DATABASES

Redis

Core Data Structures & Caching Patterns

Pub/Sub & Queues · Persistence & Durability

Replication, Sentinel & Cluster · Capstone Project

Philip Osztromok

Generated with Claude

Table of Contents

Redis — 10 Chapters

⚡ What is Redis & Why Use It

Redis has already been mentioned three times on this site without ever being taught directly: Node.js Advanced's caching chapter (`node3-5`) and message-queue chapter (`node3-6`), and the WebSockets course's scaling chapter (`web-sockets1-7`). This course is the deep dive those three chapters gestured at — a fourth database engine alongside MySQL (relational) and MongoDB (document), built around a genuinely different trade-off: speed over durability and query flexibility.

What Is Redis?

Redis (**RE**mote **DI**ctionary **S**erver) is an **in-memory data structure store** — data lives primarily in RAM rather than on disk, which is what makes Redis operations dramatically faster than a typical disk-backed database query. It's used interchangeably as a database, a cache, and a message broker, depending on which of its features a given application leans on.

In-Memory, Not Durable-by-Default

MySQL and MongoDB are built around a core promise: once a write is acknowledged, it's safely on disk and will survive a crash. Redis inverts the default: data lives in memory first, and disk persistence (Chapter 8's subject) is a configurable option layered on top, not an automatic guarantee baked into every write. That's not a flaw — it's a deliberate trade-off that buys Redis its speed.

MySQL/MongoDB vs. Redis

> **MySQL / MongoDB**
>
> Disk-first, durable by default, rich query capabilities (joins, aggregation pipelines) — built to be the system of record.

> **Redis**
>
> Memory-first, durability optional and configurable, simple but extremely fast data structures — built to be fast, not to replace the system of record.

Key Use Cases

Use Case	Covered In
Caching	Chapter 5 (deepens `node3-5`)
Session storage	Chapter 3 (data structures)
Pub/Sub messaging	Chapter 6 (deepens `web-sockets1-7`)
Queues	Chapter 7 (deepens `node3-6`)
Rate limiting	Chapter 10 (capstone)

Where Redis Already Appears On This Site

Existing Mention	What It Said	Where This Course Goes Deeper
`node3-5` (caching)	Brief mention of "Redis patterns" for caching	Chapter 5 — cache-aside pattern, TTLs, cache stampede prevention
`node3-6` (message queues)	Brief mention of "Redis Streams"	Chapter 7 — list-based queues vs. Streams, consumer groups
`web-sockets1-7` (scaling)	Brief mention of a "Redis pub/sub adapter"	Chapter 6 — `PUBLISH`/`SUBSCRIBE` from first principles

A First Glimpse: redis-cli

Chapter 2 covers installation and the CLI properly — this is just a preview of what "using Redis" looks like:

```
127.0.0.1:6379> SET greeting "hello"
OK
127.0.0.1:6379> GET greeting
"hello"
```

A key, a value, set and read back — no schema, no table definition, no query language beyond simple commands.

Redis Tends to Fit

Data that's read constantly and can tolerate being slightly stale or occasionally lost (a cache), short-lived data with a natural expiration (sessions, rate-limit counters), and messaging patterns that don't need a full message-queue product.

MySQL/MongoDB Still Win

Data that must survive a crash with zero loss, complex relational queries or aggregation, and anything that's the actual system of record rather than a fast layer sitting in front of one.

💻 Coding Challenges

Challenge 1: In-Memory vs. Disk-First

Explain, in your own words, why Redis being "in-memory first" makes it faster than MySQL for simple key lookups, and what Redis gives up to get that speed.

Goal: Practice connecting the architectural trade-off (memory vs. disk) to its practical consequence (speed vs. durability).

→ Solution

Challenge 2: Redis or MySQL?

For each, recommend Redis or MySQL and justify: (a) storing a user's shopping cart contents during an active session, (b) storing completed order records for a business's permanent financial history.

Goal: Practice applying this chapter's "when each fits" section to concrete scenarios.

→ Solution

Challenge 3: Trace the Site's Three Mentions

Name the three existing places on this site that mention Redis, what each one said about it, and which chapter of this course will go deeper on each.

Goal: Practice locating and connecting cross-references across the site's existing course material.

→ Solution

⚠️ Gotcha: Redis Is a Complement, Not a Replacement

A common misconception coming from a MySQL/MongoDB background is treating Redis as a candidate to *replace* the primary database. In the overwhelming majority of real architectures, Redis sits *alongside* MySQL or MongoDB — a fast cache in front of it, a session store next to it, a queue or pub/sub layer between services — not instead of it. The system of record stays exactly where it was; Redis takes on the specific jobs (speed-critical reads, short-lived state, messaging) that a disk-first database isn't optimized for.

🎯 What's Next

The next chapter gets hands-on: **Installing Redis & redis-cli** — getting a real Redis instance running (including a Docker shortcut) and connecting to it with the `redis-cli` shell.

🛠️ Installing Redis & redis-cli

Chapter 1's `SET`/`GET` preview assumed a running Redis instance. This chapter gets one actually running, connects to it with the `redis-cli` shell, and connects to it from real application code — the same three-step arc `mongodb1-2` followed for MongoDB.

Installing Redis

Three common paths, in order of how likely you are to reach for each:

```
# Docker — fastest path, no system install needed
docker run --name redis-dev -p 6379:6379 -d redis

# Native install — Debian/Ubuntu
sudo apt install redis-server

# Native install — macOS (Homebrew)
brew install redis
```

A managed cloud Redis offering (many cloud providers offer one) is the production-equivalent path — same protocol, no server to maintain — but local development almost always starts with Docker or a native install.

