Spring Security

Авторизация

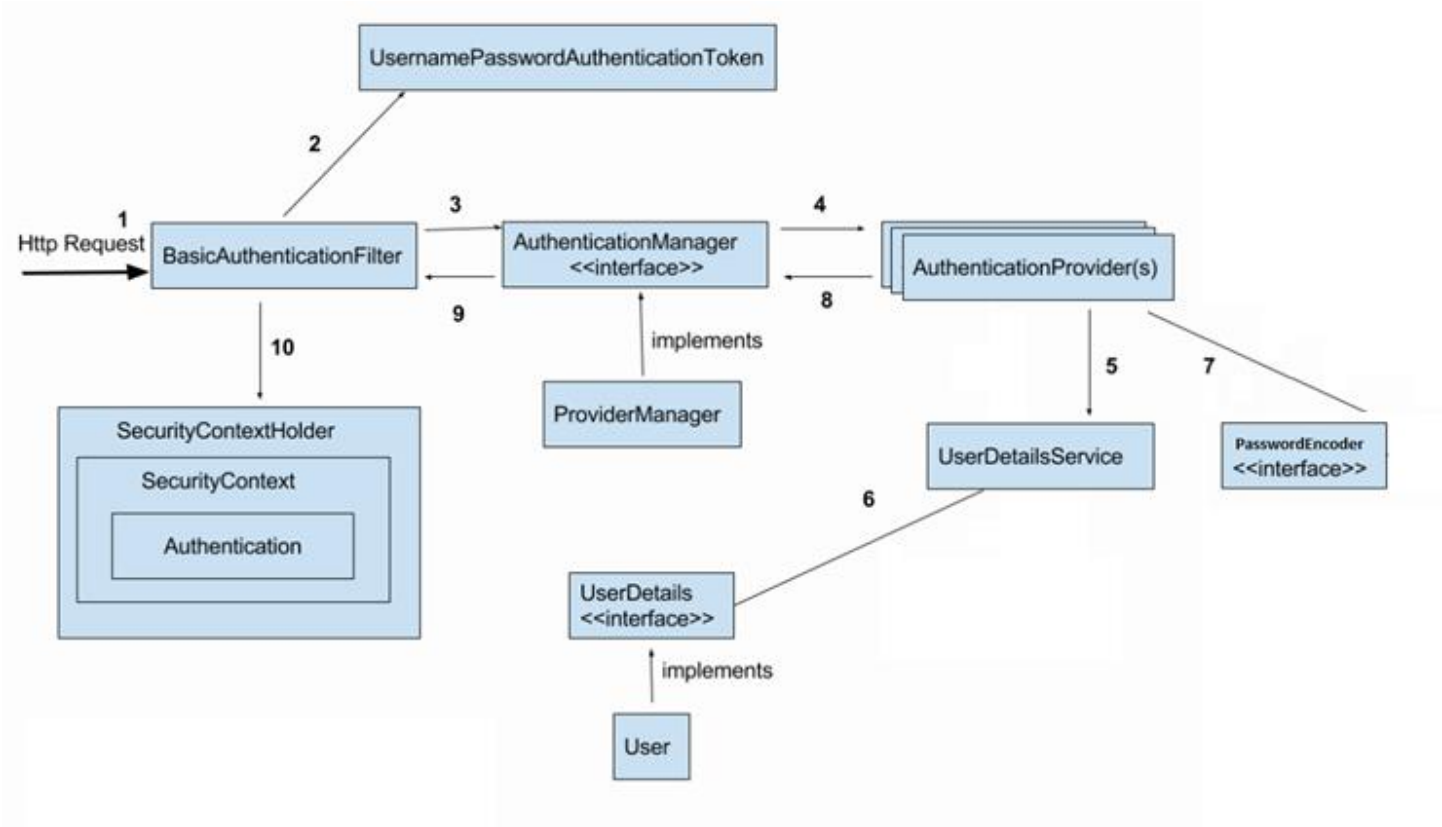
Идентификация, аутентификация, авторизация.

Идентификация – проверяет за кого пытается выдать себя пользователь

Аутентификация – проверяет действительно ли пользователь является тем, за кого он хочет себя выдать

Авторизация – проверяет какие права доступны аутентифицированному пользователю.

