Spring Boot + JPA + Redis — POC Documentation

# 1. Detailed Setup Instructions for IntelliJ & Redis

Prerequisites:

• Java JDK 17 or later  
• Maven 3.8+  
• IntelliJ IDEA (Community or Ultimate)  
• Redis server (local or WSL)   
• Postman

## 1.1 Install Java and Maven

1. Download and install Java JDK 17+: https://adoptium.net/ or Oracle JDK.  
2. Verify Java: open Command Prompt and run:  
 java -version  
3. Install Maven: https://maven.apache.org/download.cgi  
4. Verify Maven:  
 mvn -v

## 1.2 Install Redis on Windows (two recommended options)

Option A — WSL (recommended):  
1. Install WSL and Ubuntu from Microsoft Store.  
2. Open Ubuntu terminal and run:  
 sudo apt update  
 sudo apt install redis-server  
 sudo service redis-server start  
3. Verify: redis-cli ping (should return PONG)  
  
Option B — Native Windows port:  
1. Download prebuilt Redis from https://github.com/tporadowski/redis/releases  
2. Extract and run redis-server.exe from PowerShell or double-click.  
3. Verify: run redis-cli.exe and execute PING.

## 1.3 Open Project in IntelliJ

1. Unzip the project archive to a folder (e.g., C:\Projects\spring-jpa-redis-demo).  
2. Launch IntelliJ IDEA → File → Open → select the project folder.  
3. IntelliJ will detect a Maven project; allow it to import and download dependencies.  
4. Set project SDK to Java 17: File → Project Structure → Project SDK → choose JDK 17.  
5. Build the project (Build → Rebuild Project) to ensure dependencies resolve.

## 1.4 Configure Redis Connection (application.yml)

Open src/main/resources/application.yml and confirm the Redis section points to your local Redis:  
  
spring:  
 redis:  
 host: localhost  
 port: 6379  
 cache:  
 type: redis  
  
If your Redis uses authentication, add password entry and configure accordingly.

## 1.5 Run the Application

Method A — IntelliJ:  
1. Open DemoApplication.java  
2. Click the green Run icon next to main() or create a Run Configuration.  
  
Method B — Command line:  
1. Open terminal at project root  
2. Run: mvn spring-boot:run  
  
Verify: Spring Boot starts and listens on port 8080. Check logs for successful connection to Redis.

# 2. Optimized Repository & Caching Strategy

Goals:  
• Minimize number of SQL queries (prevent N+1)  
• Return lightweight DTOs for list endpoints  
• Use Redis to cache read-heavy endpoints  
• Keep writes consistent by invalidating caches

## 2.1 Repository Optimizations

Key techniques used:

• DTO Projection (JPQL constructor expression): returns only required fields for list endpoints, avoiding full entity hydration and lazy collection loading.  
• @EntityGraph: fetch required associations (e.g., author) in same query for endpoints that return entity instances, avoiding N+1.  
• Pagination (Pageable): ensures queries are bounded and indexed operations are used.  
• Native query for heavy aggregations: used for reports with GROUP BY and LIMIT where SQL-level tuning and indexes are necessary.  
• Indexes: add database indexes on frequently filtered/joined columns (author\_id, created\_at, email, post\_id in comments).

Example repository methods (explanations):

1) Post summary projection (JPQL constructor):  
@Query("SELECT new com.example.demo.dto.PostSummaryDto(p.id, p.title, p.createdAt, u.id, u.name, COUNT(c.id)) "  
 + "FROM Post p JOIN p.author u LEFT JOIN p.comments c "  
 + "GROUP BY p.id, u.id, u.name ORDER BY p.createdAt DESC")  
List<PostSummaryDto> findPostSummaries(Pageable pageable);  
  
Why: This executes one SQL query that returns aggregated counts and author info without loading full post/comment entity trees.

## 2.2 Caching Strategy (Redis)

Cache design principles:

• Cache read-heavy, idempotent endpoints (list of posts, single post view).  
• Use short-to-medium TTL (e.g., 5–15 minutes) for list caches to balance freshness and load reduction.  
• Evict caches on writes: annotate create/update/delete service methods with @CacheEvict for relevant caches.  
• Cache key design: include pagination keys (page-size) to avoid collision.  
• Prevent caching of null results using RedisCacheConfiguration.disableCachingNullValues().

Implementation notes:  
• Configure CacheManager with GenericJackson2JsonRedisSerializer for portability.  
• Use @Cacheable(cacheNames = "postSummaries", key = "#page + '-' + #size") for list method.  
• Use @Cacheable(cacheNames = "post", key = "#id") for single post retrieval.  
• On writes use @CacheEvict(cacheNames = {"post", "postSummaries"}, allEntries = true).