The redis-cli Shell

`redis-cli` is Redis's own interactive shell — conceptually identical to `mongosh` for MongoDB or the `mysql` client for MySQL.

```
redis-cli
127.0.0.1:6379> PING
PONG
```

`PING` returning `PONG` is the simplest possible confirmation the server is up and reachable.

redis-cli Basics

```
127.0.0.1:6379> SET greeting "hello"
OK
127.0.0.1:6379> GET greeting
"hello"
127.0.0.1:6379> EXISTS greeting
(integer) 1
127.0.0.1:6379> TYPE greeting
string
127.0.0.1:6379> DEL greeting
(integer) 1
```

Command	Purpose
`PING`	Confirms the server is reachable
`SET` / `GET`	Store and retrieve a value by key
`EXISTS`	Checks whether a key exists (returns 1 or 0)
`TYPE`	Reports which data structure a key holds (Chapter 3)
`DEL`	Deletes a key
`FLUSHALL`	Deletes *every* key in every database — use with real caution

Connecting From an Application

Real applications don't shell out to `redis-cli` — they use a client library, the same pattern `mongodb1-8` established for MongoDB's Node.js driver. The most common Node.js options are `node-redis` (the official client) and `ioredis`.

```
import { createClient } from 'redis';

const client = createClient();
await client.connect();

await client.set('greeting', 'hello');
const value = await client.get('greeting');
console.log(value); // "hello"
```

redis-cli vs. a Client Library

redis-cli

An interactive shell for exploring data and running one-off commands by hand — the same role `mysql`/`mongosh` play for their engines.

Client Library (node-redis, ioredis)

What real application code actually uses — the same underlying protocol, wrapped in an API a program calls programmatically.

💻 Coding Challenges

Challenge 1: Confirm a Running Instance

Using `redis-cli`, write the command to confirm a Redis server is reachable, and explain what response confirms success.

Goal: Practice the most basic connectivity check before attempting any real commands.

→ Solution

Challenge 2: Set, Check, and Remove a Key

Write the sequence of `redis-cli` commands to: set a key `session:42` to the value `"active"`, confirm it exists, check its type, and then delete it.

Goal: Practice chaining the basic commands from this chapter's reference table in a realistic sequence.

→ Solution

Challenge 3: CLI or Client Library?

For each, say whether `redis-cli` or a client library (like `node-redis`) is the right tool: (a) quickly checking what's currently cached during debugging, (b) a running Express application reading a cached value on every request.

Goal: Practice applying this chapter's CLI-vs-library distinction to realistic scenarios.

→ Solution

⚠️ Gotcha: Never Run KEYS * Against a Production Instance

Redis is fundamentally **single-threaded** for command execution — one command runs to completion before the next one starts, which is part of why individual commands are so fast and predictable. `KEYS *` (or any broad `KEYS` pattern) scans *every* key in the database in one blocking operation — on a database with millions of keys, this can freeze the entire server for every other client for seconds, all at once. The safe alternative, covered properly once it becomes relevant, is `SCAN`, which walks the keyspace incrementally without blocking. For now: reach for `KEYS` only in a small local development database, never against anything resembling production.

🎯 What's Next

The next chapter covers Redis's core data structures in depth: **Strings, Lists, and Hashes** — `SET`/`GET`/`INCR`, `LPUSH`/`RPUSH`/`LRANGE`, and `HSET`/`HGET`/`HGETALL`.

🧱 Core Data Structures I: Strings, Lists, Hashes

Chapter 2's `SET`/`GET` only scratched Redis's simplest data type. Redis is really a *data structure* store — strings, lists, hashes, and (Chapter 4) sets and sorted sets — each suited to a different shape of data, all sitting behind the same simple command style.

Strings: More Than Just SET/GET

A Redis **string** holds a single value — but beyond plain `SET`/`GET`, Redis provides `INCR`/`DECR` for **atomic** increments: safe to call from many concurrent clients at once with no race condition, since the increment happens as one indivisible server-side operation.

```
127.0.0.1:6379> SET page:home:views 0
OK
127.0.0.1:6379> INCR page:home:views
(integer) 1
127.0.0.1:6379> INCR page:home:views
(integer) 2
```

Two concurrent requests both calling `INCR` at the exact same instant will still produce two distinct, correct results (e.g. 1 and 2, never both landing on 1) — a guarantee that would need explicit locking to replicate safely in most other systems.

Lists: Ordered Collections

A Redis **list** is an ordered sequence of values — `LPUSH` inserts at the **head** (left/beginning), `RPUSH` inserts at the **tail** (right/end), and `LRANGE` reads a range of elements.

```
127.0.0.1:6379> RPUSH recent:activity "logged in"
(integer) 1
127.0.0.1:6379> RPUSH recent:activity "viewed product 42"
(integer) 2
127.0.0.1:6379> LRANGE recent:activity 0 -1
1) "logged in"
2) "viewed product 42"
```

`LRANGE key 0 -1` is the standard idiom for "give me the whole list" — `0` is the first element, `-1` refers to the last element regardless of the list's length.

Hashes: Field-Value Pairs Within One Key

A Redis **hash** groups multiple field-value pairs under a single key — the natural fit for something object-like, such as a user profile, without needing a separate top-level key per field.

```
127.0.0.1:6379> HSET user:104 name "Dana" email "dana@example.com"
(integer) 2
127.0.0.1:6379> HGET user:104 name
"Dana"
127.0.0.1:6379> HGETALL user:104
1) "name"
2) "Dana"
3) "email"
4) "dana@example.com"
```

Multiple String Keys vs. One Hash

Multiple Strings

```
SET user:104:name "Dana"
SET user:104:email "dana@..."
```

One Hash

```
HSET user:104 name "Dana" email "dana@..."
```

The hash consolidates related fields under one key — reading, updating, or deleting "this user's data" is one key to manage, not an ever-growing set of loosely related string keys sharing a naming convention.

Command	Structure	Purpose
`SET` / `GET`	String	Store/retrieve a single value
`INCR` / `DECR`	String	Atomic increment/decrement of a numeric value
`LPUSH` / `RPUSH`	List	Insert at the head / tail of an ordered list
`LRANGE`	List	Read a range of elements (`0 -1` = whole list)
`HSET` / `HGET`	Hash	Set/get one field within a hash
`HGETALL`	Hash	Read every field-value pair in a hash

Use a String When

The value is a single, simple piece of data — a counter, a flag, a cached blob of text or serialized JSON.

Use a List When

Order matters and you're modeling a sequence — a recent-activity feed, a simple queue (Chapter 7 goes deeper on this).

Use a Hash When

You're modeling one object with multiple named fields — a user profile, a product record — grouped under a single key.

Coming Up: Sets & Sorted Sets

Chapter 4 covers unique unordered collections and score-ordered collections — leaderboards, tag systems, and more.

