

## woodwhale's blog

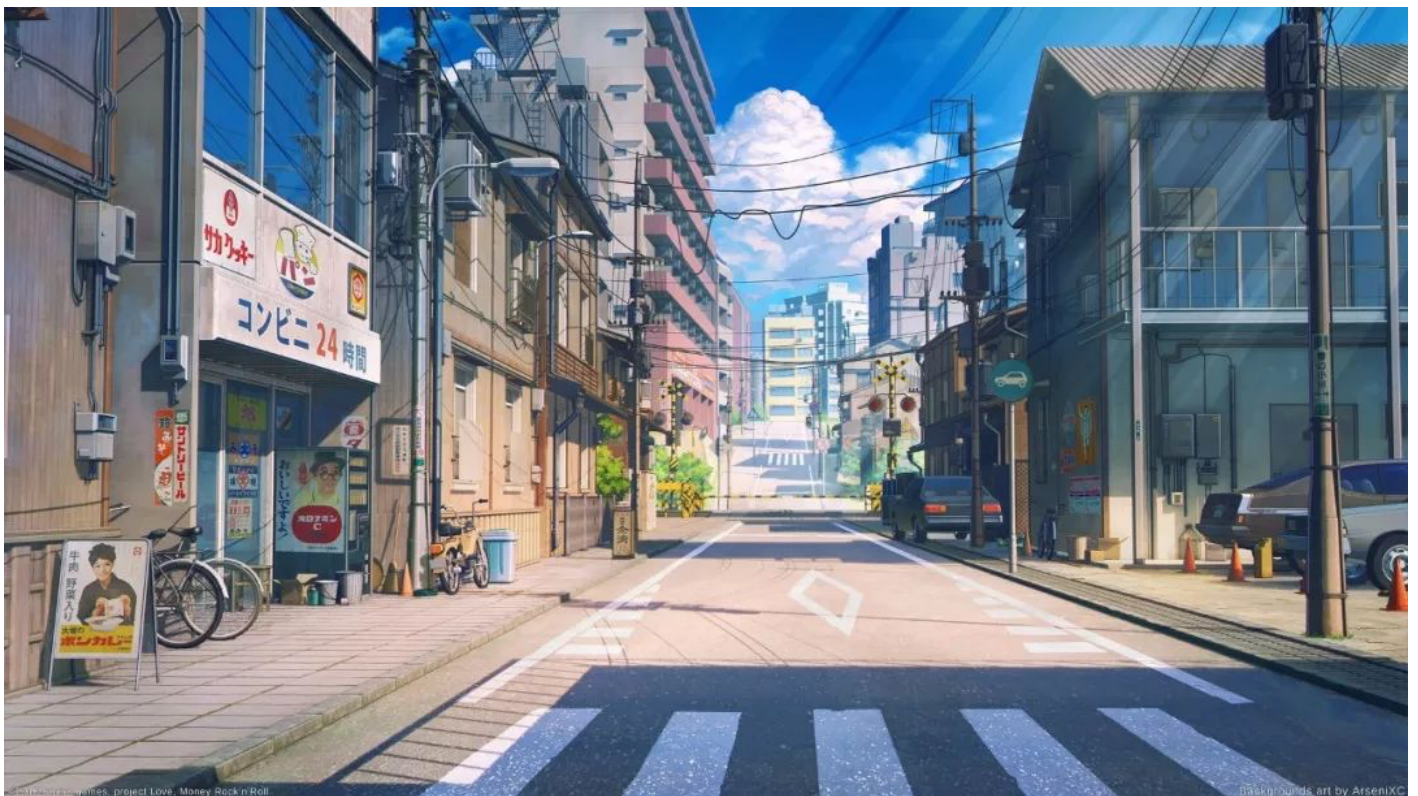
技术改变生活

首页 归档 分类 标签 工具 福利 关于 搜索

# SpringBoot + Spring Security 学习笔记（一）自定义基本使用及个性化登录配置

📅 发表于 2019-04-12 | 📁 分类 [Spring Security](#) | 💬 3 | 👁 阅读: 1480 | 📄 全文总计: 5.2k 字

自定义基本使用及个性化登录配置



官方文档参考，[5.1.2 中文参考文档](#)，[4.1 中文参考文档](#)，[4.1 官方文档中文翻译与源码解读](#)

SpringSecurity 核心功能：

- 认证（你是谁）
- 授权（你能干什么）

- 攻击防护 (防止伪造身份)

