Prathamesh Arvind Jadhav

Spring Security Notes

Introduction to Spring Security

1. What is Spring Security?

- **Spring Security** is a **framework within the Spring ecosystem** designed to **secure Java applications**, especially web applications.
- It provides:
  - **Authentication** – verifying the identity of users.
  - **Authorization** – controlling access to resources based on user roles or permissions.
- It is highly **customizable** and **extensible**, integrating seamlessly with **Spring Boot, Spring MVC, and other Spring projects**.
- Protects against **common security threats**:
  - Cross-Site Request Forgery (**CSRF**)
  - Session Fixation
  - Clickjacking
  - Brute-force attacks
  - Security headers misconfigurations

---

2. Why Do We Need Spring Security?

- To **control access** to application resources.
- To **protect sensitive data** from unauthorized users.
- To implement **authentication mechanisms** (login forms, OAuth2, JWT, LDAP).
- To provide **authorization mechanisms** for **role-based or permission-based access control**.
- To **simplify security management** with filters, annotations, and configurable security policies.
- To comply with **security standards and audits** by providing logging and auditing capabilities.

Difference between Authentication and Authorization

1. Definition

- **Authentication**: The process of **verifying the identity of a user**.
  - It answers the question: **"Who are you?"**
- **Authorization**: The process of **granting or denying access to resources** based on the user's identity and roles/permissions.
  - It answers the question: **"What are you allowed to do?"**

---

2. Key Differences

Feature	Authentication	Authorization
Purpose	Verify the identity of a user	Determine user access level to resources
Question Answered	"Who are you?"	"What can you do?"
When It Occurs	Happens **first** in the security process	Happens **after authentication**
Based On	Credentials: username, password, biometric, tokens	Roles, permissions, access policies, or privileges
Spring Security Component	AuthenticationManager, UserDetailsService, PasswordEncoder	AccessDecisionManager, GrantedAuthority, annotations (@Secured, @PreAuthorize)
Example	Login form, OAuth2 login, JWT token verification	Accessing /admin URL only for users with ROLE_ADMIN
Outcome	User identity is validated	User is allowed or denied access to specific resources

---

3. Real-World Analogy

- **Authentication**: Showing your ID card at the entrance of a building.
- **Authorization**: Based on your ID, determining which rooms you can enter inside the building.

---

4. Spring Security Implementation

- **Authentication**:
  - Handled by **AuthenticationManager**.
  - User data fetched via **UserDetailsService**.
  - Passwords encoded using **PasswordEncoder**.
- **Authorization**:
  - Handled by **AccessDecisionManager**.
  - Access controlled using **roles/authorities** and **method-level annotations**:
    - @Secured("ROLE_ADMIN")
    - @PreAuthorize("hasRole('ADMIN')")
    - URL-based access rules using HttpSecurity.

---

5. Important Points

- **Authentication is mandatory** for a user to exist in the system.
- **Authorization depends on authentication**; a user must be authenticated first to check their permissions.
- A user can be authenticated but **not authorized** to access certain resources.

Security Filter Chain Overview

1. What is Security Filter Chain?

- Spring Security uses a **chain of servlet filters** to intercept and process **HTTP requests** before they reach your application.
- Each filter in the chain is responsible for a **specific security task** like authentication, authorization, CSRF protection, or exception handling.
- This chain is called the **Security Filter Chain**, and it forms the **core mechanism** by which Spring Security secures web applications.

---

2. How it Works

1. A **client sends an HTTP request** to the server.
2. The request enters the **security filter chain**.
3. Each **filter in the chain executes in a predefined order**.
4. Filters perform tasks like:
    - Checking authentication credentials
    - Validating CSRF tokens
    - Enforcing access rules based on roles/authorities
    - Handling exceptions
5. If any filter **rejects the request**, the chain stops and the request is **denied**.
6. If all filters pass, the request reaches the **application endpoint** (controller or resource).

---

3. Key Filters in the Chain

Filter	Purpose
ChannelProcessingFilter	Enforces HTTPS if required
SecurityContextPersistenceFilter	Loads and stores the SecurityContext across requests
UsernamePasswordAuthenticationFilter	Handles form-based login
BasicAuthenticationFilter	Handles HTTP Basic authentication
CsrfFilter	Validates CSRF tokens to prevent CSRF attacks
ExceptionTranslationFilter	Handles authentication and access-denied exceptions
FilterSecurityInterceptor	Applies authorization rules to URL patterns

Note: Spring Boot auto-configures these filters, but you can **customize the filter chain** using

Java configuration (SecurityFilterChain bean).

```
Client Request
     |
[Security Filters Chain]
     |
Authentication Filter --> Auth Manager --> UserDetailsService
     |
Authorization Filter --> AccessDecisionManager
     |
Application Endpoint (Controller/Resource)
```

- **Authentication Filter:** Validates user credentials.
- **Authorization Filter:** Checks if the user has permission to access the requested resource.
- **SecurityContext:** Stores the authenticated user information for the session.

---

5. Important Points

- **Order matters:** Filters are executed **in sequence**.
- **Short-circuiting:** If a request fails a filter (e.g., invalid credentials), remaining filters are **not executed**.
- **Custom Filters:** You can add your own filter in the chain for **logging, auditing, or custom security logic**.
- **Declarative Security:** URL-based access rules are typically configured in HttpSecurity:

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

2. Authentication Basics

1. What is Authentication?

- **Authentication** is the process of **verifying the identity of a user** trying to access the application.
- It answers the question: **"Who are you?"**
- In Spring Security, authentication is **mandatory before any authorization** can be applied.

---

2. How Authentication Works in Spring Security

1. **User sends credentials** (username/password) via a login form or API request.
2. The **AuthenticationManager** receives the authentication request.
3. The **AuthenticationManager** uses a **UserDetailsService** to:
    - Load the user data from a database, LDAP, or in-memory store.
    - Retrieve **username, password, and authorities/roles**.
4. The **PasswordEncoder** verifies the **password matches** the stored hash.
5. If credentials are correct:
    - The **user is authenticated**.
    - A **SecurityContext** is created to store authentication info.
6. If credentials are invalid:
    - Authentication **fails**, and the request is denied.

---

3. Key Components in Spring Security Authentication

Component	Purpose
AuthenticationManager	Core interface that handles authentication requests
AuthenticationProvider	Contains the logic to validate credentials (can be multiple providers)
UserDetailsService	Loads user-specific data (username, password, roles)
UserDetails	Represents authenticated user's information (username, password, authorities)
PasswordEncoder	Handles password hashing and verification
SecurityContext	Stores authentication details for the current session/request

---

4. Common Authentication Methods

1. **Form-Based Authentication**
    - Traditional login page with username and password.
    - Spring Security automatically provides a **default login page**, but it can be customized.
2. **HTTP Basic Authentication**
    - Credentials sent in HTTP headers.
    - Simple but less secure unless used with HTTPS.
3. **Token-Based Authentication** (JWT)
    - Credentials verified once, then a **token** is issued.
    - Token is sent with each request for stateless authentication.
4. **LDAP Authentication**

- Uses **Lightweight Directory Access Protocol** to authenticate users against a directory service.

