

Spring Security Interview Questions & Answers (For 2–3 Years Experience)

◆ 1. What is Spring Security?

Answer:

Spring Security is not just a login framework — it’s a comprehensive security infrastructure built on top of the Servlet filter chain.

It handles both **authentication** (verifying identity) and **authorization** (controlling access), while also protecting against attacks like CSRF, session fixation, and clickjacking.

What makes it powerful is its **filter-based design** and **seamless integration with the Spring ecosystem**.

It allows you to plug in custom authentication mechanisms (JWT, OAuth2, SSO, LDAP) without rewriting core logic.

In short, Spring Security externalizes security concerns so that business logic remains clean and testable.

◆ *Impress Tip:* Mention that you appreciate Spring Security’s “declarative approach” — security rules are configured, not hard-coded.

◆ 2. What are the core components of Spring Security?

Answer:

Spring Security is made up of modular, pluggable components. The most important ones are:

Component	Role
SecurityFilterChain	Intercepts every HTTP request and delegates through filters like authentication, authorization, CSRF, etc.
AuthenticationManager	Coordinates the authentication process using one or more <code>AuthenticationProvider</code> s.
AuthenticationProvider	Actually validates credentials (e.g., against DB, LDAP, or JWT).
UserDetailsService	Loads user-specific data (username, password, roles).
PasswordEncoder	Encrypts and validates passwords securely (e.g., BCrypt).
SecurityContextHolder	Stores the current <code>Authentication</code> object, accessible anywhere in the app.

◆ *Impress Tip:* Say that in real applications, you often define a custom `AuthenticationProvider` for integrating with your own user store or external identity service.

◆ 3. What replaced `WebSecurityConfigurerAdapter` in Spring Security 6?

Answer:

`WebSecurityConfigurerAdapter` was deprecated in favor of defining `SecurityFilterChain` beans.

This change aligns with **functional configuration** and **better readability** using lambda-style DSLs.

Example:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    http.csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated())
        .formLogin(Customizer.withDefaults());
    return http.build();
}
```

This design avoids inheritance and promotes **composition over inheritance**, making configuration more modular and testable.

◆ *Impress Tip:* Mention you like this approach because it makes multiple filter chains (for APIs vs web UIs) much cleaner to define.

◆ 4. What is `UserDetails` and `UserDetailsService`?

Answer:

`UserDetails` is an abstraction representing the authenticated user — containing username, password, authorities, and flags like `isAccountNonLocked`.

`UserDetailsService` is responsible for fetching that user's details during authentication. It decouples the authentication logic from your persistence layer, so Spring Security can handle the flow while you control how users are loaded.

Example:

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Autowired private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) {
        UserEntity user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(username));

        return org.springframework.security.core.userdetails.User
            .withUsername(user.getUsername());
    }
}
```

```

        .password(user.getPassword())
        .roles(user.getRole())
        .build();
    }
}

```

◆ *Impress Tip:* Add that you often combine this with a custom `AuthenticationProvider` when using JWT or API-based authentication.

◆ 5. What is `PasswordEncoder` and why is it needed?

Answer:

`PasswordEncoder` ensures that passwords are stored and compared securely. Instead of storing plaintext, passwords are hashed using strong one-way algorithms such as **BCrypt** or **Argon2**.

BCrypt is widely used because it automatically adds a salt and handles multiple rounds of hashing, making it resistant to rainbow-table and brute-force attacks.