## 简单的开始

pom 依赖

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4      <modelVersion>4.0.0</modelVersion>
5      <parent>
6          <groupId>org.springframework.boot</groupId>
7          <artifactId>spring-boot-starter-parent</artifactId>
8          <version>2.1.4.RELEASE</version>
9          <relativePath/> <!-- lookup parent from repository -->
10     </parent>
11     <groupId>org.woodwhale.king</groupId>
12     <artifactId>security-demo</artifactId>
13     <version>1.0.0</version>
14     <name>security-demo</name>
15     <description>spring-security-demo project for Spring Boot</description>
16
17     <properties>
18         <java.version>1.8</java.version>
19     </properties>
20
21     <dependencies>
22         <dependency>
23             <groupId>org.springframework.boot</groupId>
24             <artifactId>spring-boot-starter-security</artifactId>
25         </dependency>
26         <dependency>
27             <groupId>org.springframework.boot</groupId>
28             <artifactId>spring-boot-starter-thymeleaf</artifactId>
29         </dependency>
30         <dependency>
31             <groupId>org.springframework.boot</groupId>
32             <artifactId>spring-boot-starter-web</artifactId>
33         </dependency>
34
35         <dependency>
36             <groupId>org.springframework.boot</groupId>
37             <artifactId>spring-boot-devtools</artifactId>
38             <scope>runtime</scope>
39         </dependency>
40         <dependency>
41             <groupId>mysql</groupId>
42             <artifactId>mysql-connector-java</artifactId>
43             <scope>runtime</scope>
44         </dependency>
45         <dependency>
```

```
46         <groupId>org.projectlombok</groupId>
47         <artifactId>lombok</artifactId>
48         <optional>true</optional>
49     </dependency>
50     <dependency>
51         <groupId>org.springframework.boot</groupId>
52         <artifactId>spring-boot-starter-test</artifactId>
53         <scope>test</scope>
54     </dependency>
55     <dependency>
56         <groupId>org.springframework.security</groupId>
57         <artifactId>spring-security-test</artifactId>
58         <scope>test</scope>
59     </dependency>
60 </dependencies>
61
62 <build>
63     <plugins>
64         <plugin>
65             <groupId>org.springframework.boot</groupId>
66             <artifactId>spring-boot-maven-plugin</artifactId>
67         </plugin>
68     </plugins>
69 </build>
70
71 </project>
```

编写一个最简单的用户 controller

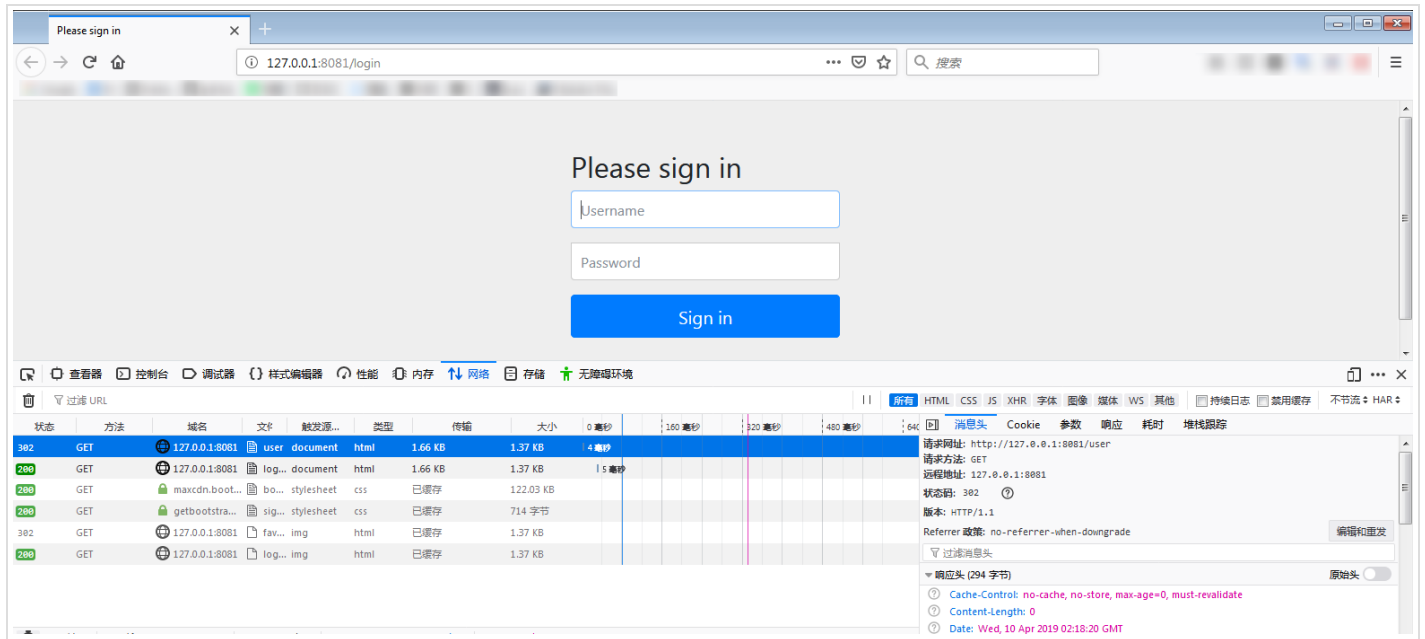
```
1  import org.springframework.web.bind.annotation.GetMapping;
2  import org.springframework.web.bind.annotation.RequestMapping;
3  import org.springframework.web.bind.annotation.RestController;
4
5  @RestController
6  @RequestMapping("/user")
7  public class UserController {
8      @GetMapping
9      public String getUsers() {
10         return "Hello Spring Security";
11     }
12 }
```

application.yml 配置IP 和端口

