

PERSONAL MULTI-TECHNOLOGY PROJECTS

Premier League Predictor: FastAPI & PostgreSQL

A real, working weekly Premier League prediction tracker — four scored prediction sources (user, BBC expert, guest(s) averaged, BBC AI), a real normalized PostgreSQL schema, a fast click-to-pair fixture-entry UI, two genuinely different league tables, promotion and relegation, and a real deployment mounted onto an existing Astro site — the full relational reference implementation of a four-course technology comparison.

12 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & STACK SETUP

Premier League Predictor: FastAPI & PostgreSQL

Chapter 1 · Project Overview & Stack Setup

This is the first of four courses building the exact same Premier League prediction tracker in four genuinely different architectures — Premier League Predictor (FastAPI & Redis), Premier League Predictor (Django & MySQL), and Premier League Predictor (Astro) are its siblings, all still outlined rather than generated yet. This course's own answer leans into a real, fully normalized relational schema and a proper backend framework — the "does everything the textbook way" reference implementation the other three variants get compared against once they exist.

What the App Actually Does

The 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.

ONE REAL DETAIL STILL GENUINELY UNDECIDED

The actual point values for a correct score versus a correct result haven't been settled yet — this course flags that honestly here rather than guessing a number and presenting it as final. Chapter 6, where results and scoring actually get built, is where that number needs to be confirmed before the scoring logic can be written for real.

Why FastAPI + PostgreSQL for This One

Of the four variants in this set, this is the one built around genuine relational integrity and a real, typed API layer — the two things a season-long, multi-source prediction tracker benefits from most directly:

- **A real normalized schema.** Teams, seasons, gameweeks, fixtures, and predictions are all genuinely separate, foreign-key-linked tables — not documents, not key-value pairs — which is exactly the shape this data already has (Chapter 2).
- **Pydantic-validated, typed request/response models.** Four prediction sources per fixture, two different scoring outcomes per prediction, is enough real structure that catching a malformed payload before it ever reaches the database is a genuine, practical win, not just tidiness.
- **PostgreSQL's own real aggregate and window functions** are a genuine fit for both league tables this app maintains — the real league table's own points/goal-difference sort, and the prediction league table's own per-predictor scoring aggregation, both covered in Chapters 7 and 8.

THIS COURSE ASSUMES REAL FASTAPI/POSTGRESQL GROUNDWORK ALREADY COVERED ELSEWHERE

This site's own `fastapi1` (Website Rebuild with FastAPI) and `fastapi-food-tracker-1` (Food Tracker) courses already cover FastAPI's own project structure, routing, and Pydantic validation in real

depth, and `postgres1` covers relational schema design and query fundamentals directly. This course doesn't re-teach any of that — it assumes it, and moves straight into this predictor's own real schema and features.

Setting Up: FastAPI, SQLAlchemy & PostgreSQL

A local Python install with `pip`, and a working local PostgreSQL server, are assumed from here on:

```
# Create and activate a virtual environment
python -m venv venv
source venv/bin/activate # venv\Scripts\activate on Windows

# Install FastAPI, an ASGI server, SQLAlchemy, and the PostgreSQL driver
pip install fastapi uvicorn[standard] sqlalchemy psycopg2-binary python-dotenv
```

Create the database this whole course builds on:

```
CREATE DATABASE pl_predictor WITH ENCODING 'UTF8';
```

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

```
pl-predictor-fastapi/
├─ main.py           # FastAPI app instance, route registration
├─ models.py         # SQLAlchemy models (Chapter 2)
├─ schemas.py        # Pydantic models (Chapter 2)
├─ database.py        # SQLAlchemy engine/session setup
├─ routers/
│   ├─ teams.py
│   ├─ fixtures.py
│   └─ predictions.py
├─ tables.py
├─ static/           # frontend, styled in Chapter 10
└─ requirements.txt
```

A minimal `main.py`, confirming the connection works before any real feature exists:

```
# main.py
from fastapi import FastAPI

app = FastAPI(title="Premier League Predictor")

@app.get("/api/health")
async def health():
    return {"status": "ok"}
```

```
uvicorn main:app --reload
```

/DOCS IS A GENUINE, PRACTICAL WIN FOR A PROJECT SHAPED LIKE THIS ONE

Every route this course builds — team management, fixture entry, four separate prediction sources per fixture, results, two league tables — shows up automatically at `http://localhost:8000/docs` as a real, working interactive explorer, generated straight from the route definitions and Pydantic models this course writes anyway. For a project with this much real internal structure, that's a genuinely useful way to test each piece as it's built, well before any frontend exists to click through.

Where This Course Is Headed

A real normalized 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 settling the still-open correct-score-vs-correct-result point values (Chapter 6); the real league table (Chapter 7); the parallel prediction league table (Chapter 8); promotion and relegation between seasons (Chapter 9); styling and the gameweek/season selector (Chapter 10); deployment (Chapter 11); and a capstone on integrating this predictor into the existing Astro-based 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 is a correct result but not a correct score.

 [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 `pl_predictor` PostgreSQL database and a working FastAPI project connected to it yourself, confirm `/api/health` responds, then check `/docs` in a browser and write one sentence on what it's showing you before any real route exists.

 [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 by correct score and 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
- **Still open** — the real point values for correct score vs. correct result, to be confirmed before Chapter 6
- **Why this variant** — a real normalized relational schema plus a typed, Pydantic-validated API — the full reference implementation this set's other three variants get compared against

- **Database** — PostgreSQL, created as `pl_predictor` with explicit UTF8 encoding
- **Assumed groundwork** — FastAPI and PostgreSQL fundamentals, both covered in their own dedicated courses on this site
- **Next chapter:** Data Modeling — teams, seasons, gameweeks & fixtures

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

Premier League Predictor: FastAPI & PostgreSQL

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: `Team`, `Season`, `SeasonTeam`, `Gameweek`, and `Fixture`. 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. `Team` 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.

`Season` 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.

SeasonTeam: 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 `Team` itself: `SeasonTeam`, 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 `SeasonTeam` 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 `SeasonTeam` rows for next season, add three new rows for the promoted teams. `Team` itself never changes in either operation — only which teams are marked as competing in a given season does.

Gameweek & Fixture: The Real Schedule

A real season has 38 gameweeks; each gameweek has (normally) 10 fixtures, one per pair of the 20 competing teams. Both get their own table:

```
# models.py
from sqlalchemy import (
    Column, Integer, String, Boolean, Date, DateTime,
    ForeignKey, UniqueConstraint, CheckConstraint, func
)
from sqlalchemy.orm import relationship
from database import Base

class Team(Base):
    __tablename__ = "teams"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, unique=True, nullable=False)
    short_name = Column(String, nullable=False)

class Season(Base):
    __tablename__ = "seasons"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, unique=True, nullable=False) # e.g. "2026/27"
    start_date = Column(Date, nullable=False)
    end_date = Column(Date, nullable=False)
    is_current = Column(Boolean, nullable=False, default=False)

    gameweeks = relationship("Gameweek", back_populates="season")
    season_teams = relationship("SeasonTeam", back_populates="season")

class SeasonTeam(Base):
    __tablename__ = "season_teams"
    __table_args__ = (
        UniqueConstraint("season_id", "team_id", name="uq_season_team"),
    )
```

```

id = Column(Integer, primary_key=True, index=True)
season_id = Column(Integer, ForeignKey("seasons.id"), nullable=False)
team_id = Column(Integer, ForeignKey("teams.id"), nullable=False)

season = relationship("Season", back_populates="season_teams")
team = relationship("Team")

class Gameweek(Base):
    __tablename__ = "gameweeks"
    __table_args__ = (
        UniqueConstraint("season_id", "number", name="uq_season_gameweek_number"),
    )

    id = Column(Integer, primary_key=True, index=True)
    season_id = Column(Integer, ForeignKey("seasons.id"), nullable=False)
    number = Column(Integer, nullable=False) # 1 through 38

    season = relationship("Season", back_populates="gameweeks")
    fixtures = relationship("Fixture", back_populates="gameweek")

class Fixture(Base):
    __tablename__ = "fixtures"
    __table_args__ = (
        CheckConstraint("home_team_id != away_team_id", name="ck_fixture_teams_differ")
    )

    id = Column(Integer, primary_key=True, index=True)
    gameweek_id = Column(Integer, ForeignKey("gameweeks.id"), nullable=False)
    home_team_id = Column(Integer, ForeignKey("teams.id"), nullable=False)
    away_team_id = Column(Integer, ForeignKey("teams.id"), nullable=False)
    kickoff_time = Column(DateTime, nullable=True)
    home_score = Column(Integer, nullable=True)
    away_score = Column(Integer, nullable=True)
    status = Column(String, nullable=False, default="scheduled")

    gameweek = relationship("Gameweek", back_populates="fixtures")
    home_team = relationship("Team", foreign_keys=[home_team_id])
    away_team = relationship("Team", foreign_keys=[away_team_id])

```

WHY FOREIGN_KEYS=[...] IS REQUIRED, NOT OPTIONAL, ON HOME_TEAM AND AWAY_TEAM

`Fixture` has **two** foreign keys pointing at the same table, `teams.id` — `home_team_id` and `away_team_id`. Without telling SQLAlchemy explicitly which column each relationship should join on, it can't work out on its own which of the two foreign keys `home_team` is even supposed to follow — the same ambiguity a human reader would have looking at "two arrows into the same table" with no labels. `foreign_keys=[home_team_id]` and `foreign_keys=[away_team_id]` resolve that ambiguity directly. A single self-referencing foreign key (like `SeasonTeam.team_id` above) never needs this, since there's only one possible column for SQLAlchemy to pick.

Nullable Scores: A Fixture Exists Before It's Played

`home_score` and `away_score` are both `nullable=True`, 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

`CheckConstraint("home_team_id != away_team_id")` stops a team from playing itself — that's a genuine per-row check the database can enforce directly. 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 plain column-level constraint can't express that. 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 foreign key column on `Fixture`, 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.

The Pydantic Layer: Request & Response Shape

Following the same two-layer split this site's own `fastapi-food-tracker-1` course already established — SQLAlchemy models describe what's actually stored, Pydantic schemas describe what a request or response is allowed to look like. Two representative examples here; the rest get built alongside their own routes in later chapters.

```
# schemas.py
from pydantic import BaseModel
from datetime import datetime
from typing import Optional

class TeamCreate(BaseModel):
    name: str
    short_name: str

class TeamResponse(BaseModel):
    id: int
    name: str
    short_name: str

class Config:
    from_attributes = True

class FixtureResponse(BaseModel):
    id: int
    gameweek_id: int
    home_team_id: int
    away_team_id: int
    kickoff_time: Optional[datetime]
    home_score: Optional[int]
    away_score: Optional[int]
    status: str

class Config:
    from_attributes = True
```

`FixtureResponse`'s own `home_score` and `away_score` being `Optional[int]` reads directly off the nullable columns above — a fixture that hasn't been played yet serializes with both fields simply absent, no separate "played: false" flag needed to know that.

