Authorization

jim stafford

Fall 2024 v2020-07-13: Built: 2024-11-19 21:35 EST

Table of Contents

Chapter 1. Introduction

We have spent a significant amount of time to date making sure we are identifying the caller, how to identify the caller, restricting access based on being properly authenticated, and the management of multiple users. In this lecture we are going to focus on expanding authorization constraints to both roles and permission-based authorities.

1.1. Goals

You will learn:

- the purpose of authorities, roles, and permissions
- how to express authorization constraints using URI-based and annotation-based constraints
- how the enforcement of the constraints is accomplished
- how to potentially customize the enforcement of constraints

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. define the purpose of a role-based and permission-based authority
2. identify how to construct an `AuthorizationManager` and supply criteria to make access decisions
3. implement URI Path-based authorization constraints
4. implement annotation-based authorization constraints
5. implement role inheritance
6. implement an `AccessDeniedException` controller advice to hide necessary stack trace information and provide useful error information to the caller
7. identify the detailed capability of expression-based constraints to be able to handle very intricate situations

Chapter 2. Authorities, Roles, Permissions

An authority is a general term used for a value that is granular enough to determine whether a user will be granted access to a resource. There are different techniques for slicing up authorities to match the security requirements of the application. Spring uses roles and permissions as types of authorities.

A role is a course-grain authority assigned to the type of user accessing the system and the prototypical uses that they perform. For example ROLE_ADMIN, ROLE_CLERK, or ROLE_CUSTOMER are relative to the roles in a business application.

A permission is a more fine-grain authority that describes the action being performed versus the role of the user. For example "PRICE_CHECK", "PRICE_MODIFY", "HOURS_GET", and "HOURS_MODIFY" are relative to the actions in a business application.

No matter which is being represented by the authority value, Spring Security looks to grant or deny access to a user based on their assigned authorities and the rules expressed to protect the resources accessed.

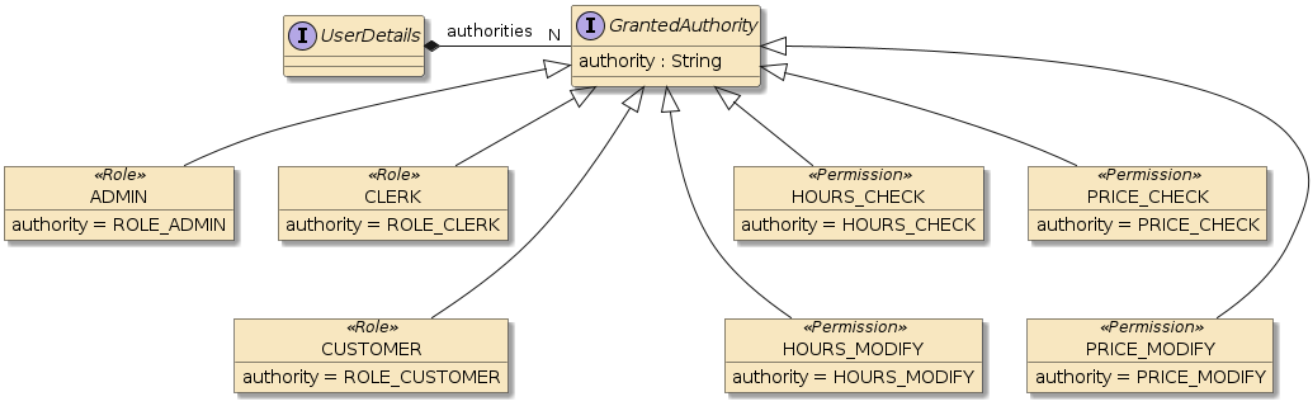

Figure 1. Role and Permission Authorities

Spring represents both roles and permissions using a `GrantedAuthority` class with an authority string carrying the value. Role authority values have, by default, a "ROLE_" prefix, which is a configurable value. Permissions/generic authorities do not have a prefix value. Aside from that, they (roles, permissions, and authorities) look very similar but are not always treated equally.

Spring refers to an authority with `ROLE_` prefix as a "role" and anything with a raw value as an "authority" or "permission". `ROLE_ADMIN` authority represents an `ADMIN` role. `PRICE_CHECK` authority represents a `PRICE_CHECK` permission.

Chapter 3. Authorization Constraint Types

There are two primary ways we can express authorization constraints within Spring Security: path-based and annotation-based.

In the previous sections we have discussed legacy `WebSecurityConfigurer` versus modern Component-based configuration approaches. In this authorization section, we will discuss another dimension of implementation options — legacy `AccessDecisionManager`/`AccessDecisionVoter` versus modern `AuthorizationManager` approach. Although the legacy `WebSecurityConfigurer` has been removed from Spring Security 6, the legacy `AccessDecisionManager` that was available in Spring Security ⇐ 5 is still available but deprecated in favor of the modern `AuthorizationManager` approach. They will both be discussed here since you are likely to encounter the legacy approach in older applications, but the focus will be on the modern approach.

3.1. Path-based Constraints

Path-based constraints are specific to web applications and controller operations since the constraint is expressed against a URI pattern. We define path-based authorizations using the same `HttpSecurity` builder we used with authentication. We can use the legacy `WebSecurityConfigurer` in Spring Security ⇐ 5 or the modern Component-based approach in Spring Security >= 5 — but never both in the same application.

- legacy `WebSecurityConfigurer` Approach

  Authn and Authz HttpSecurity Configuration

```
@Configuration
@Order(0)
@RequiredArgsConstructor
public static class APIConfiguration extends WebSecurityConfigurerAdapter {
    private final UserDetailsService jdbcUserDetailsService;
    @Override
    public void configure(WebSecurity web) throws Exception { ...}
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(cfg->cfg.antMatchers("/api/**"));
        //...
        http.httpBasic();
        //remaining authn and upcoming authz goes here
```

- modern Component-based Approach

  Authn and Authz HttpSecurity Configuration

```
@Bean
public WebSecurityCustomizer authzStaticResources() {
    return (web) -> web.ignoring().requestMatchers("/content/**");
}
@Bean
```

```
@Order(0)
public SecurityFilterChain authzSecurityFilters(HttpSecurity http) throws Exception
{
    http.securityMatchers(cfg->cfg.requestMatchers("/api/**"));
    //...
    http.httpBasic();
    //remaining authn and upcoming authz goes here
    return http.build();
}
```

The API to instantiate the legacy and modern authorization implementations are roughly the same — especially for annotation-based techniques — however:

- when using deprecated `http.authorizeRequests()` — the deprecated, legacy `AccessDecisionManager` approach will be constructed.
- when using `http.authorizeHttpRequests()` — (Note the extra `Http` in the name) — the modern `AuthorizationManager` will be constructed.

The legacy `WebSecurityConfigurer` examples will be shown with the legacy and deprecated `http.authorizeRequests()` approach. The modern Component-based examples will be shown with the modern `http.authorizeHttpRequests()` approach.