```
1  server:
2      address: 127.0.0.1
3      port: 8081
4
5  logging:
6      level:
```

7 [org.woodwhale.king:DEBUG](http://org.woodwhale.king:DEBUG)

浏览器访问<http://127.0.0.1:8081/user>, 浏览器被自动重定向到了登录的界面:



这个 `/login` 访问路径在程序中没有任何的显示代码编写, 为什么会出现这样的界面呢, 当前界面中的UI 都是哪里来的呢?

当然是 **spring-security** 进行了默认控制, 从启动日志中, 可以看到一串用户名默认为 `user` 的默认密码:

```
2019-04-10 10:16:31.612 INFO 4560 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing Executor
2019-04-10 10:16:31.612 WARN 4560 --- [ restartedMain] ion$DefaultTemplateResolverConfiguration : Cannot find template
2019-04-10 10:16:31.664 INFO 4560 --- [ restartedMain] .s.s.UserDetailsServiceAutoConfiguration :
Using generated security password: 947ba6f0-f7e7-487a-9764-3059951b32dd
2019-04-10 10:16:31.678 INFO 4560 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain
2019-04-10 10:16:31.726 INFO 4560 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is
2019-04-10 10:16:31.770 INFO 4560 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on por
```

登录成功之后, 可以正常访问服务资源了。

## 自定义默认用户名和密码

在配置文件配置用户名和密码:

```
1 spring:
2   security:
3     user:
4       name: "admin"
5       password: "admin"
```

## 关闭默认的安全访问控制

旧版的 spring security 关闭默认安全访问控制，只需要在配置文件中关闭即可：

```
1 security.basic.enabled = false
```

新版本 Spring-Boot2.xx(Spring-security5.x) 的不再提供上述配置了：

方法1：将 security 包从项目依赖中去除。

方法2：将 `org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration` 不注入spring中：

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration;
5
6 @SpringBootApplication
7 @EnableAutoConfiguration(exclude = {SecurityAutoConfiguration.class})
8 public class SecurityDemoApplication {
9
10     public static void main(String[] args) {
11         SpringApplication.run(SecurityDemoApplication.class, args);
12     }
13
14 }
```

方法3：已实现一个配置类继承自 `WebSecurityConfigurerAdapter`，并重写 `configure(HttpSecurity http)` 方法：

```
1 import org.springframework.context.annotation.Bean;
2 import org.springframework.context.annotation.Configuration;
3 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
4 import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
5 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
6 import org.springframework.security.core.userdetails.UserDetailsService;
7
8 @Configuration
9 @EnableWebSecurity
10 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
11
12     @Override
13     protected void configure(HttpSecurity http) throws Exception {
14         http.authorizeRequests().antMatchers("/**").permitAll();
15     }
16
17     /**
18     * 配置一个userDetailsService Bean
19     * 不再生成默认security.user用户
```

```

20     */
21     @Bean
22     @Override
23     protected UserDetailsService userDetailsService() {
24         return super.userDetailsService();
25     }
26 }

```

**注意：** `WebSecurityConfigurerAdapter` 是一个适配器类，所以为了使自定义的配置类见名知义，所以写成了 `WebSecurityConfig`。同时增加了 `@EnableWebSecurity` 注解到了 spring security 中。

## 自定义用户认证

### 安全认证配置注意事项

spring security 的自定义用户认证配置的核心均在上述的 `WebSecurityConfigurerAdapter` 类中，用户想要个性化的用户认证逻辑，就需要自己写一个自定义的配置类，适配到 spring security 中：

**注意：** 如果配置了两个以上的自定义实现类，那么就会报 `WebSecurityConfigurers` 不唯一的错误：`java.lang.IllegalStateException: @Order on WebSecurityConfigurers must be unique.`

```

1  @Configuration
2  @EnableWebSecurity
3  public class BrowerSecurityConfig extends WebSecurityConfigurerAdapter {
4
5      @Override
6      protected void configure(HttpSecurity http) throws Exception {
7          http.formLogin()           // 定义当需要提交表单进行用户登录时候，转到的登录页面。
8              .and()
9              .authorizeRequests()   // 定义哪些URL需要被保护、哪些不需要被保护
10             .anyRequest()          // 任何请求, 登录后可以访问
11             .authenticated();
12     }
13 }

```

## 自定义用户名和密码

### 密码加密注意事项

将用户名密码设置到内存中，用户登录的时候会校验内存中配置的用户名和密码：

在旧版本的 spring security 中，在上述自定义的 `BrowerSecurityConfig` 中配置如下代码即可：

```

1  @Override
2  protected void configure(AuthenticationManagerBuilder auth) throws Exception {

```



```
3     auth.inMemoryAuthentication().withUser("admin").password("admin").roles("ADMIN");
4 }
```

但是在新版本中，启动运行都没有问题，一旦用户正确登录的时候，会报异常：

```
1 java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
```

因为在 Spring security 5.0 中新增了多种加密方式，也改变了密码的格式。[官方文档说明：Password Storage Format](#)

