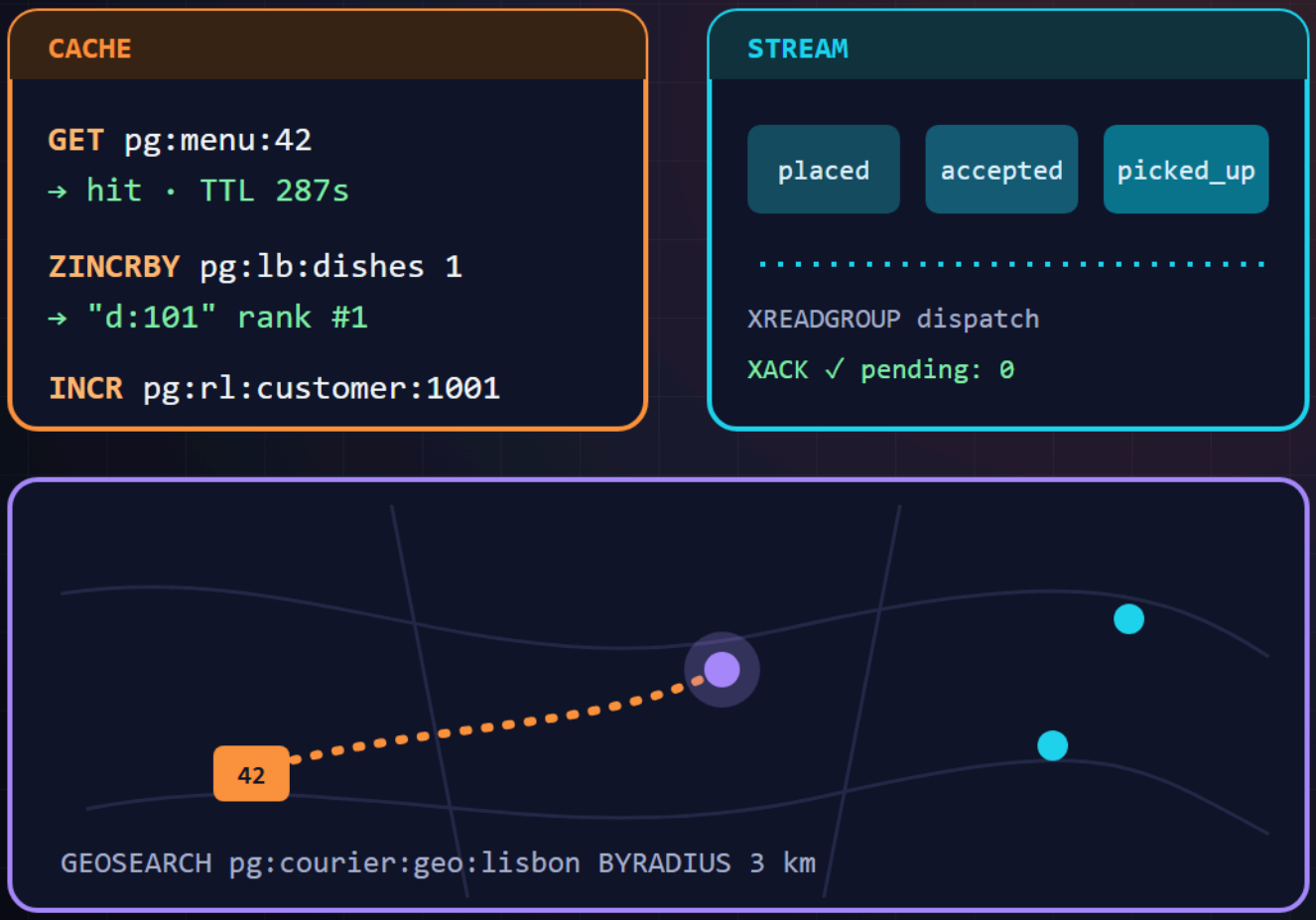


CACHE · QUEUE · STREAM · PUB/SUB · GEO · LOCK

THE COMPLETE PRACTICAL GUIDE

Redis in Practice

Caching, Queues, and Real-Time Systems

Data structures, streams, rate limits, locks, and live tracking — plus persistence, HA, Cluster, and security

CACHING · STREAMS · REAL-TIME · CLUSTER

THENEXBOOKS

Redis in Practice

Caching, Queues, and Real-Time Systems
With examples in redis-cli and Python — for Redis and compatible servers such as Valkey

First Edition

THENEXBOOKS

Redis in Practice
Caching, Queues, and Real-Time Systems

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Redis, its licensing, compatible servers, and managed services change over time; always consult the official documentation for the version and provider you use. Example outputs that contain timings, memory sizes, or counts are illustrative — your results will differ.

Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by thenexbooks is for referential purposes only and does not indicate any sponsorship, endorsement, or affiliation between Redis Ltd. and thenexbooks. Valkey is a trademark of the Linux Foundation. Python, Docker, and other product names are trademarks of their respective owners.

Plattergo, the running example in this book, is fictional, as are all restaurants, customers, couriers, and companies mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

Few pieces of infrastructure show up in as many architectures as Redis. It sits in front of databases as a cache, holds sessions and shopping carts, counts things, ranks things, rate-limits APIs, queues background jobs, streams events between services, tracks who is online and where they are, and coordinates work across many servers. Often the same Redis deployment does several of those jobs at once.

That versatility is exactly why Redis is easy to start with and surprisingly easy to misuse. A cache without an invalidation plan serves stale prices. A list used as a queue loses jobs when a worker crashes. A lock without an expiry freezes a system; a lock with an expiry but no fencing token can let two workers believe they both hold it. A single oversized key can stall every other client. None of these problems show up on a laptop — they show up in production, under load, at the worst possible moment.

This book is a practical guide to using Redis the way experienced teams do. We start with the data structures that make Redis different from a key-value store, then spend most of the book on the three jobs Redis is most often hired for — **caching**, **queues and messaging**, and **real-time features** — before going deep on atomicity, scripting, data modeling, performance, persistence, high availability, clustering, and security. We explain *why* each pattern works and where it breaks, not just *how* to type the commands.

Who This Book Is For

- **Backend developers** adding caching, background jobs, rate limits, or live features to an application.
- **Full-stack developers** who want sessions, carts, and real-time updates that hold up under traffic.
- **Engineers who already use Redis** and want solid answers on invalidation, reliable queues, streams, locks, and memory.
- **Tech leads and platform engineers** responsible for running Redis safely: persistence, replication, failover, cluster, and security.

What You Will Learn

- Strings, hashes, lists, sets, sorted sets, bitmaps, HyperLogLog, geospatial indexes, and streams — and when to use each.
- Caching patterns, TTLs, invalidation, stampede protection, hot keys, eviction, and memory management.
- Sessions, carts, client-side caching, and HTTP response caching.

- Queues with lists, pub/sub, keyspace notifications, and Redis Streams with consumer groups, retries, and dead letters.
- Background jobs with idempotency, priorities, and scheduling.
- Rate limiting, leaderboards, counters, presence, live tracking, geospatial search, and distributed locks.
- Pushing real-time updates to browsers and apps with WebSockets and Server-Sent Events.
- Transactions, pipelining, Lua scripting, Redis Functions, JSON, search, and vector similarity.
- Key design, memory efficiency, latency diagnosis, persistence, replication, Sentinel, Cluster, security, and managed services.

How This Book Is Organized

Part I — Redis Foundations (Chapters 1–5) sets up your environment and teaches every core data structure. **Part II — Caching** (Chapters 6–10) covers cache patterns, invalidation, stampedes, eviction, sessions, and client-side caching. **Part III — Queues, Streams, and Messaging** (Chapters 11–15) builds reliable queues, pub/sub, streams, and background jobs. **Part IV — Real-Time Features** (Chapters 16–20) implements rate limiting, leaderboards, presence, live tracking, locks, and push updates. **Part V — Advanced Redis** (Chapters 21–25) covers atomicity, scripting, JSON and search, data modeling, and performance. **Part VI — Production** (Chapters 26–28) covers persistence, replication, high availability, clustering, security, managed services, and a complete production case study. The appendices provide a command cheat sheet, templates and a command reference, a production checklist, and a glossary.

Each chapter includes practice exercises and ends with a **Key Takeaways** box for quick review.

The Running Example: Plattergo

Throughout the book we build the Redis side of **Plattergo**, a fictional food-ordering and delivery platform. Customers browse restaurants and menus, fill carts, and place orders; restaurants accept and prepare them; couriers share their location every few seconds; and customers watch their order move across a live map. The system of record is a relational database, and Redis sits beside it for speed, coordination, and real-time features. Every chapter uses the same key names and sample data — restaurant 42 "Casa Verde" in Lisbon, customer 1001 Maya, courier 77 Rui, order 9001 — so each new idea builds on the last. Chapter 2 includes a seed script that loads this data.

