Redis Commands Cheat Sheet

THE DESI ARCHITECT
desiarchitect.com
Redis 6.2+ syntax

Every command below carries the access pattern that justifies it and a runnable example. Decide the access pattern first, and the data structure almost picks itself.

ACCESS PATTERN	STRUCTURE	TYPICAL CALL
A single value, a TTL, or an atomic counter	**STRING**	`SET session:user:1042 token EX 3600`
An object whose individual fields are read and updated separately	**HASH**	`HINCRBY restaurant:biryani-house total_orders 1`
Insertion order, a recent-N window, or a queue	**LIST**	`LPUSH orders:recent:user:1042 order:98213`
Uniqueness, membership checks, or set math	**SET**	`SISMEMBER user:1042:viewed dish:42`
Scores, ranking, or a Top-N leaderboard	**SORTED SET**	`ZRANGE trending:dishes 0 9 REV`

1. STRING

sessions | counters | flags | cached blobs

Reach for it when: You store one scalar value per key, you need a TTL, or you need an atomic counter.
Do not use it when: The value is a multi-field object and only one field changes at a time. Use a Hash instead.

SET key value	Plain write. Note that it clears any TTL already on the key unless you add KEEPTTL. `SET config:banner "diwali-sale" -> OK`
SET key value EX secs	The default for anything cached. Value and expiry land in one atomic command, so a crash cannot leave a key without a TTL. `SET session:user:1042 "eyJhbGciOi..." EX 3600 -> OK`
SET key value NX EX secs	Write only if the key is absent. This is the basis of a simple lock or a run-once guard such as one welcome email per user. `SET lock:order:98213 worker-7 NX EX 30 -> OK or (nil)`
SET key value KEEPTTL	Refresh the cached value but let the original countdown continue. `SET cache:home:v3 "{...}" KEEPTTL -> OK`
GET key	Read one value. Returns (nil) when the key is missing or has expired. `GET session:user:1042 -> "eyJhbGciOi..."`
MGET key1 key2 ...	Read many keys in a single round trip. Use it instead of a loop of GET calls in application code. `MGET dish:12:views dish:19:views -> 1) "42" 2) "17"`
GETDEL key	Read and consume in one step. Ideal for one-time tokens, OTPs, and password reset codes. Redis 6.2+. `GETDEL otp:9876543210 -> "418293"`
GETEX key EX secs	Read and extend the TTL atomically, which gives you a sliding session. Redis 6.2+. `GETEX session:user:1042 EX 3600 -> "eyJhbGciOi..."`
INCR key	Atomic counter for views, login attempts, or rate limits. No read, modify, write cycle in your application. `INCR dish:butter-chicken:views -> (integer) 1`
INCRBY key n / DECRBY key n	Move a counter by more than one, for example reserving or releasing stock. `DECRBY stock:dish:42 2 -> (integer) 8`
INCRBYFLOAT key n	Accumulate fractional values such as running revenue or a rating sum. `INCRBYFLOAT revenue:2026-08-02 249.50 -> "1874.50"`
STRLEN key	Check how large a cached blob has grown before it becomes a big-key problem. `STRLEN cache:home:v3 -> (integer) 20481`
TTL key	Inspect remaining life. -1 means the key has no expiry, -2 means it does not exist. `TTL session:user:1042 -> (integer) 3574`
PERSIST key	Strip the expiry and make a key permanent, for example when a trial converts to a paid plan. `PERSIST plan:user:1042 -> (integer) 1`

Worked example: session plus view counter

```
SET session:user:1042 "eyJhbGciOiJIUzI1NiJ9" EX 3600 # token with a one hour life
GET session:user:1042 # "eyJhbGciOiJIUzI1NiJ9"
TTL session:user:1042 # (integer) 3597
INCR dish:butter-chicken:views # (integer) 1
INCR dish:butter-chicken:views # (integer) 2
```

PRODUCTION TRAP INCR and EXPIRE are two separate commands. If your app server crashes between them, the counter stays in Redis forever with no TTL and slowly leaks memory. Redis has no built-in atomic INCR-with-TTL, so wrap the pair in MULTI/EXEC or a small Lua script. For a plain value, always prefer SET with EX over SET followed by EXPIRE.

NUANCE A JSON blob in a String is perfectly fine when almost every request reads and writes the whole document. The cost only appears when single fields are updated frequently.

Rule: One scalar value, a TTL, or an atomic counter means String. Do not force a multi-field object into it.

