Spring Security

Сергей Киченко, апрель 2017г.

Содержание

1. Введение в **Spring Security**

Проект **Spring Security** появился в 2003 г. и назывался **"The Acegi Security System for Spring"**.

В конце 2007 г. переименован в **"Spring security"**.

Spring Security - это фреймворк, обеспечивающий декларативную безопасность **Java EE** приложений (модуль можно использовать не только в Spring based приложениях).

Фреймворк условно разделяют на 2 составляющие:

- защита web-запросов и распределение прав по URL (в основе сервлет-фильтры)
- защита вызовов методов (в основе AOP, прокси-объекты)

1.1 Основные модули

Модуль	Описание
Core (spring-security-core.jar)	Содержит базовые классы и интерфейсы для аутентификации и авторизации. Пакеты: org.springframework.security.core, org.springframework.security.access, org.springframework.security.authentication, org.springframework.security.provisioning
Config (spring-security-config.jar)	Все, что связано с парсингом java, xml конфигураций. Пакеты: org.springframework.security.config
Web (spring-security-web.jar)	Содержит фильтры и относящуюся к web-защите инфраструктуру кода (фильтры, servlet-api зависимости). Пакеты: org.springframework.security.web
Test (spring-security-test.jar)	Все, что связано с тестированием модуля
Remoting (spring-security-remoting.jar)	Интеграция с модулем Spring Remoting (RMI). Пакеты: org.springframework.security.remoting
ACL (spring-security-acl.jar)	Защита доменных объектов с помощью access-control-list. Пакеты: org.springframework.security.acls

Доп. модули: **LDAP (spring-security-ldap.jar), CAS (spring-security-cas.jar), OpenID (spring-security-openid.jar), OAuth (spring-security-oauth.jar), SAML (spring-security-saml.jar)** и другие.

2. Authentication vs. Authorization

Аутентификация - это процедура проверки подлинности

Примеры:

- проверка подлинности пользователя путём сравнения введённого им пароля с паролем, сохранённым в базе данных пользователей;
- подтверждение подлинности электронного письма путём проверки цифровой подписи письма по открытому ключу отправителя;
- проверка контрольной суммы файла на соответствие сумме, заявленной автором этого файла.

Авторизация - предоставление прав на выполнение определённых действий; а также процесс проверки данных прав при попытке выполнения этих действий

Пример:

- доступ на защищенный раздел сайта

Авторизацию не следует путать с аутентификацией: аутентификация — это процедура проверки легальности пользователя или данных, **авторизация** - это контроль доступа к ресурсам системы после успешного прохождения ими аутентификации. Процедуры **аутентификации** и **авторизации** обычно совмещаются.

3. Конфигурирование (java configuration)

Для работы со **Spring Security** модуль нужно сконфигурировать. Для примера рассмотрим конфигурацию на основе **java**.

Существуют две модели конфигурации **WebSecurity** и **MethodSecurity**.

- **WebSecurity** - конфигурирование защиты web-запросов.
- **MethodSecurity** - конфигурирование защиты вызова методов.

3.1 Конфигурирование **WebSecurity**

В модели web-защиты фреймворк использует **сервлет-контейнер**, а точнее фильтры (**Filter**).

Итак при старте сервлет-контейнера регистрируется фильтр **springSecurityFilterChain**, который и формирует инфраструктуру для защиты web-запросов (аутентификация, авторизация), конечно же для этого его нужно сконфигурировать.

Здесь важны 2 шага:

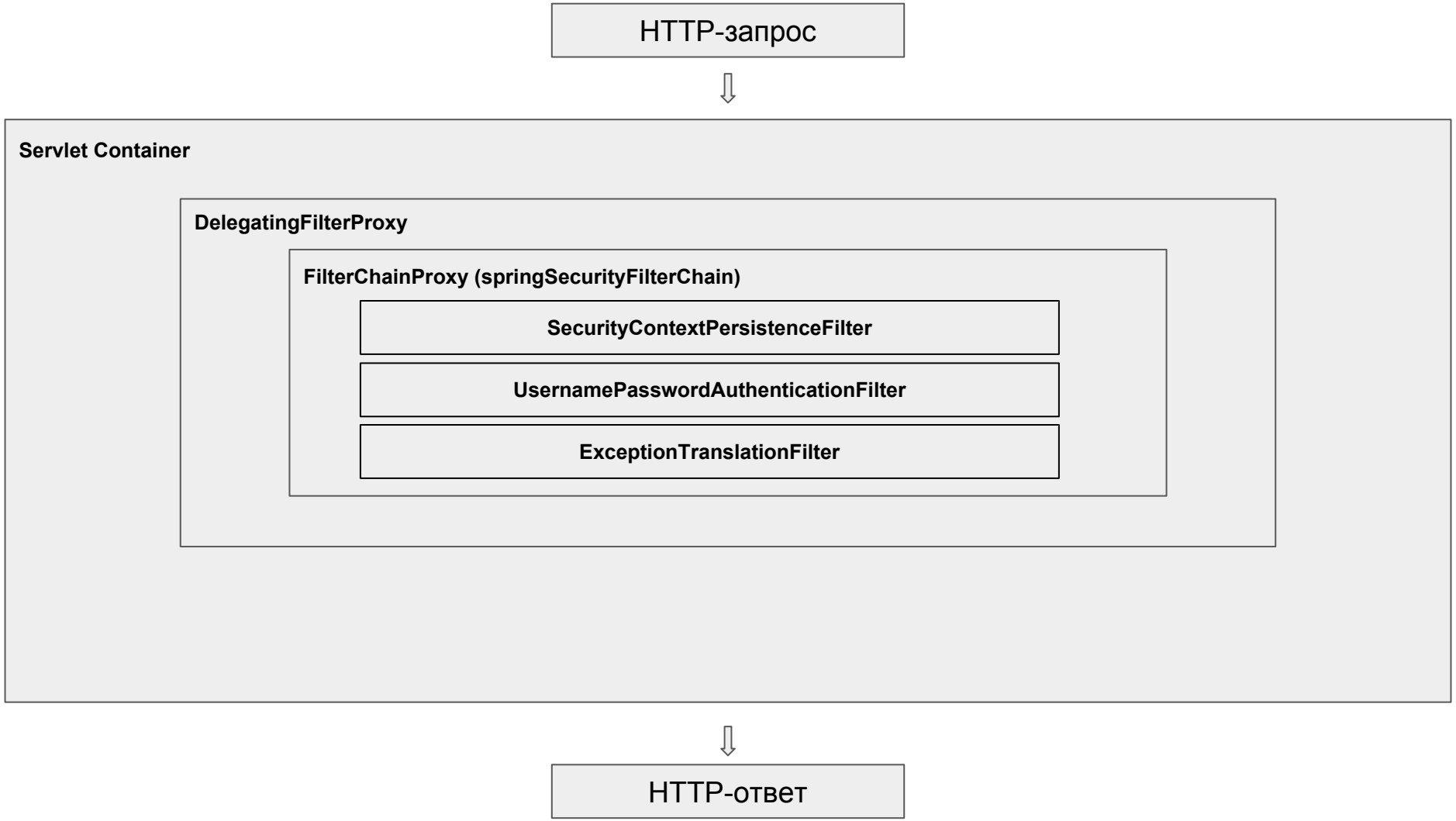
1. создание конфигурации (пример конфигурации **ru.kichenko.exampe.WebSecurityConfig**)
2. регистрация фильтра **springSecurityFilterChain** в самом web приложении

В обычных spring-web приложениях используется **AbstractSecurityWebApplicationInitializer** для регистрирации фильтра **FilterChainProxy** (**springSecurityFilterChain**).

В **spring-boot** используется конфигурация **SecurityFilterAutoConfiguration** создается **DelegatingFilterProxyRegistrationBean** и далее вызывается динамическая регистрация в сервлет-контейнере (например embed контейнер см. **EmbeddedWebApplicationContext** и **TomcatStarter**).

Регистрация в сервлет-контейнере происходит через фильтр **DelegatingFilterProxy**.

