

PERSONAL MULTI-TECHNOLOGY PROJECTS

Premier League Predictor: FastAPI & Redis

A real weekly Premier League prediction tracker built on Redis — hashes, sets and sorted sets standing in for a relational database, with every guard that a database would have provided written by hand and found by failing without it. Two league tables maintained as sorted sets, atomic Lua scripts and WATCH for check-then-act problems, a measured head-to-head against PostgreSQL, and a real deployment mounted onto an existing site.

12 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: PROJECT OVERVIEW & FASTAPI + REDIS SETUP

Premier League Predictor: FastAPI & Redis

Chapter 1 · Project Overview & FastAPI + Redis Setup

The Shared Spec

This app tracks weekly predictions for every fixture across a real 38-gameweek Premier League season, scoring four genuinely different prediction sources against what actually happened:

- **The user's own prediction** — entered by hand each week
- **The BBC's expert** (currently Chris Sutton) — their own published scoreline pick
- **The week's guest(s)** — genuinely variable; some weeks have more than one guest, whose individual scores are averaged into a single "guest" figure for that fixture
- **The BBC's own AI prediction** — manually transcribed each week from the BBC's own published article, not scraped or generated by this app

Scoring uses the real point values confirmed directly by the user during this project's own reference implementation, back in *Premier League Predictor: FastAPI & PostgreSQL*'s own Chapter 6: **40 points for a correct score** (the exact scoreline matched) and **10 points for a correct result** (the W/D/L outcome matched even if the scoreline didn't). A real league table (3/1/0 points, goal difference, won/drawn/lost) sits alongside a second, parallel *prediction* league table ranking how each of the four real sources is actually performing across the season. The season itself holds 20 teams across 38 gameweeks, with the bottom 3 auto-calculated for relegation and the 3 promoted teams entered by hand. The fixture-entry UI stays deliberately fast — paired click targets and 20 team buttons, not a dropdown-heavy form repeated ten times a week.

This Variant's Own Real Angle

The already-completed *Premier League Predictor: FastAPI & PostgreSQL* course built this exact spec as a real, ordinary relational schema — teams, fixtures, and predictions as tables, joined and aggregated with real SQL. This course asks a genuinely different question: what happens if the database underneath it is Redis instead?

Redis isn't a relational database with SQL removed — it's a fundamentally different kind of store, built around a small set of real data structures (strings, hashes, lists, sets, and — the one this course leans on hardest — sorted sets) rather than tables and rows. A sorted set stores members each tagged with a real numeric score, kept continuously ordered by that score, with real $O(\log n)$ insertion and real $O(\log n + m)$ range reads. That's not a workaround for a league table — it's close to the data structure a league table actually is.

AN HONEST SCOPE NOTE — READ THIS BEFORE CONTINUING

This is deliberately the **narrower** of the two FastAPI-backed variants in this project. Redis genuinely excels at the two things this course leans on it for — a live, continuously ordered league table and a live, continuously ordered prediction leaderboard — but it was never designed as a general-purpose store for rich relational history. The full PostgreSQL variant can run an arbitrary SQL query across every fixture and every prediction ever recorded; this course's own real answer to that same kind of question is honestly weaker, and Chapter 9 is dedicated entirely to working through exactly where that weakness shows up, with real, concrete examples rather than a vague disclaimer. The goal here isn't "prove Redis can do everything" — it's "find out, honestly, where it genuinely fits and where it genuinely doesn't."

Real Stack Setup: FastAPI + redis-py

`redis-py` — installed as the plain `redis` package — is the real, official Python client for Redis. Its own async support used to live in a separate package, `aioredis`; that project is now archived, with its functionality merged directly into `redis-py` itself as `redis.asyncio`, confirmed directly against Redis's own current client documentation.

```
$ pip install fastapi uvicorn redis
```

A REAL, DATED DIVERGENCE FROM REDIS'S OWN OFFICIAL TUTORIAL

Redis's own official FastAPI tutorial — a genuinely current article, published and updated this year — still wires its Redis client up using FastAPI's older `@app.on_event('startup')` hook. Checked directly against FastAPI's own current documentation, that decorator is real but explicitly deprecated: FastAPI's own docs now recommend the `lifespan` async context manager instead, and are direct about why — startup and shutdown code sharing one function and one scope, rather than two separately registered handlers. This chapter uses the current recommended pattern, not the one

shown in even a freshly-published official tutorial — a genuine reminder that "recently published" and "reflects the current recommendation" aren't always the same thing.

Building this app's own connection lifecycle the current way:

```
# main.py
from contextlib import asynccontextmanager
from fastapi import FastAPI
import redis.asyncio as redis

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.redis = redis.Redis(host="localhost", port=6379, decode_responses=True)
    yield
    await app.state.redis.aclose()

app = FastAPI(lifespan=lifespan)

@app.get("/health")
async def health():
    pong = await app.state.redis.ping()
    return {"status": "ok", "redis_ping": pong}
```

Run against a real, local Redis instance and confirmed with a real request:

```
$ curl http://localhost:8000/health
{"status":"ok","redis_ping":true}
```

A Real Preview of the Primitive This Course Is Built Around

Chapter 2 builds the full real data model on top of Redis's own sorted sets — this chapter closes with a small, genuine taste of the exact command shape the rest of the course leans on, so the real payoff isn't abstract:

```
@app.get("/sorted-set-demo")
async def sorted_set_demo():
```

```

r = app.state.redis
await r.delete("demo:table")
await r.zadd("demo:table", {"Arsenal": 78, "Chelsea": 71, "Liverpool": 82})
top = await r.zrange("demo:table", 0, -1, desc=True, withscores=True)
return {"top_teams": top}

```

```

$ curl http://localhost:8000/sorted-set-demo
{"top_teams":[["Liverpool",82.0],["Arsenal",78.0],["Chelsea",71.0]]}

```

ALREADY ORDERED, WITH ZERO SORTING CODE

Three teams were added in an arbitrary order — Arsenal, Chelsea, Liverpool — and the real response came back already sorted by points, highest first, with no `ORDER BY`, no application-side `.sort()` call, and no separate query at all. This is the real, structural case for choosing Redis for a live league table: the ordering isn't computed on read, it's maintained continuously as members are added, exactly the property a table that updates every time a result is entered actually wants.

WORTH KNOWING NOW, BUILT ON PROPERLY LATER

That same ordering guarantee has a real, sharp edge a Premier League table can't ignore: Redis resolves tied scores by the member's own name, not by anything meaningful like goal difference. Real Premier League points totals tie constantly. This chapter doesn't build the fix — Chapter 7's own league-table implementation does — but it's worth knowing from the very first sorted set this course creates, not discovering by surprise later.

Hands-On Exercises

EXERCISE 1

Point this chapter's own `/health` endpoint at a port with no Redis instance listening at all, and run it for real. Report the exact real exception type and message you get back, and explain what the current `/health` endpoint's own code would actually do with it (hint: nothing catches it).

 [View solution](#)

EXERCISE 2

Reproduce this chapter's own tie-breaking warning with real code: add three teams to a sorted set with the exact same score, read them back with ZRANGE in descending order, and explain the real order you get by working out what Redis's own tie-breaking rule (ascending by member name) would produce once the whole set is reversed for a descending read.

 [View solution](#)

EXERCISE 3

In your own words, explain why this course frames itself as "deliberately narrower" than its own PostgreSQL sibling, rather than simply "a different but equally capable" implementation of the same spec. What real question is this course trying to honestly answer that a course confident Redis can do everything wouldn't actually be answering?

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **Same spec, different store** — the identical four-source prediction/scoring system already built once on PostgreSQL, now built on Redis
- **redis.asyncio** — the real, current async client, merged from the now-archived aioredis package
- **lifespan, not on_event** — the real current FastAPI pattern, diverging deliberately from even a freshly-published official Redis tutorial
- **Sorted sets stay ordered on write** — verified: no sort step needed on read, the real reason this course reaches for Redis at all
- **An honest, upfront limitation** — tied scores break by member name, not by anything meaningful; Chapter 7 builds the real fix

CHAPTER 2: MODELING THIS IN REDIS

Premier League Predictor: FastAPI & Redis

Chapter 2 · Modeling This in Redis

Chapter 1 connected to Redis and previewed one of its real data structures. Before any real feature gets built, this chapter answers a more basic question: what does "teams," "seasons," and "fixtures" even look like once there's no schema, no tables, and no `CREATE TABLE` statement to write?

Redis Has No Tables — Only a Flat Keyspace

Every value Redis stores — a string, a hash, a set, a sorted set — lives under one single, flat key, in one single, shared namespace. There's no real concept of "the teams table" the way a relational database has one. The real, idiomatic substitute is entirely a matter of discipline: a consistent key-naming convention that a human reading the raw keyspace can still make sense of.

```
team:{id}                # one team's own hash
team:next_id             # a counter, standing in for AUTO_INCREMENT
season:{id}:teams        # which team IDs are in this season -- a Set
fixture:{id}             # one fixture's own hash
fixture:next_id
gameweek:{season_id}:{number}:fixtures # which fixture IDs are in this gameweek -- a Set
```

Nothing enforces this convention — it's a real, load-bearing team agreement, not a database feature. Every real finding in this chapter traces back to that one fact.

Real IDs: INCR, Not AUTO_INCREMENT

Redis has no native equivalent to SQL's `AUTO_INCREMENT` or `SERIAL`. The real, idiomatic replacement is a dedicated counter key and the atomic `INCR` command:

```
id1 = await r.incr("team:next_id")
id2 = await r.incr("team:next_id")
id3 = await r.incr("team:next_id")
# real output: 1, 2, 3
```

`INCR` is genuinely atomic — even under real concurrent requests, two calls can never both receive the same number, the same real guarantee a SQL auto-increment column provides.

A Team as a Hash

```
await r.hset(f"team:{id1}", mapping={"name": "Arsenal", "founded": 1886, "stadium": "Emirates Stadium"})
team = await r.hgetall(f"team:{id1}")
# real output: {'name': 'Arsenal', 'founded': '1886', 'stadium': 'Emirates Stadium'}
```

A hash is the closest thing Redis has to a single row — a set of named fields under one key. So far this looks like a plausible, working substitute for a `teams` table.

THE FIRST REAL COST — HSET MERGES, IT DOESN'T REPLACE

Reusing `id1` by mistake for what was meant to be a genuinely different team, calling `HSET` again on the exact same key:

```
before: {'name': 'Arsenal', 'founded': '1886', 'stadium': 'Emirates Stadium'}

await r.hset(f"team:{id1}", mapping={"name": "Not Actually Arsenal", "founded": 1999, "stadium": "Emirates Stadium"})

after:  {'name': 'Not Actually Arsenal', 'founded': '1999', 'stadium': 'Emirates Stadium'}
```

This isn't a full overwrite — it's a genuine partial **merge**. `name` and `founded` were silently replaced, but `stadium` survived untouched from the original write. There's no error, no warning, and no way to tell from the resulting hash alone that two different logical records were ever involved — the surviving `stadium` field quietly hides the fact that anything went wrong at all. A relational `INSERT` with a duplicate primary key would have failed outright with a real, loud constraint violation; this fails silently, by blending.

Season → Teams: A Set, Not a Hash

"Which teams are in this season" is genuinely different from "what are this team's own fields" — it's an unordered, deduplicated group of IDs, not a record. A Redis **Set** fits that shape directly:

```
await r.sadd(f"season:{season_id}:teams", id1, id2)
members = await r.smembers(f"season:{season_id}:teams")
# real output: {'1', '2'}
```

This is deliberately not a sorted set — nothing about season membership needs an ordering at all, so the plain, unordered Set is the honest, minimal fit. Sorted sets are saved for Chapter 7 and Chapter 8, where a real ordering (points, ranking) genuinely exists.

A Fixture as a Hash — Referencing Teams With Plain Values

```
fixture_id = await r.incr("fixture:next_id")
await r.hset(f"fixture:{fixture_id}", mapping={
    "home_team_id": id1,
    "away_team_id": 999,    # <-- this team was never created
    "gameweek": 1,
    "home_score": "",
    "away_score": "",
})
```

`home_team_id` and `away_team_id` are just plain field values — strings that happen to look like team IDs. Redis has no concept of a foreign key, so nothing checks that either one actually points at a real team.

THE SECOND REAL COST — VERIFIED: ZERO REFERENTIAL INTEGRITY

Writing that exact fixture, then checking whether `team:999` genuinely exists:

```
fixture:1 -> {'home_team_id': '1', 'away_team_id': '999', 'gameweek': '1', ...}
```

Does team:999 actually exist? NO -- genuinely does not exist
Redis accepted the fixture write anyway -- no foreign key, no error, no rejection

The write succeeds completely. This variant's own reference-implementation sibling — the real PostgreSQL-backed *Premier League Predictor: FastAPI & PostgreSQL* course — declares



insert there would raise a genuine `IntegrityError` at the database layer, before the bad row could ever be written. Here, the exact same mistake produces a real, permanently stored, genuinely broken fixture — and it stays invisible until something later tries to actually look up `team:999`.

Gameweek → Fixtures: Another Set

```
await r.sadd(f"gameweek:{season_id}:1:fixtures", fixture_id)
gw_fixtures = await r.smembers(f"gameweek:{season_id}:1:fixtures")
# real output: {'1'}
```

Which Primitive Models Which Relationship

REAL RELATIONSHIP	REDIS PRIMITIVE	WHY
A team's own fields	Hash	A named collection of fields under one key — the closest thing to a single row
Which teams are in a season	Set	Unordered, automatically deduplicated group membership — no ranking involved
Which fixtures are in a gameweek	Set	Same shape as season membership — a group, not an order
A fixture's own fields, including team references	Hash	Same as a team — but the "references" are just plain string values with no enforcement
(Chapters 7–8) The league table, the prediction leaderboard	Sorted Set	The one real relationship in this whole app that genuinely needs continuous ranking

Revisiting Chapter 1's Own Scope Note, With Real Evidence Now

Chapter 1 asked to be taken on trust that this course is "deliberately narrower" than its own PostgreSQL sibling. This chapter has now produced two real, concrete, reproduced costs of that choice rather than a promise of one: `HSET`'s own genuine partial-merge behavior can silently blend two logical records together on an accidental ID collision, and a fixture can reference a team that was never created, with the database itself offering no resistance at all. Neither failure is hypothetical — both were reproduced with real code in this chapter. Chapter 9 is where this course works through the fuller, honest picture of what that costs at scale.

Hands-On Exercises

EXERCISE 1

Run a real `HGETALL` against a team key that was never created at all -- not even accidentally, genuinely never written. Report the real return value and its real Python type, and explain why this makes the "reference to team:999" problem from this chapter even harder to catch than it already looks.

 [View solution](#)

EXERCISE 2

Run two real `SADD` calls against the same `season:1:teams` set -- the first adding two genuinely new team IDs, the second adding one team ID already in the set alongside one genuinely new one. Report `SADD`'s own real return value for each call, and explain what that return value is actually counting.

 [View solution](#)

EXERCISE 3

Explain, in your own words, why the `HSET`-merge gotcha and the missing-foreign-key gotcha are both genuinely the same underlying limitation wearing two different costumes, rather than two unrelated bugs. What single sentence about Redis explains both of them at once?