💻 Coding Challenges

Challenge 1: Build a View Counter

Write the `redis-cli` commands to initialize a view counter for a page called `page:about` at 0, then increment it three times, showing the value after each increment.

Goal: Practice using `INCR` for a realistic atomic-counter scenario.

→ Solution

Challenge 2: Predict List Order

Starting from an empty list `queue:tasks`, predict the final order of elements after running: `RPUSH queue:tasks "A"`, `RPUSH queue:tasks "B"`, `LPUSH queue:tasks "C"`. Show the `LRANGE` command that would confirm your answer.

Goal: Practice reasoning about how `LPUSH` and `RPUSH` insert relative to each other, not just in isolation.

→ Solution

Challenge 3: Model a Product as a Hash

Write the `HSET` command to store a product with key `product:200` having fields `name` ("Wireless Mouse"), `price` ("24.99"), and `category` ("electronics"), then the command to read all of it back.

Goal: Practice choosing a hash over separate string keys for a naturally object-shaped record.

→ Solution

⚠️ Gotcha: LPUSH and RPUSH Together Reverse Relative Order

`RPUSH` appends to the end, in the order called — repeated `RPUSH` calls preserve the order they were issued in. `LPUSH`, by contrast, inserts at the **head** each time — so repeated `LPUSH` calls actually end up in **reverse** order relative to the sequence they were called in, since each new element pushes the previous head further right. Mixing `LPUSH` and `RPUSH` on the same list without tracking this carefully is a common source of "why is my list in the wrong order" bugs — always be deliberate about which end a given piece of code pushes to, and check with `LRANGE` rather than assuming.

🎯 What's Next

The next chapter covers **Sets & Sorted Sets** — `SADD`/`SINTER` for unique unordered collections, and `ZADD`/`ZRANGE` for score-ordered collections, including the leaderboard and sliding-window-counter patterns.

🎯 Core Data Structures II: Sets & Sorted Sets

Chapter 3 covered Strings, Lists, and Hashes. Two more shapes complete Redis's core toolkit: **Sets**, for unique unordered collections, and **Sorted Sets**, for unique collections ordered by a numeric score — the structure behind two of Redis's most famous real-world patterns.

Sets: Unique, Unordered Collections

A Redis **set** holds unique members with no defined order — adding the same value twice has no effect, and duplicates are automatically rejected.

```
127.0.0.1:6379> SADD post:55:tags "redis" "databases" "redis"
(integer) 2
127.0.0.1:6379> SMEMBERS post:55:tags
1) "redis"
2) "databases"
```

Note the return value: `2`, not `3` — the second `"redis"` was silently ignored as a duplicate, exactly what a set is for.

`SINTER` computes the **intersection** of multiple sets — useful for "which members appear in both of these collections":

```
127.0.0.1:6379> SADD likes:redis alice bob carol
(integer) 3
127.0.0.1:6379> SADD likes:databases bob carol dave
(integer) 3
127.0.0.1:6379> SINTER likes:redis likes:databases
1) "bob"
2) "carol"
```

Sorted Sets: Ordered by Score

A **sorted set** combines a set's uniqueness with an explicit numeric **score** per member — Redis maintains the members in score order automatically, regardless of insertion order.

```
127.0.0.1:6379> ZADD scores 42 "alice"
(integer) 1
127.0.0.1:6379> ZADD scores 91 "bob"
(integer) 1
127.0.0.1:6379> ZRANGE scores 0 -1 WITHSCORES
1) "alice"
2) "42"
3) "bob"
4) "91"
```

`ZRANGE` returns members in **ascending** score order regardless of the order they were added — `alice` (added first) still comes before `bob` because 42 < 91, not because of insertion order.

The Leaderboard Pattern

A sorted set is the textbook fit for a game leaderboard: score as the ranking value, `ZREVRANGE` for highest-first, `ZSCORE` to look up one player, `ZRANK` for their position.

```
127.0.0.1:6379> ZADD leaderboard 1500 "player1" 2200 "player2" 800 "player3"
(integer) 3
127.0.0.1:6379> ZREVRANGE leaderboard 0 2 WITHSCORES
1) "player2"
2) "2200"
3) "player1"
4) "1500"
5) "player3"
6) "800"
127.0.0.1:6379> ZSCORE leaderboard "player1"
"1500"
```

The Sliding-Window Counter Pattern

Sorted sets also power a classic real-world Redis pattern: rate limiting via a **sliding window**. Each request is added with the current timestamp as its score; old entries outside the window are pruned; the remaining count is the request count within that window — the exact mechanism behind Chapter 10's capstone.

```
127.0.0.1:6379> ZADD ratelimit:user104 1720000000 "req1"
(integer) 1
-- prune anything older than 60 seconds before "now"
127.0.0.1:6379> ZREMRANGEBYSCORE ratelimit:user104 -inf 1719999940
(integer) 0
-- count how many requests remain within the window
127.0.0.1:6379> ZCARD ratelimit:user104
(integer) 1
```

Set vs. Sorted Set

Set

Unique members, no order at all — good for membership tests and intersections (tags, "has this user done X").

Sorted Set

Unique members, ordered by an explicit score — good for rankings, leaderboards, and time-windowed data.

Command	Structure	Purpose
`SADD` / `SMEMBERS`	Set	Add members; list all members
`SINTER`	Set	Intersection of multiple sets
`ZADD`	Sorted Set	Add a member with a numeric score
`ZRANGE` / `ZREVRANGE`	Sorted Set	Read members in ascending / descending score order
`ZSCORE` / `ZRANK`	Sorted Set	Look up one member's score / rank position
`ZREMRANGEBYSCORE`	Sorted Set	Remove members whose score falls in a range (pruning old entries)
`ZCARD`	Sorted Set	Count members in a sorted set

💻 Coding Challenges

Challenge 1: Find Shared Tags

Two posts have tags stored as sets: `post:1:tags` = {redis, databases, nosql} and `post:2:tags` = {redis, sql, databases}. Write the command to find tags shared by both posts.

Goal: Practice `SINTER` for a realistic "what do these two collections have in common" scenario.

→ Solution

Challenge 2: Build a Mini Leaderboard

Add three players to a sorted set called `game:leaderboard` with scores 300, 750, and 500. Write the command to retrieve them highest-score-first, with scores shown.