Важно понимать, что любой фильтр spring-security (**BasicAuthenticationFilter**, **ExceptionTranslationFilter** и т.д.) выполняется через **FilterChainProxy.**


HTTP-запрос
Servlet Container
DelegatingFilterProxy
FilterChainProxy (springSecurityFilterChain)
SecurityContextPersistenceFilter
UsernamePasswordAuthenticationFilter
ExceptionTranslationFilter
HTTP-ответ

3.2 Конфигурирование **MethodSecurity**

См. пример конфигурации **ru.kichenko.exampe.WebSecurityConfig**

4. Архитектура и реализация
4.1 Аутентификация

4.1.1 Сценарий аутентификации в **Spring Security**

Пользователь вводит логин&пароль
Логин и пароль записываются в специальный токен (например в **UsernamePasswordAuthenticationToken**)
см. **Authentication**

Проверка пароля пользователя (успешная)
Сформированный токен передается для валидации в сервис **AuthenticationManager**

Заполнение контекста безопасности информацией о пользователе (роли и т.д.)
В случае успешной валидации токен заполняется информацией о пользователе (например роли) и возвращается **AuthenticationManager**. Здесь же устанавливается контекст безопасности, вызовом метода **SecurityContextHolder.getContext().setAuthentication(…)**

Пользователь аутентифицирован и получает доступ к защищенным ресурсам, операциям (при условии, что у него есть на это соотв. права, а это уже авторизация)

4.1.2 Ключевые компоненты (**SecurityContextHolder**, **SecurityContext**, **Authentication, GrantedAuthority**)

SecurityContextHolder

SecurityContextHolder - фундаментальный компонент, хранящий контекст безопасности приложения.

Стратегии для хранения контекста безопасности:

- **MODE_THREADLOCAL** (по умолчанию; хранит контекст в **ThreadLocal**; для web-приложений)
- **MODE_INHERITABLETHREADLOCAL** (тоже, что и MODE_THREADLOCAL, но порождаемые потоки имеют тот же контекст безопасности)
- **MODE_GLOBAL** (для standalone приложений; глобальный контекст безопасности)

Как установить страгетию:

- через св-во **spring.security.strategy**
- **SecurityContextHolder.setStrategyName(...)**

Установка контекста безопасности - **Spring Security** не определяет точное место где должен быть установлен контекст безопасности, но сделано это должно быть до того как **AbstractSecurityInterceptor** будет авторизовывать действия пользователя, обычно контест устанавливается в **AuthenticationManager**.

SecurityContext и Authentication

SecurityContext - контекст безопасности приложения. Хранит токен аутентификации с информацией о пользователе.

SecurityContext ctx = SecurityContextHolder.getContext() - получение контекста безопасности

Authentication - токен аутентификации с информацией о пользователе

SecurityContextHolder.getContext().setAuthentication(...) - установить токен в контекст безопасности

Authentication auth = SecurityContextHolder.getContext().getAuthentication(...) - получить токен из контекста безопасности

Authentication.getAuthorities() - возвращает набор ролей (**GrantedAuthority**)

Authentication.getCredentials() - учетные данные (обычно пароль)

Authentication.getDetails() - возвращает доп. детали (например *ip-адрес, id-сессии и т.д.*)

Authentication.getPrincipal() - возвращает пользователя (обычно это объект **UserDetails**)

Authentication.isAuthenticated() - возвращает признак аутентификации

Authentication.setAuthenticated(...) - устанавливает флаг аутентификации (обычно вызывается в **AuthenticationProvider**)

4.1.3 Ключевые сервисы (**AuthenticationManager**, **ProviderManager**, **AuthenticationProvider**, **UserDetailsService**)

AuthenticationManager - интерфейс для реализации компонента обрабатывающего запросы на аутентификацию. Дефолтная реализация **ProviderManager** содержит в себе набор **AuthenticationProvider**'ов, которые фактически и выполняют аутентификацию, каждый **AuthenticationProvider** возвращает null, порождает AuthenticationException или заполненный токен аутентификации.