The first example below shows a URI path restricted to the `ADMIN` role. The second example shows a URI path restricted to the `ROLE_ADMIN`, or `ROLE_CLERK`, or `PRICE_CHECK` authorities.

It is worth saying multiple times. Pay attention to the use of the terms "role" and "authority" within Spring Security. `ROLE_X` authority is an "X" role. `X` authority is an "X" permission. The term permission is conceptual. Spring Security only provides builders for roles and authorities.

3.1.1. Legacy AccessDecisionManager Approach

In the legacy `AccessDecisionManager` approach, the (deprecated) `http.authorizeRequests` call is used to define accesses.

Example legacy Path-based Constraints

```
http.authorizeRequests(cfg->cfg.antMatchers(
        "/api/authorities/paths/admin/**")
        .hasRole("ADMIN")); ①
http.authorizeRequests(cfg->cfg.antMatchers(HttpMethod.GET,
        "/api/authorities/paths/price")
        .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK")); ②
```

① `ROLE_` prefix automatically added to role authorities

② `ROLE_` prefix must be manually added when expressed as a generic authority

3.1.2. Modern AuthorizationManager Approach

In the modern `AuthorizationManager` approach, the `http.authorizeHttpRequests` call is used to define accesses.

The first example below shows a terse use of `hasRole` and `hasAnyAuthority` that will construct a set of `AuthorityAuthorizationManagers` associated with the URI pattern.

Example modern Path-based Constraints

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/admin/**")
        .hasRole("ADMIN") ①
);
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            HttpMethod.GET, "/api/authorities/paths/price")
        .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK") ②
);
```

① `ROLE_` prefix automatically added to role authorities. This is building an `AuthorityAuthorizationManager` with an `MvcRequestMatcher` to identify URIs to enforce and an `ADMIN` role to check.

② `ROLE_` prefix must be manually added when expressed as a generic authority. This is building an `AuthorityAuthorizationManager` with an `MvcRequestMatcher` to identify URIs to enforce and a set of authorities ("OR"-d) to check.

The second example below shows a more complex version of `hasAnyAuthority()` to show how `anyOf()` and `allOf()` calls and the aggregate and hierarchical design of the `AuthorityAuthorizationManager` can be used to express more complex access checks.

Example modern Path-based Constraints with Manually Constructed anyOf()

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
        HttpMethod.GET, "/api/authorities/paths/price")
        .access(AuthorizationManagers.anyOf(
                AuthorityAuthorizationManager.hasAuthority("PRICE_CHECK"),
                AuthorityAuthorizationManager.hasRole("ADMIN"),
                AuthorityAuthorizationManager.hasRole("CLERK")
        )));
```

Out-of-the-box, path-based authorizations support role inheritance, roles, and permission-based constraints as well as Spring Expression Language (SpEL).

3.2. Annotation-based Constraints

Annotation-based constraints are not directly related to web applications and not associated with URIs. Annotations are placed on the classes and/or methods they are meant to impact. The processing of those annotations has default, built-in behavior that we can augment and modify. The descriptions here are constrained to out-of-the-box capability before trying to adjust anything.

There are three annotation options in Spring:

- @Secured — this was the original, basic annotation Spring used to annotate access controls for classes and/or methods.

```
@Secured("ROLE_ADMIN") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin(

@Secured({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"}) ②
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

① `ROLE_` prefix must be included in string

② Roles and Permissions supported with modern `AuthorizationManager` approach

- JSR 250 — this is an industry standard API for expressing access controls using annotations for classes and/or methods. This is also adopted by JakartaEE.

```
//@RolesAllowed("ROLE_ADMIN") -- Spring Security <= 5
@RolesAllowed("ADMIN") //Spring Security >= 6 ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin(

//@RolesAllowed({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
@RolesAllowed({"ADMIN", "CLERK", "PRICE_CHECK"}) ②
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

① `ROLE_` prefix no longer included with JSR250 role annotations — automatically added

② Roles and Permissions supported with modern `AuthorizationManager` approach

- Expressions — this annotation capability is based on the powerful Spring Expression Language (SpEL) that allows for ANDing and ORing of multiple values and includes inspection of parameters and current context.

```
@PreAuthorize("hasRole('ADMIN')") ①
@GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doAdmin(
...
@PreAuthorize("hasAnyRole('ADMIN','CLERK') or hasAuthority('PRICE_CHECK')") ②
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

① `ROLE_` prefix automatically added to role authorities

② `ROLE_` prefix not added to generic authority references

Chapter 4. Setup

The bulk of this lecture will be demonstrating the different techniques for expressing authorization constraints. To do this, I have created four controllers — configured using each technique and an additional `whoAmI` controller to return a string indicating the name of the caller and their authorities.

4.1. Who Am I Controller

To help us demonstrate authorities, I have added a controller to the application that will accept an injected user and return a string that describes who called.

WhoAmI Controller

```
@RestController
@RequestMapping("/api/whoAmI")
public class WhoAmIController {
    @GetMapping(produces={MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> getCallerInfo(
            @AuthenticationPrincipal UserDetails user) { ①

        List<?> values = (user!=null) ?
                List.of(user.getUsername(), user.getAuthorities()) :
                List.of("null");
        String text = StringUtils.join(values);

        return ResponseEntity.ok(text);
    }
}
```

① `UserDetails` of authenticated caller injected into method call

The controller will return the following when called without credentials.

Anonymous Call

```
$ curl http://localhost:8080/api/whoAmI
[null]
```

The controller will return the following when called with credentials

Authenticated Call

```
$ curl http://localhost:8080/api/whoAmI -u frasier:password
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

4.2. Demonstration Users

Our user database has been populated with the following users. All have an assigned role (Roles all start with `ROLE_` prefix). One (frasier) has an assigned permission.

```
insert into authorities(username, authority) values('sam','ROLE_ADMIN');
insert into authorities(username, authority) values('rebecca','ROLE_ADMIN');

insert into authorities(username, authority) values('woody','ROLE_CLERK');
insert into authorities(username, authority) values('carla','ROLE_CLERK');

insert into authorities(username, authority) values('norm','ROLE_CUSTOMER');
insert into authorities(username, authority) values('cliff','ROLE_CUSTOMER');
insert into authorities(username, authority) values('frasier','ROLE_CUSTOMER');
insert into authorities(username, authority) values('frasier','PRICE_CHECK'); ①
```

① frasier is assigned a (non-role) permission

4.3. Core Security FilterChain Setup

The following shows the initial/core SecurityFilterChain setup carried over from earlier examples. We will add to this in a moment.

