



Spring Security Learning Roadmap: Beginner to Professional

Based on comprehensive research and modern industry practices, here's a structured roadmap to master Spring Security and become a professional-level developer.

Phase 1: Foundations and Prerequisites (2-3 weeks)

Java Core Concepts Review

Ensure solid understanding of essential Java concepts that underpin Spring Security:[\[1\]](#) [\[2\]](#)

- **Object-Oriented Programming:** Classes, interfaces, inheritance, polymorphism[\[2\]](#)
- **Collections Framework:** List, Set, Map implementations
- **Java 8+ Features:** Lambda expressions, Streams API, Optional[\[2\]](#)
- **Exception Handling:** Try-catch blocks, custom exceptions
- **Multithreading:** Concurrent execution concepts[\[2\]](#)

Spring Framework Fundamentals

Master the core Spring concepts before diving into security:[\[3\]](#) [\[1\]](#)

- **Dependency Injection (DI) and Inversion of Control (IoC):** [\[2\]](#)
- **Spring Core:** Bean configuration, ApplicationContext[\[2\]](#)
- **Spring Boot:** Auto-configuration, starter dependencies[\[3\]](#)
- **Spring MVC:** Controllers, RequestMapping, REST endpoints[\[3\]](#)

Hands-on Exercise: Create a simple Spring Boot web application with basic REST endpoints[\[4\]](#)

Phase 2: Security Fundamentals (2-3 weeks)

Core Security Concepts

Understand the fundamental principles that guide all security implementations:[\[5\]](#) [\[6\]](#)

- **Authentication:** Verifying user identity[\[6\]](#) [\[5\]](#)
- **Authorization:** Determining user permissions[\[5\]](#) [\[6\]](#)
- **Access Control:** Restricting resource access based on permissions[\[6\]](#) [\[5\]](#)

- **Common Security Vulnerabilities:** CSRF, XSS, SQL injection [7] [8]

Spring Security Architecture

Learn how Spring Security integrates with web applications: [9] [10] [6]

- **Security Filter Chain:** How requests flow through security filters [9] [6]
- **DelegatingFilterProxy:** Bridge between Servlet container and Spring [10] [9]
- **SecurityContext and SecurityContextHolder:** Managing authentication state [10]
- **UserDetails and UserDetailsService:** User representation in Spring Security [10]

Hands-on Exercise: Set up basic Spring Security with default configuration and explore the filter chain [11]

Phase 3: Basic Authentication and Configuration (3-4 weeks)

Security Configuration

Master the new Spring Security 6 configuration approach: [12] [13]

- **SecurityFilterChain:** Replacing deprecated WebSecurityConfigurerAdapter [13]
- **authorizeHttpRequests:** Modern authorization configuration [13]
- **requestMatchers:** Path-based access control [13]
- **Password Encoding:** BCryptPasswordEncoder implementation [8] [14]

Authentication Mechanisms

Implement various authentication methods: [15] [7]

- **In-Memory Authentication:** For development and testing [15] [11]
- **Database Authentication:** Using JDBC and JPA [16] [15]
- **Custom UserDetailsService:** Loading users from custom sources [16] [15]
- **HTTP Basic Authentication:** Simple credential-based auth [16]

Hands-on Projects:

1. Basic authentication with in-memory users [1]
2. Custom login form with database authentication [1] [15]
3. User registration and login system [15] [16]

Phase 4: Advanced Authentication (4-5 weeks)

JWT Implementation

Learn stateless authentication with JSON Web Tokens: [17] [18] [4]

- **JWT Structure:** Header, payload, signature components [17]
- **Token Generation and Validation:** Creating secure tokens [17] [1]
- **Custom JWT Filter:** Processing tokens in requests [4] [1]
- **Access and Refresh Tokens:** Implementing token rotation [17]

OAuth2 and OpenID Connect

Master modern authentication protocols: [19] [12] [17]

- **OAuth2 Authorization Flows:** Authorization code, client credentials [19] [17]
- **Resource Server Configuration:** Protecting APIs with OAuth2 [19] [1]
- **OAuth2 Client:** Integrating with third-party providers [19] [17]
- **OpenID Connect:** Identity layer over OAuth2 [20] [17]

Hands-on Projects:

1. JWT-based REST API authentication [4] [1]
2. OAuth2 resource server implementation [1] [19]
3. Social login integration (Google, GitHub) [21] [17]

Phase 5: Authorization and Access Control (3-4 weeks)

Role-Based and Method-Level Security

Implement fine-grained access control: [7] [8] [13]

- **Role-Based Access Control (RBAC):** Assigning roles to users [22] [15]
- **@PreAuthorize and @PostAuthorize:** Method-level security [8] [13] [1]
- **@Secured and @RolesAllowed:** Annotation-based authorization [23] [13]
- **SpEL Expressions:** Complex authorization rules [22] [12]

Advanced Authorization

Explore sophisticated authorization patterns: [12] [1]

- **AuthorizationManager API:** New authorization framework [12]
- **Custom Authorization Rules:** Implementing business logic [12]
- **Access Control Lists (ACL):** Fine-grained object permissions [23] [1]

Hands-on Projects:

1. Role-based endpoint security [15] [1]

2. Method-level authorization with complex rules [23] [1]
3. ACL implementation for domain objects [1]

Phase 6: Microservices Security (4-5 weeks)

Distributed Security Architecture

Learn security in microservices environments: [24] [20] [19]

- **API Gateway Security:** Centralized authentication/authorization [24] [19]
- **Service-to-Service Communication:** Securing inter-service calls [24] [19]
- **Token Relay:** Propagating authentication across services [25] [19]
- **Spring Cloud Security:** Integration with Spring Cloud components [25] [19]

OAuth2 in Microservices

Implement enterprise-grade security patterns: [20] [25] [19]

- **Authorization Server:** Centralized token management [25] [19]
- **Resource Server:** Protecting individual microservices [20] [19]
- **Client Credentials Flow:** Service-to-service authentication [19]
- **Keycloak Integration:** Using enterprise identity providers [20] [25]

Hands-on Projects:

1. Microservices architecture with OAuth2 [25] [19]
2. API Gateway with Spring Cloud Gateway [25]
3. Service mesh security implementation [20]

Phase 7: Security Best Practices (2-3 weeks)

Production Security

Implement enterprise-grade security measures: [14] [22] [8]

- **HTTPS/TLS Configuration:** Securing data in transit [14] [8]
- **CSRF Protection:** Preventing cross-site request forgery [22] [8] [14]
- **CORS Configuration:** Managing cross-origin requests [26] [8]
- **Session Management:** Stateful vs stateless approaches [24] [17]

Security Hardening

Apply advanced security techniques: [27] [8] [14]

- **Input Validation:** Preventing injection attacks [8] [14]
- **Output Encoding:** XSS prevention [14]
- **Security Headers:** Content Security Policy, HSTS [8] [14]
- **Rate Limiting:** Preventing brute force attacks [8]

Hands-on Projects:

1. Production-ready security configuration [14] [8]
2. Security audit and vulnerability assessment [27]
3. Penetration testing scenarios [28]

Phase 8: Testing and Monitoring (2-3 weeks)

Security Testing

Learn to validate security implementations: [27] [17] [1]

- **Unit Testing:** @WithMockUser, @WithUserDetails [17] [1]
- **Integration Testing:** MockMvc security testing [1] [17]
- **Security Test Annotations:** Testing different user contexts [17]
- **Custom Security Test Configuration:** Test-specific setups [17]

Monitoring and Auditing

Implement security observability: [27] [8]

- **Security Event Logging:** Audit trail implementation [27] [8]
- **Authentication Failure Handling:** Custom error responses [7] [15]
- **Security Metrics:** Monitoring authentication patterns [27]
- **Alerting:** Detecting suspicious activities [27]

Hands-on Projects:

1. Comprehensive security test suite [1] [17]
2. Security monitoring dashboard [27]
3. Incident response automation [27]

Phase 9: Advanced Topics (3-4 weeks)

Modern Security Features

Explore cutting-edge security implementations:[\[29\]](#) [\[12\]](#)

- **WebAuthn/Passkeys:** Passwordless authentication[\[29\]](#)
- **Multi-Factor Authentication:** 2FA/MFA implementation[\[11\]](#)
- **Dynamic Client Registration:** Runtime OAuth2 configuration[\[12\]](#)
- **Security for Single Page Applications:** SPA-specific patterns[\[21\]](#)

Performance and Scalability

Optimize security for high-load scenarios:[\[24\]](#)

- **Caching Security Contexts:** Improving performance[\[24\]](#)
- **Distributed Session Management:** Redis/Hazelcast integration[\[24\]](#)
- **Load Balancer Considerations:** Sticky sessions vs stateless[\[24\]](#)
- **Database Connection Pooling:** Efficient resource usage[\[24\]](#)

Hands-on Projects:

1. Passwordless authentication system[\[29\]](#)
2. High-availability security architecture[\[24\]](#)
3. Performance optimization case study[\[24\]](#)

Phase 10: Professional Development (Ongoing)

Real-World Applications

Apply knowledge to complex scenarios:[\[30\]](#)

- **E-commerce Platform Security:** Multi-tenant applications[\[30\]](#)
- **Banking Application Security:** High-security requirements[\[30\]](#)
- **IoT Device Management:** Device authentication and authorization[\[30\]](#)
- **Multi-tenancy:** Isolating tenant data and access[\[30\]](#)

Continuous Learning

Stay current with security trends:[\[31\]](#)

- **Security Vulnerability Tracking:** CVE monitoring[\[31\]](#)
- **Spring Security Updates:** Following release notes[\[32\]](#) [\[31\]](#) [\[29\]](#)
- **Community Engagement:** Contributing to open source[\[31\]](#)
- **Professional Certifications:** Security-focused credentials[\[27\]](#)

Essential Resources and Tools

Documentation and Learning Materials

- **Official Spring Security Reference:** Comprehensive documentation [33]
- **Spring Security Architecture Guide:** Deep architectural insights [34]
- **Spring Guides and Tutorials:** Hands-on examples [35] [36] [21]
- **Video Tutorials:** Visual learning resources [37] [38] [26]

Development Tools

- **Spring Security Test:** Testing security configurations [17]
- **OWASP ZAP:** Security testing and vulnerability scanning [28]
- **Postman/Insomnia:** API testing with authentication [4]
- **Docker:** Containerized security testing environments [26]

Practice Platforms

- **GitHub Repositories:** Real-world examples and templates [9] [1]
- **Spring Boot Starters:** Quick project setup [3] [17]
- **Local Lab Environment:** Hands-on experimentation [20]

Success Metrics and Milestones

Technical Skills Assessment

- Successfully implement all authentication mechanisms [7] [15]
- Build secure microservices architecture [19] [25]
- Pass security code reviews and audits [27]
- Handle production security incidents [27]

Professional Competencies

- Design security architectures for enterprise applications [20] [24]
- Mentor junior developers on security best practices [14] [8]
- Contribute to security-focused open source projects [31]
- Lead security initiatives in development teams [27]

This comprehensive roadmap provides a structured path from beginner to professional-level Spring Security expertise. Each phase builds upon previous knowledge while introducing increasingly sophisticated concepts and real-world applications. The emphasis on hands-on projects ensures practical experience alongside theoretical understanding, preparing you for professional Spring Security development challenges.

Phase 2: Security Fundamentals

Welcome to **Phase 2** of your Spring Security journey! This phase establishes the critical foundation you need before diving into implementation. Let's explore the core security concepts and Spring Security's architecture.

Core Security Concepts

Understanding fundamental security principles is essential before learning any framework.[\[50\]](#) [\[51\]](#) [\[52\]](#)

1. Authentication

Authentication is the process of verifying *who* a user claims to be. Think of it as checking someone's ID at an airport security checkpoint.[\[51\]](#) [\[52\]](#) [\[50\]](#)

Key Points:

- Validates user credentials (username/password, tokens, certificates)[\[53\]](#) [\[54\]](#)
- Creates an `Authentication` object containing user details[\[55\]](#) [\[53\]](#)
- Determines if the user is legitimate[\[50\]](#) [\[51\]](#)
- Must occur before authorization[\[52\]](#) [\[51\]](#)

Real-world Example:

```
User: "I'm John Doe, my username is: johndoe"
System: "What's your password then, John?"
User: "My password is: mySecretPass123"
System: "Correct. Welcome, John!"
```

2. Authorization

Authorization determines *what* an authenticated user is permitted to do. It's like checking if someone has the right security clearance to enter a restricted area.[\[51\]](#) [\[52\]](#) [\[50\]](#)

Key Points:

- Occurs after successful authentication[\[50\]](#) [\[51\]](#)
- Checks user permissions, roles, and authorities[\[52\]](#) [\[51\]](#) [\[50\]](#)
- Controls access to resources and operations[\[54\]](#) [\[50\]](#)
- Can be applied at different levels (URL, method, domain object)[\[54\]](#)

Real-world Scenario:

Consider an e-commerce application:[\[51\]](#)

- **Customer:** Access shopping cart, view products, place orders

- **Call Center Agent:** View customer orders, track shipments
- **Administrator:** Manage users, configure system settings, access all areas

3. Common Security Vulnerabilities

Spring Security provides built-in protection against major security threats:[\[56\]](#) [\[57\]](#) [\[54\]](#)

Cross-Site Request Forgery (CSRF)

What it is: Malicious websites trick users into performing unintended actions on sites where they're authenticated.[\[57\]](#) [\[58\]](#) [\[56\]](#)

How it works: Exploits the browser's automatic cookie submission[\[58\]](#) [\[56\]](#)

Example Attack Flow:

1. User logs into banking site ([bank.com](#))
2. User visits malicious site while still logged in
3. Malicious site triggers a hidden form submission to [bank.com](#)
4. Browser automatically includes authentication cookies
5. Unauthorized transfer occurs

Spring Security Protection: CSRF tokens, SameSite cookies[\[59\]](#) [\[57\]](#) [\[58\]](#)

Cross-Site Scripting (XSS)

What it is: Injection of malicious scripts into web pages viewed by other users.[\[60\]](#) [\[56\]](#)

Types:[\[56\]](#)

- **Stored XSS:** Malicious scripts stored permanently (e.g., in database comments)
- **Reflected XSS:** Scripts reflected in search results or error messages
- **DOM-based XSS:** Client-side script manipulation

Spring Security Protection: Content Security Policy (CSP), output encoding[\[60\]](#) [\[56\]](#)

SQL Injection

What it is: Insertion of malicious SQL queries through input fields.[\[61\]](#) [\[56\]](#)

Prevention Methods:[\[61\]](#) [\[56\]](#)

- Parameterized queries/prepared statements
- ORM frameworks (JPA, Hibernate)
- Input validation and sanitization
- Principle of least privilege for database users

Spring Security Architecture

Understanding Spring Security's architecture is crucial for effective implementation.[\[62\]](#) [\[63\]](#) [\[64\]](#)

Filter Chain Architecture

Spring Security operates as a **chain of filters** that intercept HTTP requests. Each filter has a specific security responsibility.[\[63\]](#) [\[64\]](#) [\[62\]](#)

Core Components

1. DelegatingFilterProxy

- Standard Servlet Filter registered with the container[\[65\]](#) [\[66\]](#) [\[63\]](#)
- Bridges gap between Servlet container and Spring ApplicationContext[\[63\]](#) [\[65\]](#)
- Looks up Spring-managed bean named "springSecurityFilterChain"[\[66\]](#) [\[65\]](#)

```
// Simplified internal structure
public class DelegatingFilterProxy extends GenericFilterBean {
    private String targetBeanName = "springSecurityFilterChain";

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        Filter delegate = findTargetBean();
        delegate.doFilter(request, response, chain);
    }
}
```

2. FilterChainProxy (springSecurityFilterChain)

- Actual security filter chain managed by Spring[\[64\]](#) [\[67\]](#) [\[63\]](#)
- Contains and manages all security-related filters[\[64\]](#)
- Determines which SecurityFilterChain matches the request[\[63\]](#) [\[64\]](#)
- Provides centralized filter management and debugging[\[64\]](#)

3. SecurityFilterChain

- Defines which filters apply to specific URL patterns[\[64\]](#)
- Multiple chains can exist for different parts of your application[\[64\]](#)
- Filters execute in a predefined order[\[64\]](#)

Request Flow Through Filter Chain

Here's how a request travels through Spring Security:[\[62\]](#) [\[63\]](#)

1. **HTTP Request** arrives with headers, method, URL, parameters
2. **DelegatingFilterProxy** intercepts as Servlet Filter

3. **DelegatingFilterProxy** looks up "springSecurityFilterChain" bean
4. **FilterChainProxy** receives the request
5. **FilterChainProxy** determines matching SecurityFilterChain using URL patterns
6. **Security Filters** process request in order:
 - CsrfFilter (CSRF protection)
 - UsernamePasswordAuthenticationFilter (login processing)
 - BasicAuthenticationFilter (HTTP Basic auth)
 - AuthorizationFilter (access control)
7. If all filters pass, request continues to DispatcherServlet
8. If any filter fails, chain is interrupted

Security Context Management

SecurityContextHolder

The **SecurityContextHolder** is Spring Security's central repository for security information.[\[68\]](#)
[\[69\]](#) [\[55\]](#)

Key Features:

- Stores details of currently authenticated user
- Uses **ThreadLocal** by default for thread-safe access
- Available throughout request processing thread
- Automatically cleared after request completion

Usage Example:

```
// Setting security context
SecurityContext context = SecurityContextHolder.createEmptyContext();
Authentication auth = new UsernamePasswordAuthenticationToken(
    "username", "password", authorities);
context.setAuthentication(auth);
SecurityContextHolder.setContext(context);

// Retrieving current user information
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

ThreadLocal Storage Strategy

Why ThreadLocal? [\[69\]](#) [\[55\]](#) [\[68\]](#)

- Each thread has its own SecurityContext copy
- No cross-thread interference
- Accessible anywhere in the same request thread
- Automatic cleanup prevents memory leaks

Alternative Storage Strategies: [\[55\]](#) [\[69\]](#) [\[68\]](#)

- MODE_GLOBAL: All threads share same context (Swing applications)
- MODE_INHERITABLETHREADLOCAL: Child threads inherit parent's context
- MODE_THREADLOCAL: Default thread-local storage

User Representation

UserDetails Interface

Represents user information in Spring Security. [\[70\]](#) [\[71\]](#) [\[72\]](#)

Core Methods: [\[72\]](#) [\[70\]](#)

```
public interface UserDetails {  
    String getUsername();  
    String getPassword();  
    Collection<? extends GrantedAuthority> getAuthorities();  
    boolean isAccountNonExpired();  
    boolean isAccountNonLocked();  
    boolean isCredentialsNonExpired();  
    boolean isEnabled();  
}
```

UserDetailsService Interface

Loads user-specific data from various sources. [\[71\]](#) [\[73\]](#) [\[70\]](#)

Single Method: [\[70\]](#) [\[71\]](#)

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException;  
}
```

Implementation Example: [\[72\]](#) [\[70\]](#)

```
@Service  
public class CustomUserDetailsService implements UserDetailsService {
```

```

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found: " + username);
        }

        return new CustomUserDetails(
            user.getUsername(),
            user.getPassword(),
            user.getAuthorities()
        );
    }
}

```

Hands-on Exercise: Basic Setup

Let's create a simple Spring Boot application to explore these concepts:

1. Create Project Structure

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

```

2. Basic Controller

```

@RestController
public class SecurityTestController {

    @GetMapping("/public")
    public String publicEndpoint() {
        return "This is a public endpoint";
    }

    @GetMapping("/private")
    public String privateEndpoint() {
        return "This is a private endpoint";
    }

    @GetMapping("/user-info")
    public String userInfo() {
        SecurityContext context = SecurityContextHolder.getContext();

```

```
        Authentication auth = context.getAuthentication();
        return "Authenticated user: " + auth.getName() +
               ", Authorities: " + auth.getAuthorities();
    }
}
```

3. Run and Test

- Start the application
- Notice Spring Security's default configuration
- Try accessing endpoints and observe the behavior
- Check the console for the generated default password

Key Takeaways

1. **Authentication vs Authorization:** Authentication verifies identity, authorization controls access [\[52\]](#) [\[50\]](#) [\[51\]](#)
2. **Filter Chain:** Spring Security uses a chain of filters to process requests [\[62\]](#) [\[63\]](#) [\[64\]](#)
3. **Security Context:** ThreadLocal storage provides thread-safe access to user information [\[68\]](#) [\[55\]](#)
4. **Built-in Protection:** Framework automatically guards against CSRF, XSS, and other common attacks [\[57\]](#) [\[56\]](#) [\[54\]](#)
5. **Extensible Design:** All components can be customized for specific requirements [\[71\]](#) [\[72\]](#)

What's Next?

In **Phase 3**, we'll start implementing these concepts with hands-on configuration, exploring:

- Basic authentication setup
- Custom security configuration
- Password encoding
- In-memory and database-backed user stores

You now have a solid foundation in Spring Security's core concepts and architecture. These fundamentals will serve as the building blocks for all advanced topics we'll cover in subsequent phases.

**

Phase 3: Basic Authentication and Configuration

Let's dive into **Phase 3** where we'll implement Spring Security's authentication mechanisms and learn the modern configuration approach. This phase focuses on hands-on implementation using Spring Security 6.

Modern Security Configuration with Spring Security 6

Spring Security 6 introduced significant changes, replacing the deprecated `WebSecurityConfigurerAdapter` with the `SecurityFilterChain` bean approach.[\[88\]](#) [\[89\]](#) [\[90\]](#)

SecurityFilterChain Configuration

The new approach uses `@Bean` methods returning `SecurityFilterChain` instead of extending classes:[\[89\]](#) [\[90\]](#) [\[88\]](#)

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/public/**").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults())
            .httpBasic(withDefaults());
        return http.build();
    }
}
```

Key Configuration Components

1. Authorization with `authorizeHttpRequests`

The new `authorizeHttpRequests()` method replaces deprecated `authorizeRequests()`:[\[91\]](#) [\[92\]](#)

Request Matching Patterns:[\[93\]](#) [\[91\]](#)

- `requestMatchers("/api/**")` - Path-based matching
- `requestMatchers(HttpServletRequest.GET, "/api/**")` - HTTP method + path
- `requestMatchers(AntPathRequestMatcher.antMatcher("/admin/**"))` - Ant patterns

Authorization Rules:[\[94\]](#) [\[91\]](#)

- `permitAll()` - Allow all users
- `authenticated()` - Require authentication
- `hasRole("USER")` - Require specific role
- `hasAuthority("READ")` - Require specific authority
- `denyAll()` - Deny all access

Example Configuration:[\[91\]](#)

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/", "/home", "/public/**").permitAll()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/api/**").hasAuthority("READ")
            .requestMatchers(HttpMethod.POST, "/api/**").hasAuthority("WRITE")
            .anyRequest().authenticated()
        );
    return http.build();
}

```

2. Multiple SecurityFilterChain Beans

You can define multiple filter chains for different URL patterns:[\[88\]](#)

```

@Configuration
@EnableWebSecurity
public class MultipleFilterChainConfig {

    @Bean
    @Order(1) // Higher priority
    public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/api/**")
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().hasRole("API_USER")
            )
            .httpBasic(withDefaults());
        return http.build();
    }

    @Bean
    @Order(2) // Lower priority
    public SecurityFilterChain webFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/**")
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/login", "/signup").permitAll()
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults());
        return http.build();
    }
}

```

Password Encoding with BCrypt

Understanding BCrypt

BCrypt is an adaptive hashing function designed for password storage:[\[95\]](#) [\[96\]](#) [\[97\]](#)

Key Features: [\[97\]](#) [\[98\]](#) [\[95\]](#)

- **Salt Integration:** Random salt generated for each password
- **Adaptive Cost:** Configurable rounds to increase computation time
- **Rainbow Table Protection:** Salt prevents precomputed attacks
- **Consistent Length:** Always produces 60-character strings

BCrypt Structure: [\[96\]](#) [\[95\]](#)

The diagram illustrates the structure of a BCrypt hash value. It shows the string '\$2a\$10\$N9qo8uL0ickgx2ZMRZoMyeIjZAgcf17p921dGxad68LJZdL17lhWy' with various parts highlighted by brackets and arrows:

- Algorithm version (\$2a)**: The first two characters, '\$2a'.
- Cost parameter (10 = 2^10 rounds)**: The next two characters, '10', which represent 2¹⁰ rounds of computation.
- Hash value (31 chars)**: The remaining 31 characters, 'N9qo8uL0ickgx2ZMRZoMyeIjZAgcf17p921dGxad68LJZdL17lhWy'.

Implementing BCrypt in Spring Security

1. Password Encoder Bean: [\[90\]](#) [\[95\]](#)

```
@Configuration
public class SecurityConfig {

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

2. Different BCrypt Configurations: [\[95\]](#) [\[96\]](#)

```
// Default configuration (strength 10, version $2a)
BCryptPasswordEncoder defaultEncoder = new BCryptPasswordEncoder();

// Custom strength (computation cost)
BCryptPasswordEncoder strongEncoder = new BCryptPasswordEncoder(12);

// Custom version and strength
BCryptPasswordEncoder customEncoder = new BCryptPasswordEncoder(
    BCryptPasswordEncoder.BCryptVersion.$2B, 8);
```