AuthenticationManager

- ProviderManager

AuthenticationProvider

- DaoAuthenticationProvider
- RememberMeAuthenticationProvider

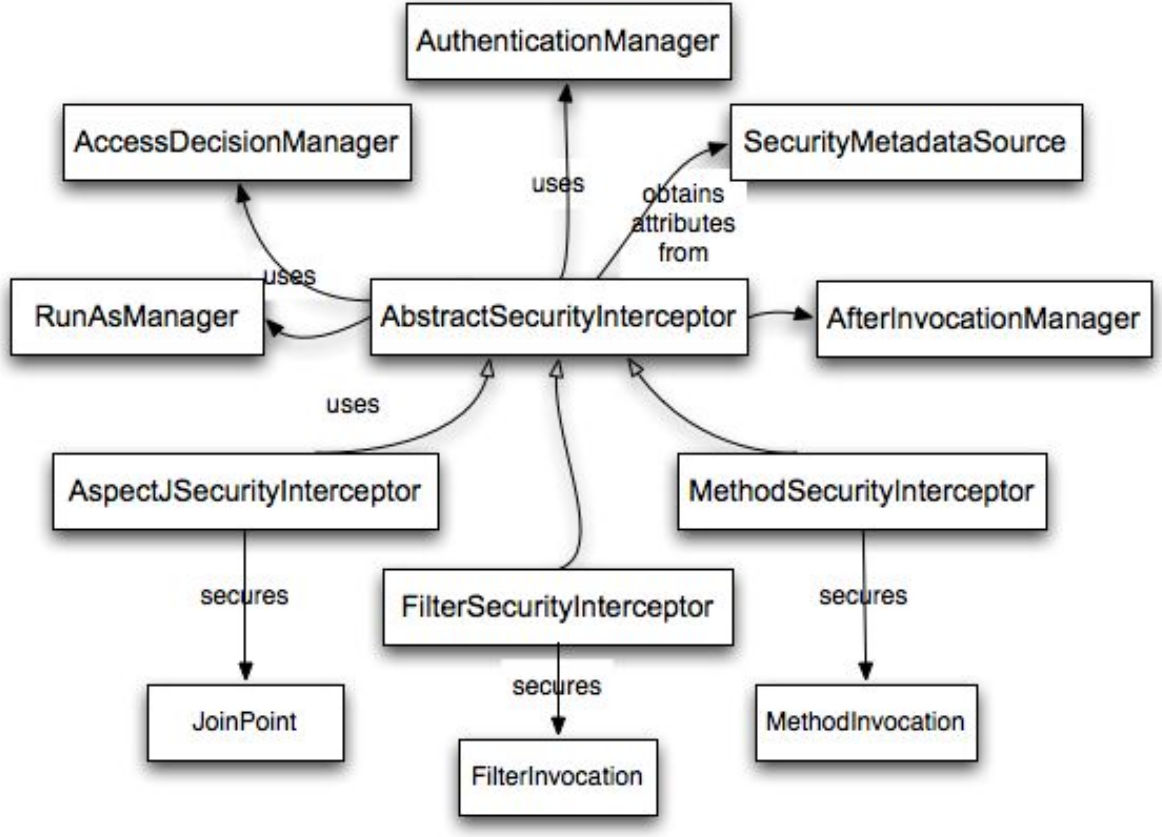
Защита пароля в **AuthenticationProvider** используется **PasswordEncoder** с хеш-функцией **bcrypt** (**BCryptPasswordEncoder**).

UserDetailsService - сервис для получения деталей пользователя (**UserDetail**).

