**Spring security –** фреймворк для обеспечения аутентификации и авторизации пользователей. Легко интегрируется с Servlet API и Web MVC, защищает от атак типа фиксация сессии, кликджекинг, межсайтовая подделка запроса и др.

Аутентификация – кто вы? Авторизация – что вам разрешено делать.

**Основные возможности:**

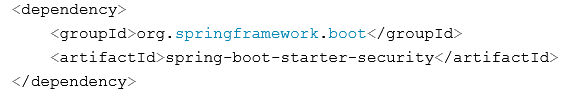
* **Шифрование паролей**
* **Аутентификация In-Memory** – для сохранения информации о пользователях и выполнения аутентификации можно использовать временную базу данных, которая остается в оперативной памяти приложения. (Полезно при разработке и тестировании)
* **LDAP**-**аутентификация** - Lightweight Directory Access Protocol (LDAP) — протокол аутентификации учетных записей пользователей в организациях. Позволяет определять структуру пользователей и групп пользователей, назначать им права доступа.
* **Управление сессией**
* **Remember Me Authentication –** встроенный механизм распознавания, благодаря которому пользователям не нужно вводить учетные данные при каждом посещении сайта.
* **OAuth 2.0 -** открытый стандарт проверки прав пользователя с помощью сервиса авторизации. Он также используется для реализации таких функций, как вход через учетные записи Facebook, Google и других крупных площадок.

В Web MVC Spring Security основана на фильтрах сервлетов. Есть цепочка фильтров, которая может предотвратить доступ к контроллеру.

**FilterChainProxy** – фильтр в цепочке, который отвечает за SpringSecurity. Он установлен по умолчанию, и применяется к каждому запросу. С точки зрения контейнера, SpringSecurity это один фильтр. Но внутри него есть дополнительные фильтры, которым он делегирует работу.

**Подключение Spring Security к проекту:**

Добавить в проект зависимость

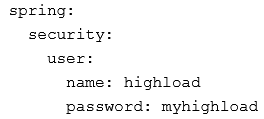


При использовании SpringBoot этого будет достаточно, чтобы добавить базовую аутентификацию. Spring security добавляет своего пользователя с именем user и генерирует ему пароль при запуске. Кроме того:

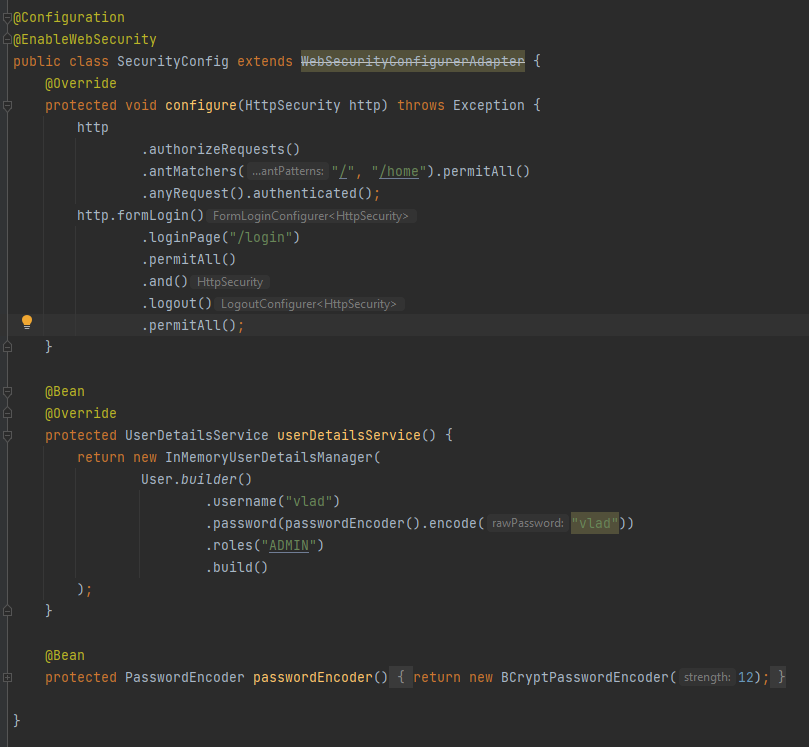
1. Формируется страница с формой для ввода имени и пароля.
2. Сразу начинает работать проверка имени и пароля.
3. Все URL недоступны, пока вы не зайдете в систему.

