**What is Spring security?**

Spring Security is a framework for securing Java applications. It provides authentication, authorization and protection against common issues/point of failuers, while integrating seamlessly with Spring Boot for easy configuration.

**Authentication**: Verifies who the user is (e.g., login with username/password, OAuth2, JWT).

**Authorization**: Determines what the user is allowed to do (e.g., access control based on roles).

**Core Components:**

**Authentication Manager** : Coordinates the authentication process

**UserDetailsService** : Loads user-specific data from a data source

**PasswordEncoder** : Encodes and verifies passwords securely

**SecurityContextHolder** :Stores authentication info for current user

**Spring Security Flow :**

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

A diagram of security flow

AI-generated content may be incorrect.

1. **User Entered Credentials**

* The process begins when a user submits their login credentials (typically username and password) via a login form.

2. **Authentication Filter**

* This filter intercepts the login request.
* It extracts the credentials and creates an Authentication object.
* Then it passes this object to the **Authentication Manager**.

3. **Authentication Manager**

* The manager Coordinates the authentication process.
* It delegates the actual authentication to one or more **Authentication Providers**.

4. **Authentication Provider**

* This component performs the core authentication logic.
* It checks if it supports the type of Authentication object passed in.
* If yes, it proceeds to validate the credentials.

5. **UserDetailsService**

* The provider uses this service to load user-specific data (like username, password, roles) from a data source (e.g., database).
* It returns a UserDetails object.

6. **PasswordEncoder**

* The raw password from the login form is compared with the encoded password from the database.
* This ensures secure password handling.

7–8. **Return Path**

* If authentication is successful, the provider returns an authenticated Authentication object.
* This flows back through the **Authentication Manager** to the **Authentication Filter**.

9. **Security Context**

* The authenticated object is stored in the **SecurityContextHolder**.
* This context is accessible throughout the application to check if a user is authenticated and what roles they have.

10. **Authentication Filter (Return Path)**

* The filter completes the process by allowing access to the requested resource or redirecting the user to a success page.

**what is securityfilterchain?**

Spring Security utilizes the filter chain to perform most of the security features. Internally Spring Security maintains a filter chain where each filter has a specific responsibility.

In **Spring Security**, the SecurityFilterChain is a core component that defines how HTTP requests are processed and secured. Think of it as a **customizable pipeline of filters** that intercept incoming requests and apply security rules.

It’s a **chain of security filters** that handle authentication, authorization, session management, CSRF protection, and more.

Each filter in the chain has a specific role—like checking **credentials**, **validating tokens**, or enforcing access rules.

**What Is AuthenticationManagerBuilder?**

AuthenticationManagerBuilder is a **helper class** used to configure and build an AuthenticationManager, which is the central component responsible for handling authentication in Spring Security.

**what is Security context?**

The Security Context in Spring Security is like a personal security vault for each user—it holds all the authentication and authorization details for the current session or request.

What Is the Security Context?

* It’s a container that stores the Authentication object for the currently authenticated user.
* This object includes:
* Username
* Roles/authorities
* Credentials (usually hidden)
* Any additional details (like IP address or session info)

**What is @Preauthowired and @Secure?**

@PreAuthorize

This is a Spring Security annotation used to apply method-level security. It checks authorization before the method is invoked.

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

**public void deleteUser(Long id) {**

**// Only users with ADMIN role can access this method**

**}**

@Secured

Another method-level security annotation, but more basic than @PreAuthorize.

**@Secured("ROLE\_ADMIN")**

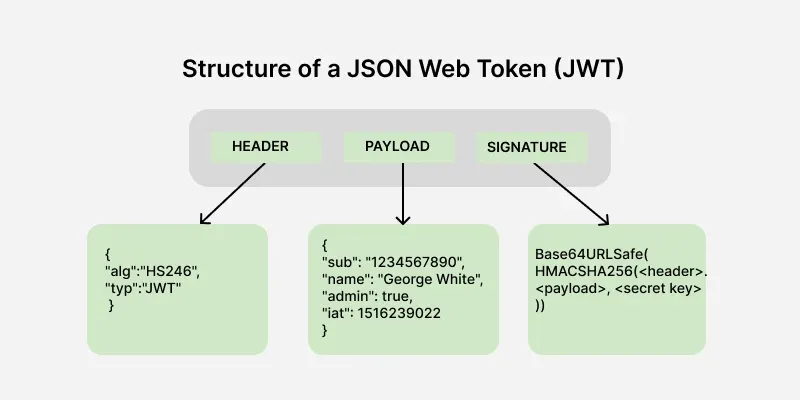
**public void updateSettings() {**

**// Only accessible to users with ROLE\_ADMIN**

**}**

**JWT**

A JSON Web Token (JWT) is a secure way to send information between a client and a server. It is mainly used in web applications and APIs to verify users and prevent unauthorized access. A JWT is JSON data secured with a cryptographic signature.



**Header:**Contains metadata about the token, such as the algorithm used for signing.

{  
 "alg": "HS256",  
 "typ": "JWT"  
}

**Payload:**Stores the claims, i.e., data being transmitted.

{  
 "userId": 123,  
 "role": "admin",  
 "exp": 1672531199  
}

**Signature:**Ensures the token's integrity and authenticity.

HMACSHA256(  
 base64UrlEncode(header) + "." + base64UrlEncode(payload),  
 secret  
)

**what is password encoder?**

A **password encoder** is a tool or function used to **securely transform a plain-text password** into an encoded (usually hashed) format before storing it in a database or verifying it during login. It's a critical part of user authentication systems.

Why Use a Password Encoder?

**Security**: Storing plain-text passwords is dangerous. Encoding ensures that even if the database is compromised, the actual passwords remain protected.

**One-way transformation**: Most encoders use hashing algorithms, which are irreversible — you can't decode the hash back to the original password.

**Verification**: During login, the input password is encoded and compared with the stored hash