5. **OAuth2 / OpenID Connect**
   - Delegates authentication to **third-party providers** (Google, Facebook, GitHub).
   - Popular for **SSO (Single Sign-On)**.

```
Client Login Request (username/password)
           |
   UsernamePasswordAuthenticationFilter
           |
   AuthenticationManager (validates credentials)
           |
   UserDetailsService --> UserDetails
           |
   PasswordEncoder verifies password
           |
     Success --> SecurityContext updated
     Failure --> Access Denied / Error
```

---

6. Important Points

- **Authentication happens first**, before any authorization rules are checked.
- **SecurityContextHolder** holds authentication information for the current request.
- **PasswordEncoder** is crucial for security; **never store plain-text passwords**.
- Spring Security allows **multiple authentication providers** (e.g., database + LDAP).

Authentication Components in Spring Security

1. UsernamePasswordAuthenticationFilter

- **Purpose:** Handles **form-based login** in Spring Security.
- **How it works:**
  1. Intercepts **HTTP POST requests** to /login (by default).
  2. Extracts **username** and **password** from the request parameters.
  3. Creates an **Authentication token** (UsernamePasswordAuthenticationToken).
  4. Passes the token to **AuthenticationManager** for authentication.
  5. On **success**, the user is authenticated, and the **SecurityContext** is updated.
  6. On **failure**, redirects to a failure page or returns an error.
- **Customization Options:**
  - Change **login URL**:

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

```
        .usernameParameter("user")
        .passwordParameter("pass");
```

- Add **success/failure handlers**:

```
http.formLogin()
    .successHandler(customSuccessHandler)
    .failureHandler(customFailureHandler);
```

---

2. UserDetails and UserDetailsService

a) UserDetails

- Represents **user-specific information** needed for authentication and authorization.
- Key attributes:
  - username – user identity
  - password – encrypted password
  - authorities – roles or permissions granted to the user
  - accountNonExpired, accountNonLocked, credentialsNonExpired, enabled – account status flags
- **Example Implementation:**

```
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import java.util.Collection;

public class MyUserDetails implements UserDetails {

    private String username;
    private String password;
    private Collection<? extends GrantedAuthority> authorities;

    public MyUserDetails(String username, String password, Collection<? extends
GrantedAuthority> authorities) {
        this.username = username;
        this.password = password;
        this.authorities = authorities;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
```

```
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }
}
```

b) UserDetailsService

- Interface responsible for **loading user-specific data**.
- Used by Spring Security during **authentication**.
- Method to implement:

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

- Example:

```
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class MyUserDetailsService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        // Fetch user from database (example hardcoded)
        if(username.equals("admin")) {
```

```
            return new MyUserDetails("admin", "$2a$10$encryptedpassword...", List.of(() ->
"ROLE_ADMIN"));
        } else {
            throw new UsernameNotFoundException("User not found");
        }
    }
}
```

---

3. Password Encoding (BCryptPasswordEncoder)

Why Password Encoding?

- Storing **plain-text passwords is insecure**.
- Password encoding **hashes the password** so even if the database is leaked, passwords are safe.
- Spring Security provides **PasswordEncoder** interface for this purpose.

BCryptPasswordEncoder

- Uses **BCrypt hashing algorithm** with a **salt** to prevent rainbow table attacks.
- Recommended for modern Spring Security applications.

Usage

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

PasswordEncoder encoder = new BCryptPasswordEncoder();
String rawPassword = "myPassword";
String encodedPassword = encoder.encode(rawPassword);

System.out.println("Encoded Password: " + encodedPassword);

// Verify password
boolean matches = encoder.matches(rawPassword, encodedPassword);
System.out.println("Password matches: " + matches);
```

- **Integration in Spring Security:**

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

Important: Always encode passwords when saving users to the database.

3. Authorization Basics

1. What is Authorization?

- **Authorization** is the process of **determining what an authenticated user is allowed to do** in the application.
- It answers the question: **"What resources or actions can this user access?"**
- Requires the user to be **authenticated first**.
- Ensures that **users with insufficient privileges cannot access restricted resources**.

---

2. How Authorization Works in Spring Security

1. A user sends a request after **authentication**.
2. Spring Security checks the **user's roles/authorities** stored in the SecurityContext.
3. The **AccessDecisionManager** evaluates whether the user has permission to access the requested resource.
4. If access is granted, the request reaches the **controller or resource**.
5. If access is denied, Spring Security returns:
    - HTTP 403 Forbidden for REST APIs
    - Redirects to an **access-denied page** for web applications.

---

3. Key Concepts

Term	Description
Role	A high-level grouping of permissions (e.g., ROLE_ADMIN, ROLE_USER).
Authority/Privilege	Fine-grained permission granted to a user (e.g., READ_PRIVILEGE, WRITE_PRIVILEGE).
GrantedAuthority	Spring Security object representing roles/authorities assigned to a user.
AccessDecisionManager	Evaluates if the authenticated user has permission to access a resource.
SecurityContext	Holds authentication and authority information for the current session.

---

4. Methods of Authorization in Spring Security

a) URL-Based Authorization

- Define access rules for HTTP endpoints in HttpSecurity.

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

b) Method-Level Authorization

- Use **annotations** to protect methods at the service or controller level.

1. **@Secured**

```
@Secured("ROLE_ADMIN")
public void deleteUser(Long id) {
   // Only ADMIN can access
}
```

2. **@PreAuthorize / @PostAuthorize**

```
@PreAuthorize("hasRole('ADMIN') and #id != null")
public void updateUser(Long id) {
   // Access granted only to ADMIN
}
```

- **Enable method-level security** in configuration:

```
@EnableMethodSecurity
@Configuration
public class SecurityConfig {
   // Security configurations
}
```

c) Expression-Based Access Control

- Allows **fine-grained control** with SpEL expressions:

```
@PreAuthorize("hasRole('ADMIN') or (hasRole('USER') and #id ==
authentication.principal.id)")
```

```
Authenticated User Request
        |
   SecurityContext (roles/authorities)
        |
   AccessDecisionManager
        |
   Authorized? --> Yes --> Controller/Resource
               --> No  --> Access Denied (403)
```

6. Important Points

- Authorization is **always after authentication**.
- Can be applied at **URL level** or **method level**.
- **Roles** are coarse-grained; **authorities/permissions** are fine-grained.
- Spring Security supports **expression-based access control** for complex scenarios.
- Unauthorized access results in **403 Forbidden** or custom access-denied handling.

Role-Based Authorization & Method-Level Security

1. Role-Based Authorization

a) What is Role-Based Authorization?

- Grants access to resources based on the **user's role** (e.g., ADMIN, USER, MANAGER).
- Roles are **coarse-grained** access controls, representing **groups of permissions**.
- Spring Security maps roles to GrantedAuthority objects internally.

---

b) Common Spring Security Methods

Method	Description	Example
hasRole('ROLE_NAME')	Checks if the user has a specific role.	.requestMatchers("/admin/**").hasRole("ADMIN")
hasAuthority('AUTHORITY_NAME')	Checks if the user has a specific authority/permi	.requestMatchers("/edit/**").hasAuthority("EDIT_PRIVILEGE")

Method	Description	Example
	ssion.	

Note:

- hasRole automatically prefixes ROLE_ internally. So hasRole("ADMIN") checks for ROLE_ADMIN.
- hasAuthority checks for **exact authority name**, no prefix added.

---

c) Example: URL-Based Role Authorization

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
         .authorizeHttpRequests()
         .requestMatchers("/admin/**").hasRole("ADMIN")
         .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
         .requestMatchers("/read/**").hasAuthority("READ_PRIVILEGE")
         .anyRequest().authenticated()
         .and()
         .formLogin();
      return http.build();
   }
}
```

- /admin/** → only accessible to ADMIN role
- /user/** → accessible to USER or ADMIN roles
- /read/** → accessible to users with READ_PRIVILEGE authority

---

2. Method-Level Security

a) What is Method-Level Security?