Versions and Conventions

Examples target recent versions of Redis (7.x and 8.x) and are compatible with Valkey unless a section says otherwise. Features tied to a specific version are marked where they appear, and Chapter 1 explains the licensing changes of recent years and how Redis, Valkey, and managed services relate. Application code uses Python with the `redis-py` client; the same patterns apply to any client library.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version or licensing details, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause data loss, outages, or security problems in production.

Commands appear in `monospace`, with redis-cli sessions shown as `127.0.0.1:6379>` followed by the reply. Key names follow the pattern `pg:{entity}:{id}`.

Contents

Preface

PART I — REDIS FOUNDATIONS

1 Why Redis: In-Memory Data Structures for Real Systems

2 Setting Up: Redis, Valkey, redis-cli, Docker, and Client Libraries

3 Keys, Strings, and Expiration: The Core Model

4 Hashes, Lists, and Sets

5 Sorted Sets, Bitmaps, HyperLogLog, and Geospatial Indexes

PART II — CACHING

6 Caching Fundamentals: Cache-Aside, Read-Through, Write-Through, and Write-Behind

7 Invalidation, TTLs, and Consistency with the Database

8 Stampedes, Hot Keys, Negative Caching, and Cache Warming

9 Eviction Policies and Memory Management

10 Sessions, Carts, Client-Side Caching, and HTTP Response Caching

PART III — QUEUES, STREAMS, AND MESSAGING

11 Lists as Queues: Blocking Pops, Reliable Queues, and Delayed Jobs

12 Pub/Sub, Sharded Pub/Sub, and Keyspace Notifications

13 Redis Streams I: Producers, Consumers, and Consumer Groups

14 Redis Streams II: Acknowledgments, Retries, Dead Letters, and Trimming

15 Background Jobs in Practice: Idempotency, Priorities, Scheduling, and Job Libraries

PART IV — REAL-TIME FEATURES

16 Rate Limiting: Fixed Window, Sliding Window, and Token Bucket

17 Leaderboards, Counters, and Real-Time Analytics

18 Presence, Live Tracking, and Geospatial Queries

19 Distributed Locks, Semaphores, and Coordination

20 Pushing Real-Time Updates to Clients: WebSockets and Server-Sent Events with Redis

PART V — ADVANCED REDIS

21 Transactions, Pipelining, and Atomicity

22 Lua Scripting and Redis Functions

23 Beyond Core Types: JSON, Search, and Vector Similarity

24 Data Modeling in Redis: Key Design, Secondary Indexes, and Memory Efficiency

25 Performance: Latency, Big Keys, the Slow Log, and Load Testing Your Workload

PART VI — PRODUCTION

26 Persistence, Replication, and High Availability with Sentinel

27 Redis Cluster, Scaling, Security, and Managed Services

28 Case Study: Running Plattergo's Real-Time Platform in Production

APPENDICES

A Redis Command Cheat Sheet

B Templates and Command Reference

C Production Checklist

D Glossary

PART I

Redis Foundations

What makes Redis different from a plain key-value store, a working setup with the Plattergo sample data, and every core data structure — strings, hashes, lists, sets, sorted sets, bitmaps, HyperLogLog, and geospatial indexes.

1 · Why Redis

2 · Setting Up

3 · Keys, Strings, and Expiration

4 · Hashes, Lists, and Sets

5 · Sorted Sets, Bitmaps, HyperLogLog, and Geo

CHAPTER 1 · PART I — REDIS FOUNDATIONS

Why Redis: In-Memory Data Structures for Real Systems

Redis is often introduced as "a cache," but that label hides most of what makes it useful. In this chapter we look at what Redis actually is, why its design makes it fast and predictable, where it fits beside a relational database, and how a food-delivery platform called Plattergo will use it throughout this book.

What Redis Is (and What It Is Not)

Redis is an in-memory data structure server. A client opens a network connection, sends a command such as `SET`, `HINCRBY`, or `ZADD`, and the server executes that command against data it keeps in RAM, then sends back a reply. The name originally stood for *REmote DIctionary Server*, and that is still a good mental model: a giant dictionary that lives on another machine, where every key maps to a value, and where the values are not just opaque blobs but typed data structures with their own operations.