Connecting to PostgreSQL & Creating the Tables

```
# database.py
import os
from dotenv import load_dotenv
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

load_dotenv()

DATABASE_URL = os.getenv(
    "DATABASE_URL", "postgresql://postgres:@localhost:5432/pl_predictor"
)

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

A single line in `main.py`, after importing `models`, creates every table above against the real database from Chapter 1:

```
import models
from database import engine

models.Base.metadata.create_all(bind=engine)
```

Where This Course Is Headed

Managing the 20 competing teams each season — real CRUD against `SeasonTeam` (Chapter 3); the fast click-to-pair fixture-entry UI that creates `Fixture` 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 `SeasonTeam` 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 `SeasonTeam` table rather than a boolean column on `Team` itself, and name the two later chapters that operate directly on `SeasonTeam` rows.

[!\[\]\(1a2e9c86c2a63dd0890db1012b677415_img.jpg\) View solution](#)

EXERCISE 2

Explain why `Fixture`'s `home_team` and `away_team` relationships both need an explicit `foreign_keys=[...]` argument, while `SeasonTeam`'s `team` relationship doesn't need one at all.

[!\[\]\(1a0ecb0f44016aa353f6ecdd79a3699d_img.jpg\) View solution](#)

EXERCISE 3

Explain why the "same team playing twice in one gameweek" rule is enforced in the fixture-entry UI rather than as a database constraint, and contrast that with the "team can't play itself" rule, which is enforced directly by `CheckConstraint`.

[!\[\]\(d3b4f22af99c507f55d7924c8d6d7349_img.jpg\) View solution](#)

CHAPTER 2 QUICK REFERENCE

- **Team** — every team ever tracked, permanent regardless of promotion/relegation
- **Season** — one real Premier League season, with an `is_current` flag
- **SeasonTeam** — the join table recording which 20 teams compete in a given season; Chapters 3 and 9 both operate directly on it
- **Gameweek** — 1 through 38 per season, unique per (season_id, number)
- **Fixture** — `home_team_id/away_team_id` both self-reference `teams.id`, needing explicit `foreign_keys=` [...] on each relationship
- **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" (`CheckConstraint`)
- **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: FastAPI & PostgreSQL

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

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

Bootstrapping a Season

Extending `schemas.py` with the Season shapes:

```
# schemas.py (additions)
from datetime import date

class SeasonCreate(BaseModel):
    name: str
    start_date: date
    end_date: date
    is_current: bool = False

class SeasonResponse(BaseModel):
    id: int
    name: str
    start_date: date
    end_date: date
    is_current: bool

class Config:
    from_attributes = True
```

```
# routers/seasons.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from database import get_db
import models, schemas
```

```

router = APIRouter(prefix="/api", tags=["seasons"])

@router.post("/seasons", response_model=schemas.SeasonResponse)
def create_season(season: schemas.SeasonCreate, db: Session = Depends(get_db)):
    if season.is_current:
        # Only one season is ever "current" — clear the flag on every other one first
        db.query(models.Season).update({models.Season.is_current: False})

    db_season = models.Season(**season.model_dump())
    db.add(db_season)
    db.commit()
    db.refresh(db_season)
    return db_season

```

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

"At most one season is marked current" spans every row in the `seasons` table, not just the one being inserted — the same category of rule as Chapter 2's own "no team twice in a gameweek,"

every other season's flag inside the same request, before the new season is even added — 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 — `Team` 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:

```

# schemas.py (additions)
class SeasonTeamAdd(BaseModel):
    team_name: str
    short_name: Optional[str] = None # only used if the team doesn't already exist

class SeasonTeamResponse(BaseModel):
    id: int

```

```

season_id: int
team: TeamResponse

class Config:
    from_attributes = True

```

```
# routers/seasons.py (additions)
```

```
MAX_TEAMS_PER_SEASON = 20
```

```

@router.post("/seasons/{season_id}/teams", response_model=schemas.SeasonTeamResponse)
def add_team_to_season(
    season_id: int, payload: schemas.SeasonTeamAdd, db: Session = Depends(get_db)
):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")

    current_count = (
        db.query(models.SeasonTeam)
        .filter(models.SeasonTeam.season_id == season_id)
        .count()
    )
    if current_count >= MAX_TEAMS_PER_SEASON:
        raise HTTPException(
            status_code=400,
            detail=f"Season already has {MAX_TEAMS_PER_SEASON} teams",
        )

    # Reuse the historical Team row if this club has been tracked before
    team = db.query(models.Team).filter(models.Team.name == payload.team_name).first()
    if not team:
        team = models.Team(name=payload.team_name, short_name=payload.short_name or pay
        db.add(team)
        db.flush() # assigns team.id without committing yet

    already_in = (
        db.query(models.SeasonTeam)
        .filter(
            models.SeasonTeam.season_id == season_id,
            models.SeasonTeam.team_id == team.id,

```

```

    )
    .first()
)
if already_in:
    raise HTTPException(
        status_code=409, detail=f"{team.name} is already in this season"
    )

season_team = models.SeasonTeam(season_id=season_id, team_id=team.id)
db.add(season_team)
db.commit()
db.refresh(season_team)
return season_team

```

WHY THE NEW TEAM IS `DB.FLUSH()`'D, NOT `DB.COMMIT()`'D, BEFORE CREATING THE `SEASONTTEAM` ROW

`flush()` sends the pending `INSERT` to PostgreSQL and lets the database assign `team.id` — needed here, since `SeasonTeam.team_id` can't be set until an id actually exists — without ending the transaction. Both the new `Team` row and the new `SeasonTeam` row stay part of the same transaction until the final `commit()`, so if anything after the flush goes wrong, both roll back together rather than leaving an orphaned `Team` row with no `SeasonTeam` to go with it.

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

`db.query(models.Team).filter(models.Team.name == payload.team_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 `Team` 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

```

@router.delete("/seasons/{season_id}/teams/{team_id}", status_code=204)
def remove_team_from_season(season_id: int, team_id: int, db: Session = Depends(get_db))
    season_team = (
        db.query(models.SeasonTeam)

```

```

        .filter(
            models.SeasonTeam.season_id == season_id,
            models.SeasonTeam.team_id == team_id,
        )
        .first()
    )
    if not season_team:
        raise HTTPException(status_code=404, detail="Team is not in this season")

    db.delete(season_team)
    db.commit()

```

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

Listing a Season's Teams

```

@router.get("/seasons/{season_id}/teams", response_model=list[schemas.TeamResponse])
def list_season_teams(season_id: int, db: Session = Depends(get_db)):
    return (
        db.query(models.Team)
        .join(models.SeasonTeam, models.SeasonTeam.team_id == models.Team.id)
        .filter(models.SeasonTeam.season_id == season_id)
        .order_by(models.Team.name)
        .all()
    )

```

A NESTED PYDANTIC MODEL READS A JOINED RESULT DIRECTLY

`SeasonTeamResponse.team: TeamResponse`, back up in the schema for the "add team" route, is a real Pydantic model nested inside another one. It works because `SeasonTeam.team` — the `relationship("Team")` defined in Chapter 2 — already gives a real `Team` object to read from; `from_attributes = True` lets Pydantic walk that relationship automatically and serialize the related team's own fields as a nested object in the JSON response, with no manual re-shaping of the query result required.

THESE ROUTES HAVE NO ACCESS CONTROL

Every route in this chapter is reachable by anyone who can reach the API — 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/{season_id}/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 `create_season` clears `is_current` on every other season with a bulk update before inserting the new one, and what would go wrong if that step were skipped.

[!\[\]\(375cabd837b97cf016d36e6dfd1b1d2f_img.jpg\) View solution](#)

EXERCISE 2

Explain why `add_team_to_season` 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 `Team` row for the same real club.

[!\[\]\(51b8a815e60dd7bf4906b87368d243e1_img.jpg\) View solution](#)

EXERCISE 3

Explain why `db.flush()` is used instead of `db.commit()` when creating a brand-new Team inside `add_team_to_season`, and what real guarantee would be lost if `commit()` were used at that point instead.

[View solution](#)

CHAPTER 3 QUICK REFERENCE

- **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
- **db.flush() vs. db.commit()** — flush assigns an id without ending the transaction, keeping a new Team and its SeasonTeam row atomic together
- **DELETE /api/seasons/{id}/teams/{team_id}** — removes the SeasonTeam row only; the historical Team row is untouched
- **GET /api/seasons/{id}/teams** — a real join, returned through a nested SeasonTeamResponse/TeamResponse Pydantic model
- **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: FastAPI & PostgreSQL

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 plain vanilla-JS/HTML/CSS frontend (served as static files, no build step — the same choice this site's own FastAPI courses already make when there's no React sibling forcing a different pick) with 20 always-visible team buttons, clicked directly into place.

Creating a Gameweek

```
# schemas.py (additions)
class GameweekCreate(BaseModel):
    number: int

class GameweekResponse(BaseModel):
    id: int
    season_id: int
    number: int

class Config:
    from_attributes = True
```

```
# routers/gameweeks.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from sqlalchemy.exc import IntegrityError
from database import get_db
import models, schemas

router = APIRouter(prefix="/api", tags=["gameweeks"])
```

```

@router.post("/seasons/{season_id}/gameweeks", response_model=schemas.GameweekResponse)
def create_gameweek(season_id: int, payload: schemas.GameweekCreate, db: Session = Depends(get_db)):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")

    if not (1 <= payload.number <= 38):
        raise HTTPException(status_code=400, detail="Gameweek number must be between 1 and 38")

    gameweek = models.Gameweek(season_id=season_id, number=payload.number)
    db.add(gameweek)
    try:
        db.commit()
    except IntegrityError:
        db.rollback()
        raise HTTPException(
            status_code=409, detail=f"Gameweek {payload.number} already exists for this season"
        )
    db.refresh(gameweek)
    return gameweek

```

TURNING CHAPTER 2'S UNIQUECONSTRAINT INTO A FRIENDLY ERROR, NOT A CRASH

Chapter 2's own `UniqueConstraint("season_id", "number")` already stops a genuinely duplicate gameweek from ever being stored. Without the `try/except` above, PostgreSQL would still reject the duplicate — but as a raw `IntegrityError` bubbling straight up through FastAPI as an unhandled `500`, with none of the useful "gameweek 12 already exists" detail this route actually returns. Catching it and calling `db.rollback()` — required, since a session that hit a failed `commit()` can't be reused until it's rolled back — turns a genuine database-level guarantee into a genuinely useful `409` response instead.

Creating a Fixture: The Real Server-Side Guard

```

# routers/fixtures.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from database import get_db
import models, schemas

```

```

router = APIRouter(prefix="/api", tags=["fixtures"])

@router.post("/gameweeks/{gameweek_id}/fixtures", response_model=schemas.FixtureResponse)
def create_fixture(gameweek_id: int, payload: schemas.FixtureCreate, db: Session = Depends(get_db)):
    gameweek = db.get(models.Gameweek, gameweek_id)
    if not gameweek:
        raise HTTPException(status_code=404, detail="Gameweek not found")

    if payload.home_team_id == payload.away_team_id:
        raise HTTPException(status_code=400, detail="A team cannot play itself")

    # The real authority: check every fixture already entered this gameweek
    existing = db.query(models.Fixture).filter(models.Fixture.gameweek_id == gameweek_id)
    used_team_ids = set()
    for fx in existing:
        used_team_ids.add(fx.home_team_id)
        used_team_ids.add(fx.away_team_id)

    if payload.home_team_id in used_team_ids or payload.away_team_id in used_team_ids:
        raise HTTPException(
            status_code=409, detail="One of these teams is already fixtured this gameweek"
        )

    fixture = models.Fixture(
        gameweek_id=gameweek_id,
        home_team_id=payload.home_team_id,
        away_team_id=payload.away_team_id,
        kickoff_time=payload.kickoff_time,
    )
    db.add(fixture)
    db.commit()
    db.refresh(fixture)
    return fixture

@router.get("/gameweeks/{gameweek_id}/fixtures", response_model=list[schemas.FixtureResponse])
def list_fixtures(gameweek_id: int, db: Session = Depends(get_db)):
    return (
        db.query(models.Fixture)
        .filter(models.Fixture.gameweek_id == gameweek_id)

```

```
.all()
)
```

`schemas.FixtureCreate`, added alongside the others:

```
class FixtureCreate(BaseModel):
    home_team_id: int
    away_team_id: int
    kickoff_time: Optional[datetime] = None
```

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." `create_fixture` above is that application layer — a real, explicit query across every existing fixture in the gameweek, not a database constraint. Checking `home_team_id == away_team_id` here as well, even though Chapter 2's `CheckConstraint` already blocks it at the database level, means a bad request gets a clean `400` with a real message instead of an unhandled `IntegrityError` — 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.

