite3` has no such object to refresh in the first place —

`result.lastInsertRowid` gives the new row's real id directly off the `.run()` result, and a plain follow-up `SELECT ... WHERE id = ?` reads the row back exactly as it was actually stored, including any column defaults SQLite itself applied.

REAL NAMES MUST MATCH EXACTLY, AND THERE'S NO FUZZY MATCHING

`WHERE name = ?` only reuses an existing team if the name matches character-for-character. Typing "Nottingham Forest" one season and "Nott'm Forest" the next creates a genuine duplicate `teams` row rather than reusing the real one — this route trusts the admin to type the name consistently rather than trying to guess a match. A dropdown of already-known team names (built from Chapter 4's own "20 clickable team buttons" pattern) would be a real, worthwhile fix, but isn't built in this chapter.

Removing a Team From a Season

```
// src/pages/api/seasons/[seasonId]/teams/[teamId].ts
import type { APIRoute } from 'astro';
import { db } from '../../lib/db';
import { json } from '../../lib/http';

export const DELETE: APIRoute = async ({ params }) => {
  const seasonId = Number(params.seasonId);
  const teamId = Number(params.teamId);

  const result = db.prepare(
    'DELETE FROM season_teams WHERE season_id = ? AND team_id = ?'
  ).run(seasonId, teamId);

  if (result.changes === 0) {
    return json({ error: 'Team is not in this season' }, 404);
  }
}
```

```
return new Response(null, { status: 204 });
};
```

.CHANGES STANDS IN FOR A SEPARATE EXISTENCE CHECK

The FastAPI sibling has to `SELECT` the `SeasonTeam` row first specifically to know whether it existed at all, before deciding whether to delete it or return a 404. `better-sqlite3`'s own `.run()` result reports `changes` — the real number of rows the statement actually affected — so a single `DELETE` already answers both questions at once: if `changes` is `0`, nothing matched, and the 404 is returned with no separate lookup query needed first.

This deletes the `season_teams` row, not the `team` itself — exactly the point of splitting the two tables back in Chapter 2. A relegated club's own historical `teams` row is untouched; only the fact "competing in this particular season" goes away.

Listing a Season's Teams

```
// src/pages/api/seasons/[seasonId]/teams/index.ts (GET, added alongside POST above)
export const GET: APIRoute = async ({ params }) => {
  const seasonId = Number(params.seasonId);
  const teams = db.prepare(`
    SELECT teams.id, teams.name, teams.short_name
    FROM season_teams
    JOIN teams ON teams.id = season_teams.team_id
    WHERE season_teams.season_id = ?
    ORDER BY teams.name
  `).all(seasonId);

  return json(teams);
};
```

NO NESTED-MODEL SERIALIZATION STEP EXISTS TO COMPARE AGAINST

The FastAPI sibling needs a nested `SeasonTeamResponse.team: TeamResponse` Pydantic model to shape a joined query result into the right JSON structure. There's no equivalent step here at all — the raw SQL `JOIN` above already selects exactly the columns the response needs, in exactly the shape it needs them, and `db.prepare(...).all()` hands back plain JavaScript objects that `json()` serializes

directly. Nothing is being reshaped between the query and the response, because the query was written to already produce the response's own shape.

THESE ROUTES HAVE NO ACCESS CONTROL

Every route in this chapter is reachable by anyone who can reach the deployed site — there's no login, no admin check, nothing gating who can add or remove a team. That's a deliberate scope decision for a personal, single-operator tool, not an oversight: this course has no dedicated authentication chapter, since the real, honest question of whether this app is ever meant to be used by more than one person is still genuinely open.

Where This Course Is Headed

The fast click-to-pair fixture-entry UI, built directly on top of this chapter's own `GET /api/seasons/{seasonId}/teams` route to populate its 20 clickable team buttons (Chapter 4); recording predictions per fixture (Chapter 5); entering results (Chapter 6); both league tables (Chapters 7-8); and promotion/relegation, which reuses this chapter's own add/remove routes directly at the season boundary (Chapter 9).

Hands-On Exercises

EXERCISE 1

Explain why the `POST /api/seasons` route clears `is_current` on every other season with a plain `UPDATE` before inserting the new one, and what would go wrong if that step were skipped.

 [View solution](#)

EXERCISE 2

Explain why the add-team route looks up an existing team by name before creating a new one, and describe a real scenario where skipping that lookup would create a duplicate teams row for the same real club.

 [View solution](#)

EXERCISE 3

Explain why the team-lookup-or-create step and the season_teams insert are both wrapped inside a single `db.transaction()` call rather than run as two separate, un-wrapped statements, and what real guarantee would be lost if the wrapper were removed.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **json() helper** — one small function wrapping every route's own JSON response and status code, reused from here on
- **POST /api/seasons** — creates a season; clears `is_current` on every other season first if the new one is marked current
- **POST /api/seasons/{id}/teams** — reuses an existing team by name if one matches, otherwise creates one; enforces a 20-team cap via a COUNT check, all wrapped in `db.transaction()`
- **db.transaction() vs. flush()/commit()** — a whole function committed or rolled back together, replacing SQLAlchemy's own mid-session flush pattern
- **No HTTPException equivalent** — known errors are thrown as plain Error objects inside the transaction, then explicitly caught and translated into `json({error}, status)` responses, since an uncaught error would otherwise become a bare 500
- **DELETE /api/seasons/{id}/teams/{teamId}** — removes the season_teams row only; `result.changes == 0` replaces a separate existence-check query
- **GET /api/seasons/{id}/teams** — a real JOIN, already shaped to match the response with no serialization step needed
- **Real limit** — team-name matching is exact, no fuzzy matching; no access control on any route in this chapter
- **Next chapter:** The fast click-to-pair fixture-entry UI

CHAPTER 4: THE FAST FIXTURE-ENTRY UI: CLICK-TO-PAIR TEAMS INTO A GAMEWEEK

Premier League Predictor: Astro

Chapter 4 · The Fast Fixture-Entry UI: Click-to-Pair Teams Into a Gameweek

Ten fixtures, every single gameweek, for 38 gameweeks — filling that in through two dropdowns per fixture would be genuinely tedious. This chapter builds the real alternative promised back in Chapter 1: a real Astro page with 20 always-visible team buttons, clicked directly into place, backed by two new API routes for gameweeks and fixtures.

Creating a Gameweek

```
// src/pages/api/seasons/[seasonId]/gameweeks/index.ts
import type { APIRoute } from 'astro';
import { db } from '../../lib/db';
import { json } from '../../lib/http';

export const POST: APIRoute = async ({ params, request }) => {
  const seasonId = Number(params.seasonId);
  const { number } = await request.json();

  const season = db.prepare('SELECT * FROM seasons WHERE id = ?').get(seasonId);
  if (!season) {
    return json({ error: 'Season not found' }, 404);
  }

  if (!(number >= 1 && number <= 38)) {
    return json({ error: 'Gameweek number must be between 1 and 38' }, 400);
  }

  try {
    const result = db.prepare(
      'INSERT INTO gameweeks (season_id, number) VALUES (?, ?)'
    ).run(seasonId, number);
    const gameweek = db.prepare('SELECT * FROM gameweeks WHERE id = ?').get(result.last);
    return json(gameweek, 201);
  } catch (err) {
    if ((err as any).code === 'SQLITE_CONSTRAINT_UNIQUE') {

```

```

    return json({ error: `Gameweek ${number} already exists for this season` }, 409);
  }
  throw err;
}
};

```

SQLITEERROR.CODE TELLS YOU EXACTLY WHICH CONSTRAINT FAILED — AND THERE'S NO ROLLBACK TO WORRY ABOUT

Chapter 2's own `UNIQUE (season_id, number)` already stops a genuinely duplicate gameweek from ever being stored. `better-sqlite3` reports the violation as a real `SqliteError` instance with a `.code` property matching SQLite's own extended result codes — `'SQLITE_CONSTRAINT_UNIQUE'` here, and `'SQLITE_CONSTRAINT_CHECK'` for the fixture chapter's own team-can't-play-itself rule below. Checking that exact string is enough to turn a real database guarantee into a genuinely useful `409` response. There's no equivalent to SQLAlchemy's own required `db.rollback()` step here either — a single `.run()` call that throws simply never took effect at all, so there's no half-finished session state left behind that needs explicitly clearing before the database can be used again.

Creating a Fixture: The Real Server-Side Guard

```

// src/pages/api/gameweeks/[gameweekId]/fixtures/index.ts
import type { APIRoute } from 'astro';
import { db } from '../../lib/db';
import { json } from '../../lib/http';

export const POST: APIRoute = async ({ params, request }) => {
  const gameweekId = Number(params.gameweekId);
  const { home_team_id, away_team_id, kickoff_time = null } = await request.json();

  const gameweek = db.prepare('SELECT * FROM gameweeks WHERE id = ?').get(gameweekId);
  if (!gameweek) {
    return json({ error: 'Gameweek not found' }, 404);
  }

  if (home_team_id === away_team_id) {
    return json({ error: 'A team cannot play itself' }, 400);
  }
}

```

```

// The real authority: check every fixture already entered this gameweek
const existing = db.prepare(
  'SELECT home_team_id, away_team_id FROM fixtures WHERE gameweek_id = ?'
).all(gameweekId) as { home_team_id: number; away_team_id: number }[];

const usedTeamIds = new Set<number>();
for (const fx of existing) {
  usedTeamIds.add(fx.home_team_id);
  usedTeamIds.add(fx.away_team_id);
}

if (usedTeamIds.has(home_team_id) || usedTeamIds.has(away_team_id)) {
  return json({ error: 'One of these teams is already fixtured this gameweek' }, 409)
}

const result = db.prepare(
  'INSERT INTO fixtures (gameweek_id, home_team_id, away_team_id, kickoff_time) VALUES'
).run(gameweekId, home_team_id, away_team_id, kickoff_time);

const fixture = db.prepare('SELECT * FROM fixtures WHERE id = ?').get(result.lastInsertId);
return json(fixture, 201);
};

export const GET: APIRoute = async ({ params }) => {
  const gameweekId = Number(params.gameweekId);
  const fixtures = db.prepare('SELECT * FROM fixtures WHERE gameweek_id = ?').all(gameweekId);
  return json(fixtures);
};

```

THIS IS THE ROUTE CHAPTER 2'S OWN WARN-BOX WAS POINTING AT

Back in Chapter 2: "a team appearing twice in the same gameweek... this schema leaves it to the application layer instead." The `POST` handler above is that application layer — a real, explicit query across every existing fixture in the gameweek, not a database constraint, run in plain JavaScript exactly the same way the FastAPI sibling runs it in Python. Checking `home_team_id === away_team_id` here as well, even though Chapter 2's own `CHECK` constraint already blocks it at the database level, means a bad request gets a clean `400` with a real message instead of an unhandled `SqliteError` — the exact same "friendly error over a raw crash" reasoning as the gameweek route above.