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **Key naming is the real schema** — Redis enforces nothing; a consistent naming convention is the only thing standing in for one
- **INCR replaces AUTO_INCREMENT** — atomic, real, verified sequential IDs with no native counterpart otherwise
- **Hash = record, Set = unordered membership** — teams and fixtures are hashes; season/gameweek membership is a Set
- **Verified: HSET merges, not replaces** — an accidental ID reuse can silently blend two logical records into one, with no error
- **Verified: zero referential integrity** — a fixture can reference a team ID that was never created, and Redis accepts it without complaint

CHAPTER 3: ADMIN: MANAGING TEAMS & SEASONS

Premier League Predictor: FastAPI & Redis

Chapter 3 · Admin: Managing Teams & Seasons

Chapter 2 modeled teams, fixtures, and their real membership relationships. This chapter builds the real admin routes that actually create and manage them — and finds both a genuine simplicity win for Redis and a sharper, more surprising limitation than anything Chapter 2 turned up.

The Current Season: One Key, Not a Cross-Row Clear

This project's own real PostgreSQL and SQLite siblings both had to solve the same problem: only one season is ever "current" at a time, so promoting a new one to current means clearing the flag on every other row first. In Redis, that whole problem simply doesn't exist, because there's no `is_current` field sitting on individual season records at all — there's one dedicated key holding the current season's own ID directly:

```
await r.set("season:current_id", 1)
# season:current_id -> "1"

# switching which season is current is a single write -- nothing else to touch
await r.set("season:current_id", 2)
# season:current_id -> "2"
```

A GENUINE SIMPLICITY WIN, NOT JUST A COST

There's no second row anywhere that could be left in a stale, half-updated state — there's only ever one value in the one place the app actually checks. Where a relational sibling needs the real guarantee that clearing every other row's own flag and setting the new one happens as a single atomic operation, this version's "atomicity" is a free, structural consequence of there being only one write to make in the first place. Chapter 2 found two real costs of this variant's own choices; this is the first genuine case where the same choice pays off instead.

Creating Teams: A Real Secondary Index, Not a UNIQUE Constraint

The PostgreSQL sibling can lean on a real `UNIQUE` constraint on a team's own name column to guarantee a relegated-then-promoted club is reused as the same permanent record, never accidentally duplicated. Redis hashes have no such thing — nothing stops two different team keys from both holding `name: "Arsenal"`. The real, idiomatic fix is a manually-maintained secondary index: a second hash mapping name directly to ID.

```
team_id = await r.incr("team:next_id")
await r.hset(f"team:{team_id}", mapping={"name": "Arsenal", "founded": 1886})
await r.hset("team:by_name", "Arsenal", team_id)

# a real reverse lookup, used before creating a new team
existing_id = await r.hget("team:by_name", "Arsenal")
# existing_id -> "1"

not_found = await r.hget("team:by_name", "Nonexistent FC")
# not_found -> None
```

A real create-team route checks this index first, and only creates a genuinely new hash and counter entry if nothing is found:

```
@app.post("/api/teams")
async def create_team(name: str, founded: int):
    r = app.state.redis
    existing_id = await r.hget("team:by_name", name)
    if existing_id:
        return {"team_id": int(existing_id), "reused": True}

    team_id = await r.incr("team:next_id")
    await r.hset(f"team:{team_id}", mapping={"name": name, "founded": founded})
    await r.hset("team:by_name", name, team_id)
    return {"team_id": team_id, "reused": False}
```

A REAL, ONGOING MAINTENANCE COST

This index is entirely this app's own responsibility to keep correct — nothing in Redis enforces that `team:by_name` and every `team:{id}` hash agree with each other. A route that ever updates a team's

own name directly, without also updating `team:by_name`, would leave the index silently pointing at the old name forever. A real SQL index on a `name` column would never need this kind of manual bookkeeping at all — it's maintained by the database itself, on every write, automatically.

The 20-Team Cap: SCARD, Not a CHECK Constraint

Adding a team to a season means adding its ID to that season's own membership Set — and the real 20-team limit has to be enforced by the application, since Redis Sets have no native maximum-size option:

```
@app.post("/api/seasons/{season_id}/teams/{team_id}")
async def add_team_to_season(season_id: int, team_id: int):
    r = app.state.redis
    count = await r.scard(f"season:{season_id}:teams")
    if count >= 20:
        return {"error": "season already has 20 teams"}, 400
    await r.sadd(f"season:{season_id}:teams", team_id)
    return {"added": team_id}
```

Verified against a real, freshly-populated season set:

```
season:1:teams size after adding 20: 20
App-level check: refusing to add a 21st team (count already at cap)
```

The Real Headline Finding: MULTI/EXEC Does Not Roll Back

This course's own real siblings each lean on a genuine atomicity guarantee for multi-step admin writes — SQLAlchemy's `flush()`-then-`commit()` pair, or a real `db.transaction()` wrapper — where, if anything inside fails, every earlier change in that same operation is undone as if it had never happened. Redis has its own real transaction mechanism, `MULTI` / `EXEC` — but Redis's own official documentation states its real guarantee directly:

"Redis does not support rollbacks of transactions since supporting rollbacks would have a significant impact on the simplicity and performance of Redis." A command that fails

inside a real Redis transaction doesn't undo the commands around it — **"even when a command fails, all the other commands in the queue are processed."**

VERIFIED WITH REAL CODE — AND MORE SURPRISING THAN THE DOCS ALONE SUGGEST

Setting up a real counter at `10` and a real Set (the wrong type for `INCR`), then queuing three commands in one real `redis-py` transaction — two valid increments around one command guaranteed to fail:

```
async with r.pipeline(transaction=True) as pipe:
    pipe.incr("counter:test")      # valid
    pipe.incr("myset:test")        # INVALID -- myset is a real Set, not a number
    pipe.incr("counter:test")      # valid
    results = await pipe.execute()
```

```
pipe.execute() raised: ResponseError Command #2 (INCRBY myset:test 1) of pipeline
                        caused error: ('WRONGTYPE Operation against a key
                        holding the wrong kind of value',)

counter:test AFTER the failed transaction: 12
```

The counter started at `10`. It ended at `12` — both valid `INCR` calls genuinely took effect, permanently, despite the transaction as a whole raising a real Python exception. This is the sharpest, most easily-misread part of the finding: `redis-py`'s own `pipe.execute()` genuinely raises an error, which reads exactly like the kind of exception a SQL transaction context manager raises specifically to signal "nothing was committed" — but here, raising an exception and rolling back are two completely unrelated things. The exception just reports that one command failed; it says nothing about the fate of the others, because Redis never undid them in the first place.

STACK	ON A MID-TRANSACTION FAILURE
PostgreSQL / SQLAlchemy sibling	Every change in the same flush/commit is undone — a genuine all-or-nothing guarantee
SQLite / better-sqlite3 sibling	The real db.transaction() wrapper rolls back every change on any thrown error

STACK**ON A MID-TRANSACTION FAILURE**

This course (Redis)

Verified: earlier successful commands in the same MULTI/EXEC permanently survive; only the one failing command is skipped

THE REAL, HONEST DESIGN RESPONSE

Given this, every admin route in this chapter is deliberately built around single, self-contained commands — one `HSET`, one `SADD`, one `SET` — rather than chaining several writes together inside one transaction and trusting a rollback that Redis genuinely doesn't provide. Where a real multi-step sequence is unavoidable, later chapters reach for Redis's own real `WATCH`-based optimistic locking instead, which aborts an entire transaction *before* any of it runs if a watched key changed — a genuinely different, and genuinely real, kind of safety net than rollback, covered honestly once this course actually needs it.

Hands-On Exercises

EXERCISE 1

Reproduce this chapter's own no-rollback finding with a genuinely different failing command -- queue a real `HSET` against a key that already holds a plain string (not a hash), sandwiched between two valid `SADD` calls on a real Set, inside one MULTI/EXEC transaction. Report whether the two valid `SADD` calls survive.

 [View solution](#)**EXERCISE 2**

Create a real team, then simulate a route that renames it by calling `HSET` on `team:{id}` with a new name field -- WITHOUT updating `team:by_name`. Show, with real code, that the old name still resolves via the secondary index while the new name does not, and explain what a real user-facing symptom of this bug would look like.

 [View solution](#)**EXERCISE 3**

Explain, in your own words, why a Python exception raised by `pipe.execute()` cannot be safely treated as "this transaction had no effect," using this chapter's own verified counter example as evidence. What real, different question does the exception actually answer?

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **Current season = one key** — a genuine simplicity win, no cross-row clearing needed at all
- **Team uniqueness needs a manual secondary index** — `team:by_name` stands in for a real UNIQUE constraint, and the app owns keeping it in sync
- **The 20-team cap is a manual SCARD check** — no native CHECK-constraint equivalent
- **Verified: MULTI/EXEC does not roll back** — earlier successful commands in a failed transaction permanently survive
- **An exception from `execute()` ≠ nothing happened** — it reports one command's own failure, not the fate of the whole transaction

CHAPTER 4: THE FAST FIXTURE-ENTRY UI

Premier League Predictor: FastAPI & Redis

Chapter 4 · The Fast Fixture-Entry UI

Chapter 3 closed with a promise: Redis's own real `WATCH`-based optimistic locking would get covered honestly, once this course actually needed it. Building the real fixture-entry interface — the fast, click-to-pair UI the shared spec calls for — is exactly where that need shows up, in a genuine, reproducible race condition rather than a hypothetical one.

The Real UI: 20 Buttons, No Dropdowns

Matching this project's own real client precedent — a plain vanilla-JS/HTML/CSS frontend, the same choice already made for the PostgreSQL sibling — the fixture-entry page shows all 20 of a season's own teams as always-visible buttons. Pairing two teams into a fixture is two clicks: the home team, then the away team.



A team already paired into a fixture this gameweek is shown struck through and disabled — a real, working piece of UX, but this chapter's own central finding is about exactly how far that client-side disabling can be trusted.

Modeling "Already Used" for This Gameweek

Alongside `gameweek:{season_id}:{number}:fixtures` from Chapter 2, this chapter adds one more real Set — every team ID already paired into a fixture this gameweek:

```
gameweek:{season_id}:{number}:used_teams    # a Set of team IDs
```

Checking whether a team can still be paired is a single, cheap `SISMEMBER` call — real, idiomatic Redis, and the obvious first attempt at enforcing "no team twice in a gameweek":

```
# the naive, unguarded version
async def naive_pair_team(r, gw_key, team_id):
    used = await r.sismember(gw_key, team_id)
    if used:
        return "REJECTED"
    await r.sadd(gw_key, team_id)
    return "ACCEPTED"
```

A REAL, REPRODUCED RACE — NOT A HYPOTHETICAL ONE

Firing two real, genuinely concurrent calls to `naive_pair_team` for the exact same team, against the exact same gameweek:

```
results = await asyncio.gather(
    naive_pair_team(r, gw_key, "5"), # "Fixture A"
    naive_pair_team(r, gw_key, "5"), # "Fixture B"
)
```

```
Fixture A (naive): ACCEPTED (team 5 paired)
Fixture B (naive): ACCEPTED (team 5 paired)
```

```
Real final used_teams set: {'5'} # only ONE member -- but BOTH requests were to
```

This is the sharp part: the Set itself ends up looking perfectly fine — a single, correct member. The real bug is invisible if you only inspect the Set afterward. What actually went wrong is that **two separate fixture records got created**, both genuinely believing they'd legitimately claimed team 5, because both requests read `SISMEMBER` as `False` before either one had written anything. The gap between the check and the write — a real, ordinary window, not a contrived one — is exactly where two near-simultaneous clicks on the real UI could land.

The Real Fix: WATCH, MULTI, EXEC

Redis's own real `WATCH` command monitors a key for changes between the moment it's watched and the moment a queued transaction actually runs. If the watched key changed in

that window, `EXEC` is refused entirely and the client library raises a real `WatchError` — the transaction never partially applies, it simply never runs at all.

```
from redis.exceptions import WatchError

async def guarded_pair_team(r, gw_key, team_id):
    async with r.pipeline(transaction=True) as pipe:
        while True:
            try:
                await pipe.watch(gw_key)
                used = await pipe.sismember(gw_key, team_id)
                if used:
                    await pipe.unwatch()
                    return "REJECTED"
                pipe.multi()
                pipe.sadd(gw_key, team_id)
                await pipe.execute()
                return "ACCEPTED"
            except WatchError:
                continue # loop back and RE-CHECK -- see the warning below
```

VERIFIED: THE RACE IS CLOSED

Re-running the identical two-concurrent-request scenario through the guarded version:

Fixture A (guarded): `ACCEPTED` (team 5 paired)

Fixture B (guarded): `REJECTED` (team 5 already used)

Real final used_teams set: {'5'} # exactly one member, exactly one real acceptar

Exactly one of the two requests wins now — the other is honestly, correctly told no.

A GENUINELY NON-TEXTBOOK NUANCE — WORTH GETTING RIGHT

Most real `WATCH` examples (Redis's own official documentation included) show a retry loop built around incrementing a counter — an operation where retrying and blindly re-running the exact same write is always correct, since the second attempt just adds 1 to whatever the value now is. This chapter's own real rule is a genuinely different shape: **first writer wins, every later attempt must**

be rejected, not retried into eventually succeeding. That's exactly why the `except WatchError:` `continue` branch loops back to `SISMEMBER` again rather than jumping straight to re-queuing the `SADD` — on the second pass, the check itself now correctly sees the team as used, and the function returns `"REJECTED"` honestly, instead of a naive retry-the-write loop silently creating a second, illegitimate acceptance.

Wiring This Into a Real Route

```
from fastapi.responses import JSONResponse

@app.post("/api/gameweeks/{season_id}/{number}/fixtures")
async def create_fixture(season_id: int, number: int, home_team_id: int, away_team_id: int):
    r = app.state.redis
    gw_key = f"gameweek:{season_id}:{number}:used_teams"

    home_result = await guarded_pair_team(r, gw_key, home_team_id)
    if home_result == "REJECTED":
        return JSONResponse({"error": f"team {home_team_id} already used this gameweek"})

    away_result = await guarded_pair_team(r, gw_key, away_team_id)
    if away_result == "REJECTED":
        await r.srem(gw_key, home_team_id) # give the home team back: no fixture was
        return JSONResponse({"error": f"team {away_team_id} already used this gameweek"})

    fixture_id = await r.incr("fixture:next_id")
    await r.hset(f"fixture:{fixture_id}", mapping={
        "home_team_id": home_team_id, "away_team_id": away_team_id, "gameweek": number,
    })
    await r.sadd(f"gameweek:{season_id}:{number}:fixtures", fixture_id)
    return {"fixture_id": fixture_id}
```

TWO BUGS IN AN EARLIER VERSION OF THIS ROUTE

An earlier version of this route had two flaws, both found by calling it through a real test client. **1.**

The status code was never sent. Returning a tuple like `( {...}, 400 )` is Flask's convention, not

FastAPI's. FastAPI serialized the tuple as a JSON array and answered **HTTP 200** with `[{"error":`

`...}, 400]`, so a client checking the status code saw success. `JSONResponse(..., status_code=400)`

sends the real status. **2. A rejected pairing still kept the home team.** The home team was claimed

first; if the away team was then rejected, the route returned an error but left the home team in the used set, so a later, valid fixture using that team was wrongly refused. The `srem` line gives it back. In the test, team 3 was rejected as "already used" after a failed pairing before the fix and accepted after it. One caution: the two claims are separate steps, so another request can briefly see the home team as used between the claim and the give-back. Making the pair a single atomic step is possible in Redis, but this chapter's guard works one team at a time.

