Introduction to Redis and Key-Value Stores - Exercises

NoSQL Database Course

September 26, 2025

1 Exercise 1: Basic Redis Commands

1. Install Redis on your local machine (or, preferably, use a docker container).
2. Connect to your Redis instance using the Redis CLI.
3. Execute the following commands:
   - Set a key-value pair: `SET user:1001:name "John Doe"`
   - Get the value of a key: `GET user:1001:name`
   - Delete a key: `DEL user:1001:name`

2 Exercise 2: Working with Data Structures

1. Using Redis, create a list of your favorite movies:
   - `LPUSH movies "Inception"`
   - `LPUSH movies "The Matrix"`
   - `LPUSH movies "Interstellar"`
2. Retrieve the list of movies: `LRANGE movies 0 -1`

3 Exercise 4: Hashes and Objects

1. Use Redis hashes to store a user profile and then update individual fields:
   - `HSET user:1001 name "John" email "john@example.com" age 29`
   - `HGETALL user:1001`
   - `HINCRBY user:1001 age 1` % try incrementing a numeric field
2. Model a small product catalog using hashes and query a single field and the whole object.

4 Exercise 5: Lists, Sets and Sorted Sets

1. Implement a simple FIFO queue for background jobs using lists. Push three jobs and pop them in order.
2. Use a set to store unique tags for an article and compute the intersection of tags between two articles using `SINTER`.
3. Create a leaderboard with sorted sets. Add three players with scores and retrieve the top 3 with scores.

5 Exercise 6: TTL and Expiration

1. Set a key with a TTL and observe TTL countdown: `SET temp:value 42` then `EXPIRE temp:value 10` and check with `TTL temp:value`.

6 Exercise 8: Continue the Video Game Store API

This exercise continues the Document DB session where you built a small HTTP API to query and modify documents (the API you implemented in the Document DB exercises). The goal is to extend that API by adding a Redis-based caching layer for read-heavy endpoints and to implement correct cache invalidation on writes.

The Redis Python client documentation is useful: `https://redis.io/docs/latest/develop/clients/redis-py/`.

1. Context and objective:
   - Take the API you implemented in the Document DB exercise (the one that reads/writes documents from your document database).
   - Add Redis as a Key-Value cache to improve read performance for resource and query endpoints.
   - Keep the document DB as the source of truth; Redis is only used for caching.
2. Setup:
   - Run Redis locally (recommended via Docker):
     - Example quick command: `docker run -p 6379:6379 --rm redis:latest`
     - Or add a `redis` service to your project `docker-compose.yml` if you already use Docker for your API.
   - Add the Python client to your API environment: `pip install redis` (or the equivalent in your project's dependency file).
3. Integration tasks:
   (a) Connect to Redis from your Flask/FastAPI app and centralise the connection (e.g. create a `cache` module that exposes `get`, `set`, `delete` helpers).
   (b) Identify read-heavy endpoints to cache. Typical candidates:
   - GET /games (list with optional filters/pagination)
   - GET /games/:id (single document)

   (c) Implement caching for these endpoints:
   - Create deterministic cache keys. Examples:
     - `games:list:page=2:per=20`
     - `game:`{id}
     - `search:`{sha1(query+params)}
   - Store JSON-serialized responses in Redis. Use a reasonable TTL (e.g. 60–300 seconds) to avoid stale data.
   - On a cache hit, return the cached JSON directly.
   - On a cache miss, query the document DB, store the serialized response in Redis with TTL, then return it.

   (d) Implement cache invalidation on writes:
   - After POST/PUT/DELETE operations that change documents, invalidate related cache keys.
   - Simple approach: delete resource-specific keys (e.g. `game:`{id}) and relevant query keys (e.g. any `game:`{id} keys). You can use predictable prefixes so you can delete by pattern (`game:`{id}:*).
4. Testing and validation:
   - Measure request latency with and without the cache (use simple curl timing or a small benchmark script).
   - Verify TTL expiry by observing that cached responses disappear after the TTL and the next request repopulates the cache.

- Verify invalidation by creating/modifying/deleting resources and ensuring subsequent reads reflect the changes (no stale data returned from cache).

You can extend this exercise by adding request-level caching, session storage, or using Redis as a primary store for ephemeral state (e.g. rate-limits, temporary counters) alongside your document DB.