Engineer

Redis

an in-memory store for things that live briefly

What it is

Redis is a database server that keeps its whole dataset in memory as typed structures and lets many application processes read and change them over the network, one short command at a time. The project describes itself as a data structure server. The freely available edition is called Redis Open Source, and Redis Ltd also sells Redis Cloud and Redis Software under commercial licences.

Every key holds one value of one type. Strings are plain bytes and double as counters, hashes are records of fields and values, lists keep the order in which items arrived, sets hold unique members, and sorted sets rank their members by a score. The list of types runs on to streams, JSON documents, geospatial indexes, time series, vector sets and probabilistic structures such as Bloom filters and HyperLogLog.

Commands work on those types directly. `INCR` adds one to a counter and returns the new value, `SADD` puts members into a set and `SPOP` takes one out, and any key can carry a timeout after which Redis deletes it by itself. Redis calls such a key volatile. When several steps have to happen as one, a Lua script runs inside the server, and Redis guarantees that nothing else runs while it does.

Memory is where the data lives, and disk is optional. Redis can write point-in-time snapshots, known as RDB files, or log every write to an append-only file (AOF) that it replays at startup, or do both, or neither. Replicas copy a primary and can serve reads. A memory ceiling, `maxmemory`, and an eviction policy decide what happens once the data outgrows the space it was given.

The licence changed with Redis 8. From that release on, Redis Open Source is offered under a choice of three licences: RSALv2, SSPLv1 and AGPLv3. The first two are source-available licences, and AGPLv3 is the open source option.

Use the concept in a decision

Put “What is Redis?” to work in a real decision. We can help relate the concept to the system you are building or buying. Finding the queries and the missing indexes behind a system that has got heavier every year, and fixing the ones that matter.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. 30 days of electricity data in 180 ms — 180 ms to aggregate a month of meter readings

30 days of electricity data in 180 ms Get in touch →

Why it matters

A web application rarely runs as one process. Behind a load balancer sit several instances, each with its own memory, and anything kept in that memory — a count of login attempts, a session, a cached page — exists only in the process that happened to handle the request. A limit counted that way can be escaped by landing on another instance. Redis gives all the instances one shared place for that kind of state, fast enough to consult on every request.

The atomic commands matter as much as the speed. Two instances that each read a counter, add one and write it back can both write the same number, and an increment disappears. `INCR` does the read and the write inside the server as one step, so a hundred concurrent requests leave the counter exactly a hundred higher. A Lua script extends the same guarantee to a short sequence: a check and a change that must not be split apart, such as confirming there is still room and taking the place, run as one.

Expiry is the third piece. A rate-limit window, a session or a one-time code ought to disappear when its time is up, and with a timeout on the key the cleanup is the store’s job rather than a scheduled task’s.

The price is durability, and it is worth knowing in numbers before choosing. With the append-only file and its default policy of one fsync per second, a crash can cost about a second of writes. With snapshots alone, which usually run every few minutes, the documentation says you “should be prepared to lose the latest minutes of data”. Replication is asynchronous by default, so a primary can acknowledge a write and stop before any replica has received it. That makes Redis a good home for state that can be rebuilt, or lost without harm, and a poor home for the only copy of an order, a payment or a balance. The design question that matters most is what the application does when Redis is unreachable, or has forgotten something it was told.

What it looks like in practice

22 rate limiters keep their state in Redis on a public directory of companies and professionals, so that a limit holds across every instance instead of living in one process, where it would be trivially escaped. Each limiter is built for the path it protects, because a login attempt, a search and a bulk export are abusive at very different rates. The deciding choice is what happens when Redis is missing: the limiters fail closed, and traffic is refused rather than waved through. That choice has a visible cost. A Redis problem becomes a problem users can see, accepted on purpose, because the alternative is a Redis problem becoming a billing one.

Selling limited stock puts Redis in front of a database as a fast gate: a Redis counter decremented by a Lua script admits about as many buyers as there are units and turns everyone else away in microseconds. The script makes the check and the decrement one step. The gate promises less than it may seem to. Redis replicates asynchronously, so after a failover the new primary may have forgotten some admissions and let a few too many through. A conditional `UPDATE` in the database still decides each sale, and a reconciler resets the gate from the database's number. When units are few and distinct, such as numbered seats, their ids can sit in a Redis set, and `SPOP` hands them out one at a time with no counter to fight over.

The most ordinary role is a plain cache. An Azure estate defined in Bicep builds each environment from the same templates, testing, staging and product alike, and each one has Redis for caching beside a PostgreSQL Flexible Server, the Container Apps that run the api and the website, and Front Door with a WAF policy. Because the cache is declared in the templates with everything else, changing it is a pull request like any other change to an environment.

Where it goes wrong

Most Redis trouble comes from treating a fast, forgetful store as a durable one, or from assuming it remembers what it was told about time.

A key that should expire stays forever. `SET` replaces the value and discards any timeout the key had, so code that refreshes a cached value with a plain `SET` turns a ten-minute entry into a permanent one:

```
> SET session:42 "v1" EX 600
OK
> SET session:42 "v2"
OK
> TTL session:42
(integer) -1
```

Pass the timeout on every write that needs one, or add `KEEPTTL`, available since 6.0.0, to keep the old one. The same leak hides in a counter built from `INCR` followed by a separate `EXPIRE`: when the second command never arrives, the key never expires. Send both inside one `MULTI` and `EXEC` block, or in one Lua script.

Writes start returning errors while reads still work. The data has reached `maxmemory`, and the eviction policy is `noeviction`, or a `volatile-` policy on a dataset where no key has a timeout, which behaves the same way. Choose the policy for the job: an `allkeys-` policy for a pure cache, where any key may go, and a `volatile-` one only when cached keys share an instance with keys that must stay. For that mix, the documentation's advice is to consider two separate instances.

Writes that Redis acknowledged are missing after a restart or a failover. A crash landed inside the fsync window, or a replica was promoted before the write reached it. That is how Redis is designed, so the fix lies in what is kept there. Store anything that must survive in a database with durable commits, put Redis in front of it as a gate or a cache, and reconcile the two from the database's side, as the stock note does.

Every client stalls at the same moment. A long Lua script is running, and Redis blocks all other server activity for the whole of a script's runtime. Keep scripts short and bounded, and keep loops over large collections out of them.

A limiter waves traffic through during an outage. The client code treats an error from Redis as "limit not reached", so the limiter opens exactly when its store is unreachable. Decide the failure mode on purpose and test it by stopping Redis: a limiter guarding a costly path should refuse, as the 22 limiters above do, with the cost that a Redis problem becomes visible to users.

Further reading

- Develop with Redis — the project's starting point: commands to try, client libraries by language, and the data types.
- Redis persistence — snapshots, the append-only file, fsync policies and what each can lose.
- Redis replication — how replicas follow a primary, and why a window for data loss remains.
- Key eviction — `maxmemory`, every eviction policy, and advice on choosing one.
- EXPIRE — key timeouts, and which commands clear them or leave them in place.
- Scripting with Lua — running a script atomically on the server, and what it blocks meanwhile.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is Redis?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/redis/