Fixing the non-atomic counter

```
# Atomic rate limiter. Increment, and set the window only on the very first hit.
# This is a single redis-cli command, wrapped here only to fit the page.
EVAL "local c = redis.call('INCR', KEYS[1])
if c == 1 then redis.call('EXPIRE', KEYS[1], ARGV[1]) end
return c" 1 rate:user:1042 60

# The same guarantee with a transaction, one command per line
MULTI
INCR rate:user:1042
EXPIRE rate:user:1042 60
EXEC
```

2. HASH

entity metadata | partial updates | field counters

Reach for it when: One key holds an object and different screens read or update different fields of it.
Do not use it when: Every request reads and rewrites the entire document anyway. A JSON String is simpler there.

HSET key f v [f v ...]	Create or update one or many fields. HMSET is deprecated, so use HSET everywhere. `HSET restaurant:biryani-house name "Biryani House" rating 4.5 is_open 1 -> (integer) 3`
HGET key field	Read exactly one field instead of pulling the whole object over the wire. `HGET restaurant:biryani-house rating -> "4.5"`
HMGET key f1 f2	Read only the fields a given screen needs. A listing card usually needs name and rating, nothing else. `HMGET restaurant:biryani-house name rating -> 1) "Biryani House" 2) "4.5"`
HGETALL key	Read the full object for a detail page. Safe on small hashes, expensive on wide ones. `HGETALL restaurant:biryani-house -> name, rating, is_open, total_orders`
HINCRBY key field n	Atomically bump a counter that lives inside the object, with no read and rewrite of the other fields. `HINCRBY restaurant:biryani-house total_orders 1 -> (integer) 8412`
HINCRBYFLOAT key field n	Same idea for decimal values such as a rating total or an average basket size. `HINCRBYFLOAT restaurant:biryani-house rating_sum 4.5 -> "38211.5"`
HDEL key field	Drop a field that no longer applies, for example a promo banner that has ended. `HDEL restaurant:biryani-house promo_text -> (integer) 1`
HEXISTS key field	Check for a field before deciding to compute or backfill it. `HEXISTS restaurant:biryani-house gst_number -> (integer) 0`
HSETNX key field v	Write a default only if the field is still unset, without clobbering a real value. `HSETNX restaurant:biryani-house currency INR -> (integer) 1`
HLEN / HKEYS / HVALS key	Field count, all field names, or all values. Useful for debugging and for schema drift checks. `HLEN restaurant:biryani-house -> (integer) 4`
HRANDFIELD key n WITHVALUES	Sample random fields, handy for shuffling featured items. Redis 6.2+. `HRANDFIELD restaurant:biryani-house 2 WITHVALUES -> two random field and value pairs`
HSCAN key cursor COUNT n	Walk a large hash in slices instead of blocking the server with HGETALL. `HSCAN restaurant:biryani-house 0 COUNT 50 -> cursor plus a batch of fields`
HEXPIRE key secs FIELDS 1 f	Expire a single field rather than the whole key, for example a temporary surge price. Redis 7.4+ only. `HEXPIRE restaurant:biryani-house 900 FIELDS 1 surge_price -> 1) (integer) 1`

Worked example: restaurant metadata cache

```
HSET restaurant:biryani-house name "Biryani House" rating 4.5 is_open 1 total_orders 8411
HMGET restaurant:biryani-house name rating # listing card, two fields only
HINCRBY restaurant:biryani-house total_orders 1 # a new order arrives
HSET restaurant:biryani-house rating 4.6 # rating update touches one field
HGETALL restaurant:biryani-house # detail page, full object
```

PRODUCTION TRAP The common shortcut is to stringify the whole object and store it under a single key with SET. Now a single rating change forces a read of the full 20KB blob, a parse, a mutation, a re-serialize, and a full write back. One update looks fine. Thousands of restaurants updating continuously push latency from 5ms toward 200ms with no crash, no CPU spike, and no alert. That is the hidden design tax.

NUANCE A Hash gives no per-key TTL granularity below Redis 7.4. If different fields need different lifetimes on an older server, split them into separate keys.

Rule: Multi-field object plus partial updates means Hash. Whole-document access means String.

3. LIST

recent-N feeds | activity logs | simple queues

Reach for it when: Insertion order matters, the newest item goes on top, or you need a producer and consumer queue.
Do not use it when: You need ranking by score, or you frequently search and remove from the middle. That is O(N).

