

PERSONAL MULTI-TECHNOLOGY PROJECTS

Premier League Predictor: Django & MySQL

A real, working weekly Premier League prediction tracker — four scored prediction sources (user, BBC expert, guest(s) averaged, BBC AI), a real normalized MySQL schema built as Django models, the Django admin doing real work largely for free, two genuinely different league tables, promotion and relegation, and a real deployment mounted onto an existing Astro site — a direct sibling to this project's own FastAPI & PostgreSQL reference implementation, comparable chapter for chapter.

12 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & DJANGO + MYSQL SETUP

Premier League Predictor: Django & MySQL

Chapter 1 · Project Overview & Django + MySQL Setup

This is the second of four courses building the exact same Premier League prediction tracker in four genuinely different architectures — Premier League Predictor (FastAPI & PostgreSQL, complete — see `plpredict-fastapi1`), Premier League Predictor (FastAPI & Redis), and Premier League Predictor (Astro) are its siblings, the latter two still outlined rather than generated yet. Where the FastAPI variant leaned into a real, typed API layer, this course leans into Django's own ready-made admin interface — a genuine fit for a project with this much real, recurring manual data entry.

What the App Actually Does

The shared spec every course in this set builds toward — a personal, weekly Premier League prediction tracker:

- **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, with multiple guests averaged into a single figure for the week — and the BBC's own published AI-generated prediction, manually transcribed 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 didn't).
- **Maintain the real league table** — points, 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 auto-calculated and removed at season end; the three promoted teams entered manually.

THE REAL POINT VALUES, ALREADY CONFIRMED — NO NEED TO RE-DERIVE THEM HERE

`plpredict-fastapi1`'s own Chapter 6 resolved this directly with the user: **40 points for a correct score, 10 points for a correct result**. This course reuses that same confirmed split rather than asking again — a real product decision, made once, that applies to every variant in this set.

Why Django + MySQL for This One

This project has a genuinely heavy, recurring manual-entry burden — 10 fixtures every gameweek, up to 5 real predictions per fixture, a real result entered for every fixture, three promoted teams typed in by name every season. That's exactly the shape of work Django's own admin interface exists to make fast: generated automatically from the models this course defines, with zero hand-built forms needed for the raw data-entry side of the job (Chapter 3). MySQL is the natural, familiar real-world pairing for a Django project — a genuinely different relational engine from the FastAPI variant's own PostgreSQL, letting the two reference implementations be read side by side later.

ONE REAL DIFFERENCE TO KEEP IN MIND WELL BEFORE CHAPTER 5

The FastAPI/PostgreSQL variant's own Chapter 5 leaned on a genuine PostgreSQL-specific feature — a partial unique index (`CREATE UNIQUE INDEX ... WHERE source != 'guest'`) — to allow multiple guest predictions per fixture while still enforcing exactly one user/expert/AI prediction each. **MySQL has no equivalent mechanism** — it doesn't support a `WHERE` clause on an index at all. This course will need a genuinely different real solution to the identical problem when Chapter 5 gets there — flagged honestly now, not silently discovered later.

Setting Up: Django & MySQL

A local Python install with `pip`, and a working local MySQL server, are assumed from here on. Creating the database first, with an explicit modern charset:

```
CREATE DATABASE pl_predictor
CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

`utf8mb4`, not plain `utf8` — MySQL's own historical `utf8` charset is actually a restricted 3-byte encoding that can't store every real Unicode character (emoji included); `utf8mb4` is the

genuinely full implementation, and the standard real-world choice for a new MySQL database today.

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

# Install Django and a MySQL driver
pip install django mysqlclient
```

MYSQLCLIENT NEEDS REAL BUILD TOOLS — PYMYSQL IS THE EASIER DEVELOPMENT ALTERNATIVE

`mysqlclient` is a genuine C extension wrapping MySQL's own native client library — fast, but it needs real system build dependencies (`libmysqlclient-dev` on Debian/Ubuntu, Microsoft's C++ Build Tools on Windows) to install at all, unlike PostgreSQL's own `psycopg2-binary`, which ships pre-compiled. If that friction genuinely gets in the way during development, `pip install PyMySQL` plus two lines at the very top of `manage.py` — `import pymysql; pymysql.install_as_MySQLdb()` — makes Django use a pure-Python MySQL driver instead, with no compiled extension required at all. The real tradeoff: PyMySQL is genuinely slower under heavy load than `mysqlclient`, a real cost worth knowing about before choosing it for anything beyond local development.

Starting the Django Project

```
# Start the Django project and the predictor app
django-admin startproject pl_predictor_site .
python manage.py startapp predictor
```

Pointing the new project at the real MySQL database in `pl_predictor_site/settings.py`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'pl_predictor',
        'USER': 'root',
        'PASSWORD': '', # set a real password in production
        'HOST': 'localhost',
```

```

        'PORT': '3306',
        'OPTIONS': {
            'charset': 'utf8mb4',
        },
    }
}

```

Add `'predictor'` to `INSTALLED_APPS`, then confirm the connection works by running Django's own initial migrations against the real database:

```

python manage.py migrate
python manage.py runserver

```

THE OPTIONS CHARSET SETTING MATTERS AS MUCH AS THE DATABASE'S OWN

Setting `'charset': 'utf8mb4'` inside `OPTIONS` makes Django's own connection use the same full-Unicode charset the database was created with — a mismatch between the two (a `utf8mb4` database with a connection defaulting to plain `utf8`) is a real, documented source of silent data corruption for anything outside the restricted 3-byte range, not just a cosmetic setting.

Where This Course Is Headed

A real Django-model schema for teams, seasons, gameweeks, and fixtures (Chapter 2); the Django admin as the real manual-entry tool for teams and fixtures (Chapter 3); the fast click-to-pair fixture-entry UI built outside the admin (Chapter 4); recording all four prediction sources, and a genuinely MySQL-specific answer to the partial-unique-index gap flagged above (Chapter 5); entering results and calculating the confirmed 40/10 scoring (Chapter 6); the real league table (Chapter 7); the prediction league table (Chapter 8); promotion and relegation (Chapter 9); styling and the gameweek/season selector (Chapter 10); deployment (Chapter 11); and a capstone integrating this predictor into the existing Astro-based site (Chapter 12).

Hands-On Exercises

EXERCISE 1

Explain why this course reuses the 40/10 point values confirmed in plpredict-fastapi1's own Chapter 6 rather than asking the question fresh, and what that implies for any future variant in this same set.

 [View solution](#)

EXERCISE 2

Explain why utf8mb4 is the correct charset choice for the pl_predictor database instead of plain utf8, and why the OPTIONS charset setting in DATABASES needs to match it.

 [View solution](#)

EXERCISE 3

Set up the pl_predictor MySQL database and a working Django project connected to it yourself (using either mysqlclient or the PyMySQL alternative), confirm python manage.py migrate runs successfully, and write one sentence explaining what that command actually confirmed.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **The shared app** — a weekly Premier League prediction tracker: 4 real prediction sources per fixture, scored 40/10/0, across a real 38-gameweek season
- **Real point values (confirmed)** — 40 for a correct score, 10 for a correct result, reused directly from plpredict-fastapi1's own Chapter 6
- **Why this variant** — Django's own ready-made admin as a fast tool for this project's genuinely heavy manual-entry workload, paired with MySQL
- **Flagged early** — MySQL has no equivalent to PostgreSQL's partial unique index; Chapter 5 will need a genuinely different real fix for the same guest-prediction problem
- **Database** — MySQL, created as `pl_predictor` with explicit `utf8mb4` charset
- **Driver choice** — `mysqlclient` (fast, needs real build tools) or `PyMySQL` (pure Python, easier install, slower under load)

- **Next chapter:** Data Modeling — teams, seasons, gameweeks & fixtures as Django models

CHAPTER 2: DATA MODELING: TEAMS, SEASONS, GAMEWEEKS & FIXTURES AS DJANGO MODELS

Premier League Predictor: Django & MySQL

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

Everything else in this course — the admin, the fixture-entry UI, predictions, results, both league tables, promotion and relegation — sits on top of the schema this chapter builds, expressed as real Django models rather than SQLAlchemy's own class-based mapping. Five real tables: `Team`, `Season`, `SeasonTeam`, `Gameweek`, and `Fixture`. Predictions themselves are deliberately left out — that's a genuinely separate concern, covered 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, since a team's own identity and history are genuinely permanent facts, not something tied to a single season. `Season` represents one real Premier League season, with a real start date, end date, and an `is_current` flag.

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 model rather than a field 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, using the Django admin directly. Chapter 9's promotion/relegation logic, at the end of a season, is the same model 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

```
# predictor/models.py
from django.db import models

class Team(models.Model):
    name = models.CharField(max_length=100, unique=True)
    short_name = models.CharField(max_length=20)

    def __str__(self):
        return self.name

class Season(models.Model):
    name = models.CharField(max_length=20, unique=True) # e.g. "2026/27"
    start_date = models.DateField()
    end_date = models.DateField()
    is_current = models.BooleanField(default=False)

    def __str__(self):
        return self.name

class SeasonTeam(models.Model):
    season = models.ForeignKey(Season, on_delete=models.CASCADE, related_name='season_teams')
    team = models.ForeignKey(Team, on_delete=models.CASCADE)

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=['season', 'team'], name='uq_season_team')
        ]

class Gameweek(models.Model):
    season = models.ForeignKey(Season, on_delete=models.CASCADE, related_name='gameweeks')
    number = models.PositiveSmallIntegerField() # 1 through 38

    class Meta:
        constraints = [
            models.UniqueConstraint(fields=['season', 'number'], name='uq_season_gameweek')
```

```

]

class Fixture(models.Model):
    STATUS_SCHEDULED = 'scheduled'
    STATUS_PLAYED = 'played'
    STATUS_CHOICES = [(STATUS_SCHEDULED, 'Scheduled'), (STATUS_PLAYED, 'Played')]

    gameweek = models.ForeignKey(Gameweek, on_delete=models.CASCADE, related_name='fixtures')
    home_team = models.ForeignKey(Team, on_delete=models.CASCADE, related_name='home_fixtures')
    away_team = models.ForeignKey(Team, on_delete=models.CASCADE, related_name='away_fixtures')
    kickoff_time = models.DateTimeField(null=True, blank=True)
    home_score = models.PositiveSmallIntegerField(null=True, blank=True)
    away_score = models.PositiveSmallIntegerField(null=True, blank=True)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default=STATUS_SCHEDULED)

    class Meta:
        constraints = [
            models.CheckConstraint(
                check=~models.Q(home_team=models.F('away_team')),
                name='ck_fixture_teams_differ',
            )
        ]

```

RELATED_NAME IS REQUIRED HERE — DJANGO'S OWN VERSION OF THE FASTAPI COURSE'S FOREIGN_KEYS=[...] FIX

`Fixture` has **two** foreign keys pointing at the same model, `Team` — `home_team` and `away_team`. Without an explicit `related_name` on each, Django would try to auto-generate a reverse accessor for both using the same default name derived from `Fixture` — a real clash Django's own system checks catch and refuse to run with ("Reverse accessor... clashes with reverse accessor...").

`related_name='home_fixtures'` and `related_name='away_fixtures'` give each foreign key its own distinct reverse accessor (`team.home_fixtures.all()`, `team.away_fixtures.all()`) — the exact same underlying ambiguity this site's own FastAPI/SQLAlchemy sibling course resolved with `foreign_keys=[home_team_id]` / `foreign_keys=[away_team_id]`, hit here at the ORM's own ability to auto-name a reverse relationship rather than at query-building time.

A GENUINELY MYSQL-SPECIFIC GOTCHA THIS COURSE'S OWN POSTGRESQL SIBLING NEVER FACES

`CheckConstraint` is real, and Django will happily create it — but MySQL itself only started actually **enforcing** CHECK constraints as of version **8.0.16**, released in 2019. Before that release, MySQL's own documentation is explicit: the `CHECK` clause was parsed but silently ignored by every storage engine — a team could be recorded as playing itself with no error at all, on an older MySQL install, despite the constraint appearing to exist. Django's own system checks are aware of this and will warn directly ("MySQL does not support check constraints") if it detects an unsupported version; Django 4.2 and later require MySQL 8+ outright, closing most of this gap by requirement rather than leaving it to be discovered. Running a real, current MySQL 8.0.16+ install — matching this course's own Chapter 1 setup — means this constraint genuinely works as written.

Nullable Scores: A Fixture Exists Before It's Played

`home_score` and `away_score` are both `null=True` (allowing a real database `NULL`) and `blank=True` (allowing a Django form or the admin to leave the field empty), with `status` defaulting 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 is a real update against a fixture row that already exists, filling in the two score fields and flipping `status` to `"played"`.

NULL VS. BLANK — TWO GENUINELY SEPARATE DJANGO CONCERNS

`null=True` is a database-level setting: the column itself may store `NULL`. `blank=True` is a validation-level setting: a Django `ModelForm` (or the admin) will accept the field being left empty. Django's own documented convention is that both are usually set together for an optional field like this — a fixture with no result yet needs the database to actually allow the absence *and* any form filling it in to not demand a value that doesn't exist yet.

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

A team appearing *twice in the same gameweek* — once as home in one fixture, once as away in a different fixture — spans multiple rows, not one. Neither `CheckConstraint` nor `UniqueConstraint` can express "compare this row against every other row sharing the same gameweek." This schema leaves that rule to the application layer, same as its FastAPI sibling: 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.

Creating and Applying the Migration