3. Encoding and Verification Example: [\[96\]](#) [\[95\]](#)

```

@Service
public class UserService {

    @Autowired
    private PasswordEncoder passwordEncoder;

    public User registerUser(String username, String plainPassword) {
        String encodedPassword = passwordEncoder.encode(plainPassword);
        User user = new User(username, encodedPassword);
        return userRepository.save(user);
    }

    public boolean verifyPassword(String plainPassword, String encodedPassword) {
        return passwordEncoder.matches(plainPassword, encodedPassword);
    }
}

```

In-Memory Authentication

In-memory authentication stores user credentials in application memory, useful for development and testing. [\[99\]](#) [\[100\]](#) [\[90\]](#)

Basic In-Memory Setup

1. Simple Configuration: [\[99\]](#) [\[90\]](#)

```

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.builder()
            .username("user")
            .password("{bcrypt}$2a$10$GRLdNijSQMUv1/au9ofL.eDwmoohzzS7.rmNSJZ.0Fx0/BTk76f")
            .roles("USER")
            .build();

        UserDetails admin = User.builder()
            .username("admin")
            .password("{bcrypt}$2a$10$GRLdNijSQMUv1/au9ofL.eDwmoohzzS7.rmNSJZ.0Fx0/BTk76f")
            .roles("USER", "ADMIN")
            .build();

        return new InMemoryUserDetailsManager(user, admin);
    }
}

```

2. Using PasswordEncoder: [\[90\]](#)

```

@Bean
public UserDetailsService userDetailsService() {

```

```

UserDetails user = User.builder()
    .username("john")
    .password(passwordEncoder().encode("password123"))
    .roles("USER")
    .build();

UserDetails manager = User.builder()
    .username("jane")
    .password(passwordEncoder().encode("manager456"))
    .roles("USER", "MANAGER")
    .authorities("READ", "WRITE", "DELETE")
    .build();

return new InMemoryUserDetailsManager(user, manager);
}

```

When to Use In-Memory Authentication

Appropriate Use Cases: [\[100\]](#) [\[101\]](#)

- **Development and Testing:** Quick setup for prototyping
- **Small Applications:** Fixed set of users (admin panels)
- **Demo Applications:** Showcasing features without database complexity
- **Unit Testing:** Predictable user scenarios

Limitations: [\[101\]](#) [\[100\]](#)

- **Non-Persistent:** Users lost when application restarts
- **Static Users:** Cannot add/modify users at runtime
- **Not Scalable:** Impractical for large user bases
- **Security Concerns:** Credentials visible in code/configuration

Database Authentication with JPA

Database authentication provides persistent, scalable user management. [\[102\]](#) [\[103\]](#) [\[104\]](#)

Setting Up Database Authentication

1. JPA Entity Implementation: [\[102\]](#)

```

@Entity
@Table(name = "users")
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String username;

```

```

private String password;

private boolean enabled = true;
private boolean accountNonExpired = true;
private boolean accountNonLocked = true;
private boolean credentialsNonExpired = true;

@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "user_roles",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "role_id"))
private Set<Role> roles = new HashSet<>();

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return roles.stream()
        .map(role -> new SimpleGrantedAuthority("ROLE_" + role.getName()))
        .collect(Collectors.toList());
}

// Implement other UserDetails methods...
}

```

2. JPA Repository: [\[103\]](#) [\[102\]](#)

```

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

    @Query("SELECT u FROM User u WHERE u.username = ?1")
    User getUserByUsername(String username);
}

```

3. Custom UserDetailsService Implementation: [\[104\]](#) [\[103\]](#) [\[102\]](#)

```

@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(
                "User not found: " + username));

        return new CustomUserDetails(user);
    }
}

```

4. UserDetails Wrapper:[\[103\]](#) [\[102\]](#)

```
public class CustomUserDetails implements UserDetails {  
  
    private User user;  
  
    public CustomUserDetails(User user) {  
        this.user = user;  
    }  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return user.getRoles().stream()  
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role.getName()))  
            .collect(Collectors.toList());  
    }  
  
    @Override  
    public String getPassword() {  
        return user.getPassword();  
    }  
  
    @Override  
    public String getUsername() {  
        return user.getUsername();  
    }  
  
    // Implement remaining methods using user fields...  
}
```

5. Security Configuration:[\[102\]](#) [\[103\]](#)

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
  
    @Autowired  
    private CustomUserDetailsService userDetailsService;  
  
    @Bean  
    public PasswordEncoder passwordEncoder() {  
        return new BCryptPasswordEncoder();  
    }  
  
    @Bean  
    public AuthenticationProvider authenticationProvider() {  
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();  
        provider.setUserDetailsService(userDetailsService);  
        provider.setPasswordEncoder(passwordEncoder());  
        return provider;  
    }  
  
    @Bean  
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
```

```

http
    .authorizeHttpRequests(authorize -> authorize
        .requestMatchers("/register", "/login").permitAll()
        .requestMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated()
    )
    .formLogin(form -> form
        .loginPage("/login")
        .defaultSuccessUrl("/dashboard")
        .failureUrl("/login?error=true")
        .permitAll()
    )
    .logout(logout -> logout
        .logoutUrl("/logout")
        .logoutSuccessUrl("/login?logout=true")
        .permitAll()
    );
    return http.build();
}
}

```

Custom Login Form

Implementing custom login forms provides better user experience and branding. [\[105\]](#) [\[106\]](#) [\[107\]](#)

Creating Custom Login Form

1. Login Controller:[\[106\]](#) [\[105\]](#)

```

@Controller
public class LoginController {

    @GetMapping("/login")
    public String showLoginPage(@RequestParam(value = "error", required = false) String error,
                               @RequestParam(value = "logout", required = false) String logout,
                               Model model) {
        if (error != null) {
            model.addAttribute("errorMessage", "Invalid username or password!");
        }
        if (logout != null) {
            model.addAttribute("logoutMessage", "You have been logged out successfully.");
        }
        return "login";
    }
}

```

2. Thymeleaf Login Template:[\[107\]](#) [\[108\]](#)

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Login Page</title>
    <link href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css" rel="stylesheet"/>

```

```

</head>
<body class="bg-gray-100">
    <div class="min-h-screen flex items-center justify-center">
        <div class="max-w-md w-full space-y-8">
            <div>
                <h2 class="mt-6 text-center text-3xl font-extrabold text-gray-900">
                    Sign in to your account
                </h2>
            </div>

            <!-- Error Message -->
            <div th:if="${errorMessage}" class="bg-red-100 border border-red-400 text-red-100 p-4 mb-4">
                <span th:text="${errorMessage}"></span>
            </div>

            <!-- Success Message -->
            <div th:if="${logoutMessage}" class="bg-green-100 border border-green-400 text-green-100 p-4 mb-4">
                <span th:text="${logoutMessage}"></span>
            </div>

            <form class="mt-8 space-y-6" th:action="@{/login}" method="post">
                <div class="rounded-md shadow-sm -space-y-px">
                    <div>
                        <label for="username" class="sr-only">Username</label>
                        <input id="username" name="username" type="text" required
                               class="relative block w-full px-3 py-2 border border-gray-300 placeholder-underline placeholder='Username'"/>
                    </div>
                    <div>
                        <label for="password" class="sr-only">Password</label>
                        <input id="password" name="password" type="password" required
                               class="relative block w-full px-3 py-2 border border-gray-300 placeholder-underline placeholder='Password'"/>
                    </div>
                </div>
                <div>
                    <button type="submit"
                           class="group relative w-full flex justify-center py-2 px-4 border border-gray-300 rounded-md font-medium text-white bg-blue-600 hover:bg-blue-700 transition-colors duration-200 ease-in-out">
                        Sign in
                    </button>
                </div>
            </form>
        </div>
    </div>
</body>
</html>

```

3. Security Configuration for Custom Form: [105] [106] [107]

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/login", "/css/**", "/js/**").permitAll()

```

```

        .anyRequest().authenticated()
    )
    .formLogin(form -> form
        .loginPage("/login")                      // Custom login page URL
        .loginProcessingUrl("/perform_login")     // Form submission URL
        .usernameParameter("username")           // Username field name
        .passwordParameter("password")           // Password field name
        .defaultSuccessUrl("/dashboard", true)   // Success redirect
        .failureUrl("/login?error=true")         // Failure redirect
        .permitAll()
    )
    .logout(logout -> logout
        .logoutUrl("/logout")
        .logoutSuccessUrl("/login?logout=true")
        .invalidateHttpSession(true)
        .deleteCookies("JSESSIONID")
        .permitAll()
    );
    return http.build();
}

```

Hands-on Projects

Let's implement three progressive projects to solidify your understanding:

Project 1: Basic In-Memory Authentication

```

@SpringBootApplication
@EnableWebSecurity
public class BasicSecurityApp {

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

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

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.builder()
            .username("user")
            .password(passwordEncoder().encode("password"))
            .roles("USER")
            .build();

        UserDetails admin = User.builder()
            .username("admin")
            .password(passwordEncoder().encode("admin"))
            .roles("ADMIN", "USER")
            .build();
    }
}

```

```

        return new InMemoryUserDetailsManager(user, admin);
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/", "/home").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .formLogin(withDefaults())
            .httpBasic(withDefaults());
        return http.build();
    }
}

```

Project 2: Custom Login Form with Database Authentication

Create a complete Spring Boot application with:

- JPA entities for User and Role
- Custom UserDetailsService implementation
- Custom login/logout pages with Thymeleaf
- Registration functionality with password encoding
- Role-based access control

Project 3: User Registration System

Build upon Project 2 to add:

- User registration form
- Email validation
- Password strength requirements
- Account activation workflow
- Remember-me functionality

Key Takeaways

- 1. Modern Configuration:** Spring Security 6 uses SecurityFilterChain beans instead of extending classes [\[89\]](#) [\[88\]](#)
- 2. BCrypt is Essential:** Always use BCrypt for password encoding in production [\[98\]](#) [\[97\]](#) [\[95\]](#)
- 3. Progressive Authentication:** Start with in-memory, move to database for production [\[99\]](#) [\[102\]](#)
- 4. Custom Forms:** Implement custom login forms for better user experience [\[107\]](#) [\[105\]](#)

5. Security by Default: Configure authorization rules that deny by default and permit explicitly^[91]

What's Next?

In **Phase 4**, we'll explore advanced authentication mechanisms including:

- JWT (JSON Web Tokens) implementation
- OAuth2 and OpenID Connect integration
- Social login with third-party providers
- Stateless authentication patterns

You now have a solid foundation in basic Spring Security configuration and authentication mechanisms. These skills form the building blocks for more advanced security patterns we'll cover next.

**

before moving phase 4 i would like to understand What is AuthenticationProvider? and how do we map roles and autorites together for an end point like role is USER and authority is read ?

what is httpBasic()? why do we need this?

Excellent questions! These are fundamental concepts that every Spring Security developer must understand. Let me break down each concept clearly.

Understanding AuthenticationProvider

What is AuthenticationProvider?

The **AuthenticationProvider** is a core interface in Spring Security that defines how authentication should be processed. It's essentially a contract that says: "Given some credentials, verify if they're valid and return an authenticated user."^{[126] [127] [128]}

AuthenticationProvider Interface Structure

The interface has two main methods:^[128]

```
public interface AuthenticationProvider {  
    // Main authentication method  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
  
    // Check if this provider supports the authentication type
```

```
        boolean supports(Class<?> authentication);  
    }
```

How AuthenticationProvider Works in the Chain

Here's the complete authentication flow:[\[127\]](#) [\[126\]](#)

1. **User submits credentials** (username/password)
2. **Authentication Filter** creates `UsernamePasswordAuthenticationToken`
3. **AuthenticationManager** (usually `ProviderManager`) receives the token
4. **ProviderManager** iterates through registered `AuthenticationProviders`
5. **AuthenticationProvider** (like `DaoAuthenticationProvider`) processes authentication
6. **Successful authentication** creates authenticated `Authentication` object
7. **SecurityContextHolder** stores the authentication for the session

DaoAuthenticationProvider - The Most Common Implementation

DaoAuthenticationProvider is Spring Security's standard implementation:[\[129\]](#) [\[126\]](#) [\[127\]](#)

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
  
    @Autowired  
    private UserDetailsService userDetailsService;  
  
    @Bean  
    public PasswordEncoder passwordEncoder() {  
        return new BCryptPasswordEncoder();  
    }  
  
    @Bean  
    public DaoAuthenticationProvider authenticationProvider() {  
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();  
        provider.setUserDetailsService(userDetailsService);  
        provider.setPasswordEncoder(passwordEncoder());  
        return provider;  
    }  
  
    // In Spring Security 6, authentication providers are auto-configured  
    // when you provide UserDetailsService and PasswordEncoder beans  
}
```

What DaoAuthenticationProvider Does:[\[126\]](#) [\[127\]](#)

1. **Extracts username** from `UsernamePasswordAuthenticationToken`
2. **Calls UserDetailsService** to load user from database/memory
3. **Compares passwords** using `PasswordEncoder.matches()`

4. Creates authenticated token with user details and authorities
5. Returns the authenticated token

Multiple AuthenticationProviders

You can configure multiple providers for different authentication methods:[\[130\]](#) [\[128\]](#)

```
@Configuration
public class MultiProviderConfig {

    @Bean
    public AuthenticationManager authenticationManager(
        HttpSecurity http,
        DaoAuthenticationProvider daoProvider,
        LdapAuthenticationProvider ldapProvider) throws Exception {

        AuthenticationManagerBuilder builder =
            http.getSharedObject(AuthenticationManagerBuilder.class);

        builder.authenticationProvider(daoProvider)
            .authenticationProvider(ldapProvider);

        return builder.build();
    }
}
```

Understanding Roles vs Authorities

This is one of the most confusing aspects of Spring Security. Let me clarify the relationship.[\[131\]](#) [\[132\]](#) [\[133\]](#)

Core Concepts

Authority (GrantedAuthority)

- **Definition:** A permission or right that a user has[\[132\]](#) [\[134\]](#) [\[133\]](#)
- **Format:** Plain string like "READ_PRODUCT", "WRITE_USER", "DELETE_ORDER"[\[132\]](#)
- **Granular:** Fine-grained permissions for specific actions[\[133\]](#) [\[135\]](#)

Role

- **Definition:** A collection of authorities with ROLE_ prefix[\[134\]](#) [\[133\]](#) [\[132\]](#)
- **Format:** String prefixed with "ROLE_" like "ROLE_ADMIN", "ROLE_USER"[\[133\]](#) [\[132\]](#)
- **Coarse-grained:** High-level groupings of permissions[\[131\]](#) [\[133\]](#)

The Key Relationship

A Role is just a special type of Authority with ROLE_ prefix: [\[134\]](#) [\[132\]](#)

```
// These are equivalent:  
hasRole("ADMIN")           // Spring automatically adds ROLE_ prefix  
hasAuthority("ROLE_ADMIN") // You specify the full authority name  
  
// These are NOT equivalent:  
hasRole("READ_PRODUCT")    // Looks for "ROLE_READ_PRODUCT" - WON'T WORK  
hasAuthority("READ_PRODUCT") // Looks for exactly "READ_PRODUCT" - WORKS
```

Practical Example: Role + Authority Mapping

Here's how to map roles and authorities together for an endpoint where role is USER and authority is READ: [\[136\]](#) [\[131\]](#) [\[133\]](#)

Scenario: E-commerce Application

User Entity with Both Roles and Authorities:

```
@Entity  
public class User implements UserDetails {  
    // ... other fields  
  
    @ManyToMany(fetch = FetchType.EAGER)  
    private Set<Role> roles = new HashSet<>();  
  
    @ManyToMany(fetch = FetchType.EAGER)  
    private Set<Authority> authorities = new HashSet<>();  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        Set<SimpleGrantedAuthority> grantedAuthorities = new HashSet<>();  
  
        // Add roles with ROLE_ prefix  
        for (Role role : roles) {  
            grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_" + role.getName()));  
        }  
  
        // Add direct authorities  
        for (Authority authority : authorities) {  
            grantedAuthorities.add(new SimpleGrantedAuthority(authority.getName()));  
        }  
  
        return grantedAuthorities;  
    }  
}
```

Role Entity:

```

@Entity
public class Role {
    private String name; // "USER", "ADMIN", "MANAGER"

    @ManyToMany
    private Set<Authority> authorities = new HashSet<>();
    // ... getters/setters
}

```

Authority Entity:

```

@Entity
public class Authority {
    private String name; // "READ", "WRITE", "DELETE", "MANAGE_USERS"
    // ... getters/setters
}

```

Security Configuration with Combined Role + Authority Check

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            // Require both USER role AND READ authority
            .requestMatchers(HttpMethod.GET, "/products/**")
            .access("hasRole('USER') and hasAuthority('READ')")

            // Alternative: Use method-level security
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/api/write/**").hasAuthority("WRITE")
            .anyRequest().authenticated()
        );
    return http.build();
}

```

Method-Level Security Example

```

@RestController
public class ProductController {

    // Requires USER role AND READ authority
    @PreAuthorize("hasRole('USER') and hasAuthority('READ')")
    @GetMapping("/products")
    public List<Product> getProducts() {
        return productService.findAll();
    }

    // Requires ADMIN role OR has MANAGE_PRODUCTS authority
    @PreAuthorize("hasRole('ADMIN') or hasAuthority('MANAGE_PRODUCTS')")
    @PostMapping("/products")
}

```

```

public Product createProduct(@RequestBody Product product) {
    return productService.save(product);
}

// Complex authorization - USER role with READ authority,
// or MANAGER role, or specific PRODUCT_ADMIN authority
@PreAuthorize("(hasRole('USER') and hasAuthority('READ')) or " +
    "hasRole('MANAGER') or " +
    "hasAuthority('PRODUCT_ADMIN')")
@GetMapping("/products/{id}")
public Product getProduct(@PathVariable Long id) {
    return productService.findById(id);
}
}

```

Role Hierarchy for Inherited Permissions

You can also set up role hierarchies where higher roles inherit lower role permissions: [\[137\]](#) [\[138\]](#)

```

@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl hierarchy = new RoleHierarchyImpl();
    hierarchy.setHierarchy(
        "ROLE_ADMIN > ROLE_MANAGER > ROLE_USER > ROLE_GUEST"
    );
    return hierarchy;
}

@Bean
public DefaultWebSecurityExpressionHandler expressionHandler() {
    DefaultWebSecurityExpressionHandler handler = new DefaultWebSecurityExpressionHandler();
    handler.setRoleHierarchy(roleHierarchy());
    return handler;
}

```

With this hierarchy:

- ROLE_ADMIN can access anything requiring ROLE_MANAGER, ROLE_USER, or ROLE_GUEST
- ROLE_MANAGER can access anything requiring ROLE_USER or ROLE_GUEST
- ROLE_USER can access anything requiring ROLE_GUEST

Understanding httpBasic()

What is httpBasic()?

The `httpBasic()` method enables **HTTP Basic Authentication** in your Spring Security configuration. [\[139\]](#) [\[140\]](#) [\[141\]](#)

HTTP Basic Authentication Explained

HTTP Basic Authentication is a simple authentication scheme built into the HTTP protocol:[\[142\]](#)
[\[143\]](#) [\[139\]](#)

1. **Client requests** a protected resource
2. **Server responds** with 401 Unauthorized and WWW-Authenticate: Basic header
3. **Browser shows login popup** (or client includes auth header)
4. **Credentials sent** in every request via Authorization header
5. **Format:** Authorization: Basic base64(username:password)

Example of HTTP Basic Auth Header

```
GET /api/products HTTP/1.1
Host: example.com
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Where dXNlcm5hbWU6cGFzc3dvcmQ= is base64 encoding of username:password.[\[144\]](#) [\[139\]](#)

How to Enable httpBasic() in Spring Security

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/**").authenticated()
            .anyRequest().permitAll()
        )
        .httpBasic(withDefaults()); // Enables HTTP Basic Authentication
    return http.build();
}
```

Custom HTTP Basic Configuration

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .httpBasic(basic -> basic
            .realmName("My Application")
            .authenticationEntryPoint(customAuthenticationEntryPoint())
        );
    return http.build();
}
```

Why Do We Need httpBasic()?

Use Cases for HTTP Basic Authentication

1. API Authentication:[\[141\]](#) [\[143\]](#) [\[144\]](#)

```
// Perfect for REST APIs accessed by other applications
@RestController
public class ApiController {

    @GetMapping("/api/data")
    public ResponseEntity<String> getData() {
        return ResponseEntity.ok("Protected API data");
    }
}
```

2. Simple Applications:[\[141\]](#)

- Quick prototyping without complex login forms
- Internal tools with basic security needs
- Legacy system integration

3. Microservices Communication:[\[145\]](#)

```
// Service-to-service authentication
@Bean
@Order(1)
public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
    http
        .securityMatcher("/api/**")
        .authorizeHttpRequests(authorize -> authorize
            .anyRequest().hasRole("SERVICE"))
        .httpBasic(withDefaults())
        .sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    return http.build();
}
```

HTTP Basic vs Form-Based Authentication

Aspect	HTTP Basic	Form-Based
User Interface	Browser popup	Custom HTML form
Customization	Limited browser-specific	Full control over design
Session Management	Stateless	Session-based
Logout	Not supported	Proper logout functionality
Security	Credentials in every request	Session tokens

Aspect	HTTP Basic	Form-Based
Use Case	APIs, simple apps	Web applications

Security Considerations

Advantages: [\[143\]](#) [\[144\]](#)

- **Simple implementation** - No login forms needed
- **Stateless** - No session management required
- **Universal support** - Works with all HTTP clients

Disadvantages: [\[142\]](#) [\[143\]](#) [\[144\]](#)

- **Credentials exposed** - Base64 is encoding, not encryption
- **No logout** - Must close browser to "logout"
- **Brute force vulnerable** - No built-in rate limiting
- **Requires HTTPS** - Essential to prevent credential interception

Complete Example: API with Basic Authentication

```

@Configuration
@EnableWebSecurity
public class ApiSecurityConfig {

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

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails apiUser = User.builder()
            .username("api-client")
            .password(passwordEncoder().encode("secret123"))
            .roles("API_USER")
            .authorities("READ", "WRITE")
            .build();

        return new InMemoryUserDetailsManager(apiUser);
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/public").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/**")
                .hasAuthority("READ")
                .requestMatchers(HttpMethod.POST, "/api/**")
                .hasAuthority("WRITE")
                .anyRequest().authenticated())
    }
}

```

```

        )
        .httpBasic(basic -> basic
            .realmName("API Access")
        )
        .sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .csrf(csrf -> csrf.disable()); // Disable CSRF for APIs

    return http.build();
}
}

```

Testing HTTP Basic Authentication

Using curl:

```

# Successful authentication
curl -u username:password http://localhost:8080/api/data

# Or with explicit header
curl -H "Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=" http://localhost:8080/api/data

```

Using Postman:

1. Select "Authorization" tab
2. Choose "Basic Auth" type
3. Enter username and password
4. Postman automatically creates the Authorization header

Key Takeaways

1. **AuthenticationProvider:** Defines how credentials are verified, with DaoAuthenticationProvider being the most common implementation [\[127\]](#) [\[126\]](#)
2. **Roles vs Authorities:** Roles are authorities with ROLE_ prefix. Use hasRole("USER") for roles, hasAuthority("READ") for specific permissions [\[132\]](#) [\[133\]](#)
3. **Combined Authorization:** You can combine roles and authorities: hasRole('USER') and hasAuthority('READ') [\[146\]](#) [\[136\]](#)
4. **httpBasic():** Enables HTTP Basic Authentication - perfect for APIs and simple applications, but requires HTTPS in production [\[139\]](#) [\[143\]](#)
5. **Security Choice:** Use HTTP Basic for APIs/services, Form-based for web applications with user interfaces [\[147\]](#) [\[148\]](#)

These concepts form the foundation for all advanced Spring Security features we'll explore in Phase 4!



few question here? when defining access control scope we give with hasRole(User) why not with hasRole(Role_user) same in method security level?

for basic authentication which is Authentication provider? what do you mean by basic->realName()?

wild cards pattern understanding

Perfect questions! These are exactly the kind of details that help you truly understand Spring Security. Let me explain each concept clearly with practical examples.

