Redis Caching Integration in Spring Boot

Architecture, Design Principles & Best Practices

Optimisation des performances avec Redis

Document Technique Avancé

Guide complet d'intégration Redis avec Spring Boot
Architecture • Configuration • Stratégies • Production

Omar Ignammas

December 25, 2025

Contents

1 Introduction

Redis is an in-memory data store widely used to improve application performance through caching. When integrated with Spring Boot, Redis enables faster response times, reduced database load, and better scalability.

This document presents the **architecture**, **design principles**, and **best practices** for using Redis caching in a Spring Boot application at an enterprise level.

2 Global Architecture Overview

The Redis caching architecture follows a layered approach where Redis acts as an intermediate layer between the application and the database.

2.1 High-Level Architecture

```
Client
  |
  v
Spring Boot Application
  |
  |-- Cache Layer (Redis)
  |
  v
Database
```

2.2 Request Flow Explanation

1. The client sends a request to the Spring Boot application.
2. The application checks Redis cache first.
3. If data exists in cache (cache hit), it is returned immediately.
4. If not (cache miss), the application queries the database.
5. The result is stored in Redis for future requests.

3 Redis Integration Architecture

3.1 Main Components

- **Spring Cache Abstraction**: Provides annotations like `@Cacheable`.
- **Redis Server**: Stores cached data in memory.
- **Redis Cache Manager**: Manages TTL, serialization, and cache behavior.
- **Database**: Persistent storage (PostgreSQL, MySQL, etc.).

Redis should never replace the database; it complements it by acting as a fast-access layer.

4 Redis Setup in a Spring Boot Project (Step-by-Step)

This section describes the concrete steps required to configure and integrate Redis caching in a Spring Boot project, from dependency management to runtime verification.

4.1 Step 1: Add Required Dependencies

To enable Redis and caching support, the following dependencies must be added to the `pom.xml` file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>

<!-- For connection pooling (recommended) -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

The `spring-boot-starter-cache` dependency activates Spring's cache abstraction, while `data-redis` handles communication with Redis. `commons-pool2` enables connection pooling for better performance.

4.2 Step 2: Run Redis Server

Redis must be running before the application starts.

Recommended approach (Docker):

```
docker run -d --name redis-local -p 6379:6379 redis:7
```

To verify Redis is running correctly:

```
redis-cli ping
```

Expected output:

```
PONG
```

4.3 Step 3: Configure Redis in Application Configuration

Redis connection settings are defined in `application.yml`:

```
spring:
  cache:
    type: redis

  data:
    redis:
      host: localhost
      port: 6379
      timeout: 2000ms
      lettuce:
        pool:
          max-active: 8
          max-idle: 8
          min-idle: 0
          max-wait: -1ms
```

Without explicitly enabling Redis as the cache type, Spring Boot will fallback to a default in-memory cache.

4.4 Step 4: Enable Caching in Spring Boot

Caching must be activated at the application level using the `@EnableCaching` annotation:

```java
@SpringBootApplication
@EnableCaching
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

4.5 Step 5: Define Redis Cache Configuration

A dedicated configuration class is used to customize TTL, serialization, and cache behavior.

```java
@Configuration
public class RedisConfig {

    @Bean
    public RedisCacheManager cacheManager(
            RedisConnectionFactory factory) {

        RedisCacheConfiguration config =
            RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(10))
                .disableCachingNullValues()
                .serializeValuesWith(
                    RedisSerializationContext.SerializationPair
                        .fromSerializer(
                          new GenericJackson2JsonRedisSerializer()
                        )
                );

        return RedisCacheManager.builder(factory)
                .cacheDefaults(config)
                .transactionAware()
                .build();
    }
}
```

Using JSON serialization improves readability, debugging, and interoperability across services.

4.6 Step 7: Validate Cache Behavior

To verify that Redis caching is working correctly:

- Call the same API endpoint multiple times.
- The first call retrieves data from the database.
- Subsequent calls retrieve data from Redis.

Redis keys can be inspected using:

```
redis-cli
keys *
```

4.7 Step 8: Enable Cache Debug Logging

To monitor cache hits and misses, enable logging:

```yaml
logging:
  level:
    org.springframework.cache: TRACE
    org.springframework.data.redis: DEBUG
```

Logging is essential during development and troubleshooting to confirm cache behavior.

5 Caching Strategy Design

5.1 Read-Through Cache Pattern

Spring Boot with Redis typically follows the **Read-Through Cache** pattern:

- Application reads from cache first.
- Cache is automatically populated on a miss.

