# **Spring Security**

Spring Security is the standard framework for securing Spring applications, offering robust **authentication**, **authorization**, and protection against common vulnerabilities. Originally developed as *Acegi Security* (2003), it was later rebranded as **Spring Security** (2008) and has since become a highly **extensible** and **convention-over-configuration** solution, especially with **Spring Boot** integration.

**Key Features & Architecture**

* Uses a **filter chain** to intercept and secure requests.
* Relies on **AuthenticationManager** (for identity verification) and **AuthorizationManager** (for access control).
* Supports **role-based security** (e.g., ROLE\_ADMIN) via annotations (@Secured, @PreAuthorize) and URL-based rules.

**Common Use Cases**

* **Web & REST API security** (form logins, stateless JWT authentication).
* **OAuth2/OpenID Connect** for third-party logins (e.g., Google, GitHub).
* **Enterprise Single Sign-On (SSO)** and **microservices security**.
* **Role-based access control (RBAC)** in applications like e-commerce platforms.

**Why Use It?**

* Seamless integration with Spring ecosystem.
* Protection against exploits (CSRF, session fixation).
* Flexibility for simple to complex security requirements.

**Concept & Origins**

**What Is Spring Security?**

Spring Security is a Java framework that provides authentication, authorization, and protection against common web exploits such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). It integrates tightly with both imperative and reactive Spring applications, making it the standard choice for securing Spring-based projects

**History & Evolution**

Originally released as the “Acegi Security System for Spring” in late 2003, Spring Security was born from a community request for a Spring-native security solution Acegi became an official Spring sub-project and was renamed Spring Security in April 2008, with the first 2.0.0 release under that name

**Core Rules & Conventions**

1. **Filter Chain Configuration**  
   All incoming requests pass through a SecurityFilterChain, which applies authentication and authorization logic in a pre-defined order
2. **Enable Security via Annotations**  
   Use @EnableWebSecurity or @SpringBootApplication (which meta-includes security) alongside a SecurityFilterChain bean to bootstrap security
3. **Role & Authority Naming**  
   Authorities are represented as GrantedAuthority instances, with roles following the ROLE\_\* prefix convention (e.g., ROLE\_USER, ROLE\_ADMIN) for clarity in access rules
4. **Method-Level Security**  
   Enable with @EnableGlobalMethodSecurity (or its newer equivalents) and annotate services/controllers with @PreAuthorize, @PostAuthorize, or @Secured to apply granular access rules
5. **CSRF & Session Management**  
   CSRF protection is enabled by default for stateful applications; for stateless REST APIs, CSRF is typically disabled and replaced with token-based mechanisms (e.g., JWT)

**Where & When to Use Spring Security**

* **Web Applications:** Protect form-based logins, logout endpoints, and apply URL-based access rules via .antMatchers() or request-matcher DSL
* **RESTful APIs:** Secure endpoints with stateless JWT authentication, disabling CSRF and managing sessions manually if needed
* **Method-Invocation Protection:** Apply method-level security in service layers to enforce business-level authorization regardless of transport (HTTP, messaging, etc.)
* **Microservices & SSO:** Centralize authentication with OAuth2/OpenID Connect, using Spring Security OAuth2 client and resource server modules for SSO across services

**Why Spring Security Came to Be**

As Java EE applications grew in complexity, developers faced fragmented security solutions and verbose XML configurations. The Spring community sought a unified, convention-driven approach that would integrate natively with Spring’s dependency-injection model. Acegi Security addressed this by offering a filter-based, extensible framework, which Spring Security has since refined to cover evolving security needs (OAuth2, reactive stacks, JWT, etc.)