Архитектура аутентификации Servlet стека



Процесс авторизации происходит после аутентификации. Взаимодействие между этими двумя процессами происходит через SecurityContext. В интерфейсе Authentication есть метод

```
Collection<? extends GrantedAuthority> getAuthorities();
```

Он возвращает реализацию интерфейса GrantedAuthority, у которой только один метод

```
String getAuthority();
```

С точки зрения Spring Security это строка, для которой конфигурируются правила доступа в приложении.

Авторизация делится на 2 вида:

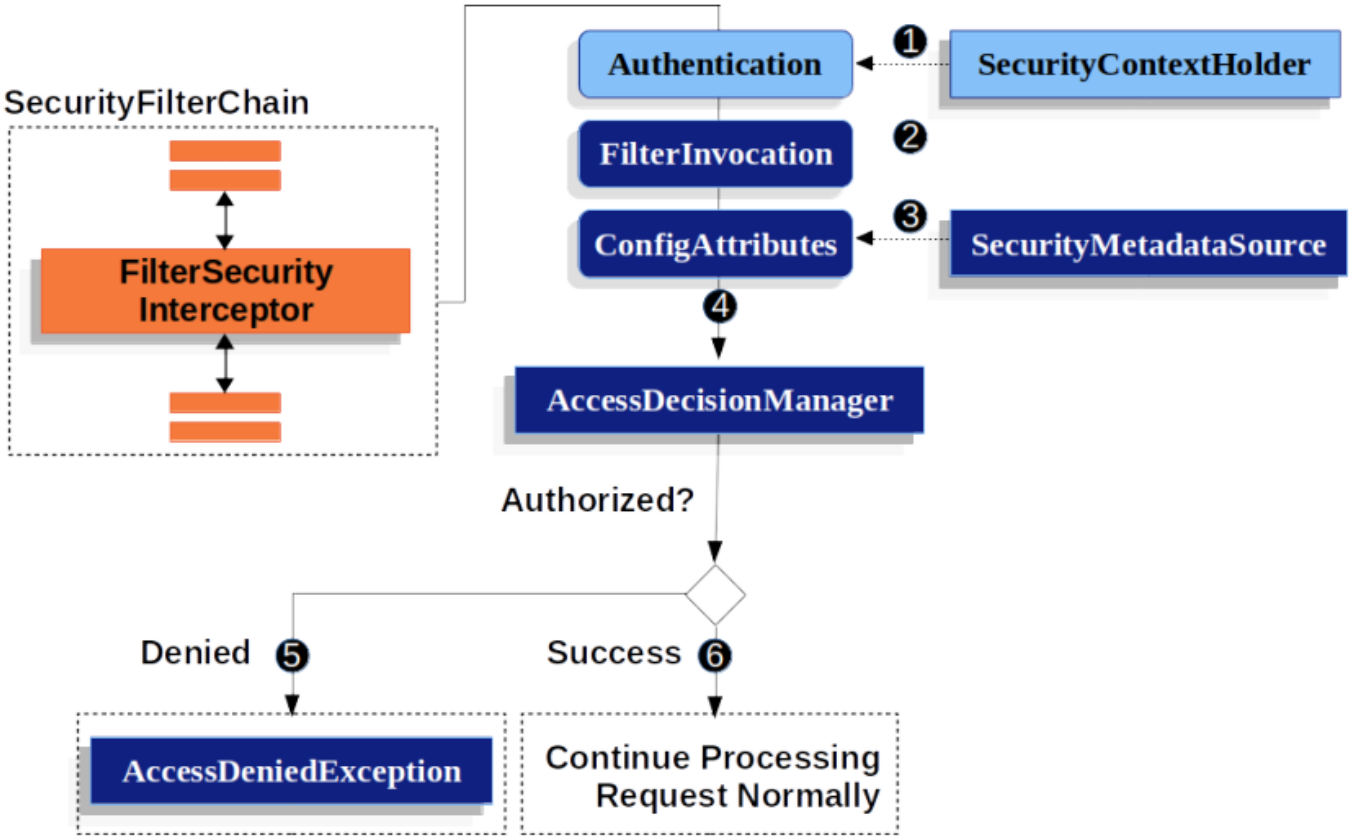
1. Авторизация по URL (Endpoint Level Authorization)
2. Авторизация по методам (Method Level Authorization)

Архитектура авторизации делится на устаревшую и новую (SS 5.5).