```
<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 onclick="clearSelection()">Clear</button>
<button id="add-fixture-btn" onclick="addFixture()" disabled>Add Fixture</button>
```

```

// static/fixtures.js
// 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 = null;
let selectedAway = null;
let usedTeamIds = new Set();

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 => {
    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) {
  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').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').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.detail); // e.g. "One of these teams is already fixtured this gameweek"
    return;
  }

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

loadTeams();

```

Clicking a team fills `home-slot` first, then `away-slot` — a two-click fixture, every time. Once both slots hold a team, **Add Fixture** enables; submitting it grays out both teams' own buttons for the rest of this gameweek, and resets the slots for the next pair.

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 `create_fixture`'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 a real gameweek/season selector to replace this chapter's own hardcoded `seasonId` / `gameweekId` (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why `create_gameweek` and `create_fixture` both check their own rule in Python (the 1-38 range, `home_team_id == away_team_id`) even though a database-level guarantee (a `CheckConstraint` or a `UniqueConstraint`) already exists for each one.

 [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.

 [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, and then write one sentence explaining what happens when you try to click an 11th team

pairing that reuses an already-fixture team.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **POST /api/seasons/{id}/gameweeks** — creates a gameweek (1-38); catches the Chapter 2 UniqueConstraint's IntegrityError as a friendly 409
- **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** — plain vanilla JS/HTML/CSS, no framework, no build step; 20 always-visible team buttons instead of dropdowns
- **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 create_fixture 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: FastAPI & PostgreSQL

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.

```
# models.py (additions)
import enum
from sqlalchemy import Enum, Index

class PredictionSource(str, enum.Enum):
    USER = "user"
    EXPERT = "expert"
    GUEST = "guest"
    AI = "ai"

class Prediction(Base):
    __tablename__ = "predictions"

    id = Column(Integer, primary_key=True, index=True)
    fixture_id = Column(Integer, ForeignKey("fixtures.id"), nullable=False)
    source = Column(Enum(PredictionSource), nullable=False)
    guest_name = Column(String, nullable=True) # set only when source == GUEST
    predicted_home_score = Column(Integer, nullable=False)
    predicted_away_score = Column(Integer, nullable=False)
```

```

created_at = Column(DateTime, server_default=func.now())

fixture = relationship("Fixture", back_populates="predictions")

# Extend Chapter 2's own Fixture class with the reverse side of this relationship:
# predictions = relationship("Prediction", back_populates="fixture")

```

A Partial Unique Index: Exactly the PostgreSQL Feature Chapter 1 Picked This Stack For

A plain `UniqueConstraint("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**, a real PostgreSQL feature (SQLite has no equivalent):

```

# models.py (after the Prediction class)
Index(
    "uq_prediction_single_source_per_fixture",
    Prediction.fixture_id,
    Prediction.source,
    unique=True,
    postgresql_where=(Prediction.source != PredictionSource.GUEST),
)

```

That compiles to a real index PostgreSQL only enforces where the condition holds:

```

CREATE UNIQUE INDEX uq_prediction_single_source_per_fixture
ON predictions (fixture_id, source)
WHERE source != 'guest';

```

A SINGLE MECHANISM DOING EXACTLY WHAT CHAPTER 1 POINTED AT, BUT NEVER NAMED

Chapter 1's own reasoning for this variant leaned on "a real normalized schema" and "PostgreSQL's own real aggregate and window functions" — this is the same spirit, one level more specific: a

genuinely PostgreSQL-native feature (a conditional index) expressing a rule this course's own data model needs — three sources locked to one prediction each, one source deliberately allowed to repeat — that a plain cross-database `UniqueConstraint` simply can't express on its own.

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:

```
# schemas.py (additions)
class PredictionCreate(BaseModel):
    source: models.PredictionSource
    guest_name: Optional[str] = None
    predicted_home_score: int
    predicted_away_score: int

class PredictionResponse(BaseModel):
    id: int
    fixture_id: int
    source: models.PredictionSource
    guest_name: Optional[str]
    predicted_home_score: int
    predicted_away_score: int

class Config:
    from_attributes = True
```

```
# routers/predictions.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from database import get_db
import models, schemas

router = APIRouter(prefix="/api", tags=["predictions"])

@router.post("/fixtures/{fixture_id}/predictions", response_model=schemas.PredictionRes
```

```

def upsert_prediction(fixture_id: int, payload: schemas.PredictionCreate, db: Session =
    fixture = db.get(models.Fixture, fixture_id)
    if not fixture:
        raise HTTPException(status_code=404, detail="Fixture not found")

    is_guest = payload.source == models.PredictionSource.GUEST
    if is_guest and not payload.guest_name:
        raise HTTPException(status_code=400, detail="guest_name is required for a guest")
    guest_name = payload.guest_name if is_guest else None

    query = db.query(models.Prediction).filter(
        models.Prediction.fixture_id == fixture_id,
        models.Prediction.source == payload.source,
    )
    if is_guest:
        query = query.filter(models.Prediction.guest_name == guest_name)
    existing = query.first()

    if existing:
        existing.predicted_home_score = payload.predicted_home_score
        existing.predicted_away_score = payload.predicted_away_score
        db.commit()
        db.refresh(existing)
        return existing

    prediction = models.Prediction(
        fixture_id=fixture_id,
        source=payload.source,
        guest_name=guest_name,
        predicted_home_score=payload.predicted_home_score,
        predicted_away_score=payload.predicted_away_score,
    )
    db.add(prediction)
    db.commit()
    db.refresh(prediction)
    return prediction

@router.get("/fixtures/{fixture_id}/predictions", response_model=list[schemas.Prediction])
def list_predictions(fixture_id: int, db: Session = Depends(get_db)):
    return (
        db.query(models.Prediction)

```

```

        .filter(models.Prediction.fixture_id == fixture_id)
        .all()
    )

```

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

If `upsert_prediction` simply inserted a new `Prediction` row on every call, submitting a second user prediction for the same fixture would collide directly with the partial unique index above — PostgreSQL would reject it as a real constraint violation, 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 `Prediction` 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:

```

[
  { "id": 1, "fixture_id": 42, "source": "user", "guest_name": null, "predicted_home_score": 1, "predicted_away_score": 0 },
  { "id": 2, "fixture_id": 42, "source": "expert", "guest_name": null, "predicted_home_score": 2, "predicted_away_score": 1 },
  { "id": 3, "fixture_id": 42, "source": "guest", "guest_name": "Micah Richards", "predicted_home_score": 1, "predicted_away_score": 0 },
  { "id": 4, "fixture_id": 42, "source": "guest", "guest_name": "Jamie Carragher", "predicted_home_score": 0, "predicted_away_score": 1 },
  { "id": 5, "fixture_id": 42, "source": "ai", "guest_name": null, "predicted_home_score": 1, "predicted_away_score": 0 }
]

```

Recording Predictions From the Frontend

A compact form, reusable across all four sources — the guest form adds a name field, and a page can render as many guest rows as that week actually needs:

```
// static/predictions.js
async function submitPrediction(fixtureId, source, guestName, homeScore, awayScore) {
  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.detail);
    return;
  }

  alert('Prediction saved.');
```



```
let guestRowCount = 0;

function addGuestRow() {
  guestRowCount += 1;
  const row = document.createElement('div');
  row.innerHTML = `
    <input type="text" placeholder="Guest name" id="guest-name-${guestRowCount}">
    <input type="number" placeholder="Home" id="guest-home-${guestRowCount}">
    <input type="number" placeholder="Away" id="guest-away-${guestRowCount}">
  `;
  document.getElementById('guest-rows').appendChild(row);
}
```

Every guest row gets its own `submitPrediction(fixtureId, '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 `UniqueConstraint("fixture_id", "source")` wouldn't work for the `Prediction` table, and what the `postgresql_where=(Prediction.source != PredictionSource.GUEST)` clause actually changes about which rows the unique rule applies to.

[!\[\]\(40ff4926cc4e80920dcd6e30a3ae173f_img.jpg\) View solution](#)**EXERCISE 2**

Explain what would happen if `upsert_prediction` 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.

[!\[\]\(23c95a79ea2fdecb85c88c5fd848f31c_img.jpg\) 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

- **Prediction** — one table, four sources (USER/EXPERT/GUEST/AI), keyed by fixture_id + source (+ guest_name for guests)
- **Partial unique index** — a real PostgreSQL feature enforcing "one prediction per source" for user/expert/AI while deliberately allowing multiple guest rows
- **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
- **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: FastAPI & PostgreSQL

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

Every fixture since Chapter 4 has sat with `home_score` / `away_score` null. Every prediction since Chapter 5 has sat unscored. This chapter fills in the real result and, at the same moment, scores everything that was predicted against it — with real, confirmed point values: **40 points for a correct score, 10 points for a correct result.**

A Pure Scoring Function, Shared by Every Source

User, expert, AI, and every individual guest are all scored by the exact same rule — there's no special case per source, because the rule genuinely doesn't need one:

```
# scoring.py
POINTS_CORRECT_SCORE = 40
POINTS_CORRECT_RESULT = 10

def _outcome(home: int, away: int) -> str:
    if home > away:
        return "home_win"
    elif away > home:
        return "away_win"
    return "draw"

def score_prediction(
    predicted_home: int, predicted_away: int, actual_home: int, actual_away: int
) -> int:
    if predicted_home == actual_home and predicted_away == actual_away:
        return POINTS_CORRECT_SCORE
    if _outcome(predicted_home, predicted_away) == _outcome(actual_home, actual_away):
        return POINTS_CORRECT_RESULT
    return 0
```

The exact-match check runs first, deliberately: as Chapter 1's own first exercise established, every correct score is automatically a correct result too, so checking the stronger condition first and returning immediately is what stops a 2-1 correctly predicted 2-1 from ever being scored as a mere 10-point correct result instead of the full 40.

Storing Points on the Prediction Itself

Extending Chapter 5's own `Prediction` model with one more column:

```
# models.py – added to the Prediction class from Chapter 5
points_awarded = Column(Integer, nullable=True) # null until a real result exists
```

STORED ONCE, NOT RECOMPUTED ON EVERY READ

Chapter 8's own prediction league table needs to sum points across potentially hundreds of predictions every time it's requested. Storing `points_awarded` directly on each `Prediction` row the moment a result is entered turns that later aggregation into a plain `SUM(points_awarded)` — exactly the kind of real PostgreSQL aggregate query Chapter 1 picked this stack for — rather than re-running `score_prediction()` against every row on every single page load.

Entering a Result

```
# schemas.py (additions)
from pydantic import Field

class ResultUpdate(BaseModel):
    home_score: int = Field(ge=0)
    away_score: int = Field(ge=0)
```

```
# routers/fixtures.py (additions)
from scoring import score_prediction

@router.patch("/fixtures/{fixture_id}/result", response_model=schemas.FixtureResponse)
def enter_result(fixture_id: int, payload: schemas.ResultUpdate, db: Session = Depends(
    fixture = db.get(models.Fixture, fixture_id)
```