- Protects **service or controller methods** using annotations.
- Works **on top of authentication and role-based rules**.
- Useful for **fine-grained access control** beyond URL-level security.

---

b) Common Annotations

Annotation	Description
@Secured	Checks for roles; simple and straightforward.
@PreAuthorize	Checks roles/authorities **before method execution**; supports **SpEL expressions**.
@PostAuthorize	Checks roles/authorities **after method execution**; useful for return-value based checks.

---

c) Enabling Method-Level Security

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {
   // Other security configurations
}
```

---

d) Examples

@Secured

```
import org.springframework.security.access.annotation.Secured;

@Service
public class UserService {

   @Secured("ROLE_ADMIN")
   public void deleteUser(Long id) {
      // Only ADMIN can delete users
   }

   @Secured({"ROLE_ADMIN", "ROLE_USER"})
   public void viewProfile() {
      // Accessible to ADMIN and USER roles
   }
}
```

@PreAuthorize

```
import org.springframework.security.access.prepost.PreAuthorize;

@Service
public class DocumentService {

   @PreAuthorize("hasRole('ADMIN')")
   public void createDocument() {
      // Only ADMIN can create documents
   }

   @PreAuthorize("hasAuthority('READ_PRIVILEGE') and #docId != null")
   public void readDocument(Long docId) {
      // User must have READ_PRIVILEGE and docId must not be null
   }

   @PreAuthorize("hasRole('ADMIN') or (hasRole('USER') and #userId ==
authentication.principal.id)")
   public void updateProfile(Long userId) {
      // ADMIN can update anyone, USER can update their own profile
   }
}
```

---

e) Notes & Best Practices

- **Use @PreAuthorize** for **complex access rules** with SpEL expressions.
- **Use @Secured** for **simple role checks**.
- Combine **URL-based security** and **method-level security** for full protection.
- Ensure that **method-level security is enabled** via @EnableMethodSecurity.

4. Configuration Basics in Spring Security

1. What is Configuration in Spring Security?

- Configuration determines **how Spring Security behaves** in your application.
- It defines:
  - **Authentication mechanism** (in-memory, database, LDAP, OAuth2)
  - **Authorization rules** (roles, URL access, method-level security)
  - **Security filters and customizations**
- Spring Security supports **two main configuration approaches**:

1. **Java Configuration (recommended)**
2. **XML Configuration (older approach, rarely used now)**

---

2. Java-Based Configuration

a) SecurityFilterChain Bean

- Spring Security 5.7+ encourages **component-based configuration** with SecurityFilterChain.
- Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
         .authorizeHttpRequests()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
            .anyRequest().authenticated()
         .and()
         .formLogin()
            .loginPage("/login")
            .permitAll()
         .and()
         .logout()
            .permitAll();
      return http.build();
   }

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

- **Key Points:**
  - authorizeHttpRequests() → define URL-based access rules
  - formLogin() → customize login page
  - logout() → customize logout behavior
  - PasswordEncoder → encode passwords

---

b) In-Memory Authentication

- Useful for testing or simple applications.

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
   UserDetails user = User.withUsername("user")
      .password(passwordEncoder().encode("password"))
      .roles("USER")
      .build();

   UserDetails admin = User.withUsername("admin")
      .password(passwordEncoder().encode("adminpass"))
      .roles("ADMIN")
      .build();

   return new InMemoryUserDetailsManager(user, admin);
}
```

---

c) Database Authentication (JDBC)

```
@Autowired
private DataSource dataSource;

@Bean
public JdbcUserDetailsManager userDetailsService() {
   JdbcUserDetailsManager manager = new JdbcUserDetailsManager(dataSource);
   manager.setUsersByUsernameQuery("SELECT username, password, enabled FROM users
WHERE username=?");
   manager.setAuthoritiesByUsernameQuery("SELECT username, authority FROM authorities
WHERE username=?");
   return manager;
}
```

- JdbcUserDetailsManager loads users and roles from **database tables**.

---

3. XML-Based Configuration (Legacy)

```
<http auto-config="true" use-expressions="true">
   <intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN')" />
   <intercept-url pattern="/user/**" access="hasAnyRole('ROLE_USER','ROLE_ADMIN')" />
   <form-login login-page="/login" default-target-url="/home" />
   <logout logout-success-url="/login?logout"/>
</http>

<authentication-manager>
   <authentication-provider>
      <user-service>
```

```
        <user name="admin" password="{noop}adminpass" authorities="ROLE_ADMIN"/>
        <user name="user" password="{noop}userpass" authorities="ROLE_USER"/>
      </user-service>
   </authentication-provider>
</authentication-manager>
```

Note: Modern Spring Boot projects rarely use XML; **Java-based configuration** is preferred.

---

4. Key Configuration Points

- **Enable Security:** @EnableWebSecurity activates Spring Security.
- **Password Encoding:** Always use BCryptPasswordEncoder or other secure encoders.
- **Method-Level Security:** Use @EnableMethodSecurity to enable annotations like @PreAuthorize and @Secured.
- **Filter Chain Customization:** You can **add or remove filters** in SecurityFilterChain.
- **Default Login Page:** Spring Boot provides a default login page if formLogin() is enabled and no custom login page is configured.

---

5. Best Practices

1. Use **Java configuration** over XML for simplicity and maintainability.
2. Always **encode passwords** using a secure encoder.
3. Combine **URL-based + method-level security** for full protection.
4. Keep **roles and authorities organized** to avoid security loopholes.
5. Use **custom login and access-denied pages** for better UX.
6. Use **environment-specific security** (e.g., disable security for dev, enable for prod).