```
python manage.py makemigrations predictor
python manage.py migrate
```

`makemigrations` compares the models above against Django's own migration history and writes a new migration file describing the five new tables and their constraints; `migrate` actually applies it against the real MySQL database from Chapter 1.

Where This Course Is Headed

Registering these models with the Django admin as the real, fast manual-entry tool for managing the 20 competing teams each season (Chapter 3); the fast click-to-pair fixture-entry UI built outside the admin, creating real `Fixture` rows against this chapter's own schema (Chapter 4); recording all four prediction sources per fixture, and the real MySQL-specific answer to Chapter 1's own flagged partial-unique-index gap (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 `Fixture`'s `home_team` and `away_team` fields both need an explicit `related_name`, and what real error Django's own system checks would raise if one were left out.

[!\[\]\(65e8f8322c024ac6fcf86b65a793ebdd_img.jpg\) View solution](#)

EXERCISE 2

Explain what would actually happen if a team were recorded as playing itself on a MySQL 8.0.15 database with this exact `CheckConstraint` in place, and why that differs from what would happen on MySQL 8.0.16 or later.

 [View solution](#)

EXERCISE 3

Run makemigrations and migrate against a real MySQL 8.0.16+ database yourself, then confirm in a MySQL client (e.g. `SHOW CREATE TABLE predictor_fixture;`) that the CHECK constraint genuinely appears in the generated table definition.

 [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 on it
- **Gameweek** — 1 through 38 per season, unique per (season, number)
- **Fixture** — `home_team/away_team` both reference `Team`, needing distinct `related_name` values to avoid a real Django reverse-accessor clash
- **MySQL-specific gotcha** — `CheckConstraint` only actually enforces on MySQL 8.0.16+; earlier versions silently parse and ignore it
- **Nullable `home_score/away_score`** — a fixture exists before it's played; Chapter 6 fills these in via a real update
- **Next chapter:** The Django admin as the real, fast manual-entry tool for teams & fixtures

CHAPTER 3: DJANGO ADMIN AS THE REAL MANUAL-ENTRY TOOL FOR TEAMS & FIXTURES

Premier League Predictor: Django & MySQL

Chapter 3 · Django Admin as the Real Manual-Entry Tool for Teams & Fixtures

Chapter 2 built the schema; this chapter turns `Team`, `Season`, `SeasonTeam`, `Gameweek`, and `Fixture` into real, working admin tooling — the actual real work happening for a project this admin-entry-heavy.

Team Admin: Search, Not Scroll

```
# predictor/admin.py
from django.contrib import admin
from .models import Team, Season, SeasonTeam, Gameweek, Fixture

@admin.register(Team)
class TeamAdmin(admin.ModelAdmin):
    list_display = ('name', 'short_name')
    search_fields = ('name', 'short_name')
```

`search_fields` does two real jobs at once: it powers the admin's own search box on `TeamAdmin`'s list page, and — as covered below — it's the exact thing that lets other admin screens offer a real, searchable autocomplete for picking an existing team instead of a plain dropdown.

SeasonTeam as an Inline: A Real Django-Specific Payoff

`plpredict-fastapi1`'s own Chapter 3 had to hand-write real `POST`/`DELETE` routes just to add or remove a team from a season. Django's admin gets the identical job largely for free, via a `TabularInline` nested directly on the `Season` edit page:

```
class SeasonTeamInline(admin.TabularInline):
    model = SeasonTeam
```

```

extra = 1
autocomplete_fields = ('team',)
max_num = 20
validate_max = True

@admin.register(Season)
class SeasonAdmin(admin.ModelAdmin):
    list_display = ('name', 'start_date', 'end_date', 'is_current')
    inlines = [SeasonTeamInline]

```

Opening a `Season` in the admin now shows every one of its current `SeasonTeam` rows as editable table rows right there on the page, with one blank row ready for the next addition — adding or removing a team is a normal save on the `Season` page itself, not a separate screen.

AUTOCOMPLETE_FIELDS DEPENDS DIRECTLY ON TEAMADMIN'S OWN SEARCH_FIELDS

`autocomplete_fields = ('team',)` turns the inline's plain foreign-key dropdown into a real, searchable, type-ahead widget — genuinely necessary once `Team` holds dozens of historical clubs, not just this season's 20. This isn't automatic in the abstract: Django's own admin requires the *target* model's admin (`TeamAdmin`) to define `search_fields` before `autocomplete_fields` can use it anywhere else — exactly why `TeamAdmin`'s own `search_fields`, defined above for its own list page, turns out to matter well beyond that one page.

VERIFIED: MAX_NUM ALONE NEVER ENFORCES THE CAP — ONLY VALIDATE_MAX=TRUE DOES

It's genuinely easy to assume `max_num = 20` stops a 21st `SeasonTeam` row from ever being saved. Checked directly against Django's own documented formset behavior: `max_num` by itself only limits how many *extra, blank* forms the inline displays for adding new rows — it does **not**, on its own, reject a submission that somehow includes more rows than that. Real enforcement on save requires the separate `validate_max = True` flag above, mirroring the identical `validate_max` option Django's own formset factories expose. Without it, this "cap" is cosmetic — the same underlying lesson `plpredict-fastapi1`'s own Chapter 3 delivered with an explicit `COUNT()` query before every insert, arrived at here from the opposite direction: a built-in-looking limit that turns out to need one more explicit setting before it's actually real.

Fixtures via the Admin: Real, Working, and Genuinely Slower

`Fixture` gets the same inline treatment, nested on `Gameweek` instead:

```
class FixtureInline(admin.TabularInline):
    model = Fixture
    extra = 0
    autocomplete_fields = ('home_team', 'away_team')
    fields = ('home_team', 'away_team', 'kickoff_time', 'home_score', 'away_score', 'st

@admin.register(Gameweek)
class GameweekAdmin(admin.ModelAdmin):
    list_display = ('season', 'number')
    list_filter = ('season',)
    inlines = [FixtureInline]
```

`Fixture`'s two foreign keys to `Team` don't need Chapter 2's own `related_name` disambiguation here — that clash only ever concerned the reverse accessor *on Team*, and this inline is nested under `Gameweek`, the model with exactly one foreign key pointing at it. Each row simply renders two separate autocomplete widgets, one per team.

A REAL, HONEST BASELINE — NOT THE FINAL ANSWER

This genuinely works: an admin can open a gameweek and add all 10 fixtures right there, each team searched by name via autocomplete. It's also genuinely slower than it needs to be — every fixture still means opening two separate search boxes and typing, ten times a week, every week. Chapter 4 builds a real, purpose-built click-to-pair interface on top of this same underlying data specifically to fix that — this chapter's own admin path stays available afterward as a real fallback, not something the next chapter replaces outright.

Where This Course Is Headed

The fast click-to-pair fixture-entry UI, built outside the admin, creating real `Fixture` rows against this chapter's own schema and reusing `Team` records exactly as they already exist (Chapter 4); recording predictions (Chapter 5); entering results (Chapter 6); both league tables (Chapters 7-8); and promotion/relegation, which reuses this chapter's own `SeasonTeamInline` directly at the season boundary (Chapter 9).

Hands-On Exercises

EXERCISE 1

Explain why `autocomplete_fields = ('team',)` on `SeasonTeamInline` requires `TeamAdmin` to define `search_fields`, and what would happen if `TeamAdmin`'s `search_fields` were removed.

 [View solution](#)

EXERCISE 2

Explain, precisely, what `max_num = 20` alone actually controls on `SeasonTeamInline`, and what `validate_max = True` adds that `max_num` by itself does not provide.

 [View solution](#)

EXERCISE 3

Register the admin classes from this chapter, create a superuser (python `manage.py createsuperuser`), open a real Season in the admin, add teams via the `SeasonTeamInline` until you reach the 20-team cap, and confirm attempting to add a 21st is genuinely rejected on save, not just visually limited.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **TeamAdmin** — `search_fields` powers both its own list-page search box and every `autocomplete_fields` reference to `Team` elsewhere in the admin
- **SeasonTeamInline** — a real `TabularInline` on `SeasonAdmin`, getting the "add/remove a team from a season" job largely for free compared to the FastAPI sibling's own hand-written routes
- **Verified gotcha** — `max_num = 20` alone only limits displayed blank rows; `validate_max = True` is what actually enforces the cap on save
- **FixtureInline** — a real, working way to enter a gameweek's fixtures through the admin, genuinely slower than what Chapter 4 builds next
- **No fk_name needed** — `FixtureInline` is nested under `Gameweek`, which has only one foreign key pointing at it; the `related_name` clash from Chapter 2 was specific to `Team`

- **Next chapter:** The fast click-to-pair fixture-entry UI, built outside the admin

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

Premier League Predictor: Django & MySQL

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

Chapter 3's own admin inline genuinely works, but means two searches per fixture, ten times a week. This chapter builds the real, faster alternative: 20 always-visible team buttons, clicked directly into place — the same underlying `Fixture` data, a purpose-built interface on top.

Rendering the Team Grid Server-Side, Not Fetched

`plpredict-fastapi1`'s own vanilla-JS frontend always fetched the season's teams from a JSON route the moment the page loaded. Django's own template engine can do that job directly, server-side, before the page is even sent to the browser:

```
# predictor/views.py
from django.shortcuts import render, get_object_or_404
from django.views.decorators.csrf import ensure_csrf_cookie

from .models import Gameweek, SeasonTeam

@ensure_csrf_cookie
def fixture_entry(request, gameweek_id):
    gameweek = get_object_or_404(Gameweek, pk=gameweek_id)
    season_teams = (
        SeasonTeam.objects
        .filter(season=gameweek.season)
        .select_related('team')
        .order_by('team__name')
    )
    teams = [st.team for st in season_teams]
    return render(request, 'predictor/fixture_entry.html', {
        'gameweek': gameweek,
        'teams': teams,
    })
```

predictor/templates/predictor/fixture_entry.html

```
<div id="team-grid" class="team-grid">
  {% for team in teams %}
    <button type="button" data-team-id="{{ team.id }}">{{ team.short_name }}</button>
  {% endfor %}
</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 type="button" onclick="clearSelection()">Clear</button>
<button type="button" id="add-fixture-btn" onclick="addFixture()" disabled>Add Fixture</button>
```

ONE REAL FETCH CALL THIS COURSE NEVER NEEDS

`{% for team in teams %}` writes every button directly into the HTML the server sends — by the time a browser paints this page, the 20 team buttons already exist, with no separate JavaScript request needed just to populate them. The FastAPI/vanilla-JS sibling's own `loadTeams()` function exists specifically because that course has no template engine standing between its data and its HTML; here, that same job is Django's own template rendering, done once, server-side, before the response is even sent.

The Real New Concern: CSRF

Adding a fixture is still a real state-changing request — this page's own JavaScript needs to `POST` to a Django view. That's the first genuinely new requirement this course introduces that its FastAPI sibling never had to think about at all:

FASTAPI SHIPS NO CSRF PROTECTION BY DEFAULT; DJANGO'S IS ON UNLESS DELIBERATELY REMOVED

`plpredict-fastapi1`'s own fixture-entry chapter never mentions CSRF, because FastAPI has no built-in cross-site request forgery protection at all — nothing stops a POST from succeeding regardless of where it originated. Django is the opposite default: `CsrfViewMiddleware`, present in every fresh Django project's own `MIDDLEWARE` setting, rejects any `POST` / `PUT` / `PATCH` / `DELETE` request that

doesn't include a valid CSRF token — including a JavaScript `fetch()` call, exactly like the one this chapter's own **Add Fixture** button needs to make.

`@ensure_csrf_cookie` on the view above guarantees the browser actually has a real CSRF token to send, even though this page renders no traditional HTML `<form>` for Django to attach one to automatically. The token then has to be read from the cookie and included on every fetch call by hand:

```
// static/predictor/fixtures.js
function getCookie(name) {
  const value = `; ${document.cookie}`;
  const parts = value.split(`; ${name}=`);
  if (parts.length === 2) return parts.pop().split(';').shift();
}

const csrftoken = getCookie('csrftoken');
```

Creating a Fixture: A Real JSON View, With the Same Server-Side Guard

```
# predictor/views.py (additions)
import json
from django.http import JsonResponse
from django.views.decorators.http import require_POST

from .models import Fixture

@require_POST
def create_fixture(request, gameweek_id):
    gameweek = get_object_or_404(Gameweek, pk=gameweek_id)
    payload = json.loads(request.body)
    home_team_id = payload.get('home_team_id')
    away_team_id = payload.get('away_team_id')

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

    # The real authority: check every fixture already entered this gameweek
```

```

existing = Fixture.objects.filter(gameweek=gameweek)
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 home_team_id in used_team_ids or away_team_id in used_team_ids:
    return JsonResponse(
        {'error': 'One of these teams is already fixtured this gameweek'}, status=400
    )

fixture = Fixture.objects.create(
    gameweek=gameweek, home_team_id=home_team_id, away_team_id=away_team_id
)
return JsonResponse({'id': fixture.id})

```

`@require_POST` is Django's own small, real decorator for rejecting any other HTTP method with a clean `405` — no need for a separate view per method the way a bare function view would otherwise need to branch on `request.method` by hand.

