Spring Security Interview Questions and Answers

[**https://www.baeldung.com/spring-security-oauth-jwt**](https://www.baeldung.com/spring-security-oauth-jwt)

**+How JWT authentication works? – Step by Step (Simple Points)**

1. **User logged in** into username & password

* "username": "john",
* "password": "password123"

1. **Server checks credentials** → Looks them up in the database.

* checks if the username exists and if the password matches using a UserDetailsService and a password encoder (like BCrypt).

1. **If correct, server creates a JWT token.**

* String token = Jwts.builder().setSubject("john")
* .claim("role", "USER").setIssuedAt(new Date())
* .setExpiration(new Date(System.currentTimeMillis() + 86400000)) // 1 day
* .signWith(SignatureAlgorithm.HS256, SECRET\_KEY).compact();

1. **Token includes user info** like username, role, and expiry time.

* "sub": "john",
* "role": "USER",
* "exp": 1723129380

1. **Server sends the token** back to the user.

* "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6..."

1. **User stores the token** in local Storage, session Storage, or cookies.

* localStorage.setItem("token", "<your\_jwt\_token>");

1. **User sends token** with every request in the header:

* Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...

1. **Server gets the token** from the header.
2. **Server checks the token** → Validates signature, expiry, and user info.

* Claims claims = Jwts.parser().setSigningKey(SECRET\_KEY)
* .parseClaimsJws(token).getBody();

1. **If invalid or expired** → Server returns 401 Unauthorized.

.

**How to implement Spring security in Spring boot project**

**Add the dependency:**

* Include spring-boot-starter-security in the pom.xml.

**Create a configuration class:**

* We create a class like SecurityConfig using @Configuration to customize security rules.

**Define access rules:**

* We allow or restrict access to endpoints using methods like .permitAll() or .authenticated().

**Set up authentication:**

* We configure in-memory users or connect to a database using UserDetailsService.

**Encode passwords:**

* We use BCryptPasswordEncoder to securely store passwords.

**Choose authentication type:**

* We can use httpBasic(), formLogin(), or JWT tokens for API security.

**How to implement spring security in your spring boot Application.**

* 1. Added Spring Security Stater Dependency in pom .xml file.
  2. Creates a **default user** with a generated password shown in logs
  3. If you want You can change this in application. Properties:

spring.security.user.name=admin

spring. security. User. password=admin123

* 1. Create a Security Configuration Class

1. **What is Spring Security?.**  
   Spring Security is a powerful and customizable authentication and access-control framework for Java applications, particularly those built using Spring. It handles security-related concerns like authentication, authorization, CSRF protection, and session management.

**2. What are the main features of Spring Security?**

* Authentication & Authorization
* Password Encoding
* CSRF Protection
* Session Management
* Security Headers
* Integration with OAuth2/JWT
* Method-level Security, LDAP and Custom Authentication Providers

1. **How does Spring Security work internally?.**  
   Spring Security uses a **filter chain** (SecurityFilterChain)that intercepts incoming HTTP requests. It passes requests through a series of filters (e.g., UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter) to perform authentication and authorization checks.

**4. What is the difference between authentication and authorization?**

* **Authentication**: Verifies identity (e.g., login with username/password).
* **Authorization**: Grants or denies access based on roles/permissions after authentication.

**5. What is the default behavior of Spring Security when added to a Spring Boot application?**

* All endpoints are secured.
* Auto-generated login page is shown.
* A default user with a random password is created.

**6. How do you configure in-memory authentication in Spring Security?**

@Bean

public UserDetailsService userDetailsService() {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("password")

.roles("USER")

.build();

return new InMemoryUserDetailsManager(user);

}

**7. What is UserDetails and UserDetailsService?**

* UserDetails: Represents a user’s data (username, password, roles).
* UserDetailsService: Interface to load user-specific data from DB or memory.

**8. What is the role of AuthenticationManager in Spring Security?**  
It handles the authentication process by verifying credentials and returning an Authentication object if successful.

**9. How do you disable CSRF protection in Spring Security?**

http.csrf().disable();

**“  
  
  
  
  
  
  
  
  
  
  
1s0. What is PasswordEncoder? Why is it used?**  
PasswordEncoder is used to hash and verify passwords securely. It prevents storing passwords in plain text.

**⚙️ Intermediate Questions**

**11. What are the different ways to implement authentication in Spring Security?**

* In-memory authentication
* JDBC-based authentication
* LDAP authentication
* OAuth2/OpenID Connect
* JWT token-based authentication
* Custom authentication providers

**12. How does role-based access control (RBAC) work in Spring Security?**  
Access is controlled using annotations (@PreAuthorize("hasRole('ADMIN')")) or config (.antMatchers("/admin/\*\*").hasRole("ADMIN")).