That last point is the one most newcomers miss. A plain key-value store lets you store a string and read it back. Redis lets you store a **list** and push to its head, a **hash** and increment one field, a **set** and intersect it with another set, a **sorted set** and ask for the top ten members by score, a **stream** and have a group of consumers read from it with acknowledgments. The operations run on the server, next to the data. Your application does not have to fetch a document, modify it, and write it back; it asks Redis to do the modification in place, in a single command.

It helps just as much to be clear about what Redis is not:

- **It is not a replacement for your system of record.** Redis can persist data to disk, and it can replicate, but its core strengths are speed and data structures, not rich querying, referential integrity, or long-term storage of everything your business owns. Most production systems keep PostgreSQL, MySQL, or another database as the source of truth and put Redis beside it.
- **It is not a general-purpose query engine.** In its core form you look data up by key. There are no joins, no ad-hoc `WHERE` clauses over values. (Recent versions do ship a query engine for secondary indexing and search; Chapter 23 covers it. But the core model is key-first.)
- **It is not unlimited.** Everything lives in memory, and memory is finite and comparatively expensive. Designing for Redis means designing for a bounded working set, with expiration and eviction as first-class concerns.

Once you internalize "remote, typed, in-memory data structures with atomic server-side operations," many of the use cases in this book stop looking like separate tricks and start looking like the same idea applied repeatedly.

Why Redis Is Fast: Mechanisms, Not Magic

You will hear that Redis is fast. That is true, but "fast" on its own is not useful engineering information. What matters is *why* it is fast, because the reasons tell you when it will stay fast and when it will not. We will not quote throughput or latency figures here; they depend on hardware, network, payload size, command mix, and client behavior, and the only numbers you should trust are the ones you measure on your own workload with tools such as `redis-benchmark`, `memtier_benchmark`, or your own load tests. Instead, here are the mechanisms.

Data lives in RAM

Reading a value from memory avoids the disk seek, page cache miss, and buffer management that a disk-oriented database has to deal with on a cold read. Persistence (covered in Chapter 26) happens in the background or through an append-only log, so a normal read never waits for disk.

Commands have known complexity

Every Redis command is documented with its time complexity. `GET`, `SET`, `HGET`, `SADD`, and `INCR` are O(1). `ZADD` is O(log N). `LRANGE` is O(S+N), where S is the offset and N the number of elements returned. `KEYS *` and `SMEMBERS` on a huge set are O(N) in the size of the whole keyspace or set. This is a gift: you can reason about cost before you write code. A design that calls only O(1) and O(log N) commands will behave predictably as data grows; a design that relies on O(N) commands over growing collections will eventually hurt.

Single-threaded command execution

A Redis instance executes commands one at a time on a single main thread. Since version 6.0, Redis can use additional threads for network I/O (reading requests and writing replies), and background threads handle some tasks like freeing large values, but the actual execution of commands against the data is serialized. This has two big consequences:

- **No locks inside the server.** There is no contention on internal data structures, no lock overhead, and no subtle concurrency bugs within a command. Each command on a single key is atomic: nobody can observe a half-applied `INCR`.
- **One slow command blocks everybody.** While Redis spends time on a big `SMEMBERS` or a `DEL` of a huge key, every other client waits. This is why command complexity matters so much, and why Chapter 25 spends time on big keys and the slow log.

Network round trips dominate

For most simple commands, the server-side work is tiny compared to the cost of sending a request over the network and waiting for the reply. That makes **round trips** the resource you most often need to economize. If your code issues a hundred separate `GET` commands in a loop, you pay a hundred round trips. If it uses `MGET` or a pipeline (Chapter 21), it pays roughly one. Many "Redis is slow" complaints turn out to be "my code talks to Redis too many times per request."

> **NOTE**
>
> "Single-threaded" is about command execution within one Redis process. You scale beyond one core by running more instances: replicas for reads, or Redis Cluster to shard data across several primaries (Chapter 27). Plenty of production deployments never need that, but it is good to know the ceiling exists and where it comes from.

