**Spring Boot Redis : How did I Improve User Login Experience by Caching?**

[[](https://dip-mazumder.medium.com/?source=post_page-----17922a5def68--------------------------------)](https://dip-mazumder.medium.com/?source=post_page-----17922a5def68--------------------------------)

[The Java Trail](https://dip-mazumder.medium.com/?source=post_page-----17922a5def68--------------------------------)

·

4 min read

·

Sep 7

Redis is an open-source, in-memory data store known for its versatility and high-performance capabilities. Often referred to as a “data structure server,” Redis provides a wide range of data structures like strings, lists, sets, hashes, and more, making it suitable for various use cases, including caching, real-time analytics, session management, and message queuing. Its in-memory nature enables lightning-fast data access and retrieval, while its ability to persist data ensures durability.

Redis can help make successive login and logout calls faster in a web application by efficiently managing user sessions and reducing the need for repeated authentication and database queries. Let’s explore how I achieved a improvement in login times and successive logins by transitioning from maintaining sessions in my primary database to a more efficient approach using Caching, Optimizing Database Queries, Connection Pooling, and Session Management:

A diagram of a login

Description automatically generated

**Session Management with Redis:**

* When a user logs in, their session data (e.g., username, authentication token) is stored in Redis.
* Subsequent login requests can quickly access the session data from Redis instead of re-authenticating the user or querying the database for user credentials.

**Faster Logout:**

* When a user logs out, their session data is immediately removed from Redis.
* This ensures that logout requests are fast and efficient since there’s no need to perform additional database queries or complex operations.

**Reduced Database Load:**

* By storing session data in Redis, you reduce the load on your primary database, as you don’t need to query the database for user credentials or session information with every request.
* This leads to faster database response times for other critical operations.

**Expiration Mechanism:**

* Redis allows you to set an expiration time for session data (e.g., 30 minutes in the provided example).
* This ensures that idle sessions are automatically cleaned up, preventing Redis from becoming cluttered with outdated data.

**Connection Pooling:**

* Connection pooling enables the reuse of existing database connections. In my previous approach, opening and closing connections for each login request added overhead.
* Connection pooling eliminates this overhead, making database interactions faster and more efficient.

A diagram of a circle with text and a purple circle

Description automatically generated

**Spring Boot With Redis Implementation:**

1. Add Dependencies: Ensure you have the necessary dependencies in your pom.xml for a Spring Boot application with Redis and caching support:

<dependencies>  
 <!-- Spring Boot Starter Web -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <!-- Spring Boot Starter Data Redis -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
 </dependency>  
  
 <!-- Spring Boot Starter Data JPA (for database access) -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
</dependencies>

2. Configure Redis: In your application.properties or application.yml, configure the connection to your Redis server:

# Redis configuration  
spring.redis.host=localhost  
spring.redis.port=6379  
  
# Database configuration  
spring.datasource.url=jdbc:mysql://localhost:3306/your\_database  
spring.datasource.username=your\_username  
spring.datasource.password=your\_password  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

3. User Model: Create a simple User model class to represent a user:

@Entity  
public class User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 private String username;  
 private String password;  
  
 // getters and setters  
}

4. UserRepository: Create a UserRepository interface for database access:

public interface UserRepository extends JpaRepository<User, Long> {  
 User findByUsername(String username);  
}

5. LoginService: Implement a LoginService class to handle user authentication and session management:

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.StringRedisTemplate;  
import org.springframework.stereotype.Service;  
import java.util.UUID;  
import java.time.Duration;  
  
@Service  
public class LoginService {  
 @Autowired  
 private UserRepository userRepository;  
  
 @Autowired  
 private StringRedisTemplate redisTemplate;  
  
 public String loginAndGetToken(String username, String password) {  
 // Check if the token is cached in Redis  
 String cachedToken = getCachedToken(username);  
  
 if (cachedToken != null) {  
 return cachedToken; // Token is already cached, return it  
 }  
  
 // Actual authentication logic (check username and password) here  
 User user = userRepository.findByUsername(username);  
  
 if (user != null && user.getPassword().equals(password)) {  
 // Generate a random token  
 String token = generateRandomToken();  
  
 // Cache the token in Redis with a TTL (e.g., 60 minutes)  
 cacheToken(username, token);  
  
 return token; // Return the generated token  
 }  
  
 return null; // Authentication failed  
 }  
  
 private String getCachedToken(String username) {  
 // Retrieve the token from Redis cache  
 return redisTemplate.opsForValue().get("token:" + username);  
 }  
  
 private void cacheToken(String username, String token) {  
 // Cache the token in Redis with a TTL (e.g., 60 minutes)  
 redisTemplate.opsForValue().set("token:" + username, token, Duration.ofMinutes(60));  
 }  
  
 private String generateRandomToken() {  
 // Generate a random UUID-based token (you can use a more secure method)  
 return UUID.randomUUID().toString();  
 }  
}

6. Controller:

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.\*;  
import java.util.Objects;  
  
@RestController  
@RequestMapping("/login")  
public class LoginController {  
 @Autowired  
 private LoginService loginService;  
  
 @PostMapping  
 public ResponseEntity<String> login(@RequestBody User user) {  
 String username = user.getUsername();  
 String password = user.getPassword();  
  
 if (Objects.isNull(username) || Objects.isNull(password)) {  
 return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body("Invalid request");  
 }  
  
 String token = loginService.loginAndGetToken(username, password);  
  
 if (token != null) {  
 return ResponseEntity.ok("Login successful! Token: " + token);  
 } else {  
 return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials");  
 }  
 }  
}

With these improvements, the login process is faster because:

* User authentication status is cached in Redis, reducing the need to query the database for every login request.
* Database queries are optimized for efficient query execution.
* Connection pooling is used for managing database connections efficiently.
* Session management is improved, using short-lived session tokens and smaller payloads.