По умолчанию данные пользователей хранятся в оперативной памяти, пока приложение запущено (In-Memory authentication).

Параметры пользователя по умолчанию можно настроить в *application.yml*



Настроить аутентификацию можно в классе конфигурации. Нужно унаследоваться от класса **WebSecurityConfigurerAdapter** и использовать аннотацию **@EnableWebSecurity.**

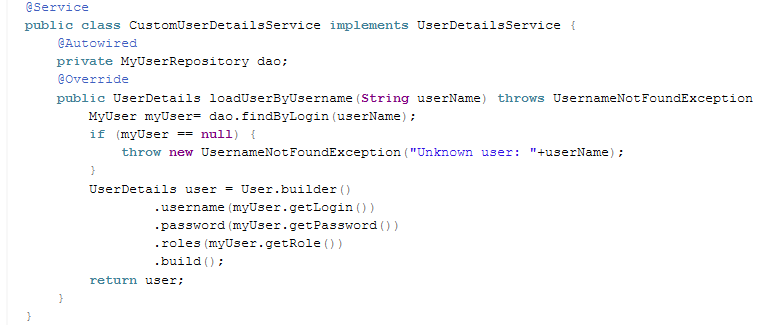


Метод **configure(HttpSecurity)** определяет какие URL пути должны быть защищены, а какие нет.

Используя **AuthenticationManagerBuilder** мы указываем тип аутентификации и добавляем юзеров.

**UserDetailsService –** объект, позволяющий получать пользователя (Объект UserDetails) по имени.

Например можно реализовать получение пользователя из БД.



**Методы интерфейса UserDetails:**

* Collection<? extends GrantedAuthority> **getAuthorities() –** возвращает полномочия, предоставленные пользователю.
* String **getPassword**() – возвращает пароль пользователя в виде хеша.
* String **getUsername**() – возвращает имя пользователя, используемое для аутентификации пользователя.
* boolean **isAccountNonExpired**() – указывает, истек ли срок действия учетной записи пользователя.
* boolean **isAccountNonLocked**() – указывает, заблокирован ли пользователь
* boolean **isCredentialsNonExpired**() – указывает, истек ли срок действия учетных данных пользователя (пароля)
* boolean **isEnabled**() – указывает доступен ли пользователь. Отключенный пользователь не может быть идентифицирован.

**Типы встроенных UserDetailsService менеджеров**:

* **InMemoryUserDetailsManager –** пользователи хранятся в оперативной памяти.
* **JdbcUserDetailsManager** – пользователи хранятся в БД. Вы можете настроить его в соответствии с вашей пользовательской структурой таблицы.
* **LDAP –** Пользователи берутся из LDAP. Lightweight Directory Access Protocol, является открытым протоколом, используемым для хранения и получения данных из каталога с иерархической структурой.
* **Кастомная**

В **AuthenticationManager** объявлен один метод **authenticate(Authentication authentication).** Этот метод может вернуть объект Authentication если удалось идентифицировать пользователя, либо выбросить исключение **AuthenticationException,** если входные данные представляют недействительного принципала. Возвращает null если не может принять решение.

**Authentication –** объект, который предоставляет *токен* *для запроса аутентификации* или *принципала* который прошел аутентификацию. Также содержит *список полномочий*, к которым получил доступ принципал.

Как происходит Аутентификация:

В метод **authenticate** **фильтра BasicAuthenticationFilter** передается токен с именем и паролем из http заголовка.

1. Фильтр **BasicAuthenticationFilter** извлекает имя и пароль пользователя из http заголовка Basic Auth.
2. **AuthenticationManager** в методе **authenticate()** в зависимости от своего типа извлекает с помощью UserDetailsService указанного пользователя (Из БД, из Оперативной памяти и т.п.) в виде объекта UserDetails
3. Пароль из http заголовка хешируется и сравнивается с тем что в объекте UserDetails. Если аутентификация прошла, возвращается новый объект Authentication.