Spring Security Configuration: Advanced Concepts

---

1. SecurityFilterChain vs WebSecurityConfigurerAdapter

Feature	WebSecurityConfigurerAdapter	SecurityFilterChain
Introduction	Old approach to configure Spring Security	Modern recommended approach (Spring Security 5.7+)
Annotation	@EnableWebSecurity + extend WebSecurityConfigurerAdapter	@EnableWebSecurity + define @Bean of type SecurityFilterChain

Feature	WebSecurityConfigurerAdapter	SecurityFilterChain
Method Override	Override configure(HttpSecurity http) and configure(AuthenticationManagerBuilder auth)	Configure via HttpSecurity inside a @Bean method
Deprecation	Deprecated in Spring Security 5.7+	Recommended for all new projects
Flexibility	Less flexible for multiple filter chains	Supports multiple SecurityFilterChain beans for different request matchers
Example	java public class SecurityConfig extends WebSecurityConfigurerAdapter { ... }	java @Bean public SecurityFilterChain filterChain(HttpSecurity http) throws Exception { ... }

Note: Modern Spring Security encourages **bean-based configuration** using SecurityFilterChain rather than inheritance.

---

2. HttpSecurity Configuration

a) URL Authorization

- Define which roles or authorities can access specific endpoints:

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

b) Form-Based Login

- Enable and customize login form:

```
http.formLogin()
   .loginPage("/login")          // custom login page
   .defaultSuccessUrl("/home")   // redirect after successful login
   .failureUrl("/login?error")   // redirect on failure
   .permitAll();                 // allow all to access login page
```

c) HTTP Basic Authentication

- Simple authentication for APIs:

```
http.httpBasic();
```

d) Logout Configuration

```
http.logout()
   .logoutUrl("/logout")
   .logoutSuccessUrl("/login?logout")
   .invalidateHttpSession(true)
   .deleteCookies("JSESSIONID");
```

e) CSRF and Other Configurations

```
http.csrf().disable(); // disable CSRF for APIs
http.rememberMe().key("uniqueAndSecret");
```

---

3. In-Memory Authentication vs JDBC Authentication

Feature	In-Memory Authentication	JDBC Authentication
Purpose	For testing or simple apps	For production, real user database
Data Storage	Hardcoded in memory	Stored in database tables (users, authorities)
UserDetailsService	InMemoryUserDetailsManager	JdbcUserDetailsManager
Password Encoder	PasswordEncoder still required	PasswordEncoder still required
Example	java UserDetails user = User.withUsername("user") .password(passwordEncoder().encode("pass")) .roles("USER") .build(); return new InMemoryUserDetailsManager(user);	java @Autowired DataSource ds; JdbcUserDetailsManager manager = new JdbcUserDetailsManager(ds); manager.setUsersByUsernameQuery("SELECT username,password,enabled FROM users WHERE username=?"); manager.setAuthoritiesByUsernameQuery("SELECT username,authority FROM authorities WHERE username=?"); return

Feature	In-Memory Authentication	JDBC Authentication
		manager;
Pros	Simple, fast, no DB required	Scalable, secure, persistent user storage
Cons	Not suitable for production	Requires database setup and maintenance

---

4. Key Points

- **SecurityFilterChain** replaces the deprecated WebSecurityConfigurerAdapter.
- HttpSecurity configures **URL-based authorization, login, logout, HTTP Basic, CSRF, remember-me**, and more.
- **In-memory authentication** is **temporary and simple**, **JDBC authentication** is production-ready.
- Always **encode passwords** even in in-memory authentication.
- Modern applications can also combine **JWT or OAuth2** with these authentication mechanisms.

5. Session Management in Spring Security

Spring Security provides powerful session management features that help control **how user sessions are created, maintained, and terminated**. Proper session handling prevents attacks like **session fixation** and helps manage concurrent logins.

---

1. Session Fixation Protection

What is Session Fixation?

- **Attack where a malicious user forces a victim to use a known session ID.**
- After login, the attacker hijacks the session using the same ID.

Spring Security Solution

- On successful authentication, Spring Security can **create a new session ID** (invalidate old one).
- Configurable via:

```
http.sessionManagement()
   .sessionFixation().migrateSession();   // default option
```

Options

- .none() → No session fixation protection.
- .newSession() → Create a brand-new session, discarding old attributes.
- .migrateSession() → Create a new session, but migrate existing attributes (default, recommended).
- .changeSessionId() → Only change the session ID (servlet 3.1+).

 Best Practice: Use **migrateSession** or **changeSessionId**.

---

2. Concurrent Session Control

Problem

- Same user credentials used to log in from multiple devices (or attackers).
- Could lead to **account sharing** or **hijacking risks**.

Spring Security Solution

- Restrict the **maximum number of active sessions per user**:

```
http.sessionManagement()
   .maximumSessions(1)                  // only 1 active session per user
   .maxSessionsPreventsLogin(true);          // new login blocked if max reached
```

Options

- maximumSessions(n) → sets the number of concurrent sessions allowed.
- maxSessionsPreventsLogin(true) → prevents a new login if limit is reached.
- maxSessionsPreventsLogin(false) → allows new login but invalidates the old session.

 Best Practice: Set maximumSessions(1) for sensitive apps (like banking).

---

3. Stateless vs Stateful Authentication

a) Stateful Authentication (Session-Based)

- Server creates a **session object** after login.
- Session stored in server memory (or DB, Redis).
- Each request → client sends **session ID** (cookie).
- Server validates session ID to authenticate user.

Pros:

- Easy to implement.
- Works well for traditional web apps.

Cons:

- Not scalable (server must store sessions).
- Requires sticky sessions in distributed environments.

---

b) Stateless Authentication (Token-Based, e.g., JWT)

- No session maintained on the server.
- After login, server issues a **token** (e.g., JWT).
- Each request → client sends **token** (in header, usually Authorization: Bearer <token>).
- Server validates token **without storing session state**.

Pros:

- Scales well (no server-side session storage).
- Perfect for REST APIs and microservices.

Cons:

- Token revocation is hard (until token expiry).
- Requires strong token protection (HTTPS, short expiry, refresh tokens).

Spring Security Configurations

Stateful (default):

```
http.sessionManagement()
   .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED);
```

Stateless (for REST APIs with JWT):

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

SessionCreationPolicy Options:

- ALWAYS → always create a session.
- IF_REQUIRED → create session only if needed (default).
- NEVER → never create session, but use existing one if present.

- STATELESS → no session created or used.

6. CSRF Protection

1. What is CSRF?

- **CSRF (Cross-Site Request Forgery)** is an attack where a **malicious website tricks a logged-in user** into performing actions on another website without their consent.
- Example:
  - User is logged in to a banking website.
  - Malicious website sends a request like POST /transfer?amount=1000&to=attacker.
  - Bank executes request because user is authenticated, **without user knowledge**.

---

2. How Spring Security Protects Against CSRF

- Spring Security **enables CSRF protection by default** for **state-changing requests** (POST, PUT, DELETE, PATCH).
- It works by:
  1. Generating a **CSRF token** per session.
  2. Sending the token in forms or request headers.
  3. Validating the token for each **state-changing request**.
- Only requests with **valid CSRF token** are accepted.

---

3. Enabling/Disabling CSRF

a) Enabling CSRF (default behavior)
http.csrf().enable(); // Explicitly enable if needed

b) Disabling CSRF
http.csrf().disable();

Important:

- **Do NOT disable CSRF** for traditional web applications.
- CSRF can safely be disabled for **stateless REST APIs** because tokens or JWTs already prevent cross-site attacks.