## **key security capabilities and features Spring Boot (via Spring Security) provides:**

**🔐 Authentication Capabilities**

* Username/Password Authentication: In-memory, JDBC-based, LDAP, or custom user stores.
* JWT/OAuth2/OpenID Connect:
  + Support for stateless authentication with JWT tokens.
  + First-class integration with OAuth2 authorization servers (Google, GitHub, Keycloak, Okta, etc.).
  + Built-in OpenID Connect support (single sign-on, identity federation).
* SAML2: For enterprise SSO integration.
* Custom Authentication Providers: Implement your own auth logic (e.g., API keys, multi-factor authentication).

**🛡 Authorization Features**

* Role-Based Access Control (RBAC): Fine-grained permissions with @PreAuthorize, @Secured, or method-level security.
* Attribute-Based Access Control (ABAC): Write complex SpEL expressions to enforce policies dynamically.
* URL & Endpoint Security: Define access rules for endpoints via HttpSecurity.
* Method-Level Security: Secure individual service methods with annotations.
* Domain Object Security: Apply ACLs (Access Control Lists) for per-resource permissions.

**🔄 Session & State Management**

* Session Management: Control session creation, concurrency, and invalidation.
* Stateless Security: For REST APIs, use JWT or OAuth2 without maintaining server-side sessions.
* Remember-Me Authentication: Persist login across sessions with secure tokens.
* CSRF Protection: Built-in Cross-Site Request Forgery protection for stateful apps (e.g., forms).

**🌐 Web & API Security**

* CORS Support: Secure cross-origin requests with fine-grained rules.
* CSRF Protection: Enabled by default for form logins, configurable for APIs.
* Filter Chains: Flexible security filters to enforce logging, request validation, and rate limiting.
* API Security: Secure REST APIs with token-based authentication and role checks.

**🧰 Security Filters & Middleware**

* Password Encoding: Strong hashing with BCrypt, PBKDF2, Argon2.
* Request/Response Security Headers (auto-configured):
  + HSTS
  + X-Content-Type-Options
  + X-Frame-Options
  + CSP (Content Security Policy)
* Security Event Handling: Custom handling of login success/failure, access denied, etc.

**🔑 Identity & Integration**

* LDAP Integration: For enterprise directory authentication.
* OAuth2 Client Support: Securely call downstream APIs using client credentials or user tokens.
* OAuth2 Resource Server: Protect APIs using JWT or opaque tokens.
* External Identity Providers: Google, Azure AD, Keycloak, Okta, etc.

**🛠 Developer Productivity**

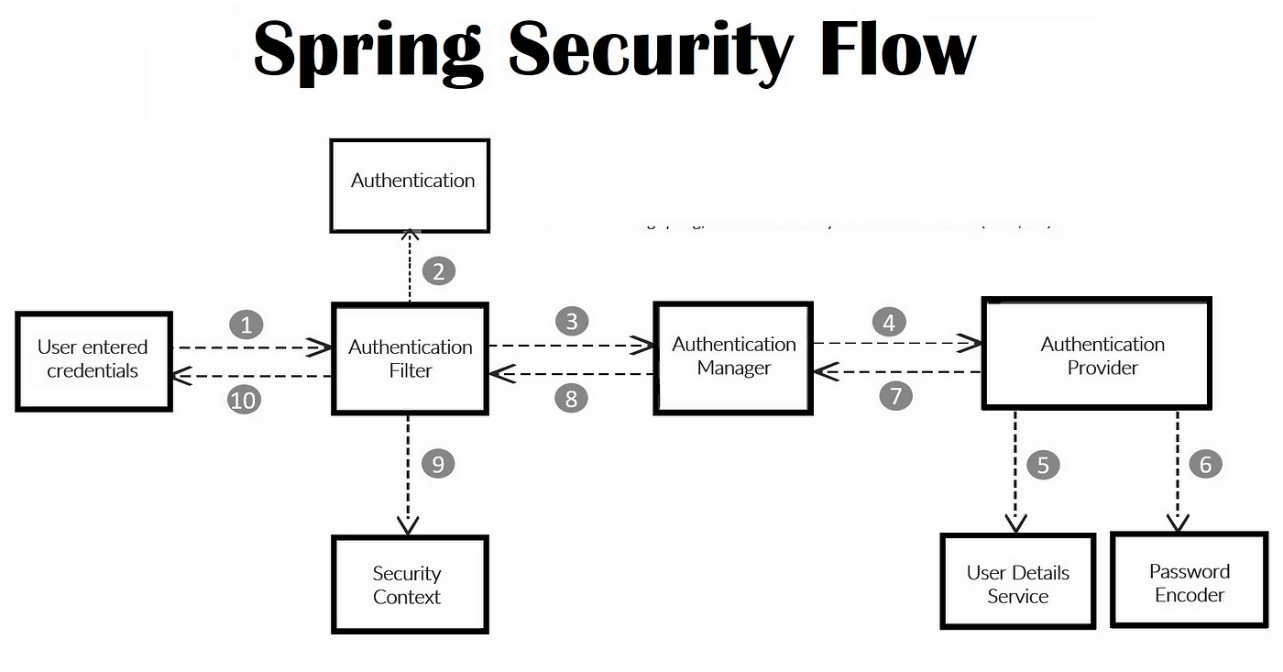
* Auto-Configuration: Secure defaults for web apps and APIs (form login, basic auth, OAuth2).
* Security Test Support: Spring Security Test module to mock users, roles, and tokens in tests.
* Actuator Security: Secure sensitive actuator endpoints with roles/tokens.
* Configuration via Properties: Easy security setup without boilerplate XML.

