Unlock the Power of Redis Cache in Spring Boot

manjultamang.com.np

What is Redis Cache ?

Redis (Remote Dictionary Server) is an in-memory data store that is often used as a cache layer to speed up data retrieval. In Spring Boot, we can easily integrate Redis using Spring Data Redis and annotations like @Cacheable, @CachePut, and @CacheEvict.

Instead of hitting the database every time, frequently accessed data is stored in Redis, leading to faster response times and reduced database load.

Why use Redis Cache?

✅ Performance Boost: Redis is lightning fast because it stores data in-memory rather than disk-based storage.

✅ Scalability: Redis can handle millions of requests per second, making it perfect for highly scalable microservices architectures.

✅ Support for Complex Data Types: Strings, Lists, Hashes, Sets, Sorted Sets – Redis is versatile!

✅ Ease of Integration: Spring Boot and Redis play very well together with minimal configuration using application.properties or application.yml.

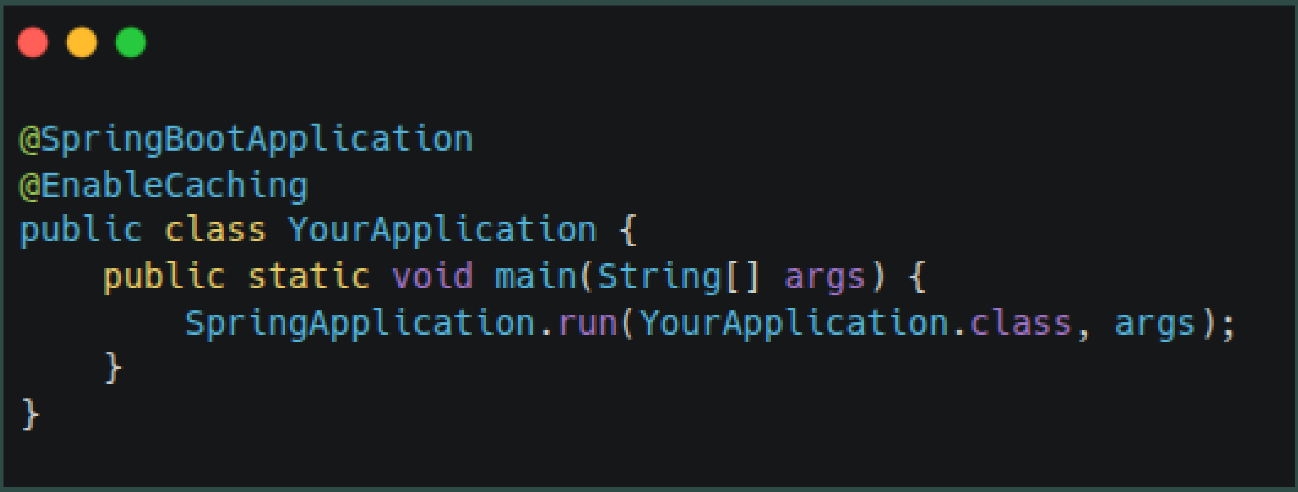
Configuring Redis Cache in Spring Boot

🛠️ First: Dependency needed for Redis Cache

Add this in your pom.xml:

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

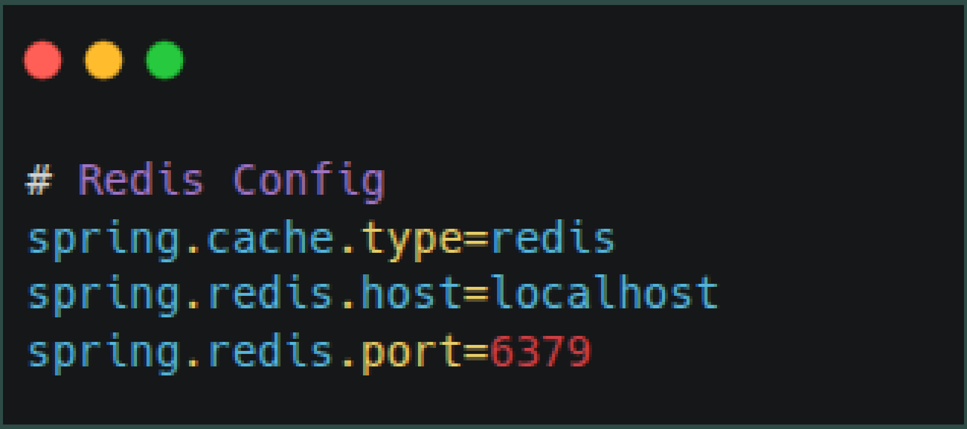
And don't forget to enable caching in your main application class:



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

You also need a running Redis Server (local or cloud).