A First Look at Redis in Action

Before going further, it helps to see a handful of commands. Chapter 2 covers setup; for now, just read along. The prompt `127.0.0.1:6379>` is `redis-cli` connected to a local server on the default port.

redis-cli session

```
127.0.0.1:6379> SET pg:menu:42 '{"id":42,"dishes":["d:101","d:102"]}' EX 300
OK
127.0.0.1:6379> GET pg:menu:42
"{\"id\":42,\"dishes\":[\"d:101\",\"d:102\"]}"
127.0.0.1:6379> TTL pg:menu:42
(integer) 297
127.0.0.1:6379> HSET pg:cart:1001 d:101 2 d:102 1
(integer) 2
127.0.0.1:6379> HINCRBY pg:cart:1001 d:101 1
(integer) 3
127.0.0.1:6379> HGETALL pg:cart:1001
1) "d:101"
2) "3"
3) "d:102"
4) "1"
127.0.0.1:6379> ZINCRBY pg:lb:dishes:2026-09-27 1 d:101
"1"
127.0.0.1:6379> ZINCRBY pg:lb:dishes:2026-09-27 1 d:201
"1"
127.0.0.1:6379> ZINCRBY pg:lb:dishes:2026-09-27 1 d:101
"2"
127.0.0.1:6379> ZRANGE pg:lb:dishes:2026-09-27 0 2 REV WITHSCORES
1) "d:101"
2) "2"
3) "d:201"
4) "1"
```

In a dozen lines we cached a menu for five minutes, built a shopping cart and incremented the quantity of one item atomically, and maintained a small "trending dishes" leaderboard. Each command did a small, well-defined amount of work on the server. The `TTL` of 297 seconds is simply whatever remained when we asked; your number will differ.

Notice also the key names. Every key begins with `pg:`, followed by an entity and an id. That naming convention is not decoration; Chapter 3 explains why consistent key design matters for operations, debugging, and memory analysis.

Where Redis Fits in a Modern Architecture

A typical web backend has application servers that are stateless (so you can run many of them), a relational database that holds the durable truth, and a collection of supporting services. Redis usually sits in the "supporting services" layer and plays several roles at once. The most common ones are listed below, with the chapters where we treat them in depth.

Role	What Redis provides	Main chapters
Cache	Fast reads of expensive or frequently used data, with TTLs	6–9
Session and cart store	Short-lived per-user state shared across app servers	10
Queue and job broker	Lists and streams for background work	11, 13–15
Messaging	Pub/sub fan-out for live updates	12, 20
Rate limiter	Atomic counters with expiration	16
Leaderboards and counters	Sorted sets, HyperLogLog, bitmaps	5, 17
Live location	Geospatial indexes and presence keys	18
Coordination	Locks, semaphores, leader election	19

Some teams run one Redis deployment for all of these roles; others separate a cache instance that may evict freely from an instance for queues and locks where eviction would be a disaster (Chapters 9 and 28).

Redis versus a relational database

It is tempting to frame Redis and PostgreSQL as competitors. In practice they are partners with different strengths. The relational database offers durable transactions across many rows, constraints, joins, flexible ad-hoc queries, and mature backup tooling. Redis offers very cheap per-operation cost, rich in-memory structures, expirations on keys, blocking operations for queues, and pub/sub. The data that goes into Redis is usually one of three kinds:

1. **Derived data** that can be rebuilt from the database: cached menus, restaurant profiles, precomputed leaderboards. Losing it costs performance, not correctness.
2. **Ephemeral data** that only matters for a short time: sessions, rate-limit counters, courier heartbeats, locks. Losing it is usually tolerable or self-healing.
3. **In-flight data** that is on its way somewhere: jobs in a queue, events in a stream. Losing it may matter, which is why Chapters 11 and 13–14 spend so much time on reliability and why persistence settings matter for those deployments.

A good first question for every piece of data you consider putting in Redis is: *which of these three kinds is it, and what happens if it disappears?* The answer determines TTLs, eviction settings, persistence configuration, and whether you need a fallback path to the database.

Redis versus an in-process cache

Another common comparison is a cache inside the application process, such as a dictionary with an LRU policy. An in-process cache avoids the network entirely, but every application instance holds its own copy, invalidation is harder, and the cache is lost on every deploy. Redis gives you one shared copy that all instances see and that survives application restarts. Many systems use both, and Redis supports that combination directly with client-side caching (Chapter 10).

Redis versus a message broker

Redis lists and streams can serve as a job queue or event log, and for many applications that is the right amount of machinery. Dedicated brokers such as Kafka or RabbitMQ aim at different goals, such as disk-first retention of very large histories or sophisticated routing. Chapters 13 through 15 give you enough detail to make that call deliberately.

The Redis Landscape: Versions, Licenses, and Valkey

A book about Redis written in 2026 has to address something that was not an issue a few years ago: "Redis" now refers to more than one thing, and licensing matters for some organizations. We will state the facts carefully and without taking sides.