- JdbcUserDetailsManager
- InMemoryUserDetailsManager

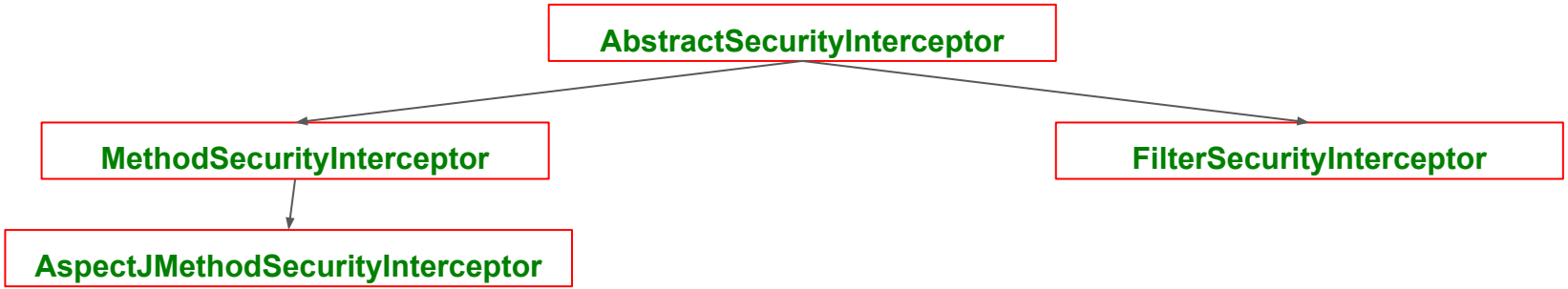
4. Архитектура и реализация
4.2 Authorization

4.2.1 Ключевые компоненты и сервисы (**Authorities**, **AccessDecisionManager**, **RoleVoter**, **HierarchicalRoles**)



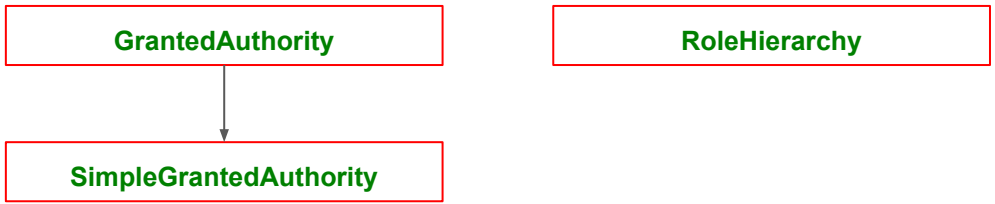
Стандартный сценарий работы **AbstractSecurityInterceptor**

1. Поиск атрибутов конфигурации ассоциированных с текущим запросом
2. Передаем объект, токен аутентификации и атрибуты конфигурации в **AccessDecisionManager** для принятия решения об авторизации запроса
3. Опционально подменяем текущую аутентификацию другой
4. Вызываем защищенный объект, если авторизация успешна
5. Если авторизация успешна - вызываем **AfterInvocationManager**, если он сконфигурирован



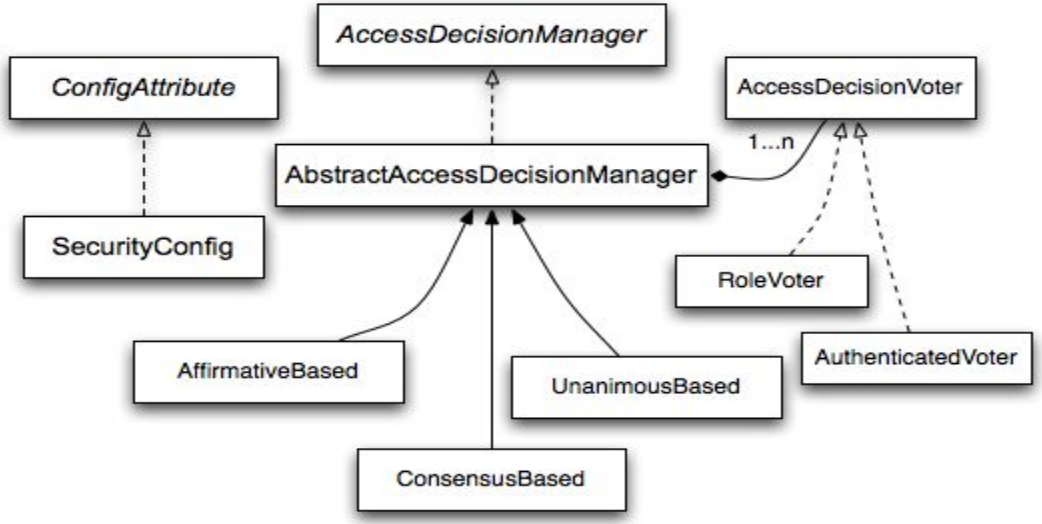
GrantedAuthority

String getAuthority() - возвращает **string || null**



AccessDecisionManager

void decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes) throws AccessDeniedException



