

Spring Security

1.1 SpringSecurity 框架基础

1.1.1 What is Spring Security?



作用：做认证和授权的；

认证：登录；

授权：权限管理，给用户颁发权限，有一些资源不能让用户看，只有有权限的人才能看；

官网: <https://spring.io/projects/spring-security>



Why Spring ▾ Learn ▾ Projects ▾ Academy ▾ Solutions ▾ Community ▾ ⚙️

- Spring Boot
- Spring Framework
- Spring Data >
- Spring Cloud >
- Spring Cloud Data Flow
- Spring Security ▾**
 - Spring Security Kerberos
- Spring Authorization Server
- Spring for GraphQL
- Spring Session >
- Spring Integration
- Spring HATEOAS
- Spring Modulith
- Spring REST Docs
- Spring AI

Spring Security 6.3.0



OVERVIEW LEARN SUPPORT SAMPLES

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements

Features

- Comprehensive and extensible support for both Authentication and Authorization
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc
- Servlet API integration
- Optional integration with Spring Web MVC
- Much more...

Spring Security 是一个功能强大且高度可定制的身份认证和访问控制框架。它是保护基于 Spring 的应用程序的事实上的标准。(Shiro 框架)

Spring Security 是一个致力于为 Java 应用程序提供身份认证和授权的框架。像所有 Spring 项目一样, Spring Security 的真正强大之处在于它可以非常轻松地扩展来满足自定义需求;

1.1.2 Spring Security 快速上手

spring security 现在开发时不会采用 spring 进行开发, 而是采用 spring boot 进行开发;

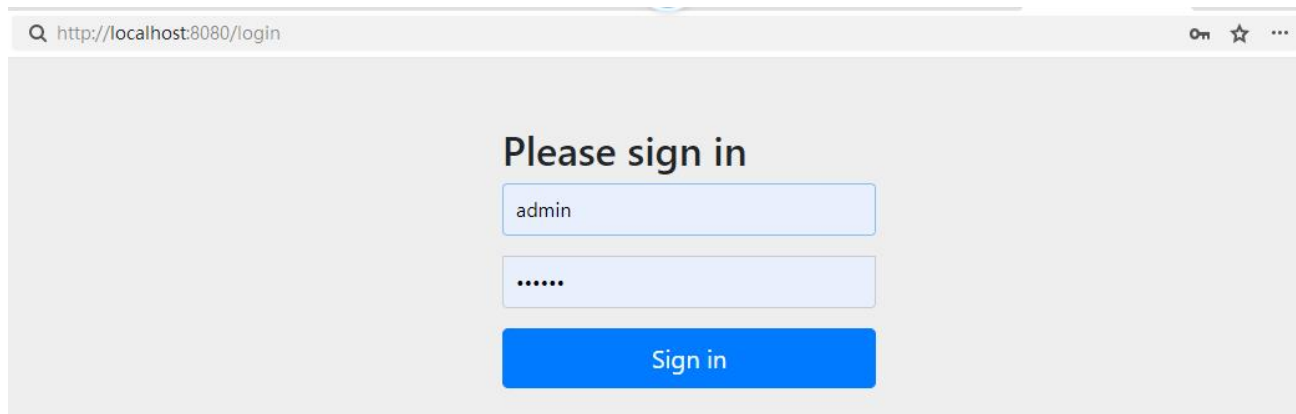
spring security 本身也是在 spring boot 流行之后才被大量使用;

spring boot 流行 2017 年

依赖:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

访问: <http://localhost:8080/hello>



项目中一旦添加了 spring security 的 jar 包依赖, 那么所有的 controller 接口路径访问时都会被 spring security 拦截, 它会检查你是否登录, 如果未登录, 就会跳转到它的一个默认登录页, 如果登录了, 那么可以直接访问 controller 的路径;