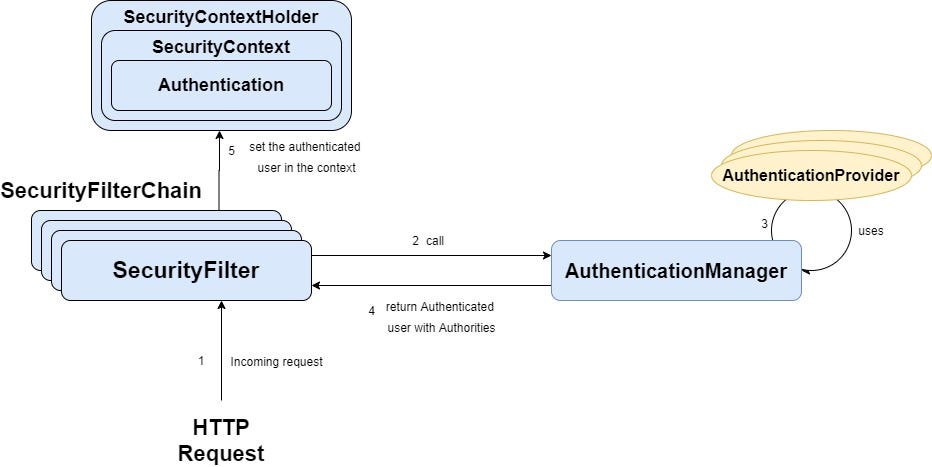
**🛡 Advanced & Enterprise Features**

* Multi-Factor Authentication (MFA): Build custom flows with OTP, SMS, email verification.
* SAML 2.0 Login: For enterprise SSO.
* Reactive Security: Security support for Spring WebFlux applications.
* Fine-Grained Security Auditing: Monitor and log security-related events.