Объект Authentication сохраняется в SecurityContext, а тот в свою очередь в SecurityContextHolder. Получить текущего пользователя можно следующим образом:



Если включены сессии, то после аутентификации клиенту отправляется уникальный **JSESSIONID.** Клиент отправляет его на сервер во всех последующих запросах. По JSeSSIONID восстанавливается сессия, из нее берется SecurityContext, а из него Authentication.

**PasswordEncoder –** объект определяющий функцию хеширования пароля. Как правило создается бином в конфигурации.

**AuthenticationProvider**

Если мы используем сторонний сервис для аутентификации, то мы не храним пароли и имена пользователей у себя в приложении. В таком случае мы не можем использовать UserDetailService. Вместо этого нам нужно реализовать и предоставить бин **AuthenticationProvider**.

@Bean

public AuthenticationProvider authenticationProvider() {

return new AtlassianCrowdAuthenticationProvider();

}

Главным методом объекта AuthenticationProvider является **authenticate**, который возвращает объект аутентификации. Наивная реализация может выглядеть так:

public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

Authentication authenticate(Authentication authentication) // **(1)**

throws AuthenticationException {

String username = authentication.getPrincipal().toString(); // **(1)**

String password = authentication.getCredentials().toString(); // **(1)**

User user = callAtlassianCrowdRestService(username, password); // **(2)**

if (user == null) { // **(3)**

throw new AuthenticationException(*"could not login"*);

}

return new UserNamePasswordAuthenticationToken(user.getUsername(), user.getPassword(), user.getAuthorities()); // **(4)**

}

*// other method ignored*

}

1. По сравнению с методом load() UserDetails, где у вас был доступ только к имени пользователя, теперь у вас есть доступ к полной попытке аутентификации, обычно содержащей имя пользователя и пароль.
2. Мы можем делать все что захотим, для аутентификации пользователя, например, вызывать REST-сервис.
3. Если аутентификация не прошла, нужно выбросить исключение **AuthenticationException.**
4. Если аутентификация прошла успешно, необходимо вернуть полностью инициализированный **UsernamePasswordAuthenticationToken.** Это реализация интерфейса Authentication, и для него необходимо, чтобы поле authentication было установлено true(которое конструктор, использованный выше, установит автоматически).

**ROLE**

Класс **роли** должен реализовывать интерфейс **GrantedAuthority.** Этот интерфейс имеет единственный метод **getAuthority(),** который возвращает имя роли. Имя роли должно соответствовать шаблону “ROLE\_ИМЯ”.

В Spring имеется простейшая реализация **SimpleGrantedAuthority.**

public final class SimpleGrantedAuthority implements GrantedAuthority {

private final String role;

@Override

public String getAuthority() {

return role;

}

}

Как правило, эти полномочия будут хранится в вашей бд. В простейшем случае это может быть просто дополнительное строковое поле. Можно также иметь отдельно таблицу AUTHORITIES.

Затем вы можете ограничить доступ к ресурсам, разрешив его только пользователям с определенной ролью.

@Configuration

@EnableWebSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers(*"/admin"*).hasAuthority(*"ROLE\_ADMIN"*) // **(1)**

.antMatchers(*"/callcenter"*).hasAnyAuthority(*"ROLE\_ADMIN"*, *"ROLE\_CALLCENTER"*) // **(2)**

.anyRequest().authenticated() // **(3)**

.and()

.formLogin()

.and()

.httpBasic();

}

}

Самый мощный способ настройки авторизации – это метод **access.** Он позволяет указать практически любые допустимые выражения SpEL.

**Хеширование**

Для более безопасного пароля его надо хешировать. Хеш функция преобразует входную строку в зашифрованную, по некому алгоритму. Главным достоинством хеша является то, что даже зная хеш, нельзя узнать исходную строку. Когда нужно аутентифицировать пользователя, его пароль преобразуется в хеш, и сравнивается с хешом, который хранится в приложении.

