Role-Based Authorization in Spring Security

Step 1: Define Roles Using an Enum

Create a new package called enums and add an enum named Roles:

```
package com.springJourney.Week5Practice.Enums;

public enum Roles {
    ADMIN,
    CREATOR,
    USER
}
```

Step 2: Add Roles to UserEntity

Modify UserEntity to include a roles attribute:

```
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"))
@Column(name = "role")
@Enumerated(EnumType.STRING)
private Set<Roles> roles;
```

Run the project and add users with different roles.

Step 3: Implement Role-Based Authorization

1. Update getAuthorities() in UserEntity

Since UserEntity implements UserDetails, override getAuthorities():

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return roles.stream()
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role.name()))
            .collect(Collectors.toSet());
}
```

2. Update JWTFilter to Assign Authorities

Modify doFilterInternal() in JWTFilter to assign roles to SecurityContextHolder:

```
UsernamePasswordAuthenticationToken authenticationToken =
    new UsernamePasswordAuthenticationToken(userEntity, null,
userEntity.getAuthorities());
SecurityContextHolder.getContext().setAuthentication(authenticationToken);
```

3. Configure Role-Based Access in WebSecurityConfig

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
```

```
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity
        .authorizeHttpRequests(auth ->
            auth
                .requestMatchers("/auth/**", "/home.html").permitAll()
                .requestMatchers("/post/**").hasRole(ADMIN.name())
                .requestMatchers(HttpMethod.GET, "/post/**").hasAnyRole(ADMIN.name(),
CREATOR.name())
                .anyRequest().authenticated())
        .csrf(csrfConfig -> csrfConfig.disable())
        .sessionManagement(sessionConfig ->
            sessionConfig.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
        .addFilterBefore(loggingFilter, JwtAuthFilter.class);
    return httpSecurity.build();
}
```

Step 4: Handling Hibernate Errors

If there is an error in Hibernate creating user_roles, manually create the table:

```
CREATE TABLE user_roles (
    user_id BIGINT NOT NULL,
    role VARCHAR(255) NOT NULL,
    PRIMARY KEY (user_id, role),
    CONSTRAINT fk_user_roles FOREIGN KEY (user_id) REFERENCES users(userid) ON
DELETE CASCADE
);
```

After successful operation, set spring.jpa.hibernate.ddl-auto=none.

Granular Role-Based Authority

Authorities define specific actions a user can perform:

- **READ**: Can view data.
- **WRITE**: Can modify data.

Roles can have multiple authorities, adding flexibility.

Security Method Annotations in Spring Security

Spring Security allows securing methods directly with annotations.

1. @Secured

Restricts method access to specific roles.

```
@Secured("ROLE_ADMIN")
```

```
public void adminOnlyTask() {
    // Only users with ROLE_ADMIN can access this
}
```

2. @PreAuthorize

Evaluates conditions before method execution.

```
@PreAuthorize("hasRole('ADMIN') and hasAuthority('MANAGE_USERS')")
public void manageUsers() {
    // Only accessible if both conditions are true
}
```

3. @PostAuthorize

Evaluates conditions after method execution.

```
@PostAuthorize("returnObject.owner == authentication.name")
public Document getDocument(Long id) {
    return documentService.findById(id);
}
```

4. Enable Method-Level Security

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // Security configuration
}
```

Summary: RequestMatchers vs. Method-Level Security

RequestMatchers (Global Security)

```
http
    .authorizeRequests()
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
    .anyRequest().authenticated();
```

Method-Level Security (Fine-Grained Control)

```
@PreAuthorize("hasRole('ADMIN')")
public void secureTask() {
    // Only accessible to admins
}
```

By combining these approaches, you achieve both **global** and **fine-grained** security in your Spring Boot application.

By Aron