Spring Security

Complete Reference Guide

From Fundamentals to Production-Ready Implementation

Created: February 12, 2026

■ What You'll Learn

✓ Core Concepts: Servlets, Filters, and Spring MVC Architecture

✓ Filter Chain and OncePerRequestFilter Patterns

✓ Spring Security Filter Chain (15+ Internal Filters)

✓ Authentication Flow: From Login to SecurityContext

✓ JWT Token Authentication (Complete Implementation)

✓ Roles vs Authorities (When to Use Each)

✓ Security Configuration (HttpSecurity Deep Dive)

✓ CSRF Protection (What, Why, When)

✓ Password Encoding with BCrypt

✓ Production-Ready Patterns and Best Practices

Chapter 1: Servlets & Filters Foundation

1.1 Understanding Servlets

Traditional Java web applications used multiple servlets (one per endpoint). Spring revolutionized this by using ONE servlet (DispatcherServlet) that routes requests to @Controller methods via HandlerMapping.

Request Flow:

```
Browser → Filters → DispatcherServlet → HandlerMapping → Controller →
Response
```

Servlet Lifecycle:

Method	When Called	Purpose
init()	Once on startup	Initialize servlet
service() / doGet() / doPost()	Per request	Handle request
destroy()	On shutdown	Cleanup resources

1.2 Filter Chain Mechanics

Filters intercept requests BEFORE they reach servlets. They execute in order going in, and reverse order going out (like Russian nesting dolls).

Filter Interface Methods:

```
void init(FilterConfig config) // Called once on startup void
doFilter(request, response, chain) // Called per request void destroy() //
Called on shutdown
```

■■ Key Concept: Filter Execution Order

Filters execute in order based on @Order annotation. Lower number = higher priority (runs first). Spring Security defaults to order -100.

1.3 OncePerRequestFilter Pattern

Standard Filter can execute multiple times per request (during forwards/includes). OncePerRequestFilter guarantees single execution using request attributes.

When to Use:

Filter Type	Use Case	Execution Guarantee
Filter	Simple filtering	May execute multiple times

OncePerRequestFilter	Authentication, logging	Exactly once per request
GenericFilterBean	Spring-aware filters	Once (with Spring context)

Chapter 2: Spring Security Core Components

2.1 SecurityContextHolder

Thread-local storage holding current user's authentication. Each HTTP request thread has an isolated SecurityContext.

Key Methods:

```
SecurityContextHolder.getContext() // Get current context
SecurityContextHolder.setContext(context) // Set authentication
SecurityContextHolder.clearContext() // Logout
```

2.2 Authentication Interface

Represents an authenticated user with credentials and authorities.

Method	Returns	Purpose
getPrincipal()	Object	User details (UserDetails or username)
getCredentials()	Object	Password (cleared after auth)
getAuthorities()	Collection	Roles/permissions
isAuthenticated()	boolean	Authentication status

2.3 UserDetails & UserDetailsService

UserDetails represents a user loaded from your database. UserDetailsService is the bridge between Spring Security and your user storage.

UserDetails Methods:

```
getUsername() // Username getPassword() // Encrypted password
getAuthorities() // Roles/permissions isAccountNonExpired() // Account status
isAccountNonLocked() // Account status isCredentialsNonExpired() // Password
status isEnabled() // Account status
```

2.4 AuthenticationManager & Provider

AuthenticationManager is the main entry point for authentication. It delegates to AuthenticationProvider implementations (like DaoAuthenticationProvider).

DaoAuthenticationProvider Flow:

```
1. Load user via UserDetailsService 2. Check account status (enabled, locked,
expired) 3. Verify password with PasswordEncoder 4. Create authenticated
token 5. Return to AuthenticationManager
```

Chapter 3: Complete Authentication Flow

3.1 Login Request Trace

Understanding the complete flow from login request to authenticated user.