**13. What is method-level security and how do you enable it?**  
Allows securing methods using annotations like @PreAuthorize, @Secured, etc.  
Enable with:

java

@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)

**14. How do you secure REST APIs using Spring Security?**

* Disable form login and CSRF
* Use stateless JWT authentication
* Secure endpoints with .antMatchers() and hasRole()

**15. Explain the difference between @PreAuthorize and @Secured.**

* @PreAuthorize: Supports complex SpEL expressions
* @Secured: Only allows role-based checks (e.g., @Secured("ROLE\_ADMIN"))

**16. What are filters in Spring Security and what is their order?**  
Filters are part of the Spring Security Filter Chain. Example:

* SecurityContextPersistenceFilter
* UsernamePasswordAuthenticationFilter
* ExceptionTranslationFilter
* FilterSecurityInterceptor

Their order is defined in SecurityFilterChain.

**17. How do you integrate JWT with Spring Security?**

* Disable session management
* Add a JWT filter before UsernamePasswordAuthenticationFilter
* Extract and validate token, then set Authentication in SecurityContextHolder

**18. How do you handle exceptions in Spring Security (access denied, unauthorized)?**  
Configure with:

http .exceptionHandling()

.authenticationEntryPoint(...)

.accessDeniedHandler(...);

**19. How do you configure custom login and logout functionality?**

http

.formLogin()

.loginPage("/login")

.defaultSuccessUrl("/home")

.and()

.logout()

.logoutUrl("/logout")

.logoutSuccessUrl("/login?logout");

**20. What is security context and how is it managed in Spring Security?**  
SecurityContext holds the Authentication object of the current user and is managed via SecurityContextHolder.

**🚀 Advanced Questions**

**21. Explain the Spring Security filter chain.**  
Each request passes through filters in the SecurityFilterChain, such as:

* SecurityContextPersistenceFilter
* AuthenticationFilter
* AuthorizationFilter  
  Each filter performs a part of the security logic (auth, access control, etc.).

**22. How do you implement custom authentication logic?**  
Implement UserDetailsService or AuthenticationProvider to authenticate against a DB or external service.

**23. What is stateless authentication and how is it supported?**  
Stateless means no session is stored. Use JWT tokens to authenticate each request independently. Use:

http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

**24. How to implement Multi-Factor Authentication (MFA)?**

* Step 1: Username + Password
* Step 2: OTP or email/SMS token
* Custom filters or controllers can manage each step.

**25. What are SecurityContextHolder and SecurityContext?**

* SecurityContext: Holds security-related information (like authenticated user).
* SecurityContextHolder: Provides access to SecurityContext, typically using a thread-local.

**26. How do you persist security context across requests?**  
Spring Security uses SecurityContextPersistenceFilter to load/save SecurityContext from HttpSession or via stateless tokens (JWT).

**27. How do you secure apps using OAuth2/OpenID Connect?**  
Use Spring Security OAuth2 Client and add:

spring.security.oauth2.client.registration.google.client-id=...

Use @RegisteredOAuth2AuthorizedClient in controllers to access tokens.

**28. What is AuthenticationProvider and how do you create a custom one?**  
It validates user credentials. Implement the interface:

public class MyAuthProvider implements AuthenticationProvider {

public Authentication authenticate(Authentication auth) { ... }

public boolean supports(Class<?> clazz) { return true; }

}

**29. How to prevent session fixation and related attacks?**  
Spring Security prevents session fixation by default with:

http.sessionManagement().sessionFixation().migrateSession();

**30. Explain how to implement refresh tokens in a JWT system.**

* Generate and store a long-lived refresh token in DB
* Provide /refresh endpoint
* On access token expiry, client sends refresh token
* Validate and issue new access + refresh token

**📘 Hands-On Questions**

**31. How to allow /login publicly and secure others?**

http

.authorizeHttpRequests()

.requestMatchers("/login").permitAll()

.anyRequest().authenticated();

**32. Spring Security config class for JWT auth**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

return http

.csrf().disable()

.authorizeHttpRequests()

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

.anyRequest().authenticated()

.and()

.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)

.build();

}

}

**33. Testing secured endpoints with Postman or MockMvc**

* Add Authorization: Bearer <token> header in Postman
* In tests:

mockMvc.perform(get("/api").header("Authorization", "Bearer token"))

.andExpect(status().isOk());

**34. Demonstrate method-level security**

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

public void deleteUser(Long id) { ... }

@Secured("ROLE\_USER")

public void viewProfile() { ... }

**35. Impact of @EnableWebSecurity and @EnableGlobalMethodSecurity**

* @EnableWebSecurity: Enables Spring Security web configuration.
* @EnableGlobalMethodSecurity: Enables method-level annotations like @PreAuthorize, @Secured.