Core SecurityFilterChain Setup

```
//HttpSecurity http
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/whoAmI",
            "/api/authorities/paths/anonymous/**")
        .permitAll());

http.httpBasic(cfg->cfg.realmName("AuthzExample"));
http.formLogin(cfg->cfg.disable());
http.headers(cfg->cfg.disable()); //disabling all security headers
http.csrf(cfg->cfg.disable());
http.cors(cfg-> cfg.configurationSource(
        req->new CorsConfiguration().applyPermitDefaultValues()));
http.sessionManagement(cfg->cfg
    .sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

- The path-based, legacy `AccessDecisionManager` approach is defined through calls to `http.authorizeRequests`

  Core SecurityFilterChain legacy `AccessDecisionManager` Setup

  ```
  http.authorizeRequests(cfg->cfg.antMatchers(
          "/api/whoami",
          "/api/authorities/paths/anonymous/**")
          .permitAll());
  ```

```
//more ...
```

- The path-based, modern `AuthorizationManager` approach is defined through calls to `http.authorizeHttpRequests`

Core SecurityFilterChain modern `AuthorizationManager` Setup

```
//HttpSecurity http
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/whoAmI",
            "/api/authorities/paths/anonymous/**")
        .permitAll());
//more ...
```

4.4. Controller Operations

The controllers in this overall example will accept API requests and delegate the call to the WhoAmIController. Many of the operations look like the snippet example below — but with a different URI.

PathAuthoritiesController Snippet

```
@RestController
@RequestMapping("/api/authorities/paths")
@RequiredArgsConstructor
public class PathAuthoritiesController {
    private final WhoAmIController whoAmI; ①

    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
            @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user); ②
    }
```

① whoAmI controller injected into each controller to provide consistent response with username and authorities

② API-invoked controller delegates to whoAmI controller along with injected `UserDetails`

Chapter 5. Path-based Authorizations

In this example, I will demonstrate how to apply security constraints on controller methods based on the URI used to invoke them. This is very similar to the security constraints of legacy servlet applications.

The following snippet shows a summary of the URIs in the controller we will be implementing.

Controller URI Summary Snippet

```
@RequestMapping("/api/authorities/paths")
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    @GetMapping(path = "clerk", produces = {MediaType.TEXT_PLAIN_VALUE})
    @GetMapping(path = "customer", produces = {MediaType.TEXT_PLAIN_VALUE})
    @GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
    @GetMapping(path = "authn", produces = {MediaType.TEXT_PLAIN_VALUE})
    @GetMapping(path = "anonymous", produces = {MediaType.TEXT_PLAIN_VALUE})
    @GetMapping(path = "nobody", produces = {MediaType.TEXT_PLAIN_VALUE})
```

5.1. Path-based Role Authorization Constraints

We have the option to apply path-based authorization constraints using roles. The following example locks down three URIs to one or more roles each.

- legacy `AccessDecisionManager` Approach

  Example Path Role Authorization Constraints — Legacy AccessDecisionManager Approach

  ```
  http.authorizeRequests(cfg->cfg.antMatchers(
          "/api/authorities/paths/admin/**")
          .hasRole("ADMIN")); ①
  http.authorizeRequests(cfg->cfg.antMatchers(
          "/api/authorities/paths/clerk/**")
          .hasAnyRole("ADMIN", "CLERK")); ②
  http.authorizeRequests(cfg->cfg.antMatchers(
          "/api/authorities/paths/customer/**")
          .hasAnyRole("CUSTOMER")); ③
  ```

  ① `admin` URI may only be called by callers having role `ADMIN` (or `ROLE_ADMIN` authority)

  ② `clerk` URI may only be called by callers having either the `ADMIN` or `CLERK` roles (or `ROLE_ADMIN` or `ROLE_CLERK` authorities)

  ③ `customer` URI may only be called by callers having the role `CUSTOMER` (or `ROLE_CUSTOMER` authority)

- modern `AuthorizationManager` approach

Example Path Role Authorization Constraints — Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/admin/**")
        .hasRole("ADMIN")); ①
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/clerk/**")
        .hasAnyRole("ADMIN", "CLERK")); ②
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/customer/**")
        .hasAnyRole("CUSTOMER")); ③
```

① `admin` URI may only be called by callers having role `ADMIN` (or `ROLE_ADMIN` authority)

② `clerk` URI may only be called by callers having either the `ADMIN` or `CLERK` roles (or `ROLE_ADMIN` or `ROLE_CLERK` authorities)

③ `customer` URI may only be called by callers having the role `CUSTOMER` (or `ROLE_CUSTOMER` authority)

5.2. Example Path-based Role Authorization (Sam)

The following is an example set of calls for `sam`, one of our users with role `ADMIN`. Remember that role `ADMIN` is basically the same as saying authority `ROLE_ADMIN`.

```
$ curl http://localhost:8080/api/authorities/paths/admin -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/paths/clerk -u sam:password ②
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/paths/customer -u sam:password ③
{"timestamp":"2020-07-14T15:12:25.927+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/customer"}
```

① `sam` has `ROLE_ADMIN` authority, so `admin` URI can be called

② `sam` has `ROLE_ADMIN` authority and `clerk` URI allows both roles `ADMIN` and `CLERK`

③ `sam` lacks role `CUSTOMER` required to call `customer` URI and is rejected with 403/Forbidden error

5.3. Example Path-based Role Authorization (Woody)

The following is an example set of calls for `woody`, one of our users with role `CLERK`.

```
$ curl http://localhost:8080/api/authorities/paths/admin -u woody:password ①
{"timestamp":"2020-07-14T15:12:46.808+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/admin"}

$ curl http://localhost:8080/api/authorities/paths/clerk -u woody:password ②
```

```
[woody, [ROLE_CLERK]]

$ curl http://localhost:8080/api/authorities/paths/customer -u woody:password ③
{"timestamp":"2020-07-14T15:13:04.158+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/customer"}
```

① `woody` lacks role `ADMIN` required to call `admin` URI and is rejected with 403/Forbidden

② `woody` has `ROLE_CLERK` authority, so `clerk` URI can be called

③ `woody` lacks role `CUSTOMER` required to call `customer` URI and is rejected with 403/Forbidden

Chapter 6. Path-based Authority Permission Constraints

The following example shows how we can assign permission authority constraints. It is also an example of being granular with the HTTP method in addition to the URI expressed.