Goal: Practice `ZADD` and `ZREVRANGE WITHSCORES` together for the standard leaderboard read.

→ Solution

Challenge 3: Explain the Sliding-Window Pattern

Explain, in your own words, why a sorted set with timestamps as scores is a good fit for rate limiting, and what role `ZREMRANGEBYSCORE` plays in keeping the count accurate over time.

Goal: Practice explaining the mechanics behind this chapter's rate-limiting pattern, not just running the commands.

→ Solution

⚠️ Gotcha: Equal Scores Break Ties Lexicographically

When two members in a sorted set have the **exact same score**, Redis doesn't leave their relative order undefined — it breaks the tie by comparing the members themselves in lexicographic (dictionary) order. This is easy to overlook until it produces a surprising leaderboard: two players tied at the same score will always appear in the same relative order (by name), not in insertion order or randomly, every time the sorted set is read. If tie-breaking by name isn't the desired behavior, the score itself needs to encode enough information to break ties meaningfully (e.g. combining a primary score with a secondary tiebreaker value).

🎯 What's Next

The next chapter is **Expiration & Caching Patterns** — `TTL`/`EXPIRE`, the cache-aside pattern, and cache stampede prevention, deepening `node3-5`'s brief mention of Redis caching.

⏱️ Expiration & Caching Patterns

`node3-5` mentioned "Redis patterns" for caching without ever teaching them. This chapter is that deep dive: how keys expire on their own, the cache-aside pattern nearly every Redis-backed cache uses, and the well-documented failure mode — cache stampede — that catches teams who skip past it.

TTL & EXPIRE

Any key can be given a **time to live** — after which Redis deletes it automatically, with no manual cleanup code required.

```
127.0.0.1:6379> SET session:abc123 "user-104" EX 3600
OK
127.0.0.1:6379> TTL session:abc123
(integer) 3598
127.0.0.1:6379> PERSIST session:abc123
(integer) 1
```

`SET ... EX 3600` sets the value and a 3600-second (1 hour) expiration in one command. `TTL` reports the remaining seconds. `PERSIST` removes the expiration entirely, making the key permanent again. Once a key's TTL reaches zero, Redis removes it on its own — no cron job, no scheduled cleanup.

The Cache-Aside Pattern

Cache-aside is the standard shape nearly every Redis cache follows: check Redis first; on a hit, return immediately; on a miss, query the real database, then store the result in Redis with a TTL so the next request hits the cache instead.

```
async function getProduct(id) {
  const cached = await redis.get(`product:${id}`);
  if (cached) return JSON.parse(cached); // cache hit
const product = await db.query('SELECT * FROM products WHERE id = ?', [id]); // cache miss
await redis.set(`product:${id}`, JSON.stringify(product), { EX: 300 });
  return product;
}
```

The database is only ever queried on a genuine miss — every subsequent call within the 300-second TTL is served entirely from Redis, at a fraction of the latency.

Cache Stampede Prevention

A popular key's TTL eventually expires — and if that key is being requested constantly, **many concurrent requests can all miss the cache at the exact same moment**, each one independently querying the database to repopulate it. This is a **cache stampede** (also called "thundering herd" or "dog-piling") — a real, well-documented production incident pattern where a single expiring cache key causes a sudden spike of duplicate load against the database.

No Stampede Protection vs. a Simple Lock

No Protection

100 concurrent requests all miss the cache at once → all 100 hit the database simultaneously to fetch the same thing.

Lock-Based Protection

Only the first request acquires a lock and queries the database; the other 99 wait briefly and then read the now-repopulated cache.

A Simple Lock Using SET NX

`SET key value NX` only succeeds if the key doesn't already exist — a lightweight mutex:

```
async function getProductSafely(id) {
  const cached = await redis.get(`product:${id}`);
  if (cached) return JSON.parse(cached);

  const gotLock = await redis.set(`lock:product:${id}`, "1", { NX: true, EX: 5 });
  if (!gotLock) {
    // someone else is already repopulating — wait briefly, then retry the cache
await sleep(50);
    return getProductSafely(id);
  }

  const product = await db.query('SELECT * FROM products WHERE id = ?', [id]);
  await redis.set(`product:${id}`, JSON.stringify(product), { EX: 300 });
  return product;
}
```

Only the request that successfully sets the lock key queries the database; every other concurrent request backs off and retries the cache shortly after, rather than all piling onto the database at once. Chapter 10's Lua-scripted approach makes this pattern fully atomic.

Command	Purpose
`EXPIRE key seconds`	Sets a TTL on an existing key
`SET key value EX seconds`	Sets a value and a TTL in one command
`TTL key`	Reports remaining seconds before expiration
`PERSIST key`	Removes a key's expiration
`SET key value NX`	Only sets if the key doesn't exist — the basis of a simple lock

Complementary technique: jittering TTLs

If many keys are all set with the exact same TTL at roughly the same time (e.g. a bulk cache-warming job), they tend to expire together — recreating stampede conditions across many keys at once rather than just one. Adding a small random amount ("jitter") to each key's TTL — e.g. 300 seconds plus a random 0–30 seconds — spreads expirations out over time, reducing the odds of many keys going stale simultaneously. This is a lightweight complement to locking, not a replacement for it.

💻 Coding Challenges

Challenge 1: Set a Session With Expiration

Write the `redis-cli` command to store a session token for user 55 that expires after 30 minutes, then the command to check how much time is left.

Goal: Practice the `SET ... EX` shorthand and `TTL` together.

→ Solution

Challenge 2: Trace a Cache-Aside Flow

Walk through what happens, step by step, for two back-to-back calls to `getProduct(42)` using this chapter's cache-aside example — assume the cache starts empty.

Goal: Practice tracing the hit/miss branches of the cache-aside pattern across multiple calls.

→ Solution

Challenge 3: Diagnose a Stampede

A popular product page's cache key expires, and the database sees a sudden 200-query spike within the same second. Diagnose what likely happened and propose a fix using this chapter's material.

Goal: Practice recognizing a cache stampede from its symptoms and applying the lock-based fix.

→ Solution

🎯 What's Next

The next chapter is **Pub/Sub & Scaling Real-Time Apps** — `PUBLISH`/`SUBSCRIBE` from first principles, deepening the WebSockets course's brief mention of a Redis pub/sub adapter (`web-sockets1-7`).

