# Introduction to Spring Security

## Introduction

**Spring Security** is a fundamental component of the Spring ecosystem, designed to provide comprehensive security features for Java-based web applications. It is crucial to safeguard applications from various security threats and ensure that only authorized users can access specific resources.

- **Authentication** is at the core of Spring Security, encompassing verifying user identities. Spring Security supports various authentication mechanisms, including form-based authentication, HTTP Basic authentication, OAuth2, OpenID Connect, JWT, and more. This flexibility allows developers to choose the most suitable method for their application's security needs.

- **Authorization** is another critical aspect of Spring Security, dictating what actions and resources authenticated users can access. It enables fine-grained control over access rights by defining authorization rules based on roles and permissions.

With user details retrieval, filters, and interceptors, Spring Security forms a robust security framework for Java applications. It seamlessly integrates with other Spring components, offering developers the tools to build secure and reliable web applications. Understanding these core concepts is essential for effectively implementing security in Spring-based projects.

## Authentication vs Authorization

Authentication and Authorization are two distinct but closely related concepts in security, often used together to control access to resources in computer systems. Here's a detailed explanation of both:

### Authentication

Authentication is verifying the identity of a user, system, or entity trying to access a particular resource. It answers the question, "*Who are you*?" It ensures that the user or entity is who they claim to be.

1. **Methods of Authentication**: There are several methods of authentication, including
    a. **Password-Based**: The user provides a username and password.

    b. **Biometric**: Authentication based on unique biological traits like fingerprints or retina scans.

    c. **Multi-Factor Authentication (MFA):** Requires multiple forms of authentication, such as a password and a one-time code sent to a mobile device.

d. **Public Key Infrastructure (PKI)**: Involves digital certificates and private/public key pairs.

2. **Authentication Process**: The process usually involves the following steps:
    a. The user provides identification credentials (e.g., username and password).

    b. These credentials are sent to an authentication server.

    c. The server checks the provided credentials against stored records.

    d. If the credentials match, the user is authenticated and granted access.

3. **Purpose**: Authentication ensures that only legitimate users gain access to a system or resource. It prevents unauthorized users from masquerading as authorized users.

## Authorization:

Authorization determines what actions or resources an authenticated user can access. It answers the question, "*What are you allowed to do?*" Authorization controls permissions and privileges.

1. **Access Control**: Authorization involves defining access control rules and policies. This is typically based on a user's roles, group memberships, or specific permissions tied to their identity.

2. **Role-Based Authorization**: Many systems use role-based authorization, where users are assigned roles (e.g., admin, editor, viewer), and each role has specific permissions. Users inherit the permissions associated with their roles.

3. **Fine-Grained Authorization**: In some cases, authorization must be more granular, specifying access permissions down to individual resources or actions.

4. **Authorization Process**: The process involves:
    a. An authenticated user attempts to access a resource or perform an action.
    b. The system checks the user's identity and associated permissions.
    c. Access is granted if the user has the necessary permissions; otherwise, it's denied.

5. Purpose: Authorization ensures that authenticated users can only perform actions or access resources for which they have been granted permission. It enforces security policies and prevents unauthorized activities.

***While authentication establishes the identity of users, authorization determines what those authenticated users are allowed to do within a system.*** Both are critical components of a comprehensive security strategy, working to protect systems and data from unauthorized access and actions.

## Overview of Spring Security

Spring Security is a robust framework for building secure applications in the Java ecosystem, especially within the Spring ecosystem. It provides comprehensive security features to protect your web applications and RESTful services from various security threats. Let's delve into the overview of Spring Security in detail:

Key Features of Spring Security:

1. **Authentication and Authorization**: Spring Security provides robust authentication and authorization mechanisms. Authentication verifies the identity of users, while authorization controls access to specific resources based on roles, permissions, or rules.

2. **Pluggable Architecture**: Spring Security offers a highly customisable and pluggable architecture. You can integrate it seamlessly with various authentication providers, user stores, and access control strategies.

3. **Protection against Common Attacks**: Spring Security helps protect your application against common security threats, including cross-site request forgery (CSRF), cross-site scripting (XSS), and session fixation attacks.

4. **Form-Based Authentication**: Spring Security supports form-based authentication, making creating login forms for users easy. It handles user authentication and session management transparently.

5. **HTTP Basic and Digest Authentication**: You can configure HTTP Basic and Digest authentication for securing RESTful APIs and other HTTP-based services.