---

4. When to Use CSRF Tokens

- CSRF protection is required **for any state-changing request** made via a **browser**:
  - **POST** → form submissions, login forms, create operations
  - **PUT/PATCH** → update operations
  - **DELETE** → delete operations
- **Safe requests** like GET, HEAD, OPTIONS, TRACE do **not require CSRF**.

Example: HTML Form with CSRF Token

```
<form action="/transfer" method="post">
   <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
   <input type="text" name="amount" />
   <button type="submit">Transfer</button>
</form>
```

- In Spring Boot with Thymeleaf, ${_csrf.token} automatically provides the CSRF token.

---

5. CSRF with REST APIs

- Typically, REST APIs are **stateless** and do not use sessions.
- Use **JWT or other token-based authentication**, and **disable CSRF**:

```
http.csrf().disable()
   .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
```

7. CORS (Cross-Origin Resource Sharing)

1. What is CORS?

- **CORS** is a security feature implemented by browsers to **restrict web pages from making requests to a different domain** than the one that served the web page.
- Example:
  - Your frontend app runs at http://localhost:3000.
  - Your Spring Boot API runs at http://localhost:8080.
  - A request from frontend → backend is **cross-origin**, and by default **blocked by the browser**.

---

2. Why Configure CORS in Spring Security

- APIs are often accessed by **frontend apps running on different domains**.
- Spring Security **blocks cross-origin requests by default** for security.
- Configuring CORS allows the server to:
  - Define **allowed origins**
  - Specify allowed **methods (GET, POST, etc.)**

- Allow specific **headers** and credentials

---

3. Configuring CORS in Spring Security

a) Using HttpSecurity Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

import java.util.Arrays;

@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
         .cors()  // enable CORS
         .and()
         .authorizeHttpRequests()
         .requestMatchers("/api/**").authenticated()
         .anyRequest().permitAll()
         .and()
         .httpBasic();

      return http.build();
   }

   @Bean
   public CorsFilter corsFilter() {
      CorsConfiguration config = new CorsConfiguration();
      config.setAllowedOrigins(Arrays.asList("http://localhost:3000")); // frontend URL

config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE","OPTIONS"));
      config.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"));
      config.setAllowCredentials(true); // allow cookies/auth headers

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

```
        return new CorsFilter(source);
    }
}
```

b) Key Points

- http.cors() → enables Spring Security CORS handling.
- CorsFilter → defines which **origins, methods, headers** are allowed.
- setAllowCredentials(true) → required if you send cookies or Authorization headers.
- Multiple endpoints can share the same CORS configuration using /**.

---

4. Common Issues

1. **Blocked by browser**: Usually because Access-Control-Allow-Origin header is missing.
2. **Credentials not allowed**: Set setAllowCredentials(true) if sending cookies or Authorization header.
3. **Preflight requests**: Browsers send an OPTIONS request before POST/PUT/DELETE. Ensure OPTIONS is included in allowed methods.

8. Custom Authentication

Spring Security provides default authentication mechanisms (form login, basic auth), but most real-world applications require **custom authentication logic** to fit business requirements.

---

1. Custom Login Pages

a) Why Custom Login Pages

- Default Spring Security login page is **generic**.
- Custom login pages allow **branding, additional fields, and custom error handling**.

b) Configuring a Custom Login Page

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests()
                .requestMatchers("/admin/**").hasRole("ADMIN")
```

```
            .anyRequest().authenticated()
        .and()
        .formLogin()
            .loginPage("/login")          // custom login URL
            .loginProcessingUrl("/perform_login") // URL for form POST
            .defaultSuccessUrl("/home", true)
            .failureUrl("/login?error=true")
            .permitAll()
        .and()
        .logout()
            .logoutUrl("/perform_logout")
            .logoutSuccessUrl("/login?logout")
            .permitAll();

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

- **Notes:**
  - /login → GET request returns **login form HTML**.
  - /perform_login → POST endpoint handled by Spring Security automatically.
  - Custom success and failure URLs improve user experience.

---

2. Custom AuthenticationProvider

a) Why Use a Custom AuthenticationProvider

- Default DaoAuthenticationProvider works with UserDetailsService.
- Use a custom provider when:
  - Authentication is done via **external APIs**.
  - Users are stored in **non-standard DB** or **LDAP**.
  - You need **multi-factor authentication** or custom validation logic.

b) Implementing Custom AuthenticationProvider

```
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.stereotype.Component;
import java.util.List;

@Component
```

```
public class CustomAuthenticationProvider implements AuthenticationProvider {

   @Override
   public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
      String username = authentication.getName();
      String password = authentication.getCredentials().toString();

      // Custom authentication logic (e.g., check DB or external API)
      if ("admin".equals(username) && "password".equals(password)) {
         List<GrantedAuthority> authorities = List.of(() -> "ROLE_ADMIN");
         return new UsernamePasswordAuthenticationToken(username, password, authorities);
      } else {
         return null; // Authentication failed
      }
   }

   @Override
   public boolean supports(Class<?> authentication) {
      return authentication.equals(UsernamePasswordAuthenticationToken.class);
   }
}
```

c) Integrate with SecurityFilterChain

```
@Autowired
private CustomAuthenticationProvider authProvider;

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http.authenticationProvider(authProvider);
   // other configurations...
   return http.build();
}
```

---

3. Custom Filters in the Security Filter Chain

a) Why Custom Filters

- Modify or inspect **requests/responses** before or after authentication.
- Common use cases:
    - Logging
    - JWT token verification
    - IP filtering or rate-limiting

b) Adding a Custom Filter

```
import org.springframework.web.filter.OncePerRequestFilter;
import javax.servlet.FilterChain;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component
public class CustomFilter extends OncePerRequestFilter {

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

      String header = request.getHeader("X-Custom-Header");
      if (header != null) {
         System.out.println("Custom Header Found: " + header);
      }

      filterChain.doFilter(request, response); // continue filter chain
   }
}
```

c) Register Filter in Security Chain

```
http.addFilterBefore(customFilter, UsernamePasswordAuthenticationFilter.class);
```

- addFilterBefore → adds the filter **before a specific filter**.
- addFilterAfter → adds the filter **after a specific filter**.

9. Remember-Me Authentication

1. What is Remember-Me Authentication?

- **Remember-Me** allows a user to remain **logged in even after closing the browser**.
- It avoids frequent logins for **trusted devices**.
- Typically implemented using **cookies** or **persistent tokens**.

---

2. Cookie-Based Remember-Me

a) How it Works

1. User logs in with **"Remember Me" checkbox**.
2. Spring Security generates a **special token** and stores it in a **cookie** on the browser.
3. On subsequent visits, the cookie is sent automatically.
4. Spring Security **validates the cookie** and authenticates the user **without entering credentials**.

b) Configuration

```
http
   .rememberMe()
      .key("uniqueAndSecret")          // secret key for hashing
      .rememberMeParameter("remember-me") // form checkbox parameter
      .tokenValiditySeconds(7 * 24 * 60 * 60); // 7 days
```

c) Notes

- Default implementation is **in-memory token storage**, which is **lost on server restart**.
- Good for **simple use-cases**, **non-critical applications**.