LPUSH key value	Push onto the head so the newest entry is first. This is the standard write for a recent activity feed. `LPUSH orders:recent:user:1042 order:98213 -> (integer) 21`
RPUSH key value	Push onto the tail. Use it when the list is a FIFO job queue rather than a feed. `RPUSH queue:invoices job:5512 -> (integer) 4`
LTRIM key 0 19	Bound the list immediately after every push so it can never grow without limit. `LTRIM orders:recent:user:1042 0 19 -> OK`
LRANGE key 0 19	Read a window. Reading from the head stays cheap even on a long list. `LRANGE orders:recent:user:1042 0 19 -> the 20 most recent order ids`
LRANGE key 0 -1	Read everything. Avoid it on production keys, this is the command that turns a large list into an incident. `LRANGE orders:recent:user:1042 0 -1 -> every element, O(N)`
LPOP key [count] / RPOP key	Consume from the head or tail. Use LPOP with RPUSH to get a FIFO worker queue. `LPOP queue:invoices 1 -> 1) "job:5512"`
BLPOP key timeout	Blocking pop for a worker that should wait for work instead of polling in a loop. `BLPOP queue:invoices 5 -> the element, or (nil) after 5s`
LMOVE src dst LEFT RIGHT	Atomically move an element between lists. This is the reliable-queue pattern and replaces the deprecated RPOPLPUSH. `LMOVE queue:invoices queue:processing LEFT RIGHT -> "job:5512"`
LLEN key	Check depth. On a queue this is your backlog metric and worth alerting on. `LLEN queue:invoices -> (integer) 3`
LINDEX key 0	Peek at a position without removing anything, for example the newest order id. `LINDEX orders:recent:user:1042 0 -> "order:98213"`
LSET key index value	Overwrite an element in place when a status changes and the position must stay. `LSET queue:processing 0 job:5512:retry -> OK`
LINSERT key BEFORE pivot v	Insert next to a known element. Convenient, but it scans to find the pivot. `LINSERT queue:invoices BEFORE job:5512 job:5599 -> (integer) 5`
LREM key count value	Remove matching elements, for example cancelling a queued job. O(N), keep it off hot paths. `LREM queue:invoices 1 job:5512 -> (integer) 1`
LPOS key value	Find the index of a value without pulling the list into your application. Redis 6.0.6+. `LPOS orders:recent:user:1042 order:98100 -> (integer) 6`

Worked example: last 20 orders on a profile page

```
LPUSH orders:recent:user:1042 order:98213 # (integer) 21
LTRIM orders:recent:user:1042 0 19 # bound it in the same code path
LRANGE orders:recent:user:1042 0 19 # profile page read
LLEN orders:recent:user:1042 # (integer) 20, forever
```

PRODUCTION TRAP The spec says show the last 20 orders, but the code never bounds the list. Two years later a single user holds two lakh order ids. The page still looks fast, because LRANGE 0 19 from the head is cheap. The real cost is memory, larger snapshots, heavier replication, and a big key that hurts whenever someone runs LRANGE 0 -1 or LREM. Error rates stay at zero the whole time.

NUANCE A List preserves insertion order, not ranking. Head and tail operations are O(1), anything in the middle is a linear scan. Never build a leaderboard on a List.

Rule: Recent-N window, queue, or activity feed means List, and every push is followed by an LTRIM.

4. SET

uniqueness | tags | membership | set math

Reach for it when: Duplicates must be impossible, you need a fast has-this-been-seen check, or you need intersections and unions.
Do not use it when: Order or ranking matters. A Set guarantees neither, so Top-N belongs in a Sorted Set.