1. Why hasRole("USER") instead of hasRole("ROLE_USER")?

Automatic ROLE_ Prefix Addition

Spring Security **automatically adds the ROLE_ prefix** when you use hasRole() method. This is a convenience feature that has been consistent since Spring Security 4.[\[164\]](#) [\[165\]](#) [\[166\]](#)

The Internal Mechanism

When you write:

```
.requestMatchers("/admin/**").hasRole("ADMIN")
```

Spring Security internally converts this to:

```
.requestMatchers("/admin/**").hasAuthority("ROLE_ADMIN")
```

The framework automatically prepends ROLE_ to whatever you specify.[\[166\]](#) [\[164\]](#)

Historical Context and Design Decision

Spring Security 3 vs 4+ Behavior: [\[164\]](#)

- **Spring Security 3:** hasRole('ROLE_XYZ') was the same as hasAuthority('ROLE_XYZ')
- **Spring Security 4+:** hasRole('XYZ') is the same as hasAuthority('ROLE_XYZ')

Why this design?: [\[167\]](#) [\[164\]](#)

- **Convenience:** Avoid repetitive ROLE_ typing
- **Consistency:** Role-based operations always use the prefix
- **Separation of Concerns:** Distinguish between roles (high-level) and authorities (granular)

Practical Examples

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                // These are EQUIVALENT
                .requestMatchers("/admin/**").hasRole("ADMIN")           // Looks for "ROLE_ADMIN"
                .requestMatchers("/user/**").hasAuthority("ROLE_USER") // Looks for "ROLE_USER")

                // These are DIFFERENT
                .requestMatchers("/manager/**").hasRole("MANAGER")      // Looks for "ROLE_MANAGER"
                .requestMatchers("/read/**").hasAuthority("READ")       // Looks for "READ"

                .anyRequest().authenticated()
            );
        return http.build();
    }
}
```

Method-Level Security - Same Behavior

```
@RestController
public class AdminController {

    // These annotations are EQUIVALENT
    @PreAuthorize("hasRole('ADMIN')")           // Spring adds ROLE_ prefix
    @PreAuthorize("hasAuthority('ROLE_ADMIN')") // You specify full authority name
    @GetMapping("/admin/users")
    public List<User> getUsers() {
        return userService.findAll();
    }

    // These are DIFFERENT
    @PreAuthorize("hasRole('MANAGER')")          // Looks for "ROLE_MANAGER"
    @PreAuthorize("hasAuthority('DELETE')")        // Looks for "DELETE" (no prefix)
    @DeleteMapping("/admin/users/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.delete(id);
    }
}
```

How to Remove ROLE_ Prefix (If Needed)

If you want to remove the automatic prefix behavior:[\[165\]](#) [\[168\]](#)

```
@Bean
public DefaultWebSecurityExpressionHandler webSecurityExpressionHandler() {
    DefaultWebSecurityExpressionHandler handler = new DefaultWebSecurityExpressionHandler();
    handler.setDefaultRolePrefix(""); // Remove ROLE_ prefix
    return handler;
}

@Bean
public DefaultMethodSecurityExpressionHandler methodSecurityExpressionHandler() {
    DefaultMethodSecurityExpressionHandler handler = new DefaultMethodSecurityExpressionHandler();
    handler.setDefaultRolePrefix(""); // Remove ROLE_ prefix
    return handler;
}
```

2. Basic Authentication Provider and realName()

Which Authentication Provider for Basic Authentication?

For HTTP Basic Authentication, Spring Security uses the **standard authentication flow** with these components:[\[169\]](#) [\[170\]](#) [\[171\]](#)

1. BasicAuthenticationFilter

The **BasicAuthenticationFilter** processes HTTP requests with Basic authorization headers:[\[169\]](#)

```
// Internal process of BasicAuthenticationFilter
public class BasicAuthenticationFilter extends OncePerRequestFilter {

    private AuthenticationManager authenticationManager;
    private AuthenticationEntryPoint authenticationEntryPoint;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws IOException, ServletException {

        String header = request.getHeader("Authorization");

        if (header != null && header.startsWith("Basic ")) {
            // Extract and decode username:password
            String token = header.substring(6);
            String decoded = new String(Base64.getDecoder().decode(token));
            String[] credentials = decoded.split(":", 2);

            // Create authentication token
            UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(credentials[^5_0], credentials[^5_1]);
            ...
        }
    }
}
```

```

        // Delegate to AuthenticationManager
        Authentication result = authenticationManager.authenticate(authToken);

        // Store in SecurityContext
        SecurityContextHolder.getContext().setAuthentication(result);
    }

    chain.doFilter(request, response);
}
}

```

2. The Authentication Provider Chain

Basic Authentication uses the same provider as form login - typically DaoAuthenticationProvider: [\[170\]](#) [\[172\]](#)

```

HTTP Basic Request with Authorization Header
    ↓
BasicAuthenticationFilter extracts credentials
    ↓
Creates UsernamePasswordAuthenticationToken
    ↓
AuthenticationManager (ProviderManager)
    ↓
DaoAuthenticationProvider.authenticate()
    ↓
UserDetailsService.loadUserByUsername()
    ↓
PasswordEncoder.matches() verification
    ↓
Returns authenticated Authentication object

```

Complete Basic Authentication Configuration

```

@Configuration
@EnableWebSecurity
public class BasicAuthConfig {

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

    @Bean
    public UserDetailsService userDetailsService() {
        // This will be used by DaoAuthenticationProvider
        UserDetails user = User.builder()
            .username("api-user")
            .password(passwordEncoder().encode("secret123"))
            .roles("API_USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}

```

```

    }

    // DaoAuthenticationProvider is auto-configured when
    // UserDetailsService and PasswordEncoder beans are present

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/**").hasRole("API_USER")
                .anyRequest().authenticated()
            )
            .httpBasic(basic -> basic
                .realmName("API Realm") // This sets the realm name
            )
            .sessionManagement(session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        return http.build();
    }
}

```

What is `realmName()` in Basic Authentication?

The **realm name** is a string that identifies the protected area on the server. It's part of the HTTP Basic Authentication specification.[\[173\]](#) [\[174\]](#) [\[171\]](#)

HTTP Response with Realm

When authentication fails, the server sends:

```

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="API Realm"
Content-Type: text/html

```

Purpose of Realm Name

1. User Information:[\[174\]](#) [\[173\]](#)

- Tells users **what they're accessing**
- Helps users choose **appropriate credentials**
- **Browser displays** this in the login popup

2. Security Boundaries:[\[174\]](#)

- **Logically separates** different protected areas
- **Same server** can have multiple realms
- Each realm can have **different user databases**

Real-World Example: Multiple Realms

```
@Configuration
@EnableWebSecurity
public class MultiRealmConfig {

    @Bean
    @Order(1)
    public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/api/**")
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().hasRole("API_USER"))
            )
            .httpBasic(basic -> basic
                .realmName("REST API Realm") // Different realm for API
                .authenticationEntryPoint(apiAuthenticationEntryPoint()))
        );
        return http.build();
    }

    @Bean
    @Order(2)
    public SecurityFilterChain adminFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/admin/**")
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().hasRole("ADMIN"))
            )
            .httpBasic(basic -> basic
                .realmName("Admin Panel") // Different realm for admin
                .authenticationEntryPoint(adminAuthenticationEntryPoint()))
        );
        return http.build();
    }

    @Bean
    public BasicAuthenticationEntryPoint apiAuthenticationEntryPoint() {
        BasicAuthenticationEntryPoint entryPoint = new BasicAuthenticationEntryPoint();
        entryPoint.setRealmName("REST API Realm");
        return entryPoint;
    }

    @Bean
    public BasicAuthenticationEntryPoint adminAuthenticationEntryPoint() {
        BasicAuthenticationEntryPoint entryPoint = new BasicAuthenticationEntryPoint();
        entryPoint.setRealmName("Admin Panel");
        return entryPoint;
    }
}
```

Custom Authentication Entry Point

```
@Component
public class CustomBasicAuthEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request,
                         HttpServletResponse response,
                         AuthenticationException authException) throws IOException {

        response.setStatus(HttpServletRequest.SC_UNAUTHORIZED);
        response.setHeader("WWW-Authenticate", "Basic realm=\"My Custom API\"");
        response.getWriter().write("Authentication required for My Custom API");
    }
}
```

3. Wildcard Pattern Understanding

Spring Security uses **Ant-style path patterns** for URL matching. Understanding these patterns is crucial for proper endpoint security.[\[175\]](#) [\[176\]](#) [\[177\]](#)

Core Wildcard Characters

Single Character Wildcard: ?

Matches **exactly one character**:[\[176\]](#) [\[177\]](#)

```
.requestMatchers("/user?").permitAll()
// Matches: /user1, /user2, /userA
// Does NOT match: /user, /user12, /users
```

Single Segment Wildcard: *

Matches **zero or more characters within a single path segment**:[\[177\]](#) [\[176\]](#)

```
.requestMatchers("/files/*.txt").permitAll()
// Matches: /files/doc.txt, /files/readme.txt, /files/.txt
// Does NOT match: /files/folder/doc.txt, /files/doc.pdf
```

Multi-Segment Wildcard: **

Matches **zero or more path segments**:[\[178\]](#) [\[176\]](#) [\[177\]](#)

```
.requestMatchers("/admin/**").hasRole("ADMIN")
// Matches: /admin, /admin/, /admin/users, /admin/users/123, /admin/deep/nested/path
```

Important Pattern Rules

Pattern Must Start with /

Critical: Patterns should always start with a forward slash:^[178]

```
// CORRECT
.requestMatchers("/api/**").permitAll()

// PROBLEMATIC - May not work as expected
.requestMatchers("api/**").permitAll()
```

Path Segments vs Characters

Understanding the difference is crucial:^[178]

```
// * matches within ONE segment
"/files/*"          → matches /files/doc.txt
                      → does NOT match /files/folder/doc.txt

// ** matches MULTIPLE segments
"/files/**"         → matches /files/doc.txt
                      → matches /files/folder/doc.txt
                      → matches /files/deep/nested/file.txt
```

Advanced Pattern Examples

Complex Wildcard Combinations

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            // Public access patterns
            .requestMatchers("/", "/home", "/public/**").permitAll()

            // API versioning patterns
            .requestMatchers("/api/v?/public/**").permitAll()      // v1, v2, etc.
            .requestMatchers("/api/v*/users/**").hasRole("USER")   // any version

            // File type patterns
            .requestMatchers("/**/*.css", "/**/*.js", "/**/*.png").permitAll()

            // Admin patterns
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/management/**").hasRole("MANAGER")

            // Specific endpoint patterns
            .requestMatchers(HttpMethod.GET, "/api/**/read").hasAuthority("READ")
            .requestMatchers(HttpMethod.POST, "/api/**/write").hasAuthority("WRITE")
```

```

    .requestMatchers(HttpMethod.DELETE, "/api/**/delete").hasAuthority("DELETE")

    // Catch-all
    .anyRequest().authenticated()
);

return http.build();
}

```

Real-World E-commerce Example

```

@Bean
public SecurityFilterChain ecommerceFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            // Public product browsing
            .requestMatchers("/products/**", "/categories/**").permitAll()
            .requestMatchers("/api/v*/products/**").permitAll()

            // User account management
            .requestMatchers("/account/**").hasRole("USER")
            .requestMatchers("/orders/**").hasRole("USER")
            .requestMatchers("/api/v*/users/*/orders").hasRole("USER")

            // Admin operations
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/api/v*/admin/**").hasRole("ADMIN")

            // Specific API operations
            .requestMatchers(HttpMethod.GET, "/api/**/reports/**").hasAuthority("READ_REPORT")
            .requestMatchers(HttpMethod.POST, "/api/**/inventory/**").hasAuthority("MANAGE_INVENTORY")
            .requestMatchers(HttpMethod.DELETE, "/api/**/users/**").hasAuthority("DELETE_USER")

            // Static resources
            .requestMatchers("/css/**", "/js/**", "/images/**").permitAll()
            .requestMatchers("/favicon.ico", "/robots.txt").permitAll()

            .anyRequest().authenticated()
);
    return http.build();
}

```

Common Pattern Pitfalls and Solutions

Problem 1: Incorrect Wildcard Usage

```

// WRONG - Tries to match ending with "liveness"
new AntPathRequestMatcher("/**liveness")

// CORRECT - Matches any path ending with /liveness
new AntPathRequestMatcher("/**/liveness")

```

Why? The `**` needs to be followed by `/` to properly match path segments. [179]

Problem 2: Order Matters

```
// WRONG ORDER - More specific patterns after general ones
.requestMatchers("/**").authenticated()           // This catches everything!
.requestMatchers("/public/**").permitAll()         // Never reached

// CORRECT ORDER - Specific patterns first
.requestMatchers("/public/**").permitAll()          // Allow public access
.requestMatchers("/**").authenticated()             // Then require auth for everything else
```

Problem 3: HTTP Method Specificity

```
// More precise control with HTTP methods
.requestMatchers(HttpMethod.GET, "/api/**").hasAuthority("READ")
.requestMatchers(HttpMethod.POST, "/api/**").hasAuthority("WRITE")
.requestMatchers(HttpMethod.PUT, "/api/**").hasAuthority("WRITE")
.requestMatchers(HttpMethod.DELETE, "/api/**").hasAuthority("DELETE")
```

Pattern Matching Testing

Here's how to test your patterns:

```
@Test
public void testPatternMatching() {
    AntPathMatcher matcher = new AntPathMatcher();

    // Test single wildcard
    assertTrue(matcher.match("/user?", "/user1"));
    assertFalse(matcher.match("/user?", "/user12"));

    // Test segment wildcard
    assertTrue(matcher.match("/files/*", "/files/doc.txt"));
    assertFalse(matcher.match("/files/*", "/files/folder/doc.txt"));

    // Test multi-segment wildcard
    assertTrue(matcher.match("/admin/**", "/admin/users/123"));
    assertTrue(matcher.match("/api/**/read", "/api/v1/users/read"));
    assertTrue(matcher.match("/api/**/read", "/api/v2/products/123/read"));
}
```

Summary

1. **hasRole() Prefix:** Spring automatically adds `ROLE_` prefix, so use `hasRole("USER")` not `hasRole("ROLE_USER")` [166] [164]
2. **Basic Authentication Provider:** Uses standard `DaoAuthenticationProvider` through `BasicAuthenticationFilter` [170] [169]

3. Realm Name: Identifies the protected area, shown in browser popup, and helps organize multiple security domains [173] [174]

4. Wildcard Patterns: ? = one char, * = one segment, ** = multiple segments. Always start patterns with / [176] [177] [178]

These fundamentals will serve you well as we move into JWT and OAuth2 in Phase 4!

**

what is this? `.securityMatcher("/**")` and `antPattern()??`

Great questions! These are crucial Spring Security concepts that often confuse developers. Let me break down each one clearly.

1. What is `.securityMatcher("/**")`?

`securityMatcher()` - Defining Which Requests Apply to a SecurityFilterChain

The `securityMatcher()` method determines **which HTTP requests should be processed by a specific SecurityFilterChain**. It's like saying "this security configuration only applies to these specific URL patterns." [202] [203] [204]

Basic Concept

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .securityMatcher("/api/**") // This chain ONLY processes /api/** requests
        .authorizeHttpRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .httpBasic(withDefaults());
    return http.build();
}
```

What happens here: [203] [202]

1. Only requests matching `/api/**` will be processed by this security chain
2. Other requests (like `/web/`, `/admin/`) will be **ignored by this chain**
3. Spring Security will look for other chains to handle non-matching requests

securityMatcher("/") - Special Case**

When you use `securityMatcher("/**")`, you're saying: [\[202\]](#)

- **This security chain applies to ALL requests**
- **Any URL pattern** will be processed by this chain
- This is often used as a "**catch-all" chain** with lower priority

```
@Configuration
@EnableWebSecurity
public class MultiChainConfig {

    @Bean
    @Order(1) // Higher priority - checked first
    public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/api/**") // Only /api/** requests
            .authorizeHttpRequests(authorize -> authorize
                .anyRequest().hasRole("API_USER"))
            .httpBasic(withDefaults());
        return http.build();
    }

    @Bean
    @Order(2) // Lower priority - checked second
    public SecurityFilterChain webFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/**") // ALL remaining requests
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated())
            .formLogin(withDefaults());
        return http.build();
    }
}
```

Request Flow Example: [\[205\]](#) [\[202\]](#)

- Request to `/api/users` → **First chain matches** → Uses HTTP Basic auth
- Request to `/web/dashboard` → **First chain doesn't match** → **Second chain matches** → Uses form login
- Request to `/public/info` → **First chain doesn't match** → **Second chain matches** → Permits all

Key Differences: securityMatcher vs requestMatchers

Method	Purpose	Scope
securityMatcher()	Which requests this SecurityFilterChain processes	Entire chain
requestMatchers()	Authorization rules within a chain	Individual rules

Example showing both: [\[204\]](#) [\[202\]](#)

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .securityMatcher("/api/**") // Chain applies to /api/** requests
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/public/**").permitAll() // Within /api/**, allow public
            .requestMatchers("/api/admin/**").hasRole("ADMIN") // Within /api/**, require ADMIN role
            .anyRequest().authenticated() // Other /api/** requests)
        );
    return http.build();
}
```

Default Behavior Without securityMatcher()

If you **don't specify** securityMatcher(), the chain **matches ALL requests**: [\[202\]](#)

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        // No securityMatcher() = matches ALL requests ("/**")
        .authorizeHttpRequests(authorize -> authorize
            .anyRequest().authenticated()
        )
        .formLogin(withDefaults());
    return http.build();
}
```

This is equivalent to:

```
.securityMatcher("/**") // Implicit - matches all requests
```

2. What is antMatcher() - DEPRECATED!

Historical Context and Deprecation

antMatcher() was used in **older Spring Security versions** but has been **DEPRECATED** and **REMOVED** [\[206\]](#) [\[207\]](#) [\[208\]](#).

Evolution of Spring Security Matching Methods

Spring Security 5.x and earlier: [\[206\]](#) [\[208\]](#)

```
// OLD WAY - DON'T USE ANYMORE
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .antMatcher("/api/**")           // DEPRECATED
        .authorizeRequests()           // DEPRECATED
        .antMatchers("/public/**").permitAll() // DEPRECATED
        .anyRequest().authenticated();
}
```

Spring Security 6.x - Modern Approach: [\[207\]](#) [\[208\]](#) [\[206\]](#)

```
// NEW WAY - USE THIS
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .securityMatcher("/api/**")           // Replaces antMatcher()
        .authorizeHttpRequests(authorize -> authorize // Replaces authorizeRequests()
            .requestMatchers("/public/**").permitAll() // Replaces antMatchers()
            .anyRequest().authenticated())
        );
    return http.build();
}
```

Migration Guide: Old vs New

Old Method (Deprecated)	New Method (Spring 6+)	Purpose
http.antMatcher()	http.securityMatcher()	Define chain scope
http.mvcMatcher()	http.securityMatcher()	Define chain scope
http.regexMatcher()	http.securityMatcher()	Define chain scope
.antMatchers()	.requestMatchers()	Authorization rules
.mvcMatchers()	.requestMatchers()	Authorization rules
.regexMatchers()	.requestMatchers()	Authorization rules
.authorizeRequests()	.authorizeHttpRequests()	Authorization config

Complete Migration Example

Before (Spring Security 5.x): [\[208\]](#) [\[206\]](#)

```
@Configuration
@EnableWebSecurity
public class OldSecurityConfig extends WebSecurityConfigurerAdapter {
```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .antMatcher("/api/**") // DEPRECATED
        .authorizeRequests() // DEPRECATED
            .antMatchers("/api/public/**").permitAll() // DEPRECATED
            .antMatchers("/api/admin/**").hasRole("ADMIN") // DEPRECATED
            .anyRequest().authenticated()
        .and()
        .httpBasic();
}
}

```

After (Spring Security 6.x): [\[206\]](#) [\[208\]](#)

```

@Configuration
@EnableWebSecurity
public class ModernSecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/api/**") // NEW
            .authorizeHttpRequests(authorize -> authorize // NEW
                .requestMatchers("/api/public/**").permitAll() // NEW
                .requestMatchers("/api/admin/**").hasRole("ADMIN") // NEW
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
        return http.build();
    }
}

```

Why the Change?

Problems with old approach: [\[209\]](#) [\[206\]](#)

1. **Security vulnerabilities** in URL matching
2. **Inconsistent behavior** between different matchers
3. **Configuration confusion** between chain-level and request-level matching

Benefits of new approach: [\[209\]](#) [\[206\]](#)

1. **More secure** by default
2. **Clearer separation** between chain matching and authorization rules
3. **Better Spring MVC integration**
4. **Unified API** with `requestMatchers()`

Modern requestMatchers() - The Replacement

The new `requestMatchers()` method **automatically chooses the best RequestMatcher**:[\[204\]](#) [\[206\]](#)

Smart Matching Selection

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            // Spring automatically chooses:
            // - MvcRequestMatcher if Spring MVC is present
            // - AntPathRequestMatcher otherwise
            .requestMatchers("/api/**").hasRole("API_USER")

            // You can also be explicit:
            .requestMatchers(new AntPathRequestMatcher("/admin/**")).hasRole("ADMIN")
            .requestMatchers(new RegexRequestMatcher("^/reports/.*", null)).hasAuthority(
                .anyRequest().authenticated()
            );
    return http.build();
}
```

Advanced requestMatchers() Usage

HTTP Method + Path matching:[\[204\]](#)

```
.authorizeHttpRequests(authorize -> authorize
    .requestMatchers(HttpMethod.GET, "/api/**").hasAuthority("READ")
    .requestMatchers(HttpMethod.POST, "/api/**").hasAuthority("WRITE")
    .requestMatchers(HttpMethod.DELETE, "/api/**").hasAuthority("DELETE")
)
```

Multiple patterns:[\[204\]](#)

```
.authorizeHttpRequests(authorize -> authorize
    .requestMatchers("/public/**", "/assets/**", "/error").permitAll()
    .requestMatchers("/admin/**", "/management/**").hasRole("ADMIN")
)
```

Explicit RequestMatcher types:[\[210\]](#) [\[204\]](#)

```
.authorizeHttpRequests(authorize -> authorize
    .requestMatchers(
        new AntPathRequestMatcher("/api/**"),
        new RegexRequestMatcher("^/reports/\d+$", null)
    ).hasRole("USER")
)
```

Complete Modern Configuration Example

Here's a comprehensive example showing modern Spring Security 6 configuration:

```
@Configuration
@EnableWebSecurity
public class ModernSecurityConfig {

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

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.builder()
            .username("user")
            .password(passwordEncoder().encode("password"))
            .roles("USER")
            .build();

        UserDetails admin = User.builder()
            .username("admin")
            .password(passwordEncoder().encode("admin"))
            .roles("ADMIN", "USER")
            .build();

        return new InMemoryUserDetailsManager(user, admin);
    }

    // API-specific security chain
    @Bean
    @Order(1)
    public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/api/**") // This chain handles /api/** requests
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/**").hasAuthority("READ")
                .requestMatchers(HttpMethod.POST, "/api/**").hasAuthority("WRITE")
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults())
            .sessionManagement(session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(csrf -> csrf.disable());
        return http.build();
    }

    // Web application security chain
    @Bean
    @Order(2)
    public SecurityFilterChain webFilterChain(HttpSecurity http) throws Exception {
        http
            .securityMatcher("/**") // This chain handles ALL other requests
```

```

        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/", "/home", "/public/**").permitAll()
            .requestMatchers("/css/**", "/js/**", "/images/**").permitAll()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
        )
        .formLogin(form -> form
            .loginPage("/login")
            .defaultSuccessUrl("/dashboard")
            .permitAll()
        )
        .logout(logout -> logout
            .logoutUrl("/logout")
            .logoutSuccessUrl("/login?logout")
            .permitAll()
        );
    return http.build();
}
}

```

Key Takeaways

1. **securityMatcher()**: Determines **which requests** a SecurityFilterChain processes - think of it as the "scope" of the chain [\[203\]](#) [\[202\]](#) [\[204\]](#)
2. **securityMatcher("/")****: Means this chain handles **ALL requests** - often used as catch-all with lower priority [\[202\]](#)
3. **antMatcher() is DEPRECATED**: Replaced by `securityMatcher()` in Spring Security 6 [\[207\]](#) [\[208\]](#) [\[206\]](#)
4. **requestMatchers()**: Modern way to define authorization rules, automatically chooses best matching strategy [\[206\]](#) [\[204\]](#)
5. **Migration Essential**: Update from old `antMatchers()/authorizeRequests()` to new `requestMatchers()/authorizeHttpRequests()` [\[208\]](#) [\[206\]](#)

These concepts are fundamental for building robust, modern Spring Security configurations. The new approach is more secure and provides better separation of concerns between chain-level and request-level matching!

**

Phase 4: Advanced Authentication

Building on your foundational knowledge, Phase 4 focuses on **stateless, token-based authentication** and **modern standards**—JSON Web Tokens (JWT), OAuth2, and OpenID Connect. These mechanisms power secure, scalable APIs and microservices.

1. JSON Web Tokens (JWT)

What Is JWT?

A **JSON Web Token** is a compact, URL-safe token that represents claims between two parties. It consists of three parts separated by dots:

1. **Header**: Specifies signing algorithm (e.g., HS256).
2. **Payload (Claims)**: Contains user data (subject, roles, expiration).
3. **Signature**: Verifies integrity using a secret or asymmetric key.

Example token:

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqb2huZG91Iiwicm9sZXMiOlsiVVNFUiJdLCJleHAiOjE2Mzc2ODAwMDBc
```

Why JWT?