- legacy `AccessDecisionManager` approach

Path-based Authority Authorization Constraints - Legacy AccessDecisionManager Approach

```
http.authorizeRequests(cfg->cfg.antMatchers(HttpMethod.GET, ①
        "/api/authorities/paths/price")
        .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK")); ②
```

① definition is limited to GET method for URI `price` URI

② must have permission `PRICE_CHECK` or roles `ADMIN` or `CLERK`

- modern `AuthorizationManager` approach

Path-based Authority Authorization Constraints - Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            HttpMethod.GET, "/api/authorities/paths/price") ①
        .hasAnyAuthority("PRICE_CHECK", "ROLE_ADMIN", "ROLE_CLERK")); ②
```

① definition is limited to GET method for URI `price` URI

② must have permission `PRICE_CHECK` or roles `ADMIN` or `CLERK`

6.1. Path-based Authority Permission (Norm)

The following example shows one of our users with the `CUSTOMER` role being rejected from calling the `GET price` URI.

Path-based Authority Permission (Norm)

```
$ curl http://localhost:8080/api/authorities/paths/customer -u norm:password ①
[norm, [ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/paths/price -u norm:password ②
{"timestamp":"2020-07-14T15:13:38.598+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/price"}
```

① `norm` has role `CUSTOMER` required to call `customer` URI

② `norm` lacks the `ROLE_ADMIN`, `ROLE_CLERK`, and `PRICE_CHECK` authorities required to invoke the `GET price` URI

6.2. Path-based Authority Permission (Frasier)

The following example shows one of our users with the `CUSTOMER` role and `PRICE_CHECK` permission. This user can call both the `customer` and `GET price` URIs.

Path-based Authority Permission (Frasier)

```
$ curl http://localhost:8080/api/authorities/paths/customer -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/paths/price -u frasier:password ②
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

① `frasier` has the `CUSTOMER` role assigned required to call `customer` URI

② `frasier` has the `PRICE_CHECK` permission assigned required to call `GET price` URI

6.3. Path-based Authority Permission (Sam and Woody)

The following example shows that users with the `ADMIN` and `CLERK` roles are able to call the `GET price` URI.

Path-based Authority Permission (Sam and Woody)

```
$ curl http://localhost:8080/api/authorities/paths/price -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/paths/price -u woody:password ②
[woody, [ROLE_CLERK]]
```

① `sam` is assigned the `ADMIN` role required to call the `GET price` URI

② `woody` is assigned the `CLERK` role required to call the `GET price` URI

6.4. Other Path Constraints

We can add a few other path constraints that do not directly relate to roles. For example, we can exclude or enable a URI for all callers.

- legacy `AccessDecisionManager` approach

  Other Path Constraints — Legacy AccessDecisionManager Approach

  ```
  http.authorizeRequests(cfg->cfg.antMatchers(
          "/api/authorities/paths/nobody/**")
          .denyAll()); ①
  http.authorizeRequests(cfg->cfg.antMatchers(
          "/api/authorities/paths/authn/**")
  ```

```
        .authenticated()); ②
```

① all callers of the `nobody` URI will be denied

② all authenticated callers of the `authn` URI will be accepted

- modern `AuthorizationManager` approach

Other Path Constraints — Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/nobody/**")
        .denyAll());
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/authn/**")
        .authenticated()); //thru customizer.builder ①
```

① `authenticated()` call constructs `AuthorizationManager` requiring authenticated caller

A few of the manual forms of configuring path-based access control are shown below to again highlight what the builder methods are conveniently constructing.

Other Path Constraints — Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/authn/**")
        .access(new AuthenticatedAuthorizationManager<>())); //using ctor ①
```

① `AuthenticatedAuthorizationManager` default to requiring authentication

Other Path Constraints — Modern AuthorizationManager Approach

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers(
            "/api/authorities/paths/authn/**")
        .access(AuthenticatedAuthorizationManager.authenticated())); //thru builder
①
```

① builder constructs default `AuthenticatedAuthorizationManager` instance

6.5. Other Path Constraints Usage

The following example shows a caller attempting to access the URIs that either deny all callers or accept all authenticated callers

```
$ curl http://localhost:8080/api/authorities/paths/authn -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/paths/nobody -u frasier:password ②
{"timestamp":"2020-07-14T18:09:38.669+00:00","status":403,
 "error":"Forbidden","message":"Forbidden","path":"/api/authorities/paths/nobody"}
```