Чтобы Spring Security начал использовать хеширование, нужно сконфигурировать бин **PasswordEncoder**, который отвечает за хеширование. Spring предоставляет множество реализаций различных алгоритмов хеширования.

На случай, если у вас используется **несколько алгоритмов хеширования** (например вы сменили алгоритм хеширования, но нужно как-то поддерживать аккаунты старых пользователей) можно использовать **DelegatingPasswordEncoder.**

@Bean

public PasswordEncoder passwordEncoder() {

return PasswordEncoderFactories.createDelegatingPasswordEncoder();

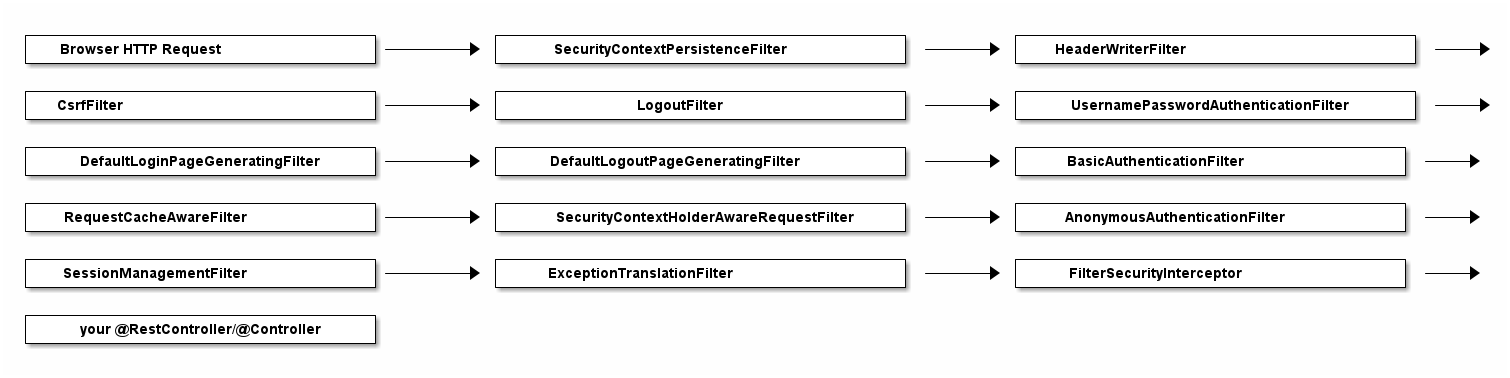
}

Чтобы encoder работал правильно, хеш пароля должен начинаться с префикса **{type}**

|  |  |  |
| --- | --- | --- |
| Таблица 1. Таблица пользователей | | |
| имя пользователя | пароль |  |
| [john@doe.com](mailto:john@doe.com) | {bcrypt}$2y$12$6t86Rpr3llMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/zA0C |  |
| [мой@user.com](mailto:my@user.com) | {sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5f6 |  |

**Цепочка фильтров FilterChainProxy**

Цепочка фильтров Spring Security состоит из 15 фильтров.



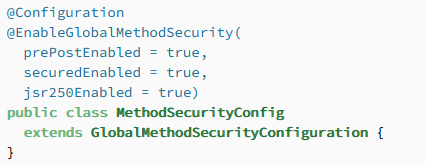
* **BasicAuthenticationFilter –** пытается найти HTTP заголовок Basic Auth в запросе, и если он найден, пытается аутентифицировать пользователя с помощью имени пользователя и пароля из заголовка.
* **UsernamePasswordAuthenticationFilter –** пытается найти имя и пароль в теле POST запроса, и аутентифицировать пользователя с помощью найденных значений.
* **DefaultLoginPageGeneratingFilter –** создает для вас страницу входа, если вы явно не отключили эту функцию. Именно поэтому мы получаем страницу входа по умолчанию.
* **DefaultLogoutPageGeneratingFilter –** создает для вас страницу выхода, если вы явно не отключили эту функцию.
* **FilterSecurityInterceptor –** выполняет авторизацию. Самый последний фильтр в цепочке. Проверяет, имеет ли аутентифицированный пользователь право доступа к запрашиваемому ресурсу.