**Password Storage Format**

The general format for a password is:

```
{id}encodedPassword
```

Such that `id` is an identifier used to look up which `PasswordEncoder` should be used and `encodedPassword` is the original encoded password for the selected `PasswordEncoder`. The `id` must be at the beginning of the password, start with `{` and end with `}`. If the `id` cannot be found, the `id` will be null. For example, the following might be a list of passwords encoded using different `id`. All of the original passwords are "password".

```
{bcrypt}$2a$10$dXJ3SW6G7P501GmMkkmwe.20cQQubK3.HZWzG3YB1t1Ry.fqvM/BG ❶
{noop}password ❷
{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca763d8dc ❸
{scrypt}$e0801$8bWJaSu2IKSn9Z9kM+TPXf0c/9bdYSrN1oD9qfVThWEwdRTn07re7Ei+fUZRJ68k91TyuTeUp4of4g24hHnazw== $0A0ec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY
{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94ffff291cffffaf8410849f27605abcbcb0 ❹
```

上面这段话的意思是，现在新的 Spring Security 中对密码的存储格式是 "`{id}.....`"。前面的 `id` 是加密方式，`id` 可以是 `bcrypt`、`sha256` 等，后面紧接着是使用这种加密类型进行加密后的密码。

因此，程序接收到内存或者数据库查询到的密码时，首先查找被 `{}` 包括起来的 `id`，以确定后面的密码是被什么加密类型方式进行加密的，如果找不到就认为 `id` 是 `null`。这也就是为什么程序会报错：`There is no PasswordEncoder mapped for the id "null"`。官方文档举的例子中是各种加密方式针对同一密码加密后的存储形式，原始密码都是"password"。

## 密码加密

要想我们的项目还能够正常登陆，需要将前端传过来的密码进行某种方式加密，官方推荐的是使用 `bcrypt` 加密方式（不用用户使用相同原密码生成的密文是不同的），因此需要在 `configure` 方法里面指定一下：

```
1 @Override
2 protected void configure(AuthenticationManagerBuilder auth) throws Exception {
3     // auth.inMemoryAuthentication().withUser("admin").password("admin").roles("ADMIN");
4     auth.inMemoryAuthentication()
5         .passwordEncoder(new BCryptPasswordEncoder())
6         .withUser("admin")
7         .password(new BCryptPasswordEncoder().encode("admin"))
8         .roles("ADMIN");
9 }
```

当然还有一种方法，将 `passwordEncoder` 配置抽离出来：

```
1  @Bean
2  public BCryptPasswordEncoder passwordEncoder() {
3      return new BCryptPasswordEncoder();
4  }
```

## 自定义到内存

```
1  @Override
2  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
3      auth.inMemoryAuthentication()
4          .withUser("admin")
5          .password(new BCryptPasswordEncoder().encode("admin"))
6          .roles("ADMIN");
7  }
```

## 自定义到代码

这里还有一种更优雅的方法，实现 `org.springframework.security.core.userdetails.UserDetailsService` 接口，重载 `loadUserByUsername(String username)` 方法，当用户登录时，会调用 `UserDetailsService` 接口的 `loadUserByUsername()` 来校验用户的合法性（密码和权限）。

这种方法为之后结合数据库或者JWT动态校验打下技术可行性基础。

```
1  @Service
2  public class MyUserDetailsService implements UserDetailsService {
3
4      @Override
5      public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
6          Collection<GrantedAuthority> authorities = new ArrayList<>();
7          authorities.add(new SimpleGrantedAuthority("ADMIN"));
8          return new User("root", new BCryptPasswordEncoder().encode("root"), authorities);
9      }
10
11 }
```

当然，“自定义到内存”中的配置文件中的 `configure(AuthenticationManagerBuilder auth)` 配置就不需要再配置一遍了。

**注意：**对于返回的 `UserDetails` 实现类，可以使用框架自己的 `User`，也可以自己实现一个 `UserDetails` 实现类，其中密码和权限都应该从数据库中读取出来，而不是写死在代码里。

## 最佳实践

将加密类型抽离出来，实现 `UserDetailsService` 接口，将两者注入到 `AuthenticationManagerBuilder` 中：

```
1  @Configuration
```



```

2  @EnableWebSecurity
3  public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
4      @Autowired
5      private UserDetailsService userDetailsService;
6
7      @Bean
8      public BCryptPasswordEncoder passwordEncoder() {
9          return new BCryptPasswordEncoder();
10     }
11
12     @Override
13     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
14         auth.userDetailsService(userDetailsService)
15             .passwordEncoder(passwordEncoder());
16     }
17 }

```

**UserDetailsService** 接口实现类：

```

1  import java.util.ArrayList;
2  import java.util.Collection;
3
4  import org.springframework.security.core.GrantedAuthority;
5  import org.springframework.security.core.authority.SimpleGrantedAuthority;
6  import org.springframework.security.core.userdetails.User;
7  import org.springframework.security.core.userdetails.UserDetails;
8  import org.springframework.security.core.userdetails.UserDetailsService;
9  import org.springframework.security.core.userdetails.UsernameNotFoundException;
10 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
11 import org.springframework.stereotype.Service;
12
13 @Service
14 public class MyUserDetailsService implements UserDetailsService {
15
16     @Override
17     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
18         Collection<GrantedAuthority> authorities = new ArrayList<>();
19         authorities.add(new SimpleGrantedAuthority("ADMIN"));
20         return new User("root", new BCryptPasswordEncoder().encode("root"), authorities);
21     }
22
23 }

```

这里的 User 对象是框架提供的一个用户对象，注意包名是：**org.springframework.security.core.userdetails.User**，里面的属性中最核心的就是 **password**，**username** 和 **authorities**。