```
$ curl http://localhost:8080/api/authorities/paths/authn ③
{"timestamp":"2020-07-14T18:15:24.945+00:00","status":401,
"error":"Unauthorized","message":"Unauthorized","path":"/api/authorities/paths/authn"}
```

① `frasier` was able to access the `authn` URI because he was authenticated

② `frasier` was not able to access the `nobody` URI because all have been denied for that URI

③ anonymous user was not able to access the `authn` URI because they were not authenticated

Chapter 7. Authorization

With that example in place, we can look behind the scenes to see how this occurred.

7.1. Review: FilterSecurityInterceptor At End of Chain

If you remember when we inspected the filter chain setup for our API during the breakpoint in FilterChainProxy.doFilterInternal() — there was a `FilterSecurityInterceptor` at the end of the chain. This is where our path-based authorization constraints get carried out.

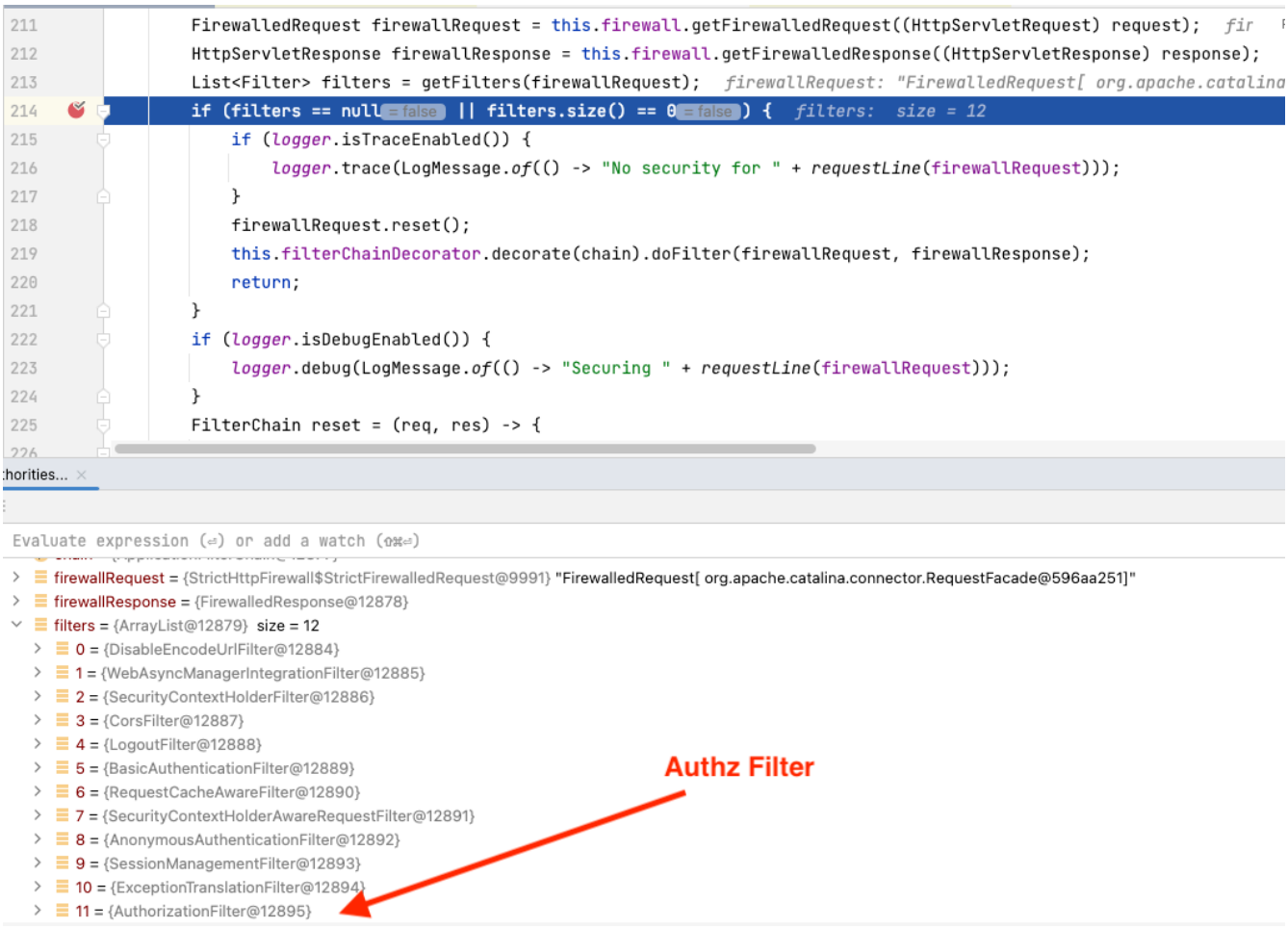

Figure 2. Review: FilterSecurityInterceptor At End of Chain

7.2. Attempt Authorization Call

We can set a breakpoint in the `AuthorizationFilter.doFilter()` method to observe the authorization process.

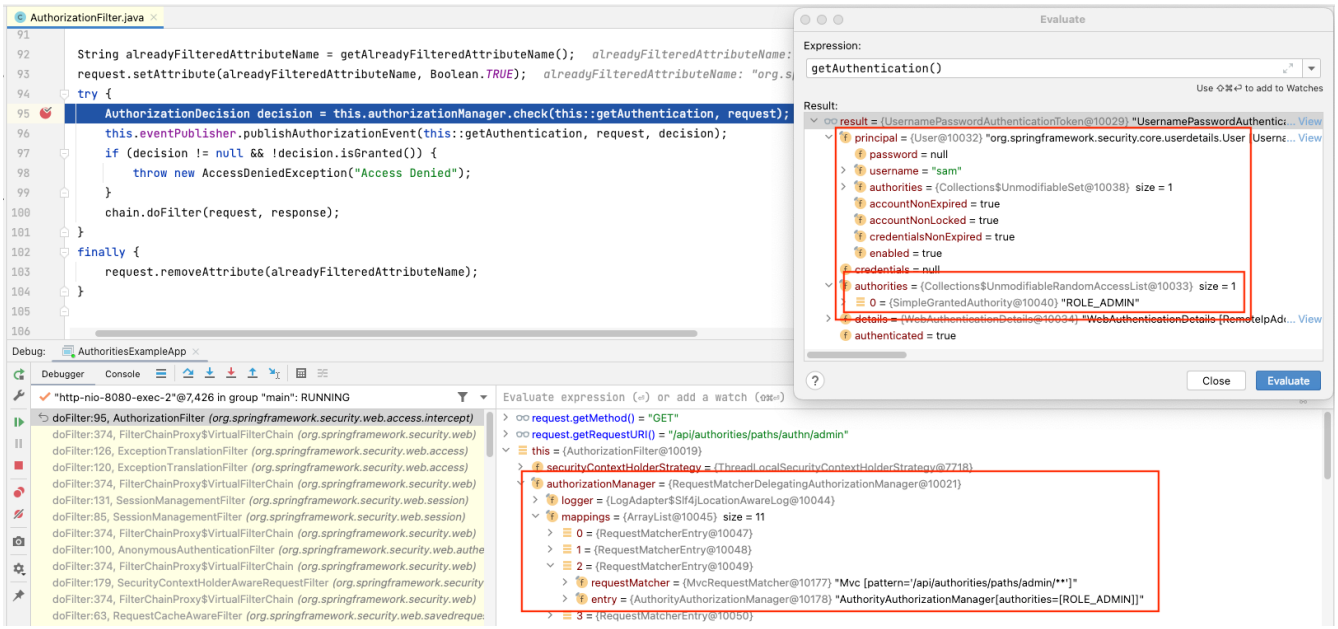

Figure 3. Authorization Call

7.3. FilterSecurityInterceptor Calls

- the `AuthorizationFilter` is at the end of the Security FilterChain and calls the `AuthorizationManager` to decide whether the caller has access to the target object. The call quietly returns without an exception if accepted and throws an `AccessDeniedException` if denied.
- the first `AuthorizationManager` is a RequestMatcherDelegatingAuthorizationManager that contains an ordered list of `AuthorizationManagers` that will be matched based upon the `RequestMatcher` expressions supplied.
- the matched `AuthorizationManager` is given a chance to determine user access based upon its definition.
- each matched `AuthorizationManager` returns an answer that is either granted, denied, or abstain.

Depending upon the parent/aggregate `AuthorizationManager`, one or all of the managers have to return a granted access.

Chapter 8. Role Inheritance

Role inheritance provides an alternative to listing individual roles per URI constraint. Let's take our case of `sam` with the `ADMIN` role. He is forbidden from calling the `customer` URI.

Admin Forbidden from Calling customer URI

```
$ curl http://localhost:8080/api/authorities/paths/customer -u sam:password
{"timestamp":"2020-07-14T20:15:19.931+00:00","status":403,"error":"Forbidden",
 "message":"Forbidden","path":"/api/authorities/paths/customer"}
```

8.1. Role Inheritance Definition

We can define a `@Bean` that provides a `RoleHierarchy` expressing which roles inherit from other roles. The syntax to this constructor is a String — based on the legacy XML definition interface.

Example Role Inheritance Definition

```
@Bean
public RoleHierarchy roleHierarchy() {
        return RoleHierarchyImpl.withDefaultRolePrefix()
                .role("ADMIN").implies("CLERK") ①
                .role("CLERK").implies("CUSTOMER") ②
                .build();
}
```

① role `ADMIN` will inherit all accessed applied to role `CLERK`

② role `CLERK` will inherit all accessed applied to role `CUSTOMER`

The `RoleHierarchy` component is automatically picked up by the various builders and placed into the runtime `AuthorityAuthorizationManagers`.

Automatically Injecting RoleHierarchy

```
...
    http.authorizeHttpRequests(cfg->cfg.requestMatchers(
                "/api/authorities/paths/customer/**")
            .hasAnyRole("CUSTOMER"));
...
```

If manually instantiating an `AuthorityAuthorizationManager`, you can also manually assign the `RoleHierarchy` by injecting it into the `@Bean` factory and adding it to the builder.

Manually Assigning RoleHierarchy