The Click-to-Pair Interface

The whole point: every one of the season's 20 teams is a button, always visible, always one click away — no scrolling through a dropdown to find a name. A minimal Astro page for now; Chapter 10 replaces the hardcoded `seasonId` / `gameweekId` below with a real selector.

```

---
// src/pages/admin/fixtures.astro
---
<html lang="en">
  <body>
    <div id="team-grid" class="team-grid"></div>

    <div class="fixture-slots">
      <div id="home-slot" class="slot">Home</div>
      <span>vs</span>
      <div id="away-slot" class="slot">Away</div>
    </div>

    <button id="clear-btn">Clear</button>
    <button id="add-fixture-btn" disabled>Add Fixture</button>

    <script>
      // In the real app, seasonId and gameweekId come from Chapter 10's own
      // gameweek/season selector — hardcoded here to keep this example focused.
      const seasonId = 1;
      const gameweekId = 1;

      let selectedHome: { id: number; short_name: string } | null = null;
      let selectedAway: { id: number; short_name: string } | null = null;
      const usedTeamIds = new Set<number>();

      async function loadTeams() {
        const res = await fetch(`/api/seasons/${seasonId}/teams`);
        const teams = await res.json();
        const grid = document.getElementById('team-grid')!;
        grid.innerHTML = '';
        teams.forEach((team: any) => {
          const btn = document.createElement('button');
          btn.textContent = team.short_name;
          btn.dataset.teamId = team.id;

```

```

        btn.addEventListener('click', () => selectTeam(team));
        grid.appendChild(btn);
    });
}

function selectTeam(team: { id: number; short_name: string }) {
    if (usedTeamIds.has(team.id)) return;
    if (!selectedHome) {
        selectedHome = team;
        document.getElementById('home-slot')!.textContent = team.short_name;
    } else if (!selectedAway && team.id !== selectedHome.id) {
        selectedAway = team;
        document.getElementById('away-slot')!.textContent = team.short_name;
    }
    (document.getElementById('add-fixture-btn') as HTMLButtonElement).disabled =
        !(selectedHome && selectedAway);
}

function clearSelection() {
    selectedHome = null;
    selectedAway = null;
    document.getElementById('home-slot')!.textContent = 'Home';
    document.getElementById('away-slot')!.textContent = 'Away';
    (document.getElementById('add-fixture-btn') as HTMLButtonElement).disabled = true;
}

async function addFixture() {
    const res = await fetch(`/api/gameweeks/${gameWeekId}/fixtures`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
            home_team_id: selectedHome!.id,
            away_team_id: selectedAway!.id,
        }),
    });

    if (!res.ok) {
        const error = await res.json();
        alert(error.error); // e.g. "One of these teams is already fixtured this game"
        return;
    }
}

```

```

        usedTeamIds.add(selectedHome!.id);
        usedTeamIds.add(selectedAway!.id);
        document.querySelectorAll<HTMLButtonElement>('#team-grid button').forEach((btn)
            if (usedTeamIds.has(Number(btn.dataset.teamId))) btn.disabled = true;
        });
        clearSelection();
    }

    // Real event listeners, not inline onclick – see the finding-box below for why
    document.getElementById('clear-btn')!.addEventListener('click', clearSelection);
    document.getElementById('add-fixture-btn')!.addEventListener('click', addFixture);

    loadTeams();
</script>
</body>
</html>

```

A GENUINE ASTRO-SPECIFIC GOTCHA: INLINE ONCLICK ATTRIBUTES SILENTLY FAIL HERE

A plain `<script>` tag written directly inside an Astro page's own template, with no extra attributes, is automatically processed by Astro: bundled, converted to a real ES module, and deduplicated. That processing has a real, non-obvious consequence — a module's own top-level functions are scoped to that module, not attached to the global `window` object the way an ordinary, unprocessed script's functions would be. Writing `<button onclick="clearSelection()">` the way the FastAPI sibling's own plain, unbundled static file safely can would silently fail here, since the browser's own inline event-handler attribute looks for `clearSelection` on `window` and finds nothing there. Every button interaction above is wired up with a real `addEventListener` call instead — Astro's own documented, recommended fix — which works regardless of whether the function it's calling was ever exposed globally.

CLIENT-SIDE DISABLING IS CONVENIENCE, NOT THE REAL RULE

`usedTeamIds` above lives only in the page's own memory — it starts empty every time the page loads, and only grows as fixtures are actually added during that visit. Reload the page halfway through entering a gameweek and every team button re-enables, even ones already fixtured. That's fine, because the fixture route's own server-side check (querying every existing fixture in the gameweek) is the real authority — a stale client would simply get a real `409` back rather than silently creating a bad duplicate. A more complete version would call `GET /api/gameweeks/{id}/fixtures` on page load and seed `usedTeamIds` from the real, current state,

rather than assuming a fresh visit means a fresh gameweek — left out here to keep the example focused on the click-to-pair interaction itself.

Where This Course Is Headed

Recording all four prediction sources against each fixture this chapter creates (Chapter 5); entering results — the real `UPDATE` Chapter 2 set up, filling in the `home_score` / `away_score` this chapter's own fixtures still leave null (Chapter 6); the real league table (Chapter 7); the prediction league table (Chapter 8); promotion and relegation (Chapter 9); and Astro islands replacing this chapter's own hardcoded `seasonId` / `gameweekId` with a real gameweek/season selector (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why the gameweek and fixture routes both check their own rule in JavaScript (the 1-38 range, `home_team_id === away_team_id`) even though a database-level guarantee (a CHECK or UNIQUE constraint) already exists for each one.

[!\[\]\(43c95d78c31334a179428700d39030ee_img.jpg\) View solution](#)

EXERCISE 2

Explain why `usedTeamIds` is described as "convenience, not the real rule," and describe exactly what happens — on both the client and the server — if a stale page somehow submits a fixture for a team that's already been used in that gameweek.

[!\[\]\(346424ad7ee3a06d33d2a012aa61dfde_img.jpg\) View solution](#)

EXERCISE 3

Build the full click-to-pair page yourself against a real season and gameweek, enter all 10 fixtures for a gameweek, then explain what would actually happen if the Clear and Add Fixture buttons used inline `onclick` attributes instead of `addEventListener`, given how Astro processes this script by default.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **POST /api/seasons/{id}/gameweeks** — creates a gameweek (1-38); catches better-sqlite3's own `SqliteError` with code `'SQLITE_CONSTRAINT_UNIQUE'` as a friendly 409, no rollback step needed
- **POST /api/gameweeks/{id}/fixtures** — the real authority for "no team twice in a gameweek," checked by querying every existing fixture in that gameweek
- **Frontend** — a real Astro page with an inline, Astro-processed script; 20 always-visible team buttons instead of dropdowns
- **Genuine Astro gotcha** — a processed script runs as an ES module, so its functions aren't on window; inline `onclick` attributes silently fail, and `addEventListener` is the real, documented fix
- **Click-to-pair flow** — first click fills Home, second (different team) fills Away, then Add Fixture
- **Real limit** — client-side `usedTeamIds` resets on page reload; the server-side check in the fixture route is what actually prevents a bad duplicate
- **Next chapter:** Recording the user, expert, guest(s) & AI predictions per fixture

CHAPTER 5: PREDICTIONS: RECORDING THE USER, EXPERT, GUEST(S) & AI PREDICTIONS PER FIXTURE

Premier League Predictor: Astro

Chapter 5 · Predictions: Recording the User, Expert, Guest(s) & AI Predictions Per Fixture

Chapter 4 creates fixtures with `home_score` / `away_score` both still null. Before either of those gets filled in, four real sources each predict what they think will happen: the user, the BBC's expert, that gameweek's guest(s), and the BBC's own published AI prediction. This chapter builds the one table that records all four.

One Table, Four Sources

Three of the four sources — user, expert, AI — genuinely predict exactly once per fixture. The fourth, guest, is different by design: some weeks have one guest, some have several, and this app deliberately tracks every individual guest prediction rather than forcing them into one row before they've even been recorded.

```
-- src/lib/schema.sql (additions)
CREATE TABLE IF NOT EXISTS predictions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    fixture_id INTEGER NOT NULL REFERENCES fixtures(id),
    source TEXT NOT NULL CHECK (source IN ('user', 'expert', 'guest', 'ai')),
    guest_name TEXT,           -- set only when source = 'guest'
    predicted_home_score INTEGER NOT NULL,
    predicted_away_score INTEGER NOT NULL,
    created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

NO NATIVE ENUM TYPE EITHER — A CHECK CONSTRAINT DOES THE JOB INSTEAD

SQLAlchemy's own `Enum(PredictionSource)` column type has no direct SQLite equivalent, since SQLite genuinely has no dedicated enum column type at all — every column is fundamentally text, integer, real, or blob. `CHECK (source IN ('user', 'expert', 'guest', 'ai'))` gives an equivalent real guarantee at the database level: an attempt to insert any other value throws a real `SqliteError` with `code == 'SQLITE_CONSTRAINT_CHECK'`, exactly like the fixture chapter's own team-can't-play-itself rule.

A Partial Unique Index: Genuinely Supported by SQLite Too

A plain `UNIQUE (fixture_id, source)` would enforce "one prediction per source per fixture" — but it would apply to *every* source equally, including guest, breaking the whole point of allowing several distinctly-named guest predictions on the same fixture. What's actually needed is a unique rule that applies to three sources and deliberately doesn't apply to the fourth — a **partial unique index**:

```
-- src/lib/schema.sql (after the predictions table)
CREATE UNIQUE INDEX IF NOT EXISTS uq_prediction_single_source_per_fixture
ON predictions (fixture_id, source)
WHERE source != 'guest';
```

WORTH CORRECTING DIRECTLY: THIS ISN'T ACTUALLY POSTGRESQL-ONLY

The FastAPI & PostgreSQL course frames its own equivalent index as "a real PostgreSQL feature (SQLite has no equivalent)." Checked directly against SQLite's own official documentation, that's not accurate — partial indexes have been fully supported in SQLite since version 3.8.0, released in 2013, using nearly identical syntax: `CREATE UNIQUE INDEX ... ON table(columns) WHERE expr`, enforcing uniqueness only among the rows that satisfy the condition. The SQL above is, in fact, close to a direct, verbatim port of the PostgreSQL version, not a workaround for a missing capability. It's a useful reminder that "feature X belongs to database Y" claims are worth checking against the other database's own real documentation before repeating them, rather than assuming a more familiar system's own feature list is the complete picture.

Recording (and Correcting) a Prediction: an Upsert

A person should be able to change their mind about a prediction right up until kickoff — the route below looks for an existing prediction before deciding whether to update it or create a new one:

```
// src/pages/api/fixtures/[fixtureId]/predictions/index.ts
import type { APIRoute } from 'astro';
import { db } from '../../../lib/db';
import { json } from '../../../lib/http';
```

```

const VALID_SOURCES = ['user', 'expert', 'guest', 'ai'];

export const POST: APIRoute = async ({ params, request }) => {
  const fixtureId = Number(params.fixtureId);
  const { source, guest_name = null, predicted_home_score, predicted_away_score } = await request.json();

  const fixture = db.prepare('SELECT * FROM fixtures WHERE id = ?').get(fixtureId);
  if (!fixture) {
    return json({ error: 'Fixture not found' }, 404);
  }

  if (!VALID_SOURCES.includes(source)) {
    return json({ error: `source must be one of: ${VALID_SOURCES.join(', ')} `, 400);
  }

  const isGuest = source === 'guest';
  if (isGuest && !guest_name) {
    return json({ error: 'guest_name is required for a guest prediction' }, 400);
  }
  const resolvedGuestName = isGuest ? guest_name : null;

  const existing = isGuest
    ? db.prepare(
        'SELECT * FROM predictions WHERE fixture_id = ? AND source = ? AND guest_name = ?'
      ).get(fixtureId, source, resolvedGuestName)
    : db.prepare(
        'SELECT * FROM predictions WHERE fixture_id = ? AND source = ?'
      ).get(fixtureId, source);

  if (existing) {
    const id = (existing as any).id;
    db.prepare(
      'UPDATE predictions SET predicted_home_score = ?, predicted_away_score = ? WHERE id = ?'
    ).run(predicted_home_score, predicted_away_score, id);
    return json(db.prepare('SELECT * FROM predictions WHERE id = ?').get(id));
  }

  const result = db.prepare(
    'INSERT INTO predictions (fixture_id, source, guest_name, predicted_home_score, predicted_away_score) VALUES (?, ?, ?, ?, ?)'
  ).run(fixtureId, source, resolvedGuestName, predicted_home_score, predicted_away_score);

  const prediction = db.prepare('SELECT * FROM predictions WHERE id = ?').get(result.lastInsertId);

```

```

    return json(prediction, 201);
  };

export const GET: APIRoute = async ({ params }) => {
  const fixtureId = Number(params.fixtureId);
  const predictions = db.prepare('SELECT * FROM predictions WHERE fixture_id = ?').all();
  return json(predictions);
};

```

A DIRECT PAYOFF OF CHAPTER 2'S OWN SYNCHRONOUS-DATABASE FINDING

Between the `SELECT` that finds `existing` and the `UPDATE` / `INSERT` that follows it, there is no `await` anywhere — `better-sqlite3`'s own calls are genuinely synchronous, so this entire find-then-write sequence runs to completion in one uninterrupted step, with no possibility of another

async database driver, where an `await`ed query genuinely yields control back to the event loop, letting a second concurrent request's own find-then-write race in before the first one finishes. No transaction wrapper is needed around this upsert for exactly that reason — not because upserts never need one in general, but because this specific stack's own synchronous calls already rule out the very race Chapter 3's own `db.transaction()` exists to guard against.

WITHOUT THE "FIND FIRST" STEP, A SECOND SUBMISSION WOULD FAIL, NOT OVERWRITE

If this route simply inserted a new row on every call, submitting a second user prediction for the same fixture would collide directly with the partial unique index above — SQLite would reject it with a real `SqliteError`, `code === 'SQLITE_CONSTRAINT_UNIQUE'`, an unhelpful `500` unless caught. Checking for an existing row first, and updating it in place when one's found, is what actually lets someone correct a prediction before kickoff instead of just being told they can't submit again.

TURNING SEVERAL GUEST SCORELINES INTO ONE COMPARABLE FIGURE IS CHAPTER 6'S JOB

This chapter only records what each individual guest actually predicted — a real, separate row per guest, each with its own `guest_name`. Averaging multiple real scorelines (2-1 and 1-0 don't average into another valid scoreline) into the single "guest" figure the prediction league table eventually scores is a genuinely separate problem, deliberately left for Chapter 6, where results and scoring actually get built.

A Real Example Response

`GET /api/fixtures/42/predictions` for a fixture with two guests that week — the row from the database and the JSON response are already the same shape, with no reshaping step in between:

```
[
  { "id": 1, "fixture_id": 42, "source": "user", "guest_name": null, "predicted_home_score": 1.5, "predicted_away_score": 1.5 },
  { "id": 2, "fixture_id": 42, "source": "expert", "guest_name": null, "predicted_home_score": 1.5, "predicted_away_score": 1.5 },
  { "id": 3, "fixture_id": 42, "source": "guest", "guest_name": "Micah Richards", "predicted_home_score": 1.5, "predicted_away_score": 1.5 },
  { "id": 4, "fixture_id": 42, "source": "guest", "guest_name": "Jamie Carragher", "predicted_home_score": 1.5, "predicted_away_score": 1.5 },
  { "id": 5, "fixture_id": 42, "source": "ai", "guest_name": null, "predicted_home_score": 1.5, "predicted_away_score": 1.5 }
]
```

Recording Predictions From the Frontend

A compact page, reusable across all four sources — the guest form adds a name field, and as many guest rows as that week actually needs, with every interaction wired up via

`addEventListener` exactly the way Chapter 4 established, not inline `onclick` attributes:

```
---
// src/pages/admin/predictions.astro
---
<html lang="en">
  <body>
    <div>
      <label>Your prediction:
        <input type="number" id="user-home"> - <input type="number" id="user-away">
      </label>
      <button id="save-user">Save</button>
    </div>

    <div id="guest-rows"></div>
    <button id="add-guest-btn">Add Guest</button>

    <script>
      // In the real app, fixtureId comes from Chapter 10's own gameweek/season
```

```

// selector – hardcoded here to keep this example focused.
const fixtureId = 42;

async function submitPrediction(
  source: string,
  guestName: string | null,
  homeScore: string,
  awayScore: string
) {
  const res = await fetch(`/api/fixtures/${fixtureId}/predictions`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      source,
      guest_name: guestName || null,
      predicted_home_score: Number(homeScore),
      predicted_away_score: Number(awayScore),
    }),
  });

  if (!res.ok) {
    const error = await res.json();
    alert(error.error);
    return;
  }
  alert('Prediction saved.');
```

```

}

document.getElementById('save-user')!.addEventListener('click', () => {
  const home = (document.getElementById('user-home') as HTMLInputElement).value;
  const away = (document.getElementById('user-away') as HTMLInputElement).value;
  submitPrediction('user', null, home, away);
});

let guestRowCount = 0;

function addGuestRow() {
  guestRowCount += 1;
  const n = guestRowCount;
  const row = document.createElement('div');
  row.innerHTML = `
    <input type="text" placeholder="Guest name" id="guest-name-${n}">
  `
}

```

```

        <input type="number" placeholder="Home" id="guest-home-${n}">
        <input type="number" placeholder="Away" id="guest-away-${n}">
        <button>Save</button>
    `;
    row.querySelector('button')!.addEventListener('click', () => {
        const name = (document.getElementById(`guest-name-${n}`) as HTMLInputElement)
        const home = (document.getElementById(`guest-home-${n}`) as HTMLInputElement)
        const away = (document.getElementById(`guest-away-${n}`) as HTMLInputElement)
        submitPrediction('guest', name, home, away);
    });
    document.getElementById('guest-rows')!.appendChild(row);
}

document.getElementById('add-guest-btn')!.addEventListener('click', addGuestRow);
</script>
</body>
</html>

```

Every guest row gets its own `submitPrediction('guest', guestName, home, away)` call — each one a genuinely separate upsert, keyed by that specific guest's own name.

NO PREDICTION DEADLINE IS ENFORCED YET

A real prediction should probably stop being editable once the fixture actually kicks off — nothing in this chapter checks `fixture.kickoff_time` against the current time before accepting an upsert. That's an honest gap, not an oversight worth expanding this chapter to close; it's flagged here as a real candidate for a later refinement rather than pretended away.

Where This Course Is Headed

Entering real results — filling in the `home_score` / `away_score` this course's fixtures have carried as null since Chapter 4, and defining, at last, how a correct score and a correct result actually get calculated against every one of these four prediction sources, including how multiple guest predictions become one comparable figure (Chapter 6); the real league table (Chapter 7); the prediction league table, scoring every source this chapter records (Chapter 8); and promotion/relegation (Chapter 9).

Hands-On Exercises

EXERCISE 1

Explain why a plain UNIQUE (fixture_id, source) index wouldn't work for the predictions table, what the WHERE source != 'guest' clause actually changes about which rows the unique rule applies to, and correct the common assumption that a conditional partial index like this is a PostgreSQL-only feature.

 [View solution](#)

EXERCISE 2

Explain what would happen if the predictions route skipped the "find an existing row first" step and always ran a plain INSERT, for both a second user prediction on the same fixture and a second guest prediction from a guest who already predicted that fixture under the same name.

 [View solution](#)

EXERCISE 3

Record a user, an expert, an AI, and two separately-named guest predictions against a single real fixture, then call GET /api/fixtures/{id}/predictions and confirm all five rows come back with the correct source and guest_name values.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- **predictions** — one table, four sources (user/expert/guest/ai) validated by a CHECK constraint standing in for SQLite's missing ENUM type, keyed by fixture_id + source (+ guest_name for guests)
- **Partial unique index** — genuinely supported by SQLite since version 3.8.0 (2013), with syntax nearly identical to PostgreSQL's, correcting the sibling course's own "PostgreSQL-only" framing
- **POST /api/fixtures/{id}/predictions** — an upsert: finds an existing row for that source (and guest_name, if a guest) and updates it, or creates a new one
- **No transaction needed here** — better-sqlite3's synchronous calls mean the find-then-write sequence can't be interleaved by another request, unlike an async driver

- **guest_name** — required when source is guest, silently nulled for every other source
- **Still open** — no prediction deadline is enforced against kickoff_time yet
- **Deferred to Chapter 6** — how multiple guest scorelines become the single "guest" figure the prediction league table actually scores
- **Next chapter:** Entering results and calculating correct score vs. correct result

CHAPTER 6: ENTERING RESULTS & CALCULATING CORRECT SCORE VS. CORRECT RESULT

Premier League Predictor: Astro

Chapter 6 · Entering Results & Calculating Correct Score vs. Correct Result

Every fixture created back in Chapter 4 still carries `home_score` and `away_score` as `NULL`, and every prediction recorded in Chapter 5 has sat unscored ever since. This chapter closes that loop: a real result gets entered, and every prediction on that fixture is scored against it — using the exact point values the user confirmed directly while working through the FastAPI & PostgreSQL sibling course's own Chapter 6: **40 points for a correct score, 10 points for a correct result**.

Two Genuinely Different Kinds of "Correct"

A **correct score** means the exact scoreline matches — predict 2-1, the result is 2-1. A **correct result** means only the outcome matches — home win, away win, or draw — even if the scoreline itself is off. Every correct score is automatically a correct result too (2-1 is a home win, and if the real result is also a home win, the outcome trivially matches), but the reverse isn't true: predicting 3-0 for a match that finishes 1-0 gets the outcome right and the score wrong.