```

if not fixture:
    raise HTTPException(status_code=404, detail="Fixture not found")

fixture.home_score = payload.home_score
fixture.away_score = payload.away_score
fixture.status = "played"

predictions = (
    db.query(models.Prediction).filter(models.Prediction.fixture_id == fixture_id)..
)
for prediction in predictions:
    prediction.points_awarded = score_prediction(
        prediction.predicted_home_score,
        prediction.predicted_away_score,
        payload.home_score,
        payload.away_score,
    )

db.commit()
db.refresh(fixture)
return fixture

```

PATCH, not **PUT** — this route changes two specific fields and derives a third (**status**), it doesn't replace the whole fixture, which is exactly what **PATCH** is for.

CORRECTING A WRONG RESULT RESCORING IS A FEATURE, NOT A SIDE EFFECT

scratch, every single time it's called — not just the first time. That means fixing a mistyped scoreline (2-1 entered when the real result was 2-2) genuinely re-scores every user, expert, guest, and AI prediction correctly against the corrected result, rather than leaving stale points from the wrong scoreline sitting in the database. A fixture with zero predictions recorded simply loops over an empty list and does nothing extra — no special case needed.

Resolving Chapter 5's Deferred Question: Average the Points, Not the Scorelines

Chapter 5 left one thing open: multiple guests predicting the same fixture produce multiple real scorelines that don't average into another valid scoreline — 2-1 and 3-0 average to 2.5-0.5, which isn't a result anything could actually finish. This chapter resolves it, and the fix is simpler than it first looks: **don't average the scorelines at all — score each guest individually, exactly like every other source, then average the resulting *points*.**

POINTS ARE JUST NUMBERS — THEY ALWAYS AVERAGE CLEANLY

Every guest's own `Prediction` row gets its own real `points_awarded` from the identical `score_prediction()` function used for user, expert, and AI — 40, 10, or 0, same as anyone else. A fixture with three guests that week now has three separate, real point values (say, 40, 10, and 0).

Averaging those three numbers into one figure — 16.67 — is completely well-defined, in a way averaging their three original scorelines never could be. Chapter 8's own prediction league table is where this average actually gets computed and folded into the guest predictor's own season total, one averaged figure per fixture rather than one raw contribution per individual guest — deliberately, so a fixture with three guests doesn't count three times as much toward the season total as a fixture with only one.

Predictions Now Include Their Own Points

Extending Chapter 5's own `PredictionResponse` with the new field:

```
class PredictionResponse(BaseModel):
    id: int
    fixture_id: int
    source: models.PredictionSource
    guest_name: Optional[str]
    predicted_home_score: int
    predicted_away_score: int
    points_awarded: Optional[int] # null until enter_result has run for this fixture

    class Config:
        from_attributes = True
```

`GET /api/fixtures/42/predictions` after a 2-1 result is entered, reusing the same five predictions from Chapter 5's own worked example:

```
[
  { "source": "user",    "guest_name": null,          "predicted_home_score": 2,
  { "source": "expert", "guest_name": null,          "predicted_home_score": 1,
  { "source": "guest",  "guest_name": "Micah Richards", "predicted_home_score": 3, "p
  { "source": "guest",  "guest_name": "Jamie Carragher", "predicted_home_score": 1, "p
  { "source": "ai",     "guest_name": null,          "predicted_home_score": 2,
]
```

The real result was 2-1, a home win. The user's exact 2-1 earns the full 40. The expert's 1-1 is a draw — a different outcome from the real home win, so it earns nothing despite being close on paper. Both guests and the AI predicted a home win without the exact score (3-0, 1-0, and 2-0 respectively), so each earns 10. The guest average for this fixture is therefore $(10 + 10) / 2 = 10$, the figure Chapter 8 would fold into the guest predictor's own season total for this one fixture.

Where This Course Is Headed

The real league table — points, goal difference, wins/draws/losses, built from every fixture this chapter has now filled in a real result for (Chapter 7); the prediction league table, aggregating every source's own `points_awarded` across the season, including the guest-averaging-by-fixture this chapter just resolved (Chapter 8); and promotion/relegation (Chapter 9).

Hands-On Exercises

EXERCISE 1

Explain why `score_prediction` checks for an exact scoreline match before checking the outcome, and what would go wrong (in terms of points actually awarded) if the outcome check ran first instead.

 [View solution](#)

EXERCISE 2

Explain why this chapter resolves the guest-averaging problem by averaging each guest's own points rather than averaging their raw predicted scorelines, using a concrete example of two guest scorelines

that cannot be meaningfully averaged directly.

 [View solution](#)

EXERCISE 3

Enter an incorrect result for a real fixture with at least two predictions already recorded, confirm their `points_awarded` values, then call `PATCH` again with the corrected result and confirm every prediction's `points_awarded` actually changed to reflect the correction.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **Real point values (confirmed)** — 40 points for a correct score, 10 points for a correct result, 0 otherwise
- **`score_prediction()`** — one pure function, shared by every source, exact-match checked before outcome
- **`Prediction.points_awarded`** — stored on the row itself, null until a result exists, so Chapter 8's own aggregation is a plain SUM
- **`PATCH /api/fixtures/{id}/result`** — fills in the real scoreline, flips status to "played," and rescores every prediction on the fixture from scratch
- **Correcting a result** — genuinely re-scores everything, not just the first entry
- **Guest averaging, resolved** — average each guest's own points per fixture, not their raw scorelines, which don't average into anything valid
- **Next chapter:** The real league table — points, goal difference, wins/draws/losses

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

Premier League Predictor: FastAPI & PostgreSQL

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

Chapter 6 scored predictions against real results. This chapter is a genuinely separate table: the actual Premier League standings — played, won, drawn, lost, goal difference, points — built entirely from the real results those fixtures now carry, with no reference to anyone's predictions at all.

Two Roles, One Team: Why This Isn't a Simple GROUP BY

A team's own record depends on fixtures where it appears as *either* `home_team_id` or `away_team_id` — two structurally different columns holding the same underlying fact ("this team played"). A plain `GROUP BY home_team_id` would only ever capture half of every team's real season. The fix is to normalize both roles into one shared shape first, with a `UNION ALL`, before aggregating anything:

```
WITH team_fixtures AS (
    SELECT home_team_id AS team_id, home_score AS goals_for, away_score AS goals_against
    FROM fixtures
    JOIN gameweeks ON gameweeks.id = fixtures.gameweek_id
    WHERE gameweeks.season_id = :season_id AND fixtures.status = 'played'

    UNION ALL

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

Every played fixture contributes exactly two rows to `team_fixtures` — one from each side's own perspective, with `goals_for` / `goals_against` already flipped correctly for whichever team

the row represents. From here, a single, ordinary `GROUP BY team_id` sees a team's *entire* season, home and away combined.

Starting From SeasonTeam, Not From Fixtures

A team with zero played fixtures — the very start of a season, or simply a gameweek not yet reached — should still appear in the table, with a real, honest zero-played row, not vanish because it has no `team_fixtures` rows to aggregate. Building the table by starting from `fixtures` and working backward to teams would silently drop exactly those teams. Starting from `season_teams` instead — Chapter 2's own real list of the 20 competing teams — and `LEFT JOIN`-ing the fixture data onto it keeps every team present regardless of how many games it's actually played:

```
# routers/tables.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy import text
from sqlalchemy.orm import Session
from database import get_db
import models, schemas

router = APIRouter(prefix="/api", tags=["tables"])

LEAGUE_TABLE_QUERY = text("""
    WITH team_fixtures AS (
        SELECT home_team_id AS team_id, home_score AS goals_for, away_score AS goals_ag
        FROM fixtures
        JOIN gameweeks ON gameweeks.id = fixtures.gameweek_id
        WHERE gameweeks.season_id = :season_id AND fixtures.status = 'played'
        UNION ALL
        SELECT away_team_id AS team_id, away_score AS goals_for, home_score AS goals_ag
        FROM fixtures
        JOIN gameweeks ON gameweeks.id = fixtures.gameweek_id
        WHERE gameweeks.season_id = :season_id AND fixtures.status = 'played'
    )
    SELECT
        teams.id AS team_id,
        teams.name,
        teams.short_name,
        COUNT(team_fixtures.team_id) AS played,
```

```

        COALESCE(SUM(CASE WHEN team_fixtures.goals_for > team_fixtures.goals_against TH
        COALESCE(SUM(CASE WHEN team_fixtures.goals_for = team_fixtures.goals_against TH
        COALESCE(SUM(CASE WHEN team_fixtures.goals_for < team_fixtures.goals_against TH
        COALESCE(SUM(team_fixtures.goals_for), 0) AS goals_for,
        COALESCE(SUM(team_fixtures.goals_against), 0) AS goals_against,
        COALESCE(SUM(team_fixtures.goals_for), 0) - COALESCE(SUM(team_fixtures.goals_ag
        COALESCE(SUM(CASE
            WHEN team_fixtures.goals_for > team_fixtures.goals_against THEN 3
            WHEN team_fixtures.goals_for = team_fixtures.goals_against THEN 1
            ELSE 0
        END), 0) AS points
    FROM season_teams
    JOIN teams ON teams.id = season_teams.team_id
    LEFT JOIN team_fixtures ON team_fixtures.team_id = teams.id
    WHERE season_teams.season_id = :season_id
    GROUP BY teams.id, teams.name, teams.short_name
    ORDER BY points DESC, goal_difference DESC, goals_for DESC, teams.name ASC
    """)

```

```

@router.get("/seasons/{season_id}/table", response_model=list[schemas.LeagueTableRow])
def get_league_table(season_id: int, db: Session = Depends(get_db)):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")

    rows = db.execute(LEAGUE_TABLE_QUERY, {"season_id": season_id}).mappings().all()
    return [schemas.LeagueTableRow(**dict(row)) for row in rows]

```

```

# schemas.py (additions)
class LeagueTableRow(BaseModel):
    team_id: int
    name: str
    short_name: str
    played: int
    won: int
    drawn: int
    lost: int
    goals_for: int
    goals_against: int

```

```
goal_difference: int
points: int
```

THE SAME LEFT JOIN REASONING AS CHAPTER 3, APPLIED TO A WHOLE TABLE THIS TIME

Chapter 3 already made this exact call once, at smaller scale: `GET /api/seasons/{id}/teams` joins from `Team` outward through `SeasonTeam`, not the other way around, precisely so a team's own existence never depends on something else (a fixture, a prediction) having happened yet. This query is the same principle at the scale of an entire table — `season_teams` is the real source of truth for "which 20 teams get a row," and `LEFT JOIN` lets the fixture data fill in around that membership rather than define it.

WHY RAW SQL HERE, WHEN EVERY EARLIER CHAPTER USED THE ORM

Every route before this one mapped cleanly onto SQLAlchemy's own query builder — filters, joins, counts. A `UNION ALL` feeding a multi-branch `CASE`-based aggregate genuinely doesn't map as cleanly; forcing it through the ORM's query API would produce something harder to read than the SQL it's generating underneath. `text()` with bound parameters (`:season_id`), never raw string interpolation — the same discipline as every parameterized query so far in this course, not a special exception for raw SQL) is the more honest choice for a query this genuinely relational, and `.mappings()` returns each row as a real dict-like object instead of a positional tuple, which is what lets `schemas.LeagueTableRow(**dict(row))` unpack it directly by column name.