The Client-Side Button State Is UX, Not Authority

The real UI disables an already-used team's own button client-side — real, immediate feedback with no round trip needed. But that's convenience only. The actual decision of whether a pairing is allowed is made exactly once, server-side, by the guarded check this chapter built — the same honest split this project's own PostgreSQL sibling already established with its own real, server-side duplicate-team guard.

STACK	HOW "NO TEAM TWICE IN A GAMEWEEK" IS ACTUALLY ENFORCED
PostgreSQL sibling	A real server-side guard query checking every existing fixture in the gameweek before insert
SQLite sibling	A real UNIQUE constraint, with a real <code>SqliteError.code</code> caught and translated into a friendly response
This course (Redis)	Verified: a WATCH-guarded SISMEMBER check, re-validated on every retry rather than assumed safe after one read

Hands-On Exercises

EXERCISE 1

Modify this chapter's own `except WatchError: continue` branch so it re-queues the SADD directly instead of looping back to re-check SISMEMBER (i.e. treat it like the textbook counter-retry pattern). Run the real two-concurrent-request scenario against this version and report what actually happens.

 [View solution](#)

EXERCISE 2

Run three genuinely concurrent calls to `guarded_pair_team` for the same team and gameweek, not just two. Report the real result for all three, and explain why WATCH-based retrying scales correctly to more than two concurrent requests without any extra code.

 [View solution](#)

EXERCISE 3

Explain, in your own words, why this chapter's own race condition could never be fixed by wrapping the naive `SISMEMBER`-then-`SADD` sequence in an ordinary `MULTI/EXEC` transaction with no `WATCH` at all. What specifically does `WATCH` add that plain `MULTI/EXEC` doesn't have?

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **20 always-visible team buttons** — click-to-pair, matching this project's own established fast-entry UI
- **Verified: a real check-then-act race** — two concurrent requests both accepted the same team into a gameweek
- **WATCH/MULTI/EXEC closes it** — verified: exactly one request wins, the other is honestly rejected
- **A genuinely non-textbook nuance** — the retry loop re-validates the condition rather than blindly retrying the write, since this is a first-writer-wins rule, not an idempotent counter
- **Client-side disabling is UX, not authority** — the same honest split already established by this project's own PostgreSQL sibling

CHAPTER 5: PREDICTIONS

Premier League Predictor: FastAPI & Redis

Chapter 5 · Predictions

Fixtures exist and can be entered quickly and safely. Now the app's real point: recording who predicted what. Every fixture can carry up to four kinds of prediction — the user's own, the BBC expert's, one or more guests', and the BBC's AI — and they need to be stored so that a later chapter can score each one against the real result. This is also the chapter where Redis has a genuine structural advantage over a relational database, and where it has three more of the costs Chapter 2 warned about.

The Model: One Hash Per Fixture, One Field Per Source

A relational design gives predictions their own table with one row per (fixture, source) pair, then needs a constraint to stop two rows for the same pair. Redis offers a shape that makes the constraint unnecessary: a single hash per fixture, with each prediction source as a *field* in it and the scoreline as the value.

```
fixture:{id}:predictions      # one hash per fixture
  user      -> "2-1"
  expert    -> "1-1"
  ai        -> "2-2"
  guest:Alan -> "3-0"      # guests are just more fields, one per guest
  guest:Bea  -> "2-0"
```

This follows Chapter 2's naming convention: the fixture's own fields stay in `fixture:{id}`, and the predictions live beside it under a related key. Writing a prediction is a single `HSET` against one field:

```
key = f"fixture:{fixture_id}:predictions"

await r.hset(key, "user", "2-1")      # returns 1
await r.hset(key, "expert", "1-1")    # returns 1
await r.hset(key, "guest:Alan", "3-0") # returns 1
```

```
await r.hset(key, "guest:Bea", "2-0")    # returns 1
await r.hset(key, "ai", "2-2")          # returns 1

await r.hgetall(key)
# {'user': '2-1', 'expert': '1-1', 'guest:Alan': '3-0', 'guest:Bea': '2-0', 'ai': '2-2'}
```

VERIFIED: "ONE PREDICTION PER SOURCE" COMES FREE

A hash cannot hold two fields with the same name, so correcting the user's prediction before kickoff needs no special handling at all:

```
await r.hset(key, "user", "3-1")    # returns 0 -- the field already existed
await r.hlen(key)                   # still 5 -- no second "user" entry, ever
```

HSET returns the number of *new* fields it created: **1** the first time a source predicts, **0** when it overwrote an existing prediction. That single number lets the route answer "created" versus "corrected" for free. Compare what the sibling stacks needed to reach the same guarantee — this is the chapter's genuine simplicity win.

STACK

HOW "ONE PREDICTION PER SOURCE PER FIXTURE" IS ENFORCED

PostgreSQL sibling	A partial unique index that deliberately excludes guest rows
Django & MySQL sibling	A GeneratedField that evaluates to NULL for guests, exploiting NULL's not-equal-to-NULL rule
Astro (SQLite) sibling	A partial unique index, plus a hand-written upsert
This course (Redis)	Nothing extra — a hash field is unique by definition, and guests differ by field name

The guest case that needed special constraint tricks in every relational sibling is a non-issue here: **guest:Alan** and **guest:Bea** are simply different field names. Averaging several guests into one comparable figure is still a Chapter 6 problem — and, as the relational siblings established, it is points that get averaged, not scorelines.

SAME-NAME GUESTS COLLIDE SILENTLY

Uniqueness by field name is only as good as the names. If two different guests are both entered as **guest:Alan**, the second **HSET** returns **0**, replaces the first person's prediction, and raises nothing.

Exercise 1 reproduces it. A relational sibling would key on a guest row's own ID; here, the key discipline has to come from the code.

Cost One: Redis Validates Nothing

A prediction is really two non-negative integers, but the hash stores an opaque string:

```
await r.hset(key, "user", "banana")    # returns 0 -- accepted without complaint
await r.hget(key, "user")              # 'banana'
```

Nothing at the Redis layer knows a score has a shape, so the only place validation can live is the FastAPI route. Pydantic does the job, and it is worth being explicit that here it is the *only* line of defence, not a second one behind a `CHECK` constraint:

```
from pydantic import BaseModel, Field

class PredictionIn(BaseModel):
    source: str = Field(pattern=r"^(user|expert|ai|guest:[A-Za-z ]{1,30})$")
    home: int = Field(ge=0, le=20)
    away: int = Field(ge=0, le=20)

# verified: {"home": 2, "away": 1} -> accepted
# verified: {"home": -1, ...} -> "Input should be greater than or equal to 0"
# verified: {"home": "x", ...} -> "Input should be a valid integer..."
```

The route validates first and only then calls `HSET` with `f"{body.home}-{body.away}"`. Anything that writes to Redis without going through the route — a script, a console session, a future feature — bypasses that validation completely.

Cost Two: A Prediction Can Target a Fixture That Doesn't Exist

This is Chapter 2's missing-foreign-key finding in a new place. The predictions hash lives under its own key, so writing to it never touches the fixture's key at all:

VERIFIED: AN ORPHAN PREDICTIONS HASH

```

await r.exists("fixture:999")           # 0 -- no such fixture
await r.hset("fixture:999:predictions", "user", "1-0") # 1 -- accepted
await r.exists("fixture:999:predictions") # 1 -- a real, permanent hash now exists
await r.exists("fixture:999")           # 0 -- the fixture still does not exist

```

A relational foreign key would have refused the row. Here the write succeeds and leaves an orphan that a scoring job in Chapter 6 might one day trip over.

Cost Three: Locking Predictions Once a Result Exists

A prediction made after the real score is known is worthless, so the route should refuse one once the fixture has a result. The natural first version reads the fixture's score, then writes:

```

async def naive_predict(r, fixture_id, source, value):
    score = await r.hget(f"fixture:{fixture_id}", "home_score")
    if score not in (None, ""):
        return "REJECTED"
    await r.hset(f"fixture:{fixture_id}:predictions", source, value)
    return "ACCEPTED"

```

That is Chapter 4's check-then-act shape again. Rather than hoping two real requests collide, this chapter forces the bad interleaving deterministically — the result gets entered in the gap between the check and the write:

VERIFIED: THE GAP IS REAL

```

check:  home_score is empty -> looks open
# --- meanwhile, the result 2-1 is entered ---
write:  HSET expert "0-0"

naive predict:  ACCEPTED

```

```
result now:      ['2', '1']
expert's stored: '0-0'      # a prediction recorded after the result was known
```

The check and the write are two separate commands, and anything can land between them.

The Fix: A Small Lua Script

Chapter 4 used `WATCH` for this class of problem. Redis offers a second tool that suits "check one key, write another, all-or-nothing": a Lua script sent with `EVAL`. Redis runs a script as a single atomic unit, so nothing can land between its lines.

```
PREDICT_LUA = """
local hs = redis.call('HGET', KEYS[1], 'home_score')
if hs and hs ~= '' then return 0 end      -- locked: a result exists
redis.call('HSET', KEYS[2], ARGV[1], ARGV[2])
return 1                                  -- written
"""

predict = r.register_script(PREDICT_LUA)

await predict(keys=[f"fixture:{fid}", f"fixture:{fid}:predictions"], args=["expert", "4
```

VERIFIED: ATOMIC, AND IT LOCKS CORRECTLY

```
no result yet:      returns 1  expert stored: '4-4'
result 2-1 entered: returns 0  expert stored: '4-4'  # the late write '9-9' was
```

The script's read of `home_score` and its write to the predictions hash cannot be separated. `register_script` handles loading the script once and calling it by its hash afterwards.

THE SCRIPT INHERITS COST TWO

Run against a fixture that was never created, the script reads `home_score` as `nil`, decides the fixture is open, and happily writes an orphan predictions hash — verified: it returned `1` and the

hash then existed. Locking and existence are separate checks, and the script only had the first. Exercise 2 adds the second inside the same atomic script.

WATCH OR LUA?

WATCH (Chapter 4) lets the client do arbitrary logic between reading and writing, at the price of a retry loop that must re-validate on every pass. A Lua script has no retries, because it is never interrupted — but the logic must be written in Lua and run inside Redis. For a small, fixed rule like this one, the script is the tidier fit; for logic that needs Python's own libraries, **WATCH** is.

Reading Predictions Back

One fixture's predictions come back in a single **HGETALL**. Scoring a whole gameweek needs the predictions for all its fixtures — the IDs come from Chapter 2's `gameweek:{season_id}:{number}:fixtures` Set, and each needs its own **HGETALL**. Sent one at a time, that is one network round trip per fixture; a pipeline sends them together:

```
async with r.pipeline(transaction=False) as pipe:
    for fid in fixture_ids:
        pipe.hgetall(f"fixture:{fid}:predictions")
    all_predictions = await pipe.execute()
```

MEASURED: 10 FIXTURES, SAME RESULTS

Fetching the predictions for 10 fixtures, median of 30 runs each: **5.95 ms** one call at a time versus **0.93 ms** pipelined — about **6.4×** faster, with identical results. Treat the exact figures as specific to this machine (Redis in a Docker container on Windows, over loopback); the shape of the result — fewer round trips, less waiting — is the point, and it grows with real network latency.

`transaction=False` is deliberate: this is a read-only batch, and there is no need to wrap it in **MULTI** / **EXEC**.

THE REVERSE QUERY HAS NO INDEX

"Give me every prediction the expert made this season" is easy in SQL and awkward here. There is no index on the field name, so the options are a pipelined **HGET** per fixture (fine when the fixture IDs

are already known, as they are for a gameweek) or a `SCAN` across every `fixture*:predictions` key, which touches the whole keyspace. Exercise 3 works through both.

Hands-On Exercises

EXERCISE 1

Enter two different guests who happen to share the same first name as `guest:Alan`, one after the other. Report each HSET return value and what HGETALL and HLEN show afterwards, then propose a key scheme that avoids the collision.

 [View solution](#)

EXERCISE 2

Extend the Lua script so it refuses to write when the fixture itself does not exist, returning -1 for that case (0 stays "locked", 1 stays "written"). Run it against a nonexistent fixture, an open fixture, and a locked fixture, and report all three results.

 [View solution](#)

EXERCISE 3

Fetch the expert's prediction across five fixtures using a pipelined HGET, then list the same fixtures' prediction keys using SCAN. Explain why SCAN returns more keys than the five you asked about, and when each approach is the right one.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- **One hash per fixture, one field per source** — `fixture:{id}:predictions`, guests as `guest:<name>` fields
- **Verified: uniqueness is free** — a hash field can't repeat, so no index or GeneratedField is needed; HSET returns 1 for a new prediction and 0 for a correction

- **Verified: Redis validates nothing** — "banana" was stored; Pydantic in the route is the only defence
- **Verified: orphan predictions** — writing to a nonexistent fixture's predictions hash succeeds
- **Verified: check-then-write gap** — a result entered between the check and the write let a late prediction through; a Lua script closes it atomically
- **Pipelining** — ~6.4× faster for 10 HGETALLs on this machine; no index exists for "all predictions by one source"

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

Premier League Predictor: FastAPI & Redis

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

Predictions are stored and locked. This chapter turns a real scoreline into points: entering a fixture's result, scoring every prediction against it, and keeping running totals that the later chapters will rank. The scoring rule itself is simple. The interesting part is what "keep a running total" means in Redis, because the obvious way to do it is wrong the moment a result is corrected.

The Scoring Rule

The point values were confirmed with the user during the PostgreSQL sibling's own scoring chapter and are reused here unchanged: **40 points for a correct score, 10 points for a correct result** (the right win, draw or loss with the wrong scoreline), and 0 otherwise. Scoring is a pure function with no Redis in it at all:

```
def sign(a, b):
    return (a > b) - (a < b)          # 1 home win, 0 draw, -1 away win

def score_prediction(pred, actual):
    if pred == actual:                # exact match FIRST
        return 40
    if sign(*pred) == sign(*actual):  # then the outcome
        return 10
    return 0
```

The order of the two checks matters, because an exact score always implies a correct outcome. Checking the outcome first would score every exact prediction as 10. Exercise 2 runs both orders side by side. Verified on six cases, including two draws:

```
predicted 2-1, actual 2-1  -> 40
predicted 1-0, actual 2-1  -> 10  # right result, wrong score
predicted 0-0, actual 2-1  -> 0
predicted 2-2, actual 1-1  -> 10  # a draw is a result too
```

```

predicted 0-0, actual 0-0  -> 40
predicted 1-2, actual 2-1  -> 0

```

Guests: Average the Points, Not the Scorelines

Chapter 5 stored each guest as their own field. A week's guests have to become one comparable figure, and the relational siblings settled how: score each guest first, then average the *points*. A scoreline can't be meaningfully averaged (half a goal isn't a result), but a point value always can.

```

def parse(s):
    h, a = s.split("-")
    return int(h), int(a)

def compute_points(predictions, actual):
    out, guests = {}, []
    for source, value in predictions.items():
        pts = score_prediction(parse(value), actual)
        if source.startswith("guest:"):
            guests.append(pts)
        else:
            out[source] = pts
    if guests:
        out["guest_avg_x100"] = round(sum(guests) / len(guests) * 100)
    return out

```

The field is called `guest_avg_x100` on purpose — the reason comes later in this chapter. Worked example, actual result 2-1: Alan predicted 3-0 (10), Bea 2-1 (40), Carl 1-1 (0). The average is $50 / 3 = 16.67$, stored as `1667`. Verified output for the full set of sources:

```
{'user': 40, 'expert': 10, 'ai': 0, 'guest_avg_x100': 1667}
```

Entering the Result — Without Creating an Orphan

Chapter 5 locked predictions once a fixture has scores, so entering the result is also what closes them. The natural write is `HSET fixture:{id} home_score ... away_score ...` — and that reproduces Chapter 2's missing-foreign-key problem in a new place:

VERIFIED: A RESULT FOR A FIXTURE THAT DOESN'T EXIST

```
await r.hset("fixture:999", mapping={"home_score": 1, "away_score": 0})
await r.hgetall("fixture:999")      # {'home_score': '1', 'away_score': '0'}
await r.exists("fixture:999")      # 1 -- a fixture with scores and no teams
```

HSET creates the key if it isn't there. A "fixture" now exists that has no home team, no away team and no gameweek — and every later existence check will say it is real.