📡 Pub/Sub & Scaling Real-Time Apps

`web-sockets1-7` named a "Redis pub/sub adapter" as the fix for rooms breaking across multiple WebSocket servers, without teaching pub/sub itself. This chapter builds `PUBLISH`/`SUBSCRIBE` from first principles, then shows exactly how it solves that scaling problem.

The Problem: WebSockets Don't Scale Across Servers by Default

A WebSocket connection is tied to **one specific server process**. If a chat application runs on multiple server instances behind a load balancer, a client connected to Server A and a client connected to Server B are, as far as either server is concerned, on entirely separate islands — Server A has no built-in way to know Server B's clients even exist, let alone deliver a message to them. This is exactly the "rooms break across servers" gotcha `web-sockets1-7` flagged without solving.

PUBLISH & SUBSCRIBE

Redis pub/sub is straightforward: a client `SUBSCRIBE`s to a named channel; any client `PUBLISH`es a message to that channel; every currently subscribed client receives it immediately.

```
# Terminal 1 — subscriber
127.0.0.1:6379> SUBSCRIBE chat:room1
Reading messages...

# Terminal 2 — publisher
127.0.0.1:6379> PUBLISH chat:room1 "hello everyone"
(integer) 1

# Terminal 1 immediately receives:
1) "message"
2) "chat:room1"
3) "hello everyone"
```

Pub/sub is **fire-and-forget**: if no one happens to be subscribed at the moment a message is published, that message is simply gone — never delivered, never stored. This is a fundamentally different guarantee from Chapter 7's queues, where messages persist until a consumer actually processes them.

Using Redis Pub/Sub to Bridge Multiple Servers

The fix for `web-sockets1-7`'s scaling problem: every WebSocket server instance also subscribes to a shared Redis channel. When a client sends a chat message, the server it's connected to publishes that message to Redis — and Redis fans it out to *every* subscribed server, including ones the original client never touched.

The Full Flow: Client A → Server B

1. Client A sends a chat message over its WebSocket connection to **Server A**.
2. Server A relays it to its own local clients *and* `PUBLISH`es it to a shared Redis channel, e.g. `chat:room1`.
3. Redis delivers that published message to *every* subscriber — including **Server B**, which also subscribed to `chat:room1` on startup.
4. Server B receives the message from Redis and relays it over its own WebSocket connections to **Client B**, who was never connected to Server A at all.

This is exactly what a "Redis pub/sub adapter" (Socket.IO's own included adapter is a real example) automates under the hood — every server instance subscribes to the same channels, and Redis becomes the shared nervous system connecting servers that otherwise know nothing about each other.

Pub/Sub vs. Queues (Chapter 7)

> **Pub/Sub**
>
> Fire-and-forget, no persistence — a message published with no subscriber listening is lost forever. Right for real-time broadcast where late delivery is meaningless anyway.

> **Queue (Lists / Streams)**
>
> Messages persist until a consumer processes them, even if no consumer is currently connected. Right when a message must eventually be handled, not just broadcast live.

Command	Purpose
`SUBSCRIBE channel`	Listen for messages on a channel
`PUBLISH channel message`	Send a message to every current subscriber of a channel
`UNSUBSCRIBE channel`	Stop listening to a channel
`PSUBSCRIBE pattern`	Subscribe to every channel matching a pattern (e.g. `chat:*`)

⚠️ Gotcha: A Published Message With No Subscriber Is Lost, Not Queued

It's an easy mistake to assume a message published to a channel will still be there when a subscriber connects later — it won't. If Server B's subscription drops (a restart, a network blip) at the exact moment a message is published, that message is simply gone for Server B, with no way to retrieve it afterward. For real-time broadcast where only "right now" matters (a live chat message, a live cursor position), this is fine — but anything that genuinely needs to survive a disconnected consumer needs Chapter 7's queue-based approach instead, not pub/sub.

💻 Coding Challenges

Challenge 1: Subscribe and Publish

Write the `redis-cli` commands needed for one client to subscribe to a channel called `notifications`, and for another client to publish the message `"new order placed"` to it.

Goal: Practice the basic `SUBSCRIBE`/`PUBLISH` pair.

→ Solution

Challenge 2: Trace the Cross-Server Flow

Using this chapter's four-step flow, explain what happens when Client A (connected to Server A) sends a message, and Client B (connected to Server B) is the intended recipient — assume both servers subscribe to the same Redis channel.

Goal: Practice tracing the full publish-and-fan-out sequence that bridges two otherwise-isolated servers.

→ Solution

Challenge 3: Pub/Sub or Queue?

For each, say whether Redis pub/sub or a queue (Chapter 7) is the right fit, and why: (a) broadcasting a live "user is typing" indicator in a chat app, (b) processing a payment that must eventually succeed even if the payment worker was briefly offline.

Goal: Practice applying the fire-and-forget vs. persistent-until-processed distinction to realistic scenarios.

→ Solution

🎯 What's Next

The next chapter is **Redis as a Queue: Lists vs. Streams** — simple list-based queues, Redis Streams, and consumer groups, deepening `node3-6`'s brief mention of Redis Streams.

📬 Redis as a Queue: Lists vs Streams

Chapter 6 drew the line: pub/sub is fire-and-forget, a queue persists until processed. `node3-6` mentioned "Redis Streams" as one queue-like option without teaching it. This chapter covers both real options — Chapter 3's Lists reused as a simple queue, and the more capable Streams structure with consumer groups.

Simple List-Based Queues

Chapter 3's `RPUSH`/`LPUSH` already build an ordered list — reused as a queue, producers `RPUSH` jobs onto one end, and workers `BLPOP` (**blocking** left-pop) from the other, waiting efficiently rather than polling in a loop.

```
# Producer
127.0.0.1:6379> RPUSH jobs "send-email:104"
(integer) 1

# Worker — blocks until a job appears, 0 = wait forever
127.0.0.1:6379> BLPOP jobs 0
1) "jobs"
2) "send-email:104"
```

This is the simplest possible queue — and it has real limits: once `BLPOP` delivers a job, it's **removed** from the list immediately. If the worker crashes before finishing, that job is simply gone — no retry, no record it was ever handed out, and no way for a second worker to notice and pick up the slack.

