Class-29: Spring Security Basics

by Pial Kanti Samadder

What is Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

Features

- Comprehensive and extensible support for both Authentication and Authorization
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc
- Servlet API integration
- Optional integration with Spring Web MVC
- And many more...

Authentication vs Authorization

Authentication

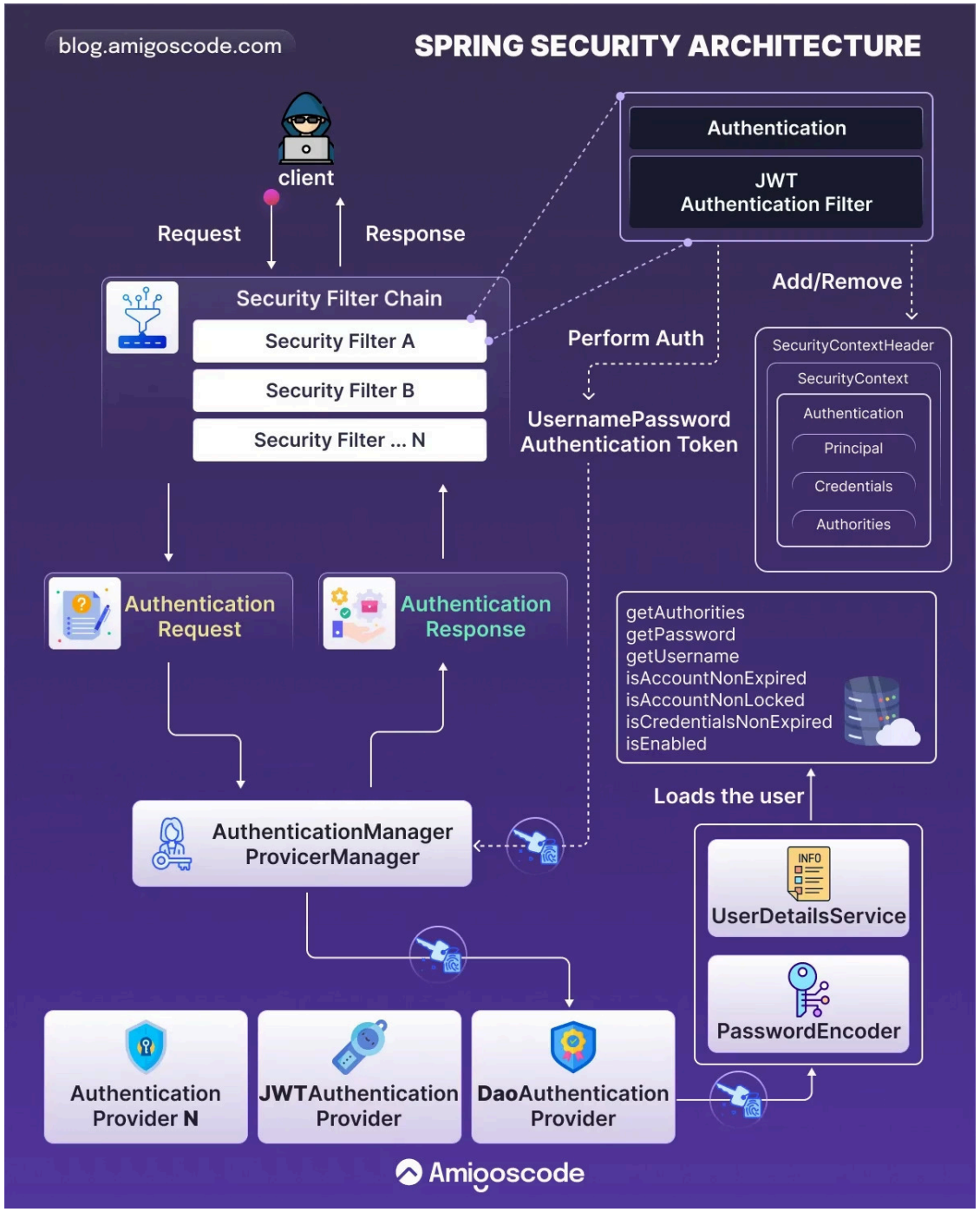
- **What it is:** Verifying the identity of a user.
- **Goal:** “Are you who you say you are?”
- **Example:** Login with username/password or JWT token.
- **Spring Security Components Involved:**
  - `UsernamePasswordAuthenticationFilter`
  - `AuthenticationManager`
  - `AuthenticationProvider`
  - `UserDetailsService`