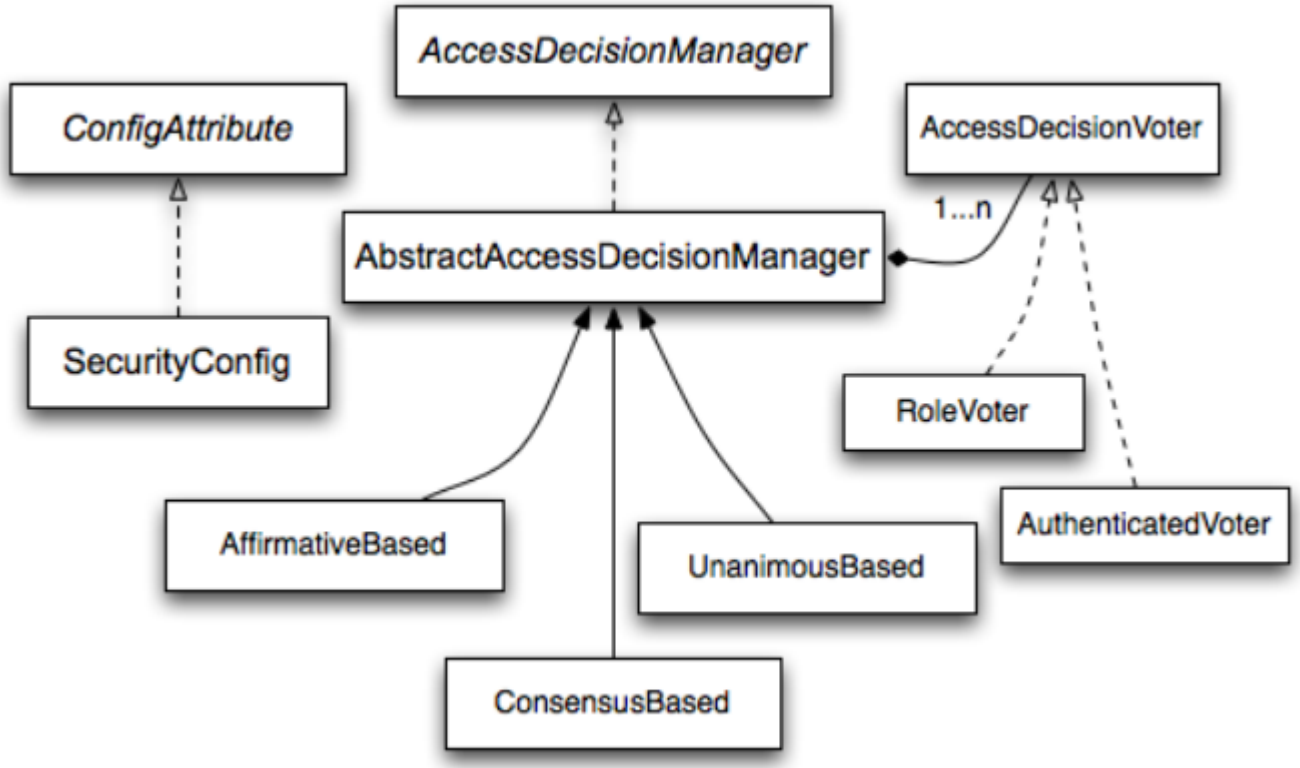
В устаревшей есть 2 этапа: Pre-Invocation Handling и After Invocation Handling**.** На этапе Pre работу по авторизации выполняет AccessDecisionManager, основываясь на AccessDecisionVoter. На этапе After работает AfterInvocationManager.