**АВТОРИЗАЦИЯ**

**FilterSecurityInterceptor** выполняет авторизацию не сам, а делегирует работу объекту **AccessDecisionManager.** Этот интерфейс решает, имеет ли данный пользователь разрешение на доступ к данному ресурсу. Он получает объект Authentication, который содержит информацию о роли пользователя;

**Method Security**

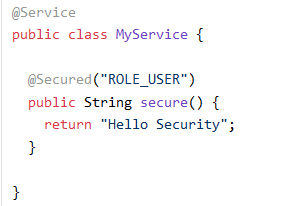
Можно применять правила доступа к выполнению методов Java. Чтобы включить ее используется аннотация **@EnableGlobalMethodSecurity(securedEnabled = true)**



Свойства

* **prepostEnabled** включает аннотации **@PreAuthorize** и **@PostAuthorize**.
* **securedEnabled** включчает аннотацию **@Secured.**
* **Jsr250Enabled** включает аннотацию **@RoleAllowed.**

Аннотация **@Secured(“ROLE\_USER”)** указывает, что данный метод может выполнить только пользователь с ролью USER. Если доступ к методу запрещен, бросается исключение **AccessDeniedException.** Данная аннотация не поддерживает SpEL.

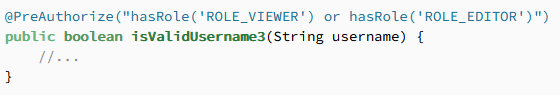


Аннотация **@RolesAllowed -** это аннотация JSR-250, эквивалентная аннотации @Secured.

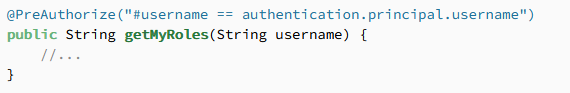
Аннотации @PreAuthorize и @PostAuthorize обеспечивают контроль доступа **на основе выражений**, предикаты могут быть написаны с использованием SpEL.

Аннотация **@PreAuthorize** проверяет заданное выражение **перед входом в метод**, тогда как аннотация **@PostAuthorize** проверяет его **после** выполнения метода и **может изменить результат**.

Например Аннотацию @RolesAllowed можно переписать с помощью @PreAuthorize следующим образом:



Более того, мы можем **использовать аргументы метода** внутри выражения. Например следующий метод будет вызван если параметр совпадает с именем текущего пользователя:



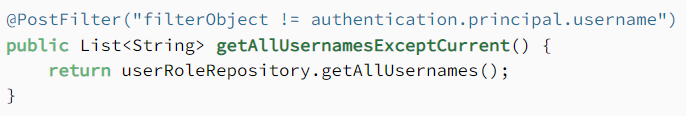
Аннотация @PostAutorize в свою очередь имеет доступ как к аргументам метода, так и к его результату.



**@PreFilter –** аннотация, которая позволяет отфильтровать аргумент-коллекцию перед выполнением метода.



**@PostFilter –** аннотация, позволяющая фильтровать возвращаемую методом коллекцию.



Если указать какую-либо из вышеперечисленных аннотаций над классом, то она будет применена ко всем методам данного класса. Так же эти аннотации можно комбинировать любым образом.

**Важно помнить** что для применения безопасности методов используется проксирование Spring AOP. Если защищенный метод A вызывается другим методом в том же классе, безопасность в А полностью игнорируется. Потому что в данном случае вызывается метод оригинального объекта, а не объекта прокси.

**Spring Security и Spring Web MVC**

1. В дополнение к antMatchers и regexMatchers вы также можете использовать **mvcMatchers.** mvcMatcher работает так же как и @RequestMapping и проверяет только точное совпадение урла. В то время как antMatchers допускает раасширения.

**antMatchers("/secured")** пропустит только **/secured** URL

**mvcMatchers("/secured")** пропустит **/secured** а так же /**secured/, /secured.html, /secured.xyz**

1. Позволяет внедрить в контроллер текущего аутентифицированного принципала.
2. Позволяет внедрить в контроллер текущий CSRFToken.