## 自定义安全认证配置

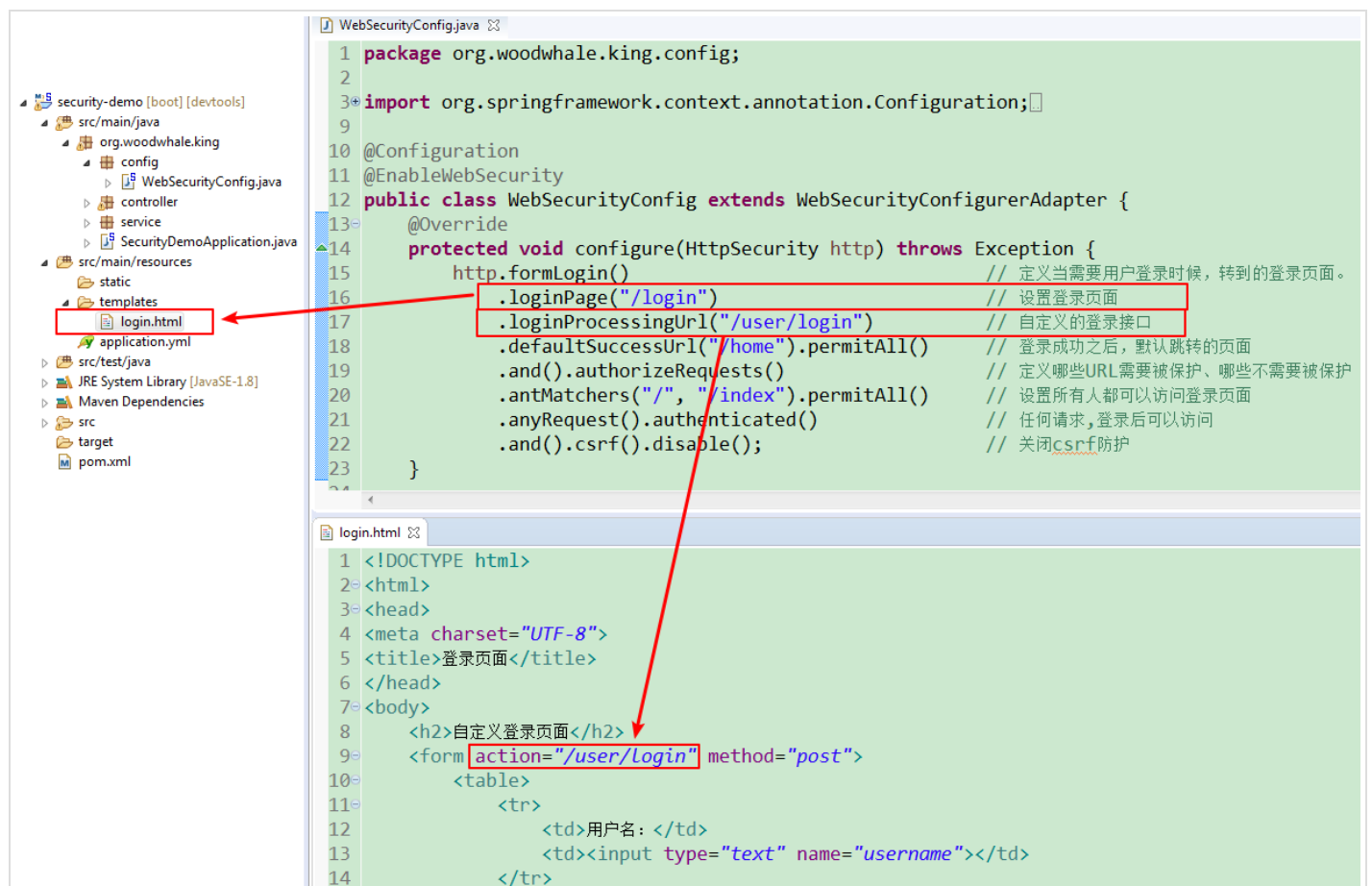
配置自定义的登录页面：

```

1  @Override
2  protected void configure(HttpSecurity http) throws Exception {
3      http.formLogin()                                     // 定义当需要用户登录时
4          .loginPage("/login")                             // 设置登录页面
5          .loginProcessingUrl("/user/login")               // 自定义的登录接口
6          .defaultSuccessUrl("/home").permitAll()         // 登录成功之后，默认跳转的页面
7          .and().authorizeRequests()                       // 定义哪些URL需要被保护、哪些不需
8          .antMatchers("/", "/index", "/user/login").permitAll() // 设置所有人都可以访问登录页面
9          .anyRequest().authenticated()                   // 任何请求，登录后可以访问
10         .and().csrf().disable();                         // 关闭csrf防护
11     }

```

从上述配置中，可以看出用可以所有访客均可以自由登录 / 和 /index 进行资源访问，同时配置了一个登录的接口 /login，使用 mvc 做了视图映射（映射到模板文件目录中的 login.html），controller 映射代码太简单就不赘述了，当用户成功登录之后，页面会自动跳转至 /home 页面。