- **Licensing changes.** For most of its history Redis was released under the permissive BSD license. With Redis 7.4 (2024), the license of Redis changed to a dual license of RSALv2 and SSPLv1. With Redis 8 (2025), AGPLv3 was added as an additional license option. What these licenses permit is a legal question for your organization; if it matters to you, read the license texts and consult your legal team.
- **Valkey.** After the 7.4 license change, the Linux Foundation started Valkey, a fork created from Redis 7.2.4 and kept under the BSD license. Valkey is broadly compatible with Redis 7.2 commands and client libraries, and it has since developed on its own path.
- **Integrated capabilities in Redis 8.** Redis 8 integrated into Redis Open Source several capabilities that previously shipped as separate modules, such as JSON, the query engine for search and secondary indexing, time series, and probabilistic data types. It also introduced vector sets, a new feature whose status you should check in the current documentation. On Redis 7.x these capabilities required Redis Stack or separately loaded modules. Valkey offers some comparable features through its own modules; check the Valkey documentation for what is available.
- **Managed services.** Cloud providers offer managed Redis-compatible services. Depending on the provider and the options you pick, the engine underneath may be Redis Open Source, Valkey, or a provider-specific engine that speaks the Redis protocol. Feature availability and version support differ, so check your provider's documentation.

For the purposes of this book, the good news is that the core model—keys, strings, hashes, lists, sets, sorted sets, streams, pub/sub, expiration, transactions, Lua—works the same way across Redis and Valkey. Where we use a command introduced in a specific version, we say so, and where a feature exists only in Redis 8 or only through modules, we call that out explicitly. Chapter 2 shows how to run either server locally.

> **TIP**
>
> Record which engine and version each environment runs. A command that works on your laptop's Redis 8 may not exist on a staging cluster pinned to an older version, or on a managed service that exposes a different engine. `INFO server` tells you the server version in a few keystrokes; making it part of your deployment checklist saves confusing debugging sessions.

Meet Plattergo

Abstract examples make for forgettable books. Throughout this book we build and operate a single fictional system called **Plattergo**, a food-ordering and delivery platform running in three cities: Lisbon, Austin, and Dublin. Every chapter uses the same restaurants, customers, couriers, orders, and key names, so that by the end you can picture the entire platform and how each Redis feature supports it.

The product

Plattergo customers browse restaurants and menus in a mobile app, fill a cart, place an order, and then watch a courier move across a map until their food arrives. Restaurants receive new orders on a tablet in the kitchen and update the order as it is accepted, prepared, and handed off. Couriers run a courier app that reports their GPS position every few seconds and receives delivery assignments.

Every order moves through a well-defined set of statuses:

```
placed -> accepted -> preparing -> ready -> picked_up -> out_for_delivery -> delivered
   \
    +-> cancelled   (possible from several earlier states)
```

The services

Plattergo's backend is a small set of services, each with a clear job:

- `api` serves the customer and restaurant apps: browsing, carts, checkout, order status.
- `dispatch` assigns couriers to orders that are ready or nearly ready.
- `notifier` sends push notifications, SMS, and email.
- `tracker` ingests courier locations and answers "where is my courier?"
- `worker` runs background jobs: receipts, reports, cache warming, cleanup.

The system of record is PostgreSQL. It stores restaurants, menus, customers, orders, payments, and courier records durably. Redis sits beside it, and here is a simplified picture of the architecture:



How Plattergo uses Redis

Almost every major Redis feature has a natural home in Plattergo. This is a quick tour; each item is a chapter's worth of detail later.

- **Caching.** Restaurant pages and menus are read far more often than they change. The `api` service caches the restaurant profile in a hash, `pg:restaurant:42`, and the full menu document as a JSON string, `pg:menu:42`, with a TTL of 300 seconds. Chapters 6–8 cover how to populate, invalidate, and protect these caches.
- **Sessions and carts.** A logged-in customer has a session hash such as `pg:customer:1001:session:{token}` with a sliding 30-minute TTL, and a cart hash `pg:cart:1001` where each field is a dish id and each value a quantity.
- **Hot order state.** While an order is active, its current status and a few key fields live in `pg:order:9001`, so the tracking screen can poll or subscribe without hammering the database.
- **Queues and streams.** New orders become jobs and events. A simple list, `pg:orders:new`, illustrates basic queues in Chapter 11; the production design uses a stream, `pg:stream:orders`, with consumer groups for `dispatch` and `notifier`.
- **Live updates.** The customer's app receives status changes on the pub/sub channel `pg:channel:order:9001`, relayed to the device over WebSockets or server-sent events.
- **Courier tracking.** Courier positions go into a geospatial index per city, such as `pg:courier:geo:lisbon`, and each courier's online status is a heartbeat key, `pg:courier:77:presence`, that expires after 30 seconds without an update.
- **Analytics.** Trending dishes per day live in `pg:lb:dishes:2026-09-27`; unique visitors per restaurant page go into a HyperLogLog; daily active customers are bits in `pg:active:2026-09-27`.