**Real-World Examples**

1. **E-Commerce Site with Role-Based Access**  
   An online store uses Spring Security to allow ROLE\_CUSTOMER to place orders while restricting ROLE\_ADMIN to inventory and order management dashboards.
2. **Secure REST API with JWT**  
   A microservice architecture issues JWTs upon authentication; Spring Security’s stateless filter chain validates tokens on each request, ensuring only valid clients access protected endpoints
3. **Enterprise SSO via OAuth2**  
   A corporate portal employs Spring Security’s OAuth2 client and resource server modules to integrate with an existing identity provider (e.g., Keycloak or Okta), enabling single-sign-on across multiple applications
4. **Method-Level Security in Banking Services**  
   Critical operations such as fund transfers are annotated with @PreAuthorize("hasAuthority('ROLE\_TELLER')"), ensuring only tellers can execute these methods regardless of how they’re invoked
5. **CSRF Protection for Traditional Web Apps**  
   A customer relationship management (CRM) system uses Spring Security’s default CSRF tokens to secure form submissions, preventing cross-site request forgery attacks without additional configuration





**1. Filter Chains**

Spring Security’s SecurityFilterChain intercepts every HTTP request in a defined order.

1. **E-Commerce Checkout Flow**
   * **Requirement:** Only authenticated users can add items to cart; only customers with verified email can proceed to payment.
   * **Implementation:**
     1. **Authentication Filter** checks for a valid session or JWT.
     2. **EmailVerificationFilter** (custom) rejects requests if user’s emailVerified == false.
     3. **Authorization Filter** applies URL-based rules (e.g. "/checkout/\*\*" → hasRole("CUSTOMER")).
   * **Outcome:** Unverified or anonymous visitors are automatically redirected to login or verification.
2. **Microservice Gateway**
   * **Requirement:** All downstream services must only see requests that have passed through centralized authentication and rate-limiting.
   * **Implementation:**
     1. **PreAuthFilter** validates an OAuth2 access token in the Authorization header.
     2. **RateLimitingFilter** tracks per-client request counts.
     3. **RouteFilter** forwards only valid, non-throttled requests to internal services.
   * **Outcome:** Downstream microservices trust that incoming requests are both authenticated and within usage quotas.
3. **API Versioning & Deprecation**
   * **Requirement:** Support v1 and v2 of a public API, deprecating v1 over time.
   * **Implementation:**
     1. **ApiVersionFilter** reads a custom header (X-API-Version) and routes to appropriate controllers.
     2. **DeprecationWarningFilter** adds Warning headers when clients call v1 endpoints.
     3. **DefaultSecurityFilter** enforces authN/authZ after version routing.
   * **Outcome:** Clients automatically receive deprecation notices, and security is uniformly applied post-routing.

**2. Role Naming**

Consistent authority names (e.g., ROLE\_USER, ROLE\_ADMIN) simplify access rules and logging.

1. **Multi-Tenant SaaS Platform**
   * **Roles:**
     + ROLE\_SUPER\_ADMIN (platform owner)
     + ROLE\_TENANT\_ADMIN (per-customer administrator)
     + ROLE\_USER (end-user within a tenant)
   * **Use:** URLs under /admin/\*\* require hasRole("TENANT\_ADMIN"), while /platform/\*\* are locked to hasRole("SUPER\_ADMIN").
2. **Healthcare Management System**
   * **Roles:**
     + ROLE\_DOCTOR
     + ROLE\_NURSE
     + ROLE\_RECEPTIONIST
     + ROLE\_PATIENT
   * **Use:**
     + Doctors and nurses see and modify patient records (hasAnyRole("DOCTOR","NURSE")).
     + Receptionists can schedule appointments (hasRole("RECEPTIONIST")).
     + Patients can only view their own data (hasRole("PATIENT") and principal.username==#patientId).
3. **Banking Application**
   * **Roles:**
     + ROLE\_TELLER
     + ROLE\_MANAGER
     + ROLE\_AUDITOR
   * **Use:**
     + Tellers process transactions (@PreAuthorize("hasRole('TELLER')")).
     + Managers can approve large transfers (hasRole("MANAGER")).
     + Auditors have read-only access to all accounts (hasRole("AUDITOR")).

**3. Annotation-Driven Configuration**

Using annotations to declare security rules at the method or class level, decoupling business logic from security plumbing.