1.1.3 Spring Security 基本原理分析

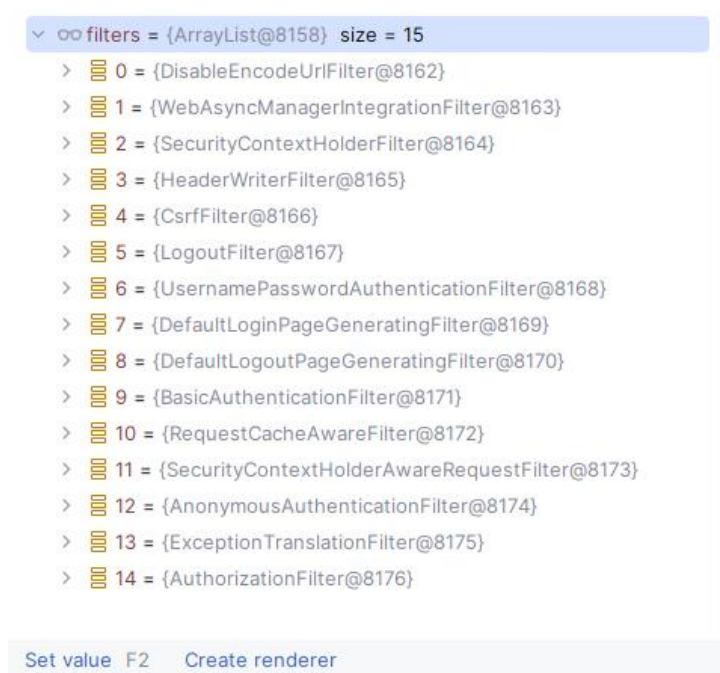
<http://localhost:8080/hello> --> <http://localhost:8080/login>

重定向;

Spring Security 采用 15 个 Filter 进行过滤拦截; (基于 session)

入口: FilterChainProxy

代码: `List<Filter> filters = this.getFilters((HttpServletRequest)request);`



[DefaultLoginPageGeneratingFilter](#) 生成登录的页面;

[DefaultLogoutPageGeneratingFilter](#) 生成退出的页面;

登录跳转地址是: [/login](#) (这是 Spring Security 框架提供的, 不是我们写的)

退出跳转地址是: [/logout](#) (这是 Spring Security 框架提供的, 不是我们写的)

默认情况下, 用户名是 [user](#), 密码是临时生成的 uuid; (来自 SecurityProperties 类)

[String name = "user";](#)

[String password = UUID.randomUUID\(\).toString\(\);](#)

可以修改默认的用户名和密码, 在配置文件 application.properties 中配置:

```
#自己指定登录的用户名和密码  
spring.security.user.name=cat  
spring.security.user.password=aaalll
```

1.2 SpringSecurity 框架登录认证

1.2.1 Spring Security 基于数据库查询登录

1.2.1.1 建项目;

1.2.1.2 加依赖;

1.2.1.3 配文件;

1.2.1.4 写代码;

核心代码是实现 UserDetailsService 接口的一个方法，实现从数据库查询用户登录;

```
@Override
public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
    TUser user = tUserMapper.selectByUsername(username);

    if (user == null) {
        throw new UsernameNotFoundException("用户不存在");
    }

    UserDetails userDetails = User.builder() //构建器模式
        .username(user.getLoginAct()) //设置用户名
        .password(user.getLoginPwd()) //设置密码
        .authorities(AuthorityUtils.NO_AUTHORITIES) //设置权限，权限暂时为空
        .build();
    return userDetails;
}
```

1.2.1.5 去运行;

(1) 老版本报错: `java.lang.IllegalArgumentException: You have entered a password with no PasswordEncoder.`

(2) 新版本报错: `You have entered a password with no PasswordEncoder. If that is your intent, it should be prefixed with `{noop}`.`

这是因为没有加入密码加密器导致的;