```
@Bean
@Order(0)
public SecurityFilterChain authzSecurityFilters(HttpSecurity http,
```

```
            RoleHierarchy roleHierarchy) throws Exception {

    //builder for use with custom access examples
    UnaryOperator<AuthorityAuthorizationManager> withRoleHierachy = (autho)->{
        autho.setRoleHierarchy(roleHierarchy);
        return autho;
    };

    http.authorizeHttpRequests(cfg->cfg.requestMatchers(
        "/api/authorities/paths/customer/**")
            .access(withRoleHierachy.apply(
                AuthorityAuthorizationManager.hasAnyRole("CUSTOMER"))));
```

With the above `@Bean` in place and restarting our application, users with role `ADMIN` or `CLERK` are able to invoke the `customer` URI.

Admin Inherits CUSTOMER ROLE

```
$ curl http://localhost:8080/api/authorities/paths/customer -u sam:password
[sam, [ROLE_ADMIN]]
```

As you will see in some of the next sections, `RoleHierarchy` is automatically picked up for some of the annotations as well.

Chapter 9. @Secured

As stated before, URIs are one way to identify a target meant for access control. However, it is not always the case that we are protecting a controller or that we want to express security constraints so far from the lower-level component method calls needing protection.

We have at least three options when implementing component method-level access control:

- @Secured
- JSR-250
- expressions

I will cover @Secured and JSR-250 first — since they have a similar, basic constraint capability and save expressions to the end.

9.1. Enabling @Secured Annotations

@Secured annotations are disabled by default.

- For legacy `AccessDecisionManager`, we can enable `@Secured` annotations by supplying a `@EnableGlobalMethodSecurity` annotation with `securedEnabled` set to true.

  Enabling @Secured in Legacy AccessDecisionManager

  ```
  import
  org.springframework.security.config.annotation.method.configuration.EnableGlobalMet
  hodSecurity

  @Configuration(proxyBeanMethods = false)
  @EnableGlobalMethodSecurity(
          securedEnabled = true  //@Secured({"ROLE_MEMBER"})
  )
  @RequiredArgsConstructor
  public class SecurityConfiguration {
  ```

- For modern `AuthorizationManager`, we can enable `@Secured` by supplying a `@EnableMethodSecurity` annotation with `securedEnabled` set to true.

  Enabling @Secured in Legacy AccessDecisionManager

  ```
  import
  org.springframework.security.config.annotation.method.configuration.EnableMethodSec
  urity;

  @Configuration(proxyBeanMethods = false)
  @EnableMethodSecurity(
          securedEnabled = true  //@Secured({"ROLE_MEMBER"})
  )
  ```

```
@RequiredArgsConstructor
public class ComponentBasedSecurityConfiguration {
```

9.2. @Secured Annotation

We can add the `@Secured` annotation to the class and method level of the targets we want protected. Values are expressed in authority value. Therefore, since the following example requires the `ADMIN` role, we must express it as `ROLE_ADMIN` authority.

Example @Secured Annotation

```
@RestController
@RequestMapping("/api/authorities/secured")
@RequiredArgsConstructor
public class SecuredAuthoritiesController {
    private final WhoAmIController whoAmI;

    @Secured("ROLE_ADMIN") ①
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
            @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user);
    }
```

① caller checked for `ROLE_ADMIN` authority when calling `doAdmin` method

9.3. @Secured Annotation Checks

`@Secured` annotation supports requiring one or more authorities in order to invoke a particular method.

Example @Secure Annotation Checks

```
$ curl http://localhost:8080/api/authorities/secured/admin -u sam:password ①
[sam, [ROLE_ADMIN]]
$ curl http://localhost:8080/api/authorities/secured/admin -u woody:password ②
{"timestamp":"2020-07-14T21:11:00.395+00:00","status":403,
 "error":"Forbidden","trace":"org.springframework.security.access.AccessDeniedExceptio
n: ...(lots!!!)
```

① `sam` has the required `ROLE_ADMIN` authority required to invoke `doAdmin`

② `woody` lacks required `ROLE_ADMIN` authority needed to invoke `doAdmin` and is rejected with an `AccessDeniedException` and a very large stack trace

9.4. @Secured Many Roles

@Secured will support many roles ORed together.

Example @Secured with Multiple Roles

```
@Secured({"ROLE_ADMIN", "ROLE_CLERK"})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

A user with either `ADMIN` or `CLERK` role will be given access to `checkPrice()`.

Example @Secured checkPrice()

```
$ curl http://localhost:8080/api/authorities/secured/price -u woody:password
[woody, [ROLE_CLERK]]

$ curl http://localhost:8080/api/authorities/secured/price -u sam:password
[sam, [ROLE_ADMIN]]
```

9.5. @Secured Now Processes Roles and Permissions

`@Secured` now supports both roles and permissions when using modern `AuthorizationManager`. That was not true with `AccessDecisionManager`.

@Secured Annotation for Roles and Permissions using Modern AuthorizationManager

```
@Secured({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"}) ①
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

① `PRICE_CHECK` permission will be honored

@Secured Annotation Support Roles for Permissions

```
$ curl http://localhost:8080/api/authorities/secured/price -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

curl http://localhost:8080/api/authorities/secured/price -u norm:password ②
{"timestamp":"2023-10-
09T20:51:21.264+00:00","status":403,"error":"Forbidden","trace":"org.springframework.s
ecurity.access.AccessDeniedException: Access Denied (lots ...)
```

① `frasier` can call URI `GET paths/price` because he has permission `PRICE_CHECK` and `@Secured` using `AuthorizationManager` supports permissions

② `norm` cannot call URI `GET secured/price` because he does not have permission `PRICE_CHECK`

9.6. @Secured Role Inheritance

Legacy `AccessDecisionManager` does not (or at least did not) support role inheritance for @Secured.

```
$ curl http://localhost:8080/api/authorities/paths/clerk -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password ②
{"timestamp":"2023-10-
09T20:55:29.651+00:00","status":403,"error":"Forbidden","trace":"org.springframework.s
ecurity.access.AccessDeniedException: Access Denied (lots!!!)
```

① `sam` is able to call `paths/clerk` URI because of `ADMIN` role inherits access from `CLERK` role

② `sam` is unable to call `doClerk()` method because `@Secured` does not honor role inheritance

Modern `AuthorizationManager` does support role inheritance for @Secured.