- **Stateless**: No server-side session storage.
- **Scalable**: Ideal for microservices and distributed systems.
- **Interoperable**: Language-agnostic, works across domains.
- **Flexible**: Custom claims for roles, permissions, metadata.

Implementing JWT in Spring Security

1. Dependencies

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
</dependency>
```

2. JWT Utility Service

```
@Service
public class JwtService {
    private final SecretKey secretKey = Keys.hmacShaKeyFor(
        Base64.getDecoder().decode("your-256-bit-secret")
    );
    private final long expirationMs = 3600000; // 1 hour

    public String generateToken(UserDetails user) {
        Map<String, Object> claims = new HashMap<>();
        claims.put("roles", user.getAuthorities()
            .stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.toList()));
        return Jwts.builder()
            .setClaims(claims)
            .setSubject(user.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + expirationMs))
            .signWith(secretKey, SignatureAlgorithm.HS256)
            .compact();
    }

    public boolean validateToken(String token, UserDetails user) {
        String username = extractUsername(token);
        return (username.equals(user.getUsername()) && !isTokenExpired(token));
    }

    public String extractUsername(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(secretKey)
            .build()
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
    }

    private boolean isTokenExpired(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(secretKey)
            .build()
            .parseClaimsJws(token)
            .getBody()
            .getExpiration()
            .before(new Date());
    }
}
```

3. JWT Filter

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    @Autowired private JwtService jwtService;
    @Autowired private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain)
        throws ServletException, IOException {
        String header = req.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);
            String username = jwtService.extractUsername(token);
            if (username != null &&
                SecurityContextHolder.getContext().getAuthentication() == null) {
                UserDetails user = userDetailsService.loadUserByUsername(username);
                if (jwtService.validateToken(token, user)) {
                    UsernamePasswordAuthenticationToken authToken =
                        new UsernamePasswordAuthenticationToken(
                            user, null, user.getAuthorities());
                    SecurityContextHolder.getContext().setAuthentication(authToken);
                }
            }
        }
        chain.doFilter(req, res);
    }
}
```

4. Security Configuration

```
@Configuration
@EnableWebSecurity
public class JwtSecurityConfig {
    @Autowired private JwtAuthenticationFilter jwtFilter;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .sessionManagement(sess ->
                sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
            )
            .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

5. Authentication Controller

```
@RestController
@RequestMapping("/auth")
public class AuthController {
    @Autowired private AuthenticationManager authManager;
    @Autowired private JwtService jwtService;

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody AuthRequest req) {
        authManager.authenticate(
            new UsernamePasswordAuthenticationToken(req.getUsername(), req.getPassword()));
        UserDetails user =
            userDetailsService.loadUserByUsername(req.getUsername());
        String token = jwtService.generateToken(user);
        return ResponseEntity.ok(new AuthResponse(token));
    }
}
```

2. OAuth2 and OpenID Connect

Why OAuth2 and OpenID Connect?

- **OAuth2:** Protocol for secure delegated access (authorizing third-party applications).
- **OpenID Connect (OIDC):** Identity layer on top of OAuth2 for authentication.

Core Concepts

- **Resource Owner:** User.
- **Client:** Application requesting access.
- **Authorization Server:** Issues tokens.
- **Resource Server:** Hosts protected resources.

Spring Security OAuth2

1. Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

2. OAuth2 Resource Server Configuration

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: https://issuer-domain/.well-known/jwks.json
```

```
@Configuration
@EnableWebSecurity
public class OAuth2ResourceServerConfig {
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests(auth -> auth
        .requestMatchers("/public/**").permitAll()
        .anyRequest().authenticated()
      )
      .oauth2ResourceServer(oauth2 -> oauth2
        .jwt(withDefaults())
      );
    return http.build();
  }
}
```

3. OAuth2 Client (Login with Google/GitHub)

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: your-google-client-id
            client-secret: your-google-client-secret
            scope: openid, profile, email
```

```
@Configuration
@EnableWebSecurity
public class OAuth2ClientConfig {
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests(auth -> auth
        .anyRequest().authenticated()
      )
      .oauth2Login(withDefaults());
    return http.build();
  }
}
```

```
}
```

3. Hands-on Projects

1. JWT-Secured API

- Implement token issuance and validation.
- Protect CRUD endpoints using claims-based authorization.

2. OAuth2 Resource Server

- Configure Spring Boot as a resource server.
- Use a public JWK set from a real authorization server.

3. Social Login Integration

- Enable "Login with Google" and "Login with GitHub" in a web app.
- Map OAuth2 user attributes to application roles.

Key Takeaways

- **JWT:** Stateless, scalable, customizable token-based authentication.
- **OAuth2:** Standard protocol for delegated authorization; OIDC for authentication.
- **Spring Security Support:** First-class support via resource-server and client modules.
- **Best Practices:**
 - Always use HTTPS.
 - Short-lived access tokens + rotating refresh tokens.
 - Validate all token claims (issuer, audience, expiration).
 - Scope tokens to minimal necessary permissions.

Phase 4 equips you with industry-standard authentication mechanisms, paving the way for secure microservices and modern web applications. Next, we will cover **Authorization and Access Control** in Phase 5.

Phase 5: Role-Based and Method-Level Authorization in Spring Security

This phase covers how to control access to resources using roles/permissions at both the URL and method level, using the latest features and annotations in Spring Security.

Role-Based Access Control (RBAC) Overview

- RBAC assigns permissions to roles, and users to those roles.^[8_1]^[8_2]
- Each endpoint or resource is protected based on user role, minimizing “who can do what?” confusion.
- Permissions can also be modeled as “authorities,” often for more granular control.^[8_3]^[8_2]

RBAC Implementation Steps

1. **Define Roles:** e.g. USER, ADMIN, MANAGER, CUSTOMER^[8_2]^[8_1]
2. **Configure Security Rules:** Map URL patterns and business actions to required roles or permissions^[8_1]^[8_2]
3. **Assign Users to Roles:** Via your database/user service, assign one or more roles per user^[8_2]^[8_1]
4. **Enforce Authorization:** Use annotations or configuration to enforce that only users with required roles/authorities can access sensitive resources^[8_1]^[8_2]

Example: URL-Based Role Access

```
http
    .authorizeHttpRequests(authorize -> authorize
        .requestMatchers("/admin/**").hasRole("ADMIN")
        .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
        .requestMatchers("/public/**").permitAll()
        .anyRequest().authenticated()
    );
}
```

Method-Level Security

With method-level security, you protect individual service or controller methods using annotations. This gives **fine-grained, business-logic-level access control**—even if someone bypasses the web layer, your backend remains protected.^[8_4]^[8_5]^[^8_6]

Enabling Method Security

Add `@EnableMethodSecurity` to your security configuration:

```
@Configuration
@EnableMethodSecurity      // Spring Security 6+ (replaces @EnableGlobalMethodSecurity)
public class SecurityConfig { ... }
```

Key Annotations

Annotation	Where from?	Role String	Supports SpEL?
@Secured	Spring Security	"ROLE_ADMIN"	No
@RolesAllowed	javax.annotation.security	"ADMIN" (no ROLE_ needed)	No
@PreAuthorize	Spring Security	Any SpEL expression	Yes (powerful)
@PostAuthorize	Spring Security	SpEL expression (after call)	Yes

8_7][8_5][^8_6

Usage Examples

1. Basic Role Restrictions

```
@Service
public class AccountService {

    @Secured("ROLE_ADMIN")
    public void deleteAllAccounts() { ... }

    @RolesAllowed("USER")
    public Account myAccount() { ... }
}
```

2. Advanced Restrictions with SpEL

- **Complex combinations** are supported with @PreAuthorize:

```
@PreAuthorize("hasRole('ADMIN') or hasAuthority('MANAGE_ACCOUNTS')")
public void resetPassword(Long userId) { ... }

@PreAuthorize("hasRole('USER') and #contact.name == authentication.name")
public void updateProfile(Contact contact) { ... }
```

- **Post-authorization checks** for filtering results after method execution:

```
@PostAuthorize("returnObject.owner == authentication.name")
public Order viewOrder(Long id) { ... }
```

SpEL (Spring Expression Language) in Security

- Spring exposes authentication, principal, #parameter, and other root objects within expressions:[^8_5]
 - hasRole('ADMIN'), hasAuthority('WRITE')
 - authentication.name == #username

- o `#id == principal.id`

Example Policy

- Only allow users to access their own details or admins to access any user:

```
@PreAuthorize("hasRole('ADMIN') or #id == authentication.principal.id")
public User getUserById(Long id) { ... }

<div align="center">*</div>

[^8_1]: https://www.geeksforgeeks.org/advance-java/example-of-rbac-in-spring-security/
[^8_2]: https://www.geeksforgeeks.org/java/spring-security-role-based-and-permission-base
[^8_3]: https://www.baeldung.com/role-and-privilege-for-spring-security-registration
[^8_4]: https://www.geeksforgeeks.org/advance-java/spring-security-at-method-level/
[^8_5]: https://docs.spring.io/spring-security/reference/servlet/authorization/method-se
[^8_6]: https://www.baeldung.com/spring-security-method-security
[^8_7]: https://www.marcobehler.com/guides/spring-security
[^8_8]: https://www.geeksforgeeks.org/advance-java/spring-security-annotations/
[^8_9]: https://blog.tericcabrel.com/role-base-access-control-spring-boot/
[^8_10]: https://www.youtube.com/watch?v=nN68jjUP_rQ
[^8_11]: https://dev.to/alphaaman/building-a-role-based-access-control-system-with-jwt-ir
[^8_12]: https://docs.spring.io/spring-security/reference/servlet/authorization/authorize
[^8_13]: https://docs.spring.io/spring-security/reference/servlet/authorization/architect
[^8_14]: https://javarevisited.blogspot.com/2024/07/difference-between-secured-vs-rolesal
[^8_15]: https://stackoverflow.com/questions/79309925/method-level-security-using-spel
[^8_16]: https://www.edstem.com/blog/blog/implementing-rbac-springboot/
[^8_17]: https://developer.okta.com/blog/2019/06/20/spring-preauthorize
[^8_18]: https://javatechonline.com/spring-security-annotations/
[^8_19]: https://stackoverflow.com/questions/43961625/rolesallowed-vs-preauthorize-vs-sec

---
```

Phase 6: Microservices Security

In a microservices architecture, each service must enforce security while allowing seamless communication between them.

1. API Gateway Security

An **API Gateway** centralizes cross-cutting concerns including authentication, authorization, and monitoring.

Benefits

- Single entry point for all clients
- Offloads security logic from individual services
- Simplifies token validation and request routing

Implementation with Spring Cloud Gateway

```
```yaml
spring:
 cloud:
 gateway:
 routes:
 - id: auth_route
```

```

uri: lb://AUTH-SERVICE
predicates:
- Path=/auth/**

- id: user_route
uri: lb://USER-SERVICE
predicates:
- Path=/api/users/**
```

```

@Configuration
public class GatewaySecurityConfig {

 @Bean
 public SecurityWebFilterChain filterChain(ServerHttpSecurity http) {
 http
 .csrf(csrf -> csrf.disable())
 .authorizeExchange(ex -> ex
 .pathMatchers("/auth/**").permitAll()
 .pathMatchers("/api/**").authenticated()
)
 .oauth2ResourceServer(oauth2 -> oauth2.jwt());
 return http.build();
 }
}
```

- **JWT Validation** at gateway ensures downstream services trust forwarded requests.

## 2. Token Relay and Propagation

Services often need to call other services on behalf of the original user. Use **token relay** to propagate JWT or OAuth2 tokens.

### Spring Cloud OpenFeign Example

```

@FeignClient(name = "order-service")
public interface OrderClient {
 @GetMapping("/orders/{id}")
 Order getOrder(@PathVariable Long id);
}
```

```

@Configuration
public class FeignConfig {
 @Bean
 public RequestInterceptor oauth2FeignRequestInterceptor(ReactiveOAuth2AuthorizedClientManager manager) {
 return requestTemplate -> {
 OAuth2AuthorizeRequest authRequest = OAuth2AuthorizeRequest.withClientRegistrationId("user")
 .principal("gateway")
 .build();
 OAuth2AuthorizedClient client = manager.authorize(authRequest);
 String token = client.getAccessToken().getTokenValue();
 requestTemplate.header("Authorization", "Bearer " + token);
 };
 }
}
```

```
 };
}
}
```

- **Reuses** the gateway's client credentials to call downstream services.

### 3. Securing Inter-Service Communication

#### OAuth2 Client Credentials Flow

For service-to-service auth without user context, use **client credentials** grant.

```
spring:
 security:
 oauth2:
 client:
 registration:
 service-a:
 client-id: service-a-id
 client-secret: secretA
 authorization-grant-type: client_credentials
 scope: read write
 provider:
 keycloak:
 token-uri: https://auth-server/auth/realms/demo/protocol/openid-connect/toker
```

```
@Bean
public SecurityFilterChain serviceChain(HttpSecurity http) throws Exception {
 http
 .authorizeHttpRequests(auth -> auth.anyRequest().hasAuthority("SCOPE_read"))
 .oauth2ResourceServer(oauth2 -> oauth2.jwt());
 return http.build();
}
```

- **Client A** obtains token and calls **Service B** with Authorization: Bearer ...
- **Service B** validates token scopes to authorize access.

### 4. Centralized Authorization Server

Implement a dedicated **Authorization Server** to issue and manage tokens, enforce policies, and handle user consent.

## Spring Authorization Server (v1.0)

```
@Configuration
public class AuthorizationServerConfig {

 @Bean
 public RegisteredClientRepository registeredClientRepository() {
 RegisteredClient client = RegisteredClient.withId(UUID.randomUUID().toString())
 .clientId("gateway-client")
 .clientSecret("{noop}secret")
 .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
 .scope("read")
 .build();
 return new InMemoryRegisteredClientRepository(client);
 }

 @Bean
 public AuthorizationServerSettings serverSettings() {
 return AuthorizationServerSettings.builder().issuer("https://auth-server").build();
 }

 @Bean
 public SecurityFilterChain authServerSecurity(HttpSecurity http) throws Exception {
 OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
 return http.build();
 }
}
```

- Issues **JWT** or **opaque tokens** for clients and users
- Supports **authorization code**, **refresh tokens**, **client credentials**, and **PKCE**

## 5. Fine-Grained Access Control with OAuth2 Scopes and Claims

### Define Scopes and Map to Authorities

```
@Bean
public JwtEncoder jwtEncoder() { ... }

@Bean
public OAuth2TokenCustomizer<JwtEncodingContext> jwtCustomizer() {
 return context -> {
 if (context.getTokenType().equals(OAuth2TokenType.ACCESS_TOKEN)) {
 Authentication auth = context.getPrincipal();
 Set<String> scopes = auth.getAuthorities().stream()
 .map(GrantedAuthority::getAuthority)
 .collect(Collectors.toSet());
 context.getClaims().claim("scope", scopes);
 }
 };
}
```

- Downstream services check scope claim for required permissions:

```
@PreAuthorize("hasAuthority('read')")
public Mono<Product> getProduct(String id) { ... }
```

## 6. Integrating Keycloak as Identity Provider

Keycloak offers a full-featured IAM solution with user federation, social login, and realm management.

### Spring Boot Resource Server Example

```
spring:
 security:
 oauth2:
 resourceserver:
 jwt:
 jwk-set-uri: https://keycloak.demo/auth/realms/myrealm/protocol/openid-connect,
```

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
 http
 .authorizeHttpRequests(auth -> auth
 .requestMatchers("/public/**").permitAll()
 .anyRequest().authenticated()
)
 .oauth2ResourceServer(oauth2 -> oauth2.jwt());
 return http.build();
}
```

- Leverages Keycloak's user management, role mappers, and policy enforcement

## Key Takeaways

1. **API Gateway** centralizes request security and token validation.
2. **Token Relay** propagates tokens across services for user context.
3. **Client Credentials** grant secures service-to-service calls without user involvement.
4. **Authorization Server** centralizes token issuance, policies, and scopes.
5. **Scopes and Claims** enable fine-grained authorization.
6. **Identity Providers** like Keycloak simplify enterprise-grade security.

Phase 6 equips you with best practices to secure microservices end-to-end, from gateway to individual services. The next phase will focus on production hardening and security best practices at scale.

# Phase 7: Security Best Practices and Hardening

Phase 7 focuses on **production-grade security** measures, ensuring your Spring applications are resilient against advanced attacks and operate safely at scale.

## 1. Enforce HTTPS/TLS

### Why HTTPS?

- **Encrypts data in transit**, preventing eavesdropping and man-in-the-middle attacks.
- **Secure cookies** (e.g., session, CSRF tokens) require Secure flag over HTTPS.

### Implementation Steps

1. Obtain a certificate from a trusted CA (Let's Encrypt, commercial CA).

2. Configure Spring Boot:

```
server:
 ssl:
 key-store: classpath:keystore.p12
 key-store-password: changeit
 key-store-type: PKCS12
 key-alias: tomcat
 port: 8443
```

3. Redirect HTTP to HTTPS:

```
@Configuration
public class HttpsRedirectConfig {
 @Bean
 public WebServerFactoryCustomizer<TomcatServletWebServerFactory> containerCustomizer() {
 return server -> server.addAdditionalTomcatConnectors(httpConnector());
 }
 private Connector httpConnector() {
 Connector connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
 connector.setScheme("http");
 connector.setPort(8080);
 connector.setSecure(false);
 connector.setRedirectPort(8443);
 return connector;
 }
}
```

## 2. CSRF Protection

### Why CSRF?

Prevents malicious sites from performing actions on behalf of authenticated users.

### Default Spring Security CSRF

- **Enabled by default** for web applications.
- **Requires** adding CSRF token to state-changing requests (POST, PUT, DELETE).

### Handling CSRF in APIs

- **Disable** CSRF for stateless REST APIs:

```
http
 .csrf(csrf -> csrf.disable())
 .sessionManagement(sess -> sess.sessionCreationPolicy(STATELESS));
```

- **Use** other protections (CORS, token-based auth).

## 3. CORS Configuration

### Why CORS?

Controls which origins may access resources, preventing unauthorized cross-origin requests.

### Spring Boot CORS Setup

```
@Configuration
public class CorsConfig {
 @Bean
 public WebMvcConfigurer corsConfigurer() {
 return new WebMvcConfigurer() {
 @Override
 public void addCorsMappings(CorsRegistry registry) {
 registry.addMapping("/api/**")
 .allowedOrigins("https://trusted-client.com")
 .allowedMethods("GET", "POST", "PUT", "DELETE")
 .allowCredentials(true)
 .maxAge(3600);
 }
 };
 }
}
```

## 4. Secure Headers

Set HTTP response headers to mitigate common vulnerabilities.

```
http
 .headers(headers -> headers
 .contentSecurityPolicy("default-src 'self'")
 .frameOptions(frame -> frame.sameOrigin())
 .httpStrictTransportSecurity(hsts -> hsts
 .maxAgeInSeconds(31536000)
 .includeSubDomains(true)
)
 .xssProtection(xss -> xss.block(true))
 .cacheControl(cache -> cache.disable())
);
}
```

- **CSP**: Prevents XSS by restricting resource loading.
- **HSTS**: Forces HTTPS, prevents protocol downgrade.
- **X-Frame-Options**: Defends clickjacking.
- **X-XSS-Protection**: Enables browser XSS filters.
- **Cache-Control**: Protects sensitive data from being cached.

## 5. Input Validation and Output Encoding

1. **Validate all user inputs** (use Bean Validation, custom validators).
2. **Encode outputs** (HTML, JavaScript, URLs) to prevent XSS.
3. Use libraries such as **OWASP Java Encoder** for consistent encoding.

## 6. Rate Limiting and Throttling

Prevent brute-force and denial-of-service attacks.

### Using Bucket4j with Spring Boot

```
@Configuration
public class RateLimitingConfig {
 @Bean
 public Bucket tokenBucket() {
 Refill refill = Refill.greedy(10, Duration.ofMinutes(1));
 Bandwidth limit = Bandwidth.classic(10, refill);
 return Bucket4j.builder().addLimit(limit).build();
 }
}

@Component
public class RateLimitFilter extends OncePerRequestFilter {
 @Autowired private Bucket bucket;
 @Override
```

```

protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain)
 throws ServletException, IOException {
 if (bucket.tryConsume(1)) {
 chain.doFilter(req, res);
 } else {
 res.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
 res.getWriter().write("Too many requests - try again later");
 }
}

```

## 7. Session Management

- **Stateless** for APIs (JWT, OAuth2).
- **Stateful** for web apps with session fixation and concurrency control:

```

http
 .sessionManagement(sess -> sess
 .sessionFixation().migrateSession()
 .maximumSessions(1)
 .expiredUrl("/login?expired"));

```

## 8. Logging, Monitoring, and Auditing

- **Log security events** (login success/failure, access denials).
- Use **Spring Security events** and listeners:

```

@Component
public class AuthEventListener {
 @EventListener
 public void onAuthSuccess(AuthenticationSuccessEvent event) {
 log.info("User logged in: {}", event.getAuthentication().getName());
 }
 @EventListener
 public void onAuthFailure(AbstractAuthenticationFailureEvent event) {
 log.warn("Authentication failed: {}", event.getAuthentication().getName());
 }
}

```

- **Audit** critical actions with database/audit logs.
- **Monitor** with tools like ELK, Prometheus, Grafana.

## 9. Dependency Management and Vulnerability Scanning

1. **Keep dependencies up to date** (use Dependabot, Renovate).
2. Use **OWASP Dependency-Check** or **Snyk** to scan for known vulnerabilities.
3. **Regularly review** Spring Security release notes and upgrade guidelines.

## 10. Penetration Testing and Security Audits

- Conduct **regular penetration tests** against APIs and web apps.
- Use **tools** like OWASP ZAP, Burp Suite for dynamic analysis.
- **Remediate** identified issues promptly and update configurations accordingly.

### Key Takeaways

1. **Enforce HTTPS** with valid certificates and HSTS.
2. **Enable CSRF** protection for web apps, disable for stateless APIs.
3. **Configure CORS** restrictively.
4. **Set secure headers** (CSP, HSTS, X-Frame-Options).
5. **Validate inputs** and encode outputs.
6. **Implement rate limiting** to prevent abuse.
7. **Manage sessions** securely.
8. **Log and audit** security events.
9. **Scan dependencies** and keep libraries current.
10. **Perform penetration testing** regularly.

Phase 7 hardens your applications for production, reducing attack surface and ensuring robust security at scale. Next, we will focus on **Security Testing and Monitoring** in Phase 8.

## Phase 8: Security Testing and Monitoring

Phase 8 covers comprehensive testing strategies and monitoring approaches to ensure your Spring Security implementation is robust, secure, and observable in production environments.

### 1. Security Testing with MockMvc

#### Setting Up Spring Security Test

Add the Spring Security Test dependency:[\[222\]](#) [\[223\]](#)

```
<dependency>
<groupId>org.springframework.security</groupId>
```

spring-security-test

```
<scope>test</scope>
</dependency>
```

## Testing Authorization with Mock Users

The most common approach is testing authorization rules without going through authentication:  
[\[224\]](#) [\[223\]](#) [\[222\]](#)

```
@SpringBootTest
@AutoConfigureMockMvc
public class SecurityIntegrationTests {

 @Autowired
 private MockMvc mockMvc;

 // Test unauthenticated access
 @Test
 public void accessPublicEndpoint_ShouldSucceed() throws Exception {
 mockMvc.perform(get("/public/info"))
 .andExpect(status().isOk())
 .andExpect(content().string("Public information"));
 }

 // Test authenticated access with @WithMockUser
 @Test
 @WithMockUser(username = "user", roles = "USER")
 public void accessUserEndpoint_WithUserRole_ShouldSucceed() throws Exception {
 mockMvc.perform(get("/user/profile"))
 .andExpect(status().isOk())
 .andExpect(content().string(containsString("user")));
 }

 // Test authorization failure
 @Test
 @WithMockUser(username = "user", roles = "USER")
 public void accessAdminEndpoint_WithUserRole_ShouldFail() throws Exception {
 mockMvc.perform(get("/admin/users"))
 .andExpect(status().isForbidden());
 }

 // Test with specific authorities
 @Test
 @WithMockUser(username = "manager", authorities = {"READ", "WRITE"})
 public void accessWithAuthorities_ShouldSucceed() throws Exception {
 mockMvc.perform(get("/api/data"))
 .andExpect(status().isOk());
 }
}
```

## Advanced Mock User Configuration

Customize mock user attributes: [\[223\]](#) [\[222\]](#)

```
@Test
@WithMockUser(
 username = "admin",
 password = "password",
```

```

 roles = {"ADMIN", "USER"},
 authorities = {"READ", "WRITE", "DELETE"}
)
public void complexAuthorizationTest() throws Exception {
 mockMvc.perform(delete("/api/users/123"))
 .andExpect(status().isNoContent());
}

```

## Testing with Real UserDetailsService

Use `@WithUserDetails` to test with actual users from your `UserDetailsService`: [\[222\]](#) [\[223\]](#)