REAL PREMIER LEAGUE TIEBREAKERS GO FURTHER THAN THIS TABLE DOES

The actual Premier League breaks a tie on points and goal difference using head-to-head results between the tied teams, then head-to-head goal difference, before finally reaching for fair play record. This table stops at points → goal difference → goals for → team name — a genuine, deliberate simplification for a personal prediction tracker, not an oversight. A real tie beyond goals for would be ordered alphabetically here, which won't always match the actual official table in a genuinely close season.

Rendering the Table

```
// static/table.js
async function loadLeagueTable(seasonId) {
  const res = await fetch(`/api/seasons/${seasonId}/table`);
```

```

const rows = await res.json();

const body = document.getElementById('table-body');
body.innerHTML = '';
rows.forEach((row, index) => {
  const tr = document.createElement('tr');
  tr.innerHTML = `
    <td>${index + 1}</td>
    <td>${row.short_name}</td>
    <td>${row.played}</td>
    <td>${row.won}</td>
    <td>${row.drawn}</td>
    <td>${row.lost}</td>
    <td>${row.goal_difference}</td>
    <td>${row.points}</td>
  `;
  body.appendChild(tr);
});
}

```

The table's own rank is just the row's own position in the already-sorted response — `index + 1` — since `ORDER BY points DESC, goal_difference DESC, goals_for DESC, teams.name ASC` in the SQL query already did the real work; the frontend never re-sorts anything.

Where This Course Is Headed

A second table, built on a genuinely different question — not "how good is each team," but "how good is each predictor" — aggregating every source's own `points_awarded` from Chapter 6 across the season, including the guest-averaging-by-fixture that chapter resolved (Chapter 8); promotion and relegation, which will remove and add real rows in `season_teams` — exactly the table this query already reads from (Chapter 9); and a real gameweek/season selector tying every chapter's own routes together in one interface (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why `team_fixtures` is built with a `UNION ALL` rather than a single `SELECT` with an `OR` condition on `home_team_id/away_team_id`, and what would go wrong with a team's own `goals_for/goals_against` totals if the `UNION ALL`'s two branches didn't flip which score column is "for" and which is "against."

 [View solution](#)

EXERCISE 2

Explain what would happen to a newly promoted team's own row in the league table, at the very start of a season before it has played any fixtures, if the query started from fixtures instead of from `season_teams`.

 [View solution](#)

EXERCISE 3

Enter real results for at least three fixtures in a real season (some involving the same team as both home and away across different gameweeks), call `GET /api/seasons/{id}/table`, and confirm by hand that at least one team's `played/won/drawn/lost/points` figures correctly combine both its home and away results.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **UNION ALL** — normalizes home and away fixture rows into one shared `team_id/goals_for/goals_against` shape before any aggregation happens
- **Starts from `season_teams`** — a team with zero played fixtures still gets a real, honest zero-played row, echoing Chapter 3's own `LEFT JOIN` reasoning
- **Real sort order** — points `DESC`, `goal_difference DESC`, `goals_for DESC`, `team name ASC`
- **Raw SQL via `text()`** — a deliberate choice for this one genuinely complex aggregate, still using bound parameters, not string interpolation
- **Real limit** — no head-to-head or fair-play tiebreakers; ties beyond goals for resolve alphabetically

- **Next chapter:** The prediction league table — ranking predictor performance, not team performance

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

Premier League Predictor: FastAPI & PostgreSQL

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

Chapter 7 answered "how good is each team." This chapter answers a genuinely different question: "how good is each predictor" — the user, the BBC's expert, that season's guests, and the BBC's AI — aggregated across every fixture Chapter 6 has scored.

Scoping Predictions to a Season, and Only Counting What's Actually Scored

```
WITH season_predictions AS (
    SELECT
        predictions.source,
        predictions.fixture_id,
        predictions.points_awarded
    FROM predictions
    JOIN fixtures ON fixtures.id = predictions.fixture_id
    JOIN gameweeks ON gameweeks.id = fixtures.gameweek_id
    WHERE gameweeks.season_id = :season_id
        AND predictions.points_awarded IS NOT NULL
)
```

`predictions.points_awarded IS NOT NULL` is the real gate: a prediction on a fixture that hasn't kicked off yet still has `points_awarded = NULL` (Chapter 6), so it never contributes to anyone's season total until a real result actually exists for it.

Three Straightforward Sources, One That Needs Two Passes

User, expert, and AI each contribute exactly one `Prediction` row per fixture (Chapter 5's own partial unique index guarantees it) — a single `SUM(points_awarded)` is their entire season total. Guest is different: Chapter 6 resolved that a fixture's own "guest" contribution is the *average* of however many individual guests predicted it, not a raw sum of every guest row. That means

guest needs its own points averaged *per fixture* first, before those per-fixture averages get summed across the season — genuinely two passes of aggregation, not one.

```
# routers/tables.py (additions)
PREDICTION_LEAGUE_QUERY = text("""
    WITH season_predictions AS (
        SELECT predictions.source, predictions.fixture_id, predictions.points_awarded
        FROM predictions
        JOIN fixtures ON fixtures.id = predictions.fixture_id
        JOIN gameweeks ON gameweeks.id = fixtures.gameweek_id
        WHERE gameweeks.season_id = :season_id AND predictions.points_awarded IS NOT NU
    ),
    non_guest_totals AS (
        SELECT
            source,
            COUNT(*) AS predictions_scored,
            SUM(points_awarded)::float AS total_points,
            SUM(CASE WHEN points_awarded = 40 THEN 1 ELSE 0 END) AS correct_scores,
            SUM(CASE WHEN points_awarded = 10 THEN 1 ELSE 0 END) AS correct_results
        FROM season_predictions
        WHERE source != 'guest'
        GROUP BY source
    ),
    guest_per_fixture AS (
        SELECT fixture_id, AVG(points_awarded) AS avg_points
        FROM season_predictions
        WHERE source = 'guest'
        GROUP BY fixture_id
    ),
    guest_totals AS (
        SELECT
            'guest' AS source,
            COUNT(*) AS predictions_scored,
            SUM(avg_points)::float AS total_points,
            NULL::int AS correct_scores,
            NULL::int AS correct_results
        FROM guest_per_fixture
    )
    SELECT * FROM non_guest_totals
    UNION ALL
    SELECT * FROM guest_totals
```

```

ORDER BY total_points DESC
"""

@router.get("/seasons/{season_id}/prediction-table", response_model=list[schemas.PredictionLeagueRow])
def get_prediction_table(season_id: int, db: Session = Depends(get_db)):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")

    rows = db.execute(PREDICTION_LEAGUE_QUERY, {"season_id": season_id}).mappings().all()
    return [schemas.PredictionLeagueRow(**dict(row)) for row in rows]

```

```

# schemas.py (additions)
class PredictionLeagueRow(BaseModel):
    source: models.PredictionSource
    predictions_scored: int
    total_points: float
    correct_scores: Optional[int]
    correct_results: Optional[int]

```

NULL::INT, NOT JUST NULL — A REAL UNION ALL TYPING REQUIREMENT

`UNION ALL` requires every corresponding column across its branches to share a compatible type. `non_guest_totals.correct_scores` is a real integer count; `guest_totals` has no meaningful equivalent to put there (explained below), but a bare, untyped `NULL` gives PostgreSQL nothing to reconcile against the integer column from the other branch. `NULL::int` is an explicitly-typed null — genuinely absent, but typed as an integer specifically so the two branches line up.

"CORRECT SCORES" AND "CORRECT RESULTS" DON'T TRANSLATE CLEANLY ONTO AN AVERAGED FIGURE

For user, expert, and AI, "correct scores" and "correct results" are meaningful counts — each source has exactly one real prediction per fixture, so counting how many of those hit 40 or 10 points is a genuine, well-defined statistic. Guest doesn't have that: the season-total figure being aggregated is already an *average* of however many individual guest predictions existed for a given fixture, and an average like 16.67 was never a 40-or-10-or-0 outcome to begin with — it's not "sometimes a correct score, sometimes not," it's a number that was never in that category at all. Rather than inventing an

arbitrary, misleading way to force a correct-score/correct-result count onto the guest row, this query leaves both columns honestly null for it.

PREDICTIONS_SCORED MEANS SOMETHING DIFFERENT FOR THE GUEST ROW

For user/expert/AI, `predictions_scored` counts real, individual `Prediction` rows — one per fixture, always. For guest, it counts rows from `guest_per_fixture` — one per *fixture that had at least one guest prediction*, regardless of whether that fixture had one guest or five. A season with 30 played fixtures, all with exactly one guest each, and a season with 30 played fixtures, all with three guests each, would report the identical `predictions_scored: 30` for guest — a deliberate consequence of scoring "the guest slot," not "every individual guest," matching Chapter 6's own resolution directly.

Rendering the Prediction Table

```
// static/prediction-table.js
const SOURCE_LABELS = {
  user: 'You',
  expert: 'BBC Expert',
  guest: 'Guest(s)',
  ai: 'BBC AI',
};

async function loadPredictionTable(seasonId) {
  const res = await fetch(`/api/seasons/${seasonId}/prediction-table`);
  const rows = await res.json();

  const body = document.getElementById('prediction-table-body');
  body.innerHTML = '';
  rows.forEach((row, index) => {
    const tr = document.createElement('tr');
    const breakdown = row.correct_scores === null
      ? '—'
      : `${row.correct_scores} exact / ${row.correct_results} result`;
    tr.innerHTML = `
      <td>${index + 1}</td>
      <td>${SOURCE_LABELS[row.source]}</td>
      <td>${row.predictions_scored}</td>
      <td>${breakdown}</td>
      <td>${row.total_points.toFixed(1)}</td>
    `;
  });
}
```

```

    `;
    body.appendChild(tr);
  });
}

```

`row.correct_scores === null` is what lets the frontend show a plain "—" for the guest row's own breakdown column, rather than a misleading "0 exact / 0 result" that would look like guests never got anything right.

Where This Course Is Headed

Promotion and relegation between seasons — real changes to `season_teams`, the same table both this chapter's and Chapter 7's own queries read from (Chapter 9); and a real gameweek/season selector, tying every route built across this course into one working interface (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why `guest_totals` needs its own separate CTE built on top of `guest_per_fixture`, rather than just adding a `WHERE source = 'guest'` branch alongside `non_guest_totals` using the same `SUM(points_awarded)` pattern.

 [View solution](#)

EXERCISE 2

Explain why `correct_scores` and `correct_results` are left as `NULL::int` for the guest row instead of being computed the same way as the other three sources, and why a bare `NULL` (without the `::int` cast) wouldn't work in this specific query.

 [View solution](#)

EXERCISE 3

Record and score predictions from all four sources across at least two fixtures, where one fixture has two guests and the other has only one, then call GET /api/seasons/{id}/prediction-table and confirm the guest row's predictions_scored equals 2, not 3.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **A genuinely different table from Chapter 7** — ranks predictors, not teams, built from points_awarded (Chapter 6)
- **points_awarded IS NOT NULL** — the real gate keeping unplayed fixtures out of anyone's season total
- **User/expert/AI** — a plain SUM(points_awarded) per source, one prediction per fixture, guaranteed by Chapter 5's partial unique index
- **Guest** — averaged per fixture first (guest_per_fixture), then summed across the season — resolving Chapter 6's own deferred question
- **NULL::int** — an explicitly-typed null, required so UNION ALL's branches line up on column type
- **correct_scores/correct_results** — meaningful for user/expert/AI, honestly left null for guest since an averaged figure was never a 40-or-10-or-0 outcome to begin with
- **Next chapter:** Promotion & relegation — real changes to season_teams, the table both league tables read from

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

Premier League Predictor: FastAPI & PostgreSQL

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

Chapter 3's own tip-box previewed this chapter directly: promotion and relegation is really just more `SeasonTeam` writes, at a season boundary instead of mid-season. What's genuinely new here is *where the bottom three come from* — not a guess, not a manual entry, but Chapter 7's own real league table, reused as the actual source of truth.

