Spring Boot Caching with Redis: Developer Guide

This document serves as a professional and interview-ready guide to help you understand caching in Spring Boot using Redis. It explains key concepts, setup steps, annotations, and provides a working example with flow.

1)What is Caching?

Caching is the process of temporarily storing frequently accessed data in a fast-access storage (usually memory) to reduce latency and avoid repetitive, expensive operations like database queries or API calls.

Real-world Example

Imagine Netflix shows the prices of different subscription plans. This data resides in a database but doesn't change often. If 100,000 users access the pricing page simultaneously and every request hits the database, the performance will degrade.

Instead, we can cache this pricing data in Redis. Now when users request the pricing page:

- First access: data is fetched from DB and cached
- Subsequent accesses: data is served from Redis (faster)

This improves speed and reduces DB load significantly.

2)Why Caching?

- Improves application performance and response time
- Reduces database load
- Lowers latency for frequently accessed data
- Helps improve scalability and availability under high traffic

3) How to Configure Spring Boot with Redis Cache

1) Add Dependencies

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>3.5.2</version>
</dependency>
```

2) Configuring the Serialization configuration for JSON

```java
@Configuration  no usages
public class RedisConfig {
    @Bean  no usages
    public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {
        // Configure Jackson serializer for both keys and values
        // serialization is required because by default it would consider the JDKSerialization which will store in the from of bits
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.findAndRegisterModules(); // support Java 8 date/time
        objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);

        // Pass ObjectMapper through constructor instead of using setObjectMapper()
        Jackson2JsonRedisSerializer<UserDto> serializer = new Jackson2JsonRedisSerializer<>(objectMapper, UserDto.class);

        RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()
                .serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(new StringRedisSerializer()))
                .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(serializer))
                .entryTtl(Duration.ofMinutes(10)); // optional TTL

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

3) Enabling Caching in Springboot Main Class

```java
@SpringBootApplication
@EnableCaching
public class RedisApplication {

    public static void main(String[] args) {
        SpringApplication.run(RedisApplication.class, args);
    }

}
```

4) Using the annotations (Used cache aside pattern)

```java
@Cacheable(value = "user",key = "#email")  1 usage
public UserDto getUserById(String email){
    System.out.println("GETTING VALUES FROM THE DATABASE");
    if(userRepository.findByEmail(email).isPresent())
        return userMapper.userToUserDto(userRepository.findByEmail(email).get());
    throw new UserDoesNotExistException("No user found with the specified email "+email);
}

@CacheEvict(value="user",key="#userDto.email")  1 usage
public UserDto editUserInfoAndReturnUpdatedInfo(UserDto userDto){
   User details = userRepository.findByEmail(userDto.getEmail()).orElseThrow(()->new UserDoesNotExistException("User does not exist"));
   details.setName(userDto.getName());
   userRepository.save(details);
   return userMapper.userToUserDto(details);
}
@CacheEvict(value = "user",key="#email")  no usages
public String deleteUserInfo(String email){
    User details = userRepository.findByEmail(email).orElseThrow(()->new UserDoesNotExistException("user does not exist"));
    userRepository.delete(details);
    return "Deleted successfully";
}
}
```

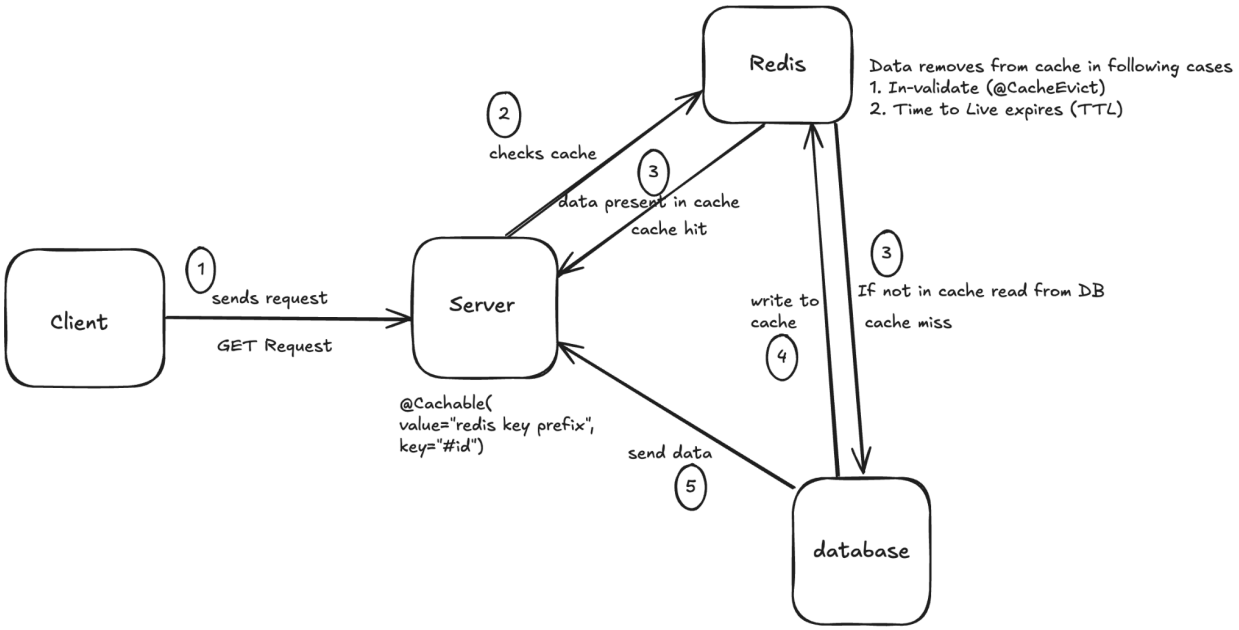

Fig - Architecture Flow of How the redis caching works for the GET requests

This diagram illustrates the **end-to-end flow** of how Spring Boot interacts with Redis for caching using `@Cacheable`.

The diagram represents the internal workflow of caching in a Spring Boot application integrated with Redis. When a client sends a request to the server to fetch data, the server does not immediately access the database. Instead, Spring's caching abstraction first checks whether the required data is already available in Redis — an in-memory data store. This check occurs through the use of the @Cacheable annotation, which specifies a unique cache key (usually derived from method parameters). This process is referred to as a cache lookup.

If the requested data is found in Redis (a condition known as a cache hit), the server bypasses the execution of the underlying method and directly returns the cached result to the client. This eliminates the need to query the database, significantly reducing latency and improving performance. However, if the data is not present in Redis (a cache miss), the server proceeds to execute the method logic, which typically involves querying the database. Once the data is retrieved from the persistent store, the result is automatically written back to Redis, so that future requests can benefit from faster access.

The final result — whether retrieved from the cache or database — is returned to the client as a response. Over time, cached entries are removed from Redis either manually or automatically. Manual invalidation is triggered by annotations such as @CacheEvict, commonly used in update or delete operations. Automatic invalidation occurs through the Time To Live (TTL) setting, which allows each cached key to expire after a defined duration.

This approach ensures that frequently accessed data is served with minimal latency, reducing the load on the database and enhancing the overall scalability and responsiveness of the application. The server, Redis cache, and database work in coordination to optimize performance while maintaining data consistency through controlled eviction strategies.

5) **Most Commonly used annotations** -

- @EnableCaching - Used on top of main class to enable the caching mechanism in the springboot application
- @Cachable - used to check the cache first before querying the database
- @CacheEvict - used to invalidate the data(remove data)
- @CachePut - used to update the cache data for update operations (if not using lazy loading or cache aside pattern then this is used on top of update methods)