Authorization

- **What it is:** Checking if the authenticated user has permission to access a resource.
- **Goal:** “Are you allowed to do this action?”
- **Example:** Roles like `ROLE_ADMIN` can access `/admin/**`, others cannot.
- **Spring Security Components Involved:**
  - `FilterSecurityInterceptor`
  - `@PreAuthorize`, `@Secured`
  - HTTP security rules (`authorizeHttpRequests`)

Spring Security Architecture

1. **Client Request:** User sends request to a protected resource.
2. **Filter Chain:** Request passes through multiple Spring Security filters.
3. **Authentication Trigger:** A filter identifies and starts authentication.
4. **Authentication Token:** Credentials/JWT are wrapped into an auth token.
5. **AuthenticationManager:** Delegates authentication to providers.
6. **AuthenticationProviders:** Validate credentials/JWT using matching provider.
7. **UserDetailsService:** Loads user data when using username/password login.
8. **PasswordEncoder:** Verifies encrypted passwords securely.
9. **SecurityContext:** Stores authenticated user and roles for the request.
10. **Authorization:** Checks if user has permission to access the resource.
11. **Controller Access:** If authorized, request reaches the controller.



Filter Chain

What is the Filter Chain?

- A sequence of security filters that process every incoming request.

How it works

- Filters run **in a fixed order**, each doing a specific security task.
- Each filter can **allow**, **reject**, or **trigger authentication** for the request.

Examples of Common Filters

- **CORS Filter** → Handles cross-origin requests
- **CSRF Filter** → Protects against cross-site request forgery
- **Logout Filter** → Manages logout process
- **UsernamePasswordAuthenticationFilter** → Handles login form authentication
- **BearerToken/JWT Filter** → Extracts and verifies JWT tokens
- **FilterSecurityInterceptor** → Performs final authorization check

The Filter Chain is the heart of Spring Security — every security decision starts here.

Authentication Token

An object that represents the user's login information or credentials. Authentication Token is the **“wrapper” that carries user credentials into the authentication process.**

Why do we need it?

- Spring Security uses the token to understand **who is trying to authenticate**.

How it works

- The filter creates a token (e.g., username/password or JWT).
- The token is passed to the **AuthenticationManager** for validation.

Common Token Types

- **UsernamePasswordAuthenticationToken** → for login with username/password
- **JwtAuthenticationToken** → for token-based authentication
- **OAuth2AuthenticationToken** → for Google/Facebook login

What it contains

- **Principal** (username or user details)
- **Credentials** (password or JWT)
- **Authorities** (roles/permissions)

Authentication Manager

The main component responsible for handling authentication in Spring Security. it doesn't authenticate by itself — it delegates the job to the right provider.

What does it do?

- Receives an Authentication Token from filters (username/password, JWT, OAuth2, etc.).
- Determines which **AuthenticationProvider** can handle that token.
- Delegates authentication to the correct provider.

How it works

- Uses `ProviderManager` internally to try multiple providers in order.
- If a provider validates the Authentication Token → returns an authenticated object.
- If no provider can authenticate → throws an exception.

Why is it important?

- Acts as the **central brain** of authentication.
- Decouples filters from actual authentication logic.
- Supports multiple ways to authenticate (DB login, JWT, OAuth2, LDAP).

Authentication Providers

A component in Spring Security responsible for **verifying user credentials**.

Why do we need it?

- Allows Spring Security to **support multiple authentication mechanisms** (DB, LDAP, JWT, OAuth).