```

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

document.querySelectorAll('#team-grid button').forEach(btn => {
    btn.addEventListener('click', () => selectTeam(btn));
});

function selectTeam(btn) {
    const teamId = Number(btn.dataset.teamId);
    if (usedTeamIds.has(teamId)) return;
    if (!selectedHome) {
        selectedHome = { id: teamId, label: btn.textContent };
        document.getElementById('home-slot').textContent = btn.textContent;
    } else if (!selectedAway && teamId !== selectedHome.id) {
        selectedAway = { id: teamId, label: btn.textContent };
        document.getElementById('away-slot').textContent = btn.textContent;
    }
    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 gameweekId = document.body.dataset.gameweekId;
  const res = await fetch(`/gameweeks/${gameweekId}/fixtures/create/`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRFToken': csrftoken,
    },
    body: JSON.stringify({
      home_team_id: selectedHome.id,
      away_team_id: selectedAway.id,
    }),
  });

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

```

THE X-CSRFToken HEADER IS REQUIRED, NOT OPTIONAL — LEAVING IT OUT FAILS LOUDLY

Without `'X-CSRFToken': csrftoken` on the fetch call, Django's own `CsrfViewMiddleware` rejects the request outright with a real `403 Forbidden`, before `create_fixture`'s own code ever runs — a

genuinely different failure mode from the duplicate-team `409` above, and a real, common early mistake when adding AJAX to an otherwise server-rendered Django page for the first time.

THE SAME CLIENT-VS-SERVER AUTHORITY DISTINCTION AS THE FASTAPI SIBLING, STILL TRUE HERE

`usedTeamIds` starts empty on every page load, seeded only by fixtures actually added during the current visit — a stale reload could show a team as available that's genuinely already fixtured.

`create_fixture`'s own server-side check, querying every real `Fixture` row for the gameweek, is what actually prevents a bad duplicate; the client-side disabling above is convenience, exactly the same honest distinction `plpredict-fastapi1`'s own Chapter 4 already established.

Where This Course Is Headed

Recording all four prediction sources against each fixture this chapter creates, and the real MySQL-specific answer to the partial-unique-index gap flagged back in Chapter 1 (Chapter 5); entering results (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, plus styling, to replace this chapter's own single-gameweek page (Chapter 10).

Hands-On Exercises

EXERCISE 1

Explain why `fixture_entry` never needs a JavaScript fetch call just to display the initial 20 team buttons, tracing the reason back to what Django's template engine does that `plpredict-fastapi1`'s own vanilla-JS frontend has no equivalent for.

 [View solution](#)

EXERCISE 2

Explain why `@ensure_csrf_cookie` is needed on `fixture_entry` even though that view renders no traditional HTML form, and what would happen to the Add Fixture button's own fetch call if the `X-CSRFToken` header were left out entirely.

 [View solution](#)

EXERCISE 3

Build the full click-to-pair page yourself against a real gameweek, enter at least 3 fixtures, then deliberately remove the X-CSRFToken header from addFixture() and confirm the real, different error you get back compared to attempting to reuse an already-fixtured team.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **Team grid** — rendered server-side via {% for team in teams %}, no client-side fetch needed to populate it, unlike the FastAPI sibling
- **@ensure_csrf_cookie** — guarantees a CSRF token exists for a page with no traditional form to attach one to automatically
- **CSRF** — a genuinely new concern this course introduces; FastAPI ships none of this by default, Django's CsrfViewMiddleware rejects an unprotected POST outright
- **X-CSRFToken header** — required on every fetch() that changes state; missing it fails with a real 403, a different failure mode from the duplicate-team 409
- **@require_POST** — Django's own small decorator for a POST-only view, no manual request.method branching
- **Same real authority split as the FastAPI sibling** — server-side duplicate-team check is the real rule; client-side disabling is convenience
- **Next chapter:** Predictions — and a genuinely MySQL-specific answer to Chapter 1's own flagged gap

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

Premier League Predictor: Django & MySQL

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

Before a fixture's real result exists, four real sources each predict what they think will happen. This chapter builds the one model that records all four — and finally resolves the real gap Chapter 1 flagged: MySQL has no equivalent to PostgreSQL's own partial unique index, and this chapter needs exactly that shape of rule.

One Model, Four Sources

```
# predictor/models.py (additions)
from django.db.models import Case, When, Value, F
from django.db.models.functions import Concat

class Prediction(models.Model):
    SOURCE_USER = 'user'
    SOURCE_EXPERT = 'expert'
    SOURCE_GUEST = 'guest'
    SOURCE_AI = 'ai'
    SOURCE_CHOICES = [
        (SOURCE_USER, 'User'), (SOURCE_EXPERT, 'Expert'),
        (SOURCE_GUEST, 'Guest'), (SOURCE_AI, 'AI'),
    ]

    fixture = models.ForeignKey(Fixture, on_delete=models.CASCADE, related_name='predictions')
    source = models.CharField(max_length=10, choices=SOURCE_CHOICES)
    guest_name = models.CharField(max_length=100, null=True, blank=True)
    predicted_home_score = models.PositiveSmallIntegerField()
    predicted_away_score = models.PositiveSmallIntegerField()
    points_awarded = models.PositiveSmallIntegerField(null=True, blank=True)

    unique_key = models.GeneratedField(
        expression=Case(
            When(source=SOURCE_GUEST, then=Value(None)),
            default=Concat(F('fixture_id'), Value('-'), F('source')),
        ),
        if_primary_key=True,
    )
```

```

    ),
    output_field=models.CharField(max_length=30, null=True),
    db_persist=True,
)

class Meta:
    constraints = [
        models.UniqueConstraint(fields=['unique_key'], name='uq_prediction_single_s
    ]

```

MySQL's Own Real Workaround: A Generated Column Exploiting NULL's Own Uniqueness Rules

Three of the four sources genuinely predict exactly once per fixture; guest deliberately doesn't — some weeks have several. `plpredict-fastapi1`'s own PostgreSQL sibling solved this with a partial unique index, `WHERE source != 'guest'`. MySQL has no such feature — it doesn't support a conditional clause on an index at all. The real, standard MySQL workaround exploits a different, universal SQL rule instead: **a unique index never treats two `NULL` values as equal to each other**, so any number of `NULL`s can coexist in a column a unique index otherwise enforces strictly.

`unique_key` above is a real, database-computed column — Django's own `GeneratedField`, added in Django 5.0 — that evaluates, for every row, to either `NULL` (for a guest prediction) or a real, deterministic string like `"42-expert"` (fixture id + source, for everyone else). The `UniqueConstraint` is placed on that generated column, not on `fixture` / `source` directly. It compiles to genuine MySQL syntax:

```

unique_key VARCHAR(30) GENERATED ALWAYS AS (
    CASE WHEN source = 'guest' THEN NULL
         ELSE CONCAT(fixture_id, '-', source)
    END
) STORED UNIQUE;

```

THE IDENTICAL REAL GUARANTEE, A GENUINELY DIFFERENT MECHANISM

A guest row's `unique_key` is always `NULL` — MySQL never rejects a second, third, or tenth `NULL` in a unique-indexed column, so any number of guest predictions coexist freely on the same fixture, exactly as intended. A user/expert/AI row's `unique_key` is a real, non-null string unique per (fixture, source) — the unique index rejects a genuine second attempt outright, exactly like the PostgreSQL sibling's own partial index does. Two structurally different database features — a conditional index in one case, NULL-exempt uniqueness on a computed column in the other — arrive at the identical real rule.

`db_persist=True` matters specifically for MySQL: `GeneratedField` with `db_persist=True` creates a real `STORED` column (its value genuinely written to disk on every insert/update), which MySQL 5.7+ supports and can build a real index on — a virtual, non-stored generated column has real limitations around indexing that make `STORED` the safer, portable choice here.

Recording (and Correcting) a Prediction: One Real Built-In Call

`plpredict-fastapi1`'s own upsert route had to manually query for an existing row, branch, and either update or insert. Django's ORM already has this exact pattern built in:

```
# predictor/views.py (additions)
from .models import Prediction

@require_POST
def upsert_prediction(request, fixture_id):
    fixture = get_object_or_404(Fixture, pk=fixture_id)
    payload = json.loads(request.body)
    source = payload.get('source')
    guest_name = payload.get('guest_name')

    is_guest = source == Prediction.SOURCE_GUEST
    if is_guest and not guest_name:
        return JsonResponse({'error': 'guest_name is required for a guest prediction'},
                             status=400)
    guest_name = guest_name if is_guest else None

    lookup = {'fixture': fixture, 'source': source}
    if is_guest:
        lookup['guest_name'] = guest_name

    prediction, created = Prediction.objects.update_or_create(
```

```

        **lookup,
        defaults={
            'predicted_home_score': payload.get('predicted_home_score'),
            'predicted_away_score': payload.get('predicted_away_score'),
        },
    )
    return JsonResponse({'id': prediction.id, 'created': created})

def list_predictions(request, fixture_id):
    predictions = Prediction.objects.filter(fixture_id=fixture_id).values(
        'id', 'source', 'guest_name', 'predicted_home_score', 'predicted_away_score', '
    )
    return JsonResponse(list(predictions), safe=False)

```

A REAL ORM BUILT-IN, WHERE THE FASTAPI SIBLING HAD TO HAND-WRITE THE SAME LOGIC

`Prediction.objects.update_or_create(**lookup, defaults={...})` looks for an existing row matching `lookup`, updates it with `defaults` if found, or creates a brand-new row with both `lookup` and `defaults` combined if not — the exact find-then-update-or-insert logic `plpredict-fastapi11`'s own Chapter 5 wrote out explicitly as a `query.first()` check followed by an `if/else`. SQLAlchemy's Session API has no single built-in call for this; Django's `QuerySet` does, the same category of "gets this for free" payoff as Chapter 3's own `SeasonTeamInline`.

`.values('id', 'source', ...)` in `list_predictions` is a small, real convenience of its own — a queryset that returns plain dictionaries directly, ready for `JsonResponse(list(...), safe=False)`, with no separate serializer layer needed for a response this simple.

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

This chapter only records what each individual guest predicted — a real, separate `Prediction` row per guest, each with its own `guest_name`. Averaging multiple real scorelines into the single "guest" figure the prediction league table eventually scores is deliberately left for Chapter 6, exactly as in the FastAPI sibling course.

NO PREDICTION DEADLINE IS ENFORCED YET

Nothing here checks `fixture.kickoff_time` against the current time before accepting an upsert — the same honest, deliberately unaddressed gap the FastAPI sibling course also carries.

Recording Predictions From the Frontend

A compact, reusable form, sending the CSRF token established in Chapter 4 on every submission:

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

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

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

Where This Course Is Headed

Entering real results, defining how a correct score and a correct result actually get calculated, and resolving how multiple guest predictions become one comparable figure (Chapter 6); the real league table (Chapter 7); the prediction league table (Chapter 8); and promotion/relegation (Chapter 9).

Hands-On Exercises

EXERCISE 1

Explain why `unique_key` evaluates to `NULL` for a guest prediction but a real string for every other source, and why that specific choice is what makes the `UniqueConstraint` on `unique_key` behave correctly for both cases.

[!\[\]\(7e46b98862b032bac4dfd70e25da77c3_img.jpg\) View solution](#)

EXERCISE 2

Explain why `db_persist=True` is specifically required for this `GeneratedField` to work correctly on MySQL, and what real database feature it corresponds to.

[!\[\]\(51fa12e9938db9b91c0132320af2b84a_img.jpg\) View solution](#)

EXERCISE 3

Record a user, an expert, an AI, and two separately-named guest predictions against a single real fixture using `upsert_prediction`, then submit a second user prediction for the same fixture and confirm it updates the existing row (`created: false`) rather than creating a duplicate.

[!\[\]\(a30b5314efefb0416f322d92d9011828_img.jpg\) View solution](#)

CHAPTER 5 QUICK REFERENCE

- **Prediction** — one model, four sources (user/expert/guest/ai), keyed by fixture + source (+ `guest_name` for guests)
- **unique_key** — a real `GeneratedField` (Django 5.0+): `NULL` for guest rows, a deterministic fixture-source string otherwise
- **Real MySQL workaround** — `NULL` is never unique-constrained against another `NULL`, exploited here to get the identical guarantee PostgreSQL's own partial index provides
- **db_persist=True** — creates a real `STORED` generated column, required for MySQL 5.7+ to index it at all

- **update_or_create()** — Django's own real built-in upsert, replacing the FastAPI sibling's own hand-written find-then-branch logic
- **Still deferred to Chapter 6** — how multiple guest scorelines become one comparable figure
- **Next chapter:** Entering results and calculating correct score vs. correct result

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

Premier League Predictor: Django & MySQL

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 with `points_awarded` null too — a field that chapter already added to the model in anticipation of this one. This chapter fills in the real result and, at the same moment, scores everything 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. The logic itself has nothing to do with Django or MySQL at all; it's plain Python, sitting in its own module:

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

def _outcome(home, away):
    if home > away:
        return 'home_win'
    elif away > home:
        return 'away_win'
    return 'draw'

def score_prediction(predicted_home, predicted_away, actual_home, actual_away):
    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: every correct score is automatically a correct result too, since the outcome is derived straight from the scoreline — but a correct result doesn't imply a correct score in the other direction. Checking the stronger condition first and returning immediately is what stops a correctly predicted 2-1 from being scored as a mere 10-point correct result instead of the full 40.

Entering a Result: One POST, Then a Real Batched Rescore

A small POST view, matching the `@require_POST` convention already established for every mutation in this course since Chapter 4:

```
# predictor/views.py (additions)
from .scoring import score_prediction

@require_POST
def enter_result(request, fixture_id):
    fixture = get_object_or_404(Fixture, pk=fixture_id)
    payload = json.loads(request.body)
    home_score = payload.get('home_score')
    away_score = payload.get('away_score')

    if home_score is None or away_score is None:
        return JsonResponse(
            {'error': 'home_score and away_score are both required'}, status=400
        )

    fixture.home_score = home_score
    fixture.away_score = away_score
    fixture.status = Fixture.STATUS_PLAYED
    fixture.save()

    predictions = list(Prediction.objects.filter(fixture=fixture))
    for prediction in predictions:
        prediction.points_awarded = score_prediction(
            prediction.predicted_home_score,
            prediction.predicted_away_score,
            home_score,
            away_score,
```

```

    )
    Prediction.objects.bulk_update(predictions, ['points_awarded'])

    return JsonResponse({
        'id': fixture.id,
        'home_score': fixture.home_score,
        'away_score': fixture.away_score,
        'status': fixture.status,
    })

```

A REAL DJANGO ORM BUILT-IN FOR UPDATING MANY ROWS IN ONE PASS

A naive version of this loop would call `prediction.save()` inside the `for` loop itself — one individual `UPDATE` statement per prediction, N round trips to MySQL for N predictions.

`Prediction.objects.bulk_update(predictions, ['points_awarded'])` instead collapses that into far fewer queries: internally, Django batches the objects (100 per batch by default) and compiles each batch into a single real SQL statement built around a `CASE WHEN id = ... THEN ... END` expression, updating every row in the batch in one round trip rather than one per row — the same category of "gets this for free from the ORM" payoff `update_or_create()` already delivered in Chapter 5, applied here to bulk rescoreing instead of a single upsert.

BULK_UPDATE() SKIPS SAVE() ENTIRELY — NO SIGNALS FIRE

Django's own documentation is explicit about this: `bulk_update()` never calls each object's `save()` method, and it sends no `pre_save` / `post_save` signals. That's genuinely fine here — nothing in this app currently listens for those signals on `Prediction` — but it's a real, honest caveat worth knowing before reaching for `bulk_update()` in a codebase that *does* hang logic off save signals, since that logic would silently never run for a bulk-updated row.

RESCORING ON CORRECTION IS A FEATURE, NOT A SIDE EFFECT

`enter_result` re-fetches and recomputes `points_awarded` for *every* prediction on the fixture every single time it's called — not just the first time. Fixing a mistyped scoreline genuinely re-scores every user, expert, guest, and AI prediction correctly against the corrected result, rather than leaving stale points sitting in the database from the wrong one. A fixture with zero predictions recorded simply loops over an empty list and calls `bulk_update()` with nothing to update — no special case needed.

The points_awarded Field Was Already Waiting

Chapter 5 added `points_awarded` to the `Prediction` model specifically for this chapter — `models.PositiveSmallIntegerField(null=True, blank=True)`, null until a real result exists. Chapter 5's own `list_predictions` view already selects it too, in its `.values('id', 'source', 'guest_name', 'predicted_home_score', 'predicted_away_score', 'points_awarded')` call — nothing about that view needs to change now that the field actually has values in it.

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 — 3-0 and 1-0 average to 2-0, which happens to look plausible by coincidence, but 3-0 and 0-2 average to "1.5-1," which isn't a result anything could actually finish.

POINTS ARE JUST NUMBERS — THEY ALWAYS AVERAGE CLEANLY

Every guest's own `Prediction` row now gets a 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. **Averaging those three numbers 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, deliberately, so a fixture with three guests doesn't count three times as much toward the season total as a fixture with only one.

A Worked Example: Five Predictions Against a Real 2-1 Result

Say fixture 42 already has five real predictions recorded from Chapter 5 — user, expert, two separately-named guests, and AI. `enter_result` is called with `{"home_score": 2, "away_score": 1}`, and `GET /fixtures/42/predictions/` comes back:

SOURCE	PREDICTED	OUTCOME	POINTS_AWARDED
user	2-1	exact match	40
expert	1-1	draw vs. real home win	0

SOURCE	PREDICTED	OUTCOME	POINTS_AWARDED
guest — Micah Richards	3-0	home win, right outcome	10
guest — Jamie Carragher	1-0	home win, right outcome	10
ai	2-0	home win, right outcome	10

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, so each earns 10. The guest average for this fixture is therefore $(10 + 10) / 2 = \mathbf{10}$, the figure Chapter 8 would fold into the guest predictor's own season total for this one fixture.

Recording a Result From the Frontend

A small form, reusing the same `csrftoken` established in Chapter 4:

```
// static/predictor/results.js
async function submitResult(fixtureId, homeScore, awayScore) {
  const res = await fetch(`/fixtures/${fixtureId}/result/`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRFToken': csrftoken,
    },
    body: JSON.stringify({
      home_score: Number(homeScore),
      away_score: Number(awayScore),
    }),
  });

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

```
alert('Result recorded – every prediction on this fixture has been rescored.');
```

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, operating on `SeasonTeam` from Chapter 2 (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.

[!\[\]\(9c300fffd88bdb3763537ae0c20e64d3_img.jpg\) 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.

[!\[\]\(3ada81272cf52cdd8ab6f0935cd6f39e_img.jpg\) View solution](#)

EXERCISE 3

Enter an incorrect result for a real fixture with at least two predictions already recorded via `enter_result`, confirm their `points_awarded` values, then call `enter_result` again with the corrected result and confirm every prediction's `points_awarded` genuinely changed via `bulk_update()` rather than only being scored the first time.

 [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
- **enter_result** — a POST view filling in the real scoreline, flipping status to "played," and rescoreing every prediction on the fixture from scratch
- **bulk_update()** — a real Django ORM built-in batching many row updates into a single CASE WHEN statement, instead of one save() per row
- **Honest caveat** — bulk_update() skips save() and fires no pre_save/post_save signals
- **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: Django & MySQL

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

Every fixture Chapter 6 fills in a result for is a real, scored match. This chapter turns that pile of results into the table every Premier League fan actually recognizes — 20 rows, sorted by points, then goal difference, then goals scored. It's a genuinely harder query than it looks, and the reason why is worth walking through in full rather than skipping to the answer.

Starting From SeasonTeam, Not From Fixture

A team with zero games played this season still needs to appear in the table — with 0 played, 0 points, a blank row waiting to be filled in. Build the table by querying `Fixture` directly and no such team ever shows up at all, since a team with no fixtures produces no rows to aggregate. Chapter 2's own `SeasonTeam` model — the real record of which 20 teams are competing this season — is the correct starting point instead, exactly the same membership reasoning Chapter 3's own admin tooling and Chapter 9's own promotion/relegation logic both already lean on.

A Tempting, Genuinely Wrong Django ORM Attempt

`Fixture` has two separate foreign keys to `Team` — `home_team` and `away_team` — so it's natural to reach for both reverse relations, `home_fixtures` and `away_fixtures`, in one `annotate()` call:

```
# DO NOT USE – this looks reasonable and compiles fine, but is genuinely wrong
from django.db.models import Sum, Case, When, IntegerField, F

teams = Team.objects.filter(seasonteam__season_id=season_id).annotate(
    home_goals=Sum(
        Case(
            When(
                home_fixtures__gameweek__season_id=season_id,
                home_fixtures__status='played',
                then=F('home_fixtures__home_score'),
```

```

    ),
    default=0, output_field=IntegerField(),
)
),
away_goals=Sum(
    Case(
        When(
            away_fixtures__gameweek__season_id=season_id,
            away_fixtures__status='played',
            then=F('away_fixtures__away_score'),
        ),
        default=0, output_field=IntegerField(),
    )
),
)

```

TWO SEPARATE REVERSE RELATIONS, JOINED IN THE SAME QUERY, SILENTLY MULTIPLY THE RESULTS

Django's own documentation warns about exactly this shape of query, under "Combining multiple aggregations": joining *two different* reverse foreign key relations in a single `annotate()` call produces one combined SQL `JOIN` across both relations, and each combination of a home fixture row and an away fixture row for the same team becomes its own row in that join — a real cross-join, not two independent sums. Say a team played 4 home fixtures (scoring 8 goals total at home) and 4 away fixtures (scoring 6 goals total away) — the join produces $4 \times 4 = 16$ combined rows.

`Sum('home_fixtures__home_score')` now adds up every home-goals value once *for each* of the 4 away-fixture rows it got paired with, inflating the true total of 8 up to $8 \times 4 = 32$. The away sum inflates the same way in the other direction, to $6 \times 4 = 24$. The team's real goals-for total is $8 + 6 = 14$ — this query would report $32 + 24 = 56$, a number that looks plausible on a screen but is four times too high, with no error raised anywhere to flag it.

Nothing about this bug is specific to football or to this schema — it's a well-documented, general Django ORM gotcha that shows up any time an `annotate()` call aggregates over more than one distinct reverse relation at once.

The Real Fix: Normalize Before Aggregating, With a Raw Query

The actual fix is the same one `plpredict-fastapi1`'s own PostgreSQL sibling reaches for: don't aggregate two separate relations in one pass at all — normalize home rows and away rows into

one shared shape first (a real SQL `UNION ALL`), *then* aggregate that single, already-flat result set. Django's ORM has no way to express a `UNION ALL` inside a subquery cleanly, so this is exactly the kind of genuinely complex aggregate worth dropping into a real raw query for — Django's own equivalent of SQLAlchemy's `text()` escape hatch is the model manager's `.raw()` method:

```
# predictor/views.py (additions)
LEAGUE_TABLE_SQL = """
    SELECT
        t.id,
        t.name AS team_name,
        COALESCE(agg.played, 0) AS played,
        COALESCE(agg.won, 0) AS won,
        COALESCE(agg.drawn, 0) AS drawn,
        COALESCE(agg.lost, 0) AS lost,
        COALESCE(agg.goals_for, 0) AS goals_for,
        COALESCE(agg.goals_against, 0) AS goals_against,
        COALESCE(agg.goals_for, 0) - COALESCE(agg.goals_against, 0) AS goal_difference,
        COALESCE(agg.points, 0) AS points
    FROM predictor_seasonteam st
    JOIN predictor_team t ON t.id = st.team_id
    LEFT JOIN (
        SELECT
            team_id,
            COUNT(*) AS played,
            SUM(CASE WHEN goals_for > goals_against THEN 1 ELSE 0 END) AS won,
            SUM(CASE WHEN goals_for = goals_against THEN 1 ELSE 0 END) AS drawn,
            SUM(CASE WHEN goals_for < goals_against THEN 1 ELSE 0 END) AS lost,
            SUM(goals_for) AS goals_for,
            SUM(goals_against) AS goals_against,
            SUM(
                CASE WHEN goals_for > goals_against THEN 3
                     WHEN goals_for = goals_against THEN 1
                     ELSE 0 END
            ) AS points
        FROM (
            SELECT f.home_team_id AS team_id, f.home_score AS goals_for, f.away_score AS goals_against
            FROM predictor_fixture f
            JOIN predictor_gameweek gw ON gw.id = f.gameweek_id
            WHERE gw.season_id = %s AND f.status = 'played'
        )
    ) AS agg
    UNION ALL
```

```

        SELECT f.away_team_id AS team_id, f.away_score AS goals_for, f.home_score AS goals_against
        FROM predictor_fixture f
        JOIN predictor_gameweek gw ON gw.id = f.gameweek_id
        WHERE gw.season_id = %s AND f.status = 'played'
    ) AS team_fixtures
    GROUP BY team_id
) AS agg ON agg.team_id = t.id
WHERE st.season_id = %s
ORDER BY points DESC, goal_difference DESC, goals_for DESC
"""

def league_table(request, season_id):
    rows = Team.objects.raw(LEAGUE_TABLE_SQL, [season_id, season_id, season_id])
    results = [
        {
            'team_id': row.id,
            'team_name': row.team_name,
            'played': row.played,
            'won': row.won,
            'drawn': row.drawn,
            'lost': row.lost,
            'goals_for': row.goals_for,
            'goals_against': row.goals_against,
            'goal_difference': row.goal_difference,
            'points': row.points,
        }
        for row in rows
    ]
    return JsonResponse(results, safe=False)

```

DJANGO'S RAW() ATTACHES EVERY EXTRA COLUMN AS A REAL ATTRIBUTE, NOT JUST THE MODEL'S OWN FIELDS

`team_name`, `played`, `won`, `goal_difference`, and every other column above aren't real fields on the `Team` model — only `id` and `name` are. Django's own documentation is explicit that `.raw()` only strictly requires the model's primary key column to appear in the result set (here, `t.id`); every other selected column, whether or not it matches a real model field, is attached to each returned instance as an ordinary Python attribute anyway. That's what makes `row.team_name`, `row.played`, and `row.points` all work above despite none of them being declared anywhere on `Team` itself — a

genuinely different mechanism from SQLAlchemy's own `text()`, which needs an explicit `.mappings()` call to get dict-like row access rather than model-instance access.