- **Protection and coordination.** Rate limits per customer use keys like `pg:rl:customer:1001`. Assigning a courier to order 9001 takes a lock, `pg:lock:assign:9001`, so two dispatchers never assign two couriers to the same order.

The sample data

To keep examples consistent, we use one small data set everywhere. Chapter 2 provides a seed script that loads it into your local Redis.

Restaurant	Id	City	Rating
Casa Verde	42	lisbon	4.7
Smoke & Salt BBQ	43	austin	4.5
Green Bowl Co.	44	dublin	4.8
Tokyo Street Kitchen	45	lisbon	4.6

Dishes use ids of the form `d:NNN`: `d:101` Bacalhau Bowl (12.50) and `d:102` Piri-Piri Chicken (14.00) from Casa Verde, `d:201` Brisket Plate (18.00) from Smoke & Salt BBQ, `d:301` Harvest Bowl (11.00) from Green Bowl Co., and `d:401` Chicken Katsu Don (13.50) from Tokyo Street Kitchen. Our customers are Maya (1001, Lisbon), Leo (1002, Austin), and Aoife (1003, Dublin). Our couriers are Rui (77, Lisbon, bike), Carla (78, Lisbon, scooter), and Jake (91, Austin, car).

Three orders are in flight. Order 9001 is Maya's order from Casa Verde, total 26.50, being delivered by Rui and currently `out_for_delivery`. Order 9002 is Leo's Brisket Plate from Smoke & Salt BBQ, total 18.00, still `preparing`. Order 9003 is Aoife's order from Green Bowl Co., just `placed`.

A Taste of Redis from Python

Application code in this book is Python using the `redis-py` client. Here is a minimal example that mirrors what the `api` service does when a customer opens Casa Verde's page. It is intentionally simple; Chapter 6 turns it into a robust cache-aside implementation.

first_look.py

```
import json
import redis

r = redis.Redis(host="localhost", port=6379, decode_responses=True)

def load_menu_from_db(restaurant_id: int) -> dict:
    # Stand-in for a PostgreSQL query.
    return {
        "restaurant_id": restaurant_id,
        "dishes": [
            {"id": "d:101", "name": "Bacalhau Bowl", "price": 12.50},
            {"id": "d:102", "name": "Piri-Piri Chicken", "price": 14.00},
        ],
```

```
    }

def get_menu(restaurant_id: int) -> dict:
    key = f"pg:menu:{restaurant_id}"
    cached = r.get(key)
    if cached is not None:
        return json.loads(cached)          # cache hit: one round trip
    menu = load_menu_from_db(restaurant_id)
    r.set(key, json.dumps(menu), ex=300)   # cache for 5 minutes
    return menu

def add_to_cart(customer_id: int, dish_id: str, qty: int = 1) -> int:
    # HINCRBY is atomic: two taps from two devices never lose an update.
    return r.hincrby(f"pg:cart:{customer_id}", dish_id, qty)

if __name__ == "__main__":
    print(get_menu(42)["dishes"][0]["name"])
    print(add_to_cart(1001, "d:101"))
```

The `decode_responses=True` argument tells `redis-py` to decode replies into Python strings instead of returning raw bytes. We use it throughout the book because it keeps examples readable; for binary payloads you would leave it off.

Two things in this small program are worth pausing on. First, the cache hit path costs exactly one round trip to Redis and no database query, which is the whole point. Second, `add_to_cart` does not read the cart, modify it in Python, and write it back. It sends `HINCRBY`, and the server does the arithmetic atomically. If Maya taps "add" on her phone and her tablet at the same moment, both increments land. A read-modify-write in application code would have a race where one update can overwrite the other.

Atomicity: The Quiet Superpower

The cart example leads to one of the most important ideas in this book. Because commands execute one at a time, **each individual command is atomic**. That gives you a toolkit of race-free building blocks:

- `INCR` and `HINCRBY` for counters, rate limits, and inventory.
- `SET key value NX` ("set if not exists") for locks and idempotency markers.
- `SADD`, which returns whether the member was new, for de-duplication.
- `LMOVE` (Redis 6.2 and later) for moving a job from a queue to an in-progress list in one step.
- `ZADD ... GT` and similar conditional flags for "only update if higher."