1. **Service-Layer Method Security**

@Service

public class OrderService {

@PreAuthorize("hasRole('CUSTOMER')")

public Order createOrder(Cart cart) { … }

@PreAuthorize("hasRole('ADMIN')")

public void cancelOrder(Long orderId) { … }

}

* + **Outcome:** Even if there’s a web-layer bug, only users with the correct role can invoke these methods.

1. **REST Controller Protection**

@RestController

@RequestMapping("/api/users")

@PreAuthorize("hasRole('ADMIN')")

public class UserAdminController {

@GetMapping

public List<User> listAll() { … }

@PostMapping

public User create(@RequestBody User u) { … }

}

* + **Outcome:** All endpoints in this controller are automatically locked to admins, reducing boilerplate.

1. **Expression-Based Access with SpEL**

@Component

public class DocumentService {

@PostAuthorize("returnObject.owner == authentication.name")

public Document getDocument(Long id) { … }

}

* + **Outcome:** Even after fetching the document, Spring Security verifies the current user matches the document’s owner before returning it.

**JSON Web Token (JWT) Authentication**

**Concept & Origins**

A JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. It consists of three Base64-URL-encoded segments—Header, Payload, and Signature—joined by dots (eyJhbGci….eyJzdWIi….SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c).

* **Header** declares the type (JWT) and signing algorithm (e.g. HS256, RS256).
* **Payload** carries a set of claims (registered like iss, sub, aud, exp; public; or private).
* **Signature** is computed by signing the Base64-URL of Header and Payload with a secret or private key.

JWT emerged around 2010 (RFC 7519, minted in 2015) to standardize stateless, self-contained tokens for web-scale, cross-domain authentication and authorization without server-side session storage.

**Core Rules & Best Practices**

1. **Token Structure & Integrity**
   * Always verify the token’s **Signature** before trusting any claims.
   * Reject tokens whose signature fails verification or whose structure isn’t three segments.
   * Example in practice: an API gateway rejects any incoming Authorization: Bearer … header that fails HMAC validation.
2. **Expiration & Clock Skew**
   * Include an exp (expiration) claim and, optionally, nbf (not-before).
   * Allow a small clock-skew window (e.g., ±60 seconds) when validating times.
   * Example in practice: a mobile backend refuses requests with tokens older than 15 minutes, forcing refresh.
3. **Audience (aud) and Issuer (iss) Checks**
   * Validate that the aud claim matches your service identifier, and iss matches the trusted issuer URL or identifier.
   * Example in practice: microservices verify aud=order-service to ensure tokens weren’t stolen from another API.
4. **Key Management & Rotation**
   * Use asymmetric keys (RSA/ECDSA) in production so you can rotate signing keys without downtime.
   * Publish a JSON Web Key Set (JWKS) endpoint for clients to fetch public keys.
   * Example in practice: an identity provider (IdP) rotates its RSA key pair weekly and updates the JWKS endpoint, while services auto-refresh cached keys.
5. **Minimal Payload & Sensitive Data**
   * Store only non-sensitive claims (user IDs, roles). Never embed passwords or personal info.
   * Keep payloads small to reduce bandwidth and avoid exposing unnecessary data.
   * Example in practice: a single-page application’s token payload includes only { sub: "alice", roles: ["user"] }—no email or profile data.

**Real-World Examples of Rules in Action**

1. **E-Commerce API Gateway**
   * Rejects expired tokens immediately (exp), and enforces that aud equals "shopping-api".
2. **Banking Microservices**
   * Uses RSA-signed tokens. Services fetch the bank’s JWKS every hour, validating JWTs without sharing private keys.
3. **Mobile App Backend**
   * Implements a 5 minute clock-skew allowance when checking nbf to accommodate device clock drift.