Step-by-Step Flow:

```
1. POST /api/auth/login {username, password} 2. Controller creates
unauthenticated UsernamePasswordAuthenticationToken 3. Passes to
AuthenticationManager.authenticate() 4. ProviderManager finds
DaoAuthenticationProvider 5. Provider loads user from database
(UserDetailsService) 6. Checks account status (enabled, non-locked,
non-expired) 7. Verifies password: passwordEncoder.matches(submitted, stored)
8. Creates authenticated token (credentials cleared, authorities added) 9.
Returns to controller 10. Controller sets in SecurityContextHolder 11.
Generates JWT token 12. Returns JWT to client
```

3.2 Subsequent Requests with JWT

After login, every request includes JWT in Authorization header.

```
1. Request enters filter chain 2. Spring Security filters execute (order
-100) 3. JwtAuthenticationFilter (before
UsernamePasswordAuthenticationFilter): - Extracts JWT from Authorization
header - Validates signature and expiration - Extracts username from token -
Loads UserDetails from database - Creates Authentication object - Sets in
SecurityContextHolder 4. FilterSecurityInterceptor checks authorization 5.
Request reaches controller
```

■■ Critical Point: Setting SecurityContext

The JWT filter MUST set Authentication in SecurityContextHolder. Without this step, Spring Security thinks the user is not authenticated, even with a valid token.

Chapter 4: JWT Token Authentication

4.1 JWT Token Provider

Service responsible for creating and validating JWT tokens.

Key Responsibilities:

```
✓ generateToken(Authentication) - Create JWT with claims ✓
validateToken(String) - Verify signature and expiration ✓
getUsernameFromToken(String) - Extract username from claims ✓
getExpirationDateFromToken(String) - Check token expiry ✓
isTokenExpired(String) - Validate token is still valid
```

4.2 JWT Authentication Filter

Custom filter that extracts and validates JWT tokens on every request.

Filter Implementation Steps:

```
1. Extract JWT from Authorization header (Bearer token) 2. Validate token
signature and expiration 3. Extract username from token claims 4. Load
UserDetails from database 5. Create UsernamePasswordAuthenticationToken 6.
Set in SecurityContextHolder 7. Continue filter chain
```

■■ Important: Filter Registration

JWT filter uses addFilterBefore(), NOT @Order annotation. This ensures it runs before UsernamePasswordAuthenticationFilter in the Spring Security filter chain.

4.3 Token Structure

JWT consists of three parts: Header.Payload.Signature

Part	Contains	Example
Header	Algorithm & Type	{"alg": "HS512", "typ": "JWT"}
Payload	Claims (username, roles, expiry)	{"sub": "suraj", "exp": 1234567890}
Signature	Encrypted hash for verification	Computed using secret key

Chapter 5: Roles vs Authorities

5.1 Understanding the Difference

Roles are high-level groupings (ROLE_ADMIN, ROLE_USER). Authorities are granular permissions (READ_POST, WRITE_POST, DELETE_POST).

Conceptual Mapping:

```
ROLE_USER: - READ_PRIVILEGE - WRITE_OWN_PROFILE ROLE_MODERATOR: -
READ_PRIVILEGE - WRITE_PRIVILEGE - DELETE_COMMENTS - BAN_USERS ROLE_ADMIN: -
READ_PRIVILEGE - WRITE_PRIVILEGE - DELETE_PRIVILEGE - MANAGE_USERS -
VIEW_AUDIT_LOGS - CONFIGURE_SYSTEM
```

5.2 The ROLE_ Prefix Convention

Spring Security requires roles to have 'ROLE_' prefix when stored, but NOT when checking.

Storage vs Checking:

```
// Storage (in database/enum/authority) "ROLE_ADMIN", "ROLE_USER" ← With
ROLE_ prefix // Checking (in hasRole()) hasRole("ADMIN") ← Without prefix
(Spring adds it automatically) hasRole("USER") ← Without prefix //
Alternative: hasAuthority() (exact match) hasAuthority("ROLE_ADMIN") ← Must
include prefix hasAuthority("READ_PRIVILEGE") ← No prefix for authorities
```

5.3 When to Separate Roles and Authorities

Approach	Best For	Flexibility	Complexity
Single Enum	Small apps, Learning	Low	Low
Enum + Mapping	Medium apps	Medium	Medium
Database Tables	Enterprise, SaaS	Very High	High

■ **Recommendation for Learning:**

Start with single enum (simple). Move to enum + mapping when you need more control. Use database tables only for production enterprise applications.

Chapter 6: Security Configuration Deep Dive

6.1 SecurityFilterChain Bean

The heart of Spring Security configuration. This bean defines all security rules.