```

@SpringBootTest
@AutoConfigureMockMvc
public class UserDetailsIntegrationTests {

 @Test
 @WithUserDetails("john.doe@example.com")
 public void testWithRealUser() throws Exception {
 // This uses your actual UserDetailsService to load user "john.doe@example.com"
 mockMvc.perform(get("/user/dashboard"))
 .andExpect(status().isOk())
 .andExpect(model().attributeExists("user"));
 }
}

```

## Custom Security Test Annotations

Create reusable test security contexts: [\[225\]](#) [\[226\]](#) [\[222\]](#)

```

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = WithMockAdminSecurityContextFactory.class)
public @interface WithMockAdmin {
 String username() default "admin";
 String name() default "Admin User";
}

public class WithMockAdminSecurityContextFactory
 implements WithSecurityContextFactory<WithMockAdmin> {

 @Override
 public SecurityContext createSecurityContext(WithMockAdmin annotation) {
 SecurityContext context = SecurityContextHolder.createEmptyContext();

 CustomUserDetails principal = new CustomUserDetails(
 annotation.username(),
 annotation.name(),
 List.of(new SimpleGrantedAuthority("ROLE_ADMIN")));
 }

 Authentication auth = new UsernamePasswordAuthenticationToken(
 principal, "password", principal.getAuthorities());
}

```

```

 context.setAuthentication(auth);
 return context;
 }
}

```

Usage:

```

@Test
@WithMockAdmin(username = "super-admin", name = "Super Administrator")
public void testAdminFunctionality() throws Exception {
 mockMvc.perform(get("/admin/system-config"))
 .andExpect(status().isOk());
}

```

## Testing Method Security

Test @PreAuthorize, @PostAuthorize, and other method-level security:[\[227\]](#) [\[228\]](#) [\[222\]](#)

```

@SpringBootTest
public class MethodSecurityTests {

 @Autowired
 private UserService userService;

 @Test
 @WithMockUser(roles = "ADMIN")
 public void deleteUser_AsAdmin_ShouldSucceed() {
 assertDoesNotThrow(() -> userService.deleteUser(123L));
 }

 @Test
 @WithMockUser(roles = "USER")
 public void deleteUser_AsUser_ShouldThrowAccessDenied() {
 assertThrows(AccessDeniedException.class,
 () -> userService.deleteUser(123L));
 }

 @Test
 @WithMockUser(username = "john")
 public void getUserProfile_OwnProfile_ShouldSucceed() {
 // Tests @PreAuthorize("#username == authentication.name")
 User user = userService.getUserProfile("john");
 assertThat(user.getUsername()).isEqualTo("john");
 }

 @Test
 @WithMockUser(username = "john")
 public void getUserProfile_OtherProfile_ShouldThrowAccessDenied() {
 assertThrows(AccessDeniedException.class,
 () -> userService.getUserProfile("jane"));
 }
}

```

```
 }
}
```

## 2. Integration Testing Strategies

### Testing Authentication Flow

Test the complete authentication process:[\[229\]](#) [\[224\]](#)

```
@SpringBootTest
@AutoConfigureMockMvc
public class AuthenticationIntegrationTests {

 @Test
 public void loginWithValidCredentials_ShouldSucceed() throws Exception {
 mockMvc.perform(post("/login")
 .param("username", "user")
 .param("password", "password")
 .with(csrf()))
 .andExpect(status().is3xxRedirection())
 .andExpected(redirectedUrl("/dashboard"));
 }

 @Test
 public void loginWithInvalidCredentials_ShouldFail() throws Exception {
 mockMvc.perform(post("/login")
 .param("username", "user")
 .param("password", "wrong")
 .with(csrf()))
 .andExpect(status().is3xxRedirection())
 .andExpected(redirectedUrl("/login?error"));
 }

 @Test
 public void accessProtectedResource_WithoutLogin_ShouldRedirectToLogin() throws Exception {
 mockMvc.perform(get("/user/profile"))
 .andExpect(status().is3xxRedirection())
 .andExpected(redirectedUrl("http://localhost/login"));
 }
}
```

### Testing JWT Authentication

Test JWT token generation and validation:[\[229\]](#)

```
@SpringBootTest
@AutoConfigureMockMvc
public class JwtAuthenticationTests {

 @Autowired
 private JwtService jwtService;
```

```

 @Test
 public void generateToken_ValidUser_ShouldReturnValidToken() {
 UserDetails user = User.builder()
 .username("testuser")
 .password("password")
 .roles("USER")
 .build();

 String token = jwtService.generateToken(user);

 assertThat(token).isNotNull();
 assertThat(jwtService.extractUsername(token)).isEqualTo("testuser");
 assertThat(jwtService.validateToken(token, user)).isTrue();
 }

 @Test
 public void accessApiWithValidJwt_ShouldSucceed() throws Exception {
 String token = jwtService.generateToken(createTestUser());

 mockMvc.perform(get("/api/data")
 .header("Authorization", "Bearer " + token))
 .andExpect(status().isOk());
 }

 @Test
 public void accessApiWithInvalidJwt_ShouldFail() throws Exception {
 mockMvc.perform(get("/api/data")
 .header("Authorization", "Bearer invalid.jwt.token"))
 .andExpect(status().isUnauthorized());
 }
}

```

## Disabling Security for Specific Tests

Sometimes you need to test business logic without security interference: [\[230\]](#)

```

// Option 1: Exclude SecurityAutoConfiguration
@SpringBootTest(exclude = SecurityAutoConfiguration.class)
@AutoConfigureMockMvc(addFilters = false)
public class BusinessLogicTests {
 // Tests run without security filters
}

// Option 2: Use test profile with permissive security
@SpringBootTest
@ActiveProfiles("test")
public class TestWithCustomSecurity {
 // application-test.yml disables security or uses permissive config
}

```

### 3. Security Monitoring and Auditing

#### Enabling Audit Events

Spring Boot Actuator provides automatic security event auditing:[\[231\]](#) [\[232\]](#) [\[233\]](#)

```
@Configuration
public class AuditConfig {

 @Bean
 public AuditEventRepository auditEventRepository() {
 return new InMemoryAuditEventRepository();
 }

 // For production, use persistent repository
 @Bean
 @Profile("production")
 public AuditEventRepository persistentAuditEventRepository() {
 return new JdbcAuditEventRepository(dataSource);
 }
}
```

#### Custom Audit Event Listeners

Listen to and process security events:[\[234\]](#) [\[235\]](#) [\[231\]](#)

```
@Component
public class SecurityAuditEventListener {

 private static final Logger logger = LoggerFactory.getLogger(SecurityAuditEventListener.class);

 @EventListener
 public void onAuthenticationSuccess(AuthenticationSuccessEvent event) {
 String username = event.getAuthentication().getName();
 String details = event.getAuthentication().getDetails().toString();

 logger.info("Successful login: user={}, details={}", username, details);

 // Send to monitoring system
 sendMetric("auth.success", username);
 }

 @EventListener
 public void onAuthenticationFailure(AbstractAuthenticationFailureEvent event) {
 String username = event.getAuthentication().getName();
 String reason = event.getException().getMessage();

 logger.warn("Failed login attempt: user={}, reason={}", username, reason);

 // Track failed attempts for potential lockout
 incrementFailedAttempts(username);
 sendMetric("auth.failure", username);
 }
}
```

```

@EventListener
public void onAuthorizationFailure(AuthorizationDeniedEvent event) {
 logger.warn("Access denied: user={}, resource={}",
 event.getAuthentication().getName(),
 event.getAuthorizationDecision());

 sendMetric("auth.access_denied",
 event.getAuthentication().getName());
}

private void sendMetric(String metricName, String username) {
 // Integration with monitoring system (Micrometer, etc.)
}

private void incrementFailedAttempts(String username) {
 // Track failed attempts in cache/database
}
}

```

## Publishing Custom Audit Events

Create custom business audit events: [\[235\]](#) [\[231\]](#)

```

@Service
public class UserManagementService {

 @Autowired
 private ApplicationEventPublisher eventPublisher;

 @Autowired
 private AuditEventRepository auditEventRepository;

 public void deleteUser(Long userId) {
 User user = userRepository.findById(userId)
 .orElseThrow(() -> new UserNotFoundException("User not found"));

 userRepository.delete(user);

 // Publish custom audit event
 AuditEvent auditEvent = new AuditEvent(
 getCurrentUsername(),
 "USER_DELETED",
 Map.of("userId", userId, "username", user.getUsername())
);

 eventPublisher.publishEvent(new AuditApplicationEvent(auditEvent));

 // Alternative: Direct repository save
 auditEventRepository.add(auditEvent);
 }

 private String getCurrentUsername() {
 return SecurityContextHolder.getContext()
 .getAuthentication()

```

```
 .getName();
 }
}
```

## 4. Metrics and Observability

### Security Metrics with Micrometer

Integrate security metrics with monitoring systems: [\[236\]](#) [\[237\]](#) [\[238\]](#)

```
@Component
public class SecurityMetricsCollector {

 private final MeterRegistry meterRegistry;
 private final Counter loginSuccessCounter;
 private final Counter loginFailureCounter;
 private final Timer authenticationTimer;

 public SecurityMetricsCollector(MeterRegistry meterRegistry) {
 this.meterRegistry = meterRegistry;
 this.loginSuccessCounter = Counter.builder("security.login.success")
 .description("Successful login attempts")
 .register(meterRegistry);
 this.loginFailureCounter = Counter.builder("security.login.failure")
 .description("Failed login attempts")
 .register(meterRegistry);
 this.authenticationTimer = Timer.builder("security.authentication.duration")
 .description("Authentication processing time")
 .register(meterRegistry);
 }

 @EventListener
 public void onAuthenticationSuccess(AuthenticationSuccessEvent event) {
 loginSuccessCounter.increment(
 Tags.of("username", event.getAuthentication().getName())
);
 }

 @EventListener
 public void onAuthenticationFailure(AbstractAuthenticationFailureEvent event) {
 loginFailureCounter.increment(
 Tags.of(
 "username", event.getAuthentication().getName(),
 "reason", event.getException().getClass().getSimpleName()
)
);
 }

 public Timer.Sample startAuthenticationTimer() {
 return Timer.start(meterRegistry);
 }
}
```

## Actuator Security Endpoints

Expose security-related metrics via Actuator:[\[239\]](#) [\[240\]](#) [\[241\]](#)

```
management:
 endpoints:
 web:
 exposure:
 include: health,metrics,auditevents,httptrace
 endpoint:
 auditevents:
 enabled: true
 httptrace:
 enabled: true
 security:
 enabled: true
```

Access audit events:

```
curl http://localhost:8080/actuator/auditevents
curl "http://localhost:8080/actuator/auditevents?principal=admin&type=AUTHENTICATION_SUCCESS"
```

## Custom Health Indicators

Monitor security-related health status:[\[241\]](#) [\[236\]](#)

```
@Component
public class SecurityHealthIndicator implements HealthIndicator {

 @Autowired
 private AuthenticationFailureTracker failureTracker;

 @Override
 public Health health() {
 int failedAttempts = failureTracker.getRecentFailedAttempts();

 if (failedAttempts > 100) {
 return Health.down()
 .withDetail("failed_attempts", failedAttempts)
 .withDetail("status", "High number of failed authentication attempts")
 .build();
 }

 return Health.up()
 .withDetail("failed_attempts", failedAttempts)
 .withDetail("status", "Security systems operational")
 .build();
 }
}
```

## 5. Automated Security Testing

### Security Test Suite

Create comprehensive security test suites:[\[242\]](#) [\[223\]](#)

```
@SpringBootTest
@AutoConfigureMockMvc
@TestMethodOrder(OrderAnnotation.class)
public class SecurityTestSuite {

 @Autowired
 private MockMvc mockMvc;

 @Test
 @Order(1)
 public void testAuthenticationMechanisms() throws Exception {
 // Test basic auth, form login, JWT, OAuth2
 }

 @Test
 @Order(2)
 public void testAuthorizationRules() throws Exception {
 // Test role-based, permission-based access
 }

 @Test
 @Order(3)
 public void testSecurityHeaders() throws Exception {
 mockMvc.perform(get("/"))
 .andExpect(header().exists("X-Content-Type-Options"))
 .andExpected(header().exists("X-Frame-Options"))
 .andExpected(header().exists("X-XSS-Protection"));
 }

 @Test
 @Order(4)
 public void testCSRFProtection() throws Exception {
 mockMvc.perform(post("/api/data"))
 .andExpected(status().isForbidden()); // Should require CSRF token
 }

 @Test
 @Order(5)
 public void testSessionManagement() throws Exception {
 // Test session fixation, concurrent sessions
 }
}
```

## Performance Testing Security

Test security performance impact:[\[224\]](#)

```
@Test
public void authenticateUser_PerformanceTest() {
 StopWatch stopWatch = new StopWatch();

 for (int i = 0; i < 1000; i++) {
 stopWatch.start();
 // Perform authentication
 authenticationManager.authenticate(
 new UsernamePasswordAuthenticationToken("user", "password")
);
 stopWatch.stop();
 }

 assertThat(stopWatch.getTotalTimeMillis()).isLessThan(5000);
 assertThat(stopWatch.getTotalTimeMillis() / 1000.0).isLessThan(5.0);
}
```

## Key Takeaways

1. **Mock Users for Testing:** Use `@WithMockUser` and `@WithUserDetails` to test authorization without authentication[\[223\]](#) [\[222\]](#)
2. **Custom Test Annotations:** Create reusable security contexts for complex testing scenarios[\[225\]](#) [\[222\]](#)
3. **Integration Testing:** Test complete authentication flows, JWT handling, and security configurations[\[224\]](#) [\[229\]](#)
4. **Audit Events:** Leverage Spring Boot Actuator for automatic security event auditing[\[232\]](#) [\[231\]](#)
5. **Custom Metrics:** Implement security-specific metrics with Micrometer for monitoring[\[237\]](#) [\[236\]](#)
6. **Health Indicators:** Monitor security health status through custom health indicators[\[241\]](#)
7. **Comprehensive Test Suites:** Create systematic security testing covering authentication, authorization, headers, and performance[\[242\]](#) [\[223\]](#)

Phase 8 establishes a robust testing and monitoring foundation for your Spring Security implementations. With comprehensive testing strategies and detailed observability, you can confidently deploy and maintain secure applications at scale.

The next phase will explore **Advanced Security Topics** including modern authentication mechanisms and enterprise security patterns.



# before proceeding further we missed to understand openid authentication flow

Excellent question! You're absolutely right - we need to understand **OpenID Connect (OIDC)** authentication flow before proceeding. OIDC is crucial for modern authentication and builds on OAuth2. Let me explain it comprehensively.

## OpenID Connect (OIDC) Authentication Flow

### What is OpenID Connect?

**OpenID Connect (OIDC)** is an **authentication protocol built on top of OAuth 2.0**. While OAuth 2.0 handles **authorization** (what you can access), OIDC adds **authentication** (who you are). [\[258\]](#) [\[259\]](#) [\[260\]](#)

### Key Differences: OAuth 2.0 vs OpenID Connect

Aspect	OAuth 2.0	OpenID Connect
<b>Purpose</b>	Authorization ("What can you access?")	Authentication + Authorization ("Who are you?" + "What can you access?")
<b>Tokens</b>	Access Token only	Access Token + <b>ID Token</b>
<b>User Info</b>	No standard user identity	Standardized user claims
<b>Scopes</b>	Custom scopes	openid <b>scope required</b> + profile, email, etc.

### The ID Token - Core of OIDC

The **ID Token** is what makes OIDC unique: [\[261\]](#) [\[262\]](#) [\[263\]](#)

- **JSON Web Token (JWT)** containing user identity information
- **Signed by the Identity Provider** for integrity verification
- **Contains standardized claims** about the authenticated user
- **Consumed by the client application** (not sent to APIs like access tokens)

Sample ID Token Structure: [\[262\]](#) [\[263\]](#)

```
{
 "iss": "https://auth.example.com", // Issuer
 "sub": "248289761001", // Subject (user ID)
 "aud": "my-client-id", // Audience (client app)
 "exp": 1311281970, // Expiration time
 "iat": 1311280970, // Issued at time
 "nonce": "n-0S6_WzA2Mj", // Replay attack prevention
 "auth_time": 1311280969, // Authentication time
 "name": "John Doe", // User's full name
 "given_name": "John", // First name
 "family_name": "Doe", // Last name
```

```
"email": "john.doe@example.com", // Email address
"email_verified": true, // Email verification status
"picture": "https://example.com/john.jpg" // Profile picture URL
}
```

## OpenID Connect Authorization Code Flow

The **Authorization Code Flow** is the most secure and commonly used OIDC flow. [\[264\]](#) [\[265\]](#) [\[266\]](#)

### Step-by-Step Flow

#### 1. User Initiates Login [\[266\]](#) [\[264\]](#)

```
User clicks "Login" in your application
↓
Application redirects to Authorization Server
```

#### 2. Authorization Request [\[267\]](#) [\[264\]](#)

```
GET /authorize?
 response_type=code
 &client_id=your-client-id
 &redirect_uri=https://yourapp.com/callback
 &scope=openid profile email
 &state=random-state-value
 &nonce=random-nonce-value
```

Host: authorization-server.com

#### Key Parameters: [\[264\]](#) [\[267\]](#)

- **response\_type=code:** Requests authorization code
- **scope=openid:** **Required for OIDC** (distinguishes from pure OAuth2)
- **scope=profile email:** Additional user info scopes
- **state:** CSRF protection
- **nonce:** Replay attack protection for ID token

#### 3. User Authentication [\[266\]](#) [\[264\]](#)

```
Authorization Server authenticates user
↓
User sees consent screen (if needed)
↓
User grants permission
```

#### 4. Authorization Response [\[264\]](#) [\[266\]](#)

```
HTTP/1.1 302 Found
Location: https://yourapp.com/callback?
 code=authorization-code-value
 &state=random-state-value
```

## 5. Token Exchange [\[266\]](#) [\[264\]](#)

Your application makes a backend request:

```
POST /token HTTP/1.1
Host: authorization-server.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic base64(client_id:client_secret)

grant_type=authorization_code
&code=authorization-code-value
&redirect_uri=https://yourapp.com/callback
&client_id=your-client-id
```

## 6. Token Response [\[264\]](#) [\[266\]](#)

```
{
 "access_token": "eyJhbGciOiJSUzI1Ni...",
 "token_type": "Bearer",
 "expires_in": 3600,
 "refresh_token": "tGzv3J0kF0XG5Qx2TlKWIA",
 "id_token": "eyJhbGciOiJSUzI1NiJ9..." // The OIDC ID Token!
}
```

## Authorization Code Flow with PKCE

**PKCE (Proof Key for Code Exchange)** enhances security, especially for public clients. [\[268\]](#) [\[269\]](#) [\[270\]](#)

### Why PKCE?

**Security Problems PKCE Solves:** [\[270\]](#) [\[268\]](#)

- **Authorization code interception** attacks
- **No client secret** required for public clients (SPAs, mobile apps)
- **Prevents malicious apps** from using intercepted authorization codes

## PKCE Flow Steps

### 1. Generate Code Verifier and Challenge [\[269\]](#) [\[268\]](#)

```
// Generate random code verifier (43-128 characters)
const codeVerifier = base64URLEncode(crypto.randomBytes(32));
```

```
// Create code challenge (SHA256 hash of verifier)
const codeChallenge = base64URLEncode(sha256(codeVerifier));
```

## 2. Authorization Request with PKCE [268] [269]

```
GET /authorize?
 response_type=code
 &client_id=your-client-id
 &redirect_uri=https://yourapp.com/callback
 &scope=openid profile email
 &state=random-state-value
 &code_challenge=base64url-encoded-challenge
 &code_challenge_method=S256
```

## 3. Token Exchange with Code Verifier [269] [268]

```
POST /token HTTP/1.1
Host: authorization-server.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=authorization-code-value
&redirect_uri=https://yourapp.com/callback
&client_id=your-client-id
&code_verifier=original-code-verifier
```

## 4. Server Validates PKCE [271] [270]

```
Server computes: SHA256(code_verifier)
Server compares with stored code_challenge
If match → issue tokens
If no match → reject request
```

# Spring Security OIDC Implementation

## 1. Dependencies

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 ...
 <artifactId>spring-boot-starter-oauth2-client</artifactId>
 ...
</dependency>
```

## 2. Application Configuration

```
spring:
 security:
 oauth2:
 client:
 registration:
 google:
 client-id: your-google-client-id
 client-secret: your-google-client-secret
 scope:
 - openid
 - profile
 - email
 redirect-uri: '{baseUrl}/login/oauth2/code/{registrationId}'
 provider:
 google:
 issuer-uri: https://accounts.google.com
 user-name-attribute: name
```

## 3. Security Configuration

```
@Configuration
@EnableWebSecurity
public class OidcSecurityConfig {

 @Bean
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
 http
 .authorizeHttpRequests(authorize -> authorize
 .requestMatchers("/", "/login").permitAll()
 .anyRequest().authenticated()
)
 .oauth2Login(oauth2Login -> oauth2Login
 .LoginPage("/login")
 .defaultSuccessUrl("/dashboard")
 .userInfoEndpoint(userInfo -> userInfo
 .oidcUserService(oidcUserService())
)
);
 return http.build();
 }

 @Bean
 public OAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {
 OidcUserService oidcUserService = new OidcUserService();

 // Customize user mapping
 oidcUserService.setOAuth2UserService(oauth2UserRequest -> {
 OAuth2User oauth2User = new DefaultOAuth2UserService()
 .loadUser(oauth2UserRequest);

 // Add custom authorities based on user claims
 Set<GrantedAuthority> authorities = new HashSet<>(oauth2User.getAuthorities())
 });
 }
}
```

```

 // Example: Add ROLE_USER to all OIDC users
 authorities.add(new SimpleGrantedAuthority("ROLE_USER"));

 // Example: Add ROLE_ADMIN based on email domain
 String email = oauth2User.getAttribute("email");
 if (email != null && email.endsWith("@yourcompany.com")) {
 authorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
 }

 return new DefaultOAuth2User(authorities, oauth2User.getAttributes(), "email");
 };

 return oidcUserService;
}
}

```

## 4. Controller Implementation

```

@RestController
public class UserController {

 // Access ID Token claims
 @GetMapping("/user")
 public Map<String, Object> getCurrentUser(@AuthenticationPrincipal OidcUser oidcUser)
 Map<String, Object> userInfo = new HashMap<>();

 // Standard OIDC claims from ID Token
 userInfo.put("sub", oidcUser.getSubject());
 userInfo.put("name", oidcUser.getFullName());
 userInfo.put("email", oidcUser.getEmail());
 userInfo.put("picture", oidcUser.getPicture());
 userInfo.put("email_verified", oidcUser.getEmailVerified());

 // All claims from ID Token
 OidcIdToken idToken = oidcUser.getIdToken();
 userInfo.put("id_token_claims", idToken.getClaims());

 // User authorities
 userInfo.put("authorities", oidcUser.getAuthorities());

 return userInfo;
 }

 // Access tokens for API calls
 @GetMapping("/api-call")
 public String callExternalApi(@RegisteredOAuth2AuthorizedClient OAuth2AuthorizedClient
 // Get access token
 String accessToken = authorizedClient.getAccessToken().getTokenValue();

 // Use access token to call external APIs
 HttpHeaders headers = new HttpHeaders();
 headers.setBearerAuth(accessToken);

 RestTemplate restTemplate = new RestTemplate();

```

```

 HttpEntity<String> entity = new HttpEntity<>(headers);

 return restTemplate.exchange(
 "https://api.example.com/data",
 HttpMethod.GET,
 entity,
 String.class
).getBody();
 }
}

```

## 5. Custom Login Page

```

<!DOCTYPE html>
<html>
<head>
 <title>Login</title>
</head>
<body>
 ...
 <h2>Login to Your App</h2>
 ...

 <button>Login with Google</button>

 <button>Login with GitHub</button>

</body>
</html>

```

## Advanced OIDC Configuration

### Custom Authorization Request

```

@Bean
public OAuth2AuthorizationRequestResolver authorizationRequestResolver(
 ClientRegistrationRepository clientRegistrationRepository) {

 DefaultOAuth2AuthorizationRequestResolver defaultResolver =
 new DefaultOAuth2AuthorizationRequestResolver(
 clientRegistrationRepository, "/oauth2/authorization");

 return request -> {
 OAuth2AuthorizationRequest authorizationRequest =
 defaultResolver.resolve(request);

 if (authorizationRequest != null) {
 // Add custom parameters
 Map<String, Object> additionalParameters = new HashMap<>(
 authorizationRequest.getAdditionalParameters());

```

```

 additionalParameters.put("prompt", "select_account");
 additionalParameters.put("max_age", "3600");

 authorizationRequest = OAuth2AuthorizationRequest
 .from(authorizationRequest)
 .additionalParameters(additionalParameters)
 .build();
 }

 return authorizationRequest;
};

}

```

## Custom Success Handler

```

@Component
public class OidcAuthenticationSuccessHandler implements AuthenticationSuccessHandler {

 @Override
 public void onAuthenticationSuccess(HttpServletRequest request,
 HttpServletResponse response,
 Authentication authentication) throws IOException {

 if (authentication.getPrincipal() instanceof OidcUser) {
 OidcUser oidcUser = (OidcUser) authentication.getPrincipal();

 // Log successful OIDC authentication
 log.info("OIDC user authenticated: {}", oidcUser.getEmail());

 // Store user in database if first login
 userService.createOrUpdateUser(oidcUser);

 // Custom redirect based on user role
 String redirectUrl = determineRedirectUrl(oidcUser);
 response.sendRedirect(redirectUrl);
 }
 }

 private String determineRedirectUrl(OidcUser oidcUser) {
 if (oidcUser.getAuthorities().stream()
 .anyMatch(auth -> auth.getAuthority().equals("ROLE_ADMIN"))) {
 return "/admin/dashboard";
 }
 return "/user/dashboard";
 }
}

```

## Key Takeaways

1. **OIDC = OAuth2 + Authentication:** OIDC adds identity verification to OAuth2's authorization capabilities [\[259\]](#) [\[258\]](#)
2. **ID Token is Key:** Contains standardized user claims and enables client applications to know who the user is [\[261\]](#) [\[262\]](#)
3. **Authorization Code Flow:** Most secure OIDC flow, suitable for web applications with backend servers [\[265\]](#) [\[264\]](#)
4. **PKCE Enhancement:** Adds security for public clients and prevents authorization code interception [\[270\]](#) [\[268\]](#)
5. **Spring Security Integration:** Provides excellent out-of-the-box OIDC support with minimal configuration [\[272\]](#) [\[273\]](#)
6. **Scopes Matter:** openid scope distinguishes OIDC from pure OAuth2, additional scopes provide more user information [\[267\]](#) [\[266\]](#)
7. **Token Validation:** Always validate ID tokens for integrity and authenticity [\[274\]](#) [\[275\]](#)

OpenID Connect provides a standardized, secure way to handle user authentication while leveraging OAuth2's authorization capabilities. With Spring Security's OIDC support, you can easily integrate with popular identity providers like Google, Microsoft, GitHub, and enterprise identity providers.