上述图片中的配置有点小小缺陷，当去掉 `.loginProcessUrl()` 的配置的时候，登录完毕，浏览器会一直重定向，直至报重定向失败。因为登录成功的 url 没有配置成所有人都可以访问，因此造成了死循环的结果。

因此，配置了登录界面就需要配置任意可访问：`.antMatchers("/user/login").permitAll()`

login.html 代码:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>登录页面</title>
6  </head>
7  <body>
8      <h2>自定义登录页面</h2>
9      <form action="/user/login" method="post">
10         <table>
11             <tr>
12                 <td>用户名: </td>
13                 <td><input type="text" name="username"></td>
14             </tr>
15             <tr>
16                 <td>密码: </td>
17                 <td><input type="password" name="password"></td>
18             </tr>
19             <tr>
20                 <td colspan="2"><button type="submit">登录</button></td>
21             </tr>
22         </table>
23     </form>
24 </body>
25 </html>
```

## 静态资源忽略配置

上述配置用户认证过程中, 会发现资源文件也被安全框架挡在了外面, 因此需要进行安全配置:

```
1  @Override
2  public void configure(WebSecurity web) throws Exception {
3      web.ignoring().antMatchers("/webjars/**/*", "/*.css", "/*.js");
4  }
```

现在前端框架的静态资源完全可以通过 `webjars` 统一管理, 因此注意配置 `/webjars/**/*`。

## 处理不同类型的请求

前后端分离的系统中, 一般后端仅提供接口 JSON 格式的数据, 以供前端自行调用。刚才那样, 调用了被保护的接口, 直接进行了页面的跳转, 在web端还可以接受, 但是在 App 端就不行了, 所以我们还需要做进一步的处理。

这里做一下简单的思路整理

这里提供一种思路, 核心在于运用安全框架的: `RequestCache` 和 `RedirectStrategy`

```
1  import java.io.IOException;
2
3  import javax.servlet.http.HttpServletRequest;
4  import javax.servlet.http.HttpServletResponse;
5
6  import org.springframework.http.HttpStatus;
7  import org.springframework.security.web.DefaultRedirectStrategy;
8  import org.springframework.security.web.RedirectStrategy;
9  import org.springframework.security.web.savedrequest.HttpSessionRequestCache;
10 import org.springframework.security.web.savedrequest.RequestCache;
11 import org.springframework.security.web.savedrequest.SavedRequest;
12 import org.springframework.util.StringUtils;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.ResponseStatus;
15 import org.springframework.web.bind.annotation.RestController;
16
17 import lombok.extern.slf4j.Slf4j;
18
19 @Slf4j
20 @RestController
21 public class BrowserSecurityController {
22
23     // 原请求信息的缓存及恢复
24     private RequestCache requestCache = new HttpSessionRequestCache();
25
26     // 用于重定向
27     private RedirectStrategy redirectStrategy = new DefaultRedirectStrategy();
28
29     /**
30      * 当需要身份认证的时候, 跳转过来
31      * @param request
32      * @param response
33      * @return
34      */
35     @RequestMapping("/authentication/require")
36     @ResponseStatus(code = HttpStatus.UNAUTHORIZED)
37     public String requireAuthentication(HttpServletRequest request, HttpServletResponse response) throws
38         SavedRequest savedRequest = requestCache.getRequest(request, response);
39
40     if (savedRequest != null) {
41         String targetUrl = savedRequest.getRedirectUrl();
42         log.info("引发跳转的请求是:" + targetUrl);
43         if (StringUtils.endsWithIgnoreCase(targetUrl, ".html")) {
44             redirectStrategy.sendRedirect(request, response, "/login.html");
45         }
46     }
47
48     return "访问的服务需要身份认证, 请引导用户到登录页";
49 }
50 }
```

**注意:** 这个 `/authentication/require` 需要配置到安全认证配置: 配置成默认登录界面, 并设置成任何人都可以访问, 并且这个重

定向的页面可以设计成配置，从配置文件中读取。

## 自定义处理登录成功/失败

在前后端分离的情况下，我们登录成功了可能需要向前端返回用户的个人信息，而不是直接进行跳转。登录失败也是同样的道理。这里涉及到了 Spring Security 中的两个接口 `AuthenticationSuccessHandler` 和 `AuthenticationFailureHandler`。自定义这两个接口的实现，并进行相应的配置就可以了。当然框架是有默认的实现类的，我们可以继承这个实现类再来自定义自己的业务：

### 成功登录处理类

```
1  import java.io.IOException;
2
3  import javax.servlet.ServletException;
4  import javax.servlet.http.HttpServletRequest;
5  import javax.servlet.http.HttpServletResponse;
6
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.security.core.Authentication;
9  import org.springframework.security.web.authentication.SimpleUrlAuthenticationSuccessHandler;
10 import org.springframework.stereotype.Component;
11
12 import com.fasterxml.jackson.databind.ObjectMapper;
13
14 import lombok.extern.slf4j.Slf4j;
15
16 @Slf4j
17 @Component("myAuthenctiationSuccessHandler")
18 public class MyAuthenctiationSuccessHandler extends SimpleUrlAuthenticationSuccessHandler {
19
20     @Autowired
21     private ObjectMapper objectMapper;
22
23     @Override
24     public void onAuthenticationSuccess(HttpServletRequest request,
25                                         HttpServletResponse response, Authentication authentication) throws IOException, ServletException {
26
27         log.info("登录成功");
28         response.setContentType("application/json;charset=UTF-8");
29         response.getWriter().write(objectMapper.writeValueAsString(authentication));
30     }
31 }
```

成功登录之后，通过 response 返回一个 JSON 字符串回去。这个方法中的第三个参数 `Authentication`，它里面包含了登录后的用户信息（UserDetails），Session 的信息，登录信息等。

登录成功之后的响应JSON：