Essential Configuration Methods:

Method	Purpose	Use Case
authorizeHttpRequests()	URL-based authorization	Define which URLs need what roles
formLogin()	Form-based login	Traditional web apps
httpBasic()	HTTP Basic auth	REST APIs (simple)
logout()	Logout configuration	Session cleanup
sessionManagement()	Session handling	Stateless vs stateful
csrf()	CSRF protection	Enable/disable based on app type
cors()	CORS configuration	Cross-origin requests
addFilterBefore()	Custom filters	JWT authentication

6.2 Authorization Rules

Define which users can access which endpoints using roles or authorities.

■■ Critical: Order Matters!

Spring Security processes rules from TOP to BOTTOM. First match wins! Always put specific rules before general rules, and anyRequest() LAST.

6.3 Session Management Policies

Policy	Behavior	Best For
STATELESS	Never create sessions	JWT/token-based APIs
IF_REQUIRED	Create if needed (default)	Traditional web apps
ALWAYS	Always create session	Session-heavy apps
NEVER	Don't create, use existing	Rare use cases

Chapter 7: CSRF Protection Explained

7.1 What is CSRF?

Cross-Site Request Forgery (CSRF) is an attack where a malicious website tricks your browser into making requests to another site where you're logged in.

Attack Scenario:

```
1. You login to bank.com (session cookie stored) 2. You visit evil.com (while
still logged in to bank.com) 3. evil.com contains hidden form that
auto-submits to bank.com/transfer 4. Browser AUTOMATICALLY includes bank.com
cookies with request 5. Bank receives request with valid session cookie 6.
Bank thinks: "Valid session, must be legitimate!" 7. Bank processes transfer
→ Money gone!
```

7.2 Why Browsers Send Cookies Automatically

This is a BROWSER FEATURE, not a bug. When you make ANY request to a domain, the browser includes ALL cookies for that domain, REGARDLESS of where the request originates.

7.3 CSRF Protection Mechanism

Server generates a SECRET TOKEN for each session. This token must be included in state-changing requests (POST/PUT/DELETE). Attackers cannot access this token due to Same-Origin Policy.

How It Works:

```
1. Server generates: CSRF_TOKEN = "random-unique-token-12345" 2. Server sends
in cookie: Set-Cookie: XSRF-TOKEN=random-unique-token-12345 3. Client
includes in request: - Cookie: XSRF-TOKEN=random-unique-token-12345
(automatic) - X-XSRF-TOKEN: random-unique-token-12345 (manual header) 4.
Server validates: if (cookieToken == headerToken) { allow } else { reject }
```

7.4 When to Enable/Disable CSRF

Authentication Type	CSRF Protection	Reason
Cookie/Session-based	ENABLE	Browsers auto-send cookies
JWT in Authorization header	DISABLE	Browsers don't auto-send headers
Form-based login	ENABLE	Traditional web apps need it
Stateless REST API	DISABLE	No cookies, no CSRF risk

■ **Key Insight: Why JWT Doesn't Need CSRF**

JWT tokens are stored in localStorage or memory, NOT cookies. Browsers don't automatically add Authorization headers to cross-origin requests. Attackers have NO WAY to access or include the JWT token.

Chapter 8: Password Encoding with BCrypt

8.1 Why BCrypt?

BCrypt is the industry standard for password hashing. It's adaptive (slow by design), automatically salted, and one-way (cannot be decoded).

Feature	Benefit
One-way hash	Cannot decode to get original password
Automatic salting	Same password = different hashes
Adaptive cost factor	Can increase difficulty as computers get faster
Slow by design (~300ms)	Prevents brute force attacks
Industry standard	Battle-tested and trusted

8.2 Cost Factor

Cost factor determines how slow the algorithm is. Higher = more secure but slower. Production recommendation: 10-12.

```
BCryptPasswordEncoder(10) // Fast, minimum production
BCryptPasswordEncoder(12) // Recommended for production
BCryptPasswordEncoder(14) // Very secure, slower
```

8.3 Usage Pattern

```
// Encoding (during registration) String rawPassword = "mypassword123";
String encoded = passwordEncoder.encode(rawPassword); // Result:
$2a$12$xYz... (60 characters) // Verification (during login) boolean matches
= passwordEncoder.matches(rawPassword, encodedPassword); // Returns: true if
passwords match
```

Chapter 9: Production-Ready Patterns

9.1 Complete JWT Configuration

A production-ready JWT authentication system requires several components working together.