```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password && echo
[sam, [ROLE_ADMIN]]
```

Chapter 10. Controller Advice

When using URI-based constraints, 403/Forbidden checks were done before calling the controller and is handled by a default exception advice that limits the amount of data emitted in the response. When using annotation-based constraints, an `AccessDeniedException` is thrown during the call to the controller and is currently missing an exception advice. That causes a very large stack trace to be returned to the caller (abbreviated here with "...(lots!!!)").

Default AccessDeniedException result

```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password ②
{"timestamp":"2020-07-14T21:48:40.063+00:00","status":403,
"error":"Forbidden","trace":"org.springframework.security.access.AccessDeniedException
...(lots!!!)
```

10.1. AccessDeniedException Exception Handler

We can correct that information bleed by adding an `@ExceptionHandler` to address `AccessDeniedException`. In the example below I am building a string with the caller's identity and filling in the standard fields for the returned MessageDTO used in the error reporting in my `BaseExceptionAdvice`.

AccessDeniedException Exception Handler

```
...
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class ExceptionAdvice extends info.ejava.examples.common.web
.BaseExceptionAdvice { ①
    @ExceptionHandler({AccessDeniedException.class}) ②
    public ResponseEntity<MessageDTO> handle(AccessDeniedException ex) {
        String text=String.format("caller[%s] is forbidden from making this request",
                        getPrincipal());
        return this.buildResponse(HttpStatus.FORBIDDEN, null, text, (Instant)null);
    }

    protected String getPrincipal() {
        try { ③
            return SecurityContextHolder.getContext().getAuthentication().getName();
        } catch (NullPointerException ex) {
            return "null";
        }
    }
}
```

① extending base class with helper methods and core set of exception handlers

② adding an exception handler to intelligently handle access denial exceptions

③ `SecurityContextHolder` provides `Authentication` object for current caller

10.2. AccessDeniedException Exception Result

With the above `@ExceptionAdvice` in place, the stack trace from the `AccessDeniedException` has been reduced to the following useful information returned to the caller. The caller is told, what they called and who the caller identity was when they called.

AccessDeniedException Filtered through @ExceptionAdvice

```
$ curl http://localhost:8080/api/authorities/secured/clerk -u sam:password
{"url":"http://localhost:8080/api/authorities/secured/clerk","method":"GET","statusCod
e":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[sam] is
forbidden from making this request","timestamp":"2023-10-10T01:18:20.080250Z"}
```

Chapter 11. JSR-250

JSR-250 is an industry Java standard — also adopted by JakartaEE — for expressing common aspects (including authorization constraints) using annotations. It has the ability to express the same things as `@Secured` and a bit more. `@Secured` lacks the ability to express "permit all" and "deny all". We can do that with JSR-250 annotations.

11.1. Enabling JSR-250

JSR-250 authorization annotations are also disabled by default. We can enable them using the `@EnableGlobalMethodSecurity` or `@EnableMethodSecurity` annotations.

- For legacy `AccessDecisionManager`, we can enable them by supplying a `@EnableGlobalMethodSecurity` annotation with `securedEnabled` set to true.

  Enabling JSR-250 in Legacy AccessDecisionManager

  ```
  @Configuration(proxyBeanMethods = false)
  @EnableGlobalMethodSecurity(
          jsr250Enabled = true  //@RolesAllowed({"ROLE_MANAGER"})
  )
  @RequiredArgsConstructor
  public class SecurityConfiguration {
  ```

- For modern `AuthorizationManager`, we can enable them by supplying a `@EnableMethodSecurity` annotation with `securedEnabled` set to true.

  Enabling JSR-250 in Legacy AccessDecisionManager

  ```
  @Configuration(proxyBeanMethods = false)
  @EnableMethodSecurity(
          jsr250Enabled = true  //@RolesAllowed({"MANAGER"})
  )
  @RequiredArgsConstructor
  public class ComponentBasedSecurityConfiguration {
  ```

11.2. @RolesAllowed Annotation

JSR-250 has a few annotations, but its core `@RolesAllowed` is a 1:1 match for what we can do with `@Secured`. The following example shows the `doAdmin()` method restricted to callers with the admin role, expressed as its `ADMIN` value. Spring 6 change now longer allows the `ROLE_` prefix.

Example @RolesAllowed Annotation

```
@RestController
@RequestMapping("/api/authorities/jsr250")
@RequiredArgsConstructor
public class Jsr250AuthoritiesController {
```

```
    private final WhoAmIController whoAmI;

    @RolesAllowed("ADMIN") ①
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
            @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user);
    }
```

① Spring 6 not longer allows `ROLE_` prefix

11.3. @RolesAllowed Annotation Checks

The @RollsAllowed annotation is restricting callers of `doAdmin()` to have authority `ROLE_ADMIN`.

@RolesAllowed Annotation Checks

```
$ curl http://localhost:8080/api/authorities/jsr250/admin -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/jsr250/admin -u woody:password ②
{"url":"http://localhost:8080/api/authorities/jsr250/admin","method":"GET","statusCode
":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[woody] is
forbidden from making this request","timestamp":"2023-10-10T01:24:59.185223Z"}
```

① `sam` can invoke `doAdmin()` because he has the `ROLE_ADMIN` authority

② `woody` cannot invoke `doAdmin()` because he does not have the `ROLE_ADMIN` authority

11.4. Multiple Roles

The @RollsAllowed annotation can express multiple authorities the caller may have.

```
@RolesAllowed({"ROLE_ADMIN", "ROLE_CLERK", "PRICE_CHECK"})
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

11.5. Multiple Role Check

The following shows were both `sam` and `woody` are able to invoke `checkPrice()` because they have one of the required authorities.

JSR-250 Supports ORing of Required Roles

```
$ curl http://localhost:8080/api/authorities/jsr250/price -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/jsr250/price -u woody:password ②
```

```
[woody, [ROLE_CLERK]]
```

① `sam` can invoke `checkPrice()` because he has the `ROLE_ADMIN` authority

② `woody` can invoke `checkPrice()` because he has the `ROLE_ADMIN` authority

11.6. JSR-250 Does not Support Non-Role Authorities

JSR-250 authorization annotation processing does not support non-Role authorizations. The following example shows where `frasier` is able to call URI `GET paths/price` but unable to call `checkPrice()` of the JSR-250 controller even though it was annotated with one of his authorities.

JSR-250 Does not Support Non-Role Authorities

```
$ curl http://localhost:8080/api/authorities/paths/price -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/jsr250/price -u frasier:password ②
{"url":"http://localhost:8080/api/authorities/jsr250/price","method":"GET","statusCode
":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[frasier] is
forbidden from making this request","timestamp":"2023-10-10T01:25:49.310610Z"}
```

① `frasier` can invoke URI `GET paths/price` because he has the `PRICE_CHECK` authority and URI-based constraints support non-role authorities

② `frasier` cannot invoke JSR-250 constrained `checkPrice()` even though he has `PRICE_CHECK` permission because JSR-250 does not support non-role authorities

A reason for the non-support is that `@RollsAllowed("X")` (e.g., `PRICE_CHECK`) gets translated as role name and translated into "ROLE_X" (`ROLE_PRICE_CHECK`) authority value.

11.7. JSR-250 Role Inheritance

Modern `AuthorizationManager` supports role inheritance. I have not been able to re-test legacy `AccessDecisionManager`.