**To implement JWT with Spring Security in a Spring Boot project.**

**Add dependencies:**

* Include spring-boot-starter-security and a JWT library like jjwt in your pom.xml.

**Disable default login:**

* We disable default formLogin or httpBasic and instead use a stateless JWT-based mechanism.

**Create an authentication endpoint:**

* We expose a /login or /authenticate endpoint where users send credentials.

  If valid, we generate and return a JWT.

**Generate JWT tokens:**

* We use a utility class to generate tokens using username, roles, and secret key.

**Create a JWT filter:**

* A custom filter intercepts incoming requests, validates the JWT, and sets authentication in the context.

**Secure endpoints:**

* We use SecurityFilterChain or configure(HttpSecurity) to allow public access to /authenticate and secure other endpoints.

----------------------------------------------------------------------------------------------------------------------------

**1. What is JWT?**

JWT (JSON Web Token) is a compact, URL-safe token used for **stateless authentication**.  
👉 *One line:* It allows secure transmission of information between parties as a JSON object.

**2. What are the main parts of a JWT?**

* **Header**, **Payload**, **Signature**.  
  👉 *One line:* JWT = Header + Payload + Signature, separated by dots (.).

**3. How does JWT authentication work?**

User logs in → Server generates token → Client sends token in Authorization header → Server validates token.  
👉 *One line:* The server authenticates the user on each request by validating the JWT.

**4. Where is JWT stored?**

Usually on the **client-side**, e.g., **LocalStorage**, **SessionStorage**, or **HTTP-only cookies**.  
👉 *One line:* JWTs are stored client-side and sent with each protected request.

**5. What are claims in JWT?**

Claims are pieces of information in the **payload**, like sub, exp, role, userId.  
👉 *One line:* Claims describe token metadata like issuer, expiry, and user details.

**✅ Intermediate Questions**

**6. How do you validate a JWT token?**

Check the **signature**, **expiration**, and **claims** using the server's secret or public key.  
👉 *One line:* A JWT is valid if it’s signed correctly and not expired.

**7. How do you secure JWT?**

* Use HTTPS
* Sign with strong secret
* Set short expiry
* Store refresh token securely  
  👉 *One line:* Security comes from strong keys, short expiration, and proper transport.

**8. What is the difference between Access Token and Refresh Token?**

* **Access Token:** Short-lived, used for accessing APIs
* **Refresh Token:** Long-lived, used to generate new access tokens  
  👉 *One line:* Access tokens authenticate requests; refresh tokens renew them.

**9. Can JWT be tampered with?**

Yes, unless **properly signed**, it can be forged.  
👉 *One line:* Signature verification ensures token integrity and authenticity.

**10. What are the common algorithms used to sign JWTs?**

* **HS256** (symmetric)
* **RS256** (asymmetric)  
  👉 *One line:* HS256 uses the same key; RS256 uses private/public key pair.

**✅ Advanced Questions**

**11. How is JWT integrated with Spring Security?**

By creating a **custom filter** that intercepts requests, validates the token, and sets the authentication context.  
👉 *One line:* JWT is added via a filter before the UsernamePasswordAuthenticationFilter.

**12. How do microservices validate JWT tokens?**

Each microservice **validates** the token using a **shared secret** or **public key**.  
👉 *One line:* JWT allows decentralized authentication across services.

**13. Can JWT be revoked?**

👉 *One line:* JWT is stateless, so revocation must be implemented manually.

**14. How do you handle expired JWT tokens?**

Use a **refresh token** mechanism to reissue access tokens.  
👉 *One line:* Expired access tokens require a refresh token to get a new one.

**15. What’s the difference between symmetric and asymmetric JWT signing?**

* **Symmetric (HS256):** Same key signs and verifies
* **Asymmetric (RS256):** Private key signs, public key verifies  
  👉 *One line:* Asymmetric is more secure for distributed systems.

**✅ Bonus: Spring Boot Integration Questions**

**16. How do you create JWT in Spring Boot?**

Use a library like io.jsonwebtoken (JJWT) to generate tokens after login.  
👉 *One line:* Create JWTs using a secret key and user details on successful authentication.

**17. How do you extract claims from a JWT in Spring Boot?**

Use Jwts.parser().setSigningKey().parseClaimsJws(token).getBody()  
👉 *One line:* Claims like username and roles are extracted from the token payload.

**18. How to test JWT-protected APIs?**

Use **Postman** or **JUnit + MockMvc** with a valid token in the Authorization header.  
👉 *One line:* JWT-protected endpoints require Bearer <token> for testing.