Redis Streams

A **Stream** is an append-only log — closer to a lightweight Kafka-style log than a simple list. `XADD` appends an entry (auto-assigned a unique, timestamp-based ID); `XREAD` reads entries **without removing them**, meaning multiple independent readers can each track their own position through the same history.

```
127.0.0.1:6379> XADD orders * customer "C104" total "42.50"
"1720000000000-0"
127.0.0.1:6379> XREAD COUNT 10 STREAMS orders 0
1) 1) "orders"
   2) 1) 1) "1720000000000-0"
         2) 1) "customer" 2) "C104" 3) "total" 4) "42.50"
```

The `*` in `XADD` tells Redis to auto-generate the entry's ID. Unlike a list's `BLPOP`, reading a stream doesn't consume the entry — the same data can be read again later, or by a completely different reader.

Consumer Groups

Consumer groups solve the list queue's crash-recovery problem directly: multiple workers cooperatively consume from one stream, each entry delivered to exactly one worker in the group, while the stream itself keeps the full history intact.

```
# Create a consumer group starting from the beginning of the stream
127.0.0.1:6379> XGROUP CREATE orders workers 0

# A worker reads its next assigned entry
127.0.0.1:6379> XREADGROUP GROUP workers worker1 COUNT 1 STREAMS orders >

# The worker acknowledges successful processing
127.0.0.1:6379> XACK orders workers 1720000000000-0
```

If `worker1` crashes *before* calling `XACK`, the entry stays in a **pending** state, visible via `XPENDING` — another worker can reclaim it with `XCLAIM` and finish the job, exactly the crash-recovery guarantee the simple list queue never had.

Lists vs. Streams

List Queue

Simple, fast, no history — once popped, a job is gone forever with no record it existed, no acknowledgment, no crash recovery.

Stream + Consumer Group

Full history retained, multiple cooperating workers, acknowledgment-based reliability, and reclaimable jobs if a worker crashes mid-processing.

Command	Structure	Purpose
`RPUSH` / `BLPOP`	List	Simplest producer/consumer queue
`XADD`	Stream	Append an entry to a stream
`XREAD`	Stream	Read entries without removing them
`XGROUP CREATE`	Stream	Create a consumer group on a stream
`XREADGROUP`	Stream	Read as part of a consumer group — each entry to one worker
`XACK`	Stream	Acknowledge an entry as successfully processed
`XPENDING` / `XCLAIM`	Stream	Inspect / reclaim entries a crashed worker never acknowledged

A Simple List Queue Is Enough When

Jobs are low-stakes and losing one occasionally on a crash is an acceptable trade-off for maximum simplicity — a single-worker background task, for instance.

Reach for Streams + Consumer Groups When

Multiple workers need to share the load reliably, a crashed worker's job must be recoverable, or the message history itself has value (auditing, replay).

💻 Coding Challenges

Challenge 1: Build a Simple Queue

Write the commands for a producer to push two jobs (`"resize-image:1"`, `"resize-image:2"`) onto a list called `image-jobs`, and for a worker to block-pop one of them.

Goal: Practice the basic `RPUSH`/`BLPOP` producer/consumer pair.

→ Solution

Challenge 2: Diagnose a Lost Job

A team uses a simple `RPUSH`/`BLPOP` list queue. A worker crashes mid-processing, and the job it was handling is never completed or retried. Explain why this happened, and what Redis structure would have prevented it.

Goal: Practice connecting the list queue's known limitation to a concrete failure and its fix.

→ Solution

Challenge 3: Design a Consumer-Group Flow

Design the commands for three workers (`worker1`, `worker2`, `worker3`) to cooperatively process entries from a stream called `notifications` as a single consumer group called `notifiers`, including how a worker confirms it finished an entry.

Goal: Practice the full `XGROUP CREATE` → `XREADGROUP` → `XACK` lifecycle for multiple cooperating workers.

→ Solution

🎯 What's Next

The next chapter is **Persistence & Durability** — RDB snapshotting vs. AOF, and when Redis data loss is (and isn't) an acceptable trade-off.

💾 Persistence & Durability

Chapter 1 established that Redis is memory-first, with durability as an optional, configurable layer — not an automatic guarantee. This chapter covers the two real persistence mechanisms Redis offers, and the actual question that decides whether to bother with either: what is this specific data being used *for*?

Why Persistence Is Optional in Redis

By default, a Redis restart or crash loses everything — data lives in RAM, and RAM doesn't survive a power-off. Redis offers **two** distinct persistence mechanisms, each trading off differently between how much data could be lost and how much performance/complexity cost that protection carries.

RDB Snapshotting

RDB writes a compact, point-in-time snapshot of the entire dataset to disk, either on demand (`SAVE`, `BGSAVE`) or automatically at configured intervals.

```
# redis.conf — save a snapshot if at least 1 key changed in 900 seconds
save 900 1

# manually trigger a snapshot in the background (non-blocking)
127.0.0.1:6379> BGSAVE
```

RDB's file is small and restarts fast from it — but anything written *since* the last snapshot is lost if a crash happens in between, a real window of potential data loss.

AOF (Append-Only File)

AOF takes the opposite approach: every write command is logged to a file as it happens, and a restart replays the log to rebuild state. A configurable **fsync policy** trades durability against performance directly.

```
# redis.conf
appendonly yes
appendfsync everysec # fsync to disk roughly once per second — a common middle ground
```

`always` fsyncs on every single write (safest, slowest), `everysec` batches to about once per second (the usual default — at most ~1 second of writes at risk), and `no` lets the OS decide when to flush

(fastest, least safe). AOF's log grows over time and is periodically compacted (`BGREWRITEAOF`) back down to a minimal representation of current state.

RDB vs. AOF

RDB

Compact single-file snapshots, fast restarts, good for backups — but a real data-loss window between snapshots.

AOF

Much smaller data-loss window (as little as ~1 second), larger file, slower restart while replaying the log.

Combining Both

Redis allows enabling **both** mechanisms together — a common real production setup: RDB snapshots for fast restarts and easy backup portability, AOF running alongside for the tighter data-loss window, each covering the other's weak point.