The fix is the same shape as Chapter 5's: put the existence check inside a small Lua script, so the check and the write can't be separated.

```
ENTER_RESULT_LUA = """
if redis.call('EXISTS', KEYS[1]) == 0 then return -1 end
redis.call('HSET', KEYS[1], 'home_score', ARGV[1], 'away_score', ARGV[2])
return 1
"""
```

Verified: entering a result for `fixture:999` returns `-1`, and afterwards neither `fixture:999` nor `fixture:999:points` exists. The route maps `-1` to a 404. As in Chapter 5, the home and away scores are validated by Pydantic (`ge=0`) before anything reaches Redis.

Running Totals: The Obvious Way Is Wrong

Chapter 8's prediction leaderboard needs each source's total across the season. The obvious Redis idiom is an increment: every time a fixture is scored, `HINCRBY totals user 40`.

VERIFIED: INCREMENTS ARE NOT IDEMPOTENT

Two ordinary events break it. First, the same result submitted twice (a double-clicked button, a retried request):

```
HINCRBY naive user 40  # first submission
HINCRBY naive user 40  # the identical one again
naive totals: {'user': '80'}  # the user only earned 40
```

Second, a corrected result. Score the fixture as 2-1, then correct it to 1-1, adding each set of points as it is computed:

```
true totals after the correction: user 0, expert 0, ai 10, guest average 13.33
naive totals: {'user': '40', 'expert': '10', 'ai': '10', 'guest_avg': '30'}
```

The 2-1 points were never taken back, so the correction only *added* to them. A relational sibling never has this problem because its totals are a `SUM` over stored rows, computed fresh each time. A stored counter has no memory of what it already counted.

The Fix: Store Points Per Fixture, Apply Only the Difference

Keep each fixture's points in their own hash, `fixture:{id}:points`, and update the running totals by *new minus old*. Re-scoring the same result then adds zero, and a correction adds exactly the change. Reading the old value and writing the new one has to be atomic — two concurrent corrections would otherwise both read the same "old" — so it goes in a Lua script, as in Chapter 5:

```
APPLY_POINTS_LUA = """
for i = 1, #ARGV, 2 do
  local old = tonumber(redis.call('HGET', KEYS[1], ARGV[i]) or '0')
  local new = tonumber(ARGV[i+1])
  redis.call('HSET', KEYS[1], ARGV[i], ARGV[i+1])      -- this fixture's points
  redis.call('HINCRBY', KEYS[2], ARGV[i], new - old)    -- season total moves by the
end
return 1
"""
```

The whole scoring route is then three steps: enter the result (Lua), read the predictions (`HGETALL`), compute and apply the points (Lua).

```

async def score_fixture(r, fixture_id, home, away):
    rc = await enter_result(keys=[f"fixture:{fixture_id}"], args=[home, away])
    if rc == -1:
        return None # route returns 404
    predictions = await r.hgetall(f"fixture:{fixture_id}:predictions")
    points = compute_points(predictions, (home, away))
    args = [x for k, v in points.items() for x in (k, str(v))]
    await apply_points(keys=[f"fixture:{fixture_id}:points", "totals"], args=args)
    return points

```

VERIFIED: IDEMPOTENT, AND CORRECTIONS LAND EXACTLY

```

enter 2-1:      totals {'user': '40', 'expert': '10', 'ai': '0', 'guest_avg_x100': '13.33'}
enter 2-1 again: totals unchanged
correct to 1-1: totals {'user': '0', 'expert': '0', 'ai': '10', 'guest_avg_x100': '13.33'}

```

The corrected totals match the true figures from the previous section exactly, and re-submitting an identical result changes nothing. Scoring a result is now something that can safely be repeated.

Why the Guest Average Is Stored as an Integer

The field is `guest_avg_x100` — hundredths of a point — rather than `16.67`. Redis has `HINCRBYFLOAT`, which looks like the natural fit for a fractional average. Applying the same sequence of ten corrections both ways:

VERIFIED: FLOAT TOTALS PICK UP NOISE

```

HINCRBYFLOAT total: '13.3299999999999984' # this fixture's own points: '13.33'
HINCRBY (x100):      '1333' # exact, every time

```

The noise appeared after a single correction, not only after many. The stored total no longer equals the sum of the per-fixture values it is supposed to be built from. Integer hundredths keep every

`HINCRBY` exact; divide by 100 only when displaying. This will matter in Chapter 8, where the guest column is ranked.

A Crash Between the Steps

Entering the result and applying the points are two separate Redis calls. If the server dies between them, the fixture has a result and no points:

```
# result written, then the process "crashes" before scoring
result: ['1', '0']    points: {}                                # a scored-looking fixture with nothing

# re-running the same route recovers it
score_fixture(r, 2, 1, 0) -> {'user': 40}    points: {'user': '40'}
```

Nothing was left half-applied, because scoring is idempotent — running it again is the whole recovery procedure. That is a direct dividend of the delta design above; the naive increment version would have needed to know whether the first attempt had already counted.

WHAT CHAPTER 7 AND 8 BUILD ON

`totals` holds each source's season total as a hash. A hash is enough to store the numbers, but not to *rank* them cheaply — that is the job of the sorted sets in Chapters 7 and 8. The delta approach carries straight across: `ZINCRBY` takes the same "difference, not total" argument that `HINCRBY` did here.

Hands-On Exercises

EXERCISE 1

Submit the identical 40-point score for the same source twice, once through a plain `HINCRBY` total and once through the `APPLY_POINTS` Lua script. Report both totals and explain what each design remembers that the other doesn't.

 [View solution](#)

EXERCISE 2

Write a second version of `score_prediction` that checks the outcome before the exact score. Run both versions on five cases (including two draws) and report which cases differ and why.

 [View solution](#)

EXERCISE 3

Enter a result for a fixture that was never created using a plain HSET, then explain why a later chapter's existence checks would treat the result as a real fixture, and what the Lua-guarded version does instead.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **40 for a correct score, 10 for a correct result** — a pure function; check the exact score *before* the outcome
- **Guests: average points, not scorelines** — stored as integer hundredths (`guest_avg_x100`)
- **Verified: HSET on a missing fixture creates one** — guard result entry with an in-script `EXISTS`
- **Verified: increment totals aren't idempotent** — a repeated or corrected result double-counts
- **Store per-fixture points, apply new-minus-old** — one Lua script, atomic, safe to re-run
- **Verified: HINCRBYFLOAT drifts** — `'13.3299999999999984'` after one correction; integers stay exact
- **Crash recovery is just re-running** — idempotent scoring needs no bookkeeping about what was applied

CHAPTER 7: THE REAL LEAGUE TABLE VIA REDIS SORTED SETS

Premier League Predictor: FastAPI & Redis

Chapter 7 · The Real League Table via Redis Sorted Sets

This is the chapter Redis was chosen for. A league table is a ranked list that changes whenever a result comes in, and a Redis **sorted set** keeps a ranked list ordered as it changes. Every relational sibling instead computed the table with a query each time it was read. Here the table is *maintained*: each result updates it as it is entered, and reading it is a single range call. The chapter's real work is making that maintenance correct — a sorted set gives each member exactly one number to sort by, and a league table sorts by three things.

Sorted Set Basics

A sorted set holds unique members, each with a numeric *score*, and always keeps them ordered by score. Where two members have the same score, Redis orders them by the member string itself, lexicographically. That last rule is quiet and consequential, and the next two sections are about it.

The Naive Table Ranks by Points Only

The direct translation is a set with points as the score. Two teams each win once — team 1 by 3-0, team 2 by 1-0 — so both have 3 points, but team 1's goal difference is better:

VERIFIED: TIES FALL BACK TO MEMBER NAME — AND REVERSED

```
await r.zincrby("naive", 3, "1")      # team 1: won 3-0
await r.zincrby("naive", 3, "2")      # team 2: won 1-0
await r.zrevrange("naive", 0, -1, withscores=True)
# [('2', 3.0), ('1', 3.0)] -- team 2 above team 1
```

The scores are equal, so Redis fell back to comparing the member strings, and `ZREVRANGE` reverses that too. Team 2 is listed above team 1 even though team 1 has the better goal difference. Nothing errors; the table is just wrong.

A second trap hides in the same rule. Members are strings, so if the members are plain team IDs, ties order `"10"` before `"2"`:

```
members 1, 2, 3, 10, all with the same score:
plain ids:  ['1', '10', '2', '3']
padded ids: ['001', '002', '003', '010']
```

Padding every team ID to three digits (`"{id:03d}"`) makes the string order match the numeric order. The member is only a label; the team's data is still under `team:{id}`.

One Score, Three Sort Keys

A Premier League table sorts by points, then goal difference, then goals scored. A sorted set has one score per member, so the three values have to be packed into a single number, most important first:

```
composite = points * 1_000_000 + (goal_difference + 500) * 1_000 + goals_for

score      = -composite          # negated -- see below
```

Adding 500 keeps the goal-difference part non-negative. Points occupy the millions, goal difference the thousands, goals scored the units, so comparing composites compares them in exactly the right priority. The score is then **negated**: with the best team's score the smallest, `ZRANGE` (ascending) lists the table from first place down, and ties that survive all three keys resolve to ascending member order — the padded IDs from the previous section — instead of the reversed order that `ZREVRANGE` gives.

THE PACKING HAS LIMITS — VERIFIED

The scheme assumes goal difference stays within ± 499 and goals scored below 1,000. Outside that, one field spills into the next and two different teams get the same number:

```

A: 3 points, goal difference +500    -> 4000000
B: 4 points, goal difference -500    -> 4000000      # identical

A: 3 points, goal difference 0,  1000 goals scored -> 3501000
B: 3 points, goal difference +1,  0 goals scored   -> 3501000      # identical

```

A real season is nowhere near those figures — the largest composite in this chapter's scale is about 1.1×10^8 , comfortably inside the 2^{53} range where a double stores integers exactly. But the limits are real and belong in a code comment. Exercise 2 works through them.

Per-Team Stats, and What a Result Changes

The composite covers points, goal difference and goals scored, but the table also shows played/won/drawn/lost. Those live in one hash per team, `team:{id}:stats`, with fields `played`, `won`, `drawn`, `lost`, `gf`, `ga`. Entering a result has to update both teams' stats and both teams' sorted-set scores.

And it has to survive being repeated or corrected — Chapter 6's lesson, applied again. A result can be entered twice or fixed later, and simply adding a result's effect every time would double-count it. The same remedy works: remember what was last applied to this fixture (`fixture:{id}:applied`), reverse that, then apply the new result. One Lua script does all of it atomically:

```

APPLY_RESULT_LUA = ""
local function bump(stats, gf, ga, s)          -- s = +1 to apply, -1 to reverse
    redis.call('HINCRBY', stats, 'played', s)
    redis.call('HINCRBY', stats, 'gf', s*gf)
    redis.call('HINCRBY', stats, 'ga', s*ga)
    if gf > ga then redis.call('HINCRBY', stats, 'won', s)
    elseif gf == ga then redis.call('HINCRBY', stats, 'drawn', s)
    else redis.call('HINCRBY', stats, 'lost', s) end
end

local function rescore(stats, member)
    local won   = tonumber(redis.call('HGET', stats, 'won')) or '0'
    local drawn = tonumber(redis.call('HGET', stats, 'drawn')) or '0'
    local gf    = tonumber(redis.call('HGET', stats, 'gf')) or '0'

```

```

local ga    = tonumber(redis.call('HGET', stats, 'ga')    or '0')
local composite = (3*won + drawn) * 1000000 + (gf - ga + 500) * 1000 + gf
redis.call('ZADD', KEYS[3], -composite, member)
end

local hs, as = tonumber(ARGV[3]), tonumber(ARGV[4])
if redis.call('EXISTS', KEYS[2]) == 1 then      -- something was applied before:
    local oh = tonumber(redis.call('HGET', KEYS[2], 'h'))
    local oa = tonumber(redis.call('HGET', KEYS[2], 'a'))
    bump(KEYS[4], oh, oa, -1)
    bump(KEYS[5], oa, oh, -1)
end
bump(KEYS[4], hs, as, 1)
bump(KEYS[5], as, hs, 1)
redis.call('HSET', KEYS[2], 'h', hs, 'a', as)
redis.call('HSET', KEYS[1], 'home_score', hs, 'away_score', as)
rescore(KEYS[4], ARGV[1])
rescore(KEYS[5], ARGV[2])
return 1
"""

```

The five keys are passed in: the fixture, its `:applied` hash, the table's sorted set, and the home and away stats hashes. The members are the padded IDs (`ARGV[1]`, `ARGV[2]`). The route looks up the two team IDs from the fixture hash first — a missing fixture returns 404 there, before the script runs, so the existence guard from Chapter 6 still holds. Because this script also writes `home_score` and `away_score`, it replaces Chapter 6's separate result-entry script; the prediction-scoring script stays as it was.

A new season starts by giving every team the zero-played score, `-500000` (0 points, goal difference 0, 0 goals scored). All teams then tie and list in padded-ID order, which is a sensible empty table.

KEYS BUILT FROM THE DATA

The script receives every key it touches through `KEYS`, which is the form Redis expects. A script that constructed key names from its arguments would work on a single server but break on a Redis Cluster, where the keys must be declared up front so Redis can route them together. This course uses one server, but passing the keys in costs nothing.

Verified Against an Independent Calculation

A hand-picked example proves little for code like this, so the check is a brute-force one. A ten-team double round robin — 90 fixtures — was given random results (0 to 4 goals each side). Then 40 of them were corrected to different random results, and 5 were re-entered with the same result they already had. The table was then compared with one computed from scratch in plain Python from the final results:

VERIFIED: REDIS TABLE EQUALS THE FROM-SCRATCH TABLE

```
order matches independent calculation:  True
all played/won/drawn/lost/gf/ga match:  True
bottom three (ZRANGE table -3 -1):      ['006', '002', '007'] # same as the inde
top three (team, pts, GD, GF):            (8, 40, 21, 42) (3, 40, 15, 47) (5, 35,
```

The top two teams are level on 40 points and separated by goal difference, so the tiebreak is exercised. No two teams tied on all three keys in this run, so the member-order fallback was not tested here; Exercise 1 does test it.

That bottom-three line is a preview: Chapter 9's relegation is a single `ZRANGE table -3 -1`.

A Field That Isn't There, and One That Is Zero

`HINCRBY` creates a field on first use, and reversing a correction leaves it at `0` rather than deleting it. Entering a 2-0 result and correcting it to 0-1 produced the same numbers as entering 0-1 fresh — but not the same hash:

```
corrected: {'played': '1', 'gf': '0', 'ga': '1', 'won': '0', 'lost': '1'}
fresh 0-1: {'played': '1', 'gf': '0', 'ga': '1', 'lost': '1'} # no 'won' field at
```

The sorted-set scores and order were identical in both runs. A missing field and a zero mean the same thing here, so any code reading a stats hash must default to `0` (`int(stats.get("won", 0))`) rather than assume every field is present.