```
// src/lib/scoring.ts
export const POINTS_CORRECT_SCORE = 40;
export const POINTS_CORRECT_RESULT = 10;

type Outcome = 'HOME_WIN' | 'AWAY_WIN' | 'DRAW';

function outcome(homeScore: number, awayScore: number): Outcome {
  if (homeScore > awayScore) return 'HOME_WIN';
  if (awayScore > homeScore) return 'AWAY_WIN';
  return 'DRAW';
}

export function scorePrediction(
  predictedHome: number,
  predictedAway: number,
  actualHome: number,
  actualAway: number
```

```

): number {
  // Exact match checked FIRST – see the warn-box below for why order matters here.
  if (predictedHome === actualHome && predictedAway === actualAway) {
    return POINTS_CORRECT_SCORE;
  }
  if (outcome(predictedHome, predictedAway) === outcome(actualHome, actualAway)) {
    return POINTS_CORRECT_RESULT;
  }
  return 0;
}

```

CHECKING OUTCOME FIRST WOULD SILENTLY UNDER-SCORE EVERY EXACT PREDICTION

Because an exact scoreline match always implies an outcome match too, checking the outcome *before* the exact score and returning immediately on a match would mean `scorePrediction` never reaches the exact-match branch at all — a prediction of 2-1 against an actual result of 2-1 would be awarded 10 points (a correct result) instead of the 40 it actually earned. The exact check has to run first, and has to `return` before the outcome check ever gets a chance to fire.

A Real Gotcha While Adding the Column

`points_awarded` needs to live on the `predictions` table, nullable until a result is entered. The natural instinct is to add it to `schema.sql` alongside everything else — but every other statement in that file uses `CREATE TABLE IF NOT EXISTS`, and `ALTER TABLE ... ADD COLUMN` has no equivalent guard clause in SQLite at all. Since `db.ts` re-runs the whole schema file on every app startup, a bare `ALTER TABLE` statement would throw `SqliteError: duplicate column name: points_awarded` on the second run.

```

// src/lib/db.ts (addition, after db.exec(schema) runs)
const predictionColumns = db.prAGMA('table_info(predictions)') as { name: string }[];
const hasPointsAwarded = predictionColumns.some((col) => col.name === 'points_awarded')

if (!hasPointsAwarded) {
  db.exec('ALTER TABLE predictions ADD COLUMN points_awarded INTEGER');
}

```

CREATE TABLE IF NOT EXISTS HAS NO ADD COLUMN COUNTERPART

Checked directly against SQLite's own documented `ALTER TABLE` grammar: it supports `RENAME TO`, `RENAME COLUMN`, `ADD COLUMN`, and `DROP COLUMN` — none of them accept an `IF NOT EXISTS` clause the way `CREATE TABLE` does. A genuinely idempotent column addition has to check `PRAGMA table_info` for the column's own presence first, exactly the pattern above. This is a real, easy trap for anyone assuming every SQLite DDL statement is as forgiving about being run twice as `CREATE TABLE IF NOT EXISTS` is.

Entering a Result: A Genuine `db.transaction()` Use Case

Chapter 5's own upsert route needed no transaction wrapper, because `better-sqlite3`'s synchronous calls already ruled out another request interleaving mid-sequence. Entering a result is a different situation: it coordinates one `UPDATE` on `fixtures` with potentially several `UPDATE`s on `predictions` — one per recorded prediction — and these writes genuinely need to succeed or fail together. If the fixture's own score updated but a crash interrupted the loop halfway through rescoreing predictions, the database would be left in a state where some predictions reflect the new result and others still carry stale points from before.

```
// src/pages/api/fixtures/[fixtureId]/result/index.ts
import type { APIRoute } from 'astro';
import { db } from '../../../lib/db';
import { json } from '../../../lib/http';
import { scorePrediction } from '../../../lib/scoring';

export const PATCH: APIRoute = async ({ params, request }) => {
  const fixtureId = Number(params.fixtureId);
  const { home_score, away_score } = await request.json();

  if (!Number.isInteger(home_score) || !Number.isInteger(away_score) || home_score < 0 || away_score < 0) {
    return json({ error: 'home_score and away_score must both be non-negative integers' }, 400);
  }

  const fixture = db.prepare('SELECT * FROM fixtures WHERE id = ?').get(fixtureId);
  if (!fixture) {
    return json({ error: 'Fixture not found' }, 404);
  }

  db.transaction(() => {
    db.prepare('UPDATE fixtures SET home_score = ?, away_score = ? WHERE id = ?').run(home_score, away_score, fixtureId);

    const predictions = db.prepare('SELECT * FROM predictions WHERE fixture_id = ?').all(fixtureId);
    for (const prediction of predictions) {
      scorePrediction(prediction, home_score, away_score);
    }
  });
}
```

```

const applyResult = db.transaction((homeScore: number, awayScore: number) => {
  db.prepare(
    'UPDATE fixtures SET home_score = ?, away_score = ?, status = ? WHERE id = ?'
  ).run(homeScore, awayScore, 'finished', fixtureId);

  // Every prediction on this fixture is rescored from scratch on every call –
  // not just the first time a result is entered. See the tip-box below.
  const predictions = db.prepare('SELECT * FROM predictions WHERE fixture_id = ?').all
  for (const prediction of predictions as any[]) {
    const points = scorePrediction(
      prediction.predicted_home_score,
      prediction.predicted_away_score,
      homeScore,
      awayScore
    );
    db.prepare('UPDATE predictions SET points_awarded = ? WHERE id = ?').run(points,
  }
});

applyResult(home_score, away_score);