```
@Cacheable(value = "userCache", key = "#id")
public User getUser(Long id) {
    return userRepository.findById(id).orElseThrow();
}
```

6 Cache Update and Invalidation

6.1 Why Invalidation Matters

Stale cache data can cause serious consistency issues if cache eviction is not handled correctly.

6.2 Eviction and Update Strategy

```
@CacheEvict(value = "userCache", key = "#id")
public void deleteUser(Long id) {
    userRepository.deleteById(id);
}

@CachePut(value = "userCache", key = "#user.id")
public User updateUser(User user) {
    return userRepository.save(user);
}
```

7 Conditional and Selective Caching

```
@Cacheable(
  value = "productCache",
  condition = "#id␣>␣0"
)
public Product getProduct(Long id) {
    return productRepository.findById(id).orElse(null);
}
```

Conditional caching prevents unnecessary cache pollution.

8 Resilience and Fault Tolerance

```
@Cacheable(value = "userCache", key = "#id")
public User getUserSafe(Long id) {
    try {
        return userRepository.findById(id).orElse(null);
    } catch (Exception e) {
        return fallbackUser(id);
    }
}
```

The application must continue to work even if Redis is unavailable.

9 TTL and Expiration Strategy

Time-To-Live (TTL) ensures that cached data does not live forever and helps prevent stale data.

```
@Bean
public RedisCacheConfiguration cacheConfiguration() {
    return RedisCacheConfiguration.defaultCacheConfig()
        .entryTtl(Duration.ofMinutes(10))
        .disableCachingNullValues();
}
```

10 Advanced Topics for Production

10.1 Redis Cluster Configuration

For high availability and scalability in production, Redis Cluster is essential:

```
spring:
  data:
    redis:
      cluster:
        nodes:
          - redis-node1:6379
          - redis-node2:6379
          - redis-node3:6379
        max-redirects: 3
```

10.2 Redis Sentinel for High Availability

Redis Sentinel provides automatic failover:

```
spring:
  data:
    redis:
      sentinel:
        master: mymaster
        nodes:
          - sentinel1:26379
          - sentinel2:26379
          - sentinel3:26379
```

Always use Redis Sentinel or Cluster in production environments.

10.3 Custom Cache Key Generation

For complex cache key strategies:

```
@Configuration
public class CacheKeyConfig {

    @Bean
    public KeyGenerator customKeyGenerator() {
        return (target, method, params) -> {
            return method.getName() + "_" +
                   StringUtils.arrayToDelimitedString(
                       params, "_"
                   );
        };
    }
}

// Usage
@Cacheable(value = "orders",
           keyGenerator = "customKeyGenerator")
public Order getOrder(Long userId, Long orderId) {
    return orderRepository.find(userId, orderId);
}
```

10.4 Cache Warming Strategy

Preload critical data into cache on application startup:

```
@Component
public class CacheWarmer implements ApplicationListener
                        <ContextRefreshedEvent> {

    @Autowired
    private ProductService productService;

    @Override
    public void onApplicationEvent(
            ContextRefreshedEvent event) {
        // Warm up cache with top products
        productService.getTopProducts().forEach(
            product -> productService
                        .getProduct(product.getId())
        );
    }
}
```

10.5 Distributed Locking with Redis

Prevent race conditions in distributed systems:

```
@Service
public class OrderService {

    @Autowired
    private StringRedisTemplate redisTemplate;

    public void processOrder(Long orderId) {
        String lockKey = "order:lock:" + orderId;
        Boolean acquired = redisTemplate.opsForValue()
            .setIfAbsent(
                lockKey,
                "locked",
                Duration.ofSeconds(30)
            );

        if (Boolean.TRUE.equals(acquired)) {
            try {
                // Process order safely
            } finally {
                redisTemplate.delete(lockKey);
            }
        }
    }
}
```

10.6 Cache Metrics and Monitoring

Enable Spring Boot Actuator for cache metrics:

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,metrics,caches
  metrics:
    export:
      prometheus:
        enabled: true
```

Monitor key metrics:

- Cache hit ratio
- Cache miss ratio
- Eviction count
- Cache size

10.7 Multi-Level Caching

Combine local cache (Caffeine) with Redis:

```java
@Bean
public CacheManager cacheManager(
        RedisConnectionFactory factory) {

    // L1: Local Caffeine cache
    CaffeineCacheManager localCache =
        new CaffeineCacheManager();
    localCache.setCaffeine(Caffeine.newBuilder()
        .maximumSize(1000)
        .expireAfterWrite(Duration.ofMinutes(5)));

    // L2: Redis cache
    RedisCacheManager redisCache =
        RedisCacheManager.builder(factory).build();

    // Combine both
    return new CompositeCacheManager(
        localCache, redisCache
    );
}
```

Multi-level caching reduces latency and Redis load significantly.