RoleHierarchyVoter

AccessDecisionVoter

RoleVoter
префикс “ROLE_”

Jsr250Voter

WebExpressionVoter

AuthenticatedVoter
IS_AUTHENTICATED_FULLY
IS_AUTHENTICATED_REMEMBERED
IS_AUTHENTICATED_ANONYMOUSLY

4.2.2 Использование SpEL для защиты (**WebSecurityExpressions, MethodSecurityExpressions, JSR-250**, **@PreAuthorize**, **@PostAuthorize**, **@PreFilter**, **@PostFilter**)

SecurityExpressionRoot

SecurityExpressionRoot - определяет базовый функционал для **MethodSecurityExpressionRoot** и **WebSecurityExpressionRoot**.

WebSecurityExpressionRoot

WebSecurityExpressionRoot - расширяет **SecurityExpressionRoot** и добавляет пару методов

```
http.authorizeRequests().antMatchers("/user/{userId}/**").access("@webSecurity.checkUserId(authentication,#userId)")
```

MethodSecurityExpressionRoot

MethodSecurityExpressionRoot - расширяет **SecurityExpressionRoot** и добавляет несколько методов get/set - FilterObject get/set - ReturnObject.

@Secured, @PreAuthorize, @PreFilter

```
@Secured("ROLE_ADMIN")
public void create(Contact contact);

@PreAuthorize("hasRole('USER')")
public void create(Contact contact);

@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("#contact.name == authentication.name")
public @interface ContactPermission {}

@PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission permission);
```

@PostAuthorize, @PostFilter

```
@PreAuthorize("hasRole('USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")
public List<Contact> getAll();
```

Аннотации JSR-250: **@RolesAllowed, @PermitAll, @DenyAll, @DeclareRoles**

```
@RolesAllowed("ROLE_ADMIN")
public void create(Contact contact);

@PermitAll
public void create(Contact contact);
```

Интерфейс **PermissionEvaluator** (**hasPermission(...)**)

PermissionEvaluator - это мост между **SpEL** и **Spring Security ACL**, при этом никаких зависимостей от **ACL** модуля нет и в принципе можно реализовать кастомную проверку **hasPermission(...)**. Реализация по умолчанию **DenyAllPermissionEvaluator** ничего не делает.

```
@PreAuthorize("hasPermission(#contact, ‘create’)")
public void create(Contact contact);

@PreAuthorize("hasPermission(#contact.id, 'com.example.domain.Contact, ‘create’)")
public void create(Contact contact);
```

4.3 Защита web-приложений

4.3.1 Ключевые компоненты и базовые security-фильтры

Web инфраструктура **Spring Security** основана на стандартных фильтрах (см. слайд №8).

DelegatingFilterProxy -> FilterChainProxy (springSecurityFilterChain) -> Набор фильтров безопасности

Порядок фильтров

- **ChannelProcessingFilter** - проверяет, что web-запрос идет через нужный канал (http.requiresChannel().antMatchers("/login*").requiresSecure())
- **SecurityContextPersistenceFilter** - отвечает за хранение **SecurityContext** между HTTP-запросами (заполнение и очистка контекста), использует **SecurityContextRepository** (**HttpSessionSecurityContextRepository** - дефолтная реализация, использует **HttpSession** для хранения контекста безопасности).
- **ConcurrentSessionFilter** - для корректной обработки параллельных обращений к сессии
- **Фильтры аутентификации** - фильтры для аутентификации, могут быть разные в зависимости от типа аутентификации (**UsernamePasswordAuthenticationFilter**, **BasicAuthenticationFilter**)
- **SecurityContextHolderAwareRequestFilter** - фильтр заполняющий **ServletRequest** враппером, реализующим **ServletAPI security methods**.
- **RememberMeAuthenticationFilter** - фильтр для обработки remember-me аутентификации
- **AnonymousAuthenticationFilter** - фильтр анонимной аутентификации, **SecurityContextHolder** всегда хранит токен аутентификации, в данном случае это будет анонимный токен аутентификации
- **ExceptionTranslationFilter** - фильтр для обработки ошибок аутентификации/авторизации
- **FilterSecurityInterceptor** - фильтр отвечающий за авторизацию web-запросов

FilterSecurityInterceptor

Фильтр отвечающий за авторизацию web-запросов.