THE ORDER BY HAS TO LIVE IN THE RAW SQL ITSELF

A `RawQuerySet` — what `.raw()` returns — has no `.order_by()` of its own the way an ordinary Django `QuerySet` does. If a raw query doesn't specify its own ordering, Django's documentation is upfront that the rows may come back in no particular order at all. That's exactly why `ORDER BY points DESC, goal_difference DESC, goals_for DESC` is written directly into `LEAGUE_TABLE_SQL` above, rather than left for Django to add afterward.

NO HEAD-TO-HEAD OR FAIR-PLAY TIEBREAK IS IMPLEMENTED

Two teams finishing genuinely level on points, goal difference, *and* goals for are left in whatever order MySQL happens to return them — the same honest scope boundary the FastAPI/PostgreSQL sibling course carries. A real head-to-head record or disciplinary-points tiebreak would need its own dedicated query on top of this one.

Where This Course Is Headed

The prediction league table — aggregating every source's own `points_awarded` across the season, including the guest-averaging-by-fixture Chapter 6 resolved (Chapter 8); promotion and relegation, calculated directly from this chapter's own bottom-three ordering (Chapter 9); and styling plus a real gameweek/season selector (Chapter 10).

Hands-On Exercises

EXERCISE 1

Using this chapter's own worked example (a team with 4 played home fixtures totaling 8 goals, and 4 played away fixtures totaling 6 goals), explain step by step exactly how the naive dual-annotate() query arrives at 56 goals for instead of the real 14, tracing through the join multiplication that causes it.

 [View solution](#)

EXERCISE 2

Explain why normalizing home and away fixture rows into one shared shape with UNION ALL before grouping avoids the join-multiplication problem entirely, rather than just producing a smaller version of the same bug.

 [View solution](#)

EXERCISE 3

Call league_table for a real season containing at least one team with zero played fixtures so far, confirm that team still appears in the results with played=0 and points=0, then record a real result for one of that team's fixtures via enter_result and confirm the row updates correctly on the next call.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- **Start from SeasonTeam** — not Fixture, so a team with zero games played still shows an honest zero-played row
- **Real Django gotcha** — annotating Sum() over two different reverse relations (home_fixtures/away_fixtures) in one call cross-joins them, silently multiplying (not just adding) the wrong totals
- **The fix** — normalize home/away rows into one shape with a real UNION ALL subquery, then aggregate once
- **Team.objects.raw()** — Django's own equivalent of SQLAlchemy's text() escape hatch; needs only the model's pk column present, attaches every other selected column as a real instance attribute
- **RawQuerySet has no order_by()** — the ORDER BY has to be written directly into the raw SQL
- **Honest scope note** — no head-to-head or fair-play tiebreaker implemented
- **Next chapter:** The prediction league table — aggregating predictor performance across the season

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

Premier League Predictor: Django & MySQL

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 already scored.

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

```
WITH season_predictions AS (
    SELECT p.source, p.fixture_id, p.points_awarded
    FROM predictor_prediction p
    JOIN predictor_fixture f ON f.id = p.fixture_id
    JOIN predictor_gameweek gw ON gw.id = f.gameweek_id
    WHERE gw.season_id = %s AND p.points_awarded IS NOT NULL
)
```

`p.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. MySQL 8.0's own `WITH` clause support — a real requirement Chapter 2 already anchored this course to (Django 4.2+ requires MySQL 8 outright) — is what makes writing this as a genuine named CTE possible at all, rather than nesting it as an anonymous subquery.

Three Straightforward Sources, One That Needs Two Passes

User, expert, and AI each contribute exactly one `Prediction` row per fixture — Chapter 5's own `unique_key` generated column 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.

```
# predictor/views.py (additions)
from django.db import connection

PREDICTION_LEAGUE_SQL = """
    WITH season_predictions AS (
        SELECT p.source, p.fixture_id, p.points_awarded
        FROM predictor_prediction p
        JOIN predictor_fixture f ON f.id = p.fixture_id
        JOIN predictor_gameweek gw ON gw.id = f.gameweek_id
        WHERE gw.season_id = %s AND p.points_awarded IS NOT NULL
    ),
    non_guest_totals AS (
        SELECT
            source,
            COUNT(*) AS predictions_scored,
            SUM(points_awarded) 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) AS total_points,
            NULL AS correct_scores,
            NULL AS correct_results
        FROM guest_per_fixture
    )

```

```

SELECT * FROM non_guest_totals
UNION ALL
SELECT * FROM guest_totals
ORDER BY total_points DESC
"""

def dictfetchall(cursor):
    # Django's own documented pattern for turning a raw cursor into plain dicts
    columns = [col[0] for col in cursor.description]
    return [dict(zip(columns, row)) for row in cursor.fetchall()]

def prediction_table(request, season_id):
    with connection.cursor() as cursor:
        cursor.execute(PREDICTION_LEAGUE_SQL, [season_id])
        rows = dictfetchall(cursor)
    return JsonResponse(rows, safe=False)

```

CONNECTION.CURSOR(), NOT MODEL.OBJECTS.RAW() — BECAUSE THESE ROWS AREN'T ROWS OF ANY MODEL

Chapter 7's own raw query used `Team.objects.raw()` because every row it returned genuinely was a real `Team` instance, with a real primary key column present in the result. This query's rows have no such anchor — `source` is a plain string ('user', 'expert', 'guest', 'ai'), not a foreign key to anything, and there's no model anywhere in this app whose primary key a "prediction league" row could map onto. `.raw()` specifically requires a model class and its pk column; without one, Django's own lower-level tool is `connection.cursor()` — a genuine, direct DB-API cursor, with `dictfetchall()` (copied almost verbatim from Django's own documented example) turning its plain tuple rows into ordinary dicts using `cursor.description` for the column names.

A REAL MYSQL-VS-POSTGRESQL DIFFERENCE — NO EXPLICIT CAST NEEDED HERE

`plpredict-fastapi1`'s own PostgreSQL sibling needs an explicitly-typed `NULL::int` for its guest row's `correct_scores` / `correct_results` columns, since PostgreSQL requires a `UNION ALL`'s corresponding columns to resolve to a matching type and won't always infer one from a bare, untyped `NULL` on its own. MySQL is more lenient here: a plain `NULL AS correct_scores` in `guest_totals` above works with no cast at all — MySQL resolves each result column's real type from whichever `UNION` branch actually supplies a concrete value for it. The underlying reason for

including the column at all is identical between both databases (explained below); only the SQL syntax needed to express "this column genuinely doesn't apply here" differs.

"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/predictor/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(`/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>${breakdown}</td>
    <td>${Number(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. `Number(row.total_points)` is a small, real MySQL-driver detail: MySQL's own `DECIMAL` aggregate results (what `SUM()` / `AVG()` over integer columns actually return) frequently arrive at the Python driver as decimal strings rather than native floats, so `JsonResponse` serializes them as JSON strings — wrapping with `Number()` on the frontend is what makes `.toFixed(1)` safe to call regardless of which shape came back.

Where This Course Is Headed

Promotion and relegation between seasons — real changes to `SeasonTeam`, 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` for the `guest` row instead of being computed the same way as the other three sources, and why MySQL doesn't need an explicit type cast on that `NULL` the way the PostgreSQL sibling course does.

 [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 `prediction_table` for that season 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 `unique_key`
- **Guest** — averaged per fixture first (`guest_per_fixture`), then summed across the season — resolving Chapter 6's own deferred question
- **`connection.cursor() + dictfetchall()`** — used instead of `.raw()` since these rows aren't real model instances with a primary key
- **MySQL vs. PostgreSQL** — a bare `NULL` in the `UNION ALL` works fine here; no `::int`-style cast needed, unlike the sibling course

- **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 SeasonTeam, the table both league tables read from

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

Premier League Predictor: Django & MySQL

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 — and it went further, promising that `SeasonTeamInline` would get reused here rather than rebuilt. What's genuinely new in this chapter 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 Real Source of Truth

Rather than duplicating the league-table SQL, Chapter 7's own `league_table` view gets a small, honest refactor — its query logic pulled into a plain function both it and this chapter's new views can call. `LEAGUE_TABLE_SQL` itself is completely unchanged from Chapter 7:

```
# predictor/views.py (refactored)
def compute_league_table(season_id):
    rows = Team.objects.raw(LEAGUE_TABLE_SQL, [season_id, season_id, season_id])
    return [
        {
            'team_id': row.id,
            'team_name': row.team_name,
            'played': row.played,
            'won': row.won,
            'drawn': row.drawn,
            'lost': row.lost,
            'goals_for': row.goals_for,
            'goals_against': row.goals_against,
            'goal_difference': row.goal_difference,
            'points': row.points,
        }
        for row in rows
    ]
```

```
def league_table(request, season_id):
    get_object_or_404(Season, pk=season_id)
    return JsonResponse(compute_league_table(season_id), safe=False)
```

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

A Read-Only Preview Before Anything Is Written

```
# predictor/views.py (additions)
REQUIRED_TEAMS_PER_SEASON = 20
RELEGATION_COUNT = 3

def relegation_preview(request, season_id):
    get_object_or_404(Season, pk=season_id)
    table = compute_league_table(season_id)
    if len(table) != REQUIRED_TEAMS_PER_SEASON:
        return JsonResponse(
            {'error': f'Season has {len(table)} teams, expected {REQUIRED_TEAMS_PER_SEASON}',
             status=400,
            })

    bottom_three_ids = [row['team_id'] for row in table[-RELEGATION_COUNT:]]
    teams = Team.objects.filter(id__in=bottom_three_ids)
    return JsonResponse(
        [{'id': t.id, 'name': t.name, 'short_name': t.short_name} for t in teams],
        safe=False,
    )
```

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

Carrying the Survivors Forward, For Real

```
# predictor/views.py (additions)
from django.db import transaction, IntegrityError

@require_POST
def rollover_survivors(request, new_season_id, old_season_id):
    new_season = get_object_or_404(Season, pk=new_season_id)
    old_season = get_object_or_404(Season, pk=old_season_id)

    if SeasonTeam.objects.filter(season=new_season).exists():
        return JsonResponse(
            {'error': 'New season already has teams; rollover expects a freshly created',
             status=400},
        )

    old_table = compute_league_table(old_season_id)
    if len(old_table) != REQUIRED_TEAMS_PER_SEASON:
        return JsonResponse(
            {'error': f'Old season has {len(old_table)} teams, expected {REQUIRED_TEAMS_PER_SEASON}',
             status=400},
        )

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

    try:
        with transaction.atomic():
            SeasonTeam.objects.bulk_create([
                SeasonTeam(season=new_season, team_id=row['team_id'])
                for row in survivors
            ])
    except IntegrityError:
        return JsonResponse(
            {'error': 'This rollover could not be completed cleanly – check whether it is a duplicate',
             status=409},
        )

    return JsonResponse({
        'new_season_id': new_season.id,
```

```
'survivors_carried_forward': len(survivors),
'next_step': "Add the 3 promoted teams via the Season admin page's own SeasonTe
})
```

BULK_CREATE() — THE WRITE-SIDE COUNTERPART TO CHAPTER 6'S OWN BULK_UPDATE()

Looping 17 individual `SeasonTeam.objects.create(...)` calls would issue 17 separate `INSERT` statements. `bulk_create()` collapses that into far fewer real `INSERT` statements — batched, the same way `bulk_update()` batched Chapter 6's own rescoring — and it still fully respects Chapter 2's own `UniqueConstraint("season_id", "team_id")` at the database level, since the constraint is enforced by MySQL itself on the real `INSERT`, regardless of which Django-level method produced it.