Reading the Table

The read path is two steps: one range call for the order, then one pipelined batch for the stats — Chapter 5's pipelining, again:

```
async def get_table(r, table_key):
    members = await r.zrange(table_key, 0, -1)          # best first
    async with r.pipeline(transaction=False) as pipe:
        for m in members:
            pipe.hgetall(f"team:{int(m)}:stats")
        stats = await pipe.execute()
    rows = []
    for position, (m, s) in enumerate(zip(members, stats), start=1):
        g = lambda k: int(s.get(k, 0))                  # missing field == 0
        rows.append({"pos": position, "team_id": int(m),
                     "played": g("played"), "won": g("won"), "drawn": g("drawn"), "lost": g("lost"),
                     "gd": g("gf") - g("ga"), "points": 3 * g("won") + g("drawn")})
    return rows
```

Points and goal difference are recomputed from the stats hash rather than decoded out of the score. The score exists only to sort; the hash is the record.

The Table Is a Derived Index — So It Can Be Rebuilt

Maintaining the table incrementally has a cost: it can drift from the fixtures if anything ever writes a result without going through the script. The remedy follows from the design.

Everything in the table can be recomputed from the fixtures' stored results, and the script is idempotent, so a rebuild is: delete the table and stats and applied keys, reset every team to `-500000`, replay every fixture that has a result.

VERIFIED: DELETE AND REPLAY GIVES THE SAME TABLE

After deleting the sorted set, all ten stats hashes and all 90 `:applied` hashes, then replaying the 90 stored results through the same script, the order matched the original exactly. The table is safe to throw away, which is what makes incremental maintenance an acceptable choice instead of a risky one.

STACK	HOW THE TABLE IS PRODUCED	COST
PostgreSQL / SQLite siblings	A <code>UNION ALL</code> query over every fixture, run on each read	Always correct; work grows with fixtures on every read
Django & MySQL sibling	A hand-written <code>UNION ALL</code> through <code>.raw()</code>	Same as above
This course (Redis)	Maintained at write time in a sorted set + stats hashes	Read is one range call plus one pipeline; every write path must go through the script

For 380 fixtures and 20 teams the read-time query is not slow, so nothing here is forced by performance — this variant exists to show the idiomatic Redis approach honestly, including the discipline it requires. Head-to-head and fair-play tiebreakers are not implemented, same as the relational siblings.

Hands-On Exercises

EXERCISE 1

Build three small cases: two teams level on points and goal difference but different on goals scored; two teams level on all three keys; and two teams level on points but different on goal difference. Report the real table order for each and say which key or fallback decided it.

 [View solution](#)

EXERCISE 2

Compute the composite score by hand for two teams that collide when goal difference reaches +500 (3 points versus 4 points at -500) and when goals scored reaches 1,000. Report the real numbers, then propose a wider packing that removes both collisions and state what its new limits are.

 [View solution](#)

EXERCISE 3

Enter a 2-0 result, then correct it to 0-1. Compare the resulting table and stats against entering 0-1 on a fresh table, and report exactly what matches and what differs.

[View solution](#)

CHAPTER 7 QUICK REFERENCE

- **A sorted set has one score per member** — a three-key table sort must be packed into one number
- **Verified: ties fall back to member string** — and `ZREVRANGE` reverses it, putting the worse team first
- **Pad the IDs** — `"10"` sorts before `"2"`; `"{id:03d}"` fixes it
- **composite = points $\times 10^6$ + (GD+500) $\times 10^3$ + GF, stored negated** — `ZRANGE` then lists first place down
- **Verified: the packing collides past ± 499 GD or 1,000 GF** — safe for a real season, worth a comment
- **Reverse the last-applied result, then apply the new one** — one atomic Lua script, safe to repeat or correct
- **Verified: matches an independent calculation** — 90 fixtures, 40 corrections, 5 repeats, all stats and order equal
- **Default missing hash fields to 0** — a reversed correction leaves `'won': '0'`, a fresh entry has no field
- **The table is derived** — delete and replay rebuilds the same order

CHAPTER 8: THE PREDICTION LEADERBOARD

Premier League Predictor: FastAPI & Redis

Chapter 8 · The Prediction Leaderboard

Chapter 7 ranked the twenty clubs. This chapter ranks the four *predictors* — the user, the BBC expert, the BBC's AI and the week's guests — by how many points their predictions have earned across the season. It is a much smaller sorted set, four members, but it is where the shortcuts taken in Chapter 6 get tested, and one of them turns out to be wrong.

What Gets Ranked

Chapter 6 left two hashes behind. `totals` holds each source's season points (`user`, `expert`, `ai`, `guest_avg_x100`), and each fixture's own points sit in `fixture:{id}:points`. The leaderboard also shows how many *correct scores* (40-point predictions) and *correct results* (10-point predictions) each source has made, and those counts don't exist yet. Guests get no counts: their figure is an average of several people's points, so "how many exact scores" has no meaning for it, matching the relational siblings.

Mixed Units: The Problem Hiding in Chapter 6

The obvious leaderboard is `ZADD leaderboard <total> <source>` using the numbers already in `totals`. But Chapter 6 stored three of them in whole points and the guest figure in *hundredths* of a point, to keep `HINCRBY` exact.

VERIFIED: THE GUEST AVERAGE OUTRANKS EVERYONE

Using totals from this chapter's test season — user 180, expert 340, AI 320 points, and a guest average of 326.65 points stored as `32665`:

```
await r.zadd("mixed", {"user": 180, "expert": 340, "ai": 320, "guest": 32665})
await r.zrevrange("mixed", 0, -1)
# ['guest', 'expert', 'ai', 'user'] -- guest "wins" by a factor of about 100
```

32665 is a bigger number than 340, so the guest tops the table. Redis has no idea that one column is in hundredths, so nothing complains. The correct order has the expert first.

The fix is a single common unit. Everything is ranked in hundredths of a point: the guest value as it is, and the other three multiplied by 100. Fixed, the same data ranks `expert 34000, guest 32665, ai 32000, user 18000`.

The Score: Points, Then Correct Scores

Ties are decided by who has more correct scores. As in Chapter 7, two numbers become one, most important first, and the result is negated so that `ZRANGE` lists first place down:

```
composite = total_in_hundredths * 1000 + correct_scores    # correct_scores stays below
score      = -composite
```

A season has at most 380 fixtures, so the correct-score count can't reach 1,000 and can't carry into the points. The members are just `user`, `expert`, `ai` and `guest`. Anything still tied after both keys falls back to member-name order, ascending — the same arbitrary-but-deterministic fallback as Chapter 7.

One Script Keeps Everything Consistent

Chapter 6's script already did the hard part — new-minus-old, so a repeated or corrected result never double-counts. This version extends it to also maintain the correct-score and correct-result counts (with the same deltas), rescore the leaderboard member, and announce a change:

```
APPLY_POINTS_V2_LUA = ""
local changed = false
for i = 2, #ARGV, 2 do
    local src = ARGV[i]
    local new = tonumber(ARGV[i+1])
    local old = tonumber(redis.call('HGET', KEYS[1], src) or '0')
    redis.call('HSET', KEYS[1], src, ARGV[i+1])    -- this fixture's points
    if new ~= old then
        changed = true
    end
end
```

```

redis.call('HINCRBY', KEYS[2], src, new - old)      -- season total moves by the
if src ~= 'guest_avg_x100' then
    redis.call('HINCRBY', KEYS[4], src .. ':scores', (new == 40 and 1 or 0) - (old == 40 and 1 or 0))
    redis.call('HINCRBY', KEYS[4], src .. ':results', (new == 10 and 1 or 0) - (old == 10 and 1 or 0))
end
end
local total = tonumber(redis.call('HGET', KEYS[2], src) or '0')
local scores = tonumber(redis.call('HGET', KEYS[4], src .. ':scores') or '0')
local x100, member = total * 100, src
if src == 'guest_avg_x100' then x100, member = total, 'guest' end
redis.call('ZADD', KEYS[3], -(x100 * 1000 + scores), member)
end
if changed then redis.call('PUBLISH', ARGV[1], 'updated') end
return changed and 1 or 0
"""

```

The keys are the fixture's points hash, `totals`, the `leaderboard` sorted set and a `records` hash (fields like `user:scores`). `ARGV[1]` is the notification channel; the rest are *source, points* pairs, exactly as Chapter 6 built them. A new season seeds each of the four members at score 0. The route passes the same arguments as before plus the channel name — nothing else in the scoring route changes.

Same method as Chapter 7. Sixty fixtures, each with predictions from the user, expert and AI plus one to three guests, all random; a random result for every fixture; then 30 random corrections. The totals, counts and order were then recomputed from scratch in plain Python from the final results:

VERIFIED: LEADERBOARD, TOTALS AND COUNTS ALL MATCH

```

totals equal the independent calculation:  True  {'ai': 320, 'expert': 340, 'guest': 360}
correct-score/result counts equal:        True
leaderboard order equals:                 True  expert, guest, ai, user

```

This checks the Redis logic — the deltas, the counts, the ranking. It does not check whether the guest average itself is accurate; that is the rounding section below.

Note the guest row's correct-score count is never used, which is why its composite is just

```
total * 1000
```

Real-Time Updates With Pub/Sub

The script ends with `PUBLISH`, and only when something actually changed. A browser or another process that subscribes to that channel gets a nudge the moment a result is scored. Verified:

```
re-scoring the same result:      script returns 0, no message
scoring a corrected result:      script returns 1, message ('message', 'updated')
subscriber that connects afterwards: nothing at all
```

Repeating an identical result stays silent, which comes free from the "only when changed" test. The last line is Pub/Sub's real limitation: it keeps no history. A message is delivered to whoever is subscribed right then and forgotten. So the message can't carry the data — it is only a signal to read again. A client should read the leaderboard when it connects, then subscribe, and re-read on each nudge. (Subscribing first and then reading avoids missing a change in the gap; a re-read is cheap with four members.)

Revisiting Chapter 6: The Guest Average Has a Rounding Error

Chapter 6 rounds each fixture's guest average to hundredths and sums those. That round-then-sum is not the same as summing exactly, and the difference accumulates with the number of fixtures. Averaging three guests who scored 10, 40 and 0 gives 16.6667, stored as `1667` — 0.0033 too high, once per such fixture. The worst case is half a hundredth per fixture: $0.005 \times 380 = 1.9$ points over a season. What actually happens was measured by simulating 2,000 seasons of 380 fixtures with one to four random guests:

MEASURED: SMALL, BUT NOT ZERO

Across the 2,000 simulated seasons, the round-then-sum guest total differed from the exact total by an average of **0.07 points** and a maximum of **0.16 points**. That is tiny against totals in the hundreds — but this is a leaderboard, and two sources within a fraction of a point of each other could be ranked the wrong way round by it.

There is an exact alternative. Every prediction is worth 0, 10 or 40 points, so a guest total is a multiple of 10, and $10 \times 60 = 600$ divides evenly by 1, 2, 3, 4, 5 and 6. Storing the average in *sixtieths* of a point makes it an exact integer for any fixture with up to six guests — checked here across 200 random cases for each guest count from one to six. Seven guests is the first count that fails (10/7 is not a whole number of sixtieths). Exercise 3 works it through.

Reading the Leaderboard

```
async def get_leaderboard(r):
    members = await r.zrange("leaderboard", 0, -1)    # best first
    async with r.pipeline(transaction=False) as pipe:
        pipe.hgetall("totals")
        pipe.hgetall("records")
        totals, records = await pipe.execute()
    rows = []
    for pos, m in enumerate(members, start=1):
        if m == "guest":
            rows.append({"pos": pos, "source": m,
                        "points": int(totals.get("guest_avg_x100", 0)) / 100,
                        "correct_scores": None, "correct_results": None})
        else:
            rows.append({"pos": pos, "source": m,
                        "points": int(totals.get(m, 0)),
                        "correct_scores": int(records.get(f"{m}:scores", 0)),
                        "correct_results": int(records.get(f"{m}:results", 0))})
    return rows
```

Missing fields default to zero, as in Chapter 7. The guest's counts are `None`, so the client renders a dash rather than a misleading zero.

STACK	HOW THE PREDICTION TABLE IS PRODUCED
PostgreSQL / Django / SQLite siblings	A two-pass query on each read: average guest points per fixture, then sum every source across the season
This course (Redis)	Maintained at write time; the read is a range call and two hash reads. The guest's correct-score count is <code>None</code> in both

Hands-On Exercises

EXERCISE 1

Reproduce the mixed-unit bug with the chapter's season totals (user 180, expert 340, AI 320, guest 32665), report the order, then convert everything to hundredths and report the corrected order and scores.

 [View solution](#)

EXERCISE 2

Use the script to make the user and the expert finish on the same points but with different correct-score counts, then make three sources tie on both keys. Report the leaderboard for each case and say what decided it.

 [View solution](#)

EXERCISE 3

Compute the guest average in hundredths and in sixtieths for guest sets scoring [10,40,0], [40,10,10,0,0] and seven guests [10,0,0,0,0,0,0]. Report the rounding error of each, and explain why the sixtieths scheme stops working at seven guests.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **Rank in one unit** — verified: mixing whole points with the guest's hundredths put the guest first by a factor of ~ 100
- **composite = total in hundredths $\times 1000$ + correct scores, negated** — guest total is already in hundredths and has no counts
- **Extend the delta script** — totals, correct-score/result counts and the ZADD in one atomic call
- **Verified against an independent calculation** — 60 fixtures, 30 corrections: totals, counts and order equal
- **Pub/Sub is a nudge, not a store** — no message on an unchanged result; a late subscriber sees nothing, so read on connect
- **Chapter 6's rounding accumulates** — measured mean 0.07, max 0.16 points per season; sixtieths are exact for up to six guests

CHAPTER 9: PROMOTION & RELEGATION, AND WHAT REDIS ALONE CAN'T MODEL CLEANLY

Premier League Predictor: FastAPI & Redis

Chapter 9 · Promotion & Relegation, and What Redis Alone Can't Model Cleanly

A season ends: the bottom three clubs go down, the other seventeen carry into the next season, and three newly promoted clubs are added by hand. The relegation half is genuinely a one-liner in Redis, thanks to Chapter 7's sorted set. But building the rollover exposes two mistakes in the key design of Chapters 7 and 8 that only show up once there is a second season — and the chapter then does what Chapter 1 promised, an honest accounting of what this variant costs.

A Bug the First Season Couldn't Show

Chapter 7 kept each team's stats in `team:{id}:stats` — keyed by team, not by season. That works for exactly one season. Playing one game in season 1 and one in season 2, both with those keys:

VERIFIED: STATS CARRY INTO THE NEXT SEASON

```
team:1:stats after one game in each of two seasons:
{'played': '2', 'gf': '2', 'ga': '0', 'won': '2'}
```

Nothing is wrong with the script; it dutifully added to whatever hash it was given. A new season's table would start with last season's records already in it.