- **AuthenticationManager**
- **AccessDecisionManager**
- **SecurityMetadataSource** (**FilterInvocationSecurityMetadataSource**)

ExceptionTranslationFilter

Данный фильтр находится над **FilterSecurityInterceptor** и обрабатывает ошибки аутентификации | авторизации.

- **AuthenticationEntryPoint** - вызывается, если запрос не прошел аутентификацию (**AuthenticationException** | **AccessDeniedException**) Результатом будет например редирект на login URL (для **LoginUrlAuthenticationEntryPoint**)
- **AccessDeniedHandler** - вызывается если юзер аутентифицирован, но у пользователя недостаточно прав (**AccessDeniedException**) результатом будет http code 403 (**AccessDeniedHandlerImpl**)
- **RequestCache** - сохраняет запрос в кеше перед вызовом **AuthenticationEntryPoint**, что позволяет восстановить его поле успешной аутентфикации пользователем (например редирект на url с которого пришли **SavedRequestAwareAuthenticationSuccessHandler**)

Фильтр аутентификации (на примере UsernamePasswordAuthenticationFilter)

Фильтр вызывает сконфигурированный **AuthenticationManager**, далее в зависимости от результата аутентификации вызываются обработчик **AuthenticationSuccessHandler** или **AuthenticationFailureHandler**.

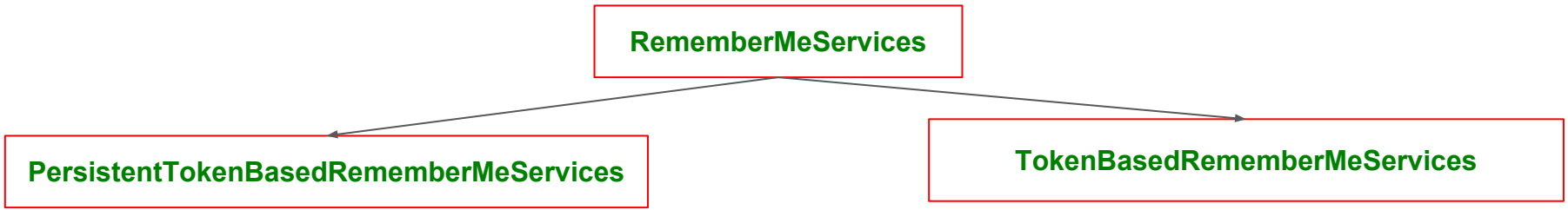
- **SavedRequestAwareAuthenticationSuccessHandler** - если в кеше есть прежний запрос получает из кеша и редирект на url с которого пришли
- **SimpleUrlAuthenticationFailureHandler** - редирект на url или 401 http code

Аутентификация через фильтр **RememberMeAuthenticationFilter**

Функционал реализован через хуки в базовом классе фильтра аутентификации (**AbstractAuthenticationProcessingFilter**)

Используется в **UsernamePasswordAuthenticationFilter** и **BasicAuthenticationFilter** фильтрах.

- **UsernamePasswordAuthenticationFilter.setRememberMeServices()**
- **AuthenticationManager.setProviders(RememberMeAuthenticationProvider)**
- Добавить **RememberMeAuthenticationFilter** после **UsernamePasswordAuthenticationFilter**