"REMOVING" A RELEGATED TEAM DOESN'T DELETE ANYTHING

Nothing about `rollover_survivors` 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 view is accidentally called twice for the same `new_season_id`, the second call would try to insert 17 `SeasonTeam` rows that already exist, colliding with Chapter 2's own `UniqueConstraint`. The `try/except IntegrityError` here is the exact same discipline Chapter 4 established for a single duplicate gameweek, just wrapping an operation that touches 17 rows instead of one — `transaction.atomic()` is what guarantees the whole batch either commits together or none of it does, so a mid-batch failure never leaves the new season half-populated.

Adding the Promoted Teams: Chapter 3's Admin Tool, Reused Rather Than Rebuilt

`plpredict-fastapi1`'s own PostgreSQL sibling builds one combined route that both carries the survivors forward *and* accepts the 3 promoted teams' own names in the same request body. This course deliberately splits the two halves instead — `rollover_survivors` above handles

only the mechanical part (auto-calculated, no human judgment involved), and the 3 promoted teams — a genuinely human decision, not something derivable from any query — are added directly through the Season admin page's own `SeasonTeamInline`, exactly the tool Chapter 3 built and explicitly promised this chapter would reuse.

THE "+" POPUP COVERS THE CASE WHERE A PROMOTED TEAM DOESN'T EXIST YET

`SeasonTeamInline`'s own `team` field uses `autocomplete_fields` (Chapter 3), which still shows Django admin's real "add another" "+" icon next to the search widget — clicking it opens a small popup to create a brand-new `Team` without ever leaving the Season admin page, provided the logged-in user has `add_team` permission. A promoted team newly arrived from the Championship, with no existing `Team` row at all, is handled by this popup directly; a promoted team that's been in the Premier League before (and already has a `Team` row from a previous season) is handled by the ordinary autocomplete search instead — both cases go through the exact same widget.

In practice: a site admin runs `rollover_survivors` (or clicks a button that calls it), then opens the new `Season` in the admin and adds exactly 3 rows to `SeasonTeamInline` — the same `max_num = 20` / `validate_max = True` cap from Chapter 3 means the inline itself refuses to let the season exceed 20 teams once the 17 survivors are already there, a small, real safety net that a hand-rolled promoted-teams route would have had to reimplement by hand.

NO UNDO, AND NO PARTIAL-ROLLOVER RECOVERY

If `rollover_survivors` is run against the wrong old season, there's no dedicated "undo" route — fixing it means manually removing the incorrect `SeasonTeam` rows through the admin (or the shell) and running it again correctly. `transaction.atomic()` guarantees the 17-row batch itself is all-or-nothing; it says nothing about a mistake made one level up, in which two seasons were confused for each other.

Using It From the Frontend

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

```

async function rolloverSurvivors(newSeasonId, oldSeasonId) {
  const res = await fetch(`/seasons/${newSeasonId}/rollover-survivors-from/${oldSeasonId}`
    method: 'POST',
    headers: { 'X-CSRFToken': csrftoken },
  });

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

  const result = await res.json();
  alert(`${result.survivors_carried_forward} survivors carried forward. ${result.next_s`
}

```

Where This Course Is Headed

A real gameweek/season selector — replacing every hardcoded season/gameweek id this course has used since Chapter 4 with a genuine dropdown backed by `GET /seasons/`, tying together the built-in Django admin interface with the existing form (Chapter 10), the Django REST framework (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]` directly from `compute_league_table`'s own result, 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 why this chapter deliberately splits promotion/relegation into a programmatic `rollover_survivors` view plus Chapter 3's own `SeasonTeamInline`, rather than building one combined route that also accepts the 3 promoted teams' names the way the FastAPI/PostgreSQL sibling course does — and why that split is a genuinely good fit for Django specifically.

 [View solution](#)

EXERCISE 3

Complete a full real season (enter results for enough fixtures to give every team a real record), call `relegation_preview`, create a new empty Season in the admin, call `rollover_survivors`, confirm the new season has exactly 17 `SeasonTeam` rows, then add 3 promoted teams via `SeasonTeamInline` (using the "+" popup for at least one genuinely new team) and confirm the new season's roster reaches exactly 20.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- **`compute_league_table()`** — Chapter 7's own query logic, extracted so this chapter can reuse it as the real source of truth for relegation
- **`relegation_preview`** — a read-only GET, a sanity check before anything real is committed
- **`rollover_survivors`** — carries the 17 real survivors forward via `bulk_create()` inside `transaction.atomic()`, the write-side counterpart to Chapter 6's `bulk_update()`
- **Deliberate split from the sibling course** — survivors are automated; the 3 promoted teams go through Chapter 3's own `SeasonTeamInline`, including its real "+" popup for a genuinely new team
- **"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
- **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: Django & MySQL

Chapter 10 · Styling & the Gameweek/Season Selector UI

Every page this course has built since Chapter 4 has been visited against a specific, hardcoded `gameweek_id` or `season_id` baked straight into the URL. This chapter replaces that with a real, working selector — and gives the app the dark-theme styling it's been missing since Chapter 1. The selector itself, though, ends up looking genuinely different from the FastAPI sibling's own version, for a real, structural reason worth walking through rather than skipping past.

A Server-Rendered Selector, Not a Client-Side One

`plpredict-fastapi1`'s own `fixture_entry.js` fetches its team list from a JSON route on every page load, because FastAPI's vanilla-JS frontend has nothing else to hand it that data — so its selector, quite reasonably, broadcasts a `CustomEvent` and lets JavaScript re-fetch and re-render whatever changed. Chapter 4's own `fixture_entry` view is built completely differently: it already renders the team grid, the gameweek, and everything else *server-side*, via `{% for team in teams %}`, before the page is ever sent to the browser. A season/gameweek selector for a page built this way doesn't need to swap data into an already-loaded page at all — the far simpler, more idiomatic answer is **real navigation**: picking a different gameweek just loads a different, freshly server-rendered page.

A Context Processor: Every Page Gets the Selector's Own Data, Automatically

The FastAPI sibling's own real problem — letting `fixtures.js`, `predictions.js`, and both table scripts each react to a selection change without any of them needing to know about the others — still exists here, just one layer down the stack. Every page this course has built needs the same season/gameweek dropdown data available in its own template context, without every single view having to remember to fetch and pass it in by hand. Django's own answer to exactly that problem is a **context processor**:

```
# predictor/context_processors.py
from .models import Season, Gameweek

def selector_context(request):
    seasons = Season.objects.order_by('-start_date')

    gameweek_id = request.resolver_match.kwargs.get('gameweek_id') if request.resolver_match.kwargs.get('gameweek_id') else None
    if gameweek_id:
        current_gameweek = Gameweek.objects.filter(pk=gameweek_id).select_related('season')
        current_season = current_gameweek.season if current_gameweek else None
    else:
        current_season = seasons.filter(is_current=True).first() or seasons.first()

    gameweeks = (
        Gameweek.objects.filter(season=current_season).order_by('number')
        if current_season else Gameweek.objects.none()
    )
    return {
        'all_seasons': seasons,
        'all_gameweeks': gameweeks,
        'current_season': current_season,
        'current_gameweek_id': int(gameweek_id) if gameweek_id else None,
    }
```

```
# settings.py (TEMPLATES, additions)
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.request',
                # ...the usual defaults...
                'predictor.context_processors.selector_context',
            ],
        },
    },
]
```

DJANGO'S OWN IDIOMATIC ANSWER TO THE SAME DECOUPLING PROBLEM

`request.resolver_match.kwargs` is what makes this genuinely automatic: Django sets `resolver_match` on the request the moment its own URL routing has matched a view, well before any template ever renders — so `selector_context()` can look at whichever URL the current request actually hit, pull `gameweek_id` straight out of it if present, and resolve the right season from there. Any template rendered anywhere in the app — the fixture-entry page, a predictions page, either league table — automatically receives `all_seasons` / `all_gameweeks` / `current_season` without its own view ever mentioning the selector at all. That's the same real goal `selection-changed` serves for the FastAPI sibling — no page needs to know the selector's own internals — solved at the template-context layer instead of with a browser event, because Django's own rendering pipeline already has a real, built-in mechanism for exactly this.

The Selector Partial & Real Navigation

```
<!-- predictor/templates/predictor/_selector.html -->
<div class="selector-bar">
  <select onchange="location.href = '/seasons/' + this.value + '/default-gameweek/'">
    {% for season in all_seasons %}
      <option value="{{ season.id }}" {% if season == current_season %}selected{% endif %}>
        {{ season.name }}
      </option>
    {% endfor %}
  </select>

  <select onchange="location.href = '/gameweeks/' + this.value + '/fixtures/'">
    {% for gw in all_gameweeks %}
      <option value="{{ gw.id }}" {% if gw.id == current_gameweek_id %}selected{% endif %}>
        Gameweek {{ gw.number }}
      </option>
    {% endfor %}
  </select>
</div>
```

`base.html` includes it once, with `{% include 'predictor/_selector.html' %}` near the top of the page, so every template that extends `base.html` gets the same live selector for free. Switching the gameweek dropdown navigates directly to that gameweek's own fixture-entry

page. Switching the season dropdown is one step more involved — there's no single "the" gameweek to jump to until one is picked — so it goes through one small, real redirect view instead:

```
# predictor/views.py (additions)
from django.shortcuts import redirect

def season_default_gameweek(request, season_id):
    season = get_object_or_404(Season, pk=season_id)
    first_gameweek = Gameweek.objects.filter(season=season).order_by('number').first()
    if not first_gameweek:
        return JsonResponse({'error': 'This season has no gameweeks yet'}, status=404)
    return redirect(f'/gameweeks/{first_gameweek.id}/fixtures/')
```

TWO JSON ROUTES THIS COURSE NEVER NEEDS — ANOTHER "ONE FETCH CALL SKIPPED" MOMENT

plpredict-fastapi1's own Chapter 10 has to add `GET /api/seasons` and `GET /api/seasons/{id}/gameweeks` specifically to feed its own selector's two dropdowns. This chapter needs neither: `all_seasons` / `all_gameweeks` already arrive in every template's own context via `selector_context()`, rendered directly with `{% for %}` the same way Chapter 4's own team grid was — the exact same "server-side rendering skips a whole client-side fetch" pattern Chapter 4's own finding-box already established, showing up here a second time.

Resolving Chapter 4's Own Flagged Gap — Server-Side, Not With a Re-Fetch

Chapter 4 was honest that `usedTeamIds` only lived in the page's own JavaScript memory, empty on every reload and reflecting only fixtures added during that exact visit. The FastAPI sibling fixes this with a live client-side re-fetch on every selection change. Django's own `fixture_entry` view already queries the database directly on every render — the far more direct fix is to compute the real, correct set of used teams *there*, and bake it straight into the HTML the server sends:

```
# predictor/views.py (fixture_entry, updated)
@ensure_csrf_cookie
```

```
def fixture_entry(request, gameweek_id):
    gameweek = get_object_or_404(Gameweek, pk=gameweek_id)
    season_teams = (
        SeasonTeam.objects
        .filter(season=gameweek.season)
        .select_related('team')
        .order_by('team__name')
    )
    teams = [st.team for st in season_teams]

    existing_fixtures = Fixture.objects.filter(gameweek=gameweek)
    used_team_ids = set()
    for fx in existing_fixtures:
        used_team_ids.add(fx.home_team_id)
        used_team_ids.add(fx.away_team_id)

    return render(request, 'predictor/fixture_entry.html', {
        'gameweek': gameweek,
        'teams': teams,
        'used_team_ids': used_team_ids,
    })
```

```
<!-- predictor/templates/predictor/fixture_entry.html (updated) -->
<div id="team-grid" class="team-grid">
    {% for team in teams %}
        <button type="button" data-team-id="{{ team.id }}"
            {% if team.id in used_team_ids %}disabled{% endif %}>{{ team.short_name }}<
    {% endfor %}
</div>
```

`fixtures.js` then only needs to seed its own JavaScript `Set` from what the DOM already correctly shows, instead of trusting a variable that started life empty:

```
// static/predictor/fixtures.js (updated)
let usedTeamIds = new Set(
    Array.from(document.querySelectorAll('#team-grid button[disabled]'))
        .map(btn => Number(btn.dataset.teamId))
);
```

```
function disableUsedButtons() {
  document.querySelectorAll('#team-grid button').forEach(btn => {
    btn.disabled = usedTeamIds.has(Number(btn.dataset.teamId));
  });
}
```

`addFixture()`, from Chapter 4, is updated to call `disableUsedButtons()` too, replacing its own original inline disabling loop — the same logic, now written once and shared, rather than kept as two near-identical copies.