6. **Role-Based Access Control (RBAC)**: Spring Security simplifies role-based access control by allowing you to define and associate roles with users. It also supports hierarchical roles for more complex access control scenarios.

7. **Method-Level Security**: You can apply security at the method level using annotations like @**PreAuthorize** and @**PostAuthorize**. This fine-grained security enables you to secure specific methods within your application.

8. **Single Sign-On (SSO)**: Spring Security integrates with various SSO providers and protocols like OAuth2 and OpenID Connect, allowing users to log in once and access multiple applications seamlessly.

9. **Customisable Filters**: Spring Security provides a set of customisable filters to be added to the security filter chain. These filters allow you to implement custom security logic for various application parts.

10. **Session Management**: It offers session management features like session fixation protection, concurrent session control, and support for persistent logins (remember-me functionality).

11. **Event-Driven Model**: Spring Security provides an event-driven model that allows you to monitor and react to security-related events in your application.

Basic Flow of Spring Security:

1. **User Authentication**: When a user tries to access a secured resource or log in, Spring Security intercepts the request and authenticates the user using configured authentication mechanisms.

2. **Access Control**: Spring Security checks whether the authenticated user has the necessary roles and permissions to access the requested resource after authentication.

3. **Security Filter Chain**: Spring Security operates based on a chain of security filters that process incoming requests. These filters handle authentication, authorization, CSRF protection, and more.

4. **User Details Service**: Spring Security interacts with a user details service (typically a database or LDAP) to fetch user details and verify credentials.

5. **Session Management**: It manages user sessions, ensuring secure session creation and expiration. Features like session fixation protection and concurrent session control enhance security.

6. **Exception Handling**: Spring Security provides robust exception handling for authentication failures, access-denied situations, and other security-related errors.

Spring Security is a comprehensive framework that addresses the complex and evolving application security landscape. It enables developers to quickly build secure applications, protecting them from various security threats and vulnerabilities. Whether you're developing web applications or RESTful APIs, Spring Security offers the tools and flexibility needed to ensure the safety of your applications and data.

## Authentication Mechanisms

Authentication mechanisms in Spring Security are vital for verifying the identity of users and ensuring that only authorized individuals can access protected resources. Spring Security supports various authentication methods to suit different application needs. Let's explore these authentication mechanisms in detail:

1. **Form-Based Authentication**: Form-based authentication is commonly used for web applications. In this method, users enter their credentials (usually username and password) into a login form. Spring Security captures these credentials, validates them, and grants access if they are correct. It's suitable for applications with a user interface, such as websites and web applications.

2. **HTTP Basic Authentication**: HTTP Basic Authentication is a simple and widely supported authentication method for RESTful APIs and web services. The client requests an "Authorization" header containing the Base64-encoded username and password. Spring Security intercepts the request, decodes the credentials, and validates them. It's efficient but less secure than other methods like OAuth2.

3. **OAuth2:** OAuth2 is a popular authorization framework that Spring Security supports for authentication and authorization. It enables secure delegated access, allowing users to grant third-party applications limited access to their resources without revealing their credentials. OAuth2 includes various grant types, such as authorization code, implicit password, and client credentials, making it versatile for different scenarios.

4. **OpenID Connect**: OpenID Connect builds on top of OAuth2 and provides identity layer functionality. It allows applications to verify the end-user's identity based on the authentication performed by an authorization server. OpenID Connect is widely used for single sign-on (SSO) and is often used with OAuth2.

5. **JSON Web Tokens (JWT)**: JWT is a compact, self-contained means of securely transmitting information between two parties. In Spring Security, JWT can be used for authentication by generating user tokens during the login process. These tokens can then authenticate subsequent requests without needing sessions or cookies. JWT is suitable for stateless and distributed systems.

6. **LDAP (Lightweight Directory Access Protocol)**: a directory service protocol commonly used for centralised user authentication and authorization. Spring Security can integrate with LDAP servers to authenticate users against a directory, such as Active Directory or OpenLDAP. It's prevalent in enterprise environments.

7. **Custom Authentication Providers**: You can implement custom authentication providers for unique authentication requirements. Spring Security provides a flexible extension point for creating custom authentication logic, such as integrating with external identity providers or proprietary systems.