Required Components:

```
✓ JwtTokenProvider - Generate and validate tokens ✓ JwtAuthenticationFilter
- Extract and authenticate ✓ CustomUserDetailsService - Load users from
database ✓ SecurityConfig - Wire everything together ✓ AuthController -
Login/register endpoints ✓ User & Role entities - Database models ✓
PasswordEncoder bean - BCrypt encoder ✓ AuthenticationEntryPoint - Handle
auth errors
```

9.2 Security Best Practices

1. Use HTTPS in production (prevent token interception)

2. Store JWT secret in environment variables (never hardcode)

3. Set reasonable token expiration (24 hours for web, 7 days for mobile)

4. Implement token refresh mechanism for better UX

5. Use BCrypt with cost factor 12 for passwords

6. Validate input to prevent injection attacks

7. Implement rate limiting to prevent brute force

8. Log authentication attempts for security monitoring

9. Use @PreAuthorize for method-level security

10. Implement proper exception handling for auth errors

9.3 Common Mistakes to Avoid

■ Forgetting to set Authentication in SecurityContextHolder

■ Using @Order for JWT filter (use addFilterBefore instead)

■ Returning empty list from getAuthorities()

■ Not using EAGER fetch for roles (causes LazyInitializationException)

■ Enabling CSRF for stateless JWT APIs

■ Storing passwords in plain text (always use BCrypt)

■ Not handling token expiration gracefully

■ Putting anyRequest().authenticated() before specific rules

■ Using weak JWT secrets (minimum 256 bits)

■ Not implementing proper logout (token revocation)

Chapter 10: Quick Reference Guide

10.1 Filter Order Cheat Sheet

Filter Priority	Order Value	Use Case
Infrastructure	-200 to -151	Request ID, tenant resolution
Pre-Authentication	-150 to -101	API key validation
Spring Security	-100	Built-in security filters
Post-Authentication	-99 to -1	Audit logging
Business Logic	1 to 100	Application-specific filters

10.2 Authorization Methods Comparison

Method	Checks For	Example
hasRole()	'ROLE_' + name	hasRole('ADMIN') → ROLE_ADMIN
hasAnyRole()	Any of roles	hasAnyRole('ADMIN', 'USER')
hasAuthority()	Exact match	hasAuthority('READ_PRIVILEGE')
hasAnyAuthority()	Any of authorities	hasAnyAuthority('READ', 'WRITE')
access()	Custom logic	Custom authorization decisions

10.3 Key Takeaways

✓ Spring uses ONE servlet (DispatcherServlet) routing to multiple controllers

✓ Filters run before servlet, in order going in, reverse order going out

✓ OncePerRequestFilter prevents duplicate execution during forwards

✓ JWT filters use addFilterBefore(), not @Order annotation

✓ SecurityContextHolder is thread-local, isolated per request

✓ Authentication flow: unauthenticated token → AuthenticationManager → provider → database → password check → authenticated token

✓ JWT pattern: Extract token → validate → load user → set SecurityContext

✓ Spring Security = 15+ filters in FilterChainProxy at order -100

✓ BCrypt for password hashing (one-way, salted, adaptive)

✓ CSRF needed for cookies, NOT needed for JWT in headers

✓ Roles have ROLE_ prefix when stored, not when checking with hasRole()

✓ Session STATELESS for JWT APIs, IF_REQUIRED for traditional web apps

■ Learning Path Complete!

You now understand Spring Security from fundamentals to production-ready implementation. This knowledge forms the foundation for building secure, enterprise-grade applications.

Next Steps:

→ Build a complete authentication system with JWT

→ Implement role-based access control (RBAC)

→ Add OAuth2/OIDC integration for social login

→ Implement method-level security with @PreAuthorize

→ Add refresh token mechanism for better UX

→ Explore Spring Security's OAuth2 Resource Server

→ Learn about Multi-factor Authentication (MFA)

→ Study security best practices and OWASP Top 10

Keep this guide handy as a reference. Security is a journey, not a destination. Stay curious, keep learning, and build secure applications!

— Happy Coding! ■