Class-30: Endpoint Authorization with Spring Security

by Pial Kanti Samadder

Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is a security model where **permissions are assigned to roles**, and **users are assigned to those roles**.
Instead of giving permissions directly to each user, roles act as a bridge.

Key Concepts

- **User** → A person or system accessing the application
- **Role** → A group of permissions (e.g., `ADMIN`, `USER`, `MODERATOR`)
- **Permission** → A specific allowed action (e.g., `READ_POST`, `DELETE_POST`)

How RBAC Works

1. Define **roles** in the system
2. Assign **permissions** to each role
3. Assign **users** to roles
4. Users get permissions **indirectly** through their roles

Why RBAC?

- Simplifies permission management
- Scales better for large systems
- Easier to audit
- Ensures consistency in access rights
- Reduces risk of human error by avoiding per-user permissions

Path-Level Security in Spring Security

What Is It?

Path-level (URL-based) security lets you define **who can access which URL paths**.
Example: `/admin/**` → Admin only, `/public/**` → Everyone.

Core Idea

Spring Security checks authorization **before** requests reach controllers.
By default → **All requests require authentication**.

Key Configuration

```
http.authorizeHttpRequests(auth -> auth
    .requestMatchers("/public/**").permitAll()
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated()
);
```

Why Use Path-Level Security?

- Simple & centralized access rules
- Easy to read and maintain
- Useful for grouping and protecting large URL sections

Matching Requests

There are two ways to match requests:

Matching Using Ant

Ant is the default language that Spring Security uses to match requests.

You can use it to match a single endpoint or a directory, and you can even capture placeholders for later use. You can also refine it to match a specific set of HTTP methods.

```
http
    .authorizeHttpRequests((authorize) -> authorize
        .requestMatchers("/resource/**").hasAuthority("USER")
        .anyRequest().authenticated()
    )
```

Matching Using Regular Expressions

Spring Security supports matching requests against a regular expression. This can come in handy if you want to apply more strict matching criteria than `**` on a subdirectory.

For example, consider a path that contains the username and the rule that all usernames must be alphanumeric. You can use `RegexRequestMatcher` to respect this rule, like so:

```
http
    .authorizeHttpRequests((authorize) -> authorize
        .requestMatchers(RegexRequestMatcher.regexMatcher("/resource/[A-Za-z0-9]+")).hasAuthority("USER")
        .anyRequest().denyAll()
    )
```

Matching By Http Method

You can also match rules by HTTP method. One place where this is handy is when authorizing by permissions granted, like being granted a `read` or `write` privilege.

```
http
    .authorizeHttpRequests((authorize) -> authorize
        .requestMatchers(HttpMethod.GET).hasAuthority("read")
        .requestMatchers(HttpMethod.POST).hasAuthority("write")
        .anyRequest().denyAll()
    )
```

Method-Level Security in Spring Security

In addition to modeling authorization at the request level, Spring Security also supports modeling at the method level.

You can activate it in your application by annotating any `@Configuration` class with `@EnableMethodSecurity` since **Spring Boot Starter Security does not activate method-level authorization by default.**

Why Method-Level Security

- Extracting fine-grained authorization logic; for example, when the method parameters and return values contribute to the authorization decision.
- Enforcing security at the service layer
- Stylistically favoring annotation-based over `HttpSecurity`-based configuration

Example

```
@PreAuthorize("#userId == authentication.name or hasRole('ADMIN')")
public UserProfile getUserProfile(String userId) {
    return userService.findProfile(userId);
}
```

Annotations in Method-Level Security

Spring Security provides several annotations to enforce **fine-grained authorization** directly on methods.

- `@PreAuthorize`

Runs **before** method execution. Used for checking roles, permissions, and expression-based rules.

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

- `@PostAuthorize`

Runs **after** method execution. Useful when decisions depend on the returned object.

```
@PostAuthorize("returnObject.owner == authentication.name")
public Order getOrder(Long id) {}
```

- `@Secured`

Simple role-based checks.

```
@Secured("ROLE_MANAGER")
public void updateSettings() {}
```

- `@RolesAllowed`

(JSR-250 Standard) — similar to `@Secured`.

```
@RolesAllowed({"ADMIN", "MODERATOR"})
public void archivePost() {}
```

CSRF Token

What is a CSRF Token?

A unique, secret value generated by the server per user session.

Prevents Cross-Site Request Forgery (CSRF) attacks.

Must be submitted with state-changing requests: POST, PUT, PATCH, DELETE (not GET).

How Spring MVC Uses CSRF Tokens

In Spring MVC apps, Spring automatically:

- Generates a CSRF token per session
- Adds it to each generated view (HTML forms)
- Expects the token in every modifying request

An attacker cannot get the token from their own page, so forged requests fail.

Spring Security Default Behavior

- Spring Security 4.x and later: CSRF protection enabled by default
- The token is stored in the request attribute: `_csrf`
- Forms can use `${_csrf.parameterName}` and `${_csrf.token}` to include it automatically

Disable CSRF (if needed):

```
http.csrf(AbstractHttpConfigurer::disable);
```

Logout in Token-Based Authentication

How Logout Works in Token-Based Systems

- Unlike session-based apps, **tokens are stateless**
- The server does **not store tokens**, so “logging out” cannot simply invalidate a session on the server
- Logout usually involves **client-side token removal** or **server-side blacklist/revocation**

Common Approaches

- **Client-Side Token Removal**
    - Simply delete the token from browser storage (localStorage, sessionStorage, cookies)
    - Prevents further requests with the token

```
localStorage.removeItem("accessToken");
```

- **Token Blacklisting**
    - Maintain a **server-side blacklist** of tokens until they expire
    - Any request with a blacklisted token is rejected
    - Useful for **logout-before-token-expiry** scenarios
- **Short-Lived Tokens with Refresh Tokens**
    - Access tokens expire quickly (e.g., 15 mins)
    - Refresh token can be invalidated on logout
    - Ensures minimal risk if a token is stolen

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-30-role-permission):

```
git fetch
git switch class-30-role-permission
```

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-30-role-permission
```

- ## 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.