	RDB	AOF
Data-loss window	Since the last snapshot (minutes, typically)	As little as ~1 second (`everysec`)
Restart speed	Fast — loads one compact file	Slower — replays the full log
File size	Small	Larger, needs periodic rewriting
Best for	Backups, fast recovery	Minimizing data loss

When Redis Data Loss Is (and Isn't) Acceptable

The real decision isn't "which persistence setting is best" in the abstract — it's **what is this Redis instance actually being used for**, per Chapter 1's use-case table. A cache (Chapter 5) can be safely rebuilt from the real database after any restart — losing it is a non-event, since Redis was never the source of truth for that data in the first place. A queue or stream (Chapter 7) with no other durable record of its jobs is a different story entirely: losing unacknowledged work on a crash is a real, consequential loss.

Acceptable to Lose

A cache that can be repopulated from MySQL/MongoDB on the next request; session data that can force a re-login without real harm.

Needs Real Persistence

Queue/stream jobs with no other durable record; any case where Redis is genuinely the only copy of data that matters.

⚠️ Gotcha: Assuming Persistence Is On By Default

Coming from MySQL or MongoDB, where durability is automatic, it's an easy mistake to assume Redis is doing something similar unless told otherwise — Chapter 1 named this trade-off explicitly. Neither RDB save points nor AOF are guaranteed to be meaningfully configured out of the box on every install; a default local development setup may have far weaker persistence than production actually needs. Before treating any Redis instance as durable enough for a given use case, check `redis.conf` directly rather than assuming.

💻 Coding Challenges

Challenge 1: Choose a Persistence Strategy

For each, recommend RDB, AOF, both, or neither, and justify: (a) a pure cache in front of MySQL, (b) a Redis Streams-based job queue with no other durable record of pending jobs.

Goal: Practice applying "what is this data used for" to concrete scenarios rather than picking a setting in the abstract.

→ Solution

Challenge 2: Explain an fsync Policy Choice

Explain the difference between `appendfsync always`, `everysec`, and `no`, and why `everysec` is the common default middle ground.

Goal: Practice explaining the durability/performance trade-off each fsync setting represents.

→ Solution

Challenge 3: Diagnose a Surprising Data Loss

A team's Redis-backed job queue lost 10 minutes of unprocessed jobs after an unexpected server restart. They had RDB configured with `save 600 1`. Explain why this happened and what change would reduce the loss window.

Goal: Practice connecting an RDB save-point interval directly to a real observed data-loss window.

→ Solution

🎯 What's Next

The next chapter is **Replication, Sentinel & Cluster** — primary-replica setup, automatic failover, and sharding basics across nodes.

🔗 Replication, Sentinel & Cluster

Chapter 8 protected data against a single node restarting. This chapter protects against a node going down entirely (replication and automatic failover), and scaling beyond what one node can hold (sharding) — the same two goals `mongodb2-5` and `mongodb2-6` covered for MongoDB, achieved with Redis's own distinct tools.

Primary-Replica Replication

A replica continuously receives a stream of write commands from its primary, keeping an up-to-date copy — set up with `REPLICAOF` (the modern name; `SLAVEOF` is the older, still-functional alias).

```
# run on the replica, pointing it at the primary
127.0.0.1:6380> REPLICAOF 10.0.1.10 6379
```

Reads can be offloaded to replicas to spread load, the same idea as MongoDB's read preference — but Redis replication is **asynchronous** by default, meaning a replica can lag slightly behind the primary, the same replication-lag trade-off `mongodb2-5` covered for MongoDB's secondaries.

Automatic Failover With Sentinel

Here's a genuine difference from MongoDB: plain Redis replication has **no built-in automatic failover** — if the primary goes down, replicas don't elect a new primary on their own the way a MongoDB replica set does. **Sentinel** is a separate process (typically run as a small cluster of its own, 3+ instances) that monitors the primary and its replicas, and when it detects the primary is genuinely down, promotes a replica to primary and reconfigures the others automatically.

```
# sentinel.conf
sentinel monitor mymaster 10.0.1.10 6379 2
# "2" = at least 2 Sentinels must agree the primary is down before failover
```

Unlike MongoDB, where election logic is baked into the replica set itself, Redis deliberately separates "replicate the data" (plain replication) from "detect failure and promote automatically" (Sentinel) — an application connecting through Sentinel-aware clients can keep working through a failover without needing to know a promotion even happened.

Redis Cluster

Redis Cluster is Redis's sharding solution: the entire keyspace is divided into **16,384 hash slots**, and each node in the cluster owns a subset of them. A key's slot is determined by hashing

the key (CRC16) — a client that queries the wrong node gets redirected via a `MOVED` response to the node that actually owns that slot.

```
127.0.0.1:6379> CLUSTER INFO
cluster_enabled:1
cluster_known_nodes:6
cluster_size:3
```

Sentinel vs. Cluster — Different Problems

Sentinel

High availability for **one logical dataset** — automatic failover if the primary dies, without splitting the data itself.

Cluster

Horizontal **scaling** — splits the dataset itself across many nodes, each owning a slice, similar in purpose to MongoDB's sharding (`mongodb2-6`).

These solve genuinely different problems and can even complement each other — Redis Cluster has its own built-in failure detection per shard, but a common point of confusion is assuming Sentinel and Cluster are two competing ways to do the same thing, when they're actually answering different questions ("is my one dataset still available?" vs. "can my dataset be bigger than one node?").

Tool	Solves
Plain replication (`REPLICAOF`)	Keeping copies of data in sync; read scaling
Sentinel	Automatic failover when the primary goes down
Cluster	Sharding the dataset itself across multiple nodes

⚠️ Gotcha: Multi-Key Operations Fail Across Cluster Unless Keys Share a Slot

An operation touching multiple keys at once (a transaction, a Lua script, even a plain `MSET` across several keys) only works in Redis Cluster if every key involved hashes to the **same slot** — otherwise Redis refuses the operation, since the keys may live on entirely different nodes. The fix is a **hash tag**: wrapping the part of the key that should determine its slot in `{}`. `user:{104}:profile` and `user:{104}:orders` both hash *only* on `104`, guaranteeing they land on the same node regardless of the rest of the key — the same deliberate design discipline `mongodb2-6` required when choosing a good shard key, just expressed through key naming instead of an upfront schema decision.

💻 Coding Challenges

Challenge 1: Sentinel or Cluster?

