



Spring Security at Method Level

Last Updated : 23 Jul, 2025

Spring Security provides a powerful way to secure Java applications, allowing developers to control authentication and access at different levels. One of its key features is **method-level security**, which lets us use security constraints on specific methods instead of the entire class or application.

Enabling Method-Level Security

In Spring Security, method-level security is enabled using the **@EnableMethodSecurity** annotation instead of the deprecated **@EnableGlobalMethodSecurity**.

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

Using Annotations for Method-Level Security

Spring Security provides annotations to restrict access to methods based on roles and conditions:

1. @Secured

The **@Secured** annotation allows access control based on user roles.

```
@Secured("ROLE_ADMIN")
public void deleteUser(int userId) {
```

```
// Method logic here  
}
```

Only users with the **ROLE_ADMIN** role can execute the **deleteUser()** method.

2. @PreAuthorize

The [@PreAuthorize](#) annotation provides more flexibility by using [Spring Expression Language](#) (SpEL) for defining conditions.

```
@PreAuthorize("hasRole('ROLE_ADMIN') or  
(hasRole('ROLE_USER') and #userId == authentication.principal.id)")  
public void updateUser(int userId) {  
    // Method logic here  
}
```

This allows either an **ROLE_ADMIN** user or a **ROLE_USER** with the same **userId** as the currently authenticated user to update user details.

3. @PostAuthorize

The [@PostAuthorize](#) annotation is used to apply security constraints after the method execution for filtering return values.

```
@PostAuthorize("returnObject.ownerId ==  
authentication.principal.id")  
public User getUserDetails(int userId) {  
    // Fetch user details  
    return userService.findById(userId);  
}
```

Ensures that only the owner of the user account can access the returned user details.

Complete Example in a Spring Boot Application

1. Add Dependency

Include Spring Security in your project (Maven):

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

2. Security Configuration

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {
    // Other security configurations can be added here
}
```

3. Service Layer with Method-Level Security

```
@Service
public class UserService {

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public void deleteUser(int userId) {
        // Logic for deleting user
    }

    @PreAuthorize("hasRole('ROLE_USER') and #userId == authentication.principal.id")
    public void updateUser(int userId) {
        // Logic for updating user
    }

    @PostAuthorize("returnObject.ownerId == authentication.principal.id")
    public User getUserDetails(int userId) {
        // Fetch and return user details
        return userService.findById(userId);
    }
}
```

```
}
```

4. Running the Application

Start the application and test method-level security using an authenticated user with appropriate roles.

Spring Security's method-level security provides fine-grained access control to secure specific business logic methods. With **@EnableMethodSecurity** along with **@Secured**, **@PreAuthorize**, and **@PostAuthorize**, we can use role-based and condition-based security.

[Comment](#)[More info](#)[Advertise with us](#)