---

3. Persistent Token-Based Remember-Me

a) How it Works

- Tokens are stored in a **database** for persistence.
- Each token has:
    - **Series** → unique identifier for the user
    - **Token Value** → generated hash
    - **Username**
    - **Last Used Date**
- Provides **secure, long-term remember-me functionality**, surviving **server restarts**.

b) Database Setup

```
CREATE TABLE persistent_logins (
   username VARCHAR(64) NOT NULL,
   series VARCHAR(64) PRIMARY KEY,
   token VARCHAR(64) NOT NULL,
   last_used TIMESTAMP NOT NULL
);
```

c) Configuration

```
@Autowired
private DataSource dataSource;
```

```
http
   .rememberMe()
      .tokenRepository(persistentTokenRepository())
      .tokenValiditySeconds(14 * 24 * 60 * 60) // 14 days
      .userDetailsService(userDetailsService());

@Bean
public PersistentTokenRepository persistentTokenRepository() {
   JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();
   tokenRepository.setDataSource(dataSource);
   // tokenRepository.setCreateTableOnStartup(true); // Uncomment to create table automatically
   return tokenRepository;
}
```

d) Notes

- Requires **userDetailsService** to load users.
- Tokens in DB ensure **security and persistence**.
- Each login generates a **new token**.

---

4. Remember-Me in Login Form

```
<form action="/login" method="post">
   <input type="text" name="username" />
   <input type="password" name="password" />
   <input type="checkbox" name="remember-me" /> Remember Me
   <button type="submit">Login</button>
</form>
```

- Checkbox name should match .rememberMeParameter() in configuration.
- If selected, Spring Security issues a **remember-me token**.

10. Exception Handling in Spring Security

Spring Security provides mechanisms to **handle authentication and authorization errors** gracefully.

---

1. Overview

- **Authentication errors** → When a user is **not logged in** or provides invalid credentials.
- **Authorization errors** → When a logged-in user **does not have access** to a resource.
- Spring Security handles these via:

- **AuthenticationEntryPoint** → Handles **unauthenticated** requests.
- **AccessDeniedHandler** → Handles **unauthorized** requests (authenticated but insufficient permissions).

---

2. AuthenticationEntryPoint

a) What is it?

- Triggered when a user **tries to access a protected resource without authentication**.
- Responsible for sending a **response indicating login is required**.

b) Default Implementations

- LoginUrlAuthenticationEntryPoint → redirects to login page for form-based authentication.
- Http403ForbiddenEntryPoint → sends 403 Forbidden response (commonly used in REST APIs).

c) Custom AuthenticationEntryPoint Example

```
@Component
public class CustomAuthenticationEntryPoint implements AuthenticationEntryPoint {

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

      response.setContentType("application/json");
      response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // 401
      response.getOutputStream().println("{\"error\": \"You are not authenticated\"}");
   }
}
```

d) Registering in SecurityConfig

```
http.exceptionHandling()
   .authenticationEntryPoint(customAuthenticationEntryPoint);
```

---

3. AccessDeniedHandler

a) What is it?

- Triggered when a user **is authenticated but does not have sufficient privileges** to access a resource.
- Example: USER role trying to access /admin/**.

b) Default Implementation

- AccessDeniedHandlerImpl → redirects to /access-denied page (web apps).

c) Custom AccessDeniedHandler Example

```
@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {

   @Override
   public void handle(HttpServletRequest request, HttpServletResponse response,
                      AccessDeniedException accessDeniedException) throws IOException {

      response.setContentType("application/json");
      response.setStatus(HttpServletResponse.SC_FORBIDDEN); // 403
      response.getOutputStream().println("{\"error\": \"Access Denied\"}");
   }
}
```

d) Registering in SecurityConfig

```
http.exceptionHandling()
   .accessDeniedHandler(customAccessDeniedHandler);
```

---

4. Key Points

Exception Type	Triggered When	HTTP Status	Default Behavior
AuthenticationException	Unauthenticated user accesses protected resource	401 Unauthorized	Redirect to login page
AccessDeniedException	Authenticated user lacks authority/role	403 Forbidden	Redirect to access-denied page

- For **REST APIs**, customize both handlers to **return JSON responses** instead of redirects.
- For **web apps**, you can redirect users to **custom error pages**.

11. JWT (JSON Web Token) Authentication

JWT is widely used in modern **REST APIs** for **stateless, token-based authentication**.

---

1. Stateless Authentication with JWT

- Unlike session-based authentication, **server does not store user session**.
- After successful login:
    1. Server generates a **JWT token** containing user information and roles.
    2. Client stores the token (usually in **localStorage** or **Authorization header**).
    3. Each request includes the token in Authorization: Bearer <token>.
    4. Server validates token and authenticates user without sessions.

Advantages:

- Scalable (no server-side session storage)
- Works for **microservices and mobile apps**
- Self-contained (token carries user data)

---

2. Creating and Validating JWT Tokens

a) Dependencies

- Use io.jsonwebtoken (jjwt) or any JWT library.

```
<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>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-jackson</artifactId>
   <version>0.11.5</version>
</dependency>
```

b) JWT Utility Class

```
import io.jsonwebtoken.*;
import org.springframework.stereotype.Component;
import java.util.Date;
import java.util.function.Function;

@Component
public class JwtUtil {

   private final String SECRET_KEY = "mysecretkey12345"; // keep secret in config

   // Generate token
   public String generateToken(String username) {
      return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10)) // 10
hours
            .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
            .compact();
   }

   // Extract username
   public String extractUsername(String token) {
      return extractClaim(token, Claims::getSubject);
   }

   // Extract claim
   public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
      Claims claims =
Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
      return claimsResolver.apply(claims);
   }

   // Validate token
   public boolean validateToken(String token, String username) {
      return username.equals(extractUsername(token)) && !isTokenExpired(token);
   }

   private boolean isTokenExpired(String token) {
      return extractClaim(token, Claims::getExpiration).before(new Date());
   }
}
```

3. Adding JWT Filter in the Security Filter Chain

a) JWT Filter

```
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component
public class JwtFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;
    private final MyUserDetailsService userDetailsService;

    public JwtFilter(JwtUtil jwtUtil, MyUserDetailsService userDetailsService) {
        this.jwtUtil = jwtUtil;
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain chain)
            throws java.io.IOException, javax.servlet.ServletException {

        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) {
            var userDetails = userDetailsService.loadUserByUsername(username);
            if (jwtUtil.validateToken(jwt, userDetails.getUsername())) {
                UsernamePasswordAuthenticationToken authToken =
                        new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
```

```
            authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
    }

    chain.doFilter(request, response);
  }
}
```

b) Register JWT Filter

```
http
  .csrf().disable()
  .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
  .and()
  .authorizeHttpRequests()
    .requestMatchers("/authenticate").permitAll()
    .anyRequest().authenticated()
  .and()
  .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);
```

---

4. Authentication Controller

```
@RestController
public class AuthController {

  private final JwtUtil jwtUtil;
  private final AuthenticationManager authenticationManager;

  public AuthController(JwtUtil jwtUtil, AuthenticationManager authenticationManager) {
    this.jwtUtil = jwtUtil;
    this.authenticationManager = authenticationManager;
  }

  @PostMapping("/authenticate")
  public ResponseEntity<?> createAuthenticationToken(@RequestBody AuthRequest
authRequest) throws Exception {
    authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(authRequest.getUsername(),
authRequest.getPassword())
    );
    final String token = jwtUtil.generateToken(authRequest.getUsername());
    return ResponseEntity.ok(new AuthResponse(token));
  }
}
```

12. OAuth2 and OpenID Connect (OIDC)

OAuth2 and OpenID Connect are widely used for **third-party authentication and authorization**, enabling users to log in using **Google, GitHub, Facebook, etc.**.