В новой архитектуре есть только AuthorizationManager

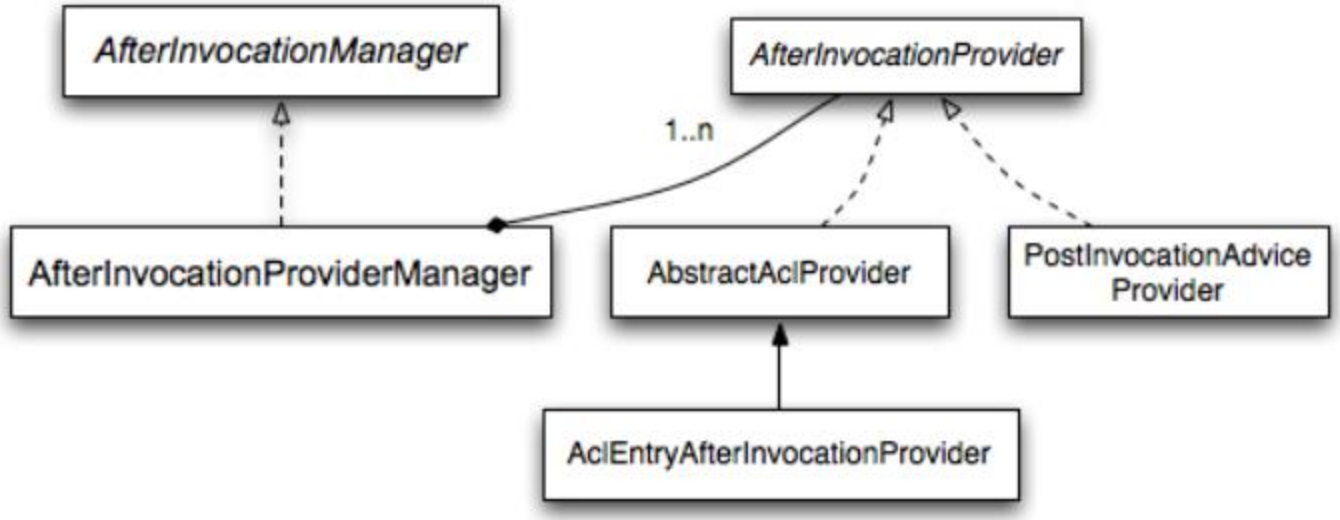
Устаревшая архитектура. Пример для авторизации по URL