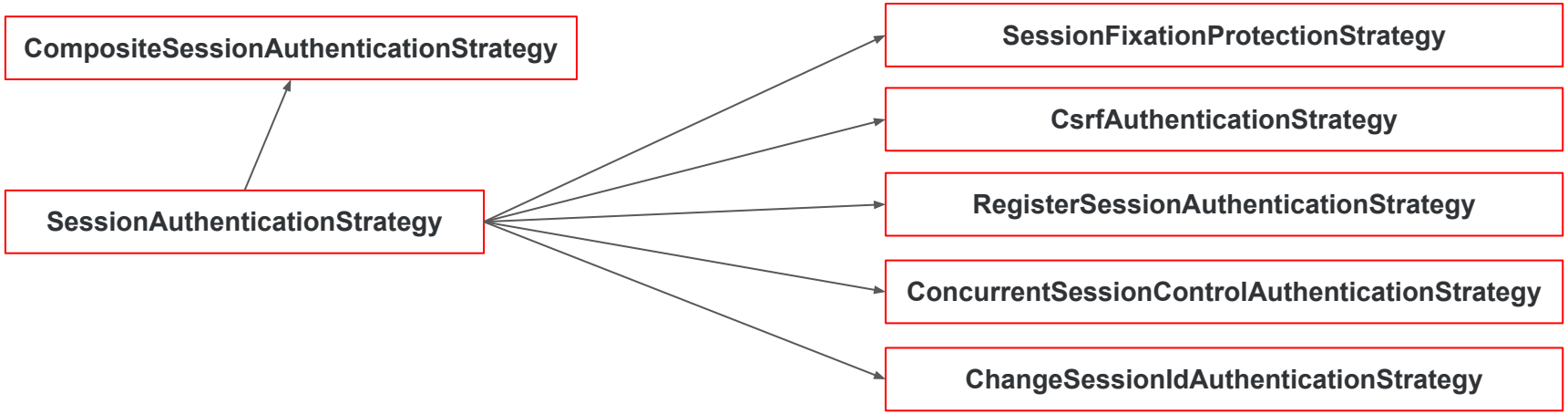


Менеджмент сессий (**SessionManagementFilter**)

Работа с **HTTP-session** относится к **SessionManagementFilter** и **SessionAuthenticationStrategy**.

- session-fixation protection attack prevention
- session timeouts
- concurrent session limit

SessionManagementFilter проверяет наличие токена аут. в **SecurityContextRepository** (**HttpSessionSecurityContextRepository**), если токена аутентификацции нет, то фильтр ничего не делает. Если нашелся, проверяется токен из **SecurityContextHolder**, если он содержит не анонимный токен, тогда вызывается **SessionAuthenticationStrategy**.
Если текущий пользователь не аутентифицирован сессия проверяется на валидность (например тайм-аут) и в случае невалидной сессии вызывается **InvalidSessionStrategy**.



Анонимная аутентификация (**AnonymousAuthenticationFilter**)

Cross Site Request Forgery (CSRF, XSRF) [CsrfFilter]

CSRF - межсайтовая подделка запроса.
Защита от CSRF:

- Правильно использовать HTTP verbs (PATCH, POST, PUT, DELETE для всего, что модифицирует данные)
- Конфигурирование CSRF Protection
- Использование CSRF Token

Конфигурирование CSRF Protection:


```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.csrf().disable();
     //http.csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
   }
}
```

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
            method="post">
<input type="submit"
            value="Log out" />
<input type="hidden"
            name="${_csrf.parameterName}"
            value="${_csrf.token}"/>
</form>
```

```
<html>
<head>
            <meta name="_csrf" content="${_csrf.token}"/>
            <!-- default header name is X-CSRF-TOKEN -->
            <meta name="_csrf_header"
content="${_csrf.headerName}"/>
            <!-- ... -->
</head>
```

```
$(function () {
var token = $("meta[name='_csrf']").attr("content");
var header =
$("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(e, xhr, options) {
            xhr.setRequestHeader(header, token);
});
});
```

5. Интеграция со spring-data

Нужно добавить бин **org.springframework.security.data.repository.query.SecurityEvaluationContextExtension**

```
//Authentication.getPrincipal().getId()
@Repository
public interface MessageRepository extends PagingAndSortingRepository<Message,Long> {
        @Query("select m from Message m where m.to.id = ?#{ principal?.id }")
        Page<Message> findInbox(Pageable pageable);
}
```

6. Тестирование

- @WithMockUser
- @WithAnonymousUser
- @WithUserDetails
- @WithSecurityContext