11 Security Considerations

11.1 Redis Authentication

Always enable authentication in production:

```yaml
spring:
  data:
    redis:
      password: ${REDIS_PASSWORD}
      ssl:
        enabled: true
```

11.2 Sensitive Data Handling

Never cache:

- Passwords or authentication tokens
- Credit card information
- Personal Identifiable Information (PII) without encryption
- Sensitive business logic results

11.3 Data Encryption

Encrypt sensitive cached data:

```java
@Bean
public RedisSerializer<Object> encryptedSerializer() {
    return new EncryptedRedisSerializer(
        new GenericJackson2JsonRedisSerializer(),
        encryptionKey
    );
}
```

12 Performance Optimization

12.1 Connection Pooling

Properly configure connection pools:

```yaml
spring:
  data:
    redis:
      lettuce:
        pool:
          max-active: 20
          max-idle: 10
          min-idle: 5
          max-wait: 2000ms
        shutdown-timeout: 100ms
```

12.2 Pipeline Operations

Batch multiple Redis commands:

```java
redisTemplate.executePipelined(
    new RedisCallback<Object>() {
        public Object doInRedis(
                RedisConnection connection) {
            for (User user : users) {
                connection.set(
                    ("user:" + user.getId())
                        .getBytes(),
                    serialize(user)
                );
            }
            return null;
        }
    }
);
```

12.3 Memory Optimization

Configure eviction policies:

```
# In redis.conf
maxmemory 2gb
maxmemory-policy allkeys-lru
```

13 Testing Redis Cache

13.1 Integration Testing

Use Testcontainers for integration tests:

```java
@SpringBootTest
@Testcontainers
public class RedisCacheIntegrationTest {

    @Container
    static GenericContainer<?> redis =
        new GenericContainer<>("redis:7")
            .withExposedPorts(6379);

    @DynamicPropertySource
    static void redisProperties(
            DynamicPropertyRegistry registry) {
        registry.add("spring.data.redis.host",
            redis::getHost);
        registry.add("spring.data.redis.port",
            redis::getFirstMappedPort);
    }

    @Test
    void testCaching() {
        // Test cache behavior
    }
}
```

14 Best Practices for Redis Caching

14.1 Recommended Practices

- Cache **read-heavy** data.
- Use **TTL** for all cache entries.
- Avoid caching large objects ($>$ 1MB).
- Use meaningful cache names.
- Monitor cache hit/miss ratios.
- Implement cache warming for critical data.
- Use connection pooling in production.
- Enable Redis persistence (RDB or AOF).
- Set up monitoring and alerting.
- Document your caching strategy.

14.2 Practices to Avoid

- Do not cache sensitive data without encryption.
- Do not cache frequently changing data.
- Do not use Redis as the primary data store.
- Avoid unbounded cache keys.
- Don't ignore cache invalidation.
- Never run Redis without authentication in production.

15 Troubleshooting Common Issues

15.1 Cache Stampede

Problem: Multiple threads hit database simultaneously on cache miss.

Solution: Use locking or "dog-pile" prevention:

```
@Cacheable(value = "users",
           key = "#id",
           sync = true)  // Synchronizes access
public User getUser(Long id) {
    return userRepository.findById(id)
                         .orElseThrow();
}
```

15.2 Memory Leaks

Monitor Redis memory usage and set appropriate eviction policies.

15.3 Connection Timeouts

Increase timeout values and check network latency:

```
spring:
  data:
    redis:
      timeout: 5000ms
      connect-timeout: 3000ms
```

16 Monitoring and Observability

16.1 Key Metrics to Monitor

- **Hit Rate**: Percentage of cache hits
- **Memory Usage**: Current memory consumption
- **Evictions**: Number of evicted keys
- **Connections**: Active connections count
- **Latency**: Average response time
- **CPU Usage**: Redis server CPU load

16.2 Integration with Monitoring Tools

Redis can be integrated with:

- Prometheus + Grafana
- Datadog
- New Relic
- CloudWatch (AWS)

```
# Redis monitoring commands
INFO stats
INFO memory
SLOWLOG GET 10
```

17 Production Deployment Checklist

1. Enable Redis authentication
2. Configure SSL/TLS encryption
3. Set up Redis Sentinel or Cluster
4. Enable persistence (RDB + AOF)
5. Configure memory limits and eviction policies
6. Set up monitoring and alerting
7. Configure connection pooling
8. Implement cache warming
9. Document cache keys and TTLs
10. Set up backup strategy
11. Test failover scenarios
12. Configure network security (firewall rules)

18 Conclusion

Redis caching, when properly designed and integrated into a Spring Boot application, significantly enhances performance and scalability. A clear caching strategy, proper TTL configuration, disciplined cache invalidation, and robust production practices are key to building reliable and efficient systems.

For production environments, always prioritize security, monitoring, and high availability. Regular performance reviews and cache optimization ensure sustained system performance as your application scales.