THE SAME REAL GAP, FIXED A GENUINELY DIFFERENT WAY

Reloading the page — or navigating to a different gameweek and back through the new selector — now reliably shows the real, correct disabled buttons every single time, because `used_team_ids` is recomputed fresh from `Fixture.objects.filter(gameweek=gameweek)` on every request, not carried over in memory from a previous page view at all. The FastAPI sibling achieves the identical real outcome with a dedicated client-side re-fetch call on every selection change; this course achieves it by never letting the client-side state be wrong in the first place, since the server-rendered HTML was already correct the moment it arrived.

Real Styling

```
/* static/predictor/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; }

.selector-bar {
  display: flex;
  gap: 0.7rem;
  margin-bottom: 1.5rem;
}

.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: #0c1f18;
  color: #7fd6b4;
  text-transform: uppercase;
  font-size: 0.7rem;
  letter-spacing: 0.07em;
}
```

The same green accent (`#44b78b` / `#7fd6b4`) 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 this chapter's selector uses real page navigation plus a Django context processor rather than a JS CustomEvent the way the FastAPI/PostgreSQL sibling does, naming the specific architectural fact about `fixture_entry` (and every other page in this course) that makes navigation the more natural choice here.

 [View solution](#)

EXERCISE 2

Trace through exactly what `selector_context()` does when a request hits `/gameweeks/5/fixtures/`, explaining how it correctly resolves gameweek 5's own season for the season dropdown even though the function's own signature never receives a `season_id` directly.

 [View solution](#)

EXERCISE 3

Build the selector into a real setup with at least two seasons and two gameweeks, enter a fixture in one gameweek, navigate to the other gameweek and back using only the selector dropdowns, and confirm the previously-entered fixture's own two teams are still shown disabled — then explain specifically why this works even though no client-side state was ever preserved across the navigation at all.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE

- **Real navigation, not a CustomEvent** — `fixture_entry` is already fully server-rendered per gameweek, so a different selection is just a different page
- **`selector_context()`** — a Django context processor giving every template access to `all_seasons/all_gameweeks/current_season` automatically, the backend-layer analog to the sibling's own selection-changed event
- **`request.resolver_match.kwargs`** — how the context processor knows which season "belongs" to the current URL with no view needing to say so explicitly
- **`season_default_gameweek`** — a small redirect view handling the "season picked, no gameweek chosen yet" case
- **Two JSON routes skipped** — no GET `/seasons` or GET `/seasons/{id}/gameweeks` needed at all, the same server-rendering payoff Chapter 4 already established for the team grid
- **`usedTeamIds`, fixed server-side** — computed fresh in `fixture_entry` on every request and baked into the disabled attribute directly, rather than re-fetched by the client

- **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: Django & MySQL

Chapter 11 · Deployment

Every earlier chapter ran with `python manage.py runserver` — a genuinely fine development server, and a genuinely wrong choice for real production traffic.

Running With Multiple Workers

```
pip install gunicorn
```

```
gunicorn pl_predictor_site.wsgi:application -w 4 --bind 0.0.0.0:8000
```

`pl_predictor_site.wsgi:application` — not `main:app` — is the real entry point Chapter 1's own `django-admin startproject pl_predictor_site .` already generated, sitting at `pl_predictor_site/wsgi.py`. Gunicorn manages several separate worker **processes** running that WSGI application — real request-handling capacity beyond what one process provides, with automatic restarts if a worker crashes.

NO CORS NEEDED HERE EITHER, FOR THE SAME UNDERLYING REASON AS THE FASTAPI SIBLING

Every page this course has built since Chapter 4 has been rendered by this same Django project, and every `fetch()` call in its own JavaScript has always targeted a relative, same-origin path — `/fixtures/${id}/predictions/upsert/`, never a separate host. There's never been a second origin in the picture, the same real fact this site's own `plpredict-fastapi1` and `fastapi-food-tracker-1` courses both already established — just arrived at here via Django's own template rendering rather than FastAPI's `StaticFiles` mount.

Static Files: A Real New Requirement `DEBUG=False` Introduces

Every chapter since Chapter 4 has relied on Django's own development server quietly serving `predictor/static/predictor/fixtures.js`, `style.css`, and everything else — a genuine, documented convenience that only exists while `DEBUG = True`. The moment `DEBUG` is turned

off for production, Django's dev-server-only static handling stops working entirely, and nothing at all is served at `/static/...` unless something else is configured to do it.

```
# pl_predictor_site/settings.py (additions)
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / 'staticfiles'

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    # ...the rest of the default middleware, unchanged...
]

STORAGES = {
    'staticfiles': {
        'BACKEND': 'whitenoise.storage.CompressedManifestStaticFilesStorage',
    },
}
```

```
pip install whitenoise
python manage.py collectstatic --noinput
```

A REAL REQUIREMENT DJANGO INTRODUCES THAT FASTAPI'S OWN STATICFILES MOUNT NEVER HAS

`plpredict-fastapi1`'s own `StaticFiles` mount serves its frontend files identically in development and production — nothing changes at deployment time. Django's own dev-server static handling is explicitly, deliberately development-only; `collectstatic` gathers every app's own static files (including `predictor/static/predictor/fixtures.js`, auto-discovered by Django's own `AppDirectoriesFinder` with no extra config needed) into one real directory, `STATIC_ROOT`, and WhiteNoise's own middleware serves that directory efficiently straight from the WSGI app itself — no separate static file server required, though a reverse proxy serving `STATIC_ROOT` directly is an equally valid, slightly faster alternative for a busier deployment.

A Full-Circle Caveat on the Admin

Chapter 3's own tip-box named the Django admin as a genuine advantage this course earns largely for free — `SeasonTeamInline` alone replaced what would otherwise have been real,

hand-written CRUD routes. Before a real deployment, that same feature deserves an honest second look: it's a real, working interface for editing every table in this app, reachable by anyone who finds the URL and guesses (or brute-forces) a password.

```
# pl_predictor_site/settings.py (production-ready)
import os

DEBUG = os.environ.get('DJANGO_DEBUG', 'False') == 'True'
ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS', '').split(',')

```

```
# pl_predictor_site/urls.py (production-ready)
import os
from django.contrib import admin
from django.urls import path

admin_path = os.environ.get('DJANGO_ADMIN_PATH', 'admin/')

urlpatterns = [
    path(admin_path, admin.site.urls),
    # ...the rest of this course's own urlpatterns, unchanged...
]
```

THE SAME FEATURE, PRAISED IN CHAPTER 3, CAVEATED HERE

`DEBUG = False` stops Django's own real debug error pages — which, left on, expose settings, SQL queries, and full stack traces to any visitor who triggers a 500 — from ever being shown in production. `ALLOWED_HOSTS` is a second, separate real guard: with `DEBUG = False`, Django refuses any request whose `Host` header isn't in this list outright, with a real `400` and no view code ever running at all. Moving the admin off its default `/admin/` path is genuinely only security-through-obscurity — a real deterrent against the automated scanners that specifically probe for the well-known default path, not a substitute for a strong password on every admin account, which still matters more.

MySQL Connections at Scale: Django's Own Default Is the Opposite Problem

`plpredict-fastapi1`'s own PostgreSQL sibling has to cap down an aggressively pooling SQLAlchemy engine. Django's own MySQL backend starts from the completely opposite default:

CONN_MAX_AGE DEFAULTS TO 0 — A BRAND-NEW CONNECTION, EVERY SINGLE REQUEST

With no `CONN_MAX_AGE` set, Django's own documented default is `0`: every request opens a fresh MySQL connection at the start and closes it again at the end, regardless of which of the 4 gunicorn workers handles it. There's no connection-count explosion risk the way SQLAlchemy's own pool-per-worker-process default creates for the PostgreSQL sibling — the real cost here is paid on *every single request* instead, as a genuine TCP handshake plus a real MySQL authentication exchange, adding measurable latency to literally every page view under this app's own unmodified defaults.

```
# pl_predictor_site/settings.py (production-ready)
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get('DB_NAME', 'pl_predictor'),
        'USER': os.environ.get('DB_USER', 'root'),
        'PASSWORD': os.environ.get('DB_PASSWORD', ''),
        'HOST': os.environ.get('DB_HOST', 'localhost'),
        'PORT': os.environ.get('DB_PORT', '3306'),
        'OPTIONS': {'charset': 'utf8mb4'},
        'CONN_MAX_AGE': 60,
        'CONN_HEALTH_CHECKS': True,
    }
}
```

`CONN_MAX_AGE=60` is the standard fix: each worker process's own thread reuses the same real MySQL connection across requests for up to 60 idle seconds before Django closes and reopens it, instead of paying the connection cost on every single call. `CONN_HEALTH_CHECKS=True` — a real Django 4.1+ setting, safely available given this course's own MySQL 8+/Django 4.2+ baseline from Chapter 2 — pings a reused connection before handing it to a view, transparently reconnecting if MySQL's own `wait_timeout` already closed it server-side in the meantime, rather than surfacing a stale-connection error to a real user.

TURNING PERSISTENCE ON REOPENS THE SAME REAL SIZING QUESTION, FROM DJANGO'S OWN ANGLE

Django's own database connections are held per *thread*, not per worker process. Running gunicorn as `-w 4 --threads 2` for real request-handling concurrency means up to $4 \times 2 = 8$ genuine persistent MySQL connections held open at once with `CONN_MAX_AGE=60` in place — not just 4, since worker count alone doesn't capture the real multiplier once threads are involved. Comfortably inside MySQL's own default `max_connections` of **151**, with real headroom left for an admin session, a monitoring agent, or anything else connecting to the same database — but worth knowing the real formula (workers $\times$ threads) rather than assuming the worker count alone tells the whole story, the same discipline the PostgreSQL sibling's own worked calculation teaches from the opposite direction.

Chapter 1's Own Password Gap, Closed

Chapter 1's own `DATABASES` block shipped with `'PASSWORD': ''` and an explicit comment — `# set a real password in production` — flagging exactly this moment. The block above finally makes good on it, pulling every credential from a real environment variable instead of a hardcoded value, with the empty string surviving only as a local-development fallback.

`SECRET_KEY` needs the identical treatment:

```
# pl_predictor_site/settings.py (production-ready)
SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY')
```

SECRET_KEY HAS NO FASTAPI EQUIVALENT TO POINT BACK TO — IT'S GENUINELY DJANGO-SPECIFIC

Django's own `SECRET_KEY` — the random value `startproject` generates and writes straight into `settings.py` — signs session cookies and CSRF tokens, the exact CSRF mechanism Chapter 4 already leaned on directly. Deploying with the auto-generated development value still committed to version control means anyone who's ever seen this project's own repository history could forge a valid session or CSRF token against the live site. A real production `SECRET_KEY`, generated fresh and set only as an environment variable on the deployment host, is non-negotiable in a way that has no real analog in the FastAPI/PostgreSQL sibling's own deployment chapter at all.

Schema Changes After Deployment — Already Solved, With a Caution

`plpredict-fastapi1`'s own SQLAlchemy schema is created with `Base.metadata.create_all(bind=engine)` — a one-shot table creator that can't alter a table that already exists, leaving that sibling course needing a genuinely separate tool (Alembic) it

never actually builds. This course has never had that gap: `python manage.py makemigrations / migrate` has been the real, working schema-change tool since Chapter 2, and it's exactly as usable after deployment as it was in development.

THE REAL CAUTION ISN'T "DOES THIS WORK" — IT'S "DON'T RUN MAKEMIGRATIONS BLINDLY AGAINST PRODUCTION DATA"

The safe, standard practice is running `makemigrations` locally against a development database, reviewing the real migration file it generates, committing that file to version control, and only ever running `migrate` — never `makemigrations` itself — on the production server. Running `makemigrations` directly against live production data risks Django prompting for a default value on a genuinely ambiguous schema change (a new required field, say) interactively, in a context — a deployment script, a production shell — where there's no real developer present to answer that prompt correctly.

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. The same reverse proxy is also the natural place to serve `STATIC_ROOT` directly, as a faster alternative to WhiteNoise, once traffic genuinely warrants it.