A team has a dataset that comfortably fits on one Redis instance but needs to survive that instance failing without manual intervention. Which tool fits, and why isn't the other one relevant here?

Goal: Practice distinguishing the "stay available" problem from the "scale beyond one node" problem.

→ Solution

Challenge 2: Explain Redis's Failover Gap

Explain why plain Redis replication alone doesn't provide automatic failover the way a MongoDB replica set does, and what component fills that gap.

Goal: Practice articulating the specific architectural difference between Redis and MongoDB's approaches to high availability.

→ Solution

Challenge 3: Fix a Cross-Slot Operation

A Cluster deployment fails when running a multi-key operation on `order:5001:items` and `order:5001:total` together. Explain why, and rewrite the keys using a hash tag so the operation succeeds.

Goal: Practice applying the hash-tag fix to a concrete cross-slot failure.

→ Solution

🎯 What's Next

The final chapter is the **Capstone: Building a Rate-Limited, Cached API** — combining caching, rate limiting, pub/sub, and a Lua-scripted atomic operation into one small real application.

🏁 Capstone: Building a Rate-Limited, Cached API

This capstone builds a small, real **Product API** combining caching (Ch.5), an atomic rate limiter built on Chapter 4's sliding-window pattern, and pub/sub-based cache invalidation (Ch.6) — end to end, in one application.

The Application: A Rate-Limited Product API

Requirements: `GET /products/:id` should be cached and rate-limited per client; when a product is updated elsewhere (an admin panel), every running API instance should invalidate its own cached copy immediately, without polling.

Step 1: Caching the Product Endpoint (Chapter 5)

The cache-aside pattern, unchanged from Chapter 5:

```
async function getProduct(id) {
  const cached = await redis.get(`product:${id}`);
  if (cached) return JSON.parse(cached);

  const product = await db.query('SELECT * FROM products WHERE id = ?', [id]);
  await redis.set(`product:${id}`, JSON.stringify(product), { EX: 300 });
  return product;
}
```

Step 2: Rate Limiting With a Lua Script

Chapter 4's sliding-window pattern (prune old entries, then count) has a subtle problem under real concurrency: checking the count and recording the new request are **two separate commands** — two concurrent requests could both check "under the limit" before either one's new entry is recorded, both slipping through when only one should have. The fix is an **atomic Lua script**, run with `EVAL`: Redis executes the entire script as one indivisible unit — no other command, from any other client, can interleave partway through, turning Chapter 2's single-threaded architecture from a caveat into exactly the guarantee this needs.

```
-- rate_limit.lua
local key = KEYS[1]
local now = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local limit = tonumber(ARGV[3])

redis.call('ZREMRANGEBYSCORE', key, '-inf', now - window)
local count = redis.call('ZCARD', key)

if count >= limit then
return 0 -- over the limit
end

redis.call('ZADD', key, now, now)
redis.call('EXPIRE', key, window)
return 1 -- allowed
// calling it from Node.js
const allowed = await redis.eval(rateLimitScript, {
 keys: [`ratelimit:${clientId}`],
 arguments: [String(Date.now()), '60000', '100']
});
if (!allowed) return res.status(429).send('Too Many Requests');
```

Prune, count-check, and record now all happen inside one atomic `EVAL` call — no race window exists between checking the limit and recording the request, regardless of how many requests arrive at the same instant.

Step 3: Invalidating the Cache via Pub/Sub (Chapter 6)

When a product is updated, the updating instance publishes the product's ID to a shared channel; every API instance — including ones that never touched the update request — subscribes and deletes its own local cache entry.

```
// on update
async function updateProduct(id, changes) {
 await db.query('UPDATE products SET ... WHERE id = ?', [id]);
 await redis.publish('product-updates', id);
}

// on every instance, at startup
subscriber.subscribe('product-updates', async (id) => {
 await redis.del(`product:${id}`);
});
```

The next `getProduct(id)` call on any instance after this point sees a genuine cache miss and re-populates from the database — every instance stays consistent with no polling and no shared coordination beyond the channel itself.

Course Concept	Where It's Used in This App
Cache-aside pattern (Ch.5)	`getProduct()` — cache first, database on miss
Sliding-window rate limiting (Ch.4)	The pattern the Lua script implements atomically
Atomic scripting / single-threaded execution (Ch.2)	The Lua script's check-then-record race condition fix
Pub/Sub (Ch.6)	Broadcasting cache invalidation to every running instance

💻 Coding Challenges

Challenge 1: Explain the Race Condition

Explain, concretely, how two concurrent requests could both bypass Chapter 4's non-atomic sliding-window rate limiter, even though each one individually checked the count correctly.

Goal: Practice articulating a check-then-act race condition in terms of the specific commands involved.

→ Solution

Challenge 2: Add a Cache-Warming Step

Modify Step 3's `updateProduct` function so that, instead of just invalidating the cache, it immediately re-caches the fresh product data — avoiding a cache miss on the very next read.

Goal: Practice combining the cache-aside write path with the pub/sub invalidation flow.

→ Solution

Challenge 3: Deploying to Cluster

This app is deployed onto Redis Cluster (Chapter 9). Explain what could break in the rate-limiting Lua script if a client's rate-limit key and cache key don't share a hash tag, and how to fix it.

Goal: Practice applying Chapter 9's hash-tag requirement to a script that only ever touches one key per call versus one that might touch several.

→ Solution

⚠️ Gotcha: Lua Scripts Have the Same Cross-Slot Restriction as Multi-Key Commands

Chapter 9's hash-tag gotcha applies just as much to Lua scripts as to plain multi-key commands: if a script's `KEYS` table ever includes keys that hash to different slots, Redis Cluster rejects the script entirely. This capstone's rate-limit script only ever touches one key per call, so it's safe as written — but extending it (say, to also update a shared global counter) would need every key involved to share a hash tag, exactly the same discipline Chapter 9 required for any multi-key operation.

🎉 Course Complete

That's the full **Redis** course — from the in-memory data model through every core data structure, expiration and caching patterns, pub/sub, queues and streams, persistence, replication/failover/sharding, and finally a real rate-limited, cached, pub/sub-synchronized API combining all of it. Together with MySQL and MongoDB, this completes the site's three-engine database coverage — relational, document, and in-memory.