# 3. Entity Relationship Diagram (ERD)

3.1 ASCII ERD:

+--------+ +--------+ +-----------+  
| User |1 ----<| Post |1 ----<| Comment |  
+--------+ +--------+ +-----------+  
  
Columns:  
User(id, name, email)  
Post(id, title, body, createdAt, author\_id)  
Comment(id, content, createdAt, post\_id)

## 3.2 PlantUML ERD (copy-paste into PlantUML)

@startuml  
entity User {  
 \*id : Long  
 \*name : String  
 \*email : String  
}  
entity Post {  
 \*id : Long  
 \*title : String  
 \*body : String  
 \*createdAt : String  
}  
entity Comment {  
 \*id : Long  
 \*content : String  
 \*createdAt : String  
}  
User ||--o{ Post : author  
Post ||--o{ Comment : comments  
@enduml

3.3 Visual notes:  
• User has one-to-many Posts (author relationship)  
• Post has one-to-many Comments  
• All foreign keys (author\_id, post\_id) should be indexed

# 4. Performance Improvement Analysis

4.1 What we measured:

• Endpoint: GET /api/posts?page=0&size=20 (Post summaries)  
• Baseline (no caching, naive entity mapping): observed repeated queries and N+1 problems  
• Optimized (DTO projection + indexes + Redis cache): single aggregated query, cached responses

## 4.2 SQL Logging & Investigation

Logging setup (application.yml):  
org.hibernate.SQL = DEBUG  
org.hibernate.type.descriptor.sql.BasicBinder = TRACE

Typical problematic pattern (N+1):  
SELECT \* FROM posts ...  
Then for each post: SELECT \* FROM users WHERE id = ?  
This means N additional queries for N posts.

Fix applied:  
• Use JPQL projection to get post + author + comment count in one query.  
• Use @EntityGraph for endpoints that return entity instances and require author loaded.

## 4.3 Before vs After (representative numbers)

Note: numbers are illustrative. Run your own measurements with EXPLAIN ANALYZE on your DB.

• Before optimizations: avg response time 250–450 ms, DB queries: 1 + N (author fetch) per page  
• After DTO + indexes: avg response time 90–150 ms, DB queries: 1 per page  
• After caching (Redis, TTL=10m): first request 120 ms (cache fill), subsequent requests < 20–40 ms (served from Redis)

## 4.4 Recommendations when analyzing logs

• Look for repeated similar SELECT statements with different bound parameters -> N+1  
• Copy problematic SQL and run EXPLAIN/EXPLAIN ANALYZE in Postgres (or EXPLAIN in MySQL)  
• Add indexes on columns used in JOIN/WHERE/GROUP BY  
• Consider materialized views or counters for heavy aggregates  
• Enable hibernate.generate\_statistics=true for detailed Hibernate metrics

# 5. Postman Testing Guide (end-to-end)

5.1 Create a new Postman Collection:

1. Open Postman → New → Collection → name it 'spring-jpa-redis-demo'.  
2. Add requests as described below.

## 5.2 Req: GET all posts (cached)

• Method: GET  
• URL: http://localhost:8080/api/posts  
• Query params: page=0 size=20  
• Expected: JSON array of PostSummaryDto  
  
Behavior:  
• First request hits DB and fills cache  
• Subsequent requests (within TTL) are served from Redis (no DB SQL logs for main query).

## 5.3 Req: GET single post (cached)

• Method: GET  
• URL: http://localhost:8080/api/posts/{id}  
• Expected: Full Post entity with author (if using @EntityGraph) but beware of serialization loops — use DTO or @JsonBackReference/@JsonManagedReference.  
  
Behavior:  
• Cached by key 'post::{id}'  
• On update/delete, cache is evicted.

## 5.4 Req: POST create a new post

• Method: POST  
• URL: http://localhost:8080/api/posts  
• Body type: x-www-form-urlencoded or form-data with fields: title, body, authorEmail  
• Expected: 200 OK with saved Post JSON  
  
Behavior:  
• Service method annotated with @CacheEvict invalidates postSummaries cache so subsequent GET queries are fresh.

## 5.5 Req: DELETE post

• Method: DELETE  
• URL: http://localhost:8080/api/posts/{id}  
• Expected: 204 No Content  
  
Behavior:  
• On delete, caches 'post' and 'postSummaries' are evicted to maintain consistency.

5.6 Postman tests to observe caching effect:

1. Send GET /api/posts — note response time and SQL logs.  
2. Send GET /api/posts again — response time should drop, and main SQL query should not appear in logs.  
3. POST a new post. Then GET /api/posts — should trigger DB fetch (cache invalidated).

# 6. Additional Operational Notes

• Use H2 console for quick DB inspection: enable spring.h2.console in application.yml and open /h2-console.  
• For production, replace H2 with Postgres/MySQL and tune connection pool (HikariCP), set spring.jpa.properties.hibernate.jdbc.batch\_size for batching, and enable SSL/TLS for Redis.

# 7. Appendix - Useful Commands & Examples

Redis CLI:  
• redis-cli  
• keys \*  
• get <key>  
  
Maven:  
• mvn clean install  
• mvn spring-boot:run  
  
Postgres EXPLAIN example:  
• EXPLAIN ANALYZE <copied query>;