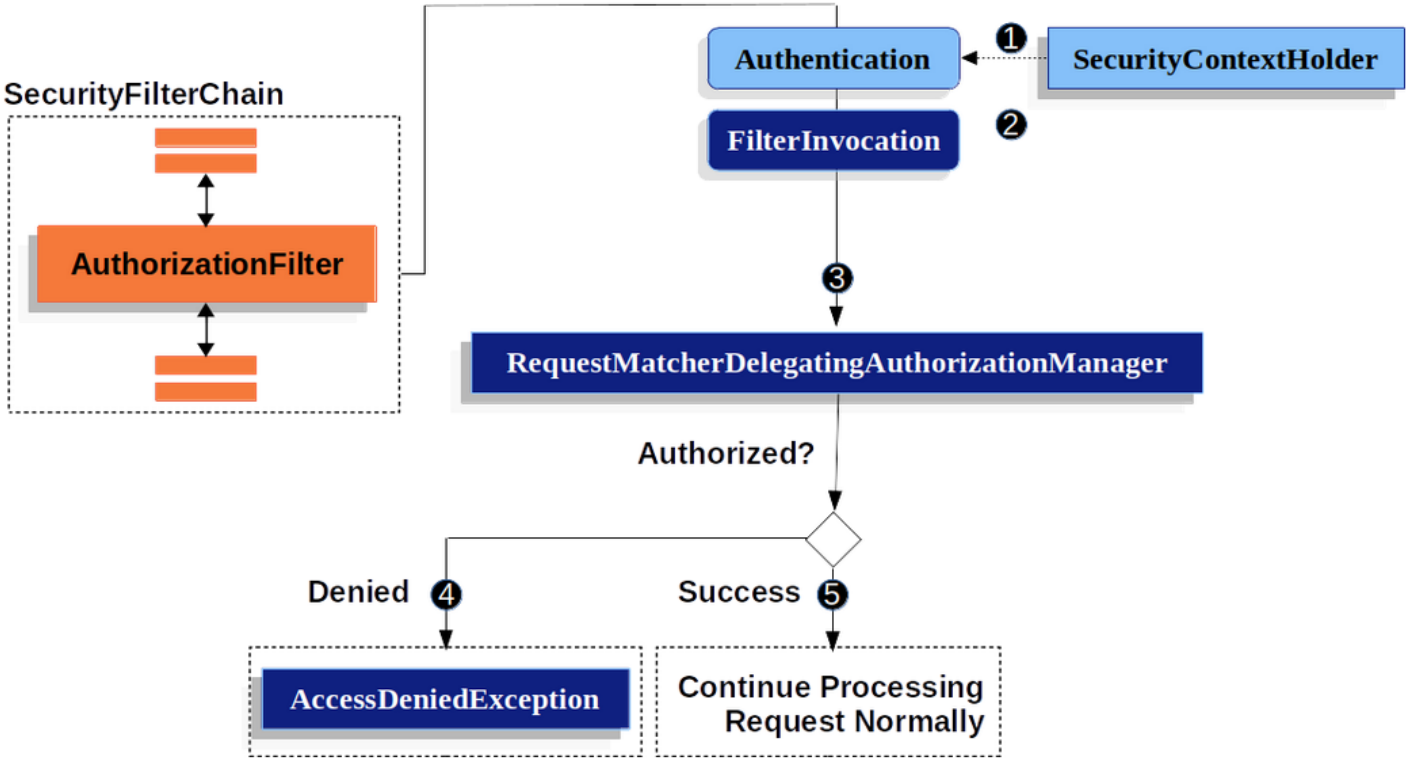
Реализации AccessDecisionManager



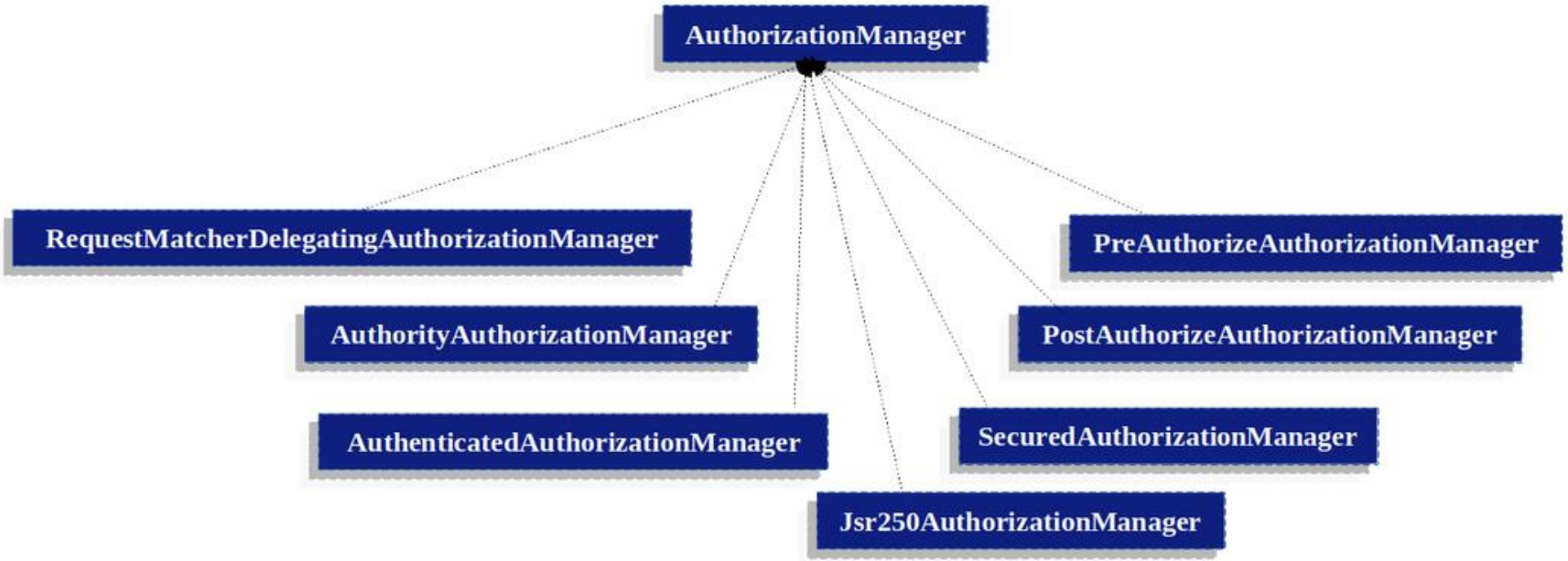
Реализации AfterInvocationManager



Новая архитектура. Пример для авторизации по URL



Реализации AuthorizationManager



Модель авторизации

Модель основывается на правах (authorities) и ролях (roles). С точки зрения Spring Security это только строки, пока мы для них не назначим какие-то правила.
Роль от права отличается наличием префикса “ROLE_”.

```
public interface GrantedAuthority extends Serializable {
```

> If the `GrantedAuthority` can be represented as a `String` and that `String` is sufficient in precision to be relied upon for an access control decision by an `AccessDecisionManager` (or delegate), this method should return such a `String`.
>
> If the `GrantedAuthority` cannot be expressed with sufficient precision as a `String`, `null` should be returned. Returning `null` will require an `AccessDecisionManager` (or delegate) to specifically support the `GrantedAuthority` implementation, so returning `null` should be avoided unless actually required.
>
> Returns: a representation of the granted authority (or `null` if the granted authority cannot be expressed

3 implementations

```
    String getAuthority();

}
```

Endpoint Level Authorization

За данный вид авторизации отвечает один из фильтров из SecurityFilterChain. В устаревшей реализации – это FilterSecurityInterceptor. В более новой – это AuthorizationFilter.

Для того, чтобы настроить URL, к которым требуется защищенный доступ, используются следующие матчеры:

1. anyRequest
2. antMatchers
3. mvcMatchers
4. regexMatchers

Используются ant-паттерны для указания URL

Endpoint Level Authorization

Для указания URL используются ant-паттерны

Expression	Description
`/a`	Only path /a.
`/a/*`	The * operator replaces one pathname. In this case, it matches /a/b or /a/c, but not /a/b/c.
`/a/**`	The ** operator replaces multiple pathnames. In this case, /a as well as /a/b and /a/b/c are a match for this expression.