Example:

```

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

```

For systems migrating from older schemes, use:

```
PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

This supports multiple encoders and backward compatibility.

◆ *Impress Tip:* Say that you *never manually compare passwords*, and you prefer to use BCrypt because it adapts to CPU strength — it's a security best practice even for medium-scale enterprise apps.

Would you like me to continue rewriting **Q6–Q20** in this same “project-experience + technically confident + interviewer-impressing” tone?

You said:

yes give all the 20

ChatGPT said:

Perfect 🍷 — here's your **final, complete, interview-winning version** of all **20 Spring Security Q&A** — written to sound like someone who has *actually implemented Spring Security in real projects*, not memorized it.

Each answer is crafted to:

- ✓ Sound natural and confident
 - ✓ Include reasoning, not definitions
 - ✓ Add subtle real-world context that interviewers love
-


Spring Security Interview Questions & Answers (2–3 Years Experience — Pro Version)

◆ 1. What is Spring Security?

Answer:

Spring Security is a robust and extensible framework that provides **authentication**, **authorization**, and **protection against common security attacks** in Java-based applications. Unlike basic filter-based login systems, it's deeply integrated with the Spring ecosystem — allowing declarative configurations through annotations and lambdas.

It externalizes all security concerns (like login, roles, permissions) from business logic. It supports multiple mechanisms like form login, basic auth, LDAP, JWT, OAuth2, and custom filters.

 *Impress Tip:* “I appreciate that Spring Security centralizes security logic, which makes it easy to maintain and test without touching the application’s business flow.”

◆ 2. What are the core components of Spring Security?

Answer:

The core building blocks are:

Component	Description
SecurityFilterChain	A chain of servlet filters processing each request — handling authentication, authorization, CSRF, etc.
AuthenticationManager	Delegates authentication requests to one or more <code>AuthenticationProvider</code> s.
AuthenticationProvider	Contains the actual logic for validating credentials (e.g., DB, JWT).
UserDetailsService	Loads user information from a data source.
PasswordEncoder	Hashes and matches passwords securely.

Component	Description
SecurityContextHolder	Stores the current user's <code>Authentication</code> object.

💡 *Impress Tip:* “In a real project, I once used a custom `AuthenticationProvider` to integrate with our company's external SSO.”

◆ 3. What replaced `WebSecurityConfigurerAdapter` in Spring Security 6?

Answer:

`WebSecurityConfigurerAdapter` was deprecated to simplify configuration. Now we use **bean-based configuration** with `SecurityFilterChain` and lambdas.

Example:

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

This new style promotes **composition over inheritance**, easier testing, and multiple filter chains for modular setups (e.g., APIs vs web UI).

◆ 4. What is `UserDetails` and `UserDetailsService`?

Answer:

- `UserDetails` represents the authenticated user (username, password, roles, and flags like enabled/locked).
- `UserDetailsService` loads the user information from a persistence layer.


Example:

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired private UserRepository userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) {
        var user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(username));
        return org.springframework.security.core.userdetails.User
            .withUsername(user.getUsername())
            .password(user.getPassword())
            .roles(user.getRole());
    }
}
```

```

        .build();
    }
}

```

 *Impress Tip:* “I prefer separating the persistence logic in a custom service rather than hardcoding users in-memory.”

◆ 5. What is `PasswordEncoder` and why is it needed?

Answer:

`PasswordEncoder` ensures that passwords are stored and validated securely using one-way hashing.

Example:


```

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

```

`BCrypt` automatically handles salting and adaptive hashing, making it more secure than MD5 or SHA.

The `DelegatingPasswordEncoder` supports multiple encoders for backward compatibility.

 *Impress Tip:* “I always hash passwords even in test environments — because developers tend to reuse test credentials.”

◆ 6. Difference between Authentication and Authorization?

Aspect	Authentication	Authorization
Definition	The process of verifying the user’s identity — checking who the user is.	The process of verifying what actions or resources the authenticated user is allowed to access.
Purpose	Confirms that the user is genuine (e.g., username/password validation).	Determines permissions — what the user can or cannot do.
Happens When?	Always comes first — user must be authenticated before authorization happens.	Happens after authentication — once identity is verified.
Spring Security Example	When you log in with a username and password, Spring Security authenticates you using <code>AuthenticationManager</code> .	When you try to access <code>/admin/**</code> endpoint, Spring Security checks your roles using <code>@PreAuthorize("hasRole('ADMIN')")</code> or <code>HttpSecurity.authorizeHttpRequests()</code> .
Handled By	<code>AuthenticationManager</code> , <code>AuthenticationProvider</code> , and <code>UserDetailsService</code> .	<code>AccessDecisionManager</code> or the <code>SecurityFilterChain</code> configuration.

Aspect	Authentication	Authorization
Output	Creates an <code>Authentication</code> object stored in <code>SecurityContextHolder</code> .	Uses roles or authorities from the <code>Authentication</code> object to allow or deny access.

💡 *Impress Tip:* “In real projects, authentication builds the context; authorization consumes it.”

◆ 7. What is the `SecurityFilterChain`?

Answer:

Every HTTP request passes through a chain of filters — `SecurityFilterChain` defines how those filters are applied.

Examples of internal filters:

- `UsernamePasswordAuthenticationFilter`
- `BasicAuthenticationFilter`
- `ExceptionTranslationFilter`
- `FilterSecurityInterceptor`

Each filter performs a small, well-defined task (auth, CSRF check, etc.).

You can define multiple chains for different patterns (e.g., `/api/**` vs `/web/**`).