Modern `AuthorizationManager` Supports Role Inheritance

```
$ curl http://localhost:8080/api/authorities/jsr250/clerk -u sam:password && echo
[sam, [ROLE_ADMIN]]
```

Chapter 12. Expressions

As demonstrated, `@Secured` and JSR-250-based (`@RolesAllowed`) constraints are functional but very basic. If we need more robust handling of constraints we can use Spring Expression Language and Pre-/Post-Constraints. Expression support is enabled by default for modern `AuthorizationManager` when `@EnableMethodSecurity` is supplied but disabled by default for legacy `AccessDecisionManager` when `@EnableGlobalMethodSecurity` is used.

- For legacy `AccessDecisionManager`, we can enable them by supplying a `@EnableGlobalMethodSecurity` annotation with `prePostEnabled` set to true.

Enabling JSR-250 in Legacy AccessDecisionManager

```
@Configuration(proxyBeanMethods = false)
@EnableGlobalMethodSecurity(
  prePostEnabled = true //@PreAuthorize("hasAuthority('ROLE_ADMIN')"),
@PreAuthorize("hasRole('ADMIN'")
)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

- For modern `AuthorizationManager`, we can enable them by supplying a `@EnableMethodSecurity` annotation with `prePostEnabled` set to true. However, that is the default for this annotation. Therefore, it is unnecessary to define it as `true`.

Enabling JSR-250 in Legacy AccessDecisionManager

```
@Configuration(proxyBeanMethods = false)
@EnableMethodSecurity(
  prePostEnabled = true //@PreAuthorize("hasAuthority('ROLE_ADMIN')"),
@PreAuthorize("hasRole('ADMIN'")
)
@RequiredArgsConstructor
public class ComponentBasedSecurityConfiguration {
```

12.1. Expression Role Constraint

Expressions support many callable features, and I am only going to scratch the surface here. The primary annotation is `@PreAuthorize` and whatever the constraint is — it is checked prior to calling the method. There are also features to filter inputs and outputs based on flexible configurations. I will be sticking to the authorization basics and not be demonstrating the other features here. Notice that the contents of the string looks like a function call — and it is. The following example constrains the `doAdmin()` method to users with the role `ADMIN`.

Example Expression Role Constraint

```
@RestController
@RequestMapping("/api/authorities/expressions")
```

```
@RequiredArgsConstructor
public class ExpressionsAuthoritiesController {
    private final WhoAmIController whoAmI;

    @PreAuthorize("hasRole('ADMIN')") ①
    @GetMapping(path = "admin", produces = {MediaType.TEXT_PLAIN_VALUE})
    public ResponseEntity<String> doAdmin(
            @AuthenticationPrincipal UserDetails user) {
        return whoAmI.getCallerInfo(user);
    }
```

① `hasRole` automatically adds the `ROLE` prefix

12.2. Expression Role Constraint Checks

Much like @Secured and JSR-250, the following shows the caller being checked by expression whether they have the `ADMIN` role. The `ROLE_` prefix is automatically applied.

Example Expression Role Constraint

```
$ curl http://localhost:8080/api/authorities/expressions/admin -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/expressions/admin -u woody:password ②
{"url":"http://localhost:8080/api/authorities/expressions/admin","method":"GET","statu
sCode":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[woody]
is forbidden from making this request","timestamp":"2023-10-10T01:26:38.853321Z"}
```

① `sam` can invoke `doAdmin()` because he has the `ADMIN` role

② `woody` cannot invoke `doAdmin()` because he does not have the `ADMIN` role

12.3. Expressions Support Permissions and Role Inheritance

As noted earlier with URI-based constraints, expressions support non-role authorities and role inheritance for both legacy `AccessDecisionManager` and modern `AuthorizationManager`. The following example checks whether the caller has an authority and chooses to manually supply the `ROLE_` prefix.

Expressions Support non-Role Authorities

```
@PreAuthorize("hasAuthority('ROLE_CLERK')")
@GetMapping(path = "clerk", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> doClerk(
```

The following execution demonstrates that a caller with `ADMIN` role will be able to call a method that requires the `CLERK` role because we earlier configured `ADMIN` role to inherit all `CLERK` role accesses.

Example Expression Role Inheritance Checks

```
$ curl http://localhost:8080/api/authorities/expressions/clerk -u sam:password
[sam, [ROLE_ADMIN]]
```

12.4. Supports Permissions and Boolean Logic

Expressions can get very detailed. The following shows two evaluations being called and their result ORed together. The first evaluation checks whether the caller has certain roles. The second checks whether the caller has a certain permission.

Example Evaluation Logic

```
@PreAuthorize("hasAnyRole('ADMIN','CLERK') or hasAuthority('PRICE_CHECK')")
@GetMapping(path = "price", produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> checkPrice(
```

Example Evaluation Logic Checks for Role

```
$ curl http://localhost:8080/api/authorities/expressions/price -u sam:password ①
[sam, [ROLE_ADMIN]]

$ curl http://localhost:8080/api/authorities/expressions/price -u woody:password ②
[woody, [ROLE_CLERK]]
```

① `sam` can call `checkPrice()` because he satisfied the `hasAnyRole()` check by having the `ADMIN` role

② `woody` can call `checkPrice()` because he satisfied the `hasAnyRole()` check by having the `CLERK` role

Example Evaluation Logic Checks for Permission

```
$ curl http://localhost:8080/api/authorities/expressions/price -u frasier:password ①
[frasier, [PRICE_CHECK, ROLE_CUSTOMER]]
```

① `frasier` can call `checkPrice()` because he satisfied the `hasAuthority()` check by having the `PRICE_CHECK` permission

Example Evaluation Logic Checks

```
$ curl http://localhost:8080/api/authorities/expressions/customer -u norm:password ①
[norm, [ROLE_CUSTOMER]]

$ curl http://localhost:8080/api/authorities/expressions/price -u norm:password ②
{"url":"http://localhost:8080/api/authorities/expressions/price","method":"GET","statu
sCode":403,"statusName":"FORBIDDEN","message":"Forbidden","description":"caller[norm]
is forbidden from making this request","timestamp":"2023-10-10T01:27:16.457797Z"}
```

① `norm` can call `doCustomer()` because he satisfied the `hasRole()` check by having the `CUSTOMER` role

② `norm` cannot call `checkPrice()` because failed both the `hasAnyRole()` and `hasAuthority()` checks by not having any of the looked for authorities.

Chapter 13. Summary

In this module, we learned:

- the purpose of authorities, roles, and permissions
- how to express authorization constraints using URI-based and annotation-based constraints
- how to enforcement of the constraints is accomplished
- how the access control framework centers around a legacy `AccessDecisionManager` /`AccessDecisionVoter` and modern `AuthorizationManager` classes
- how to implement role inheritance for URI and expression-based constraints
- to implement an `AccessDeniedException` controller advice to hide necessary stack trace information and provide useful error information to the caller
- expression-based constraints are limitless in what they can express