GeekBrains

Основы веб-разработки на Spring Framework

Защита приложения

Spring Security. Авторизация. Защита на уровне запросов, представлений, методов.

Оглавление

Начало работы с Spring Security

Spring Security — это мощный и гибкий фреймворк, предоставляющий механизмы защиты веб-приложения. Предлагает несколько уровней защиты:

- защита на уровне запросов — ограничение доступа к ресурсам, имеющим определенный URL;
- защита на уровне представлений — отображение элементов представления в зависимости от привилегий пользователя;
- защита на уровне методов — ограничение вызова методов сервисов/контроллеров в зависимости от привилегий пользователя.

Для подключения Spring Security к проекту, в pom.xml необходимо добавить следующие зависимости:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
    <version>3.0.4.RELEASE</version>
</dependency>
```

Первая зависимость представляет собой сам модуль Spring Security, а вторая — библиотеку для его интеграции с шаблонизатором Thymeleaf. После того, как модуль Spring Security был подключен, Spring Boot автоматически выполнит базовую настройку правила безопасности. Как правило, такой настройки недостаточно, поэтому рассмотрим “ручное” конфигурирование этих правил.

Для начала добавим в проект конфигурационный класс SecurityConfig, наследуемый от WebSecurityConfigurerAdapter.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   // ...
}
```

Аннотация @Configuration говорит о том, что данный класс является конфигурационным. @EnableWebSecurity отключает стандартные настройки безопасности Spring Security и начинает использовать правила, прописанные в SecurityConfig. @EnableGlobalMethodSecurity активирует возможность ставить защиту на уровне методов (для этого над методами ставятся аннотации @Secured и @PreAuthorized).

Вот так может выглядеть полноценная настройка безопасности приложения.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    private DataSource dataSource;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.jdbcAuthentication().dataSource(dataSource);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/").hasAnyRole("USER")
                .antMatchers("/admin/**").hasRole("ADMIN")
                  .and()
                .formLogin()
                  .loginPage("/login")
                  .loginProcessingUrl("/authenticateTheUser")
                .permitAll();
    }
}
```

За настройку способа аутентификации отвечает метод configure(AuthenticationManagerBuilder auth). Существует несколько стандартных вариантов настройки этого метода:

1. Spring Security при проверке логина/пароля будет искать в стандартной паре таблиц: users и authorities. Все что нужно для такого способа, это заинжектить DataSource, создать таблицы в базе, и добавить туда пользователей. Настройка самого метода будет выглядеть вот так:

   ```
   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws
   Exception {
       auth.jdbcAuthentication().dataSource(dataSource);
   }
   ```

   Бин DataSource настраивается через Spring Boot application.properties:

   ```
   spring.datasource.driver-class-name=org.postgresql.Driver
   spring.datasource.url=jdbc:postgresql://localhost:5432/postgres?currentSch
   ema=spring_app
   spring.datasource.username=postgres
   spring.datasource.password=iejfJ1Kw
   ```

2. В некоторых случаях нет необходимости в создании новых пользователей и хранении их в базе данных. Например, мы можем хранить просто в памяти стандартный набор пользователей и работать с ними (при этом отпадает необходимость в бине DataSource). В этом случае подойдет вариант с inMemoryAuthentication():

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
  User.UserBuilder users = User.withDefaultPasswordEncoder();
  auth.inMemoryAuthentication()
   .withUser(users.username("user1").password("pass1").roles("USER",
"ADMIN"))
   .withUser(users.username("user2").password("pass2").roles("USER"));
}
```

User.UserBuilder users служит для создания пользователей (их логина, пароля и ролей). Метод auth.inMemoryAuthentication() создает пользователей и указывает необходимость брать информацию о них из памяти.

3. Если же мы хотим хранить пользователей и их роли в базе данных, и у нас есть собственные таблицы для этого, то можно воспользоваться бином DaoAuthenticationProvider (при таком подходе отпадает необходимость в методе configure(AuthenticationBuilder)). Бин UserService implements UserDetailsService описывает как должна производиться загрузка пользовательских данных из базы, чуть ниже будет приведен пример с реализацией. BCryptPasswordEncoder указывает что пароли должны храниться не в “чистом” виде и хэшироваться с помощью BCrypt.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   private UserService userService;

   @Autowired
   public void setUserService(UserService userService) {
       this.userService = userService;
   }

   @Bean
   public BCryptPasswordEncoder passwordEncoder() {
       return new BCryptPasswordEncoder();
   }

   @Bean
   public DaoAuthenticationProvider authenticationProvider() {
       DaoAuthenticationProvider auth = new DaoAuthenticationProvider();
       auth.setUserDetailsService(userService);
       auth.setPasswordEncoder(passwordEncoder());
       return auth;
   }
}
```

Давайте посмотрим на пример реализации UserService. В данном примере User и Role это

самые обычные сущности. Для того, чтобы достать их из базы используем Spring Data репозитории, поэтому Security Config не требует явного указания бина DataSource.