**Where & When to Use JWT**

* **Stateless Microservices**: Ideal when services need to authenticate requests without centralized session stores.
* **Single-Page Applications (SPAs)**: Enables the frontend to present a token on each XHR/fetch request.
* **Mobile & IoT Clients**: Compact tokens that can be stored safely on-device and sent over limited networks.
* **Cross-Domain SSO**: Facilitates single-sign-on between different domains or sub-domains without sharing cookies.

**Real-World Usage Scenarios**

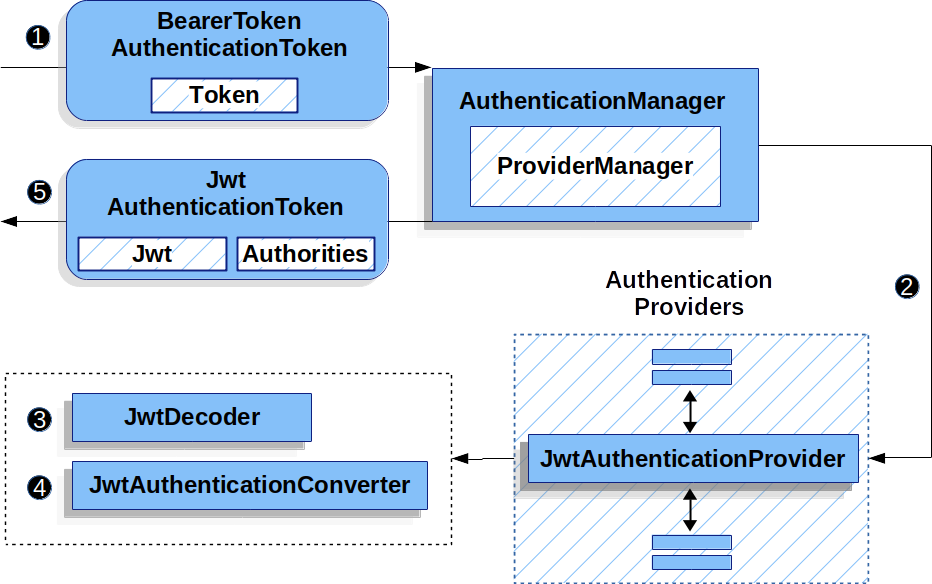
1. **Microservices Architecture**
   * An order-processing microservice accepts only JWTs issued by the central Auth service, avoiding Redis or database lookups on each call.
2. **Angular/React SPA**
   * After login, the app stores the JWT in memory (or secure HTTP-only cookie) and appends it to Authorization headers for all API calls.
3. **Native Mobile App**
   * A fitness tracking app stores a short-lived JWT and a longer-lived refresh token; it silently refreshes tokens when the access token expires.

**Why JWT Came to Be**

1. **Scalability & Statelessness**
   * Traditional session cookies require server-side storage, which doesn’t scale easily across distributed services; JWTs carry session state client-side.
2. **Cross-Language & Cross-Platform Standard**
   * JSON-based, language-agnostic format that works uniformly across Java, JavaScript, Python, Go, etc., without custom serialization.
3. **Decoupling Authentication from Business Logic**
   * Services can be purely resource-oriented, focusing on token validation rather than session lookup, greatly simplifying code.

**Real-World Motivations Leading to JWT Adoption**

1. **Breakdown of Monoliths**
   * When a large retail monolith split into microservices, JWTs enabled each new service to authenticate without a central session DB bottleneck.
2. **Mobile-First Strategies**
   * As companies built native apps, they needed a unified, lightweight token format that worked the same way for web and mobile.
3. **Third-Party Integrations**
   * APIs exposed to partners and B2B clients adopted JWT so that external developers could obtain tokens from an OAuth2 provider and call services directly.