---

1. Key Concepts

Term	Description
OAuth2	Protocol for **authorization**, allowing a client app to access resources on behalf of a user.
OpenID Connect (OIDC)	Layer on top of OAuth2 for **authentication**, provides user identity info.
Resource Server	The server hosting **protected resources** (APIs).
Authorization Server	Server responsible for **authenticating users and issuing tokens** (JWT/OAuth2 tokens).

---

2. OAuth2 Login in Spring Security

a) 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-security</artifactId>
</dependency>
```

b) Application Properties (Example: Google Login)

```
spring.security.oauth2.client.registration.google.client-id=YOUR_CLIENT_ID
spring.security.oauth2.client.registration.google.client-secret=YOUR_CLIENT_SECRET
spring.security.oauth2.client.registration.google.scope=openid,profile,email
spring.security.oauth2.client.registration.google.redirect-
uri={baseUrl}/login/oauth2/code/{registrationId}
```

c) Security Configuration

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http
      .authorizeHttpRequests(auth -> auth
         .requestMatchers("/login", "/oauth2/**").permitAll()
         .anyRequest().authenticated()
      )
      .oauth2Login(oauth2 -> oauth2
         .loginPage("/login")         // custom login page
         .defaultSuccessUrl("/home")   // redirect after login
      );

   return http.build();
}
```

- oauth2Login() → Enables OAuth2 login.
- Supports **Google, GitHub, Facebook** out-of-the-box.

---

3. Resource Server vs Authorization Server

Feature	Resource Server	Authorization Server
Purpose	Protect APIs and validate access tokens	Authenticate users & issue tokens (JWT, OAuth2)
Spring Boot Starter	spring-boot-starter-oauth2-resource-server	Custom implementation or Spring Authorization Server
Token Validation	Validates JWT or opaque tokens from Auth Server	Issues JWT/opaque tokens after authentication
Use Case	REST APIs	OAuth2 login, Single Sign-On (SSO)

---

4. JWT-Based OAuth2 Resource Server

- Resource server **validates JWT access tokens** issued by authorization server.
- Configuration:

a) Dependencies

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

```
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

b) Security Configuration

```
@Bean
public SecurityFilterChain resourceFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt() // enable JWT token validation
        );

    return http.build();
}
```

c) Application Properties

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

- jwk-set-uri → URL where the **Authorization Server publishes public keys** for JWT verification.
- Resource server **decodes and validates the JWT** to authorize access.

13. Spring Security with Microservices

Securing **microservices architectures** requires a **centralized authentication system**, token propagation, and API gateway integration.

---

1. Token Relay in Microservices

a) Concept

- In microservices, a client request often passes through **multiple services**.
- **Token relay** ensures the **JWT or OAuth2 access token** is forwarded to all downstream services to maintain authentication and authorization.

b) How it Works

1. Client authenticates with **Auth Server** → receives JWT token.

2. Client calls **API Gateway** with token in Authorization: Bearer <token>.
3. API Gateway forwards token to **downstream microservices**.
4. Each microservice **validates the token** locally (stateless).

c) Implementation

- Spring Security + OAuth2 Resource Server:

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

- Use **RestTemplate or WebClient** to forward token:

```
webClient.get()
   .uri("/serviceB/data")
   .header("Authorization", "Bearer " + token)
   .retrieve()
   .bodyToMono(String.class);
```

---

2. API Gateway Security

a) Role of API Gateway

- Acts as **entry point** for all client requests.
- Can enforce **authentication, rate limiting, and request validation**.
- Centralizes **security logic** to reduce duplication in microservices.

b) Example with Spring Cloud Gateway

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
   http
      .authorizeExchange(exchanges -> exchanges
         .pathMatchers("/public/**").permitAll()
         .anyExchange().authenticated()
      )
      .oauth2ResourceServer(ServerHttpSecurity.OAuth2ResourceServerSpec::jwt);
   return http.build();
```

}

- API Gateway **validates JWT** before forwarding to microservices.
- Optionally, it can **enrich requests** with user roles or authorities.

---

3. Centralized Authentication with Keycloak/Okta

a) Why Centralized Auth

- Microservices shouldn't manage their own authentication.
- A **centralized identity provider** ensures:
  - Single Sign-On (SSO)
  - Standard OAuth2 / OpenID Connect protocols
  - Token issuance, refresh, and revocation

b) Keycloak Integration

1. **Keycloak server** → manages users, roles, and tokens.
2. Each microservice → configured as a **resource server**.
3. Spring Boot Resource Server validates JWT issued by Keycloak:

spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:8080/realms/myrealm/protocol/openid-connect/certs

c) Okta Integration

- Okta acts as **cloud-based authorization server**.
- Steps:
  1. Register microservices as **OAuth2 applications**.
  2. Use Spring Security spring-boot-starter-oauth2-resource-server.
  3. Configure **JWK URI** provided by Okta to validate JWT tokens.

14. Advanced Method Security

Spring Security provides **method-level security annotations** to secure methods based on **roles, permissions, or complex expressions**. Advanced method security uses **SpEL (Spring Expression Language)** and **pre/post filtering**.

---

1. Enabling Method-Level Security

```
@Configuration
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class MethodSecurityConfig {
```

```
    // No additional code needed
}
```

- prePostEnabled = true → Enables @PreAuthorize and @PostAuthorize.
- securedEnabled = true → Enables @Secured annotations.

---

2. SpEL in Security Annotations

- **SpEL** allows **dynamic expressions** based on method parameters or authentication info.
- Common variables in SpEL:
  - authentication → full Authentication object
  - principal → currently logged-in user
  - #id, #username → method parameters

Examples:

a) Using @PreAuthorize

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

@PreAuthorize("#username == authentication.name")
public void updateProfile(String username) { ... }

@PreAuthorize("hasAuthority('WRITE_PRIVILEGE') and #data.owner ==
authentication.name")
public void editData(Data data) { ... }
```

- Checks **user roles, authorities, or dynamic conditions** before method execution.

b) Using @PostAuthorize

```
@PostAuthorize("returnObject.owner == authentication.name")
public Data getDataById(Long id) {
    return dataService.findById(id);
}
```

- Checks the **return value** after method execution.
- Example: Only allow access if the **data owner matches the logged-in user**.

---

3. Pre/Post Filtering

- Allows filtering **collections or lists** based on user roles or attributes.

a) @PreFilter

- Filters **method arguments** before execution.