Reusing Chapter 7's Table as the Source of Truth

Rather than duplicating the league-table SQL, Chapter 7's own route gets a small, honest refactor — its query logic pulled into a plain function both it and this chapter's new routes can call:

```
# routers/tables.py (refactored)
def compute_league_table(db: Session, season_id: int) -> list[dict]:
    rows = db.execute(LEAGUE_TABLE_QUERY, {"season_id": season_id}).mappings().all()
    return [dict(row) for row in rows]

@router.get("/seasons/{season_id}/table", response_model=list[schemas.LeagueTableRow])
def get_league_table(season_id: int, db: Session = Depends(get_db)):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")
    return compute_league_table(db, season_id)
```

`compute_league_table` still runs the exact same `ORDER BY points DESC, goal_difference DESC, goals_for DESC, teams.name ASC` from Chapter 7 — which matters here more than it did there, since this chapter is about to trust that ordering to identify real relegation.

Previewing the Bottom Three Before Committing to Anything

```
# routers/rollover.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from sqlalchemy.exc import IntegrityError
from database import get_db
from routers.tables import compute_league_table
import models, schemas

router = APIRouter(prefix="/api", tags=["rollover"])

REQUIRED_TEAMS_PER_SEASON = 20
RELEGATION_COUNT = 3

@router.get("/seasons/{season_id}/relegation-preview", response_model=list[schemas.Team])
def preview_relegation(season_id: int, db: Session = Depends(get_db)):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")

    table = compute_league_table(db, season_id)
    if len(table) != REQUIRED_TEAMS_PER_SEASON:
        raise HTTPException(
            status_code=400,
            detail=f"Season has {len(table)} teams, expected {REQUIRED_TEAMS_PER_SEASON}"
        )

    bottom_three_ids = [row["team_id"] for row in table[-RELEGATION_COUNT:]]
    return (
        db.query(models.Team)
        .filter(models.Team.id.in_(bottom_three_ids))
        .all()
    )
```

A plain `GET` — nothing about calling it changes any data. Checking this before the real rollover is exactly how a mistake (a wrong result entered somewhere, silently shifting who's actually bottom three) gets caught before it's baked into next season's own competing set.

Rolling Over a Season

```
# schemas.py (additions)
from pydantic import field_validator

class PromotedTeam(BaseModel):
    team_name: str
    short_name: Optional[str] = None

class RolloverRequest(BaseModel):
    promoted_teams: list[PromotedTeam]

    @field_validator("promoted_teams")
    @classmethod
    def exactly_three(cls, value):
        if len(value) != 3:
            raise ValueError("Exactly 3 promoted teams are required")
        return value
```

```
# routers/rollover.py (additions)
@router.post("/seasons/{new_season_id}/rollover-from/{old_season_id}", response_model=List[Season])
def rollover_season(
    new_season_id: int, old_season_id: int, payload: schemas.RolloverRequest, db: Session
):
    old_season = db.get(models.Season, old_season_id)
    new_season = db.get(models.Season, new_season_id)
    if not old_season or not new_season:
        raise HTTPException(status_code=404, detail="Season not found")

    existing_new_season_teams = (
        db.query(models.SeasonTeam).filter(models.SeasonTeam.season_id == new_season_id)
    )
    if existing_new_season_teams > 0:
        raise HTTPException(
            status_code=400,
            detail="New season already has teams; rollover expects a freshly created, empty season"
        )

    old_table = compute_league_table(db, old_season_id)
```

```

if len(old_table) != REQUIRED_TEAMS_PER_SEASON:
    raise HTTPException(
        status_code=400,
        detail=f"Old season has {len(old_table)} teams, expected {REQUIRED_TEAMS_PER_SEASON}"
    )

# old_table is already correctly ordered by compute_league_table's own real sort
survivors = old_table[:-RELEGATION_COUNT]

for row in survivors:
    db.add(models.SeasonTeam(season_id=new_season_id, team_id=row["team_id"]))

for promoted in payload.promoted_teams:
    team = db.query(models.Team).filter(models.Team.name == promoted.team_name).first()
    if not team:
        team = models.Team(name=promoted.team_name, short_name=promoted.short_name)
        db.add(team)
        db.flush()
    db.add(models.SeasonTeam(season_id=new_season_id, team_id=team.id))

try:
    db.commit()
except IntegrityError:
    db.rollback()
    raise HTTPException(
        status_code=409,
        detail="This rollover could not be completed cleanly – check whether it already exists"
    )

return (
    db.query(models.Team)
    .join(models.SeasonTeam, models.SeasonTeam.team_id == models.Team.id)
    .filter(models.SeasonTeam.season_id == new_season_id)
    .order_by(models.Team.name)
    .all()
)

```

"REMOVING" A RELEGATED TEAM DOESN'T DELETE ANYTHING

Nothing about this route touches `old_season`'s own `SeasonTeam` rows at all — the relegated three keep their real, historical record of having competed in that season, exactly as Chapter 2's own "Team

rows persist forever" design intended. "Removing" a relegated team means exactly one thing here: it simply never gets a matching `SeasonTeam` row created for `new_season`. `survivors` — every row in `old_table` except the bottom three — is what actually gets carried forward; the omission itself is the entire relegation mechanism.

THE SAME FRIENDLY-INTEGRITYERROR PATTERN AS CHAPTER 4, ON A BIGGER OPERATION

If this route is accidentally called twice for the same `new_season_id`, the second call would try to insert 20 `SeasonTeam` rows that already exist, colliding with Chapter 2's own `UniqueConstraint("season_id", "team_id")`. The `try/except IntegrityError` here is the exact same discipline Chapter 4 established for a single duplicate gameweek, just wrapping an operation that touches 20 rows instead of one — catching a real constraint violation and turning it into one clean `409` instead of a raw crash partway through.

NO UNDO, AND NO PARTIAL-ROLLOVER RECOVERY

If the promoted teams' own names are entered wrong, there's no dedicated "undo the rollover" route — fixing it means manually removing the incorrect `SeasonTeam` rows (Chapter 3's own `DELETE /api/seasons/{id}/teams/{team_id}`) and adding the correct ones by hand. This route also assumes the whole operation either fully succeeds or is caught cleanly by the `IntegrityError` case above — a real database failure partway through committing (rare, but possible) isn't specially handled beyond what PostgreSQL's own transaction already guarantees.

Using It From the Frontend

```
// static/rollover.js
async function previewRelegation(seasonId) {
  const res = await fetch(`/api/seasons/${seasonId}/relegation-preview`);
  const teams = await res.json();
  alert('Relegated: ' + teams.map(t => t.short_name).join(', '));
}

async function rolloverSeason(newSeasonId, oldSeasonId, promotedTeams) {
  const res = await fetch(`/api/seasons/${newSeasonId}/rollover-from/${oldSeasonId}`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ promoted_teams: promotedTeams }),
  });
}
```

```

if (!res.ok) {
  const error = await res.json();
  alert(error.detail);
  return;
}

const roster = await res.json();
alert(`New season roster: ${roster.length} teams`);
}

```

Where This Course Is Headed

A real gameweek/season selector — replacing every hardcoded `seasonId` / `gameweekId` constant this course has used since Chapter 4 with a genuine dropdown backed by `GET /api/seasons`, tying every route built across the whole course into one working interface (Chapter 10); deployment (Chapter 11); and a capstone integrating the finished predictor into the existing Astro-based site (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain why survivors is computed as `old_table[: -RELEGATION_COUNT]` rather than by querying the database a second time for "every team except the bottom three," and what real guarantee this relies on from `compute_league_table`.

 [View solution](#)

EXERCISE 2

Explain what "removing" a relegated team from next season's `SeasonTeam` set actually means in this implementation, and confirm whether that team's own historical record in the old season is affected at all.

 [View solution](#)

EXERCISE 3

Complete a full real season (enter results for all fixtures needed to give every team a real record), call the relegation preview, create a new season, and roll it over with three real promoted team names — then confirm the new season's roster has exactly 20 teams: 17 real survivors plus the 3 promoted teams.

 [View solution](#)
CHAPTER 9 QUICK REFERENCE

- **compute_league_table()** — Chapter 7's own logic, extracted so this chapter can reuse it as the real source of truth for relegation
- **GET /api/seasons/{id}/relegation-preview** — a read-only sanity check before anything real is committed
- **POST /api/seasons/{new}/rollover-from/{old}** — carries the surviving 17 teams forward, reuses Chapter 3's team-lookup pattern for the 3 promoted teams
- **"Removing" a relegated team** — never deletes anything; the old season's own history is untouched, the team simply gets no SeasonTeam row in the new season
- **Precondition** — the new season must start with zero SeasonTeam rows; the old season must have exactly 20
- **Real limit** — no dedicated undo route; a double-call is caught by the same friendly-IntegrityError pattern from Chapter 4
- **Next chapter:** A real gameweek/season selector, replacing every hardcoded ID this course has used since Chapter 4

CHAPTER 10: STYLING & THE GAMEWEEK/SEASON SELECTOR UI

Premier League Predictor: FastAPI & PostgreSQL

Chapter 10 · Styling & the Gameweek/Season Selector UI

Every real route this course has built since Chapter 4 has been exercised against a hardcoded `seasonId` / `gameweekId` pair, deliberately left that way to keep each chapter's own example focused. This chapter replaces every one of those constants with a real, working dropdown selector — and gives the whole app the dark-theme styling it's been missing since Chapter 1.

Two Small List Routes This Course Never Quite Needed Until Now

```
# routers/seasons.py (additions)
@router.get("/seasons", response_model=list[schemas.SeasonResponse])
def list_seasons(db: Session = Depends(get_db)):
    return db.query(models.Season).order_by(models.Season.start_date.desc()).all()
```

```
# routers/gameweeks.py (additions)
@router.get("/seasons/{season_id}/gameweeks", response_model=list[schemas.GameweekResponse])
def list_gameweeks(season_id: int, db: Session = Depends(get_db)):
    season = db.get(models.Season, season_id)
    if not season:
        raise HTTPException(status_code=404, detail="Season not found")
    return (
        db.query(models.Gameweek)
        .filter(models.Gameweek.season_id == season_id)
        .order_by(models.Gameweek.number)
        .all()
    )
```

A Shared Selector, Broadcasting One Event

Every page built since Chapter 4 — fixtures, predictions, both league tables — needs to know the currently selected season and gameweek. Rather than having each of those pages reach directly into a selector's own internal variables, the selector broadcasts a single `CustomEvent` whenever the selection changes, and every page independently listens for it:

```
// static/selector.js
let selectedSeasonId = null;
let selectedGameweekId = null;

async function initSelectors() {
  const seasonSelect = document.getElementById('season-select');
  const gameweekSelect = document.getElementById('gameweek-select');

  const res = await fetch('/api/seasons');
  const seasons = await res.json();

  seasonSelect.innerHTML = '';
  seasons.forEach(season => {
    const option = document.createElement('option');
    option.value = season.id;
    option.textContent = season.name;
    if (season.is_current) option.selected = true;
    seasonSelect.appendChild(option);
  });

  seasonSelect.addEventListener('change', () => loadGameweeks(Number(seasonSelect.value));
  gameweekSelect.addEventListener('change', () => selectGameweek(Number(gameweekSelect.value));

  await loadGameweeks(Number(seasonSelect.value));
}

async function loadGameweeks(seasonId) {
  selectedSeasonId = seasonId;
  const gameweekSelect = document.getElementById('gameweek-select');

  const res = await fetch(`/api/seasons/${seasonId}/gameweeks`);
  const gameweeks = await res.json();

  gameweekSelect.innerHTML = '';
  gameweeks.forEach(gw => {
    const option = document.createElement('option');
```

```

    option.value = gw.id;
    option.textContent = `Gameweek ${gw.number}`;
    gameweekSelect.appendChild(option);
  });

  if (gameweeks.length > 0) selectGameweek(gameweeks[0].id);
}

function selectGameweek(gameweekId) {
  selectedGameweekId = gameweekId;
  window.dispatchEvent(new CustomEvent('selection-changed', {
    detail: { seasonId: selectedSeasonId, gameweekId: selectedGameweekId },
  }));
}

initSelectors();

```

WHY AN EVENT, NOT JUST EXPORTED FUNCTIONS OTHER SCRIPTS CALL DIRECTLY

`selector.js` could have exported `getSelectedSeasonId()` / `getSelectedGameweekId()` for every other script to call whenever it needs the current selection — but that would mean fixtures.js, predictions.js, and both table scripts would each need to know exactly when to call them (on page load? on some other event?), coupling every page's own logic to selector.js's own timing.