A Production Checklist

- Run with `gunicorn pl_predictor_site.wsgi:application -w 4`, never `runserver`.
- Run `collectstatic` and serve `STATIC_ROOT` via WhiteNoise or a reverse proxy — Django's own dev server no longer serves static files at all once `DEBUG=False`.
- Set `DEBUG=False`, a real `ALLOWED_HOSTS`, and consider moving the admin off its default path.
- Set `CONN_MAX_AGE` and `CONN_HEALTH_CHECKS=True`, and size the real workers × threads connection count against MySQL's own `max_connections`.
- Set a real `DATABASES` password and a fresh, real `SECRET_KEY` — both as genuine environment variables, closing the exact gap Chapter 1's own comment flagged.
- Run `makemigrations` locally and commit the result; only run `migrate` itself against production.

- 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, integrated into the existing Astro-based site.

Hands-On Exercises

EXERCISE 1

Explain why this course never needed CORS configuration anywhere, tracing the reason back to how templates and static files have been served since Chapter 4.

[!\[\]\(d61c93dcb4252e81d45d493f87b61f4c_img.jpg\) View solution](#)

EXERCISE 2

Explain why Django's own default `CONN_MAX_AGE=0` creates a genuinely different kind of problem than SQLAlchemy's own pooling defaults do for the PostgreSQL sibling, then calculate the real number of persistent MySQL connections a `unicorn -w 4 --threads 2` deployment can hold open at once with `CONN_MAX_AGE=60` in place.

[!\[\]\(85738c880ffba1a5c2358403c2432780_img.jpg\) View solution](#)

EXERCISE 3

Explain what would actually happen if this app were deployed with `DEBUG` left `True` and `ALLOWED_HOSTS` left empty — both the real information a visitor could see on a triggered error, and the real error a normal request would hit regardless.

[!\[\]\(02131cbb0324c9ee76b27b0fe6fec80d_img.jpg\) View solution](#)

CHAPTER 11 QUICK REFERENCE

- **Run with:** `gunicorn pl_predictor_site.wsgi:application -w 4`, never runserver in production
- **No CORS anywhere:** same real reason as the FastAPI sibling — one origin, the whole way through
- **Static files:** a real new requirement `DEBUG=False` introduces — `collectstatic` + `WhiteNoise`, since Django's dev server no longer serves them
- **Full-circle caveat:** the admin, Chapter 3's own praised feature, needs `DEBUG=False`, real `ALLOWED_HOSTS`, and ideally a non-default path
- **Real connection math, the opposite problem:** `CONN_MAX_AGE=0` by default means a fresh connection per request; `CONN_MAX_AGE=60` + 4 workers × 2 threads = up to 8 real persistent connections, against MySQL's own default `max_connections` of 151
- **Chapter 1's own flagged gap, closed:** real `DATABASES` credentials and a real `SECRET_KEY`, both from environment variables
- **Schema changes:** already solved by `makemigrations/migrate`, unlike the sibling's own `create_all()` gap — just never run `makemigrations` itself against production
- **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: Django & MySQL

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 built largely for free, a fast fixture-entry UI, four scored prediction sources, two real league tables, promotion/relegation, a real navigation-based 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 Django 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, two-part fix to this course's own hardcoded paths
JSON views + Astro frontend	Django returns data only; an Astro page/island renders it	Highest — every server-rendered page's own 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-view rewrite defeats much of the point of already having a working, server-rendered Django app. 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 Django itself sees plain `/gameweeks/7/fixtures/` requests exactly as every earlier chapter tested them, with no server-side URL pattern changes needed at all.

Two Real Kinds of Absolute Path — And Only One of Them

`FORCE_SCRIPT_NAME` Actually Fixes

`catalog-django1`'s own sibling capstone found real, server-generated `{% url %}` links breaking once mounted at a sub-path, fixed with Django's own `FORCE_SCRIPT_NAME` setting. This course hits a genuinely more layered version of that same problem, for a real, traceable reason: since Chapter 9, this course has deliberately hand-written every route as a raw path string rather than using Django's `reverse()` /named-URL system at all.

```

# pl_predictor_site/settings.py (additions)
BASE_PATH = os.environ.get('DJANGO_BASE_PATH', '') # e.g. '/predictor' in production,

```

```
FORCE_SCRIPT_NAME = BASE_PATH or None
```

THE GOOD NEWS: THE ADMIN'S OWN LINKS ARE FIXED AUTOMATICALLY

`FORCE_SCRIPT_NAME` genuinely does fix every link the Django admin itself generates — `SeasonTeamInline`'s own save-and-continue redirects, its "view on site" links, its changelist navigation — because `admin.site.urls` builds every one of those internally via `reverse()`, and `reverse()` is exactly the machinery `FORCE_SCRIPT_NAME` teaches to prepend the real mount prefix. This payoff is real, and it costs nothing extra beyond the one setting above.

THE BAD NEWS: IT HAS ZERO EFFECT ON A RAW STRING THIS COURSE WROTE BY HAND

`season_default_gameweek`'s own real route, from Chapter 10, reads `return redirect(f'/gameweeks/{first_gameweek.id}/fixtures/')` — a literal, hardcoded absolute string, not anything built via `reverse()`. `FORCE_SCRIPT_NAME` only ever rewrites paths Django's *own* URL-generation functions produce; it has no way to know that this particular string was ever meant to be treated as one of them. Even with `FORCE_SCRIPT_NAME = '/predictor'` correctly set, this exact line still redirects a browser to `/gameweeks/7/fixtures/` — missing the prefix entirely — which nginx has no `location` block for, falling through to the Astro site's own catch-all instead of the predictor. Every fetch call in `fixtures.js`, `predictions.js`, `rollover.js`, and the selector partial has the exact same real problem, for the exact same reason: this course chose raw path strings deliberately, back in Chapter 9, "matching established convention" — and that choice has a real, concrete consequence right here.

The Real Fix: One Settings-Driven Prefix, Applied Everywhere by Hand

Every hand-written path across the whole app gets the same treatment — built from `BASE_PATH` explicitly, rather than left to `FORCE_SCRIPT_NAME` to rescue on its own:

```
# predictor/views.py (season_default_gameweek, fixed)
from django.conf import settings

def season_default_gameweek(request, season_id):
    season = get_object_or_404(Season, pk=season_id)
    first_gameweek = Gameweek.objects.filter(season=season).order_by('number').first()
    if not first_gameweek:
```

```
return JsonResponse({'error': 'This season has no gameweeks yet'}, status=404)
return redirect(f'{settings.BASE_PATH}/gameweeks/{first_gameweek.id}/fixtures/')

```

```
// static/predictor/fixtures.js – before (absolute, breaks once mounted at a sub-path)
fetch(`/gameweeks/${gameweekId}/fixtures/create/`, { ... });

// after – relative, resolves against the page's own <base> URL
fetch(`gameweeks/${gameweekId}/fixtures/create/`, { ... });

```

The client-side JS fix is the FastAPI/PostgreSQL sibling course's own real technique, applied here for the same underlying reason: browser-side `fetch()` resolution has nothing to do with Django's own `FORCE_SCRIPT_NAME` at all — it's resolved entirely client-side, against the document's own base URL, which needs a real `<base>` tag to be correct:

predictor/templates/predictor/base.html

```
<head>
  {% if request.META.SCRIPT_NAME %}<base href="{ request.META.SCRIPT_NAME }"/">{% endif %}
  ...
</head>

```

THE SAME ONE SETTING QUIETLY FEEDS BOTH FIXES

Django's own `WSGIRequest` sets `request.META['SCRIPT_NAME']` directly from `FORCE_SCRIPT_NAME` whenever it's configured — the exact setting already added above for the admin's own sake turns out to be precisely the right value for the `<base>` tag too, with no second setting needed. Locally, with `BASE_PATH = ''`, `FORCE_SCRIPT_NAME` is `None`, `request.META.SCRIPT_NAME` is empty, and the `{% if %}` above simply omits the tag — `fetch('gameweeks/...')` then resolves correctly against `http://localhost:8000/` with no base tag present at all, exactly the same "localhost needs nothing extra" property the FastAPI sibling's own version has.

`STATIC_URL`, from Chapter 11, needs the identical prefix for the same reason every other hand-built path does: `STATIC_URL = f'{BASE_PATH}/static/'`.

FOUR REAL SIBLING COURSES, TWO OF THEM LANDING ON THE IDENTICAL MECHANISM COMBINATION

Personal Catalogue (PHP & MySQL) found absolute `header('Location: ...')` redirects breaking. Personal Catalogue (Django & PostgreSQL) found `{% url %}` links breaking, fixed entirely by `FORCE_SCRIPT_NAME` alone, since that course used `reverse()`-backed URLs consistently. Premier League Predictor (FastAPI & PostgreSQL) found absolute `fetch()` calls breaking, fixed with relative paths plus one `<base>` tag. This course needed *both* of those last two techniques at once, in the same app — `FORCE_SCRIPT_NAME` for the `reverse()`-backed admin, and the relative-path/base-tag technique for everything this course itself chose to hand-write — a direct, concrete consequence of Chapter 9's own deliberate "raw path strings, not named URLs" convention, four courses and one recurring underlying lesson later: the app never knows it's not at the web root unless something explicit tells it so, and exactly what that "something" looks like depends entirely on how each individual path was originally built.

Making It Feel Like Part of the Same Site

A reverse proxy solves reachability, not visual consistency. Chapter 10's own dark-theme CSS already gives the predictor a real look 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: JSON Views That Already Exist

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 — `league_table`, `prediction_table`, and `relegation_preview` already return real JSON from Chapters 7-9, with no new backend code required at all for an Astro island to `fetch()` them directly. 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, a working Django + MySQL connection, and Chapter 1's own flagged empty-password comment, closed in Chapter 11
2	The real Team/Season/SeasonTeam/Gameweek/Fixture schema, the dual related_name fix on Fixture, and the MySQL 8.0.16+ CheckConstraint gotcha
3	SeasonTeamInline and the Django admin as a real manual-entry tool, largely for free, plus the max_num/validate_max 20-team cap reused twice more later
4	The server-rendered click-to-pair fixture-entry UI, and Django's own real CSRF requirement — a genuinely new concern the FastAPI sibling never faces
5	The Prediction model, a real GeneratedField exploiting NULL's own uniqueness rules as MySQL's answer to a partial unique index, and update_or_create()
6	The real, confirmed point values (40/10), score_prediction(), and bulk_update() rescoring every prediction on a corrected result
7	The real league table via a hand-written UNION ALL and Team.objects.raw(), and a genuine Django ORM join-multiplication gotcha found and avoided
8	The prediction league table's two-pass guest-averaging CTE, connection.cursor()+dictfetchall(), and MySQL's more lenient NULL-in-UNION typing
9	Promotion/relegation — compute_league_table() reused, bulk_create()+transaction.atomic(), and a deliberate split reusing SeasonTeamInline for promoted teams
10	A real navigation-based selector and a context processor, replacing a JS CustomEvent, plus a server-side fix to Chapter 4's own flagged usedTeamIds gap
11	Deployment — gunicorn, collectstatic+WhiteNoise, DEBUG/ALLOWED_HOSTS/admin hardening, and the real CONN_MAX_AGE connection-math calculation
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 this course's own lack of any login requirement already anticipates). 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 fixed version of `season_default_gameweek` using `settings.BASE_PATH`, then show the exact Location header value the unfixed version and the fixed version would each send to the browser when mounted at `/predictor/` — confirming `FORCE_SCRIPT_NAME` alone changes neither one.

 [View solution](#)

EXERCISE 2

Explain why the same `FORCE_SCRIPT_NAME` setting that fixes every link the Django admin generates has no effect at all on `season_default_gameweek`'s own `redirect()` call, tracing the distinction back to `reverse()`-built paths versus a raw hardcoded string.

 [View solution](#)

EXERCISE 3

Mount the app behind a real nginx location `/predictor/` block with `FORCE_SCRIPT_NAME` set but `BASE_PATH` not yet applied to any hand-written path, confirm a `SeasonTeamInline` save-and-continue redirect inside the admin works correctly, then confirm `season_default_gameweek` genuinely 404s at the wrong URL until the `BASE_PATH` fix from this chapter is applied to it too.

 [View solution](#)

CHAPTER 12 QUICK REFERENCE

- **Path-mounted reverse proxy** — one nginx `location /predictor/` block using `proxy_pass` to Unicorn, stripping the prefix

- **FORCE_SCRIPT_NAME** — fixes the Django admin's own links automatically, since they're built via `reverse()`
- **The real gap** — every route this course hand-wrote since Chapter 9 uses a raw absolute path string, which `FORCE_SCRIPT_NAME` cannot touch at all
- **The fix** — a real `BASE_PATH` setting, applied explicitly to every `redirect()` call, every `fetch()` call, and `STATIC_URL`, plus one `<base>` tag templated from `request.META.SCRIPT_NAME`
- **Four sibling courses, a genuine hybrid here** — this course needs both `FORCE_SCRIPT_NAME` (for the admin) and the relative-path/base-tag technique (for everything hand-written), unlike either sibling alone
- **JSON views** — already exist, for free, at `league_table/prediction_table/relegation_preview`; 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: Django & MySQL (12/12 chapters)