```
@PreFilter("filterObject.owner == authentication.name")
public void updateMultipleData(List<Data> dataList) {
   // Only items owned by current user will be processed
}
```

b) @PostFilter

- Filters **method return values** after execution.

```
@PostFilter("filterObject.owner == authentication.name")
public List<Data> getAllData() {
   return dataRepository.findAll();
}
```

- Only returns items **owned by the current user**.

---

4. @Secured Annotation

- Simple role-based method security.

```
@Secured("ROLE_ADMIN")
public void deleteUser(Long userId) { ... }

@Secured({"ROLE_USER", "ROLE_ADMIN"})
public void updateUser(Long userId) { ... }
```

- Only checks **roles**, no complex SpEL support.

15. Security Testing in Spring Security

Testing is crucial to ensure that **authentication and authorization** rules are enforced correctly in Spring Security.

---

1. Unit Testing Secured Methods

Spring Security provides **annotations and utilities** to mock authenticated users in unit tests.

a) Using @WithMockUser

- Allows simulating a user with **specific roles or authorities** in tests.
- Example:

```
import org.junit.jupiter.api.Test;
import org.springframework.security.test.context.support.WithMockUser;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class UserServiceTest {

   @Autowired
   private UserService userService;

   @Test
   @WithMockUser(username = "admin", roles = {"ADMIN"})
   public void testAdminAccess() {
      userService.adminOnlyOperation(); // Should pass
   }

   @Test
   @WithMockUser(username = "user", roles = {"USER"})
   public void testAdminAccessDenied() {
      assertThrows(org.springframework.security.access.AccessDeniedException.class,
               () -> userService.adminOnlyOperation());
   }
}
```

Key Points:

- @WithMockUser creates a **mock authentication context**.
- You can specify:
  - username (default: "user")
  - roles (default: "USER")
  - authorities for fine-grained permissions

---

b) Testing Custom Users

```
@WithUserDetails("john") // Loads user from UserDetailsService
@Test
public void testUserSpecificAccess() {
   userService.updateProfile("john"); // Should pass
}
```

- Uses **real user data** from your UserDetailsService.

2. Integration Testing with Security

- Use @SpringBootTest with MockMvc to test **HTTP endpoints** with security applied.

a) Example

```
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.junit.jupiter.api.Test;
import
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
public class SecurityIntegrationTest {

   @Autowired
   private MockMvc mockMvc;

   @Test
   public void accessPublicEndpoint() throws Exception {
      mockMvc.perform(get("/public"))
            .andExpect(status().isOk());
   }

   @Test
   public void accessSecuredEndpointWithoutAuth() throws Exception {
      mockMvc.perform(get("/admin"))
            .andExpect(status().isUnauthorized());
   }

   @Test
   public void accessSecuredEndpointWithAuth() throws Exception {
      mockMvc.perform(get("/admin")
            .with(SecurityMockMvcRequestPostProcessors.user("admin").roles("ADMIN")))
            .andExpect(status().isOk());
   }
}
```

b) Key Points

- SecurityMockMvcRequestPostProcessors.user(...) → mock authenticated user.
- Can test **roles, authorities, and username-specific access**.
- Ensures **security configuration works as expected** in real HTTP calls.

---

3. Common Security Testing Scenarios

1. **Unauthorized access** → endpoints without authentication should return 401.
2. **Forbidden access** → authenticated user with insufficient role should return 403.
3. **Role-based access** → ensure only allowed roles can execute methods or access endpoints.
4. **Token-based security** → verify JWT or OAuth2 token authentication works for protected APIs.
5. **Remember-me / session tests** → validate session persistence and logout.

---

4. Best Practices

- Always test **both allowed and denied cases**.
- Use **mock users** for unit tests (@WithMockUser) and **MockMvc for integration tests**.
- For **JWT / OAuth2**, use real token generation for integration tests.
- Cover **method-level security** and **HTTP endpoint security** in tests.

16. Reactive Spring Security (WebFlux)

Reactive Spring Security is designed for **non-blocking, reactive applications** built using **Spring WebFlux**. It integrates seamlessly with **Mono** and **Flux** types for reactive programming.

---

1. Key Concepts

Feature	Description
Reactive Security Context	ReactiveSecurityContextHolder stores authentication info in a **reactive context**.
Mono/Flux support	Security operations return **Mono<Authentication>** or **Flux<Authentication>** for asynchronous handling.
Stateless nature	Works well with **JWT or token-based authentication**, enabling non-blocking microservices.

---

2. Dependencies

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

---

3. Reactive Authentication

a) Configuring Security in WebFlux

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

   @Bean
   public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
      http
         .csrf().disable()
         .authorizeExchange(exchanges -> exchanges
            .pathMatchers("/public/**").permitAll()
            .anyExchange().authenticated()
         )
         .httpBasic().and()
         .formLogin();
      return http.build();
   }
}
```

- authorizeExchange() → reactive equivalent of authorizeHttpRequests().
- Works with **Mono/Flux** responses.

---

b) Reactive UserDetailsService

```
@Bean
public MapReactiveUserDetailsService userDetailsService() {
   UserDetails user = User.withUsername("user")
                  .password("{noop}password")
                  .roles("USER")
                  .build();
```

```
    return new MapReactiveUserDetailsService(user);
}
```

- Returns **MapReactiveUserDetailsService** instead of UserDetailsService.
- Works **non-blocking** with reactive streams.

---

4. Reactive Authorization

a) Using SecurityContext in Reactive Flows

```
@GetMapping("/profile")
public Mono<String> getProfile() {
    return ReactiveSecurityContextHolder.getContext()
            .map(securityContext -> securityContext.getAuthentication().getName());
}
```

- ReactiveSecurityContextHolder.getContext() → retrieves the current **Authentication** in reactive streams.
- Can perform **role or authority checks** dynamically:

```
.map(ctx -> ctx.getAuthentication().getAuthorities().stream()
              .anyMatch(a -> a.getAuthority().equals("ROLE_ADMIN")))
```

---

b) JWT Authentication in Reactive Apps

- JWT validation is **non-blocking** using AuthenticationWebFilter.

```
@Bean
public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http,
      ReactiveAuthenticationManager authManager, JwtAuthenticationConverter converter) {

    AuthenticationWebFilter jwtFilter = new AuthenticationWebFilter(authManager);
    jwtFilter.setServerAuthenticationConverter(converter);

    http
       .csrf().disable()
       .authorizeExchange()
          .pathMatchers("/public/**").permitAll()
          .anyExchange().authenticated()
       .and()
       .addFilterAt(jwtFilter, SecurityWebFiltersOrder.AUTHENTICATION);

    return http.build();
```

```
}
```

- Works with **Mono<Authentication>** for reactive flows.

---

5. Pre/Post Authorization in Reactive Services

- @PreAuthorize and @PostAuthorize work with reactive methods as well:

```
@PreAuthorize("hasRole('ADMIN')")
public Mono<String> adminEndpoint() {
   return Mono.just("Welcome Admin");
}
```

- Reactive annotations integrate with **reactive security context**.