Broadcasting one `selection-changed` event instead means `selector.js` doesn't need to know or care what else is listening, and every other page only needs to know one event name — the same decoupling this course's own backend has leaned on repeatedly: Chapter 3's `SeasonTeam` as the one shared table two later chapters both read and write, Chapter 7's `compute_league_table()` reused directly rather than re-derived. This is that same idea, applied to the frontend.

Resolving Chapter 4's Own Flagged Gap, For Real

Chapter 4 was honest that `usedTeamIds` only lived in the page's own memory, resetting to empty on every reload and never reflecting fixtures already entered in an earlier visit. Wiring `fixtures.js` into the new `selection-changed` event is exactly the natural place to fix that — not by trusting a stale in-memory set, but by re-seeding it from the real server state every time the gameweek actually changes:

```
// static/fixtures.js (updated)
let seasonId = null;
let gameweekId = null;

window.addEventListener('selection-changed', async (event) => {
  seasonId = event.detail.seasonId;
  gameweekId = event.detail.gameweekId;
  await loadTeams();
  await seedUsedTeams();
});

async function seedUsedTeams() {
  const res = await fetch(`/api/gameweeks/${gameweekId}/fixtures`);
  const existingFixtures = await res.json();

  usedTeamIds = new Set();
  existingFixtures.forEach(fx => {
    usedTeamIds.add(fx.home_team_id);
    usedTeamIds.add(fx.away_team_id);
  });
  disableUsedButtons(); // the exact function Chapter 4 already defined
}
```

Reloading the page — or switching to a different gameweek and back — now reliably shows the real, correct set of already-used teams, fetched fresh from `GET /api/gameweeks/{id}/fixtures` (Chapter 4's own listing route) every time, rather than an empty set that happened to be correct only by coincidence of a fresh page load.

Real Styling

```
/* static/style.css */
body {
  background: #0d1117;
  color: #c9d1d9;
  font-family: system-ui, -apple-system, sans-serif;
  margin: 0;
  padding: 1.5rem;
}
```

```
select, button {
  background: #161b22;
  color: #c9d1d9;
  border: 1px solid #30363d;
  border-radius: 6px;
  padding: 0.5rem 0.8rem;
  font-size: 0.9rem;
  cursor: pointer;
}

button:hover:not(:disabled) { background: #30363d; }
button:disabled { opacity: 0.4; cursor: not-allowed; }

.team-grid {
  display: grid;
  grid-template-columns: repeat(5, 1fr);
  gap: 0.5rem;
  margin: 1rem 0;
}

.fixture-slots {
  display: flex;
  align-items: center;
  gap: 1rem;
  margin: 1rem 0;
}

.slot {
  background: #161b22;
  border: 1px dashed #30363d;
  border-radius: 6px;
  padding: 0.6rem 1.2rem;
  min-width: 120px;
  text-align: center;
}

table {
  width: 100%;
  border-collapse: collapse;
  margin: 1rem 0;
}
```

```

th, td {
  padding: 0.5rem 0.7rem;
  text-align: left;
  border-bottom: 1px solid #21262d;
}

th {
  background: #062120;
  color: #5eead4;
  text-transform: uppercase;
  font-size: 0.7rem;
  letter-spacing: 0.07em;
}

```

The same teal accent (`#00C7B7` / `#5EEAD4`) and dark surfaces (`#0d1117` / `#161b22`) used across every chapter of this course's own documentation — the finished app looks like a genuine continuation of the material teaching it, not a visually disconnected afterthought.

NOT A DESIGN SYSTEM, ON PURPOSE

This is real, working CSS — enough to make the app pleasant and legible — not a component library or a design system with reusable tokens beyond a couple of shared colors. A larger app would want that; a personal prediction tracker genuinely doesn't need it yet.

Where This Course Is Headed

Deployment — getting this app running somewhere real, not just `localhost` (Chapter 11); and a capstone integrating the finished predictor into the existing Astro-based site (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain why `selector.js` broadcasts a single `CustomEvent` rather than exporting `getSelectedSeasonId()/getSelectedGameweekId()` functions for other scripts to call, and name one other place in this course where a similar decoupling choice was already made.

 [View solution](#)

EXERCISE 2

Explain exactly how `seedUsedTeams` resolves the gap Chapter 4 flagged about `usedTeamIds` resetting on page reload, and what real request it makes to do so.

 [View solution](#)

EXERCISE 3

Build the selector into a real page with at least two seasons and two gameweeks to choose from, enter a fixture in one gameweek, switch to the other gameweek and back, and confirm the previously-entered fixture's own two teams are still shown as disabled after switching back — without reloading the page.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE

- **GET /api/seasons** and **GET /api/seasons/{id}/gameweeks** — the two small list routes this course never quite needed until the selector required them
- **selection-changed** — one CustomEvent broadcast whenever season/gameweek changes, decoupling selector.js from every page that reacts to it
- **seedUsedTeams()** — genuinely resolves Chapter 4's own flagged gap by re-fetching real fixture data on every gameweek switch
- **Styling** — real, working dark-theme CSS reusing this course's own documentation accent colors, deliberately not a full design system
- **Next chapter:** Deployment

CHAPTER 11: DEPLOYMENT

Premier League Predictor: FastAPI & PostgreSQL

Chapter 11 · Deployment

Every earlier chapter ran with `uvicorn main:app --reload` — a genuinely fine development server, and a genuinely wrong choice for real production traffic.

Running With Multiple Workers

```
pip install gunicorn
```

```
gunicorn main:app -k uvicorn.workers.UvicornWorker -w 4 --bind 0.0.0.0:8000
```

Gunicorn manages several separate Uvicorn worker **processes** — real request-handling capacity beyond what one process provides, with automatic restarts if a worker crashes.

NO CORS ANYWHERE IN THIS COURSE, FOR THE SAME REASON AS ITS OWN FASTAPI SIBLING

This site's own `fastapi-food-tracker-1` course names the exact same fact: mounting the frontend via `StaticFiles` on the same FastAPI app since Chapter 1 means the frontend and API have shared one origin in every environment, development included — there was never a cross-origin situation for CORS middleware to solve.

A Full-Circle Caveat on Chapter 1's Own Favorite Feature

Chapter 1 named automatic interactive docs at `/docs` as a genuine advantage this course's own reference-implementation choice earns for free. Before a real deployment, that feature deserves an honest second look — it lets anyone browsing it try real API calls, including writes, directly against the live database:

```
# main.py
import os
```

```
app = FastAPI(
    title="Premier League Predictor",
    docs_url="/docs" if os.getenv("ENVIRONMENT") != "production" else None,
    redoc_url=None,
)
```

THE SAME FEATURE, PRAISED IN CHAPTER 1, CAVEATED HERE

A genuinely useful development tool and a genuinely public API explorer aren't automatically the same thing to want live — especially in this course, where nothing gates `POST` / `PATCH` / `DELETE` routes behind any login (Chapter 3's own honest warning). Disabling `/docs` conditionally, as above, keeps Chapter 1's own advantage intact during development while making its production exposure a deliberate choice.

PostgreSQL Connections at Scale: A Real Math Problem

Unlike the FastAPI/SQLite sibling course — which has no real connection pool at all — SQLAlchemy's `create_engine()` against PostgreSQL genuinely does pool connections by default. That's a real advantage this course's own stack choice earns... but it introduces a different, equally real problem once `gunicorn` enters the picture.

EACH WORKER PROCESS GETS ITS OWN ENGINE, AND ITS OWN POOL

`database.py`'s own `engine = create_engine(DATABASE_URL)` runs once per Python process. With `-w 4`, gunicorn starts four completely separate OS processes, each importing `database.py` independently and creating its *own* engine with its own connection pool — they don't share one pool between them. SQLAlchemy's real defaults are `pool_size=5` plus `max_overflow=10`, meaning a single worker can open up to **15 real connections** to PostgreSQL under load. Four workers at that default: up to **60 real connections**, all from one deployed app — a meaningful fraction of PostgreSQL's own default `max_connections` of 100, before counting anything else (an admin tool, a monitoring agent, a second app on the same database) that might also be connecting.

A concrete, sized fix rather than an abstract warning — capping each worker's own pool explicitly:

```
# database.py (production-sized)
engine = create_engine(DATABASE_URL, pool_size=5, max_overflow=2)
```

Four workers at `pool_size=5, max_overflow=2`: up to **7 connections per worker, 28 total** at genuine peak — comfortably under the default 100-connection ceiling, with real headroom left for anything else connecting to the same database.

NEVER DEPLOY WITH CHAPTER 1'S OWN LOCAL FALLBACK DATABASE_URL

Chapter 1's own `os.getenv("DATABASE_URL", "postgresql://postgres:@localhost:5432/pl_predictor")` fallback has no password at all — fine for a local development database nobody outside the machine can reach, a real security problem for anything actually deployed. A real, strong `DATABASE_URL` — with a genuine username and password, and pointed at the real production database host — must be set as an actual environment variable on the deployment host; the local fallback should never be what's actually running in production.

Schema Changes After Deployment

Chapter 2's own `Base.metadata.create_all(bind=engine)` creates tables that don't yet exist — it does **not** alter a table that already exists. Adding a new column to `Fixture` after this app is already deployed and running against real data would need a genuine, manual `ALTER TABLE`, or a real migration tool like Alembic, neither of which this course builds. That's an honest, deliberate scope boundary for a personal project at this size, not an oversight — worth knowing about before it's actually needed rather than discovering it the hard way mid-season.

TLS Termination

Gunicorn itself doesn't handle TLS certificates — a reverse proxy (nginx, the same pattern this site's own Nginx In Depth course already covers) sits in front of it, terminating HTTPS and forwarding plain HTTP internally.

A Production Checklist

- Run with `gunicorn` + Uvicorn workers, never `--reload`.
- Decide deliberately whether `/docs` should be public, and disable it if not.
- Set a real `DATABASE_URL` with genuine credentials as an environment variable — never the local, passwordless fallback.

- Size `pool_size` / `max_overflow` against worker count and PostgreSQL's own `max_connections`, not the unmodified defaults.
- Know that a schema change after deployment needs a manual `ALTER TABLE` or a real migration tool — `create_all` alone won't do it.
- Put a reverse proxy in front for TLS termination.

Where This Course Is Headed

One chapter left: a capstone tying every route and page built across this course into one complete, working predictor.