How it works

1. AuthenticationManager sends the authentication request to a provider.
2. Provider checks if it can handle the authentication type.
3. If yes → validates credentials and returns an authenticated object.
4. If no → passes to the next provider in the chain.

Common Authentication Providers

- **DaoAuthenticationProvider** → Checks username & password from database.
- **JwtAuthenticationProvider** → Validates JWT tokens.
- **LdapAuthenticationProvider** → Authenticates against LDAP.
- **AnonymousAuthenticationProvider** → Handles anonymous users.

Key Benefits

- Pluggable & flexible
- Supports multiple auth methods
- Easy to extend with custom providers

UserDetailsService

- A Spring Security interface that **loads user-specific data** for authentication.

Why is it important?

- Provides user information (username, password, roles) to **AuthenticationProvider**.
- Decouples authentication logic from user storage (DB, LDAP, etc.).

How it works

1. **AuthenticationProvider** calls:

```
UserDetails user = userDetailsService.loadUserByUsername(username);
```

1. `UserDetailsService` fetches user from **database or other storage**.
2. Returns a `UserDetails` object containing:
    - `username`
    - `password` (hashed)
    - `authorities` (roles/permissions)
    - account status (enabled, locked, expired)

Common Implementation

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username) {
        // Load user from DB and return UserDetails
    }
}
```

PasswordEncoder

- A Spring Security interface that **handles password hashing and verification**.

Why is it important?

- Never store plain text passwords in the database.
- Ensures passwords are securely hashed and compared safely during login.

How it works

1. **Hashing passwords when saving**:

```
String hashed = passwordEncoder.encode(rawPassword);
```

1. **Matching passwords during login**:

```
passwordEncoder.matches(rawPassword, hashedPasswordFromDB);
```

Common Implementations

- `BCryptPasswordEncoder` → Strong, recommended default
- `Argon2PasswordEncoder` → Modern, secure option
- `NoOpPasswordEncoder` → For testing only, not secure

SecurityContext

- A Spring Security component that **stores the authentication information of the current user**.

Why is it important?

- Allows the application to know **who is logged in** and what **roles/permissions** they have.
- Used by **authorization checks** (`@PreAuthorize`, `hasRole`, URL rules).

How it works

1. After authentication (login or JWT validation), Spring Security creates an `Authentication` object.
2. The object is stored in the SecurityContext:

```
SecurityContextHolder.getContext().setAuthentication(authentication);
```

1. During the request, filters and controllers can access the user:

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
```

What it contains

- **Principal** → the logged-in user (`UserDetails`)
- **Credentials** → password or token (`null` after authentication)
- **Authorities** → roles/permissions
- **Account status flags** → enabled, locked, expired

SecurityContext holds the current user's authentication and authorities for the duration of a request, enabling secure access control.

Running Sample Project

- **Clone the Repository**

If you haven't cloned the repository yet, run the following command (ensure `git` is installed):

```
git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git
```

Then switch to the correct branch for today's class (replace with the actual branch name, e.g., `class-24-jpa-relationship`):

```
git fetch
git switch class-26-pagination-sorting
```

Or,

If You Already Have the Repository Cloned, simply open your existing project folder and switch (or checkout) to the appropriate branch:

```
git fetch
git switch class-26-pagination-sorting
```

- **Set Up and Run PostgreSQL Database**

You can run PostgreSQL **either via Docker** or a **desktop installation**.

Option 1: Run via Docker

A `compose.yml` file is available in the root of the repository.
Run the following command from the project root:

```
docker compose up -d
```

This will start a PostgreSQL container automatically.

Or,

Option 2: Run via PostgreSQL Desktop (Manual Setup)

If you already have PostgreSQL installed locally:

1. Start your PostgreSQL server.
2. Create a new database named `crud_db` if not exists.

- **Open the Project in IntelliJ IDEA**

1. Open IntelliJ IDEA.
2. Click **File → Open** and select the `crud-sample` folder inside the repository.
3. Let IntelliJ import Maven/Gradle dependencies automatically.