Now we can proceed to Phase 9 with a solid understanding of modern authentication flows!

\*\*

## how to implement Refresh token concept ?

Excellent question! **Refresh tokens** are crucial for maintaining secure, long-term authentication without compromising security. Let me provide a comprehensive guide on implementing refresh tokens in Spring Security.

## Refresh Token Implementation Guide

### What are Refresh Tokens?

**Refresh tokens** are long-lived credentials used to obtain new access tokens when the current access token expires. They solve the problem of balancing security (short-lived tokens) with user experience (not forcing frequent re-authentication). [\[293\]](#) [\[294\]](#) [\[295\]](#)

### Key Characteristics

Aspect	Access Token	Refresh Token
Lifespan	Short (15-30 minutes)	Long (days/weeks)
Purpose	Access protected resources	Obtain new access tokens

Aspect	Access Token	Refresh Token
<b>Usage Frequency</b>	Every API call	Only when access token expires
<b>Storage</b>	Memory/client	Secure storage (database)
<b>Security Risk</b>	Lower (short-lived)	Higher (long-lived)

## Why Use Refresh Tokens?

- Better Security:** Short-lived access tokens limit exposure window [\[296\]](#) [\[294\]](#)
- Better UX:** Users don't need to re-authenticate frequently [\[295\]](#) [\[293\]](#)
- Token Rotation:** Can implement rotation for enhanced security [\[294\]](#) [\[296\]](#)
- Selective Revocation:** Can revoke refresh tokens without affecting current sessions [\[293\]](#)

## Complete Refresh Token Implementation

### 1. Refresh Token Entity

```

@Entity
@Table(name = "refresh_tokens")
public class RefreshToken {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 @OneToOne
 @JoinColumn(name = "user_id", referencedColumnName = "id")
 private User user;

 @Column(nullable = false, unique = true)
 private String token;

 @Column(nullable = false)
 private Instant expiryDate;

 @Column(nullable = false)
 private Instant createdAt;

 @Column
 private Instant lastUsedAt;

 // Constructors, getters, setters
 public RefreshToken() {
 this.createdAt = Instant.now();
 }

 public RefreshToken(User user, String token, Instant expiryDate) {
 this();
 this.user = user;
 }
}

```

```

 this.token = token;
 this.expiryDate = expiryDate;
 }

 // Getters and setters...
}

```

## 2. Refresh Token Repository

```

@Repository
public interface RefreshTokenRepository extends JpaRepository<RefreshToken, Long> {

 Optional<RefreshToken> findByToken(String token);

 Optional<RefreshToken> findByUser(User user);

 @Modifying
 @Transactional
 int deleteByUser(User user);

 @Modifying
 @Transactional
 int deleteByToken(String token);

 // Clean up expired tokens
 @Modifying
 @Transactional
 @Query("DELETE FROM RefreshToken rt WHERE rt.expiryDate < :now")
 int deleteExpiredTokens(@Param("now") Instant now);
}

```

## 3. Enhanced JWT Utility Service

```

@Service
public class JwtService {

 @Value("${jwt.secret}")
 private String jwtSecret;

 @Value("${jwt.accessTokenExpirationMs}")
 private long accessTokenExpirationMs;

 @Value("${jwt.refreshTokenExpirationMs}")
 private long refreshTokenExpirationMs;

 private final SecretKey secretKey;

 public JwtService() {
 this.secretKey = Keys.hmacShaKeyFor(jwtSecret.getBytes());
 }

 // Generate short-lived access token
 public String generateAccessToken(UserDetails userDetails) {

```

```

 return generateAccessToken(userDetails.getUsername(), userDetails.getAuthorities());
 }

 public String generateAccessToken(String username, Collection<? extends GrantedAuthority> authorities) {
 Map<String, Object> claims = new HashMap<>();
 claims.put("roles", authorities.stream()
 .map(GrantedAuthority::getAuthority)
 .collect(Collectors.toList())));
 claims.put("type", "access");

 return Jwts.builder()
 .setClaims(claims)
 .setSubject(username)
 .setIssuedAt(new Date())
 .setExpiration(new Date(System.currentTimeMillis() + accessTokenExpirationMs))
 .signWith(secretKey, SignatureAlgorithm.HS256)
 .compact();
 }

 // Generate refresh token (just a UUID - not JWT)
 public String generateRefreshToken() {
 return UUID.randomUUID().toString();
 }

 // Validate access token
 public boolean validateAccessToken(String token) {
 try {
 Jwts.parserBuilder()
 .setSigningKey(secretKey)
 .build()
 .parseClaimsJws(token);
 return true;
 } catch (JwtException | IllegalArgumentException e) {
 return false;
 }
 }

 // Extract username from access token
 public String getUsernameFromAccessToken(String token) {
 return Jwts.parserBuilder()
 .setSigningKey(secretKey)
 .build()
 .parseClaimsJws(token)
 .getBody()
 .getSubject();
 }

 // Check if access token is expired
 public boolean isAccessTokenExpired(String token) {
 try {
 Claims claims = Jwts.parserBuilder()
 .setSigningKey(secretKey)
 .build()
 .parseClaimsJws(token)
 .getBody();
 return claims.getExpiration().before(new Date());
 }
 }
}

```

```

 } catch (JwtException e) {
 return true;
 }
 }
}

```

## 4. Refresh Token Service

```

@Service
@Transactional
public class RefreshTokenService {

 @Value("${jwt.refreshTokenExpirationMs}")
 private long refreshTokenExpirationMs;

 @Autowired
 private RefreshTokenRepository refreshTokenRepository;

 @Autowired
 private UserRepository userRepository;

 @Autowired
 private JwtService jwtService;

 // Create refresh token for user
 public RefreshToken createRefreshToken(Long userId) {
 User user = userRepository.findById(userId)
 .orElseThrow(() -> new RuntimeException("User not found"));

 // Delete existing refresh token for this user (single device login)
 refreshTokenRepository.deleteByUser(user);

 // Create new refresh token
 String tokenValue = jwtService.generateRefreshToken();
 Instant expiryDate = Instant.now().plusMillis(refreshTokenExpirationMs);

 RefreshToken refreshToken = new RefreshToken(user, tokenValue, expiryDate);
 return refreshTokenRepository.save(refreshToken);
 }

 // Find refresh token by token value
 public Optional<RefreshToken> findByToken(String token) {
 return refreshTokenRepository.findByToken(token);
 }

 // Verify refresh token expiration
 public RefreshToken verifyExpiration(RefreshToken token) {
 if (token.getExpiryDate().compareTo(Instant.now()) < 0) {
 refreshTokenRepository.delete(token);
 throw new TokenRefreshException(token.getToken(),
 "Refresh token was expired. Please make a new signin request");
 }
 }

 // Update last used timestamp
 token.setLastUsedAt(Instant.now());
}

```

```

 refreshTokenRepository.save(token);

 return token;
 }

 // Delete refresh token by user
 public int deleteByUserId(Long userId) {
 User user = userRepository.findById(userId)
 .orElseThrow(() -> new RuntimeException("User not found"));
 return refreshTokenRepository.deleteByUser(user);
 }

 // Refresh token rotation - issue new refresh token and invalidate old one
 public RefreshToken rotateRefreshToken(RefreshToken oldToken) {
 // Delete old token
 refreshTokenRepository.delete(oldToken);

 // Create new refresh token
 return createRefreshToken(oldToken.getUser().getId());
 }

 // Clean up expired tokens (scheduled job)
 @Scheduled(fixedRate = 3600000) // Run every hour
 public void cleanupExpiredTokens() {
 int deletedCount = refreshTokenRepository.deleteExpiredTokens(Instant.now());
 if (deletedCount > 0) {
 log.info("Cleaned up {} expired refresh tokens", deletedCount);
 }
 }
}

```

## 5. Authentication Controller with Refresh Token

```

@RestController
@RequestMapping("/api/auth")
public class AuthController {

 @Autowired
 private AuthenticationManager authenticationManager;

 @Autowired
 private JwtService jwtService;

 @Autowired
 private RefreshTokenService refreshTokenService;

 @Autowired
 private UserDetailsService userDetailsService;

 // Enhanced signin with refresh token
 @PostMapping("/signin")
 public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest)
 try {
 // Authenticate user
 Authentication authentication = authenticationManager.authenticate(

```

```

 new UsernamePasswordAuthenticationToken(
 loginRequest.getUsername(),
 loginRequest.getPassword()
);

 SecurityContextHolder.getContext().setAuthentication(authentication);

 UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal()

 // Generate access token
 String accessToken = jwtService.generateAccessToken(userDetails);

 // Generate refresh token
 RefreshToken refreshToken = refreshTokenService.createRefreshToken(userDetails);

 // Extract user roles
 List<String> roles = userDetails.getAuthorities().stream()
 .map(GrantedAuthority::getAuthority)
 .collect(Collectors.toList());

 return ResponseEntity.ok(new JwtResponse(
 accessToken,
 refreshToken.getToken(),
 userDetails.getId(),
 userDetails.getUsername(),
 userDetails.getEmail(),
 roles
));
 }

 } catch (BadCredentialsException e) {
 return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
 .body(new MessageResponse("Invalid username or password"));
 }
}

// Refresh token endpoint
@PostMapping("/refresh-token")
public ResponseEntity<?> refreshToken(@Valid @RequestBody TokenRefreshRequest request)
{
 String requestRefreshToken = request.getRefreshToken();

 return refreshTokenService.findByToken(requestRefreshToken)
 .map(refreshTokenService::verifyExpiration)
 .map(refreshToken -> {
 User user = refreshToken.getUser();

 // Load user details for new access token
 UserDetails userDetails = userDetailsService.loadUserByUsername(user.getUsername());

 // Generate new access token
 String newAccessToken = jwtService.generateAccessToken(userDetails);

 // Optional: Implement refresh token rotation
 RefreshToken newRefreshToken = refreshTokenService.rotateRefreshToken(requestRefreshToken);

 return ResponseEntity.ok(new TokenRefreshResponse(
 newAccessToken,

```

```

 newRefreshToken.getToken()
)));
}
.orElseThrow(() -> new TokenRefreshException(requestRefreshToken,
 "Refresh token is not in database!"));
}

// Logout endpoint - invalidate refresh token
@PostMapping("/signout")
public ResponseEntity<?> logoutUser() {
 UserDetailsImpl userDetails = (UserDetailsImpl) SecurityContextHolder.getContext()
 .getAuthentication().getPrincipal();

 if (userDetails != null) {
 refreshTokenService.deleteByUserId(userDetails.getId());
 }

 return ResponseEntity.ok(new MessageResponse("User signed out successfully!"));
}
}

```

## 6. Enhanced JWT Authentication Filter

```

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

 @Autowired
 private JwtService jwtService;

 @Autowired
 private UserDetailsService userDetailsService;

 @Override
 protected void doFilterInternal(HttpServletRequest request,
 HttpServletResponse response,
 FilterChain filterChain)
 throws ServletException, IOException {

 try {
 String jwt = parseJwt(request);

 if (jwt != null && jwtService.validateAccessToken(jwt)) {
 String username = jwtService.getUsernameFromAccessToken(jwt);

 UserDetails userDetails = userDetailsService.loadUserByUsername(username)
 UsernamePasswordAuthenticationToken authentication =
 new UsernamePasswordAuthenticationToken(userDetails,
 null,
 userDetails.getAuthorities());
 authentication.setDetails(new WebAuthenticationDetailsSource().buildDetail

 SecurityContextHolder.getContext().setAuthentication(authentication);
 }
 } catch (Exception e) {
 logger.error("Cannot set user authentication: ", e);
 }
 }
}

```

```

 }

 filterChain.doFilter(request, response);
}

private String parseJwt(HttpServletRequest request) {
 String headerAuth = request.getHeader("Authorization");

 if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer ")) {
 return headerAuth.substring(7);
 }

 return null;
}

}

```

## 7. Request/Response DTOs

```

// JWT Response with both tokens
public class JwtResponse {
 private String accessToken;
 private String refreshToken;
 private String tokenType = "Bearer";
 private Long id;
 private String username;
 private String email;
 private List<String> roles;

 // Constructor, getters, setters
 public JwtResponse(String accessToken, String refreshToken, Long id,
 String username, String email, List<String> roles) {
 this.accessToken = accessToken;
 this.refreshToken = refreshToken;
 this.id = id;
 this.username = username;
 this.email = email;
 this.roles = roles;
 }

 // Getters and setters...
}

// Token refresh request
public class TokenRefreshRequest {
 @NotBlank
 private String refreshToken;

 // Constructor, getters, setters
}

// Token refresh response
public class TokenRefreshResponse {
 private String accessToken;
 private String refreshToken;
 private String tokenType = "Bearer";
}
```

```

// Constructor, getters, setters
public TokenRefreshResponse(String accessToken, String refreshToken) {
 this.accessToken = accessToken;
 this.refreshToken = refreshToken;
}

// Getters and setters...
}

```

## 8. Exception Handling

```

@ResponseStatus(HttpStatus.FORBIDDEN)
public class TokenRefreshException extends RuntimeException {
 private static final long serialVersionUID = 1L;

 public TokenRefreshException(String token, String message) {
 super(String.format("Failed for [%s]: %s", token, message));
 }
}

@RestControllerAdvice
public class TokenControllerAdvice {

 @ExceptionHandler(value = TokenRefreshException.class)
 @ResponseStatus(HttpStatus.FORBIDDEN)
 public ErrorMessage handleTokenRefreshException(TokenRefreshException ex, WebRequest
 return new ErrorMessage(
 HttpStatus.FORBIDDEN.value(),
 new Date(),
 ex.getMessage(),
 request.getDescription(false)
);
 }
}

```

## Refresh Token Rotation (Advanced Security)

**Token rotation** issues a new refresh token every time it's used, invalidating the old one. [\[297\]](#) [\[296\]](#) [\[294\]](#)

### Benefits of Token Rotation

1. **Breach Detection:** Detect if refresh tokens are compromised [\[296\]](#) [\[294\]](#)
2. **Reduced Attack Window:** Limits exposure time [\[297\]](#)
3. **Automatic Revocation:** Invalid usage triggers token family revocation [\[296\]](#)

## Implementation

```
@Service
public class TokenRotationService {

 @Autowired
 private RefreshTokenService refreshTokenService;

 @Autowired
 private JwtService jwtService;

 public TokenRefreshResponse refreshWithRotation(String oldRefreshToken) {
 return refreshTokenService.findByToken(oldRefreshToken)
 .map(refreshTokenService::verifyExpiration)
 .map(refreshToken -> {
 User user = refreshToken.getUser();

 // Generate new access token
 UserDetails userDetails = loadUserDetails(user.getUsername());
 String newAccessToken = jwtService.generateAccessToken(userDetails);

 // Rotate refresh token
 RefreshToken newRefreshToken = refreshTokenService.rotateRefreshToken(refreshToken);

 return new TokenRefreshResponse(newAccessToken, newRefreshToken.getToken());
 })
 .orElseThrow(() -> new TokenRefreshException(oldRefreshToken,
 "Refresh token not found or already used"));
 }
}
```

## Client-Side Implementation

### JavaScript Example

```
class TokenManager {
 constructor() {
 this.accessToken = localStorage.getItem('accessToken');
 this.refreshToken = localStorage.getItem('refreshToken');
 }

 async makeApiCall(url, options = {}) {
 // Try with current access token
 let response = await fetch(url, {
 ...options,
 headers: {
 ...options.headers,
 'Authorization': `Bearer ${this.accessToken}`
 }
 });

 // If access token expired, refresh and retry
 }
}
```

```

 if (response.status === 401) {
 const refreshed = await this.refreshAccessToken();
 if (refreshed) {
 response = await fetch(url, {
 ...options,
 headers: {
 ...options.headers,
 'Authorization': `Bearer ${this.accessToken}`
 }
 });
 } else {
 // Refresh failed - redirect to login
 window.location.href = '/login';
 return;
 }
 }

 return response;
 }

 async refreshAccessToken() {
 try {
 const response = await fetch('/api/auth/refresh-token', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json'
 },
 body: JSON.stringify({
 refreshToken: this.refreshToken
 })
 });

 if (response.ok) {
 const data = await response.json();
 this.accessToken = data.accessToken;
 this.refreshToken = data.refreshToken; // If using rotation

 localStorage.setItem('accessToken', this.accessToken);
 localStorage.setItem('refreshToken', this.refreshToken);

 return true;
 } else {
 // Refresh token invalid
 this.clearTokens();
 return false;
 }
 } catch (error) {
 console.error('Token refresh failed:', error);
 this.clearTokens();
 return false;
 }
 }

 clearTokens() {
 this.accessToken = null;
 this.refreshToken = null;
 }
}

```

```

 localStorage.removeItem('accessToken');
 localStorage.removeItem('refreshToken');
 }
}

// Usage
const tokenManager = new TokenManager();
tokenManager.makeApiCall('/api/protected-endpoint');

```

## Configuration Properties

```

jwt:
 secret: mySecretKey
 accessTokenExpirationMs: 900000 # 15 minutes
 refreshTokenExpirationMs: 604800000 # 7 days

spring:
 jpa:
 hibernate:
 ddl-auto: update
 properties:
 hibernate:
 dialect: org.hibernate.dialect.MySQL8Dialect

```

## Best Practices

### Security Best Practices [\[298\]](#) [\[294\]](#) [\[295\]](#)

1. **Short-lived Access Tokens:** 15-30 minutes maximum
2. **Secure Storage:** Store refresh tokens securely (HttpOnly cookies for web, Keychain for mobile)
3. **Token Rotation:** Implement refresh token rotation for enhanced security
4. **Proper Validation:** Always validate token expiration and integrity
5. **Revocation Support:** Ability to revoke refresh tokens
6. **Rate Limiting:** Limit refresh token usage frequency

### Performance Considerations

1. **Database Cleanup:** Regularly clean up expired tokens
2. **Caching:** Consider caching valid tokens
3. **Async Processing:** Handle token operations asynchronously where possible

## Monitoring and Logging

```
@Component
public class TokenAuditService {

 private static final Logger log = LoggerFactory.getLogger(TokenAuditService.class);

 public void logTokenRefresh(String username, String oldToken, String newToken) {
 log.info("Token refreshed for user: {}, old token: {}, new token: {}",
 username, oldToken.substring(0, 8) + "...", newToken.substring(0, 8) + "...")
 }

 public void logTokenRevocation(String username, String token, String reason) {
 log.warn("Token revoked for user: {}, token: {}, reason: {}",
 username, token.substring(0, 8) + "...", reason);
 }
}
```

## Key Takeaways

1. **Dual Token Strategy:** Short-lived access tokens + long-lived refresh tokens balance security and UX [\[295\]](#) [\[293\]](#)
2. **Secure Storage:** Refresh tokens need secure storage and proper validation [\[294\]](#) [\[295\]](#)
3. **Token Rotation:** Enhances security by limiting token reuse [\[297\]](#) [\[296\]](#)
4. **Proper Exception Handling:** Handle token expiration and refresh failures gracefully [\[299\]](#) [\[293\]](#)
5. **Database Management:** Store refresh tokens persistently and clean up expired ones [\[299\]](#) [\[293\]](#)
6. **Client Implementation:** Automatic token refresh in client applications for seamless UX [\[300\]](#)

This comprehensive refresh token implementation provides security, scalability, and excellent user experience. The rotation mechanism adds an extra layer of security by detecting and preventing token abuse.

\*\*

## Phase 9: Advanced Security Topics and Modern Features

This phase explores cutting-edge security features and enterprise-grade patterns that represent the future of application security in Spring.