return json(db.prepare('SELECT * FROM fixtures WHERE id = ?').get(fixtureId));
};

```

A DIRECT CONTRAST WITH CHAPTER 5'S OWN NO-TRANSACTION FINDING

Chapter 5 needed no `db.transaction()` because it was a single find-then-write sequence with no other request able to interleave inside it — the synchronous nature of `better-sqlite3` already made it atomic in every way that mattered. This route is different: it's coordinating *multiple, separate writes* — one fixture update plus one prediction update per recorded prediction — that together represent a single logical operation. `db.transaction()` here isn't guarding against another request racing in; it's guarding against a crash or thrown error leaving the fixture's own score updated while only some of its predictions have been rescored. Wrapping the whole callback means either every one of those writes commits, or none of them do.

RE-SCORING IS DELIBERATE, NOT WASTED WORK

The `PATCH` route above re-reads and re-scores *every* prediction on the fixture each time it's called — even predictions that were already scored against a previous, since-corrected result. That's intentional: a result entered in error and then corrected needs every affected prediction's

`points_awarded` to reflect the corrected result, not the original mistake. Scoring only newly-added predictions would leave stale points sitting on rows from before the correction.

A Real Result Applied to Chapter 5's Own Example

Chapter 5 closed with five real predictions recorded against fixture 42: a user prediction of 2-1, an expert prediction of 1-1, two guest predictions (Micah Richards at 3-0, Jamie Carragher at 1-0), and an AI prediction of 2-0. A real `PATCH /api/fixtures/42/result` call with `{ "home_score": 2, "away_score": 1 }` — the match actually finishing 2-1 — scores all five:

SOURCE	PREDICTED	ACTUAL	MATCH TYPE	POINTS
user	2-1	2-1	Exact score	40
expert	1-1	2-1	Draw predicted, home win actual — neither	0
guest — Micah Richards	3-0	2-1	Home win predicted, home win actual	10
guest — Jamie Carragher	1-0	2-1	Home win predicted, home win actual	10
ai	2-0	2-1	Home win predicted, home win actual	10

Resolving Chapter 5's Own Deferred Question: Averaging Guest Points

Chapter 5 deliberately left one question open: with a variable number of guests each recording their own scoreline, how does the prediction league table turn several guest rows into a single comparable "guest" figure for that fixture? Averaging the raw scorelines themselves doesn't work — 3-0 and 1-0 don't average into another valid scoreline, and even if they did, a fractional goal count is meaningless. What genuinely averages is the **points each guest earned**, since a point value is always a plain number regardless of how many guests predicted or what they each guessed.

In this fixture's own example, both guests happened to earn 10 points each, so the averaged guest figure for fixture 42 is also 10. That's not a coincidence worth reading too much into — it's just what this particular result produced. A fixture where one guest predicted the exact score (40 points) and a second guest got only the outcome right (10 points) would average to

$(40 + 10) / 2 = 25$ — a number that never appears as anyone's actual `points_awarded`, but is exactly the figure the prediction league table needs to compare "guest" against "user," "expert," and "ai" on equal footing.

THE ACTUAL AGGREGATION QUERY LIVES IN CHAPTER 8

This chapter establishes the *principle* — average points, not scorelines — and the real per-prediction `points_awarded` values this route calculates are exactly what that later average will be computed from. The SQL that actually groups every guest prediction by fixture and season to build the full prediction league table is Chapter 8's own job, once there's a real season's worth of scored fixtures to aggregate across rather than one single example.

Where This Course Is Headed

The real league table, built from every fixture's own actual result rather than any prediction (Chapter 7); the prediction league table, finally aggregating the `points_awarded` values this chapter calculates — including the guest-averaging principle established above — across a full season (Chapter 8); promotion and relegation (Chapter 9); Astro Islands and interactivity (Chapter 10); deployment (Chapter 11); and the capstone (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain, with a concrete example, why `scorePrediction()` checks for an exact score match before checking for a correct-result match, and what would go wrong if the outcome check ran first and returned immediately on a match.

 [View solution](#)

EXERCISE 2

A fixture has three guest predictions, scored (after a result is entered) at 40, 10, and 0 points respectively. Explain why the prediction league table should average these point values rather than averaging the guests' raw scorelines, and compute the correct averaged guest figure for this fixture.

[View solution](#)

EXERCISE 3

Enter an incorrect result for a fixture with at least two recorded predictions, confirm both predictions' `points_awarded` values, then call `PATCH /api/fixtures/{id}/result` again with the corrected result and confirm that every prediction's `points_awarded` actually changes to reflect the correction — not just predictions added after the fix.

[View solution](#)

CHAPTER 6 QUICK REFERENCE

- **POINTS_CORRECT_SCORE = 40, POINTS_CORRECT_RESULT = 10** — the real values confirmed by the user during the FastAPI & PostgreSQL sibling course's own Chapter 6
- **scorePrediction()** — checks exact score match first (returns 40), then outcome match (returns 10), else 0 — the order matters, since an exact match always implies an outcome match too
- **points_awarded** — a nullable INTEGER column added to predictions via a guarded ALTER TABLE, since SQLite's own ADD COLUMN has no IF NOT EXISTS clause the way CREATE TABLE does
- **PATCH /api/fixtures/{id}/result** — updates the fixture's score/status and rescores every prediction on it, wrapped in `db.transaction()` since it's coordinating multiple writes as one logical operation
- **A genuine db.transaction() use case** — contrasted directly against Chapter 5's own no-transaction upsert: this route needs atomicity across several writes, not protection from request interleaving
- **Re-scoring is total, not incremental** — every prediction is recalculated on every PATCH call, so a corrected result also corrects every prediction's own points
- **Guest averaging resolved** — average each guest's own points, never their raw scorelines, since points are always a plain comparable number
- **Next chapter:** The real league table — points, goal difference, won/drawn/lost

CHAPTER 7: THE REAL LEAGUE TABLE: POINTS, GOAL DIFFERENCE, WON/DRAWN/LOST

Premier League Predictor: Astro

Chapter 7 · The Real League Table: Points, Goal Difference, Won/Drawn/Lost

Every fixture entered via Chapter 6 now carries a real `home_score` / `away_score` once played. This chapter turns those results into the table anyone actually recognizes as "the league table" — points, goal difference, won/drawn/lost — built entirely from real results, with no prediction data involved at all. That's a genuinely separate thing from the **prediction** league table Chapter 8 builds next, which ranks how well the user, expert, guests, and AI predicted, not how the real teams actually performed.

The Real Problem: A Team Is Sometimes Home, Sometimes Away

`fixtures` stores exactly one row per match, with `home_team_id` / `home_score` on one side and `away_team_id` / `away_score` on the other. But a league table needs a row *per team*, not per fixture — and a given team's own goals-for and goals-against sit in different columns depending on whether that team happened to be playing at home or away in a particular fixture. A single fixture row can't be read directly as "one team's result"; it has to be looked at twice, once from each side.

Normalizing Home and Away Rows With UNION ALL

The real fix is to turn every fixture into two rows — one from the home team's own perspective, one from the away team's — each carrying just `team_id`, `goals_for`, and `goals_against` in the same shape regardless of which side that team was actually on. SQLite's own `UNION ALL` does exactly this, and a `WITH` clause names the combined result so the rest of the query can treat it as one ordinary table:

```
-- The normalizing CTE – every played fixture becomes two rows, one per side
WITH team_fixture_stats AS (
  SELECT
    f.home_team_id AS team_id,
    f.home_score AS goals_for,
    f.away_score AS goals_against
```

```

FROM fixtures f
JOIN gameweeks g ON g.id = f.gameweek_id
WHERE g.season_id = ? AND f.status = 'played'

UNION ALL

SELECT
  f.away_team_id AS team_id,
  f.away_score AS goals_for,
  f.home_score AS goals_against
FROM fixtures f
JOIN gameweeks g ON g.id = f.gameweek_id
WHERE g.season_id = ? AND f.status = 'played'
)

```

Once `team_fixture_stats` exists, every row in it already answers "how did this one team do in this one fixture," in the same three columns no matter which side of the original fixture it came from. Aggregating it per team from here is ordinary `GROUP BY` arithmetic.

Starting From `season_teams`, Not From Fixtures

If the outer query started from `team_fixture_stats` and grouped by `team_id`, a team with genuinely zero played fixtures — the very first gameweek of a season, before anything has kicked off — would simply be absent from the result entirely, since there'd be no rows for it to group. A real league table shouldn't *omit* a team that hasn't played yet; it should show that team with 0 played, 0 points, sitting at the bottom alongside everyone else who also hasn't played. The fix is the same one Chapter 3's own `GET /api/seasons/{id}/teams` route already leaned on: start from **membership** — `season_teams` — and `LEFT JOIN` the stats onto it, rather than starting from the stats and hoping every team happens to already be represented in them.

```

-- src/lib/leagueTable.ts
import { db } from './db';

export interface LeagueTableRow {
  team_id: number;
  name: string;
  short_name: string;

```

```

    played: number;
    won: number;
    drawn: number;
    lost: number;
    goals_for: number;
    goals_against: number;
    goal_difference: number;
    points: number;
}

const LEAGUE_TABLE_SQL = `
    WITH team_fixture_stats AS (
        SELECT f.home_team_id AS team_id, f.home_score AS goals_for, f.away_score AS goals_
        FROM fixtures f
        JOIN gameweeks g ON g.id = f.gameweek_id
        WHERE g.season_id = ? AND f.status = 'played'

        UNION ALL

        SELECT f.away_team_id AS team_id, f.away_score AS goals_for, f.home_score AS goals_
        FROM fixtures f
        JOIN gameweeks g ON g.id = f.gameweek_id
        WHERE g.season_id = ? AND f.status = 'played'
    )
    SELECT
        t.id AS team_id,
        t.name,
        t.short_name,
        COUNT(tfs.team_id) AS played,
        COALESCE(SUM(CASE WHEN tfs.goals_for > tfs.goals_against THEN 1 ELSE 0 END), 0) AS
        COALESCE(SUM(CASE WHEN tfs.goals_for = tfs.goals_against THEN 1 ELSE 0 END), 0) AS
        COALESCE(SUM(CASE WHEN tfs.goals_for < tfs.goals_against THEN 1 ELSE 0 END), 0) AS
        COALESCE(SUM(tfs.goals_for), 0) AS goals_for,
        COALESCE(SUM(tfs.goals_against), 0) AS goals_against,
        COALESCE(SUM(tfs.goals_for - tfs.goals_against), 0) AS goal_difference,
        COALESCE(SUM(
            CASE
                WHEN tfs.goals_for > tfs.goals_against THEN 3
                WHEN tfs.goals_for = tfs.goals_against THEN 1
                ELSE 0
            END
        ), 0) AS points

```

```

FROM season_teams st
JOIN teams t ON t.id = st.team_id
LEFT JOIN team_fixture_stats tfs ON tfs.team_id = t.id
WHERE st.season_id = ?
GROUP BY t.id, t.name, t.short_name
ORDER BY points DESC, goal_difference DESC, goals_for DESC
;

```

```

export function getLeagueTable(seasonId: number): LeagueTableRow[] {
  // seasonId is bound three times — twice inside the CTE's own UNION ALL, once in the outer WHERE
  return db.prepare(LEAGUE_TABLE_SQL).all(seasonId, seasonId, seasonId) as LeagueTableRow[]
}

```

THE SAME SEASONID PARAMETER, BOUND THREE SEPARATE TIMES

`?` placeholders in `better-sqlite3` are positional, not named — each one is a genuinely separate slot that needs its own bound value, even when the actual value happens to be identical every time. This query has three `?` placeholders (two inside the CTE's two `WHERE g.season_id = ?` clauses, one in the outer `WHERE st.season_id = ?`), so `.all(seasonId, seasonId, seasonId)` passes the same

real `RangeError: too few parameter values were provided` the moment the statement is executed.

A Real Consequence of Never Having an ORM Here

The Django & MySQL sibling course found a genuinely severe bug attempting this exact same aggregation: annotating `Sum()` across two separate reverse relations — `home_fixtures` and `away_fixtures` — in a single Django ORM call silently cross-joins them, inflating a true 14 goals into a wrong 56 in that chapter's own worked example. The fix there was to drop to `Team.objects.raw()` and hand-write the identical `UNION ALL` shape built above.

THIS COURSE WAS NEVER AT RISK OF THAT BUG IN THE FIRST PLACE

This stack has no ORM query-generation layer sitting between the code above and the SQL that actually runs — the `UNION ALL` / `WITH` query *is* the code, typed out by hand, string for string. The Django bug happened because its ORM tried to combine two separate annotated aggregates automatically and generated an unintended cross join while doing it; there's no equivalent automatic-join-generation step here that could go wrong the same way, because nothing here is generated at all. This isn't "SQLite is smarter than Django's ORM" — it's that a hand-written query

has no automatic translation step standing between the intent and the SQL that could introduce a bug neither the intent nor the SQL itself contains.

A Real, Verified SQLite Version Check

Chapter 5 verified directly against SQLite's own documentation that partial indexes have been supported since SQLite 3.8.0, released in August 2013. The `WITH` clause this chapter's own query depends on is a genuinely close neighbor in SQLite's real release history:

WITH CLAUSE SUPPORT: SQLITE 3.8.3, RELEASED 2014-02-03

Checked directly against SQLite's own official release log for version 3.8.3: its first listed feature addition reads "Added support for common table expressions and the WITH clause." That release landed about six months after 3.8.0's own partial-index support — both genuinely old, stable features by any modern measure, not something requiring an unusually recent SQLite build to rely on.

The Real League Table API Route