🧩 Overview of Components

UserController – handles login

JWT Utility – to generate/validate tokens

JWT Filter – to intercept requests

SecurityConfig – configures Spring Security

📁 Project Structure

src/

└── main/

├── java/

│ └── com.example.jwt/

│ ├── config/SecurityConfig.java

│ ├── controller/AuthController.java

│ ├── filter/JwtAuthFilter.java

│ ├── service/UserService.java

│ ├── util/JwtUtil.java

│ └── model/AuthRequest.java

└── resources/

└── application.properties

**✅ 1. JwtUtil.java – JWT Utility**

@Component

public class JwtUtil {

private final String SECRET\_KEY = "mySecretKey";

public String generateToken(UserDetails userDetails) {

return Jwts.builder()

.setSubject(userDetails.getUsername())

.setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60)) // 1 hour

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

}

public boolean validateToken(String token, UserDetails userDetails) {

return userDetails.getUsername().equals(extractUsername(token)) && !isTokenExpired(token);

}

public String extractUsername(String token) {

return extractClaim(token, Claims::getSubject);

}

public Date extractExpiration(String token) {

return extractClaim(token, Claims::getExpiration);

}

public <T> T extractClaim(String token, Function<Claims, T> resolver) {

Claims claims = Jwts.parser()

.setSigningKey(SECRET\_KEY)

.parseClaimsJws(token)

.getBody();

return resolver.apply(claims);

}

private boolean isTokenExpired(String token) {

return extractExpiration(token).before(new Date());

}

}

**✅ 2. JwtAuthFilter.java – JWT Filter**

@Component

public class JwtAuthFilter extends OncePerRequestFilter {

@Autowired

private JwtUtil jwtUtil;

@Autowired

private UserService userService;

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain) throws ServletException, IOException {

final String authHeader = request.getHeader("Authorization");

String username = null;

String jwt = null;

if (authHeader != null && authHeader.startsWith("Bearer ")) {

jwt = authHeader.substring(7);

username = jwtUtil.extractUsername(jwt);

}

if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

UserDetails userDetails = userService.loadUserByUsername(username);

if (jwtUtil.validateToken(jwt, userDetails)) {

UsernamePasswordAuthenticationToken authToken =

new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authToken);

}

}

filterChain.doFilter(request, response);

}

}

✅ **3. SecurityConfig.java – Spring Security Config**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Autowired

private JwtAuthFilter jwtAuthFilter;

@Autowired

private UserService userService;

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

return http.csrf().disable()

.authorizeHttpRequests(auth -> auth

.requestMatchers("/auth/\*\*").permitAll()

.anyRequest().authenticated()

)

.sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

.userDetailsService(userService)

.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)

.build();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

}

**✅ 4. AuthController.java – Login API**

@RestController

@RequestMapping("/auth")

public class AuthController {

@Autowired

private AuthenticationManager authManager;

@Autowired

private JwtUtil jwtUtil;

@Autowired

private UserService userService;

@PostMapping("/login")

public ResponseEntity<?> login(@RequestBody AuthRequest request) {

Authentication authentication = authManager.authenticate(

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

final UserDetails userDetails = userService.loadUserByUsername(request.getUsername());

final String jwt = jwtUtil.generateToken(userDetails);

return ResponseEntity.ok(Collections.singletonMap("token", jwt));

}

}

**✅ 5. AuthRequest.java – Login Request DTO**

@Data

public class AuthRequest {

private String username;

private String password;

}

✅ 6. UserService.java – UserDetailsService Impl

@Service

public class UserService implements UserDetailsService {

private static final Map<String, String> USERS = Map.of("admin", "$2a$10$xxxxx...");

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

if (!USERS.containsKey(username)) {

throw new UsernameNotFoundException("User not found");

}

return new User(username, USERS.get(username), List.of(new SimpleGrantedAuthority("ROLE\_USER")));

}

}

💡 You can replace USERS with a real database in production.

✅ Test with Postman or Curl

Login:

POST /auth/login

{

"username": "admin",

"password": "password"

}

→ Returns a JWT.

Access a protected route:

Add Authorization: Bearer <token> header to request.

**Summary**

JWT simplifies stateless, scalable auth.

