SPRING SECURITY CORE CONCEPTS

Sandra Kumi

Reviewer: Mr. Thomas

Title: Understanding Spring Security: Core Concepts, Authentication, and Authorization

---

1. Introduction to Spring Security

Spring Security is a powerful and highly customizable authentication and access control framework for Java applications. It provides comprehensive security services for Java EE-based enterprise software applications. Spring Security is a part of the larger Spring Framework, allowing easy integration with existing Spring applications.

2. Core Concepts in Spring Security

2.1 Authentication

Authentication is the process of verifying the identity of a user. Spring Security supports multiple authentication methods, including form-based login, HTTP Basic authentication, OAuth2, and JWT.

In this lab, we implemented **HTTP Basic authentication** which sends the user credentials (username and password) as part of the HTTP header. This is suitable for API-based applications.

Code Example:

java

Copy code

```
http.csrf(AbstractHttpConfigurer::disable)
    .authorizeHttpRequests(authorize -> authorize
        .requestMatchers("/api/authenticate", "/api/public/**").permitAll()
        .requestMatchers("/api/admin/**").hasRole("ADMIN")
        .requestMatchers("/api/private/**").hasAnyRole("ADMIN", "USER")
        .anyRequest().authenticated()
    )
    .httpBasic();
```

2.2 Authorization

Authorization is the process of deciding whether a user has the right to perform a specific action. In Spring Security, this is managed through roles and authorities.

In this lab, we configured role-based access control using the hasRole() method to restrict access to certain endpoints based on user roles (ADMIN and USER).

Code Example:

java

Copy code

```
.authorizeHttpRequests(authorize -> authorize
    .requestMatchers("/api/admin/**").hasRole("ADMIN")
    .requestMatchers("/api/private/**").hasAnyRole("ADMIN", "USER")
    .anyRequest().authenticated()
)
```

2.3 Security Filter Chain

The Security Filter Chain is the core of Spring Security. It contains all the security filters that process HTTP requests. Each filter in the chain has a specific responsibility, such as authentication, authorization, or CSRF protection.

In our implementation, the filter chain is defined using SecurityFilterChain bean, where we disabled CSRF (since it's commonly unnecessary for APIs) and configured session management to be stateless.

Code Example:

java

Copy code

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
```

```
    http.csrf(AbstractHttpConfigurer::disable)
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/authenticate", "/api/public/**").permitAll()
            .requestMatchers("/api/admin/**").hasRole("ADMIN")
            .requestMatchers("/api/private/**").hasAnyRole("ADMIN", "USER")
            .anyRequest().authenticated()
        )
        .httpBasic();
    return http.build();
}
```

2.4 UserDetailsService

UserDetailsService is an interface that loads user-specific data. It is crucial for authentication and authorization mechanisms. We implemented this interface to load user data from a database, validate user credentials, and manage roles.

Code Example:

```
java

Copy code

@Service

public class UserService implements UserDetailsService {


    @Override

    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

        User user = userRepository.findByUsername(username);

        if (user == null) {
```

```
            throw new UsernameNotFoundException("User not found");
        }
        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                List.of(new SimpleGrantedAuthority(user.getRole().name()))
        );
    }
}
```

2.5 Password Encoding

Password encoding is essential for storing passwords securely in a database. Spring Security provides PasswordEncoder interface, and we used BCryptPasswordEncoder for hashing passwords before saving them to the database.

Code Example:

java

Copy code

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

2.6 AuthenticationManager

The AuthenticationManager interface is responsible for managing the authentication process. We configured an AuthenticationManager bean to support the authentication flow in our application.

Code Example:

java

Copy code

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
   return authenticationConfiguration.getAuthenticationManager();
}
```

3. Summary of Implementation

In this lab, we successfully integrated Spring Security into a web application with the following key security features:

- **HTTP Basic Authentication** to protect API endpoints.
- **Role-Based Access Control** to ensure only authorized users can access certain endpoints.
- **User Management** using UserDetailsService and password encoding.
- **Security Filter Chain** configuration for request authorization and stateless session management.