SADD key m1 m2 ...	Add members. Duplicates are silently ignored, which is exactly the point. `SADD restaurant:biryani-house:tags biryani spicy north-indian -> (integer) 3`
SISMEMBER key member	O(1) membership check. This replaces pulling an array into the app and running includes(). `SISMEMBER user:1042:viewed dish:42 -> (integer) 1`
SMISMEMBER key m1 m2	Check several members in one round trip, for example filtering a recommendation batch. Redis 6.2+. `SMISMEMBER user:1042:viewed dish:42 dish:77 -> 1) (integer) 1 2) (integer) 0`
SREM key member	Remove a member, for example when a user clears history or a tag is retired. `SREM restaurant:biryani-house:tags north-indian -> (integer) 1`
SCARD key	Count members without transferring them. Cheap enough for dashboards. `SCARD user:1042:viewed -> (integer) 137`
SINTER key1 key2	Common members computed inside Redis. This is how you get shared tags between two restaurants with no nested loops. `SINTER restaurant:biryani-house:tags restaurant:karim:tags -> 1) "spicy"`
SINTERCARD n k1 k2 LIMIT x	Size of the intersection without materializing it. Perfect for a similarity score. Redis 7.0+. `SINTERCARD 2 restaurant:biryani-house:tags restaurant:karim:tags LIMIT 10 -> (integer) 1`
SUNION key1 key2	Combine sets, for example all tags across a user's favourite restaurants. `SUNION restaurant:biryani-house:tags restaurant:karim:tags -> the merged tag list`
SDIFF key1 key2	Members in the first set only. This is your not-yet-seen candidate list. `SDIFF dishes:all user:1042:viewed -> unseen dish ids`
SINTERSTORE dst k1 k2	Store the result for reuse instead of recomputing it on every request. SUNIONSTORE and SDIFFSTORE work the same way. `SINTERSTORE tags:common:bh:karim restaurant:biryani-house:tags restaurant:karim:tags -> (integer) 1`
SRANDMEMBER key n	Random sample without removing anything, for a surprise-me or shuffled carousel. `SRANDMEMBER dishes:veg 3 -> three random dish ids`
SPOP key [count]	Pop random members, which fits raffles, coupon pools, and one-shot assignment. `SPOP coupons:launch 1 -> 1) "TADKA50-8812"`
SMEMBERS key	Return everything. Fine for a small tag set, dangerous on a set with lakhs of members. `SMEMBERS restaurant:biryani-house:tags -> all tags`
SSCAN key cursor COUNT n	Iterate a large set in slices. Use this instead of SMEMBERS on anything user-generated. `SSCAN user:1042:viewed 0 COUNT 100 -> cursor plus a batch`

Worked example: shared tags and view deduplication

```
SADD restaurant:biryani-house:tags biryani spicy north-indian # (integer) 3
SADD restaurant:karim:tags mughlai spicy delhi # (integer) 3
SINTER restaurant:biryani-house:tags restaurant:karim:tags # 1) "spicy"
SADD user:1042:viewed dish:42 # (integer) 1
SISMEMBER user:1042:viewed dish:42 # (integer) 1
```

PRODUCTION TRAP Viewed dishes and tags get stored in a List, and every request runs includes() plus nested loops in application code. With ten items on a laptop it is invisible. In production the app server scans on every call, CPU climbs, and latency drifts up. A recommendation engine repeating dishes looks like a broken algorithm, but the algorithm is fine. The data structure was wrong.

NUANCE A Set enforces uniqueness only. It carries no order and no score, so it cannot answer top ten or most popular questions.

Rule: Uniqueness, membership checks, intersection, or union means Set, and the work happens inside Redis.

5. SORTED SET (ZSET)

leaderboards | trending | Top-N | time indexes

Reach for it when: Every member carries a score and Redis should maintain the ordering for you.

Do not use it when: You only need uniqueness with no ordering. A plain Set is cheaper in memory and simpler.