Use Spring filters to intercept and validate JWTs.

Define secure APIs using Spring Security rules.

# ✅ Spring Security Quiz

**1. What is the primary purpose of Spring Security?**  
A. Database integration  
B. Caching data  
C. Securing Java applications  
D. Managing REST endpoints

**Answer:** C. Securing Java applications

**2. Which of the following is the default authentication mechanism in Spring Security?**  
A. JWT  
B. OAuth2  
C. Basic authentication with in-memory user store  
D. LDAP

**Answer:** C. Basic authentication with in-memory user store

**3. What is the role of UserDetailsService in Spring Security?**  
A. Validates passwords  
B. Encodes JWT tokens  
C. Loads user-specific data  
D. Filters web requests

**Answer:** C. Loads user-specific data

**4. Which annotation is used to enable method-level security?**  
A. @EnableWebSecurity  
B. @EnableMethodSecurity  
C. @Secured  
D. @EnableGlobalAuthentication

**Answer:** B. @EnableMethodSecurity

**5. What does the SecurityFilterChain bean define in Spring Security 6+?**  
A. The entire Spring context  
B. Custom user roles  
C. The order and configuration of security filters  
D. OAuth client credentials

**Answer:** C. The order and configuration of security filters

**6. What is the purpose of PasswordEncoder?**  
A. Convert plain text to uppercase  
B. Encode and match passwords securely  
C. Store passwords in memory  
D. Encrypt the database

**Answer:** B. Encode and match passwords securely

**7. How do you restrict access to a controller method to authenticated users only?**  
A. @PermitAll  
B. @Secured("IS\_AUTHENTICATED")  
C. @PreAuthorize("isAuthenticated()")  
D. @AuthenticatedOnly

**Answer:** C. @PreAuthorize("isAuthenticated()")

**8. Which filter is responsible for processing username and password login requests?**  
A. AuthorizationFilter  
B. SecurityContextPersistenceFilter  
C. UsernamePasswordAuthenticationFilter  
D. BasicAuthenticationFilter

**Answer:** C. UsernamePasswordAuthenticationFilter

**9. What is CSRF protection in Spring Security?**  
A. Cross-Site Resource Fetching  
B. Cross-Site Request Forgery protection  
C. Credential Storage Retention Filter  
D. Custom Servlet Routing Filter

**Answer:** B. Cross-Site Request Forgery protection

**10. What does the @Secured("ROLE\_ADMIN") annotation do?**  
A. Allows only users with "ROLE\_ADMIN" to access the method  
B. Logs user login activity  
C. Enables secure cookies  
D. Encrypts sensitive parameters

**Answer:** A. Allows only users with "ROLE\_ADMIN" to access the method

**11. Which interface represents the authenticated user in Spring Security?**  
A. AuthenticationManager  
B. UserDetails  
C. SecurityContextHolder  
D. Authentication

**Answer:** D. Authentication

**12. What is the role of SecurityContextHolder?**  
A. Holds user sessions in cache  
B. Stores security context including authentication for current thread  
C. Encrypts user data  
D. Handles login requests

**Answer:** B. Stores security context including authentication for current thread

**13. How can you define custom login logic in Spring Security?**  
A. Override SecurityFilterChain  
B. Use @LoginHandler  
C. Implement AuthenticationProvider  
D. Use @ControllerSecurity

**Answer:** C. Implement AuthenticationProvider

**14. What is AuthenticationManager used for?**  
A. Logging access attempts  
B. Decoding tokens  
C. Authenticating credentials against UserDetailsService  
D. Displaying login pages

**Answer:** C. Authenticating credentials against UserDetailsService

**15. Which class is commonly used to customize access rules in Spring Security?**  
A. WebSecurityCustomizer  
B. SecurityRulesConfigurer  
C. AccessDeniedFilter  
D. SecurityInterceptor

**Answer:** A. WebSecurityCustomizer