```
// src/pages/api/seasons/[seasonId]/table.ts
import type { APIRoute } from 'astro';
import { db } from '../../lib/db';
import { json } from '../../lib/http';
import { getLeagueTable } from '../../lib/leagueTable';

export const GET: APIRoute = async ({ params }) => {
  const seasonId = Number(params.seasonId);

  const season = db.prepare('SELECT * FROM seasons WHERE id = ?').get(seasonId);
  if (!season) {
    return json({ error: 'Season not found' }, 404);
  }

  return json(getLeagueTable(seasonId));
};
```

Splitting the query itself into `src/lib/leagueTable.ts`, separate from the route file that calls it, is deliberate: Chapter 9's own promotion/relegation logic needs the identical league table — specifically to find the bottom three teams — and importing `getLeagueTable()` there is simpler and less error-prone than either duplicating the whole query a second time or making one API route call another one internally.

A Worked Example

Three played fixtures, one unplayed, across a four-team mini-season: Team A beat Team B 3-1, Team C drew Team D 1-1, and Team A beat Team D 2-0. Team B and Team C haven't yet played each other.

TEAM	P	W	D	L	GF	GA	GD	PTS
Team A	2	2	0	0	5	1	+4	6
Team C	1	0	1	0	1	1	0	1
Team D	2	0	1	1	1	3	-2	1
Team B	1	0	0	1	1	3	-2	0

Team C and Team D are genuinely tied on points, and this query's own `ORDER BY` resolves it purely by goal difference (0 beats -2) — Team D's own single win over nobody doesn't come into it, since goal difference is the very next tiebreaker after points, ahead of goals scored.

REAL TIEBREAKERS THIS QUERY DOESN'T IMPLEMENT

A genuine Premier League table also uses head-to-head results as a real tiebreaker in some situations, and has historically applied a fair-play points deduction in others. Neither is implemented here — `ORDER BY points DESC, goal_difference DESC, goals_for DESC` covers the three criteria that decide the overwhelming majority of real placings, and is an honest, deliberately incomplete stand-in for the full, more intricate real rule set rather than a claim that every possible tie is resolved exactly the way the real Premier League would resolve it.

Where This Course Is Headed

The prediction league table — a genuinely different aggregation, over `points_awarded` rather than real results, with Chapter 6's own guest-averaging principle finally put to use across a real season (Chapter 8); promotion and relegation, reusing `getLeagueTable()`'s own output directly to find the real bottom three (Chapter 9); Astro Islands and interactivity (Chapter 10); deployment (Chapter 11); and the capstone (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a single fixtures row can't be read directly as one team's own league-table result, and describe exactly what the UNION ALL inside `team_fixture_stats` does to fix that.

 [View solution](#)

EXERCISE 2

Explain why the league-table query starts from `season_teams` with a LEFT JOIN onto `team_fixture_stats`, rather than starting from `team_fixture_stats` itself and grouping by `team_id`, and describe concretely what would go wrong for a team with zero played fixtures under the second approach.

 [View solution](#)

EXERCISE 3

Set up the four-team, three-fixture mini-season from this chapter's own worked example against a real database, call GET `/api/seasons/{seasonId}/table`, and confirm the returned table matches the worked-example rows exactly — including the Team C vs. Team D tie being broken by goal difference rather than points.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **The real problem** — a team's own goals-for/against live in different columns depending on home vs. away, so one fixtures row can't be read as one team's result directly
- **UNION ALL normalization** — team_fixture_stats turns every played fixture into two rows, one per side, in the same team_id/goals_for/goals_against shape
- **Start from season_teams, LEFT JOIN the stats** — echoing Chapter 3's own membership-first reasoning, so a team with zero played fixtures still shows a real 0-played row instead of being omitted
- **No ORM cross-join bug possible here** — a direct contrast with the Django & MySQL sibling's own real Sum()-over-two-reverse-relations bug (14 goals inflated to 56); there's no automatic join-generation layer in this stack that could make the same mistake
- **WITH clause support: SQLite 3.8.3 (2014-02-03)** — verified directly against SQLite's own release log, landing about six months after Chapter 5's own verified 3.8.0 partial-index support
- **seasonId bound three times** — twice inside the CTE's own UNION ALL, once in the outer WHERE; better-sqlite3's ? placeholders are positional, not named
- **getLeagueTable() lives in its own module** — reused directly by Chapter 9's promotion/relegation logic, not duplicated
- **ORDER BY points DESC, goal_difference DESC, goals_for DESC** — real head-to-head and fair-play tiebreakers are honestly not implemented
- **Next chapter:** The prediction league table — aggregating predictor performance across a full season

CHAPTER 8: THE PREDICTION LEAGUE TABLE: AGGREGATING PREDICTOR PERFORMANCE ACROSS THE SEASON

Premier League Predictor: Astro

Chapter 8 · The Prediction League Table: Aggregating Predictor Performance Across the Season

Chapter 7's real league table ranks the 20 actual teams by what they actually did on the pitch. This table asks a genuinely different question: across the whole season, which of the four prediction sources — user, expert, guest, AI — has actually been the best predictor? Every fixture already carries a real `points_awarded` value per prediction since Chapter 6; this chapter sums it correctly across an entire season.

Pass One: Averaging Guest Points Per Fixture

Chapter 6 established the real principle — a fixture with several guest predictions gets one comparable figure by averaging each guest's own `points_awarded`, never their raw scorelines. That averaging has to happen once per fixture, before anything gets summed across the season, or a fixture with three guests would otherwise be counted three separate times against a fixture with only one:

```
-- Pass 1: one averaged "guest" points figure per fixture
WITH guest_fixture_points AS (
  SELECT
    fixture_id,
    AVG(points_awarded) AS points_awarded
  FROM predictions
  WHERE source = 'guest' AND points_awarded IS NOT NULL
  GROUP BY fixture_id
)
```

`points_awarded IS NOT NULL` excludes any fixture that hasn't had a result entered yet — Chapter 6's own `PATCH` route only fills that column in once a fixture is marked `'played'`, so an unplayed fixture's guest predictions simply don't contribute to anyone's total yet, on either side of this table.

Pass Two: One Shared Shape for Every Source

User, expert, and AI predictions already sit one row per fixture in `predictions` — no averaging needed. A second `UNION ALL` puts those rows alongside the newly-averaged guest rows from Pass One, in the same `(fixture_id, source, points_awarded)` shape:

```
-- Pass 2: every source, one row per fixture, in the same shape
source_fixture_points AS (
  SELECT fixture_id, source, points_awarded
  FROM predictions
  WHERE source != 'guest' AND points_awarded IS NOT NULL

  UNION ALL

  SELECT fixture_id, 'guest' AS source, points_awarded
  FROM guest_fixture_points
)
```

NO NULL-TYPING QUESTION TO RUN INTO HERE AT ALL

Both siblings' own equivalent queries had to carry each prediction's real `predicted_home_score` / `predicted_away_score` through their own `UNION ALL`, needing a placeholder `NULL` in whichever branch didn't have a real scoreline to offer for the averaged guest row — the FastAPI & PostgreSQL course needed an explicit `NULL::int` cast there, and the Django & MySQL course found MySQL infers the same bare `NULL`'s type automatically. This query never runs into that question at all, for a much simpler reason than either database's own typing rules: because Chapter 6 already computed and stored a real `points_awarded` value on every prediction row, both branches of this `UNION ALL` select a genuine number — there's no column here that only has meaning for one branch and needs a placeholder in the other.

SQLITE WOULDN'T HAVE NEEDED THE CAST EVEN IF A NULL WERE INVOLVED

Checked directly against SQLite's own documented compound-SELECT rules: "no affinity transformations are applied to any values when comparing rows as part of a compound SELECT." SQLite's own columns are dynamically typed to begin with — there's no fixed column type for a `UNION`'s own result set the way PostgreSQL enforces one, so a bare, untyped `NULL` sitting in one branch next to a real integer in another would simply work, with nothing to cast in the first place. This

isn't a case of SQLite being cleverer than either sibling database — it genuinely never asks the question those two answer in opposite ways.

Aggregating: Totals, and an Honestly Nulled Breakdown

The outer query sums points per source across the whole season, and separately counts how many of those points came from an exact score (40) versus a correct result (10) — but only for the three sources where that breakdown actually means something. A guest's own row is an *average*, and an average was never really 40, 10, or 0 to begin with, so counting how many "guest" rows equal exactly 40 would be a real number that answers a question nobody actually asked.

```
// src/lib/predictionTable.ts
import { db } from './db';

export interface PredictionTableRow {
  source: string;
  fixtures_scored: number;
  total_points: number;
  correct_scores: number | null;
  correct_results: number | null;
}

const PREDICTION_TABLE_SQL = `
  WITH guest_fixture_points AS (
    SELECT fixture_id, AVG(points_awarded) AS points_awarded
    FROM predictions
    WHERE source = 'guest' AND points_awarded IS NOT NULL
    GROUP BY fixture_id
  ),
  source_fixture_points AS (
    SELECT fixture_id, source, points_awarded
    FROM predictions
    WHERE source != 'guest' AND points_awarded IS NOT NULL

    UNION ALL

    SELECT fixture_id, 'guest' AS source, points_awarded
    FROM guest_fixture_points
  )
```

```

)
SELECT
  sfp.source,
  COUNT(*) AS fixtures_scored,
  SUM(sfp.points_awarded) AS total_points,
  CASE WHEN sfp.source = 'guest' THEN NULL
        ELSE SUM(CASE WHEN sfp.points_awarded = 40 THEN 1 ELSE 0 END)
  END AS correct_scores,
  CASE WHEN sfp.source = 'guest' THEN NULL
        ELSE SUM(CASE WHEN sfp.points_awarded = 10 THEN 1 ELSE 0 END)
  END AS correct_results
FROM source_fixture_points sfp
JOIN fixtures f ON f.id = sfp.fixture_id
JOIN gameweeks g ON g.id = f.gameweek_id
WHERE g.season_id = ?
GROUP BY sfp.source
ORDER BY total_points DESC
`;

export function getPredictionTable(seasonId: number): PredictionTableRow[] {
  return db.prepare(PREDICTION_TABLE_SQL).all(seasonId) as PredictionTableRow[];
}

```

THE INNER SUM HAS TO STAY HONEST EVEN FOR SOURCES IT'S ABOUT TO NULL OUT

`CASE WHEN sfp.points_awarded = 40 THEN 1 ELSE 0 END` is deliberately written to sum real 1s and 0s, not 1s and `NULL`s — that's what makes a genuine "zero correct scores this season" come back as a real `0` for user, expert, or AI, rather than being indistinguishable from a group with no rows at all. The outer `CASE WHEN sfp.source = 'guest' THEN NULL` is what actually does the nulling, and it's applied once, after the real count has already been computed correctly — not baked into the counting logic itself, where it would have wrongly turned every real "user got 0 correct scores" result into a meaningless `NULL` too.

A Single-parameter Query, Unlike Chapter 7's Own Three

`seasonId` is bound exactly once here, in the outer `WHERE g.season_id = ?` — a genuine contrast with Chapter 7's league-table query, which needed the identical value bound three separate times across its own `UNION ALL`. The difference is structural: Chapter 7's `UNION ALL` filters by season inside *both* of its own branches (home rows and away rows each need the

same season filter applied independently), while this chapter's season filter is applied once, in the outer query, after both CTEs have already produced their rows — there's simply nowhere else in this particular query that `season_id` needs to be checked a second time.

A Real API Route