* 1. **What is Role-Based Access Control (RBAC)?**
* **It** is a security method that gives access to resources **based on the user’s role**.
* **ADMIN** – Full access to manage the system
* **MANAGER** – Access to team or department-level data
* **USER** – Regular access for general users
* A **user** is assigned one or more **roles** (e.g., ADMIN, USER).
* Each **API or method** is protected to allow only specific roles to access it.
* Spring Security automatically **checks the user's role** during each request.
* If the role matches → access is **granted**.
* If the role doesn’t match → access is **denied** with a 403 Forbidden error.
* @Override
* protected void configure(HttpSecurity http) throws Exception {
* http
* .authorizeRequests()
* .**antMatchers("/admin/\*\*").hasRole("ADMIN")**
* **.antMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN")**
* .anyRequest().authenticated()
* .and()
* .formLogin();
* Enable annotation-based security:
* @EnableGlobalMethodSecurity(prePostEnabled = true)

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

public void adminOnlyMethod() {

// only accessible by ADMIN

}

**@PreAuthorize("hasAnyRole('USER', 'ADMIN')")**

public void userOrAdminMethod() {

// accessible by USER or ADMIN

}

# 21. JWT authentication in Spring Boot, I follow these steps:

1. **Add JJWT Dependencies** → I include **jjwt-api**, **jjwt-impl**, and **jjwt-jackson** to create, parse, and validate JWT tokens.
2. **Configure Secret & Expiry** → In application.properties, I define a **secret key** for signing tokens and set the **token expiration time**.
3. **Create JwtUtil Class** → This utility class handles **generating tokens**, **extracting claims** like username, and **validating tokens** using the secret key.
4. **Authentication Endpoint** → In AuthController, after successful **username/password authentication**, I **generate the JWT** using JwtUtil and return it to the client.
5. **JWT Filter** → I create a custom filter JwtRequestFilter that **intercepts every request**, extracts the token from the Authorization header, validates it, and sets the **Spring Security context**.
6. **Security Configuration** → In SecurityConfig, I **permit** the /authenticate endpoint, secure other endpoints, **register the JWT filter**, and make the app **stateless** since JWT doesn’t use sessions.
7. **Testing Flow** → First, **POST** credentials to /authenticate → receive a JWT → then send it in the Authorization: Bearer <token> header → server validates and allows access.

# 22.Enable Method-Level Security

1. **Enable Method Security** → Add @EnableGlobalMethodSecurity (prePostEnabled = true) in the security configuration class.
2. **Use Security Annotations**:
   * **@PreAuthorize** → Runs **before** the method executes.
   * **@PostAuthorize** → Runs **after** the method executes.
   * **@Secured** → Restricts access to specific roles.

# **23.****How to create JWT token**

**1: Add the dependencies:**

* + **jjwt-api** → Provides core classes for creating, parsing, and validating JWTs.
  + **jjwt-impl** → Implementation for JWT generation and validation.
  + **jjwt-jackson** → Handles JSON serialization and deserialization of JWT claims.

**2: Generate Secret Key**

* A secret key is required to **sign the JWT**.
* In production, the key should be stored securely (like environment variables or config server).

Key secretKey = Keys.secretKeyFor(SignatureAlgorithm.HS256);

**3: Build the JWT Token**

String token = Jwts.builder() **// to Create the token.**

**.**setSubject("username") **// username or userId**

**.**setIssuedAt(new Date()) **// Token Creation time**

. setExpiration(new Date(System.currentTimeMillis() + 3600000)) // 1 **hour //expiry time**

. signWith(secretKey) **// signing the token**

.compact();

**4: Validate JWT Token**

* To validate, parse the token with the same secret key.
* If parsing succeeds, token is valid; otherwise, throw exception.

Jwts.parserBuilder() **// Create a JWT parser (used to read/validate tokens)**

. setSigningKey(secretKey) **// Provide the secret key to verify the token’s signature.**

. build ()**// Build the parser instance**

. parseClaimsJws(token); **// Parse and validate the token, returns claims if valid**

**Step 5: Extract Data from JWT**

* Claims (payload) can be extracted, such as username, roles, etc.

String username = Jwts.parserBuilder() // Create a JWT parser

.setSigningKey (secretKey) // Set secret key to verify signature

.build() // Build the parser

.parseClaimsJws(token) // Parse and validate the JWT token

.getBody() // Get the claims (payload)

.getSubject(); // Extract the 'subject' (username)

**🔹 Summary for Interview**

👉 To create a JWT token in Spring Boot:

1. Add **JJWT dependency**.
2. Create a **secret key** for signing.
3. Use Jwts.builder() to generate the token with subject, issue date, and expiry date.
4. **Sign** the token using HMAC SHA or RSA algorithm.
5. **Validate** tokens by parsing with the secret key.
6. Extract claims like **username** or **roles** for authorization.