```
@Service
public class UserService implements UserDetailsService {
   private UserRepository userRepository;
   private RoleRepository roleRepository;

   @Autowired
   public void setUserRepository(UserRepository userRepository) {
       this.userRepository = userRepository;
   }

   @Autowired
   public void setRoleRepository(RoleRepository roleRepository) {
       this.roleRepository = roleRepository;
   }

   public User findByUsername(String username) {
       return userRepository.findOneByUsername(username);
   }

   @Override
   public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
       User user = userRepository.findOneByUsername(username);
       if (user == null) {
           throw new UsernameNotFoundException("Invalid username or
password");
       }
       return new
org.springframework.security.core.userdetails.User(user.getPhone(),
user.getPassword(), mapRolesToAuthorities(user.getRoles()));
   }

   private Collection<? extends GrantedAuthority>
mapRolesToAuthorities(Collection<Role> roles) {
       return roles.stream().map(role -> new
SimpleGrantedAuthority(role.getName())).collect(Collectors.toList());
   }
}
```

С правилами аутентификации разобрались. Переходим непосредственно к защите приложения. За настройку безопасности отвечает метод configure(HttpSecurity http). В нем можно выполнить настройку процесса логина и логаута, правил доступа к определенным частям веб-приложения, и т.д. Пример настройки представлен ниже.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
```

```
                .antMatchers("/").hasAnyRole("USER")
                .antMatchers("/admin/**").hasRole("ADMIN")
                  .and()
                .formLogin()
                  .loginPage("/login")
                  .loginProcessingUrl("/authenticateTheUser")
                .permitAll();
    }
}
```

- Метод **configure(HttpSecurity http)** отвечает за настройку защиты на уровне запросов и конфигурирование процессов авторизации.
- **antMatchers** — с помощью данного метода указывается http-метод и URL (или шаблон URL), доступ к которому необходимо ограничить.
- **hasRole(String role)**, **hasAnyRole(String... roles)** — в нем указывается одна роль или набор ролей, необходимых пользователю для доступа к данному ресурсу.
- **formLogin()** — дает возможность настроить форму для авторизации.
- **loginPage** — URL формы авторизации.
- **loginProcessingUrl** — URL, на который будут отправляться данные формы (методом POST).
- * **logout()** — позволяет настроить правила выхода из учетной записи.
- * **failureUrl** — адрес для перенаправления пользователя в случае неудачной авторизации.
- * **logoutSuccessUrl** — URL, на который будет перенаправлен пользователь при выходе из аккаунта автора.
- * **usernameParameter** и **passwordParameter** — имена полей формы, содержащие логин и пароль, если не используются стандартные имена username и password;

(*) — параметры, не использующиеся в примере.

В примере выше **configure(HttpSecurity http)** указывает, что для доступа к сайту пользователь должен быть обязательно авторизован и иметь роль USER, иначе он будет перенаправлен на страницу авторизации. Для доступа к запросам, начинающимся с **/admin/**, необходимо иметь право доступа ADMIN. Для авторизации используется форма, для доступа к которой необходимо обратиться по адресу **/login**. Результаты заполнения этой формы в виде POST-запроса будут отправлены на **URL /authentificateTheUser**.

Подготовка базы данных

Рассмотрим работу через jdbcAuthentication. Чтобы хранить информацию о пользователях и работать с ней, применим MySQL базу данных. По умолчанию Spring Security будет использовать стандартный шаблон таблиц в БД: пользователи хранятся в таблице **users**, а роли — в **authorities**.

В таблице **users** три столбца:

- **username** — имя пользователя;

- **password** — пароль, который может храниться как в открытом, так и в хешированном виде;
- **enabled** — возможность пользователя войти под данной учетной записью.

Запрос на создание такой таблицы и добавление тестовых пользователей:

```
CREATE TABLE users (
    username varchar(50) NOT NULL,
    password varchar(100) NOT NULL,
    enabled tinyint(1) NOT NULL,

    PRIMARY KEY (username)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

INSERT INTO users
VALUES
('user1', '{noop}123', 1),
('user2', '{noop}123', 1);
```

Таблица **authorities** включает имя пользователя и соответствующую ему роль — каждому можно добавить по несколько. Запрос на создание этой таблицы и добавление тестовых данных:

```
CREATE TABLE authorities (
    username varchar(50) NOT NULL,
    authority varchar(50) NOT NULL,

    UNIQUE KEY authorities_idx_1 (username, authority),

    CONSTRAINT authorities_ibfk_1
    FOREIGN KEY (username)
    REFERENCES users (username)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

INSERT INTO authorities
VALUES
('user1', 'ROLE_ADMIN'),
('user1', 'ROLE_USER'),
('user2', 'ROLE_USER');
```

Выполнив эти два скрипта, мы подготовили БД для использования в качестве источника данных — при авторизации и указании прав при использовании веб-приложения.