Hands-On Exercises

EXERCISE 1

Explain why this course never needed CORS configuration anywhere, tracing the reason back to a decision made in Chapter 1.

[!\[\]\(87b1d3ef84917af7042aca35293ed0a4_img.jpg\) View solution](#)

EXERCISE 2

Explain why four gunicorn workers running with SQLAlchemy's default `pool_size`/`max_overflow` settings can collectively open up to 60 connections to PostgreSQL, not 15, and calculate the real total after applying this chapter's own `pool_size=5`, `max_overflow=2` fix.

[!\[\]\(865c38b637feb68f22a10b934b9e1536_img.jpg\) View solution](#)

EXERCISE 3

Explain what `Base.metadata.create_all` actually does and doesn't do, and describe what would genuinely need to happen to add a new column to the Fixture table after this app is already deployed and running.

[!\[\]\(c2302d0b34dac669c22e1f7de1975efa_img.jpg\) View solution](#)

CHAPTER 11 QUICK REFERENCE

- **Run with:** `gunicorn -k uvicorn.workers.UvicornWorker -w 4`, never `--reload` in production
- **No CORS anywhere:** StaticFiles has served the frontend same-origin since Chapter 1, same as the FastAPI/SQLite sibling course
- **Full-circle caveat:** `/docs`, Chapter 1's own praised feature, should be a deliberate production decision, especially given no route in this course is auth-gated
- **Real connection math:** 4 workers $\times$ unmodified defaults (`pool_size=5 + max_overflow=10`) = up to 60 real PostgreSQL connections; the same 4 workers at `pool_size=5/max_overflow=2` = up to 28
- **Never deploy the local fallback DATABASE_URL** — it has no password
- **Schema changes after deployment** need a manual `ALTER TABLE` or a real migration tool — `create_all` alone can't do it
- **TLS:** a reverse proxy in front, same pattern as this site's own Nginx In Depth course
- **Next chapter:** Capstone

CHAPTER 12: CAPSTONE — INTEGRATING THE PREDICTOR INTO THE EXISTING ASTRO SITE

Premier League Predictor: FastAPI & PostgreSQL

Chapter 12 (Capstone) · Integrating the Predictor Into the Existing Astro Site

Eleven chapters have built a genuinely working predictor — a real normalized schema, admin tooling, a fast fixture-entry UI, four scored prediction sources, two real league tables, promotion/relegation, a working selector, and a real deployment plan. This closing chapter covers the actual point of building it: getting it in front of the user, mounted onto their real, existing Astro-based site, then reviews the whole course chapter by chapter.

Three Real Ways to Mount a FastAPI App Onto an Astro Site

Astro's own live site is a static build served by a web server — it has no Python runtime of its own, and doesn't need one for its own pages. Integrating this project is a matter of getting both apps served correctly, side by side, behind the same web server:

OPTION	WHAT IT LOOKS LIKE	REAL EFFORT
Subdomain	<code>predictor.example.com</code> , a fully separate vhost	Lowest — no code changes at all, just DNS + a new nginx server block
Path-mounted reverse proxy	<code>example.com/predictor/</code> , same domain as the Astro site	Low — one nginx <code>location</code> block plus a real fix to this course's own JS fetch calls
JSON API + Astro frontend	FastAPI returns data only; an Astro page/island renders it	Highest — every page's own vanilla-JS logic effectively rebuilt in Astro/React

The Realistic Choice: Path-Mounted Reverse Proxy

A subdomain works but feels like more infrastructure than a personal project needs; a full JSON-API rewrite defeats much of the point of having already built a working vanilla-JS frontend across ten chapters. A path-mounted reverse proxy is the real middle ground:

```
/etc/nginx/sites-available/example.com (excerpt)
```

```

server {
    listen 443 ssl;
    server_name example.com;

    # The existing Astro static site
    root /var/www/example.com/dist;
    index index.html;

    location / {
        try_files $uri $uri/ =404;
    }

    # The predictor, mounted at /predictor/, proxied to Gunicorn
    location /predictor/ {
        proxy_pass http://127.0.0.1:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}

```

`proxy_pass http://127.0.0.1:8000/;` — the trailing slash matters — strips `/predictor/` before forwarding, so FastAPI itself sees plain `/api/seasons` requests exactly as every earlier chapter tested them, with no server-side routing changes needed at all.

THE REAL BUG: ABSOLUTE FETCH() PATHS SILENTLY BREAK, AND LOCALHOST NEVER CATCHES IT

Every script since Chapter 4 calls `fetch('/api/...')` — an **absolute** path. A browser resolves an absolute path from the site's own root, not from the current page's own folder. Loaded at `https://example.com/predictor/index.html`, a call to `fetch('/api/seasons')` resolves to `https://example.com/api/seasons` — a path nginx has no `location` block for at all, since only `/predictor/` and `/` exist. It falls through to the Astro site's own catch-all and returns Astro's 404 page instead of real JSON. This never shows up on `localhost:8000`, where the app already sits at the root — it only appears the moment the app is genuinely mounted at a sub-path, which is exactly why it's easy to miss until a real deployment surfaces it.

The fix, and it's a genuine one-time change: every `fetch()` call across `selector.js`, `fixtures.js`, `predictions.js`, both table scripts, and `rollover.js` drops its leading slash —

a relative path instead of an absolute one:

```
// Before (Chapter 4 onward) – absolute, breaks once mounted at a sub-path
fetch(`/api/seasons/${seasonId}/teams`);

// After – relative, resolves against the page's own URL
fetch(`api/seasons/${seasonId}/teams`);
```

A relative path resolves against the document's own **base URL** — and rather than depending on which folder each individual page happens to sit in, one shared `<base>` tag sets that base URL explicitly, for every page, in one place:

static/index.html (and every other page)

```
<head>
  <base href="/predictor/">
  ...
</head>
```

A THIRD GENUINELY DIFFERENT MECHANISM, SAME UNDERLYING ISSUE AS ITS TWO SIBLING COURSES

Personal Catalogue (PHP & MySQL)'s own capstone found absolute `header('Location: ...')` redirects breaking once mounted at a sub-path; Personal Catalogue (Django & PostgreSQL)'s own capstone found server-generated `{% url %}` links needing Django's real `FORCE_SCRIPT_NAME` setting. This course's own version of the identical underlying problem — "the app doesn't know it's not at the web root" — shows up in neither of those places, since nothing here redirects server-side or renders a template; it shows up in static JavaScript's own absolute `fetch()` paths instead, fixed not by a single backend setting but by a relative-path convention plus one shared `<base>` tag. Three real frameworks, three genuinely different failure points, one recurring lesson.

LOCAL DEVELOPMENT NEEDS NO BASE TAG AT ALL

On `localhost:8000`, the app already sits at the root — a page loaded from `http://localhost:8000/index.html` resolves `fetch('api/seasons')` to `http://localhost:8000/api/seasons` correctly with no `<base>` tag present at all. The tag only needs to exist (or be templated in) for the deployed, sub-path-mounted version — every relative fetch call itself never needs to change between the two environments.

Making It Feel Like Part of the Same Site

A reverse proxy solves reachability, not visual consistency. Chapter 10's own `style.css` already gives the predictor a real dark theme matching this course's own documentation — matching its accent colors to whatever the Astro site's own build actually uses is a small, one-time styling pass, and the natural place for a real "← Back to Main Site" link.

A Real Alternative for Deeper Integration: A JSON Endpoint

If, later, the current gameweek's own real league position or predictor standings need to appear directly on an Astro page rather than linking out, the reverse-proxy setup already built supports that too — FastAPI already returns JSON from every route in this course, so an Astro island could `fetch()` the exact same `/api/seasons/{id}/table` route this course already built, with no new backend code required at all. This isn't built out further here — it's flagged honestly as the real next step if the project's own scope ever grows toward a tighter Astro-embedded experience.

Capstone: What Each Chapter Actually Built

CHAPTER	WHAT IT CONTRIBUTED
1	The real project spec (4 real prediction sources, two league tables, promotion/relegation) and a working FastAPI + SQLAlchemy + PostgreSQL connection
2	The real Team/Season/SeasonTeam/Gameweek/Fixture schema, the dual self-referencing foreign key fix on Fixture, and nullable scores as "not yet played"
3	Season/team admin routes, the reuse-vs-create Team lookup keeping a relegated-then-promoted club as one permanent row, and the 20-team cap
4	The click-to-pair fixture-entry UI, and the real server-side duplicate-team guard client-side disabling only assists
5	The Prediction table, a real PostgreSQL partial unique index allowing multiple guests but one user/expert/AI prediction, and the upsert pattern
6	The real, confirmed point values (40/10), <code>score_prediction()</code> , and the guest-averaging problem resolved by averaging points, not scorelines

CHAPTER	WHAT IT CONTRIBUTED
7	The real league table — a UNION ALL normalizing home/away rows, LEFT JOIN from SeasonTeam so an unplayed team still gets a real row
8	The prediction league table — a two-pass CTE averaging guest points per fixture before summing across the season
9	Promotion/relegation — reusing Chapter 7's own table as the real source of truth, a relegation preview, and an honest note that "removing" a team deletes nothing
10	The gameweek/season selector, and a genuine fix to Chapter 4's own flagged usedTeamIds gap
11	Deployment — gunicorn workers, a real connection-pool sizing calculation, and the /docs production caveat
12	Mounting the finished predictor behind the same domain as the real, existing Astro site

What's Still Genuinely Open

`project_issues.md`'s own P3 entry named several real questions this course deliberately didn't resolve — not oversights, but genuine product decisions left for whenever they're actually needed: whether past seasons get backfilled retroactively, whether a real fixture ever needs to move between gameweeks after being entered, and whether this stays a genuinely single-user tool or eventually needs real multi-user support (a question Chapter 3's own honest "no access control" warning already anticipated). None of these blocked building a genuinely working predictor — they're simply the honest list of what comes next, if it ever needs to.

Hands-On Exercises

EXERCISE 1

Write the nginx location `/predictor/` block from this chapter, update at least two real `fetch()` calls from earlier chapters to relative paths, add the `<base href="/predictor/">` tag, and confirm a request that previously would have resolved to the site root now correctly reaches the proxied FastAPI app instead.

 [View solution](#)

EXERCISE 2

Explain, in your own words, why this course's own fetch-path fix, Django's `FORCE_SCRIPT_NAME`, and PHP's absolute-vs-relative redirect fix are described as three genuinely different mechanisms solving the same underlying issue — what is that shared underlying issue?

 [View solution](#)

EXERCISE 3

Explain why the tag fix works without ever needing to change on the `localhost:8000` version of the app, tracing the explanation back to how a browser resolves a relative URL.

 [View solution](#)

CHAPTER 12 QUICK REFERENCE

- **Path-mounted reverse proxy** — one nginx `location /predictor/` block using `proxy_pass` to Unicorn, stripping the prefix
- **The real bug** — absolute `fetch('/api/...')` calls resolve from the site root, not the current page, and silently 404 once mounted at a sub-path
- **The fix** — relative fetch paths (no leading slash) plus one shared `<base href="/predictor/">` tag; localhost needs no base tag at all
- **Three sibling courses, three real mechanisms** — this course's relative-path/base-tag fix, Django's `FORCE_SCRIPT_NAME`, PHP's absolute-vs-relative redirect — one shared underlying issue
- **JSON endpoint** — already exists, for free, at every route this course built; a real, honest next step for tighter Astro-embedded integration
- **Still genuinely open** — historical backfill, fixture rescheduling workflow, and single-vs-multi-user support, per `project_issues.md` P3

Course Complete — Premier League Predictor: FastAPI & PostgreSQL (12/12 chapters)