## 1. WebAuthn and Passwordless Authentication (Passkeys)

### What are Passkeys?

**Passkeys** are FIDO credentials that replace passwords with cryptographic key pairs for **phishing-resistant authentication**. They use: [\[313\]](#) [\[314\]](#) [\[315\]](#)

- **Biometric authentication** (Face ID, Touch ID, fingerprints)
- **Hardware security keys** (YubiKey, etc.)
- **Device-based authentication** (stored in secure device hardware)

### Benefits over Traditional Passwords

Traditional Passwords	Passkeys
Can be stolen/phished	Phishing-resistant
Weak/reused passwords	Strong cryptographic keys
Centrally stored	Decentralized (device-based)
Manual entry required	Seamless biometric auth
Forgotten passwords	Cannot be forgotten

### Spring Security 6.4+ WebAuthn Implementation

Spring Security now provides **native WebAuthn support**: [\[314\]](#) [\[316\]](#)

### 1. Enable WebAuthn Configuration

```
@Configuration
@EnableWebSecurity
public class WebAuthnConfig {

 @Bean
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
 http
 .authorizeHttpRequests(authorize -> authorize
 .requestMatchers("/", "/login", "/webauthn/**").permitAll()
 .anyRequest().authenticated()
)
 .formLogin(withDefaults())
 .webAuthn(withDefaults()); // Enable WebAuthn
 return http.build();
 }

 @Bean
 public WebAuthnUserDetailsService webAuthnUserDetailsService() {
 return new InMemoryWebAuthnUserDetailsService();
 }
}
```

## 2. WebAuthn Registration Flow

```
@RestController
@RequestMapping("/webauthn")
public class WebAuthnController {

 @Autowired
 private WebAuthnRegistrationOptionsProvider optionsProvider;

 @Autowired
 private WebAuthnUserDetailsService userDetailsService;

 // Step 1: Generate registration options
 @PostMapping("/register/options")
 public PublicKeyCredentialCreationOptions createRegistrationOptions(
 @AuthenticationPrincipal UserDetails user) {

 return optionsProvider.getCreationOptions(
 WebAuthnRegistrationOptionsProvider.CreateOptionsRequest.builder()
 .username(user.getUsername())
 .displayName(user.getUsername())
 .build()
);
 }

 // Step 2: Complete registration
 @PostMapping("/register")
 public ResponseEntity<?> registerCredential(
 @RequestBody PublicKeyCredentialRegistrationRequest request,
 @AuthenticationPrincipal UserDetails user) {

 try {
 WebAuthnUser webAuthnUser = userDetailsService.loadUserByUsername(user.getUsername());

 // Verify and store the credential
 PublicKeyCredential credential = request.getPublicKey().getCredential();
 webAuthnUser.addCredential(credential);

 userDetailsService.save(webAuthnUser);

 return ResponseEntity.ok(Map.of("success", true, "message", "Passkey registered"));
 } catch (Exception e) {
 return ResponseEntity.badRequest()
 .body(Map.of("success", false, "error", e.getMessage()));
 }
 }
}
```

### 3. WebAuthn Authentication Flow

```
@RestController
public class WebAuthnAuthController {

 @PostMapping("/login/webauthn/options")
 public PublicKeyCredentialRequestOptions createAuthenticationOptions() {
 return WebAuthnRegistrationOptionsProvider.getRequestOptions();
 }

 @PostMapping("/login/webauthn")
 public ResponseEntity<?> authenticateWithWebAuthn(
 @RequestBody PublicKeyCredentialAuthenticationRequest request) {

 try {
 // Verify the WebAuthn assertion
 String username = verifyWebAuthnAssertion(request);

 // Create authentication token
 UserDetails userDetails = userDetailsService.loadUserByUsername(username);
 Authentication auth = new UsernamePasswordAuthenticationToken(
 userDetails, null, userDetails.getAuthorities());

 SecurityContextHolder.getContext().setAuthentication(auth);

 // Generate JWT token
 String jwtToken = jwtService.generateToken(userDetails);

 return ResponseEntity.ok(new JwtResponse(jwtToken));

 } catch (Exception e) {
 return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
 .body(Map.of("error", "WebAuthn authentication failed"));
 }
 }
}
```

### 4. Frontend WebAuthn Integration

```
// Registration flow
async function registerPasskey() {
 try {
 // Get registration options from server
 const optionsResponse = await fetch('/webauthn/register/options', {
 method: 'POST',
 headers: { 'Authorization': `Bearer ${accessToken}` }
 });
 const options = await optionsResponse.json();

 // Create credential using WebAuthn API
 const credential = await navigator.credentials.create({
 publicKey: {
 challenge: base64ToUint8Array(options.challenge),
 rp: options.rp,
```

```

 user: {
 id: base64ToUint8Array(options.user.id),
 name: options.user.name,
 displayName: options.user.displayName
 },
 pubKeyCredParams: options.pubKeyCredParams,
 timeout: options.timeout,
 excludeCredentials: options.excludeCredentials?.map(cred => ({
 ...cred,
 id: base64ToUint8Array(cred.id)
 }))
 },
};

// Send credential to server
const registerResponse = await fetch('/webauthn/register', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 'Authorization': `Bearer ${accessToken}`
 },
 body: JSON.stringify({
 publicKey: {
 credential: {
 id: credential.id,
 rawId: uint8ArrayToBase64(credential.rawId),
 response: {
 clientDataJSON: uint8ArrayToBase64(credential.response.clientData),
 attestationObject: uint8ArrayToBase64(credential.response.attestationObject)
 },
 type: credential.type
 }
 }
 })
});

const result = await registerResponse.json();
console.log('Passkey registered:', result);

} catch (error) {
 console.error('Passkey registration failed:', error);
}
}

// Authentication flow
async function authenticateWithPasskey() {
 try {
 // Get authentication options
 const optionsResponse = await fetch('/login/webauthn/options', {
 method: 'POST'
 });
 const options = await optionsResponse.json();

 // Get assertion using WebAuthn API
 const assertion = await navigator.credentials.get({
 publicKey: {

```

```

 challenge: base64ToUint8Array(options.challenge),
 allowCredentials: options.allowCredentials?.map(cred => ({
 ...cred,
 id: base64ToUint8Array(cred.id)
 })),
 timeout: options.timeout
 }
});

// Send assertion to server
const authResponse = await fetch('/login/webauthn', {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify({
 publicKey: {
 credential: {
 id: assertion.id,
 rawId: uint8ArrayToBase64(assertion.rawId),
 response: {
 clientDataJSON: uint8ArrayToBase64(assertion.response.clientData),
 authenticatorData: uint8ArrayToBase64(assertion.response.authenticatorData),
 signature: uint8ArrayToBase64(assertion.response.signature),
 userHandle: assertion.response.userHandle ?
 uint8ArrayToBase64(assertion.response.userHandle) : null
 },
 type: assertion.type
 }
 }
 }
});
}

const result = await authResponse.json();
if (result.accessToken) {
 localStorage.setItem('accessToken', result.accessToken);
 window.location.href = '/dashboard';
}

} catch (error) {
 console.error('Passkey authentication failed:', error);
}
}
}

```

## 2. Multi-Factor Authentication (2FA/MFA)

### TOTP (Time-based One-Time Password) Implementation

The most common MFA approach using authenticator apps like Google Authenticator: [\[317\]](#) [\[318\]](#) [\[319\]](#)

## 1. Dependencies

```
<dependency>
 <groupId>dev.samstevens.totp</groupId>
 ...
 <artifactId>totp-spring-boot-starter</artifactId>
 ...
 <version>1.7.1</version>
</dependency>
```

## 2. User Entity with MFA Support

```
@Entity
public class User implements UserDetails {

 // ... existing fields

 @Column(name = "mfa_enabled")
 private boolean mfaEnabled = false;

 @Column(name = "totp_secret")
 private String totpSecret;

 @Column(name = "backup_codes")
 @ElementCollection
 private Set<String> backupCodes = new HashSet<>();

 @Column(name = "mfa_setup_complete")
 private boolean mfaSetupComplete = false;

 // Constructors, getters, setters...
}
```

## 3. MFA Service Implementation

```
@Service
@Transactional
public class MfaService {

 @Autowired
 private SecretGenerator secretGenerator;

 @Autowired
 private CodeGenerator codeGenerator;

 @Autowired
 private CodeVerifier codeVerifier;

 @Autowired
 private QrGenerator qrGenerator;

 @Autowired
```

```

private UserRepository userRepository;

// Generate TOTP secret for new user
public String generateTotpSecret(String username) {
 User user = userRepository.findByUsername(username)
 .orElseThrow(() -> new UsernameNotFoundException("User not found"));

 String secret = secretGenerator.generate();
 user.setTotpSecret(secret);
 user.setMfaEnabled(true);

 userRepository.save(user);
 return secret;
}

// Generate QR code for authenticator app setup
public String generateQrCode(String username) throws QrGenerationException {
 User user = userRepository.findByUsername(username)
 .orElseThrow(() -> new UsernameNotFoundException("User not found"));

 String secret = user.getTotpSecret();
 if (secret == null) {
 secret = generateTotpSecret(username);
 }

 return qrGenerator.generate(new QrData.Builder()
 .label(username)
 .secret(secret)
 .issuer("MyApplication")
 .algorithm(HashingAlgorithm.SHA1)
 .digits(6)
 .period(30)
 .build()
);
}

// Verify TOTP code
public boolean verifyTotpCode(String username, String code) {
 User user = userRepository.findByUsername(username)
 .orElseThrow(() -> new UsernameNotFoundException("User not found"));

 if (!user.isMfaEnabled() || user.getTotpSecret() == null) {
 return false;
 }

 return codeVerifier.isValidCode(user.getTotpSecret(), code);
}

// Generate backup codes
public Set<String> generateBackupCodes(String username) {
 User user = userRepository.findByUsername(username)
 .orElseThrow(() -> new UsernameNotFoundException("User not found"));

 Set<String> backupCodes = new HashSet<>();
 SecureRandom random = new SecureRandom();

```

```

 for (int i = 0; i < 10; i++) {
 backupCodes.add(String.format("%08d", random.nextInt(100000000)));
 }

 // Hash backup codes before storing
 Set<String> hashedCodes = backupCodes.stream()
 .map(code -> passwordEncoder.encode(code))
 .collect(Collectors.toSet());

 user.setBackupCodes(hashedCodes);
 userRepository.save(user);

 return backupCodes; // Return plain codes to user (one time only)
 }

 // Verify backup code
 public boolean verifyBackupCode(String username, String code) {
 User user = userRepository.findByUsername(username)
 .orElseThrow(() -> new UsernameNotFoundException("User not found"));

 return user.getBackupCodes().stream()
 .anyMatch(hashedCode -> {
 if (passwordEncoder.matches(code, hashedCode)) {
 // Remove used backup code
 user.getBackupCodes().remove(hashedCode);
 userRepository.save(user);
 return true;
 }
 return false;
 });
 }
}

```

## 4. Two-Step Authentication Filter

```

@Component
public class MfaAuthenticationFilter extends OncePerRequestFilter {

 @Autowired
 private MfaService mfaService;

 @Autowired
 private JwtService jwtService;

 @Override
 protected void doFilterInternal(HttpServletRequest request,
 HttpServletResponse response,
 FilterChain filterChain)
 throws ServletException, IOException {

 // Skip MFA for certain endpoints
 String requestURI = request.getRequestURI();
 if (shouldSkipMfaCheck(requestURI)) {
 filterChain.doFilter(request, response);
 return;
 }

 // Generate and store MFA code
 String mfaCode = generateMfaCode();
 HttpSession session = request.getSession();
 session.setAttribute("mfaCode", mfaCode);
 response.setStatus(HttpStatus.OK.value());
 response.getWriter().write(mfaCode);
 }
}

```

```

 }

 Authentication auth = SecurityContextHolder.getContext().getAuthentication();

 if (auth != null && auth.isAuthenticated() &&
 !(auth instanceof MfaAuthenticationToken)) {

 String username = auth.getName();
 User user = userRepository.findByUsername(username).orElse(null);

 if (user != null && user.isMfaEnabled() && !user.isMfaSetupComplete()) {
 // Redirect to MFA setup
 response.sendRedirect("/mfa/setup");
 return;
 }

 if (user != null && user.isMfaEnabled() && user.isMfaSetupComplete()) {
 // Check if MFA is already verified in this session
 String mfaToken = request.getHeader("X-MFA-Token");
 if (mfaToken == null || !jwtService.validateMfaToken(mfaToken, username))
 response.setStatus(HttpStatus.UNAUTHORIZED.value());
 response.getWriter().write("{\"error\":\"MFA verification required\"");
 return;
 }
 }

 filterChain.doFilter(request, response);
}

private boolean shouldSkipMfaCheck(String requestURI) {
 return requestURI.startsWith("/mfa/") ||
 requestURI.startsWith("/auth/") ||
 requestURI.startsWith("/public/");
}
}

```

## 5. MFA Controller

```

@RestController
@RequestMapping("/mfa")
public class MfaController {

 @Autowired
 private MfaService mfaService;

 @Autowired
 private JwtService jwtService;

 // Setup MFA - generate QR code
 @PostMapping("/setup")
 public ResponseEntity<?> setupMfa(@AuthenticationPrincipal UserDetails userDetails) {
 try {
 String qrCode = mfaService.generateQrCode(userDetails.getUsername());
 Set<String> backupCodes = mfaService.generateBackupCodes(userDetails.getUsername());
 return ResponseEntity.ok().body(Map.of("qrCode", qrCode, "backupCodes", backupCodes));
 } catch (Exception e) {
 return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(e.getMessage());
 }
 }
}

```

```

 return ResponseEntity.ok(Map.of(
 "qrCode", qrCode,
 "backupCodes", backupCodes,
 "message", "Scan QR code with your authenticator app"
));

 } catch (Exception e) {
 return ResponseEntity.badRequest()
 .body(Map.of("error", "Failed to setup MFA: " + e.getMessage()));
 }
}

// Verify MFA setup
@PostMapping("/verify-setup")
public ResponseEntity<?> verifyMfaSetup(
 @RequestBody MfaVerificationRequest request,
 @AuthenticationPrincipal UserDetails userDetails) {

 boolean isValid = mfaService.verifyTotpCode(userDetails.getUsername(), request.getCode());

 if (isValid) {
 // Mark MFA setup as complete
 User user = userRepository.findByUsername(userDetails.getUsername()).orElseThrow();
 user.setMfaSetupComplete(true);
 userRepository.save(user);

 return ResponseEntity.ok(Map.of("success", true, "message", "MFA setup complete"));
 } else {
 return ResponseEntity.badRequest()
 .body(Map.of("error", "Invalid verification code"));
 }
}

// Verify MFA during login
@PostMapping("/verify")
public ResponseEntity<?> verifyMfa(@RequestBody MfaVerificationRequest request) {
 try {
 String username = getCurrentUsername();
 boolean isValid = false;

 // Try TOTP code first
 if (request.getCode().length() == 6) {
 isValid = mfaService.verifyTotpCode(username, request.getCode());
 }
 // Try backup code if TOTP fails
 else if (request.getCode().length() == 8) {
 isValid = mfaService.verifyBackupCode(username, request.getCode());
 }

 if (isValid) {
 // Generate MFA verification token
 String mfaToken = jwtService.generateMfaToken(username);

 return ResponseEntity.ok(Map.of(
 "success", true,

```

```

 "mfaToken", mfaToken,
 "message", "MFA verification successful"
);
} else {
 return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
 .body(Map.of("error", "Invalid MFA code"));
}

} catch (Exception e) {
 return ResponseEntity.badRequest()
 .body(Map.of("error", "MFA verification failed: " + e.getMessage()));
}
}

// Disable MFA
@PostMapping("/disable")
public ResponseEntity<?> disableMfa(
 @RequestBody MfaVerificationRequest request,
 @AuthenticationPrincipal UserDetails userDetails) {

 boolean isValid = mfaService.verifyTotpCode(userDetails.getUsername(), request.get
 if (isValid) {
 User user = userRepository.findByUsername(userDetails.getUsername()).orElseT
 user.setMfaEnabled(false);
 user.setTotpSecret(null);
 user.setBackupCodes(new HashSet<>());
 user.setMfaSetupComplete(false);
 userRepository.save(user);

 return ResponseEntity.ok(Map.of("success", true, "message", "MFA disabled suc
 } else {
 return ResponseEntity.badRequest()
 .body(Map.of("error", "Invalid verification code"));
 }
}

private String getCurrentUsername() {
 Authentication auth = SecurityContextHolder.getContext().getAuthentication();
 return auth != null ? auth.getName() : null;
}
}

```

### 3. Dynamic Client Registration

**Dynamic Client Registration (DCR)** enables clients to register themselves programmatically at runtime: [\[320\]](#) [\[321\]](#) [\[322\]](#)

## Benefits

- **Automated registration** without manual intervention
- **Scalable for microservices** and ephemeral workloads
- **Self-service client management** for developers
- **Support for transient clients** like AI agents

## Spring Authorization Server DCR Implementation

### 1. Enable DCR

```
@Configuration
@EnableWebSecurity
public class AuthorizationServerConfig {

 @Bean
 public SecurityFilterChain authorizationServerSecurityFilterChain(HttpSecurity http)
 OAuth2AuthorizationServerConfigurer authorizationServerConfigurer =
 OAuth2AuthorizationServerConfigurer.authorizationServer();

 http
 .securityMatcher(authorizationServerConfigurer.getEndpointsMatcher())
 .with(authorizationServerConfigurer, authorizationServer ->
 authorizationServer
 .oidc(oidc -> oidc
 .clientRegistrationEndpoint(clientRegistration ->
 clientRegistration
 .authenticationProviders(configureCustomClientMetadataConverte
)
)
)
 .authorizeHttpRequests(authorize -> authorize
 .anyRequest().authenticated()
);
 return http.build();
}

private Consumer<List<AuthenticationProvider>> configureCustomClientMetadataConverte
 return providers -> {
 providers.add(new CustomClientMetadataAuthenticationProvider());
 };
}
```

### 2. Custom Client Metadata Provider

```
@Component
public class CustomClientMetadataAuthenticationProvider implements AuthenticationProvide
 @Override
 public Authentication authenticate(Authentication authentication) throws AuthenticationEx
```

```

OAuth2ClientRegistrationAuthenticationToken authenticationToken =
 (OAuth2ClientRegistrationAuthenticationToken) authentication;

OAuth2ClientRegistration.Builder clientRegistrationBuilder =
 OAuth2ClientRegistration.withRegistration(authenticationToken.getClientRegis

 // Add custom validation logic
 Map<String, Object> additionalParameters = authenticationToken.getAdditionalParam

 // Validate custom client metadata
 if (additionalParameters.containsKey("application_type")) {
 String applicationType = (String) additionalParameters.get("application_type")
 if (!"web".equals(applicationType) && !"native".equals(applicationType)) {
 throw new OAuth2AuthenticationException("invalid_client_metadata");
 }
 }

 // Set default values if not provided
 if (!additionalParameters.containsKey("token_endpoint_auth_method")) {
 clientRegistrationBuilder.tokenEndpointAuthenticationMethod(
 ClientAuthenticationMethod.CLIENT_SECRET_BASIC);
 }

 OAuth2ClientRegistration clientRegistration = clientRegistrationBuilder.build();

 return new OAuth2ClientRegistrationAuthenticationToken(
 clientRegistration, additionalParameters);
}

@Override
public boolean supports(Class<?> authentication) {
 return OAuth2ClientRegistrationAuthenticationToken.class.isAssignableFrom(authent
}

}

```

### 3. DCR Client Example

```

@RestController
@RequestMapping("/api/clients")
public class DynamicClientController {

 @Autowired
 private WebClient webClient;

 @Value("${authorization-server.registration-endpoint}")
 private String registrationEndpoint;

 // Register new client dynamically
 @PostMapping("/register")
 public ResponseEntity<?> registerClient(@RequestBody ClientRegistrationRequest request)

 Map<String, Object> registrationRequest = Map.of(
 "redirect_uris", List.of(request.getRedirectUri()),
 "grant_types", List.of("authorization_code", "refresh_token"),
 "response_types", List.of("code"),

```

```

 "scope", "openid profile email",
 "application_type", "web",
 "client_name", request.getClientName(),
 "token_endpoint_auth_method", "client_secret_basic"
);

try {
 ClientRegistrationResponse response = webClient.post()
 .uri(registrationEndpoint)
 .header("Content-Type", "application/json")
 .header("Authorization", "Bearer " + getRegistrationAccessToken())
 .body(Mono.just(registrationRequest), Map.class)
 .retrieve()
 .bodyToMono(ClientRegistrationResponse.class)
 .block();

 return ResponseEntity.ok(response);
}

} catch (Exception e) {
 return ResponseEntity.badRequest()
 .body(Map.of("error", "Client registration failed: " + e.getMessage()));
}
}

// Update existing client
@PutMapping("/{clientId}")
public ResponseEntity<?> updateClient(
 @PathVariable String clientId,
 @RequestBody ClientUpdateRequest request) {

 // Implementation for client updates using registration_access_token
 return ResponseEntity.ok(Map.of("message", "Client updated successfully"));
}

// Delete client
@DeleteMapping("/{clientId}")
public ResponseEntity<?> deleteClient(@PathVariable String clientId) {

 // Implementation for client deletion
 return ResponseEntity.ok(Map.of("message", "Client deleted successfully"));
}

private String getRegistrationAccessToken() {
 // Implement logic to obtain registration access token
 // This could be from a service account or admin credentials
 return "registration-access-token";
}
}

```

## Key Takeaways

1. **WebAuthn/Passkeys:** Represent the future of authentication - phishing-resistant and user-friendly [316] [313] [314]
2. **Multi-Factor Authentication:** Essential security layer using TOTP, SMS, or hardware tokens [318] [317]
3. **Dynamic Client Registration:** Enables scalable, automated client management for modern architectures [321] [320]
4. **Performance Optimization:** Critical for production systems handling thousands of concurrent users
5. **Modern Security Standards:** Stay current with latest security specifications and best practices

Phase 9 equips you with the most advanced Spring Security capabilities available today. These features represent enterprise-grade security suitable for high-scale, mission-critical applications.

**Next and Final Phase:** Phase 10 will cover **Professional Development and Real-World Applications** - putting it all together in complex, production scenarios.

\*\*

1. [https://www.perplexity.ai/page/spring-security-learning-roadmap-Z\\_DbsYlvTuGX.GOqLu7dBA](https://www.perplexity.ai/page/spring-security-learning-roadmap-Z_DbsYlvTuGX.GOqLu7dBA)
2. [https://dev.to/akshay\\_gengaje/roadmap-to-mastering-the-spring-framework-2o5n](https://dev.to/akshay_gengaje/roadmap-to-mastering-the-spring-framework-2o5n)
3. <https://www.geeksforgeeks.org/springboot/best-way-to-master-spring-boot-a-complete-roadmap/>
4. <https://www.toptal.com/spring/spring-security-tutorial>
5. <https://howtodoinjava.com/spring-security/spring-security-tutorial/>
6. <https://www.geeksforgeeks.org/springboot/spring-security-architecture/>
7. <https://www.geeksforgeeks.org/advance-java/spring-security-tutorial/>
8. <https://escape.tech/blog/security-best-practices-for-spring-boot-applications/>
9. <https://javalaunchpad.com/spring-security-6-x-architecture-and-step-by-step-setup-guide/>
10. <https://docs.spring.io/spring-security/site/docs/5.2.12.RELEASE/reference/html/overall-architecture.html>
11. <https://www.geeksforgeeks.org/springboot/spring-security-project-example-using-java-configuration/>
12. <https://www.javacodegeeks.com/2024/12/spring-security-6-enhanced-authentication-and-authorization.html>
13. <https://www.tothenew.com/blog/migrating-to-spring-security-6/>
14. <https://corgea.com/Learn/spring-boot-security-best-practices-2025>
15. <https://www.codejava.net/spring-security-tutorials>
16. <https://dev.to/theraldumbprogrammer/spring-security-for-beginners-part-2-4e9>
17. <https://www.codingshuttle.com/spring-boot-handbook/spring-boot-tutorial-a-comprehensive-guide-for-beginners>
18. <https://www.baeldung.com/spring-security-oauth-jwt>
19. <https://dzone.com/articles/how-to-achieve-oauth2-security-in-microservices-di>
20. <https://developer.auth0.com/resources/labs/authorization/spring-boot-microservices-security>

21. <https://spring.io/guides/tutorials/spring-security-and-angular-js>
22. [https://dev.to/douaa19\\_20/best-practices-for-securing-applications-with-spring-security-54hn](https://dev.to/douaa19_20/best-practices-for-securing-applications-with-spring-security-54hn)
23. <http://websystique.com/spring-security-tutorial/>
24. <https://dev.to/wittedtech-by-harshit/implementing-spring-security-in-microservices-architecture-a-deep-dive-1fl5>
25. <https://piotrminkowski.com/2024/03/01/microservices-with-spring-cloud-gateway-oauth2-and-keycloak/>
26. <https://www.youtube.com/watch?v=o02eZ4RHMLo>
27. <https://www.geeksforgeeks.org/advance-java/spring-boot-security-best-practices/>
28. <https://snyk.io/blog/spring-boot-security-best-practices/>
29. <https://docs.spring.io/spring-security/reference/whats-new.html>
30. <https://www.upgrad.com/blog/spring-boot-projects-topics-for-beginners/>
31. <https://spring.io/blog/2025/09/15/spring-security-6-4-10-and-6-5-4-released>
32. <https://docs.spring.io/spring-security/reference/6.0/whats-new.html>
33. <https://docs.spring.io/spring-security/reference/index.html>
34. <https://docs.spring.io/spring-security/reference/servlet/architecture.html>
35. <https://spring.io/guides/gs/securing-web>
36. <https://spring.io/guides/tutorials/spring-boot-oauth2>
37. [https://www.youtube.com/watch?v=oeni\\_9g7too](https://www.youtube.com/watch?v=oeni_9g7too)
38. <https://www.youtube.com/watch?v=bl1CPY5qXiE>
39. <https://www.codingshuttle.com/blog/best-way-to-master-spring-boot-a-complete-roadmap-2025>
40. <https://javadevjournal.com/spring-security-tutorial/>
41. <https://roadmap.sh/spring-boot>
42. <https://spring.io/projects/spring-security>
43. <https://vmsoftwarehouse.com/12-security-best-practices-in-java-development>
44. <https://www.youtube.com/watch?v=YFVzzwwTwfc>
45. <https://spring.io/projects>
46. <https://dev.to/theredlumbprogrammer/spring-security-for-beginners-part-1-4ki7>
47. <https://www.marcobehler.com/guides/spring-security>
48. <https://www.baeldung.com/courses/learn-spring-security-course>
49. <https://www.youtube.com/watch?v=X80nJ5T7YpE>
50. <https://dev.to/isaactony/learn-the-basics-of-spring-security-50b7>
51. <https://www.marcobehler.com/guides/spring-security>
52. <https://dzone.com/articles/spring-security-authentication>
53. <https://moldstud.com/articles/p-spring-security-basics-common-questions-every-developer-asks>
54. <https://www.geeksforgeeks.org/advance-java/introduction-to-spring-security-and-its-features/>
55. <https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html>
56. <https://www.javacodegeeks.com/2025/02/preventing-xss-csrf-and-sql-injection-in-javascript-applications.html>