@Controller

public class MyController {

@RequestMapping(*"/messages/inbox"*)

public ModelAndView findMessagesForUser(@AuthenticationPrincipal CustomUser customUser, CsrfToken token) { // **(1)** **(2)**

*// .. find messages for this user and return them ...*

}

}

**Spring Security & Thymeleaf**

Существует специальный диалект **Spring Security Thymeleaf**, который позволяет вам помещать выражения безопасности непосредственно в ваши HTML-шаблоны Thymeleaf.

<div sec:authorize=*"isAuthenticated()"*>

This content is only shown to authenticated users.

</div>

<div sec:authorize=*"hasRole('ROLE\_ADMIN')"*>

This content is only shown to administrators.

</div>

<div sec:authorize=*"hasRole('ROLE\_USER')"*>

This content is only shown to users.

</div>

**CORS**

**Cross-Origin Resource Sharing –** или правило единого источника. По умолчанию браузер запрещает доступ к ресурсам, которые находятся в другом домене. Это помогает в борьбе с **фишинговыми ссылками**.

Допустим вы перешли по ссылке на мошеннический сайт, который с помощью фрейма загружает интерфейс сайта вашего банка и успешно залогинивает вас с помощью сохраненных куки.

Чтобы разрешить доступ к ресурсу в другом домене, клиент передает заголовок **Origin**, в котором указывается домен с которого поступил запрос. Чтобы браузер разрешил запрос, сервер должен прислать в ответе заголовок **Access-Control-Allow-Origin.** Его значение определяет, из каких источников можно получить доступ к ресурсам на сервере**.** Если в этом заголовке есть наш домен, доступ будет разрешен.

Заголовок **Access-Control-Allow-Methods** указывает какие методы http разрешены к этому ресурсу.

С **PUT**, **PATCH** и **DELETE** CORS работает с **по-другому**. В этих случаях браузер сначала отправляет предварительный запрос (Обычно Options). Сервер отправляет пустой ответ с CORS-заголовками, и браузер проверяет, разрешен ли HTTP-запрос.

**Предварительный запрос** — отличный способ **уберечь** нас **от** получения доступа или **изменения ресурсов на серверах**, **у которых** (пока что**) не настроены правила CORS**. Сервера защищены от потенциально нежелательных запросов из других источников.

Для инициализации cross-origin запроса браузер клиента добавляет в HTTP-запрос указание *Origin* (домена сайта, с которого происходит запрос). Например, страница http://www.a.com/page.html пытается получить данные со страницы http://www.b.com/cors.txt. В случае если браузер клиента поддерживает технологию CORS, запрос будет выглядеть так:

GET /cors.txt HTTP/1.1

Host: www.b.com

Origin: www.a.com

Если сервер www.b.com разрешает получение данных с www.a.com, то в ответе сервера будет присутствовать строка:

Access-Control-Allow-Origin: http://www.a.com

При включенных CORS spring будет добавлять этот заголовок в ответ.

Для этого нужно реализовать метод **addCorsMappings()** интерфейса **WebMvcConfigurer.**

****

**Remember me**

Долгие сессии могут нагрузить сервер, т.к. все объекты пользовательских сессий хранятся в куче контейнера. Поэтому сессии имеет смысл сделать короче, а идентичность запоминать с помощью специального долгосрочного **Hash-Based** токена.

Токен содержит только **имя пользователя и хэш**, с помощью которого можно проверить подлинность токена. При этом истекшие сессии не восстанавливаются, а начинаются заново. Зато пользователь может не совершать вход заново, его помнят благодаря Remember-me токену.

Remember-me токен позволяет помнить пользователя даже после перезапуска сервера.

Токен высылается клиенту в Set-Cookie аналогично сессии:



Но восстановить из него можно только имя пользователя, никакие другие данные по нему не восстанавливаются – **хранить в нем объекты нельзя** (а в сессии можно).

При каждом запросе выполняется **проверка подлинности токена**. Remember-me аутентификация будет работать и если несколько серверов, т.к. она не завязана на конкретный Tomcat-контейнер (и хранящуюся в его памяти сессию). Опознание пользователя происходит не путем обращения в Map сессий, а путем проверки подлинности токена.