```
// src/pages/api/seasons/[seasonId]/prediction-table.ts
import type { APIRoute } from 'astro';
import { db } from '../../lib/db';
import { json } from '../../lib/http';
import { getPredictionTable } from '../../lib/predictionTable';

export const GET: APIRoute = async ({ params }) => {
  const seasonId = Number(params.seasonId);

  const season = db.prepare('SELECT * FROM seasons WHERE id = ?').get(seasonId);
  if (!season) {
    return json({ error: 'Season not found' }, 404);
  }

  return json(getPredictionTable(seasonId));
};
```

A Worked Example, Extending Chapter 6's Own Fixture 42

Fixture 42, scored back in Chapter 6: user 40, expert 0, both guests scored 10 (averaging to exactly 10), AI 10. A second fixture, 43, later the same season: user 0, expert 40, three guests scoring 40, 10, and 0 (averaging to `50 / 3 = 16.67` — the exact figure Chapter 6's own Exercise 2 already computed by hand), AI 10.

SOURCE	FIXTURES SCORED	TOTAL POINTS	CORRECT SCORES	CORRECT RESULTS
user	2	40	1	0
expert	2	40	1	0
guest	2	26.67	<i>null</i>	<i>null</i>

SOURCE	FIXTURES SCORED	TOTAL POINTS	CORRECT SCORES	CORRECT RESULTS
ai	2	20	0	2

user and expert genuinely tie on 40 total points, arrived at through opposite fixtures — user nailed fixture 42 and missed fixture 43 entirely, expert did the reverse. Both really did get exactly one correct score across the two fixtures, and the table shows that plainly rather than hiding it behind the tied total.

TIES HAVE NO TIEBREAKER IN THIS QUERY

`ORDER BY total_points DESC` is the query's only sort key. When two sources tie exactly, as user and expert do above, which one appears first in the returned array is whatever order SQLite happens to produce them in — not a documented, guaranteed ordering, and not something this chapter resolves with a second sort key the way Chapter 7's real league table resolves its own ties with goal difference. A real UI built on this route should treat a tie as a tie visually (e.g. showing both at "1st"), rather than implying one source is definitively ranked above the other.

Where This Course Is Headed

Promotion and relegation, reusing Chapter 7's own `getLeagueTable()` directly to find the real bottom three (Chapter 9); Astro Islands and interactivity (Chapter 10); deployment (Chapter 11); and the capstone (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain why `guest_fixture_points` has to run as its own separate pass before the `UNION ALL` in `source_fixture_points`, rather than averaging guest points and unioning everything together in one single query.

 [View solution](#)

EXERCISE 2

Explain why `correct_scores` and `correct_results` are computed as a real SUM of 1s and 0s first, and only nulled out afterward by an outer CASE, rather than writing the null-for-guest logic directly into the inner CASE that does the counting.

 [View solution](#)

EXERCISE 3

Set up fixtures 42 and 43 from this chapter's own worked example against a real database, call GET `/api/seasons/{seasonId}/prediction-table`, and confirm the response matches the worked-example table exactly, including guest's own averaged total of 26.67 and null `correct_scores/correct_results` for guest specifically.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **Two passes, not one** — `guest_fixture_points` averages guest points per fixture first, then `source_fixture_points` UNION ALLs every source into one shared (`fixture_id`, `source`, `points_awarded`) shape
- **No NULL-typing question here** — unlike both sibling courses' own placeholder-NULL predicted-score columns, this query's two UNION ALL branches both select a real, already-computed `points_awarded` value
- **SQLite wouldn't have required a cast anyway** — verified directly against SQLite's own compound-SELECT documentation: no affinity transformations are applied when comparing rows in a UNION, unlike PostgreSQL's own stricter typing
- **Honest correct_scores/correct_results** — computed as a genuine SUM of 1s and 0s first, then nulled specifically for the guest source by an outer CASE, so a real "zero correct scores" for user/expert/ai is never confused with "this number doesn't apply"
- **One bound parameter, not three** — `season_id` is filtered once, in the outer query, unlike Chapter 7's own three-times-bound league-table query
- **getPredictionTable()** — GET `/api/seasons/{id}/prediction-table`, the direct counterpart to Chapter 7's `getLeagueTable()`
- **Real ties are just ties** — ORDER BY `total_points` DESC has no second sort key; a tied pair's own returned order isn't guaranteed or documented

- **Next chapter:** Promotion & relegation — reusing Chapter 7's own league table to find the real bottom three

CHAPTER 9: PROMOTION & RELEGATION: AUTO-CALCULATING THE BOTTOM THREE, MANUALLY ENTERING THE PROMOTED THREE

Premier League Predictor: Astro

Chapter 9 · Promotion & Relegation: Auto-Calculating the Bottom Three, Manually Entering the Promoted Three

A real Premier League season ends with three of its 20 teams relegated and three new ones promoted up from the Championship. Because `season_teams` is scoped per season (Chapter 2), "next season's 20 teams" is really just a brand-new set of `season_teams` rows — this chapter is about deciding, and inserting, exactly which `team_ids` those rows should point at.

The Bottom Three Are Auto-Calculated — No Refactor Required

Both sibling courses reach this chapter needing to first refactor their own Chapter 7 league-table logic out of an inline route and into a separate, reusable function before they can call it a second time here. This course never has that step to do: Chapter 7's own `getLeagueTable()` was written into `src/lib/leagueTable.ts` from the very start, with a tip-box at the time explicitly naming this exact reuse as the reason. Finding the real bottom three is nothing more than calling the same function again, now with a different question in mind:

```
// src/lib/rollover.ts
import { db } from './db';
import { getLeagueTable, type LeagueTableRow } from './leagueTable';

const TEAMS_PER_SEASON = 20;
const RELEGATED_COUNT = 3;

export function getRelegated(seasonId: number): LeagueTableRow[] {
  const table = getLeagueTable(seasonId);
  return table.slice(-RELEGATED_COUNT);
}

export function getSurvivors(seasonId: number): LeagueTableRow[] {
  const table = getLeagueTable(seasonId);
  return table.slice(0, -RELEGATED_COUNT);
}
```

SLICE(-3) AND SLICE(0, -3) ALWAYS MEAN "THE LAST THREE" AND "EVERYTHING BUT THE LAST THREE"

Because `getLeagueTable()` already returns its rows sorted `ORDER BY points DESC, goal_difference DESC, goals_for DESC`, the worst-performing team is always the very last element of the returned array, regardless of how many teams are actually in it. `Array.prototype.slice(-3)` reads as "the last three elements" and `slice(0, -3)` as "every element except the last three" — for a genuine 20-team table, that's positions 18-20 and positions 1-17 respectively, with no index arithmetic to get wrong by hand.

THIS ASSUMES THE SEASON ACTUALLY FINISHED WITH ALL 20 TEAMS INTACT

Nothing in `getRelegated()` or `getSurvivors()` checks that a season is actually over, or that every one of its 20 teams played a full 38-gameweek schedule. The rollover route below adds exactly one real safeguard — checking the table has exactly 20 rows before proceeding — but doesn't verify anything about gameweeks played. Calling this mid-season would return a real, well-formed answer to the wrong question, not an error.

Rolling the Survivors Forward: An Ordinary Loop, Not a Bulk Insert

The Django & MySQL sibling carries its own 17 survivors forward with a real `bulk_create()` call, wrapped in `transaction.atomic()`. `better-sqlite3` has no equivalent bulk-insert API at all — there's no method that accepts an array of rows and inserts them all in one call. The identical atomicity guarantee is reached the same way every other multi-write operation in this course has reached it since Chapter 3: wrap a plain, ordinary `for` loop of individual `.run()` calls in `db.transaction()`.

```
// src/pages/api/seasons/[seasonId]/rollover.ts
import type { APIRoute } from 'astro';
import { db } from '../../lib/db';
import { json } from '../../lib/http';
import { getLeagueTable, getRelegated, getSurvivors } from '../../lib/rollover';

export const POST: APIRoute = async ({ params, request }) => {
  const oldSeasonId = Number(params.seasonId);
  const { new_season_id } = await request.json();

  const oldSeason = db.prepare('SELECT * FROM seasons WHERE id = ?').get(oldSeasonId);
```

```

const newSeason = db.prepare('SELECT * FROM seasons WHERE id = ?').get(new_season_id)
if (!oldSeason || !newSeason) {
  return json({ error: 'Old or new season not found' }, 404);
}

const table = getLeagueTable(oldSeasonId);
if (table.length !== 20) {
  return json(
    { error: `Expected 20 teams in the old season's table, found ${table.length}` },
    400
  );
}

const survivors = getSurvivors(oldSeasonId);
const relegated = getRelegated(oldSeasonId);

const rollover = db.transaction(() => {
  for (const team of survivors) {
    const existing = db.prepare(
      'SELECT * FROM season_teams WHERE season_id = ? AND team_id = ?'
    ).get(new_season_id, team.team_id);

    if (existing) {
      throw new Error('ALREADY_ROLLED_OVER');
    }

    db.prepare(
      'INSERT INTO season_teams (season_id, team_id) VALUES (?, ?)'
    ).run(new_season_id, team.team_id);
  }
});

try {
  rollover();
} catch (err) {
  if ((err as Error).message === 'ALREADY_ROLLED_OVER') {
    return json(
      { error: 'One or more surviving teams are already in the new season – rollover failed' },
      409
    );
  }
  throw err;
}

```