Spring Security offers a range of authentication mechanisms to cater to different application types and security needs. The choice of authentication method depends on your application's architecture, user experience, and the level of security required. These mechanisms can be combined or customised to provide a comprehensive authentication solution for your Spring-based applications.

## Lombok

Lombok is a widely used Java library recognised for its capacity to streamline and diminish the need for repetitive code in Java applications. It achieves this through the provision of annotations that generate frequently used code constructs like getters, setters, constructors, and more during the compilation phase. Below, you'll find some crucial aspects of Lombok:

- **Reduction of Repetitive Code:** Lombok is pivotal in helping developers alleviate the tedium of writing redundant code. In the realm of Java, classes frequently necessitate the creation of getter and setter methods, constructors, equals(), hashCode(), and toString() methods. Lombok allows developers to adorn their classes and fields with annotations, automating the generation of these methods and sparing them from manual implementation.

- **Annotations**: Lombok furnishes a set of annotations like *@Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor, @EqualsAndHashCode, and @ToString*, to name a few. These annotations can be applied to classes and fields to specify which code should be automatically generated.

- **Compile-Time Code Generation**: Lombok operates during compilation, ensuring that code generation transpires when the project is built using a Java compiler. This generated code becomes an integral part of your compiled classes and doesn't feature in the source code, contributing to a more concise and legible source code base.

- **Enhanced Readability and Maintainability**: Lombok's ability to curtail boilerplate code can result in more concise and understandable Java code. Additionally, it diminishes the likelihood of errors arising from manually composed getter, setter, or other methods.

- **Seamless Integration**: Lombok can seamlessly integrate into popular Java development environments like Eclipse, IntelliJ IDEA, and Maven. Many widely used build tools and integrated development environments offer plugins that support Lombok, making it highly convenient for developers.

- **Compatibility Considerations**: It's essential to remember that Lombok simplifies code and augments readability by altering your bytecode during compilation. Consequently, the generated code may not be visible within your source files, which could perplex developers unfamiliar with Lombok.

## DTO (Data Transfer Object)

A Data Transfer Object (DTO) is a design pattern commonly used in software development to transfer data between different layers, components, or modules of an application. DTOs are lightweight objects designed solely to carry data and do not contain any business logic or behaviour. Here are some critical points about DTOs:

**Data Transfer**: DTOs are primarily used for transferring data between different parts of an application. This could be between a client and a server, between different layers of an application (e.g., the presentation layer and the data access layer), or between various components or services.

**No Behavior**: DTOs are essentially data containers, and they do not have any behaviour or methods associated with them. They typically consist of fields (attributes), getters, and setters to access and modify those fields.

**Encapsulation**: DTOs encapsulate a specific set of data attributes that must be exchanged between components. They provide a structured way to package data, making it easier to manage and pass around.

**Mapping**: In many cases, DTOs are used to map data between different data models or structures. For example, they can convert data from a database entity to a format suitable for presentation on a user interface.

**Security**: DTOs can also be used to control the data exposed to clients. They allow you to exclude sensitive or unnecessary data from being sent to the client, enhancing security.

If there’s an entity class like this

```
@Entity
@Table(name = "Book")
public class Book {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String name;
	private Double price;
	private String author;
	//getter and setter
}
```

Then it’s DTO class with Lombok annotations can look like this:

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class BookDTO {

       private String name;
       private Double price;
       private String author;
}
```

This class would only use relevant information from the Entity class that needs to be transferred between different application layers and can be mapped with the entity class wherever required.

DTOs are a valuable tool for maintaining the separation of concerns and ensuring efficient data transfer between different parts of an application. They help improve code readability and maintainability by clearly defining the data exchange structure.

## Basic Authentication And AntMacthers

**Basic Authentication** is a widely used method for securing web applications. It's a simple and effective way to authenticate users.

In Spring Security, Basic Authentication involves the exchange of a username and password encoded in Base64 format.

### Role-Based Access Control

- Role-based access control (RBAC) is a crucial aspect of securing applications. It ensures that users only access functionalities they are authorized to use.
- Spring Security facilitates the implementation of RBAC through configuration and annotations.
- Typical roles include "ADMIN," "USER," and "GUEST," but you can define custom roles based on the application requirements.

### AntMatchers

Ant Matchers are a valuable feature in Spring Security for defining URL patterns.
They allow you to specify which URLs should be protected and which should be accessible without authentication.

With Ant Matchers, you can demonstrate how to restrict access to specific functionalities based on user roles.

Below is a sample code for configuring spring security using basic authentication and ant matchers for a generic Spring Boot application.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

	@Bean
	public   SecurityFilterChain   filterChain(HttpSecurity   http)   throws
Exception
	{
		http.csrf().disable()
			.authorizeHttpRequests()
			.antMatchers("/pathForAdmin/**").hasRole("ADMIN")
			.anyRequest()
			.authenticated()
			.and()
			.httpBasic();
		return http.build();
	}

	@Bean
	public UserDetailsService users()
	{
		UserDetails user1 = User.builder()
				.username("SampleUsername")
				.password(passwordEncoder().encode("password"))
				.roles("ADMIN")
				.build();

		return new InMemoryUserDetailsManager(user1,user2);
	}

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

In this code:

1. *@EnableWebSecurity* and *@Configuration* enable Spring Security in the application, while the latter indicates that the class is a configuration class for Spring Security.

2. The filterChain() method configures security filters in Spring Security.

    - .csrf().disable(): This line disables CSRF protection, which is Cross-Site Request Forgery protection.

- .antMatchers*("/pathForAdmin/**").hasRole("ADMIN"):* Here, it's defined that any request starting with *"/pathForAdmin/"* should require the "ADMIN" role for access.

- .anyRequest().authenticated(): This line specifies that any other requests (not matched by antMatchers) should be authenticated, meaning users must log in.

- .httpBasic(): This enables HTTP Basic Authentication for the application.

3. The *users()* method configures user details for authentication.

    - .username("SampleUsername"): Sets the username for the user.
    - .password(passwordEncoder().encode("password")): Sets the password for the user. It also encodes the password using the configured password encoder.
    - .roles("ADMIN"): Assigns roles to the user.

In this way, this code configures Spring Security to use HTTP Basic Authentication, specifies authorization rules, provides user details for authentication, and sets up a password encoder for secure password storage. It's a basic configuration for securing a web application with role-based access control.

## Method Level Security

Method-level security in a Spring Boot application allows you to control access to specific methods or endpoints based on user roles (or other criteria). You can specify different access rules for different methods within the same controller or service.

Spring Security provides several annotations for method-level security, and one of the most commonly used ones is *@PreAuthorize*. *@PreAuthorize* annotation allows you to specify an expression that defines access rules for a particular method. The method can be accessed only if the specified condition in the expression evaluates to true.

Refer to the code snippet below.

```
@RestController
@RequestMapping("/basePath")
public class Controller {

        @Autowired
        Service service;

        @PostMapping("/create")
        @PreAuthorize("hasRole('ADMIN')")
```

```
	public void createEntity(@RequestBody SampleEntity sampleEntity)
	{

		service.createHotel(sampleEntity);
	}
```

In this code, the *@PreAuthorize("hasRole('ADMIN')")* is used to define access rules. It checks if the authenticated user has the "ADMIN" role before allowing access to the method.

Method-level security in a Spring Boot application allows you to define access rules at a granular level, ensuring that only authorized users can execute specific methods. It provides fine-grained control over access and is a powerful tool for securing your application's functionalities.

## Conclusion

In summary, Spring Security is a versatile and powerful framework designed to address the security needs of Spring-based applications comprehensively. It provides robust authentication and authorization mechanisms, allowing developers to create secure systems. Authentication verifies user identities, while authorization defines what actions users can perform within an application, offering a clear separation of concerns.

One of Spring Security's key strengths lies in its flexibility. It supports various authentication methods, from form-based and HTTP Basic to advanced options like OAuth2, OpenID Connect, and LDAP. This adaptability empowers developers to choose the most suitable approach for their project requirements.

Additionally, Spring Security promotes customisation by enabling developers to implement custom providers, filters, and access control rules. This level of control ensures that security measures align precisely with an application's unique security demands.

Furthermore, the framework seamlessly integrates with other Spring modules and third-party libraries, simplifying the implementation of security features and offering comprehensive session management, logging, monitoring, and scalability capabilities. Spring Security is a valuable resource for safeguarding applications, protecting sensitive data, and mitigating security risks in today's ever-evolving digital landscape.

## Instructor Codes

- Hotel Application

## References

1. Official Documentation
2. Authentication vs. Authorization
3. Spring security overview
4. Authentication Mechanisms
5. Lombok
6. DTO
7. AntMatchers
8. Basic Authentication
9. Method Level Security