The fix is to put the season in the key. Chapter 8's `totals`, `records` and `leaderboard` have the same flaw and the same fix:

```
season:{sid}:table           # Chapter 7's sorted set
season:{sid}:team:{tid}:stats # Chapter 7's per-team hash
season:{sid}:totals          # Chapter 8
season:{sid}:records
season:{sid}:leaderboard
```

VERIFIED: ONLY THE CALLER CHANGED

Both Lua scripts receive every key through `KEYS` (the Chapter 7 tip about not building key names inside a script), so neither script needed editing — only the routes that call them build different key names. With season-scoped keys the same two games give `played = 1` in each season, and running Chapter 8's script for two seasons kept the two totals hashes separate. Fixtures, predictions and per-fixture points stay unscoped: a fixture ID is already unique across seasons.

Relegation: One Call

Chapter 7's table is ordered best first, so the relegated clubs are simply the last three:

```
await r.zrange("season:1:table", -3, -1)    # the bottom three, e.g. ['004', '015', '016']
```

That is only meaningful once the season is over. A table half-way through a season ranks clubs on a few games each. The route should refuse to relegate anyone until every team has played all 38 games:

```
async def season_complete(r, sid, games=38):
    members = await r.zrange(f"season:{sid}:table", 0, -1)
    async with r.pipeline(transaction=False) as pipe:
        for m in members:
            pipe.hget(f"season:{sid}:team:{int(m)}:stats", "played")
        played = await pipe.execute()
    return len(members) == 20 and all(int(p or 0) == games for p in played)
```

VERIFIED ON A REAL 380-FIXTURE SEASON

```
after 50 fixtures:    season_complete -> False    played: [21, 21, 14, 3, 3, ...]
after all 380:        season_complete -> True     played: [38, 38, 38, 38, 38, ...]
```

The Rollover, in One Atomic Script

Carrying the survivors forward is several writes that must happen together: create the new season, copy seventeen teams into its membership set, seed each at zero in its table, mark the old season as rolled over, and make the new one current. As in earlier chapters, a Lua script makes that a single unit — and gives a double-click on "rollover" nothing to double:

```

ROLLOVER_LUA = """
if redis.call('EXISTS', KEYS[2]) == 1 then return -1 end      -- this season was already
local survivors = redis.call('ZRANGE', KEYS[1], 0, -4)      -- everyone except the bo
for _, mem in ipairs(survivors) do
    redis.call('SADD', KEYS[3], tostring(tonumber(mem)))    -- '008' -> team id '8'
    redis.call('ZADD', KEYS[4], -500000, mem)               -- zero-played score, as
end
redis.call('HSET', KEYS[5], 'name', ARGV[2])
redis.call('SET', KEYS[2], ARGV[1])                         -- mark old season done:
redis.call('SET', 'season:current_id', ARGV[1])
return #survivors
"""

new_id = await r.incr("season:next_id")
await rollover(keys=["season:1:table", "season:1:rolled", f"season:{new_id}:teams",
                    f"season:{new_id}:table", f"season:{new_id}"], args=[new_id, "2025

```

The route calls `season_complete` first and returns 409 if it is false, then takes a new season ID from `INCR` as in Chapter 2, and passes the script the keys it needs.

VERIFIED: TWO CONCURRENT ROLLOVERS, ONE WINS

Two calls for the same finished season fired together with `asyncio.gather`:

```

results: [17, -1]      # one rolled 17 clubs forward, the other was refused
season:current_id -> 2    season:1:rolled -> 2
new season: 17 teams in the Set and 17 in the table
relegated (season 1's last three): ['004', '015', '005'] -- none of them in the ne

```

The losing call had already taken season ID 3 from `INCR`, so ID 3 is never used. A gap in season numbering is harmless; the alternative — claiming the flag before taking an ID — would leave a half-finished rollover if the process died in between.

THE SCRIPT DOES NOT CHECK COMPLETENESS

Only the route does. Called directly on a season with a single fixture entered, the script rolled 17 clubs forward and relegated whoever happened to be at the bottom of a one-game table (`['019', '020', '002']`). The script also names one key, `season:current_id`, inside itself rather than receiving it in `KEYS` — fine on a single server, the same Cluster caveat as Chapter 7. Exercise 2 reproduces the first.

Relegation Deletes Nothing

Removing a club from the next season is an omission, not a deletion. After the rollover, season 1 still has 20 teams in its Set and 20 in its table, and every `team:{id}` hash is untouched. The relegated clubs' history is intact and can come back if they are promoted later. The same design choice as the relational siblings, and simpler here because a Set membership is just a list of IDs.

The Three Promoted Clubs

Which clubs are promoted is real-world knowledge no query here could produce, so — as in every sibling — the three are entered by hand through Chapter 3's add-team route, which also enforces the 20-team cap with `SCARD`. That route needs one change. Season membership now lives in *two* structures: the Set (Chapter 3) and the table (Chapter 7).

VERIFIED: A PROMOTED CLUB MISSING FROM THE TABLE

After the rollover (17 clubs), adding three new team IDs with only Chapter 3's `SADD`:

```
SADD only:                Set = 20 teams,   table = 17 teams
after new team 21's first result: table = 18   # teams 22 and 23 still absent
adding ZADD for 22 and 23:   table = 20
```

The Set says 20 clubs; the table shows 17. Chapter 7's script does `ZADD` a team the first time it plays, so the table heals itself one club at a time — but until then a promoted club is simply not listed. The add-team route must `ZADD` the member to the table in the same step as the `SADD`.

This is the theme of the whole variant showing up once more: where a relational database has one `season_teams` row that every query reads, this design has two structures that the code must keep in agreement.

What Redis Alone Can't Model Cleanly Here

Chapter 1 asked to be taken on trust that this course was deliberately narrower than its PostgreSQL sibling. Nine chapters of evidence can now be added up. Each item below was checked, not assumed.

1. Durability — the one that matters most

A default Redis server keeps data in memory and only occasionally writes a snapshot. Checked against the standard image: `save = 3600 1 300 100 60 10000` (snapshot after an hour if at least one key changed, after five minutes if 100 changed, after one minute if 10,000 changed) and `appendonly = no`. Two keys were written, then the container was killed abruptly, as in a crash or power loss:

VERIFIED: WHAT SURVIVES A HARD KILL

```
default config, docker kill -> dbsize 0    season:current_id -> None
default config, docker stop  -> dbsize 2    season:current_id -> '7'      # graceful
--appendonly yes --appendfsync always, docker kill -> dbsize 2    season:current_
```

Under the defaults, everything written since the last snapshot is lost on a crash — here, all of it. Turning on the append-only file with `appendfsync always` kept both writes through the same hard kill, at the cost of an fsync per write (there is a cheaper setting in between that risks a short window of writes; this chapter didn't test it).

For this app the stakes are specific. The relegation-table and leaderboard are *derived* and can be rebuilt (Chapter 7 showed that). But the fixtures and the predictions are hand-typed source

data — there is nothing to recompute them from. Running this variant as the only copy means either enabling the append-only file or accepting that a crash means re-entering data.

2. Size is not the problem

The usual worry about an in-memory store is fitting the data. Ten seasons of 380 fixtures each — 3,800 fixtures, each with five predictions, scored, with all the tables, stats, totals and leaderboards — came to 15,450 keys and about **1.7 MiB** (roughly 170 KiB per season, growing steadily). That fits in memory many thousands of times over.

3. A second axis needs a second structure

"Which seasons has team 8 played in?" has no direct answer, because membership is stored season → teams, not team → seasons. The workable options are a scan or a second index:

```
seasons = [k async for k in r.scan_iter(match="season*:teams")]
hits = [k for k in seasons if await r.sismember(k, "8")]
# ['season:1:teams', 'season:2:teams'] -- correct
```

It works, and for a few seasons it is instant, but **SCAN** walks the keyspace rather than using an index. The alternative is a **team:{id}:seasons** Set that the add-team route also maintains — which is the same "two structures to keep in agreement" cost as the promoted-club bug above.

4. Integrity lives entirely in the application

Across the course, a fixture could reference a team that doesn't exist (Chapter 2), a prediction or a result could be written for a fixture that doesn't exist (Chapters 5 and 6), and a hash could be silently merged into by a reused ID (Chapter 2). Every guard was a Lua script or a route check the author had to remember to write.

NEED	RELATIONAL SIBLINGS	THIS COURSE (REDIS)
Ranked table, ranked leaderboard	A query on each read	Better fit: maintained sorted set; read is one range call
Increment safely under concurrency	Transactions, or a SUM	Good fit: atomic HINCRBY , scripts
Survive a crash by default	Yes	No — needs the append-only file turned on

NEED	RELATIONAL SIBLINGS	THIS COURSE (REDIS)
Referential integrity	Foreign keys	None — checks written by hand
Ask a new question of old data	Write a new query	Usually add a new structure and maintain it on every write
Fit in memory	Not a concern	Not a concern here (about 170 KiB per season)

THE HONEST ANSWER TO "IS REDIS EVER THE RIGHT FIT HERE?"

For the table and the leaderboard, yes — they are exactly what sorted sets are for, and both are derived data that can be rebuilt. As the *only* home for the fixtures and predictions, not without turning on persistence, and even then with integrity and new-question costs that a relational database gives for free. The design that fits this data best is often the two together: a durable database as the source of truth, with Redis holding the ranked views. Chapter 12 puts this variant side by side with the PostgreSQL one.

Hands-On Exercises

EXERCISE 1

Enter one 1-0 result in each of two seasons using team-keyed stats hashes, report team 1's stats, then repeat with season-scoped stats keys and report each season's played count. State what changed in the script (nothing) and what changed in the caller.

 [View solution](#)

EXERCISE 2

Call the rollover script directly on a 20-team season with only one fixture entered, and report which clubs it relegates. Then call it a second time and report the result and what happened to the season ID that call took.

 [View solution](#)

EXERCISE 3

Start a default Redis container and one with the append-only file and fsync always, write two keys to each, kill both with `docker kill`, restart them and count the keys. Then repeat the default one with `docker stop`. Report all three results and say which restart behaviour to rely on for hand-entered fixtures.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- **Verified: team-keyed stats carry across seasons** — scope every per-season key by season; scripts that take `KEYS` need no change
- **Relegation is** `ZRANGE table -3 -1` — but only after every team has played 38
- **Rollover is one atomic script** — survivors = `ZRANGE 0 -4`; a rolled-over flag makes a second call return -1 (verified: `[17, -1]`)
- **Relegation deletes nothing** — the old season's Set and table stay intact
- **Verified: two structures, one membership** — `SADD` alone left a promoted club out of the table; also `ZADD`
- **Verified: default Redis loses everything since the last snapshot on a hard kill** — the append-only file with `fsync always` survived
- **About 170 KiB per season** — memory is not the constraint; durability, integrity and new-question cost are

CHAPTER 10: STYLING & THE GAMEWEEK/SEASON SELECTOR UI

Premier League Predictor: FastAPI & Redis

Chapter 10 · Styling & the Gameweek/Season Selector UI

Nine chapters of backend, and the app still has no way to say *which* gameweek you are looking at. This chapter adds the selector — a season menu and a gameweek menu that drive the fixture-entry page — and gives the whole page a proper dark theme. It also closes a gap Chapter 4 left open, and finds two bugs in Chapter 4's own route while doing it. Everything here was run: the routes against a real Redis with real HTTP requests, the JavaScript in Node, and the finished page rendered in headless Chrome.

Redis Doesn't Keep a List of Seasons

The selector needs two lists: every season, and every gameweek in a season. A relational database answers both with a `SELECT`. In Redis the seasons exist only as scattered keys, and the obvious way to find them is `SCAN` with a pattern:

VERIFIED: `SEASON:*` MATCHES FAR MORE THAN SEASONS

```
await [k async for k in r.scan_iter(match="season:*")]
# ['season:1', 'season:1:gameweeks', 'season:1:teams', 'season:2', 'season:2:gamev
# 'season:2:teams', 'season:current_id', 'season:next_id']
```

One pattern returns season records, their sets, and two unrelated counters, all mixed together and in no particular order. The same problem hits gameweeks: `SCAN` over `gameweek:1:*:fixtures` found the right keys, but pulling the numbers out and sorting them as strings gives `['1', '10', '2', '3']` — Chapter 7's "10 sorts before 2" trap again.

The idiomatic answer is to stop searching the keyspace and keep the list yourself, in a sorted set whose *score* is the number. A sorted set is ordered numerically, so the string-order trap disappears:

```
seasons                                # sorted set: member = season id, score = season id
season:{sid}:gameweeks                 # sorted set: member = gameweek number, score = gameweek number

await r.zrange("season:1:gameweeks", 0, -1)    # ['1', '2', '3', '10'] -- numeric order
```

The cost is the one this course keeps meeting: a second structure that has to be written to at the right moments. Two write paths change. The fixture-creation route adds the gameweek (`ZADD season:{sid}:gameweeks n n`, which does nothing if it is already there). And the rollover script from Chapter 9 must add the new season:

VERIFIED: CHAPTER 9'S ROLLOVER LEAVES THE NEW SEASON OFF THE LIST

Running Chapter 9's script as published, then asking the API for the seasons:

```
rolled 17 clubs -> /api/seasons: [2024/25, 2025/26 (no longer current)]
season:current_id = 3, and season 3 exists -- but is not listed
```

The new current season exists and isn't in the menu, and no listed season is marked current. Adding one line to the script, `redis.call('ZADD', KEYS[6], ARGV[1], ARGV[1])`, with `"seasons"` passed as the sixth key, fixed it: the next rollover appeared, marked current.

The Read Routes

Three routes feed the page. The third returns everything the fixture-entry screen needs in one response — teams, fixtures and the already-used team IDs — so the client never has to stitch several requests together:

```
@app.get("/api/seasons")
async def list_seasons():
    r = app.state.redis
    ids = await r.zrange("seasons", 0, -1)
    async with r.pipeline(transaction=False) as p:
        for i in ids: p.hget(f"season:{i}", "name")
    names = await p.execute()
    current = await r.get("season:current_id")
```

```

        return [{"id": int(i), "name": n, "current": i == current} for i, n in zip(ids, names)]

@app.get("/api/seasons/{sid}/gameweeks")
async def list_gameweeks(sid: int):
    r = app.state.redis
    if await r.zscore("seasons", sid) is None:
        return JSONResponse({"error": "no such season"}, status_code=404)
    return [int(n) for n in await r.zrange(f"season:{sid}:gameweeks", 0, -1)]

@app.get("/api/gameweeks/{sid}/{n}")
async def gameweek_state(sid: int, n: int):
    # same 404 guard, then: SMEMBERS the fixtures, the season's teams and the used team
    # and one pipeline of HGETALL/HGET for the fixture hashes and team names
    ...

```

Fixture and team lookups are pipelined, as in Chapters 5 and 7. The season guard matters because of Chapter 2's missing referential integrity: without it, asking for a season that doesn't exist returns an empty list with a 200, indistinguishable from a real season with no gameweeks yet. With it, the call gets a 404 (verified). A real season with a gameweek nobody has entered still returns 200 with empty lists, which is correct.

Fixing the errors to Chapter 4

Building the page meant driving Chapter 4's create-fixture route from a real client, and it turned out to have two problems.

1. FastAPI ignores a status code returned in a tuple

Chapter 4's route ended with lines like `return {"error": ...}, 400`. That is Flask's convention. In FastAPI:

VERIFIED: THE 400 NEVER HAPPENS

```

@app.get("/x")
async def x(): return {"error": "team used"}, 400

GET /x -> 200 [{"error": "team used"}, 400] # a two-element JSON array

```

The client receives success and an array. A page checking `res.ok` would never see a refusal. Use `JSONResponse(..., status_code=...)` or raise `HTTPException`.