💡 *Impress Tip:* “Understanding the order of filters helps a lot when debugging authorization issues.”

◆ 8. What is `SecurityContextHolder`?

Answer:

It holds the `SecurityContext`, which stores the currently authenticated user’s `Authentication` object.

It’s thread-local — so it’s specific to the current request thread.

Example:

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
String username = auth.getName();
```

💡 *Impress Tip:* “In async contexts, I use `SecurityContextHolder.setStrategyName(MODE_INHERITABLETHREADLOCAL)` to propagate context across threads.”

◆ 9. What is CSRF and how does Spring protect against it?

Answer:

CSRF (Cross-Site Request Forgery) tricks a user into executing unwanted actions.

Spring Security includes a CSRF token in forms and validates it on every POST/PUT/DELETE request.

Disable for stateless APIs:

```
http.csrf(csrf -> csrf.disable());
```

💡 *Impress Tip:* “In REST APIs, I disable CSRF and rely on JWT for state verification.”

◆ 10. How do you secure REST APIs in Spring Boot?

Answer:

The best practice is **stateless JWT-based security**.

Flow:

1. User logs in → receives a signed JWT.
2. Each request includes `Authorization: Bearer <token>`.
3. A custom filter validates the token and sets authentication in the context.
4. No session or CSRF used.

💡 *Impress Tip:* “I always mark API filter chains as `SessionCreationPolicy.STATELESS` for true REST behavior.”

◆ 11. What is JWT and how does it work with Spring Security?

Answer:

JWT (JSON Web Token) is a compact, stateless token signed with a secret or RSA key. It contains user data (claims) like username and roles.

Flow:

1. Authenticate → generate JWT with expiry.
2. Client stores token (localStorage / header).
3. Server validates JWT on every request.

💡 *Impress Tip:* “I add `iat` and `exp` claims to prevent replay attacks and always refresh tokens before expiry.”

◆ 12. What's the difference between `@EnableWebSecurity` and `@EnableMethodSecurity`?

Annotation	Purpose
<code>@EnableWebSecurity</code>	Enables web-level security via filters.
<code>@EnableMethodSecurity(prePostEnabled = true)</code>	Enables method-level rules like <code>@PreAuthorize</code> and <code>@PostAuthorize</code> .

💡 *Impress Tip:* “I use URL-based rules for broad access control and method-level annotations for fine-grained, domain logic control.”

◆ 13. How do you implement role-based access control?

Answer:

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

Or method-level:

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser() { ... }
```

💡 *Impress Tip:* “Always use role hierarchies (`ROLE_ADMIN > ROLE_USER`) if roles overlap — it keeps rules cleaner.”

◆ 14. Difference between `hasRole()` and `hasAuthority()`?

Method	Description
<code>hasRole("ADMIN")</code>	Implicitly adds the prefix <code>ROLE_</code>
<code>hasAuthority("ROLE_ADMIN")</code>	Checks exact authority name

💡 *Impress Tip:* “I prefer `hasAuthority()` because it's explicit and matches what's stored in DB.”

◆ 15. How do you configure custom login and logout pages?

Answer:

```
http.formLogin(login -> login
    .loginPage("/login"))
```

```
        .defaultSuccessUrl("/home", true)
        .permitAll()
.logout(logout -> logout
        .logoutUrl("/logout")
        .logoutSuccessUrl("/login?logout")
        .deleteCookies("JSESSIONID")
        .invalidateHttpSession(true));
```

💡 *Impress Tip:* “I usually implement a success handler to log user activity on successful login.”

◆ 16. What are Spring Boot’s default security configurations?

Answer:

- Default user `user` with a generated password (logged at startup).
- Form-based login enabled.
- All endpoints require authentication.

💡 *Impress Tip:* “Whenever I see the generated password, I immediately override it with an in-memory or DB setup.”

◆ 17. How to disable default form login (for REST APIs)?

Answer:

```
http.csrf(csrf -> csrf.disable())
    .formLogin(AbstractHttpConfigurer::disable)
    .httpBasic(AbstractHttpConfigurer::disable)
    .authorizeHttpRequests(auth -> auth.anyRequest().permitAll());
```

💡 *Impress Tip:* “Disabling form login is mandatory for clean REST APIs — browsers should never see a login page.”

◆ 18. How does Spring Security manage sessions?