ZADD key score member	Add a member or overwrite its score. The ordering updates itself. `ZADD trending:dishes 5100 masala-dosa -> (integer) 1`
ZADD key GT CH score m	Update only when the new score is higher, which is what a high-score board wants. Redis 6.2+. `ZADD highscore:weekly GT CH 940 user:1042 -> (integer) 1`
ZINCRBY key n member	Bump a score by a delta. One command per order, and the ranking re-sorts itself. `ZINCRBY trending:dishes 700 butter-chicken -> "5221"`
ZRANGE key 0 9 REV WITHSCORES	The Top 10, highest first, straight from Redis. This is the homepage query. `ZRANGE trending:dishes 0 9 REV WITHSCORES -> top ten members with scores`
ZRANGE key min max BYSCORE	Slice by score range, for example everything above a popularity threshold. Replaces the deprecated ZRANGEBYSCORE. `ZRANGE trending:dishes (4000 +inf BYSCORE -> members scoring above 4000`
ZRANGE key min max BYLEX	Lexicographic range when all members share one score, for autocomplete style lookups. `ZRANGE autocomplete:dishes [bi [bj BYLEX -> entries starting with bi`
ZSCORE key member	Read one member's score, for example the current popularity of a dish. `ZSCORE trending:dishes butter-chicken -> "5221"`
ZMSCORE key m1 m2	Read several scores in one call rather than looping ZSCORE. Redis 6.2+. `ZMSCORE trending:dishes butter-chicken masala-dosa -> 1) "5221" 2) "5100"`
ZREVRANK key member	The member's position counting from the top. This is the your rank is #14 line in a UI. `ZREVRANK trending:dishes masala-dosa -> (integer) 1`
ZCARD key / ZCOUNT key min max	Total members, or how many fall inside a score band. `ZCOUNT trending:dishes 4000 +inf -> (integer) 3`
ZREM key member	Remove a member, for example a dish that goes out of stock. `ZREM trending:dishes idli-sambar -> (integer) 1`
ZREMRANGEBYRANK key 0 -101	Keep only the top 100 and discard the tail. This is how you bound a leaderboard. `ZREMRANGEBYRANK trending:dishes 0 -101 -> (integer) 340`
ZREMRANGEBYSCORE key min max	Drop everything older or lower than a cutoff. With timestamps as scores this is your sliding window cleanup. `ZREMRANGEBYSCORE events:user:1042 -inf 1735689600 -> (integer) 812`
ZPOPMAX key / ZPOPMIN key	Pop the highest or lowest scoring member, which turns a ZSET into a priority queue. `ZPOPMAX trending:dishes -> 1) "butter-chicken" 2) "5221"`
BZPOPMIN key timeout	Blocking version for a scheduler worker that waits for the next due job. `BZPOPMIN jobs:scheduled 10 -> the earliest job, or (nil)`
ZUNIONSTORE dst 2 k1 k2 WEIGHTS 1 0.5	Merge boards with weighting, for example this week counting double against last week. `ZUNIONSTORE trending:blended 2 trending:today trending:week WEIGHTS 1 0.5 -> (integer) 48`
ZRANDMEMBER key n WITHSCORES	Random sample that still carries scores, useful for a rotating spotlight. Redis 6.2+. `ZRANDMEMBER trending:dishes 3 WITHSCORES -> three random members`

Worked example: trending dishes on the homepage

```
ZADD trending:dishes 5100 masala-dosa 4521 butter-chicken 4100 dal-makhani
ZINCRBY trending:dishes 700 butter-chicken # "5221", now rank 1
ZRANGE trending:dishes 0 9 REV WITHSCORES # homepage Top 10
ZREVRANK trending:dishes masala-dosa # (integer) 1
ZREMRANGEBYRANK trending:dishes 0 -101 # keep the board bounded
```

PRODUCTION TRAP The first version stores each dish count as a String, reads them all on every request with GET or MGET, sorts in application memory, and returns the top ten. MGET does cut round trips, but the sort still runs on your app server and the entire dataset still crosses the network on every page view, to produce ten rows.

NUANCE A Sorted Set maintains ranking, not object detail. Keep only the id and the score in the ZSET and fetch names, images, and prices from a Hash or the database. Internally it uses a skiplist plus a hash table, which keeps inserts, score updates, and rank lookups efficient.

Rule: Scores, ranking, or Top-N means Sorted Set. Never emulate it with a List.

6. KEYS, TTL, AND SAFETY

works across every structure

Reach for it when: You need expiry, inspection, safe iteration, or atomic multi-command behaviour.

Do not use it when: You are tempted to run KEYS or FLUSHALL on a production instance. Neither belongs there.