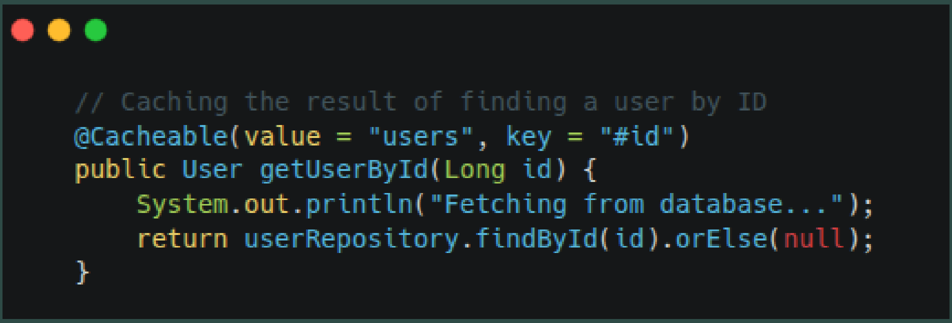
⚙️ application.properties setup:



```properties
# Redis Config
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

Service Layer with Cache Annotations

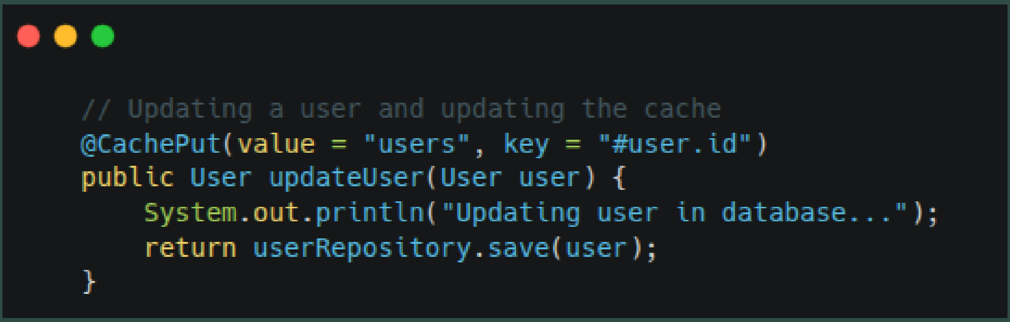
1. @Cacheable()



```java
// Caching the result of finding a user by ID
@Cacheable(value = "users", key = "#id")
public User getUserById(Long id) {
    System.out.println("Fetching from database...");
    return userRepository.findById(id).orElse(null);
}
```

@Cacheable(value = "users", key = "#id"): Caches the result of the method based on the id of the user. If the cache already contains this user, it returns the cached value instead of querying the database.

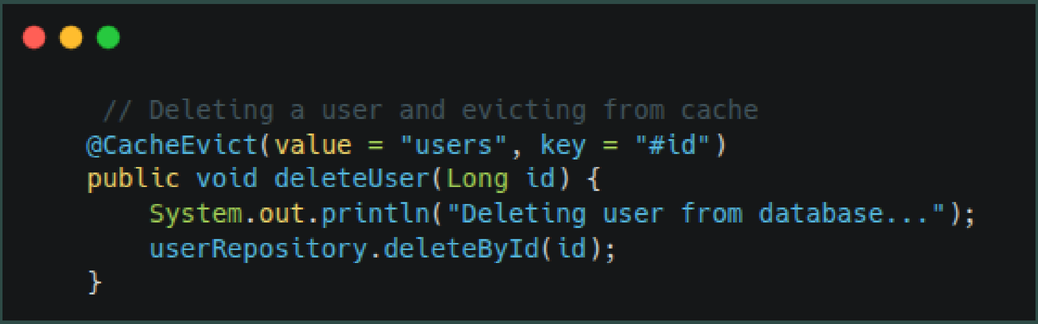
2. @CachePut()



```java
// Updating a user and updating the cache
@CachePut(value = "users", key = "#user.id")
public User updateUser(User user) {
    System.out.println("Updating user in database...");
    return userRepository.save(user);
}
```

@CachePut(value = "users", key = "#user.id"): This annotation always runs the method and updates the cache with the new user data after saving it to the database.

3. @CacheEvict()



```java
 // Deleting a user and evicting from cache
@CacheEvict(value = "users", key = "#id")
public void deleteUser(Long id) {
    System.out.println("Deleting user from database...");
    userRepository.deleteById(id);
}
```

@CacheEvict(value = "users", key = "#id"): When a user is deleted, the cache entry for that user is evicted, ensuring that stale data isn't used.

✨ Did I Miss Anything?

If you think I missed any important points or if you’d like to add something else, comment below! Let's keep improving and learning together.

🙏 Thank You!

If you found this post helpful, feel free to share it with your peers

Also, I'd love to hear your thoughts! Leave a comment or ask any questions — let’s keep the discussion going.

Manjul Tamang (Software Developer)