2. A rejected pairing still consumed the home team

Chapter 4 claimed the home team, then separately the away team. If the second claim is refused, the first has already happened:

VERIFIED: A TEAM BURNED WITH NO FIXTURE

```
team 4 already used this gameweek; try to pair team 3 (home) with team 4 (away):
home: ACCEPTED    away: REJECTED    -> used_teams = ['3', '4']    # team 3 is now
```

Each claim was individually race-safe, but the two together weren't atomic. The fix is one `WATCH` transaction that checks *both* teams and claims both, or neither:

```
async def claim_both(r, key, a, b):
    async with r.pipeline(transaction=True) as pipe:
        while True:
            try:
                await pipe.watch(key)
                if await pipe.sismember(key, a) or await pipe.sismember(key, b):
                    await pipe.unwatch()
                    return False
                pipe.multi()
                pipe.sadd(key, a, b)                # both members in one command
                await pipe.execute()
                return True
            except WatchError:
                continue                            # re-check, as in Chapter 4
```

VERIFIED THROUGH THE REAL ROUTE

```
team 3 v team 4 (4 already used):    409    used_teams still ['4']    # nothing t
two simultaneous POSTs, same pair: [200 {'fixture_id': 15}, 409 ...]    fixtures cr
```

The route also refuses a team playing itself (400) and, via the guard, a nonexistent season (404). After a successful claim it creates the fixture hash, adds it to the gameweek's fixture set, and **ZADD**s the gameweek to the season's list, all in one **MULTI** / **EXEC**.

The Client: Resolving Chapter 4's Open Gap

Chapter 4's page kept a JavaScript **usedTeamIds** set in memory and flagged that it would go stale on a gameweek switch. Two separate failures are hiding there, and both were reproduced in Node with a stubbed **fetch**.

```
// selector.js -- holds the selection and announces changes
const state = { seasonId: null, gameweek: null };
export function getSelection() { return { ...state }; }
export function select(seasonId, gameweek) {
  state.seasonId = seasonId; state.gameweek = gameweek;
  window.dispatchEvent(new CustomEvent("selectionchange", { detail: { seasonId, gameweek } }
}
```

VERIFIED: FAILURE ONE — MERGING INSTEAD OF REPLACING

A set that only ever has IDs added to it, shown after switching from gameweek 1 (teams 1–6 used) to gameweek 2 (teams 7 and 8 used):

```
client shows used:  [1, 2, 3, 4, 5, 6, 7, 8]      server says:  [7, 8]
```

Six teams are disabled that are perfectly free. The state has to be *replaced* from the server's answer on every switch, never merged into.

VERIFIED: FAILURE TWO — A SLOW RESPONSE ARRIVING LAST

Selecting gameweek 1 and then gameweek 2 in quick succession, where gameweek 1's response is slower:

```
naive:   showing gameweek 1, used [1, 2, 3, 4, 5, 6]      # the menu says gameweek
```

Whichever response arrives last wins, and it isn't necessarily the one for the current selection. The fix is a ticket: each request takes a number, and only the response holding the latest number is allowed to render.

```
// loader.js
export function createLoader(fetchState, onData) {
  let latest = 0;
  return async function load(selection) {
    const ticket = ++latest;
    const data = await fetchState(selection);
    if (ticket !== latest) return; // a newer selection superseded this one
    onData(data); // replaces what was shown
  };
}
```

VERIFIED: THE TICKETED LOADER

ticketed loader, same two clicks: showing gameweek 2, used [7, 8]

Chapter 4's "usedTeamIds" gap is closed: the used teams come from the server's `used_team_ids` on every load, and the same server-side guard that decides whether a pairing is allowed also produces the list the buttons are drawn from. The page never decides anything.

The superseded request is still *sent* — the ticket only stops its answer being shown. For two menus and a one-person tool that is fine.

Styling

The page uses the same dark palette as the rest of the course, with Redis's red as the accent. The parts that carry meaning are small: a used team is dimmed, struck through and disabled, and the picked home team gets the accent fill.

```
:root { --bg:#0d1117; --card:#161b22; --line:#30363d; --text:#c9d1d9; --accent:#dc382d;

.grid { display:grid; grid-template-columns:repeat(auto-fill,minmax(140px,1fr)); ga
```

```
.team-btn { background:var(--card); color:var(--text); border:1px solid var(--line); }
.team-btn:hover:not(:disabled) { border-color:var(--accent2); background:#24100e; }
.team-btn.picked { border-color:var(--accent); background:#5c1a15; color:#fff; }
.team-btn.used { opacity:.4; text-decoration:line-through; cursor:not-allowed; }
```

`auto-fill` with a 140 px minimum lets the twenty buttons reflow to whatever width the page has, and a small media query makes the selectors full-width on narrow screens. Rendered in Headless Chrome at 1,000 px wide against the seeded data, the page showed the selector bar, twenty team buttons in a five-column grid with the four already-used teams struck through, and the two entered fixtures listed underneath. The narrow-screen rule wasn't rendered.

STACK

WHERE THE SELECTOR'S LISTS COME FROM

Relational
siblings

A `SELECT` over the seasons and gameweeks tables

This course
(Redis)

Two sorted sets maintained by the routes that create seasons and fixtures; a `SCAN` over the keyspace was tried and rejected

Hands-On Exercises

EXERCISE 1

List season 1's gameweeks using `SCAN` over `gameweek:1*:fixtures`, extract the numbers and sort them as strings and as numbers, then compare with the sorted-set answer. Explain what the string order would do to the gameweek menu.

 [View solution](#)

EXERCISE 2

Call the create-fixture route with a team playing itself, with a nonexistent season, and request an empty but valid gameweek's state. Report the status code and body for each, and whether anything was left behind in Redis.

 [View solution](#)

EXERCISE 3

Select gameweeks 1, 2 and 3 in quick succession where the responses take 80, 40 and 10 ms respectively. Report which gameweek a naive handler and the ticketed loader end up showing, and how many requests were still sent.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE

- **Verified:** `SCAN season:*` **is not a season list** — it returns counters and sets too; keep `seasons` and `season:{sid}:gameweeks` as sorted sets
- **Sorted sets give numeric order** — `[1, 2, 3, 10]`, not the string order `['1', '10', '2', '3']`
- **Verified: Chapter 9's rollover must also** `ZADD seasons` — otherwise the new current season isn't listed
- **Guard reads with the season check** — an unknown season is a 404, not an empty 200
- **Verified correction: FastAPI ignores** `return body, 400` — use `JSONResponse` or `HTTPException`
- **Verified correction: claim both teams in one WATCH transaction** — a refused away team no longer burns the home team
- **Replace, don't merge, on every switch** — verified: a merged set showed 8 used teams where the server said 2
- **Ticket each request** — verified: without it a slow gameweek 1 response overwrote gameweek 2

CHAPTER 11: DEPLOYMENT

Premier League Predictor: FastAPI & Redis

Chapter 11 · Deployment

Deploying the PostgreSQL sibling meant running a web server and configuring a database it connects to. Deploying this variant is the same plus one thing that changes the picture: Redis is a running server with its own defaults, and Chapter 9 already showed that those defaults lose data. This chapter covers the app side briefly — it follows the earlier siblings — and spends its time on the Redis side, where five things needed checking. Every result below was run: the app under gunicorn with four workers in a Linux container, talking to a real Redis 8.10.1 container.

The App Side

Gunicorn doesn't run on Windows, so it was run inside a `python:3.13-slim` container with Redis on the same Docker network. The command and the settings that change between environments:

```
# environment
REDIS_URL=redis://plredis:6379/0
REDIS_MAX_CONNECTIONS=10

gunicorn app_prod:app -k uvicorn.workers.UvicornWorker -w 4 -b 0.0.0.0:8000
```

The static page is still served by the same app (`StaticFiles`, as in Chapter 1), so the page and the API share an origin and no CORS configuration exists to get wrong. The interactive API documentation is switched off unless a debug flag is set:

```
DEBUG = os.environ.get("APP_DEBUG") == "1"
app = FastAPI(lifespan=lifespan,
              docs_url="/docs" if DEBUG else None,
              redoc_url=None,
              openapi_url="/openapi.json" if DEBUG else None)
```

Verified in the running container: `/docs` and `/openapi.json` both return 404. (Switching off `/docs` alone would have left the schema itself readable at `/openapi.json`.)

Connections: The Default Pool Fails Under Load

The PostgreSQL sibling's headline deployment finding was connection arithmetic: four workers, each with its own pool, opened up to 60 connections. Redis has a version of the same problem, and this one is specific to how this app uses Redis. Chapter 4's `WATCH` transaction — and the `claim_both` function from Chapter 10 — holds one connection exclusively from the `WATCH` until the `EXEC`. Each request doing that occupies a connection for the whole check-then-write gap.

redis-py 8.1.0, the version tested, gives every client a pool with a default `max_connections` of **100**. To see what that does, 200 concurrent claims were fired, each holding its `WATCH` for 50 ms to mimic a realistic gap, while a monitor counted the server's connected clients:

VERIFIED: THREE POOLS, 200 CONCURRENT WATCH CLAIMS

pool	peak clients	succeeded	failed	
default (max_connections 100)	101	100	100	MaxConnection
ConnectionPool(max_connections=20)	21	20	180	MaxConnection
BlockingConnectionPool(max=20)	21	200	0	

(The peaks include the one monitoring connection.) A plain pool refuses the request that would exceed it, immediately. A `BlockingConnectionPool` makes the request *wait* for a free connection instead, and all 200 completed within the pool's limit of 20. For a request that briefly holds a connection, waiting is what is wanted.

The app therefore builds its pool explicitly, with a wait timeout so that a stuck pool becomes an error rather than a hang:

```
from redis.asyncio import BlockingConnectionPool

@asynccontextmanager
async def lifespan(app):
    pool = BlockingConnectionPool.from_url(REDIS_URL, max_connections=MAX_CONN,
```

```

                                timeout=10, decode_responses=True)
app.state.redis = redis.Redis(connection_pool=pool)
yield
await app.state.redis.aclose(close_connection_pool=True)

```

A SHUTDOWN DETAIL THAT COST A WRONG NUMBER

A client given a pool of its own doesn't close that pool when it is closed. An earlier version of the test above called plain `aclose()` between runs and reported a peak of 41 instead of 21, because the previous run's connections were still open. Passing `close_connection_pool=True` fixed it, and the app's shutdown uses it for the same reason.

With `REDIS_MAX_CONNECTIONS=10` and four workers, the ceiling is 40 connections. Under real unicorn, 300 simultaneous fixture-creation requests (each doing the `WATCH` claim, the `INCR` and the transaction) all succeeded in 3.2 seconds, with a peak of **35** application connections — within the 40. Redis's own `maxclients` is 10,000 by default, so Redis isn't the limit here; the pool size is a control on how much waiting and memory the app tolerates, and it should be workers × per-worker maximum, kept well below `maxclients`.

Persistence, Set Deliberately

Chapter 9's finding is the reason this section exists: the default server lost everything written since its last snapshot on a hard kill, while the append-only file with `appendfsync always` survived. For a deployment where the fixtures and predictions are the only copy, the configuration is a decision to be made and written down, not left to defaults. The settings were exercised as command-line flags (`--appendonly yes --appendfsync always`); the same names go in `redis.conf`:

```

appendonly yes
appendfsync always           # survived a hard kill in Chapter 9; 'everysec' is the cheap

```

Memory Policy: Choose to Fail Loudly

The default is `maxmemory 0` (no limit) with `maxmemory-policy noeviction`. Someone hardening the server, or copying a caching tutorial, might set a limit and an eviction policy like `allkeys-lru`. For a *cache* that is right; for source data it is not. With a deliberately tiny 2 MB limit and 4,000 writes of 500-byte values:

VERIFIED: ONE POLICY REFUSES, THE OTHER DELETES

```
noeviction:    wrote 752 keys, then "command not allowed when used memory > 'maxmemory'
allkeys-lru:   wrote 4000 keys, no error at all
```

With `allkeys-lru` every write reported success and 3,341 keys were silently discarded to make room, including the oldest fixture. With `noeviction` the server stopped accepting writes and said so. For this app, choose `noeviction` and, if a limit is set at all, leave plenty of headroom: ten seasons of data measured about 1.7 MiB in Chapter 9, so even a limit in the hundreds of megabytes is generous.

Security: The Defaults Are Open

Checked against the official image: `bind` is `*::*` (all interfaces), `protected-mode` is `no`, and there is no password. With the port published, any client that can reach it can run any command:

VERIFIED: ANYONE WHO CAN CONNECT CAN WIPE IT

```
await r.flushall() # from the host, no credentials -> True
```

Two layers of defence, both worth having. First, don't expose Redis at all: in the unicorn test the app reached it by container name over a private Docker network, and only the app's port needs to be published. Second, give the app its own account instead of the all-powerful default:

```
ACL SETUSER app on >apppw ~* &* +@all -@dangerous
CONFIG SET requirepass adminpw
```

VERIFIED: THE APP STILL WORKS; THE DANGEROUS COMMANDS DON'T

Running as that user, every command the app uses succeeded — `INCR`, `HSET`, `SADD`, `ZADD`, `ZRANGE`, a `WATCH` / `MULTI` / `EXEC` transaction, a Lua script (the Chapter 7 result script), `PUBLISH`, and `SCAN`. `FLUSHALL`, `CONFIG GET` and `KEYS` were each refused with "User app has no permissions to run the '...' command". After `requirepass`, a client with no credentials was refused outright (redis-py reports it as a failed `HELLO`, because it tries to authenticate as it connects).

Keys can be restricted too. A user limited to `~season:*` and `~gameweek:*` could `SADD` to a gameweek set and `ZADD` to a season table, but `HSET fixture:1` and `INCR fixture:next_id` were refused, and a Lua script that touched an out-of-pattern key was refused partway through. The refusal did *not* undo the script's earlier, permitted write: the permitted key existed afterwards and the forbidden one didn't — Redis doesn't roll back a script any more than it rolls back a failed transaction (Chapter 3's finding; Exercise 2). The app's full key set is broad, so `~*` is the practical setting here; the narrow form matters if a second service ever gets its own account. These were set with runtime `ACL SETUSER` commands; keeping them across restarts needs them in `redis.conf` or an ACL file, which wasn't tested.

Two Things That Turned Out Not to Be Problems

The Lua script cache. Every Lua script in this course is loaded into Redis and then called by its hash. A Redis restart or `SCRIPT FLUSH` empties that cache, which could break a running app with "no matching script" errors. It doesn't: after `SCRIPT FLUSH`, the same `register_script` object ran again and returned the right result (verified) — redis-py reloads the script when needed.

A Redis restart under a running app. With Redis stopped, a health route that does a `PING` returned `503 {"status": "redis unavailable"}`; after Redis was started again, the route returned to `200` without restarting the app. The first check a few seconds after the restart was still 503 (Redis hadn't finished starting) and later checks were 200 — so a health check should be retried, not trusted once.

```
@app.get("/healthz")
async def healthz():
    try:
        await app.state.redis.ping()
    except RedisError:
```

```
return JsonResponse({"status": "redis unavailable"}, status_code=503)
return {"status": "ok"}
```

Backups

Persistence protects against a crash; it doesn't protect against a mistake or a lost disk. A snapshot file can be copied off the machine. The procedure was tested end to end: write data, run `BGSAVE`, wait for `rdb_bgsave_in_progress` to reach 0, copy `/data/dump.rdb` out, place it in a brand-new container's `/data`, and start it.