```

}

return json(
  {
    survivors: survivors.map((t) => t.name),
    relegated: relegated.map((t) => t.name),
  },
  201
);
};

```

SAME ATOMICITY, REACHED BY WRAPPING A LOOP INSTEAD OF CALLING A DEDICATED BULK API

17 separate `INSERT` statements run inside `db.transaction()` either all commit together or, if any one of them throws — including the deliberate `'ALREADY_ROLLED_OVER'` guard above — none of them do. That's the identical all-or-nothing guarantee `bulk_create()` gives the Django sibling, reached through a genuinely different, plainer mechanism: there's no special bulk-insert code path here at all, just the same `db.transaction(fn)` wrapper this course has reused since Chapter 3, wrapped around an ordinary loop instead of a single statement.

THE DOUBLE-ROLLOVER GUARD CHECKS ONE ROW AT A TIME. ON PURPOSE

Checking each survivor individually for an existing `season_teams` row — rather than a single coarse check for "does the new season have any rows in it at all" — matters for a genuinely realistic reason: an admin might reasonably add one or more of the three promoted teams via Chapter 3's own route *before* ever calling `rollover`, simply because they already know which clubs came up. A blanket "any row exists yet" check would see those promoted-team rows and refuse to run the rollover at all, even though none of them collide with any of the 17 *survivor* `team_ids` the loop is actually about to insert. Checking each survivor specifically means the promoted teams' own already-added rows are simply irrelevant to this check — only a genuine duplicate among the 17 survivors themselves ever trips the guard.

A GENUINE PARTIAL FAILURE NEVER NEEDS THIS GUARD AT ALL

Because the entire loop runs inside one `db.transaction()`, a real failure partway through — a crash, an unrelated thrown error — rolls back every insert that transaction attempted, leaving the new season's own survivor rows exactly as empty as before the call started. A retry after that kind of failure simply reinserts all 17 from scratch with no conflicts to detect; the per-team check above isn't

a resume mechanism for a half-finished transaction, since the transaction wrapper already guarantees there's no such thing as "half-finished" to resume.

The Three Promoted Teams: Reusing Chapter 3's Own Route, Not Rebuilding It

A promoted club needs exactly the same handling Chapter 3 already built: check whether this club has a historical `teams` row already (a team coming back up after a previous relegation) or genuinely needs a new one, then add a `season_teams` row for the new season. That's precisely what `POST /api/seasons/{seasonId}/teams` already does — so rather than building a second, parallel route here that duplicates that exact reuse-or-create logic, the three promoted teams are simply added to the new season with three ordinary calls to the route Chapter 3 already wrote:

```
// Three plain calls to Chapter 3's own route – no new endpoint needed
await fetch(`/api/seasons/${newSeasonId}/teams`, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ team_name: 'Leeds United', short_name: 'Leeds' }),
});
// ...and again for the second and third promoted clubs
```

A DELIBERATE ARCHITECTURAL CHOICE, NOT AN OVERSIGHT

Which three teams actually get promoted is a genuine, real-world piece of knowledge — it's decided by the Championship's own final table, something no query against this app's own database could ever determine, since this app doesn't track any division below the Premier League. That's exactly the kind of decision Chapter 3's route already exists to handle by hand: an admin who knows the real, current promoted clubs types their names in, the route reuses or creates their `teams` row as appropriate, and adds them to the new season. The `rollover` route above only automates the part that genuinely *is* mechanical — carrying forward the 17 teams this app's own data already knows survived.

Relegating a Team Deletes Nothing

It's worth being precise about what "relegated" actually means here, since nothing in this chapter runs a single `DELETE` statement. The old season's own `season_teams` rows — including the three relegated teams' own rows — are never touched by any of the code above. Relegation is really just an **omission**: the rollover loop only ever inserts rows for the 17 *survivors*, so a relegated team simply never gets a row in the new season's own `season_teams` at all. Its historical membership in every season it actually played in, including the one it just got relegated from, stays exactly as recorded, permanently — the same "teams persist forever, `season_teams` tracks membership per season" split Chapter 2 established from the very beginning of this course.

A Read-Only Relegation Preview

Deciding to run the rollover route is a real, one-way step for a season — a lightweight, read-only route lets an admin check who's actually getting relegated first, with no risk of accidentally triggering anything:

```
// src/pages/api/seasons/[seasonId]/relegation-preview.ts
import type { APIRoute } from 'astro';
import { getRelegated, getSurvivors } from '../../../lib/rollover';
import { json } from '../../../lib/http';

export const GET: APIRoute = async ({ params }) => {
  const seasonId = Number(params.seasonId);
  return json({
    relegated: getRelegated(seasonId),
    survivors: getSurvivors(seasonId),
  });
};
```

Where This Course Is Headed

Astro Islands and interactivity — replacing the hardcoded `seasonId` / `gameweekId` / `fixtureId` values every chapter since Chapter 4 has deliberately left in place with a real, working selector (Chapter 10); deployment (Chapter 11); and the capstone, mounting this app directly onto the real, live Astro site (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain why this chapter needs no refactoring step to reuse Chapter 7's own league-table logic, and contrast that directly with what the FastAPI and Django sibling courses had to do at this exact point in their own Chapter 9s.

 [View solution](#)

EXERCISE 2

Explain why the rollover route checks each survivor individually for an existing season_teams row inside the loop, rather than checking once, up front, whether the new season already has any season_teams rows at all before starting, and explain separately why a genuine partial-failure retry never needs this guard to "resume" anything.

 [View solution](#)

EXERCISE 3

Set up a complete 20-team season with a full league table, call GET /api/seasons/{seasonId}/relegation-preview to confirm the correct bottom three, then call POST /api/seasons/{seasonId}/rollover against a newly created next season and confirm the new season's season_teams contains exactly the 17 survivors — followed by three calls to Chapter 3's own POST /api/seasons/{id}/teams route to add three promoted clubs, bringing the new season back up to 20.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- **No refactor needed** — getLeagueTable() has lived in its own module since Chapter 7 specifically for this reuse, unlike both sibling courses' own inline-query-to-function refactor at this point
- **getRelegated()/getSurvivors()** — table.slice(-3) and table.slice(0, -3) against the already-sorted league table; correct regardless of table size

- **POST /api/seasons/{id}/rollover** — verifies 20 teams exist, then inserts the 17 survivors' own season_teams rows for the new season, wrapped in db.transaction()
- **Plain loop, not bulk_create()** — better-sqlite3 has no bulk-insert API; the identical atomicity guarantee comes from wrapping an ordinary for loop in the same db.transaction() this course has used since Chapter 3
- **Double-rollover guard checks per-team** — a coarse "does the new season have any rows yet" check would wrongly block a legitimate rollover after promoted teams were already added via Chapter 3's own route; a genuine partial failure needs no such guard at all, since db.transaction() never leaves a half-finished result to resume
- **Promoted teams reuse Chapter 3's own route** — no second add-team endpoint is built; which clubs are promoted is real-world knowledge no query here could ever determine
- **Relegation deletes nothing** — the old season's own season_teams rows are untouched forever; a relegated team simply never gets a row in the new season
- **GET /api/seasons/{id}/relegation-preview** — a read-only check before the one-way rollover step is actually triggered
- **Next chapter:** Astro Islands & interactivity — a real gameweek/season selector, replacing every hardcoded ID since Chapter 4

CHAPTER 10: ASTRO ISLANDS & INTERACTIVITY: WHERE CLIENT-SIDE JS ACTUALLY LIVES IN A STATIC-FIRST FRAMEWORK

Premier League Predictor: Astro

Chapter 10 · Astro Islands & Interactivity: Where Client-Side JS Actually Lives in a Static-First Framework

Two real promises have been sitting open since earlier chapters: Chapter 4's own hardcoded `seasonId` / `gameweekId`, waiting for "a real gameweek/season selector," and Chapter 1's own note that `output: 'server'` makes every page dynamic by default — "a deliberate, temporary tradeoff" — pending a return to the question of which pages could genuinely opt back into static prerendering. This chapter closes both, and starts with what "Astro Islands" actually means, since this course has never used one.

What an Island Actually Is — and Why This Course Has Never Needed One

Astro's real Islands Architecture is specifically about embedding components from a UI framework — React, Vue, Svelte, Preact, Solid — inside an otherwise static or server-rendered Astro page, and choosing exactly when each one hydrates on the client via a `client:*` directive: `client:load` (immediately), `client:idle`, `client:visible`, `client:media`, or `client:only`. Each one is a genuinely isolated "island" of client-side JavaScript-framework interactivity sitting in a sea of otherwise plain HTML.

Every interactive page this course has built — Chapter 4's fixture-entry grid, Chapter 5's prediction forms — has used a plain `<script>` tag instead, with no framework component and no `client:*` directive anywhere. That's a genuinely different, simpler mechanism: Astro bundles that script into a real ES module and ships it to the browser, but there's no framework runtime, no component hydration, and nothing being "reactivated" the way an island's own framework component is. This wasn't an oversight — every interaction this app needs (clicking a team button, submitting a prediction) is a handful of DOM updates driven by plain event listeners, and installing a UI framework specifically to build a five-button grid or a two-input form would be a real, ongoing dependency added for no real gain.

THIS IS THE HONEST, STRUCTURAL REASON THIS CHAPTER NEVER INSTALLS REACT OR VUE

Reaching for an island only pays off once a piece of UI genuinely needs a framework's own state-management and re-rendering model — a live-updating form with several interdependent fields, for instance. Nothing this app has built so far crosses that line; every "interactive" page is really just "read some data, update a few DOM nodes when it's clicked." The real selector built in this chapter uses Astro's own actual strength instead — real, server-rendered dynamic routes — which turns out to solve the exact problem a client-side island would otherwise have been reached for.

The Real Selector: A Dynamic Route, Not Client-Side State

Rather than keeping one fixed `/admin/fixtures` page and switching gameweeks with client-side JavaScript, the season and gameweek become real parts of the URL — a plain Astro dynamic route, resolved server-side on every request:

```
---
// src/pages/admin/seasons/[seasonId]/gameweeks/[gameweekNumber]/fixtures.astro
import { db } from '../../../../lib/db';

const seasonId = Number(Astro.params.seasonId);
const gameweekNumber = Number(Astro.params.gameweekNumber);

const season = db.prepare('SELECT * FROM seasons WHERE id = ?').get(seasonId);
if (!season) {
  return Astro.redirect('/admin/seasons');
}

const gameweek = db.prepare(
  'SELECT * FROM gameweeks WHERE season_id = ? AND number = ?'
).get(seasonId, gameweekNumber) as { id: number } | undefined;

// Every season this app has, for the season 
```