`/a/{param}`	This expression applies to the path /a with a given path parameter.
`/a/{param:regex}`	This expression applies to the path /a with a given path parameter only when the value of the parameter matches the given regular expression.

Endpoint Level Authorization

Разница между mvcMatcher и antMatcher.

> `antMatcher(String antPattern)` - Allows configuring the `HttpSecurity` to only be invoked when matching the provided ant pattern.
>
> `mvcMatcher(String mvcPattern)` - Allows configuring the `HttpSecurity` to only be invoked when matching the provided Spring MVC pattern.

Generally `mvcMatcher` is more secure than an `antMatcher`. As an example:

- `antMatchers("/secured")` matches only *the exact* `/secured` URL
- `mvcMatchers("/secured")` matches `/secured` as well as `/secured/`, `/secured.html`, `/secured.xyz`

and therefore is more general and can also handle some possible configuration mistakes.

`mvcMatcher` uses the same rules that Spring MVC uses for matching (when using `@RequestMapping` annotation).

> If the current request will not be processed by Spring MVC, a reasonable default using the pattern as a ant pattern will be used. Source

It may be added that `mvcMatchers` API (since 4.1.1) is *newer* than the `antMatchers` API (since 3.1).

Endpoint Level Authorization

Для настройки правил доступа к URL используется следующие выражения:

1. hasRole(String role)
2. hasAnyRole(String… roles)
3. hasAuthority(String authority)
4. hasAnyAuthority(String… authorities)
5. permitAll()
6. denyAll()
7. access()
8. authenticated()

Endpoint Level Authorization

Так же есть иерархическая модель для ролей. Можно указать иерархию, которая позволит пользователя получать доступ к ресурсу, если у него нет конкретной роли, но есть более высокая в иерархии. Это настраивается следующим образом:

```
@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();

    roleHierarchy.setHierarchy("ROLE_ADMIN > ROLE_USER");

    return roleHierarchy;
}
```