When one command is not enough, Redis offers larger atomic units. `MULTI` / `EXEC` transactions queue commands and execute them together without interleaving other clients' commands, though with an important caveat: if one command fails at runtime (for example, applying a list command to a string), the others still run—there is no rollback. Lua scripts and Redis Functions (Redis 7.0 and later) run a

whole piece of logic atomically on the server, blocking other clients while they execute. Chapters 21 and 22 explore these in detail, including when their atomicity helps and when a long script becomes a liability.

> **WARNING**
> Atomicity inside Redis does not extend to your database. If your code updates PostgreSQL and then updates Redis, a crash between the two leaves them inconsistent, and no Redis feature can prevent that on its own. Designing for this gap—with TTLs, idempotent updates, events, and reconciliation—is the subject of Chapter 7. Assume from day one that the cache can be stale or empty.

Durability and Availability in One Page

Because data lives in memory, a natural question is what happens when the process stops. The short answer is that Redis offers configurable persistence, and you should choose it per deployment based on what kind of data that deployment holds.

- **RDB snapshots** write a point-in-time copy of the dataset to disk periodically. They are compact and fast to load, but you can lose writes made since the last snapshot.
- **AOF (append-only file)** logs every write operation. With the commonly used `appendfsync everysec` setting, a crash can lose roughly the last second of writes. Since Redis 7.0 the AOF is stored as multiple files (a base plus incremental files) managed by the server.
- **No persistence** is a legitimate choice for a pure cache that can be rebuilt from the database.

For availability, Redis supports asynchronous replication from a primary to replicas, automatic failover with Redis Sentinel, and sharding with Redis Cluster. Asynchronous replication means a failover can lose writes that the primary acknowledged but had not yet sent to replicas. That is an acceptable trade-off for caches and many real-time features, and something to design around for queues and locks. Chapters 26 and 27 go into the details; for now, keep the principle: *match the durability configuration to the data's importance, and never assume Redis is as durable as your relational database unless you have configured and tested it to be close.*

What Goes Wrong in Production

Redis is easy to start using and easy to misuse. Most production incidents with Redis are not bugs in Redis; they are predictable consequences of design choices. Here are the patterns we will see again and again, so you can recognize them early.

Unbounded growth

Keys created without expiration and never deleted accumulate until memory runs out. Then, depending on configuration, Redis either rejects writes or starts evicting keys, possibly ones you cared about. The fix is discipline: every key should have a clear lifecycle, either a TTL or an explicit owner that deletes it.

Big keys and O(N) commands

A single list with millions of elements, a hash that grew to hold an entire table, or a `KEYS *` in production code can block the server long enough to cause timeouts across every service that shares it. Prefer bounded collections, use `SCAN` instead of `KEYS`, and fetch large collections incrementally.

Chatty clients

Hundreds of round trips per request add up even when each one is quick. Batching with `MGET`, `HMGET`-style commands, and pipelines is often the single biggest performance win available.

Treating the cache as the truth

If a code path writes only to Redis and assumes the data will be there later, eviction, a failover, or a restart will eventually prove it wrong. Know which keys are derived, which are ephemeral, and which are in-flight, and configure accordingly.

One instance for everything

Sharing a single Redis between a cache that is allowed to evict and a queue that must not lose jobs forces one eviction policy on both. Separate them, or at least choose the policy consciously.

How to Read This Book

The book has six parts: foundations (Part I), caching (Part II), queues, pub/sub, and streams (Part III), real-time features (Part IV), advanced topics such as scripting, data modeling, and performance (Part V), and production operations (Part VI). If you are new to Redis, read Part I in order; everything after it assumes the commands from Chapters 3 through 5. If you already use Redis, jump to the chapter you need—the shared sample data and key names make every example self-explanatory. We show `redis-cli` first because it makes the commands visible, then the Python equivalent; in other languages only the client syntax changes.

Practice

These exercises need no setup beyond reading; you will be able to run them after Chapter 2.

1. **Classify Plattergo's data.** For each of these keys, decide whether it holds derived, ephemeral, or in-flight data, and what should happen if it vanished: `pg:menu:42`, `pg:courier:77:presence`, `pg:stream:orders`, `pg:lock:assign:9001`, `pg:cart:1001`.