Авторизация

Настройка базы данных, Spring Security и пользователей выполнена. Теперь реализуем возможность авторизации на сайте. Необходимы форма и метод обработки GET-запроса для нее.

```
@GetMapping("/login")
public String showMyLoginPage() {
    return "modern-login";
}
```

Метод очень простой — всего лишь возвращает html-страницу с именем **modern-login.html**. Код страницы:

```
<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org" lang="en">

<head>
    <title>Login Page</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></scri
pt>
</head>

<body>

<div>
    <div id="loginbox" style="margin-top: 50px;" class="mainbox col-md-3
col-md-offset-2 col-sm-6 col-sm-offset-2">
        <div class="panel panel-info">
            <div class="panel-heading">
                <div class="panel-title">Sign In</div>
            </div>
            <div style="padding-top: 30px" class="panel-body">
                <form th:action="@{/authenticateTheUser}" method="POST"
class="form-horizontal">
                    <div class="form-group">
                        <div class="col-xs-15">
                            <div>
                                <div th:if="${param.error} != null">
                                    <div class="alert alert-danger
col-xs-offset-1 col-xs-10">
                                        Invalid username or password
                                    </div>
                                </div>
                                <div th:if="${param.logout} != null">
                                    <div class="alert alert-success
col-xs-offset-1 col-xs-10">
                                        You have been logged out.
                                    </div>
                                </div>
                            </div>
                        </div>
                    </div>
                    <div style="margin-bottom: 25px" class="input-group">
```

```
                        <span class="input-group-addon"><i class="glyphicon
glyphicon-user"></i></span>
                        <input type="text" name="username"
placeholder="username" class="form-control">
                    </div>
                    <div style="margin-bottom: 25px" class="input-group">
                        <span class="input-group-addon"><i class="glyphicon
glyphicon-lock"></i></span>
                        <input type="password" name="password"
placeholder="password" class="form-control">
                    </div>
                    <div style="margin-top: 10px" class="form-group">
                        <div class="col-sm-6 controls">
                            <button type="submit" class="btn
btn-success">Login</button>
                        </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
</body>
</html>
```

У полей для ввода логина и пароля стандартные имена: **username** и **password**. Форма посылает POST-запрос по адресу **@{/authentificateTheUser}**. Если мы попали на эту страницу после неудачной попытки авторизации или после выхода из учетной записи, на форме будут показаны соответствующие сообщения. После авторизации пользователь получает набор прав, указанный в БД, и может пользоваться веб-приложением.

Защита на уровне представлений

Рассмотрим, как изменять видимость элементов на странице в зависимости от прав пользователей. Для этого используется зависимость **thymeleaf-extras-springsecurity5** из **pom.xml**.
Пример использования таких дополнительных возможностей:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<html xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<!--...-->
<body>
<div class="container">
    <h1>Welcome page</h1>
    <div sec:authorize="isAuthenticated()">
        Authenticated username:
        <div sec:authentication="principal.username"></div>
        Authenticated user roles:
        <div sec:authentication="principal.authorities"></div>
    </div>
```

```
    <!--<div sec:authorize="hasAnyRole('ADMIN', 'USER')">-->
    <div sec:authorize="hasRole('ADMIN')">
        This content will only be visible to ADMIN users.
    </div>

    <h2>Index:</h2>
<!--...-->
</div>
</body>
</html>
```

Здесь ** ** отвечает за проверку авторизации пользователя. Если он авторизован, на странице отображаются его **username** и права доступа (например, **ROLE_ADMIN** или **ROLE_USER**). Методы **hasRole()** и **hasAnyRole()** проверяют у пользователя наличие определенной роли.

Защита на уровне методов

Последний шаг для создания надежной защиты приложения — ограничить доступ на уровне методов. Для этого можно использовать аннотацию **@Secured**, которая ограничивает доступ к отдельным методам на основе информации о правах текущего пользователя. Пример кода:

```
@Secured({"ROLE_ADMIN"})
@RequestMapping("/onlyYou")
@ResponseBody
public String pageOnlyForAdmins() {
    return "index";
}
```

Доступ к методу `pageOnlyForAdmins()` имеют только пользователи с ролью ADMIN.

Практическое задание

1. Создать страницу со списком товаров, на которой можно добавлять позиции и редактировать существующие. На эту страницу должны иметь доступ админы и менеджеры.
2. Создать страницу со списком всех пользователей, к которой имеют доступ только админы.
3. * Добавить роль суперадмина и дать ему возможность создавать новых пользователей и указывать роли существующим.

Дополнительные материалы

1. Вебинар “Как устроен Spring Security”: https://geekbrains.ru/events/2176
2. Обзор Spring Security.
3. Хеширование паролей в Spring Security.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание).
2. Крейг Уоллс. Spring в действии.