Spring Security имеет 2 реализации remember-me:

1. **Simple Hash-Based Token** - Использует хеширование для сохранения безопасности токена на основе cookie
2. **Persistence Token** - Использует базу данных или другое постоянное хранилище для хранения сгенерированных токенов.

При **Simple Hash-Based** токен содержит:

* Имя пользователя и срок годности токена в открытом виде (Base64)
* Некий хеш (md5Hex) – значение, вычисляемое на основе имени, пароля, срока годности токена и секретного ключа.

Из хэша пароль обратно не восстановить, но на бэкенде можно по доступному из токена имени найти пароль и вычислить хеш заново.

Это так же улучшает безопасность, ведь мы реже передаем пароль в открытом виде.

**Включить remember me** и настроить время жизни токена:



Чтобы задать секретный ключ используем **key.(“secretkey”)**.



Для **Persistence Token** нам нужна таблица, в которой мы будем хранить данные о входе.

**JWT-токен**

**JWT-токен (JSON Web Token)** похож на Simple Hash-Based Remember-Me токен.

1. JWT-токен выдается клиенту после успешного ввода имени и пароля.
2. С последующими запросами клиент отправляет его серверу в заголовке, сервер проверяет его подлинность. Удостоверяется что клиент есть тот, за кого себя выдает.
3. JWT-токен продолжает работать и после того, как сервер перезапущен
4. JWT-токен работает, если серверов несколько (с балансировщиком нагрузки). На конкретный сервер ничего не завязано: сессий нет, данных хранящихся в сессии нет.

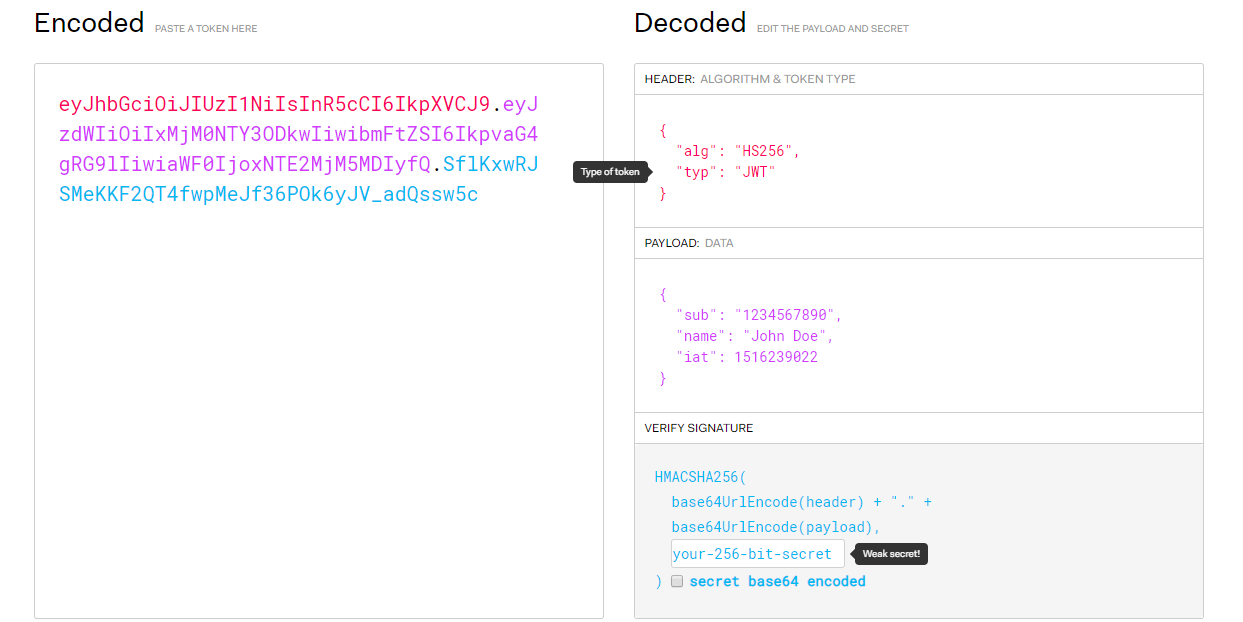
В отличии от Simple Hash-Based:

* В JWT-токене можно хранить больше данных. В нем есть специальная часть **Payload –** полезная нагрузка, туда можно записать роли, например. Обычно достаточно имени пользователя. Ничего секретного в Payload записывать нельзя.
* Для вычисления подписи в JWT-токене не используется пароль пользователя.