Демо

Method Level Authorization

Данный вид авторизации работает за счет аспекта. Может использоваться не только в веб приложении. Позволяет ограничивать доступ и фильтровать входящие и исходящие параметры. Для активации требуется использовать аннотацию EnableMethodGlobalSecurity

```
public @interface EnableGlobalMethodSecurity {

    Determines if Spring Security's pre post annotations should be enabled. Default is false.
    Returns: true if pre post annotations should be enabled false otherwise.
    boolean prePostEnabled() default false;

    Determines if Spring Security's Secured annotations should be enabled.
    Returns: true if Secured annotations should be enabled false otherwise. Default is false.
    boolean securedEnabled() default false;

    Determines if JSR-250 annotations should be enabled. Default is false.
    Returns: true if JSR-250 should be enabled false otherwise.
    boolean jsr250Enabled() default false;

    Indicate whether subclass-based (CGLIB) proxies are to be created (true) as opposed to standard Java
    interface-based proxies (false). The default is false. Applicable only if mode() is set to AdviceMode.
    PROXY.
    Note that setting this attribute to true will affect all Spring-managed beans requiring proxying, not just
    those marked with the Security annotations. For example, other beans marked with Spring's
    @Transactional annotation will be upgraded to subclass proxying at the same time. This approach has
    no negative impact in practice unless one is explicitly expecting one type of proxy vs another, e.g. in tests.
    Returns: true if CGILIB proxies should be created instead of interface based proxies, else false
    boolean proxyTargetClass() default false;

    Indicate how security advice should be applied. The default is AdviceMode.PROXY.
    Returns:  the AdviceMode to use
    See Also: AdviceMode
    AdviceMode mode() default AdviceMode.PROXY;

    Indicate the ordering of the execution of the security advisor when multiple advices are applied at a
    specific joinpoint. The default is Ordered.LOWEST_PRECEDENCE.
    Returns: the order the security advisor should be applied
    int order() default Ordered.LOWEST_PRECEDENCE;

}
```

Method Level Authorization

Есть 3 вида аннотаций, которые можно использовать для авторизации доступа к методам.

1. @PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter
2. @Secured
3. @RolesAllowed

Каждый из видов включается дополнительно через атрибуты аннотации EnableMethodGlobalSecurity.
В Spring Security 5.6 появилась новая аннотация EnableMethodSecurity, которая имеет улучшения.

This improves upon `@EnableGlobalMethodSecurity` in a number of ways. `@EnableMethodSecurity`:

1. Uses the simplified `AuthorizationManager` API instead of metadata sources, config attributes, decision managers, and voters. This simplifies reuse and customization.
2. Favors direct bean-based configuration, instead of requiring extending `GlobalMethodSecurityConfiguration` to customize beans
3. Is built using native Spring AOP, removing abstractions and allowing you to use Spring AOP building blocks to customize
4. Checks for conflicting annotations to ensure an unambiguous security configuration
5. Complies with JSR-250
6. Enables `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, and `@PostFilter` by default

Демо

Method Level Authorization

В аннотациях @PreAuthorize, @PostAuthorize используется SpEL, можно указывать hasRole(), hasAuthority(). Для более сложных правил, используется hasPermission(). За логику, которая проверяется в процессе авторизации отвечает реализация PermissionEvaluator. Но гораздо легче создать свой собственный класс и вызывать его метод.

```
public interface PermissionEvaluator {

    boolean hasPermission(
              Authentication a,
              Object subject,
              Object permission);

    boolean hasPermission(
              Authentication a,
              Serializable id,
              String type,
              Object permission);
}
```

Демо

Method Level Authorization

Spring Security имеет интеграцию со Spring Data. Для использования этой функции необходимо добавить зависимость и создать бин.

```xml
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-data</artifactId>
</dependency>
```

```java
@Bean
public SecurityEvaluationContextExtension securityEvaluationContextExtension() {
    return new SecurityEvaluationContextExtension();
}
```

И можно интегрировать информацию из контекста в запросы

```java
@Query("""
select d from DocumentEntity d where d.ownerName = ?#{authentication.name}
""")
```

Демо