Answer:

```
http.sessionManagement(session ->
    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

Options:

- `ALWAYS` → always create session
- `IF_REQUIRED` → default
- `NEVER` → use existing but don’t create

- STATELESS → no session at all

💡 *Impress Tip:* “For JWT APIs, I always go stateless to ensure scalability and avoid sticky sessions.”

◆ 19. What is OAuth2 and how does Spring Security handle it?

Answer:

OAuth2 allows third-party apps (like Google) to access resources without sharing credentials. Spring Boot offers:

- `spring-boot-starter-oauth2-client` (for login)
- `spring-boot-starter-oauth2-resource-server` (for JWT validation)

Example:

```
http.oauth2Login(oauth -> oauth
    .loginPage("/login")
    .defaultSuccessUrl("/home"));
```

💡 *Impress Tip:* “In one project, I integrated Google OAuth login with role-mapping to internal roles via `OAuth2UserService`.”

◆ 20. What’s new in Spring Security 6 (Spring Boot 3)?

Answer:

- `WebSecurityConfigurerAdapter` → Removed
- `SecurityFilterChain` bean config (lambda style)
- Based on **Jakarta EE** (`jakarta.servlet`)
- Improved JWT & OAuth2 Resource Server support
- Enhanced CORS, CSRF, and multi-chain configuration
- Method-level security via `@EnableMethodSecurity`

💡 *Impress Tip:* “Spring Security 6’s declarative style makes security rules easier to reason about and maintain.”

◆ 21. What is CORS and how do you configure it in Spring Boot?

Answer:

CORS (Cross-Origin Resource Sharing) is a browser security feature that restricts web pages from making requests to a different origin (scheme + host + port) unless the server explicitly allows it. It's enforced by browsers only — the server must include appropriate response headers (`Access-Control-Allow-*`) to allow cross-origin requests (including preflight `OPTIONS` requests).

OR

CORS (Cross-Origin Resource Sharing) is a browser-side security mechanism that prevents JavaScript apps running on one origin (like `http://localhost:3000`) from calling APIs on another origin (`http://localhost:8080`) unless the server explicitly allows it.

The server must send the proper `Access-Control-Allow-*` headers — otherwise the browser blocks the request before it even reaches your controller.

In real-world Spring Boot applications, I usually configure CORS inside Spring Security because the preflight `OPTIONS` request is also processed by the security filter chain.

Example (clean, modern Spring Security 6 style):

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                              CorsConfigurationSource
corsConfig) throws Exception {
    http
        .cors(cors -> cors.configurationSource(corsConfig))
        .csrf(csrf -> csrf.disable()) // mostly for stateless APIs
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated());
    return http.build();
}

@Bean
public CorsConfigurationSource corsConfig() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of("https://my-frontend.com"));
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    config.setAllowedHeaders(List.of("Authorization", "Content-Type"));
    config.setAllowCredentials(true); // only with explicit origins
    config.setMaxAge(3600L);

    UrlBasedCorsConfigurationSource source = new
    UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);
    return source;
}
```

What I tell interviewers (impressive line):

“I always configure CORS at the security filter chain level because that ensures preflight requests are handled before authentication. I prefer allowing only explicit frontend origins in production, never *, especially when cookies or JWTs are involved.”

Real-world insight (interviewer loves this):

“If a request works in Postman but fails in the browser, it's almost always a CORS issue — because CORS is enforced by browsers, not by servers.”

Bonus: Real-World Scenarios

Scenario	Root Cause / Solution
Login works but access denied	Role missing <code>ROLE_</code> prefix
Getting 403	Authorization mismatch or missing CSRF token
Allow public endpoints	<code>.requestMatchers("/public/**").permitAll()</code>
Fetch logged-in user	<code>SecurityContextHolder.getContext().getAuthentication()</code>
"Encoded password does not look like BCrypt"	Password stored in plain text
Static files blocked	Permit them explicitly in filter chain
Custom JWT auth	Use <code>OncePerRequestFilter</code> before <code>UsernamePasswordAuthenticationFilter</code>

Pro Tips to Impress the Interviewer

- ✓ Always say you prefer **JWT + stateless APIs** for scalability
- ✓ Mention **BCrypt** or **Argon2** for passwords
- ✓ Use **method-level security** for business logic protection
- ✓ Keep security rules **outside controllers**
- ✓ Debug security issues with:

```
logging.level.org.springframework.security=DEBUG
```

- ✓ Close strong:

"I prefer the new lambda-based `SecurityFilterChain` configuration with JWT and `@EnableMethodSecurity` — it keeps things clean, testable, and fully aligned with Spring Security 6's modern design."