JWT **состоит из трех частей,** разделенных точкой:

* Заголовок
* Полезная нагрузка
* Подпись (в виде хеша)



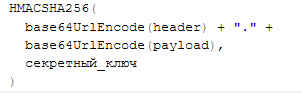


**Как проверяется подлинность токена.**

Берется **заголовок**, **полезная нагрузка** и **секретный ключ**, из них вычисляется некоторое значение – подпись.

Секретный ключ хранится на стороне сервера. С помощью него подпись вычисляется при первоначальной выдаче токена, и с помощью него же перевычисляется каждый раз, когда приходи токен (иначе говоря, токен проверяется на валидность).

Этот принцип проверки, аналогичен тому что используется в Simple Hash-Based Remember-Me токене. Только для JWT не надо обращаться к базе и находить пароль пользователя. Формула такая:

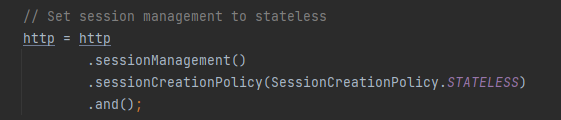
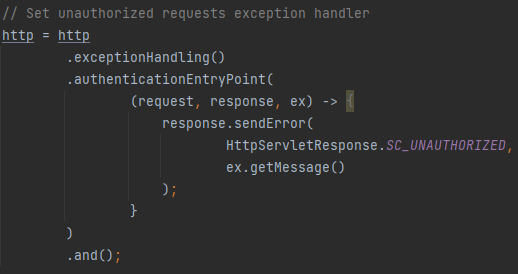
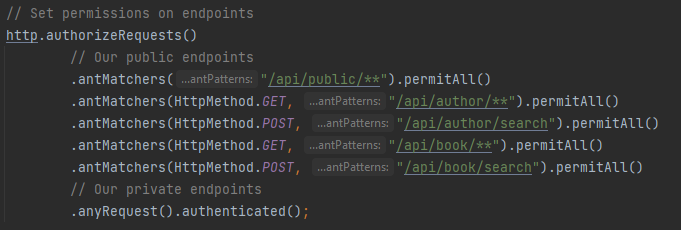
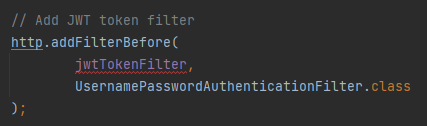


**JWT-токен не хранится на сервере**! Он каждый раз проверяется на подлинность с помощью вышепреведенной формулы. Хранится только секретный ключ – он един для всех JWT-токенов, выпускаемых приложением.

JWT-токен обычно передается в следующем заголовке:



Для настройки JWT токена нужно:

1. Включить CORS и отключить CSRF  
   
2. Установить управление сессиями на stateless.  
   
3. Установить обработчик исключений неавторизованных запросов.  
   
4. Установить разрешения на конечные точки.  
   
5. Добавить фильтр JWT-токенов. Обратите внимание, что мы добавляем JwtTokenFilter перед внутренним фильтром аутентификации UsernamePasswordAuthenticationFilter в Spring Security.  
   

Реализацию фильтра пишем сами, наследуясь от класса **OncePerRequestFilter** и переопределяя метод **doFilterInternal.**

В фильтре достаем токен из заголовка Authorization, извлекаем из него заголовок, полезную нагрузку. Хэшируем с помощью секретного ключа заголовок, полезную нагрузку и проверяем на равенство с подписью. Обычно в этом фильтре мы должны установить аутентификацию через SecurityContextHolder.



**oauth, oauth2**

Open Authorization 2.0 — открытый стандарт проверки прав пользователя с помощью сервиса авторизации. Он также используется для реализации таких функций, как вход через учетные записи Facebook, Google и других крупных площадок.