```
1  {
2      "authorities": [
3          {
4              "authority": "ROLE_admin"
5          }
6      ],
7      "details": {
8          "remoteAddress": "127.0.0.1",
9          "sessionId": "8BFA4F61A7CEA774C00F616AAE8C307C"
10     },
11     "authenticated": true,
12     "principal": {
13         "password": null,
14         "username": "admin",
15         "authorities": [
16             {
17                 "authority": "ROLE_admin"
18             }
19         ],
20         "accountNonExpired": true,
21         "accountNonLocked": true,
22         "credentialsNonExpired": true,
23         "enabled": true
24     },
25     "credentials": null,
26     "name": "admin"
27 }
```

这里有个细节需要注意：

`principal` 中有个权限数组集合 `authorities`，里面的权限值是：`ROLE_admin`，而自定义的安全认证配置中配置的是：`admin`，所以 `ROLE_` 前缀是框架自己加的，后期取出权限集合的时候需要注意这个细节，以取决于判断是否有权限是使用字符串的包含关系还是等值关系。

## 登录失败处理类

```
1  import java.io.IOException;
2
3  import javax.servlet.ServletException;
4  import javax.servlet.http.HttpServletRequest;
5  import javax.servlet.http.HttpServletResponse;
6
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.http.HttpStatus;
9  import org.springframework.security.core.AuthenticationException;
10 import org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler;
11 import org.springframework.stereotype.Component;
12
13 import com.fasterxml.jackson.databind.ObjectMapper;
```

```

14
15 import lombok.extern.slf4j.Slf4j;
16
17 @Slf4j
18 @Component("myAuthenctiationFailureHandler")
19 public class MyAuthenctiationFailureHandler extends SimpleUrlAuthenticationFailureHandler {
20
21     @Autowired
22     private ObjectMapper objectMapper;
23
24     @Override
25     public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,
26                                         AuthenticationException exception) throws IOException, ServletException {
27
28         log.info("登录失败");
29         response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
30         response.setContentType("application/json;charset=UTF-8");
31         response.getWriter().write(objectMapper.writeValueAsString(exception.getMessage()));
32     }
33 }

```

将两个自定义的处理类配置到自定义配置文件中:

```

1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.context.annotation.Configuration;
3 import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
4 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
5 import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
6 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
7 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
8 import org.woodwhale.king.handler.MyAuthenctiationFailureHandler;
9 import org.woodwhale.king.handler.MyAuthenctiationSuccessHandler;
10
11 @Configuration
12 @EnableWebSecurity
13 public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
14
15     @Autowired
16     private MyAuthenctiationFailureHandler myAuthenctiationFailureHandler;
17
18     @Autowired
19     private MyAuthenctiationSuccessHandler myAuthenctiationSuccessHandler;
20
21     @Override
22     protected void configure(HttpSecurity http) throws Exception {
23         http.formLogin() // 定义当需要
24             .loginPage("/login") // 设置登录页面
25             .loginProcessingUrl("/user/login") // 自定义的登录接口
26             .successHandler(myAuthenctiationSuccessHandler)
27             .failureHandler(myAuthenctiationFailureHandler)
28             .defaultSuccessUrl("/home").permitAll() // 登录成功之后, 默认跳转的页面

```



```

29         .and().authorizeRequests() // 定义哪些URL需要被保护、哪些不需
30         .antMatchers("/", "/index").permitAll() // 设置所有人都可以访问登录页面
31         .anyRequest().authenticated() // 任何请求, 登录后可以访问
32         .and().csrf().disable(); // 关闭csrf防护
33     }
34
35     @Override
36     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
37         auth.inMemoryAuthentication()
38             .passwordEncoder(new BCryptPasswordEncoder()).withUser("admin")
39             .password(new BCryptPasswordEncoder().encode("admin"))
40             .roles("admin");
41     }
42
43 }

```

注意: `defaultSuccessUrl` 不需要再配置了, 实测如果配置了, 成功登录的 handler 就不起作用了。

## 小结

可以看出, 通过自定义的登录成功或者失败类, 进行登录响应控制, 可以设计一个配置, 以灵活适配响应返回的是页面还是 JSON 数据。

## 结合thymeleaf

在前端使用了 `Thymeleaf` 进行渲染, 特使是结合 `Spring Security` 在前端获取用户信息

依赖添加:

```

1 <dependency>
2     <groupId>org.thymeleaf.extras</groupId>
3     <artifactId>thymeleaf-extras-springsecurity5</artifactId>
4 </dependency>

```

## 注意:

因为本项目使用了spring boot 自动管理版本号, 所以引入的一定是完全匹配的, 如果是旧的 spring security 版本需要手动引入对应的版本。

引用官方版本引用说明:

```

1 thymeleaf-extras-springsecurity3 for integration with Spring Security 3.x
2 thymeleaf-extras-springsecurity4 for integration with Spring Security 4.x
3 thymeleaf-extras-springsecurity5 for integration with Spring Security 5.x

```

具体语法可查看：

<https://github.com/thymeleaf/thymeleaf-extras-springsecurity>

## 常用的语法标签

这里为了表述方便，引用了上小节中的“自定义处理登录成功/失败”的成功响应JSON数据：

```
1  {
2      "authorities": [
3          {
4              "authority": "ROLE_admin"
5          }
6      ],
7      "details": {
8          "remoteAddress": "127.0.0.1",
9          "sessionId": "8BFA4F61A7CEA774C00F616AAE8C307C"
10     },
11     "authenticated": true,
12     "principal": {
13         "password": null,
14         "username": "admin",
15         "authorities": [
16             {
17                 "authority": "ROLE_admin"
18             }
19         ],
20         "accountNonExpired": true,
21         "accountNonLocked": true,
22         "credentialsNonExpired": true,
23         "enabled": true
24     },
25     "credentials": null,
26     "name": "admin"
27 }
```

`sec:authorize="isAuthenticated()`：判断是否有认证通过

`sec:authorize="hasRole('ROLE_ADMIN')"` 判断是否有 `ROLE_ADMIN` 权限

**注意：**上述的 `hasRole()` 标签使用能成功的前提是：自定义用户的权限字符集必须是以 `ROLE_` 为前缀的，否则解析不到，即自定义的 `UserDetailsService` 实现类的返回用户的权限数组列表的权限字段必须是 `ROLE_***`，同时在 html 页面中注意引入对应的 `xmlns`，本例这里引用了：

```
1  <html xmlns:th="http://www.thymeleaf.org"
2      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity5">
```

`sec:authentication="principal.authorities"`：得到该用户的所有权限列表

`sec:authentication="principal.username"`：得到该用户的用户名

当然也可以获取更多的信息，只要 `UserDetailsService` 实现类中返回的用户中携带有的信息均可以获取。

## 常见异常类

1	<code>AuthenticationException</code>	常用的子类：（会被底层换掉，不推荐使用）
2	<code>UsernameNotFoundException</code>	用户找不到
3	<code>BadCredentialsException</code>	坏的凭据
4	<code>AccountStatusException</code>	用户状态异常它包含如下子类：（推荐使用）
5	<code>AccountExpiredException</code>	账户过期
6	<code>LockedException</code>	账户锁定
7	<code>DisabledException</code>	账户不可用
8	<code>CredentialsExpiredException</code>	证书过期

参考资料：

<https://blog.csdn.net/u013435893/article/details/79596628>

[https://blog.csdn.net/canon\\_in\\_d\\_major/article/details/79675033](https://blog.csdn.net/canon_in_d_major/article/details/79675033)

<https://juejin.im/post/5c46a49e51882528735ef2d9>

<https://www.jianshu.com/p/6307c89fe3fa/>

<https://mp.weixin.qq.com/s/NKhwU6qKKU0Q0diA0hg13Q>

[https://mp.weixin.qq.com/s/sMi1\\_Rw\\_s75YDaldmTWKw](https://mp.weixin.qq.com/s/sMi1_Rw_s75YDaldmTWKw)

<https://blog.csdn.net/smd2575624555/article/details/82759863>

<https://www.cnblogs.com/yyxxn/p/8808851.html>

[https://blog.csdn.net/coder\\_py/article/details/80330868](https://blog.csdn.net/coder_py/article/details/80330868)

参考项目源码：

<https://github.com/whyalwaysmea/Spring-Security>

<https://github.com/oycyqr/SpringSecurity>

<https://github.com/chengjiansheng/cjs-springsecurity-example>

updated 2019-07-06


本文结束 ★ 感谢阅读

本文标题: SpringBoot + Spring Security 学习笔记（一）自定义基本使用及个性化登录配置

本文作者: 木鲸鱼

微信公号: 木鲸鱼 | woodwhales

原始链接: <https://woodwhales.github.io/2019/04/12/026/> 

许可协议:  署名-非商业性使用-禁止演绎 4.0 国际 转载请保留原文链接及作者。

 Spring Security[← Mybatis Plus 学习笔记](#)[SpringBoot + Spring Security 学习笔记（二）安全认证流](#) [程源码详解](#)

昵称

邮箱

网址(http://)

网络一线牵珍惜,珍惜这段缘 您可以填写邮箱，回复评论时，您会收到邮件通知。

表情 | 预览

回复

### 3 评论



Anonymous Chrome 78.0.3904.70 Windows 10.0

2019-10-28

666

[回复](#)

nalán Chrome 73.0.3683.86 Windows 10.0



2019-09-13

回复

更新一下邮箱



Anonymous Chrome 73.0.3683.86 Windows 10.0

2019-09-13

回复

文章质量非常高，我竟然是第一个评论的！

Powered By [Valine](#)

v1.3.10

© 2018 – 2019 ♥ 木鲸鱼

👤 访问人数 11771 人 | 👁 访问总量 20429 次