57. <https://www.codingshuttle.com/spring-boot-handbook/spring-security-basic-introduction>
58. [https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)
59. <https://docs.spring.io/spring-security/site/docs/5.1.5.RELEASE/reference/html/web-app-security.html>
60. <https://www.baeldung.com/spring-prevent-xss>
61. <https://stackoverflow.com/questions/7722159/csrf-xss-and-sql-injection-attack-prevention-in-jsf>
62. [https://www2.microstrategy.com/producthelp/Current/LibrarySDK/Content/topics/CustomAuth\\_SpringSecurityArchitecture.htm](https://www2.microstrategy.com/producthelp/Current/LibrarySDK/Content/topics/CustomAuth_SpringSecurityArchitecture.htm)
63. <https://javalaunchpad.com/spring-security-6-x-architecture-and-step-by-step-setup-guide/>
64. <https://docs.spring.io/spring-security/reference/servlet/architecture.html>
65. <https://javalaunchpad.com/spring-security-delegatingfilterproxy-a-comprehensive-guide/>
66. <https://stackoverflow.com/questions/6725234/whats-the-point-of-spring-mvcs-delegatingfilterproxy>
67. <https://docs.spring.io/spring-security/site/docs/3.0.x/reference/security-filter-chain.html>
68. <https://docs.spring.io/spring-security/site/docs/4.0.2.RELEASE/reference/html/technical-overview.html>
69. <https://docs.spring.io/spring-security/site/docs/5.2.11.RELEASE/reference/html/overall-architecture.html>
70. <https://www.geeksforgeeks.org/advance-java/spring-security-userdetailsservice-and-userdetails-with-example/>
71. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/user-details-service.html>
72. <https://stackoverflow.com/questions/41770156/spring-add-custom-user-details-to-spring-security-user>
73. <https://howtodoinjava.com/spring-security/inmemory-jdbc-userdetails-service/>
74. <https://www.codingshuttle.com/spring-boot-handbook/internal-working-of-spring-security-advance>
75. <https://www.youtube.com/watch?v=3pYioNIPj84>
76. <https://docs.spring.io/spring-security/site/docs/4.2.x/reference/html/security-filter-chain.html>
77. <https://docs.spring.io/spring-security/reference/index.html>
78. <https://www.baeldung.com/spring-security-custom-filter>
79. <https://docs.spring.io/spring-security/reference/servlet/authentication/index.html>
80. <https://stackoverflow.com/questions/45725888/why-does-spring-security-store-securitycontext-in-the-ad-local-variable>
81. <https://howtodoinjava.com/spring-security/spring-security-context-propagation/>
82. <https://javatechonline.com/spring-security-userdetailsservice-using-spring-boot-3/>
83. <https://www.baeldung.com/spring-delegating-filter-proxy>
84. [https://www.reddit.com/r/javahelp/comments/8etx5x/spring\\_security\\_what\\_is\\_the\\_purpose\\_of\\_storing/](https://www.reddit.com/r/javahelp/comments/8etx5x/spring_security_what_is_the_purpose_of_storing/)
85. <https://www.baeldung.com/spring-security-authentication-with-a-database>
86. <https://www.youtube.com/watch?v=LThUMJByuGU>
87. <https://jdriven.com/blog/2021/07/Propagating-the-Spring-SecurityContext-to-your-Kotlin-Coroutines>
88. <https://docs.spring.io/spring-security/reference/servlet/configuration/java.html>
89. <https://stackoverflow.com/questions/76090520/spring-security-version-6-issues-with-securityfilterchain>
90. <https://www.javaguides.net/2023/04/spring-security-tutorial-in-memory-authentication.html>

91. <https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html>
92. <https://www.danvega.dev/blog/spring-security-6>
93. <https://stackoverflow.com/questions/52029258/understanding-requestmatchers-on-spring-security>
94. <https://www.baeldung.com/spring-security-configuring-urls>
95. <https://www.aegissofttech.com/articles/bcrypt-password-encoding-spring-security.html>
96. <https://www.devglan.com/spring-security/spring-boot-security-password-encoding-bcrypt-encoder>
97. <https://www.geeksforgeeks.org/advance-java/spring-security-implementation-of-bcryptpasswordencoder/>
98. <https://docs.spring.io/spring-security/reference/features/authentication/password-storage.html>
99. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/in-memory.html>
100. <https://www.geeksforgeeks.org/java/spring-security-in-memory-authentication/>
101. <https://stackoverflow.com/questions/64303106/spring-when-to-use-in-memory-authentication>
102. <https://www.initgrep.com/posts/java/spring/Spring-Security-jpa-authprovider>
103. <https://www.codejava.net/frameworks/spring-boot/spring-boot-security-authentication-with-jpa-hibernate-and-mysql>
104. <https://roytuts.com/spring-boot-security-form-based-jdbc-authentication-using-userdetailsservice/>
105. <https://github.com/TilmeezUrRehmanBhatti/spring-security-demo-03-custom-login-form>
106. <https://docs.spring.io/spring-security/site/docs/5.4.9/guides/form-javacfg.html>
107. <https://bootify.io/spring-security/form-login-with-spring-boot-thymeleaf.html>
108. <https://www.youtube.com/watch?v=ZpBayFcvtM>
109. <https://www.youtube.com/watch?v=iWwv9r9inns>
110. <https://docs.spring.io/spring-security/reference/servlet/architecture.html>
111. <https://javalaunchpad.com/spring-security-6-x-architecture-and-step-by-step-setup-guide/>
112. <https://www.baeldung.com/spring-security-registration-password-encoding-bcrypt>
113. <https://www.tothenew.com/blog/migrating-to-spring-security-6/>
114. <https://www.geeksforgeeks.org/java/spring-security-filter-chain-with-example/>
115. <https://www.youtube.com/watch?v=Rl3UpqUZq50>
116. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/index.html>
117. <https://www.youtube.com/watch?v=KuzdIOlugJ4>
118. <https://stackoverflow.com/questions/55548290/using-bcrypt-in-spring>
119. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/user-details-service.html>
120. <https://www.baeldung.com/spring-security-authentication-with-a-database>
121. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/form.html>
122. <https://www.geeksforgeeks.org/advance-java/spring-security-securing-endpoints-using-antmatchers/>
123. <https://javatechonline.com/spring-security-userdetailsservice-using-spring-boot-3/>
124. <https://www.baeldung.com/spring-security-login>
125. <https://www.geeksforgeeks.org/advance-java/implementing-database-authentication-and-authentication-with-spring-security-6/>

126. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/dao-authentication-provider.html>
127. <https://docs.spring.io/spring-security/site/docs/2.0.x/reference/html/dao-provider.html>
128. <https://www.geeksforgeeks.org/java/spring-security-authentication-providers/>
129. <https://docs.gigaspaces.com/latest/security/authenticating-against-a-database.html>
130. <https://springframework.guru/spring-boot-web-application-part-6-spring-security-with-dao-authentication-provider/>
131. <https://www.geeksforgeeks.org/advance-java/difference-between-role-and-grantedauthority-in-spring-security/>
132. <https://stackoverflow.com/questions/19525380/difference-between-role-and-grantedauthority-in-spring-security>
133. <https://javadevjournal.com/spring-security/granted-authority-vs-role-in-spring-security/>
134. <https://www.marcobehler.com/guides/spring-security>
135. [https://codemedia.io/knowledge-hub/path/difference\\_between\\_role\\_and\\_grantedauthority\\_in\\_spring\\_security](https://codemedia.io/knowledge-hub/path/difference_between_role_and_grantedauthority_in_spring_security)
136. <https://www.geeksforgeeks.org/advance-java/difference-between-hasrole-and-hasauthority-in-spring-security/>
137. <https://docs.spring.io/spring-security/site/docs/4.1.x/reference/html/authz-arch.html>
138. <https://www.baeldung.com/role-and-privilege-for-spring-security-registration>
139. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/basic.html>
140. <https://stackoverflow.com/questions/57574981/what-is-httpbasic-method-in-spring-security>
141. <https://www.codejava.net/frameworks/spring-boot/http-basic-authentication-with-in-memory-users>
142. <https://docs.oracle.com/javaee/1.4/tutorial/doc/Security5.html>
143. <https://konghq.com/blog/engineering/common-api-authentication-methods>
144. <https://testdriven.io/blog/web-authentication-methods/>
145. <https://stackoverflow.com/questions/43524211/how-do-i-add-http-basic-auth-for-a-specific-endpoint-with-spring-security>
146. <https://developer.okta.com/blog/2019/06/20/spring-preauthorize>
147. <https://www.javaguides.net/2023/04/difference-between-basic-auth-and-form-based-auth.html>
148. <http://www.bensoftware.com/forum/discussion/3996/web-access-basic-authentication-vs-http-form-based-login>
149. <https://docs.spring.io/spring-security/reference/servlet/test/mockmvc/http-basic.html>
150. <https://www.baeldung.com/spring-security-granted-authority-vs-role>
151. <https://stackoverflow.com/questions/73622268/make-daoauthenticationprovider-work-with-the-authenticationmanager-after-websecu>
152. <https://www.baeldung.com/spring-security-basic-authentication>
153. <https://www.baeldung.com/spring-security-authentication-provider>
154. <https://docs.spring.io/spring-security/reference/servlet/authorization/architecture.html>
155. [https://www.linkedin.com/posts/mohsen-noori-ardestani\\_java-springsecurityinterviewquestion-activity-7343537056809193473-9zR2](https://www.linkedin.com/posts/mohsen-noori-ardestani_java-springsecurityinterviewquestion-activity-7343537056809193473-9zR2)

156. <https://stackoverflow.com/questions/57146905/what-is-the-difference-between-hasrole-and-hasauthority-in-spring-security>
157. <https://www.youtube.com/watch?v=mg5oUXcAXL4>
158. <https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html>
159. <https://stackoverflow.com/questions/22611029/implementing-hierarchical-roles-in-spring-security>
160. <https://gist.github.com/thomasdarimont/ee9ffffdef1adb9243b12ad247478aad4>
161. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Authentication>
162. <https://www.baeldung.com/spring-security-expressions>
163. <https://www.geeksforgeeks.org/java/spring-security-role-based-and-permission-based-access-control/>
164. <http://ducmanhphan.github.io/2019-02-20-Problem-about-role-name-in-Spring-Security/>
165. <https://stackoverflow.com/questions/38134121/how-do-i-remove-the-role-prefix-from-spring-security-with-javacfg>
166. <https://www.geeksforgeeks.org/advance-java/how-to-resolve-spring-security-hasrole-not-working/>
167. <https://docs.spring.io/spring-security/reference/servlet/authorization/architecture.html>
168. [https://www.baeldung.com/spring-security-remove-role\\_prefix](https://www.baeldung.com/spring-security-remove-role_prefix)
169. <https://docs.spring.io/spring-security/reference/api/java/org/springframework/security/web/authentication/www/BasicAuthenticationFilter.html>
170. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/basic.html>
171. <https://docs.spring.io/spring-security/site/docs/3.1.x/reference/basic.html>
172. <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/dao-authentication-provider.html>
173. <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/authentication/www/BasicAuthenticationEntryPoint.html>
174. <https://www.codecentric.de/en/knowledge-hub/blog/spring-security-two-security-realms-in-one-application>
175. <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/util/matcher/AntPathRequestMatcher.html>
176. <https://docs.spring.io/spring-security/reference/api/java/org/springframework/security/web/util/matcher/AntPathRequestMatcher.html>
177. <https://www.geeksforgeeks.org/advance-java/spring-security-secluding-endpoints-using-antmatchers/>
178. <https://www.securecodewarrior.com/article/a-look-into-the-mvcrequestmatcher-spring-vulnerability>
179. <https://stackoverflow.com/questions/63578908/antpathrequestmatcher-is-not-matching-string-with-wildcards>
180. <https://docs.spring.io/spring-security/site/docs/3.2.0.RELEASE/apidocs/org/springframework/security/web/util/AntPathRequestMatcher.html>
181. <https://docs.spring.io/spring-security/reference/servlet/architecture.html>
182. <https://www.geeksforgeeks.org/advance-java/difference-between-hasrole-and-hasauthority-in-spring-security/>
183. <https://www.baeldung.com/spring-security-basic-authentication>
184. <https://stackoverflow.com/questions/44435295/authorize-request-with-wildcard-preceding-url-in-spring-web-security>

185. <https://www.marcobehler.com/guides/spring-security>
186. <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/AntPathMatcher.html>
187. <https://stackoverflow.com/questions/65708171/how-to-create-a-custom-authentication-filter-in-spring-security>
188. <https://www.baeldung.com/spring-5-mvc-url-matching>
189. <https://www.baeldung.com/spring-security-method-security>
190. <https://www.youtube.com/watch?v=CTdPnaWmSuY>
191. <https://stackoverflow.com/questions/26964590/spring-security-mapping-for-wildcards>
192. <https://jstobigdata.com/spring-security/the-ultimate-guide-to-spring-security-basic-authentication/>
193. <https://wisdom.gitbook.io/gyan/security/complete-guide-to-url-matchers-in-spring-security-types-examples-pros-cons-and-best-use-cases>
194. [https://www.reddit.com/r/springsource/comments/15mf1sy/antmatcher\\_vs\\_requestmatchers/](https://www.reddit.com/r/springsource/comments/15mf1sy/antmatcher_vs_requestmatchers/)
195. <https://spring.io/blog/2020/06/30/url-matching-with-pathpattern-in-spring-mvc>
196. <https://stackoverflow.com/questions/70145976/what-is-the-difference-between-pathpattern-and-antpathmatcher-in-spring-mvc>
197. <https://www.baeldung.com/spring-security-expressions>
198. <https://spring.io/blog/2013/07/03/spring-security-java-config-preview-web-security>
199. <https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-servlet/handlermapping-path.html>
200. <https://stackoverflow.com/questions/33409105/what-is-the-simplest-way-to-override-basicauthentication-onentrypoint-in-springsecu>
201. <https://www.baeldung.com/spring-exclude-filter>
202. <https://docs.spring.io/spring-security/reference/servlet/configuration/java.html>
203. [https://www.reddit.com/r/springsource/comments/113k4z2/spring\\_security\\_securitymatcher\\_vs\\_request\\_matcher/](https://www.reddit.com/r/springsource/comments/113k4z2/spring_security_securitymatcher_vs_request_matcher/)
204. <https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html>
205. <https://docs.spring.io/spring-security/reference/servlet/architecture.html>
206. <https://stackoverflow.com/questions/74683225/updating-to-spring-security-6-0-replacing-removed-and-deprecated-functionality>
207. <https://www.codejava.net/frameworks/spring-boot/spring-security-fix-deprecated-methods>
208. <https://howtodoinjava.com/spring-security/configurations-without-websecurityconfigureradapter/>
209. <https://www.baeldung.com/spring-security-migrate-5-to-6>
210. [https://jcs.ep.jhu.edu/ejava-springboot/coursedocs/content/html\\_single/authn-notes.html](https://jcs.ep.jhu.edu/ejava-springboot/coursedocs/content/html_single/authn-notes.html)
211. <https://github.com/spring-projects/spring-security/issues/12950>
212. <https://stackoverflow.com/questions/52029258/understanding-requestmatchers-on-spring-security>
213. <https://bootify.io/spring-security/spring-security-multiple-configurations.html>
214. <https://vaadin.com/forum/t/antmatchers-deprecated-in-24-8-0/178118>
215. <https://stackoverflow.com/questions/71620692/difference-between-requestmatchers-mvcmatchers-and-mvcmatcher>
216. <https://docs.openrewrite.org/recipes/java/spring/security5/usenewsecuritymatchers>

217. <https://www.youtube.com/watch?v=KuzdIOlugJ4>
218. <https://docs.spring.io/spring-security/site/docs/5.8.0/api/deprecated-list.html>
219. <https://www.geeksforgeeks.org/advance-java/spring-security-securign-endpoints-using-antmatchers/>
220. <https://www.geeksforgeeks.org/java/spring-security-filter-chain-with-example/>
221. <https://docs.openrewrite.org/recipes/java/spring/security5/usenewrequestmatchers>
222. <https://docs.spring.io/spring-security/reference/servlet/test/method.html>
223. <https://www.javaadvent.com/2023/12/mastering-spring-security-integration-testing-for-your-apps.html>
224. <https://www.youtube.com/watch?v=qteZNSDksyA>
225. <https://docs.spring.io/spring-security/site/docs/5.2.0.RELEASE/reference/html/test.html>
226. <https://spring.io/blog/2014/05/07/preview-spring-security-test-method-security>
227. <https://www.baeldung.com/spring-security-method-security>
228. <https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html>
229. <https://developer.okta.com/blog/2021/05/19/spring-security-testing>
230. <https://stackoverflow.com/questions/44467133/spring-boot-authentication-for-integration-tests>
231. <https://docs.spring.io/spring-boot/reference/actuator/auditing.html>
232. <https://dzone.com/articles/application-monitoring-with-spring-boot>
233. <https://www.jvt.me/posts/2022/01/13/spring-boot-actuator-audit/>
234. <https://docs.spring.io/spring-boot/docs/1.4.0.RC1/reference/html/production-ready-auditing.html>
235. <https://www.javacodegeeks.com/2017/03/spring-boot-security-events-actuator.html>
236. <https://www.tencentcloud.com/techpedia/102650>
237. <https://docs.spring.io/spring-boot/reference/actuator/metrics.html>
238. <https://faun.pub/java-23-springboot-3-3-4-metrics-micrometer-prometheus-actuator-part-5-f67f0581815c>
239. <https://docs.spring.io/spring-boot/api/rest/actuator/auditevents.html>
240. <https://docs.spring.io/spring-boot/reference/actuator/endpoints.html>
241. <https://www.baeldung.com/spring-boot-actuators>
242. <https://www.baeldung.com/spring-security-integration-tests>
243. <https://last9.io/blog/a-guide-to-spring-boot-logging/>
244. <https://stackoverflow.com/questions/54187300/withuserdetails-and-withmockuser-when-using-springs-security-in-mockmvc-with-p>
245. <https://docs.spring.io/spring-security/site/docs/5.1.6.RELEASE/reference/html/test.html>
246. <https://spring.io/projects/spring-security>
247. <https://docs.spring.io/spring-security/reference/servlet/test/mockmvc/authentication.html>
248. <https://www.marcobehler.com/guides/spring-security>
249. <https://www.springcloud.io/post/2022-02/spring-security-unit-testing/>
250. <https://docs.spring.io/spring-security/reference/servlet/test/index.html>
251. <https://docs.spring.io/spring-framework/reference/testing/testcontext-framework/tel-config.html>
252. <https://docs.spring.io/spring-security/site/docs/5.2.x/reference/html/test.html>
253. <https://stackoverflow.com/questions/53067071/how-does-customauditeventrepository-work>

- 254. <https://stackoverflow.com/questions/360520/unit-testing-with-spring-security>
- 255. <https://www.wiz.io/blog/spring-boot-actuator-misconfigurations>
- 256. <https://dev.to/wn/monitoring-your-spring-boot-application-with-custom-metrics-and-prometheus-36h9>
- 257. <https://www.baeldung.com/spring-boot-authentication-audit>
- 258. <https://www.geeksforgeeks.org/websites-apps/oauth-vs-openid-connect/>
- 259. <https://supertokens.com/blog/openid-connect-vs-oauth2>
- 260. <https://developer.okta.com/docs/concepts/oauth openid/>
- 261. <https://auth0.com/blog/id-token-access-token-what-is-the-difference/>
- 262. <https://auth0.com/docs/secure/tokens/id-tokens/id-token-structure>
- 263. <https://connect2id.com/learn/openid-connect>
- 264. <https://curity.io/resources/learn/openid-code-flow/>
- 265. [https://openid.net/specs/openid-connect-core-1\\_0.html](https://openid.net/specs/openid-connect-core-1_0.html)
- 266. <https://www.ory.sh/docs/oauth2-oidc/authorization-code-flow>
- 267. <https://www.oauth.com/playground/oidc.html>
- 268. <https://www.miniorange.com/blog/auth-flow-with-pkce/>
- 269. <https://developers.onelogin.com/openid-connect/guides/auth-flow-pkce>
- 270. <https://curity.io/resources/learn/oauth-pkce/>
- 271. <https://xebia.com/blog/get-rid-of-client-secrets-with-oauth-authorization-code-pkce-flow/>
- 272. <https://curity.io/resources/learn/oidc-spring-boot/>
- 273. <https://docs.spring.io/spring-security/reference/servlet/oauth2/login/advanced.html>
- 274. <https://developer.okta.com/docs/api/openapi/okta-oauth/guides/overview/>
- 275. <https://curity.io/resources/learn/validating-an-id-token/>
- 276. <https://www.youtube.com/watch?v=IKXwxDM9pOk>
- 277. <https://www.baeldung.com/spring-security-openid-connect>
- 278. <https://stackoverflow.com/questions/1087031/whats-the-difference-between-openid-and-oauth>
- 279. <https://spring.io/guides/tutorials/spring-boot-oauth2>
- 280. <https://konghq.com/blog/engineering/openid-vs-oauth-what-is-the-difference>
- 281. <https://serengetitech.com/tech/simplifying-oauth-2-0-and-openid-connect-implementation-with-spring-authorization-server/>
- 282. <https://auth0.com/docs/authenticate/login/oidc-conformant-authentication/oidc-adoption-auth-code-flow>
- 283. <https://learn.microsoft.com/en-us/entra/identity-platform/v2-protocols>
- 284. <https://openclassrooms.com/en/courses/5683681-secure-your-web-application-with-spring-security/6695831-configure-oauth-2.0-with-openid-connect-on-a-spring-web-application>
- 285. <https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-auth-code-flow>
- 286. <https://www.okta.com/identity-101/whats-the-difference-between-oauth-openid-connect-and-saml/>
- 287. <https://www.cripto.com/blog/jwt-claims-example>
- 288. <https://stackoverflow.com/questions/57222738/oppnenid-connect-configuration-over-spring-security-oauth2>

289. <https://docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html>
290. <https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce>
291. <https://learn.microsoft.com/en-us/entra/identity-platform/id-token-claims-reference>
292. <https://dzone.com/articles/spring-oauth2-resource-servers>
293. <https://www.bezkoder.com/spring-boot-refresh-token-jwt/>
294. <https://stateful.com/blog/oauth-refresh-token-best-practices>
295. <https://frontegg.com/blog/oauth-2-refresh-tokens>
296. <https://auth0.com/docs/secure/tokens/refresh-tokens/refresh-token-rotation>
297. <https://www.descope.com/blog/post/refresh-token-rotation>
298. <https://auth0.com/docs/secure/tokens/token-best-practices>
299. <https://www.bezkoder.com/spring-security-refresh-token/>
300. <https://stackoverflow.com/questions/73733680/setup-spring-webclient-filter-from-an-oauth2-refresh-token>
301. <https://www.javainuse.com/webseries/spring-security-jwt/chap7>
302. <https://www.youtube.com/watch?v=EsVybSJr7zU>
303. <https://www.javacodegeeks.com/2024/12/managing-jwt-refresh-tokens-in-spring-security-a-complete-guide.html>
304. <https://developer.okta.com/docs/guides/refresh-tokens/main/>
305. <https://www.baeldung.com/spring-security-oauth2-refresh-token-angular>
306. [https://www.youtube.com/watch?v=Wp4h\\_wYXqmU](https://www.youtube.com/watch?v=Wp4h_wYXqmU)
307. <https://spring.io/guides/tutorials/spring-boot-oauth2>
308. <https://www.codingshuttle.com/spring-boot-handbook/jwt-refresh-token-and-access-token>
309. <https://blog.stackademic.com/jwt-authentication-in-spring-security-0061d82ddf12>
310. <https://www.vincenzoracca.com/en/blog/framework/spring/jwt/>
311. <https://www.serverion.com/uncategorized/refresh-token-rotation-best-practices-for-developers/>
312. <https://github.com/bezkoder/spring-security-refresh-token-jwt>
313. <https://documentation.cloud-iam.com/resources/passwordless-webauthn.html>
314. <https://www.linkedin.com/pulse/passwordless-authentication-spring-security-passkeys-rodrigues-yloye>
315. <https://github.com/deepu105/webauthn4j-spring-boot-passkeys-demo>
316. <https://docs.spring.io/spring-security/reference/servlet/authentication/passkeys.html>
317. <https://www.geeksforgeeks.org/advance-java/spring-security-two-factor-authentication/>
318. <https://www.baeldung.com/spring-security-two-factor-authentication-with-soft-token>
319. <https://blog.stackademic.com/step-by-step-ex-implementing-mfa-with-spring-boot-and-totp-1435588e49ca>
320. <https://docs.spring.io/spring-authorization-server/reference/guides/how-to-dynamic-client-registration.html>
321. <https://www.scalekit.com/blog/dynamic-client-registration-oauth2>
322. <https://www.couchbase.com/blog/oauth-2-dynamic-client-registration/>

323. <https://stackoverflow.com/questions/4990813/does-spring-security-support-multi-factor-authentication>
324. <https://webauthn4j.github.io/webauthn4j-spring-security/en/>
325. <https://docs.spring.io/spring-security/reference/index.html>
326. <https://developer.okta.com/blog/2021/05/05/client-credentials-spring-security>
327. <https://github.com/making/demo-two-factor-authentication>
328. <https://spring.io/guides/tutorials/spring-boot-oauth2>
329. [https://www.youtube.com/watch?v=qNy\\_Q9fth-4](https://www.youtube.com/watch?v=qNy_Q9fth-4)
330. <https://howtodoinjava.com/spring-security/2fa-auth-with-jwt-token/>
331. <https://stackoverflow.com/questions/63771864/is-there-any-suggestion-for-implementing-dynamic-client-registration-using-oauth>
332. <https://auth0.com/blog/spring-security-overview/>