EXPIRE key secs	Attach a TTL to a key that already exists, for example after a bulk import. `EXPIRE cart:user:1042 1800 -> (integer) 1`
EXPIREAT key timestamp	Expire at a fixed unix time, which suits daily counters that must reset at midnight. `EXPIREAT counter:2026-08-02 1754179199 -> (integer) 1`
TTL key / PTTL key	Remaining life in seconds or milliseconds. Your first debugging step when a cache misbehaves. `TTL cart:user:1042 -> (integer) 1793`
PERSIST key	Remove the expiry so the key survives. `PERSIST cart:user:1042 -> (integer) 1`
EXISTS key	Check presence before an expensive recompute. `EXISTS restaurant:biryani-house -> (integer) 1`
TYPE key	Find out which structure a key holds. Invaluable when debugging a WRONGTYPE error. `TYPE trending:dishes -> zset`
DEL key	Delete immediately. Blocks while freeing memory, so it is risky on very large keys. `DEL cache:home:v2 -> (integer) 1`
UNLINK key	Delete with background reclamation. This is the safe choice for big keys on a live system. `UNLINK orders:recent:user:1042 -> (integer) 1`
SCAN cursor MATCH p COUNT n	Iterate the keyspace in slices. Use it always, and never run KEYS in production, which blocks the entire server. `SCAN 0 MATCH session:* COUNT 100 -> cursor plus a batch of keys`
OBJECT ENCODING key	See the internal encoding, such as listpack against hashtable or skiplist. Small objects use a compact encoding and cross over as they grow. `OBJECT ENCODING restaurant:biryani-house -> listpack`
MEMORY USAGE key	Bytes consumed by one key. This is how you hunt down the big keys behind slow snapshots. `MEMORY USAGE orders:recent:user:1042 -> (integer) 4194328`
COPY src dst / RENAME src dst	Duplicate or rename a key, which makes atomic cache rebuilds and version swaps easy. COPY is Redis 6.2+. `COPY cache:home:v3 cache:home:v4 -> (integer) 1`
MULTI ... EXEC	Queue commands and run them without another client interleaving. Use it whenever two writes must land together. `MULTI; INCR rate:user:1042; EXPIRE rate:user:1042 60; EXEC -> 1) 1 2) 1`
WATCH key	Optimistic locking. The following EXEC aborts if the watched key changed underneath you. `WATCH stock:dish:42 -> OK`
EVAL script numkeys k arg	Run several commands atomically with logic in between, which a transaction cannot do. `EVAL "return redis.call('INCR', KEYS[1])" 1 rate:user:1042 -> (integer) 1`
INFO memory / DBSIZE	Instance level health: memory usage, fragmentation, and how many keys exist. `DBSIZE -> (integer) 184203`

Rule: Give every cached key a TTL by default, and treat a key without an expiry as a decision you made on purpose.

KEY NAMING AND COMPOSITION

one feature, several structures

```
session:user:1042 -> String SET ... EX 3600
rate:user:1042 -> String INCR plus EXPIRE inside Lua
restaurant:biryani-house -> Hash name, rating, is_open, total_orders
restaurant:biryani-house:tags -> Set SADD, SINTER for shared tags
orders:recent:user:1042 -> List LPUSH followed by LTRIM 0 19
user:1042:viewed -> Set SADD, SISMEMBER for deduplication
trending:dishes -> ZSET ZINCRBY, ZRANGE ... REV
```

A single homepage usually touches three structures at once. Ranking lives in the Sorted Set, the display detail lives in Hashes, and the session lives in a String. Use a colon separated namespace of the form app:entity:id:field so that SCAN patterns and key expiry policies stay predictable.

THE FOUR SILENT DESIGN TAXES

no crash, no alert, rising latency

Everything forced into a String	A single field change costs a full blob read, parse, rewrite. Move the object into a Hash.
Top-N sorted in the application layer	The whole dataset crosses the network every request to produce ten rows. Move it into a Sorted Set.

Ranking built on a List	Insertion order is not ranking, and mid-list operations are O(N). Move it into a Sorted Set.
Unbounded lists and leaderboards	Memory, snapshot size, and replication all grow silently. Bound them with LTRIM or ZREMRANGEBYRANK.

VERSION NOTES

assumes Redis 6.2 or newer

ZREVRANGE, ZRANGEBYSCORE, and ZRANGEBYLEX were deprecated in 6.2, so use ZRANGE with REV, BYSCORE, or BYLEX instead. HMSET is deprecated in favour of HSET, and RPOPLPUSH in favour of LMOVE. GETDEL, GETEX, SMISMEMBER, ZMSCORE, COPY, and the ZADD GT and LT flags arrived in 6.2. SINTERCARD arrived in 7.0. Per-field hash expiry through HEXPIRE arrived in 7.4. Run INFO server to confirm your version before relying on the newer commands.

THIRTY SECOND RECAP

problem, structure, command, mistake

PROBLEM	STRUCTURE	COMMAND YOU WILL RUN	MISTAKE TO AVOID
Session token, OTP, flag, counter	**STRING**	`SET k v EX 3600 / INCR k`	Splitting INCR and EXPIRE into two commands
Restaurant or product metadata	**HASH**	`HSET / HMGET / HINCRBY`	Storing the object as one JSON blob
Last 20 orders, activity feed, queue	**LIST**	`LPUSH then LTRIM 0 19`	Never bounding the list
Viewed items, tags, dedup	**SET**	`SADD / SISMEMBER / SINTER`	Scanning an array in application code
Trending, leaderboard, Top-N	**SORTED SET**	`ZINCRBY / ZRANGE 0 9 REV`	Sorting the whole dataset in the app

A junior engineer asks which Redis data structure to use.
An architect asks how the data will be accessed.