VERIFIED: A SNAPSHOT RESTORES A WHOLE DATABASE

```
source: 202 keys (200 fixtures, a season table, the current-season pointer), dur
restored: dbsize 202 | fixture:200 {'home_team_id': '1', 'away_team_id': '2'} | ta
```

A BACKUP IS ONLY AS RECENT AS ITS SNAPSHOT

The snapshot holds what existed when `BGSAVE` ran. Fifty fixtures written afterwards were absent from the copied file: the source had 150 keys, the restore had 100 (Exercise 3). A backup job should run `BGSAVE`, wait for it, then copy — not just copy whatever file is there.

CONCERN	POSTGRES SQL SIBLING	THIS COURSE (REDIS)
Connection sizing	4 workers × default pool = up to 60; sized down to 28	Default pool is 100 per worker and <i>fails</i> at the limit; a blocking pool of 10 per worker peaked at 35
Crash safety	Default	Off by default; needs <code>appendonly yes</code>
Out-of-memory behaviour	Errors	Depends on policy; <code>allkeys-lru</code> deletes silently
Access control	Dedicated role	Default is open; an ACL user with <code>~@dangerous</code>
Backup	<code>pg_dump</code>	<code>BGSAVE</code> + copy <code>dump.rdb</code>

Hands-On Exercises

EXERCISE 1

Run 200 concurrent WATCH-guarded claims (with a 50 ms hold) through a default pool, a `ConnectionPool` with `max_connections=20`, and a `BlockingConnectionPool` with `max_connections=20`. Report how many succeed, how many fail and with what error, and the peak client count for each.

 [View solution](#)

EXERCISE 2

Create an ACL user limited to keys matching `season:*` and `gameweek:*`. Try SADD on a gameweek key, ZADD on a season key, HSET on a fixture key, INCR on `fixture:next_id`, and a Lua script that writes one permitted and one forbidden key. Report each result, and whether the permitted write inside the script took effect.

 [View solution](#)

EXERCISE 3

Write 100 fixtures, run BGSAVE and wait for it, write 50 more, then copy `dump.rdb` into a fresh container and start it. Report the live key count and the restored key count, and explain what a correct backup job does differently.

 [View solution](#)

CHAPTER 11 QUICK REFERENCE

- **gunicorn + `UvicornWorker`**, **settings from the environment** — page and API on one origin; `/docs` and `/openapi.json` both off (verified 404)
- **Verified: the default pool fails at its limit** — 100 of 200 concurrent WATCH claims got `MaxConnectionsError`; a `BlockingConnectionPool` completed all 200
- **Pool = workers × per-worker max** — 4×10 held 300 concurrent requests at a peak of 35 connections
- **Close the pool you built** — `aclose(close_connection_pool=True)`

- **Persistence is a decision** — `appendonly yes` (Chapter 9: survived a hard kill)
- **Verified:** `allkeys-lru` **silently deletes source data** — keep `noeviction` so full memory is a loud error
- **Verified: the defaults are open** — no password, all interfaces; keep it off public networks and give the app an ACL user with `-@dangerous`
- **Lua scripts and Redis restarts are fine** — the script cache reloads itself; health checks should be retried after a restart
- **Backups:** `BGSAVE`, **wait, copy** — verified restore; a stale snapshot restored 100 keys of 150

CHAPTER 12: CAPSTONE: COMPARING AGAINST THE POSTGRESQL VARIANT, AND INTEGRATING INTO THE EXISTING ASTRO SITE

Premier League Predictor: FastAPI & Redis

Chapter 12 · Capstone: Comparing This Against the PostgreSQL Variant, and Integrating Into the Existing Astro Site

Two jobs remain. First, the question Chapter 1 asked and eleven chapters have been answering piece by piece: how does this variant actually compare with its PostgreSQL sibling? Second, the reason the whole project exists — getting the tool onto the live Astro site. As before, everything below was run: a real PostgreSQL 16 loaded with the same season for a head-to-head, and a real nginx mounting the app at a sub-path in front of the Chapter 11 deployment.

What Each Chapter Found

CH.	WHAT IT BUILT	THE REDIS-SPECIFIC FINDING (ALL REPRODUCED)
1	Setup, the shared spec	FastAPI's lifespan handler, not the deprecated startup hook, even where an official tutorial still shows the old one
2	Modeling	Key names are the only schema; <code>HSET</code> merges two records silently; nothing checks a fixture's team IDs exist
3	Admin	One <code>season:current_id</code> key replaces the relational "clear the old flag" step; but <code>MULTI</code> / <code>EXEC</code> does not roll back a failed command
4	Fixture entry	A real check-then-act race, closed with <code>WATCH</code> , whose retry must re-validate
5	Predictions	A hash field per source makes "one prediction per source" free; Redis validates nothing; a Lua script closes the lock-after-result gap
6	Results and scoring	Increment-based totals aren't idempotent; store per-fixture points and apply new-minus-old; <code>HINCRBYFLOAT</code> drifts
7	League table	A sorted set has one score, so three sort keys are packed into one; ties fall back to member string
8	Prediction leaderboard	Mixed units ranked a guest first by ~100×; Pub/Sub keeps no history; Chapter 6's rounding accumulates

CH.	WHAT IT BUILT	THE REDIS-SPECIFIC FINDING (ALL REPRODUCED)
9	Promotion and relegation	Keys need season scope; a second structure for one fact goes stale; defaults lose data on a hard kill
10	Selector UI	<code>SCAN</code> is not a list of seasons; two bugs in Chapter 4's route; stale-state bugs in the client
11	Deployment	The default pool fails at its limit; <code>allkeys-lru</code> silently deletes data; the defaults are open; backups are as old as the snapshot

Read down the right-hand column and one pattern repeats: a relational database enforces something, and here the application has to. Uniqueness, integrity, atomicity across two writes, keeping a derived view consistent, surviving a crash — each had a guard that was written by hand, and each was found by failing without it.

A Measured Head-to-Head

The same 380 random results (20 teams, a full double round robin) were loaded into this course's Redis structures and into a PostgreSQL 16 database with a `teams` table and a `fixtures` table. The PostgreSQL league table was computed with the standard query: a `UNION ALL` of home and away rows, aggregated per team. (This is an equivalent query, written for this comparison, not a copy of the sibling course's own.)

VERIFIED: SAME ANSWER FROM BOTH

```
fixtures loaded: 380
played/won/drawn/lost identical for every team:      True
(points, goal difference, goals scored) order equal: True
after 300 random corrections, still equal:            True
```

The maintained sorted-set table and the recomputed SQL table agree exactly — including after the corrections that Chapter 7's reverse-then-apply script exists to handle.

Then the timings, each the median of 300 runs from the same Python process over local Docker containers:

OPERATION	REDIS (DEFAULT CONFIG)	REDIS (APPEND-ONLY, FSYNC ALWAYS)	POSTGRESQL 16
Read the league table	2.30 ms	not measured	1.01 ms
Enter or correct one result	0.57–0.80 ms	3.12 ms	1.75 ms

HOW TO READ THAT HONESTLY

The read: PostgreSQL was more than twice as fast. Reading the Redis table takes two round trips — the `ZRANGE`, then a pipeline of twenty `HGETALL` calls for the stats — while the SQL query is one. A single query over 380 rows costs almost nothing, so there was no scale at which Redis's maintained table paid for itself here.

The write: Redis looks faster, but only with its defaults, which don't flush to disk (Chapter 9). Configured to survive a crash, as Chapter 11 recommends, it was *slower* than PostgreSQL, which commits durably by default. The fair comparison is the middle column against the right-hand one.

These are one machine, one connection and local Docker containers, so treat the numbers as an ordering at this scale, not as benchmarks. They do say plainly that performance was never a reason to choose Redis for this app.

So Which One?

A judgement, resting on the evidence above. For this app the PostgreSQL variant is the better fit as the system of record: it is durable by default, it enforces integrity without hand-written guards, its table is one query, and it is at least as fast at this size. What this variant contributes is what it was built to show — sorted sets are an excellent way to keep a ranked list current, atomic scripts and `WATCH` are real tools for check-then-act problems, and knowing where the guards have to go is a useful skill. A reasonable design that uses both keeps PostgreSQL as the source of truth and Redis for ranked views, rebuilt from it whenever needed; Chapter 7 already showed the table can be rebuilt by replaying results through the script.

Getting It Onto the Site

The live site is an Astro site. Four realistic ways to add this app:

OPTION	HOW IT WORKS	TRADE-OFF
Subdomain	The app runs at <code>predictor.example.com</code> , separately from the site	Simplest; nothing shares a path. Needs its own DNS and certificate; visitors leave the main site's address
Path-mounted reverse proxy	nginx serves the site at <code>/</code> and forwards <code>/predictor/</code> to the app	One address; needs a server the owner controls. Built out below
API behind the existing frontend	The app becomes API-only and the Astro site renders the pages	Most integrated; a rewrite of the page, and a decision on how the site calls the API
Fuller migration	Move the site itself onto a framework that can host both	A large project in its own right

As in the Astro sibling, whether the site's actual hosting can run a reverse proxy at all is a separate question this course can't answer; if it can't, the subdomain is the fallback. The rest of this chapter builds the second option.

The nginx Configuration

```
server {
  listen 80;
  root /usr/share/nginx/html;           # the existing site

  location = /predictor { return 301 /predictor/; }
  location /predictor/ {
    proxy_pass http://plapp:8000/;      # trailing slash: strips /predictor/ before
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $remote_addr;
  }
  location / { try_files $uri $uri/ =404; }
}
```

The app itself is untouched: it still believes it lives at `/`, because nginx removes the prefix. Redis appears nowhere in this file. It stays on the private network (the app reached it by container name in Chapter 11's test), and only nginx's port is public — a published Redis port was used in this chapter's measurements only so the test scripts could reach it.

The Absolute-Path Problem, Again

Each sibling capstone hit the same underlying issue: an app that assumes it sits at the web root breaks when mounted under a prefix. Here it shows up in exactly one place. The page's stylesheet and scripts are referenced without a leading slash, so they resolved fine under `/predictor/` (all four returned 200). The `fetch()` calls, though, were written as `/api/...`:

VERIFIED: THE PAGE LOADS AND SHOWS NOTHING

Loaded in headless Chrome at `/predictor/`: the heading, the styled selector bar and the empty "Fixtures" section rendered, with **both menus empty and no team buttons**. The browser's `/api/seasons` went to the existing site's root, which returned 404 (the same request under `/predictor/api/seasons` returned 200). The app was fine; the client was asking the wrong address. Nothing on the server side ever noticed, because the requests never arrived.

The fix is to drop the leading slash from the four `fetch()` URLs (`fetch("api/seasons")`), so they resolve against the page's own address. Same page, mounted, after the change: menus populated, twenty team buttons, the four already-used teams struck through, two fixtures listed — identical to the unmounted page. A fixture created through the proxy (`POST /predictor/api/gameweeks/2/20/fixtures`) returned 200.

VERIFIED: WHY THE TRAILING SLASH MATTERS

A relative URL resolves against the page's address, and the address without its trailing slash is a different base:

```
new URL("api/seasons", "http://h/predictor") -> http://h/api/seasons
new URL("api/seasons", "http://h/predictor/") -> http://h/predictor/api/seasons
```

That is what the `location = /predictor { return 301 /predictor/; }` line is for (it returned 301 as intended). Without it, someone who types the address without the slash gets the broken page above. This is the counterpart of the FastAPI sibling's `<base>` tag and the Astro sibling's base-path variable — with a static page and no server-rendered links, the client's URLs are the only place a path lives.

Protecting the Entry Page

Chapter 3 noted the app has no authentication at all, and Chapter 11's Redis account protects Redis, not the routes. Anyone who can reach `/predictor/` can enter fixtures. As in the Astro

sibling, a pragmatic fix is HTTP Basic Auth at nginx:

```
location /predictor/ {
    auth_basic "Predictor";
    auth_basic_user_file /etc/nginx/conf.d/htpasswd;    # created with: openssl passwd
    proxy_pass http://plapp:8000/;
    ...
}
```

VERIFIED

no credentials, page:	401	no credentials, POST:	401 (nothing written)
wrong password, page:	401	right password, page:	200 right page
existing site's root:	200	# unaffected	

Basic Auth is credentials in every request, so it needs HTTPS in front of it (see the HTTPS/TLS Fundamentals course) — this test ran over plain HTTP on localhost, and that would not be acceptable on a real site.

What's Still Genuinely Open

The relational siblings ended by naming the unresolved items from the project notes, and they apply here with a Redis flavour:

- **Historical backfill.** Importing past seasons is a strength of the design: the result-entry script is idempotent, so a bulk import is just replaying results through it. Replaying a full 380-result season twice left the order, scores and every stats hash unchanged (Exercise 3).
- **Guest count.** Still undecided as a product question. Chapter 8's sixtieths scheme is exact for up to six guests in a week, and not beyond.
- **Fixture rescheduling.** Not built here. Moving a fixture between gameweeks would touch several structures at once — its own hash, two gameweeks' fixture sets, both used-teams sets, and the season's gameweek list — so it is exactly the kind of multi-structure update this course has warned about, and would need its own script.

- **Single user.** The app has no user concept; the Basic Auth above is a gate, not accounts. Multiple users would mean predictions keyed by user, which the current one-field-per-source hash does not model.
- **Championship data.** Promoted clubs arrive as manual input, as in every variant.

Hands-On Exercises

EXERCISE 1

Mount the app at `/tools/predictor/` instead, and separately try a location whose `proxy_pass` has no trailing slash. Report the status codes for the page, a script file and an API call in each case, and say what changed in the app (nothing) versus in nginx.

 [View solution](#)

EXERCISE 2

Load the same 380 results into Redis and PostgreSQL and compare the league tables, then time reading the table 300 times in each and entering a result 300 times in each, including Redis with the append-only file and `fsync` always. Report the medians and say which pair of numbers is the fair comparison for writes.

 [View solution](#)

EXERCISE 3

Import a full 380-result season through the Chapter 7 script, snapshot the sorted set and every team's stats hash, import the same 380 results again, and snapshot again. Report whether the two snapshots are identical and what that means for a backfill that gets interrupted and re-run.

 [View solution](#)

CHAPTER 12 QUICK REFERENCE

- **The pattern across the course** — a relational database enforces it; here the application must, and each guard was found by failing without it
- **Verified: identical tables** — 380 results in Redis and PostgreSQL agree on every count and the full order, before and after 300 corrections
- **Measured: no performance case for Redis here** — read 2.30 ms vs 1.01 ms; a durable write is 3.12 ms vs 1.75 ms (the default Redis write, 0.57–0.80 ms, doesn't flush to disk)
- **Path-mounting** — `proxy_pass` with a trailing slash strips the prefix; the app is unchanged; Redis stays private
- **Verified: absolute** `fetch("/api/...")` **breaks under a prefix** — the page renders with empty menus; relative URLs fix it, and need the trailing-slash redirect
- **Verified: nginx Basic Auth gates the page and the API** — needs HTTPS in real use
- **Open** — rescheduling, multiple users, guest count; backfill is safe to re-run

COURSE COMPLETE

This closes *Premier League Predictor: FastAPI & Redis*, the last of the four Premier League Predictor courses, joining its PostgreSQL, Django and Astro siblings. Its honest conclusion is not that Redis is wrong for this app, but that it is the right tool for the ranked views and the wrong place, by default, for the only copy of the data.