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

1.2.2 Spring Security 数据库登录流程分析

- 1、访问 <http://localhost:8080/>
- 2、被 spring security 的 filter 过滤器拦截 (里面有 15 个 Filter) ;
- 3、由于没有登录过, 所以 spring security 就跳转到登录页 (登录页是框架生成的)
- 4、我们在登录页输入账号和密码去登录提交; (账号和密码是数据库的账号密码)
- 5、spring security 里面的 `UsernamePasswordAuthenticationFilter` 接收账号和密码;
- 6、第 5 步的这个 filter 会调用 `loadUserByUsername(String username)` 方法去数据库查询用户;
- 7、从数据库查询到用户后, 把用户组装成 `UserDetail` 对象, 然后返回给 SpringSecurity 框架;

8、第 7 步返回后，再回到框架的 filter 里面进行用户状态的判断，用户对象中默认有 4 个状态字段，如果这 4 个状态字段的值都是 true，该用户才能登录，否则就是提示用户状态不正常，不能登录的（框架中实际上只判断 3 个状态值，那个密码是否过期没有做判断）；

9、第 7 步返回后，再回到框架的 filter 里面进行密码的匹配，如果密码匹配上了，就登录成功，否则失败；

10、比较密码代码：

```
this.additionalAuthenticationChecks(user,
(UsernamePasswordAuthenticationToken) authentication);

String presentedPassword = authentication.getCredentials().toString();

if (!this.passwordEncoder.matches(presentedPassword, userDetails.getPassword())) {

    this.logger.debug("Failed to authenticate since password does not match stored value");

    throw new BadCredentialsException(this.messages.getMessage("AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad credentials"));

}
```

1.2.3 Spring Security 自定义登录页

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
    return httpSecurity
        .formLogin((formLogin) -> {
            formLogin.loginPage("/toLogin") //配置登录页面
                .loginProcessingUrl("/login"); //配置登录处理的 url，也就是登录请求提交到哪个地址上
        })
        .authorizeHttpRequests((authorizeHttpRequests) -> {
            authorizeHttpRequests.requestMatchers("/toLogin").permitAll() ///toLogin 允许未登录就可以访问
                .anyRequest().authenticated(); //所有请求都需要认证(登录)
        })
        .build();
}
```

```
}
```

1.2.4 Spring Security 验证码登录

1、用户名、密码、图形验证码登录;

图形验证码 (java.awt.*, javax.swing.*, Java 的图形编程)

验证码生成: <https://www.hutool.cn/> (糊涂) 这个 jar 包提供了很多工具类;

2、提交登录, 要从数据库查询登录;

```
<body>
<!-- action="/userLogin" 要和 .loginProcessingUrl("/userLogin") 保持一致 -->
<form action="/userLogin" method="post">
    账号: <input type="text" name="username" /> <br/>
    密码: <input type="password" name="password"> <br/>
    验证码: <input type="text" name="captcha" />
    
    <span th:text="${session?.errorMsg}" style="color: #FF0000;" ></span>
    <br/>
    <button type="submit">登 录</button>
</form>

<script th:inline="javascript">
    function reImg() {
        var img = document.getElementById("captchaCode");
        img.src = "/captchaCode?random=" + Math.random();
    }
</script>
</body>
```

```

```

```
<a th:href="@{ '/api/user' }">链接</a>
```

```
@Bean
```

```
public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
    // 配置登录之前添加一个验证码的过滤器 (一定要加)
```



```
httpSecurity.addFilterBefore(captchaFilter, UsernamePasswordAuthenticationFilter.class);

return httpSecurity.formLogin(formLogin -> {
    formLogin.loginPage("/toLogin")
        .loginProcessingUrl("/userLogin") //filter 实现的登录, 你自己不用处理这个 url 的请求
        .failureUrl("/toLogin?error")
        .defaultSuccessUrl("/index"); //登录成功
})
.logout(logout -> {
    logout.logoutUrl("/toLogout").logoutSuccessUrl("/toLogin");
})
.authorizeRequests(authorize -> {
    authorize.requestMatchers(
        "/toLogin",
        "/toLogout",
        "/hello",
        "/captchaCode").permitAll()
    .anyRequest().authenticated();
})
.csrf(csrf -> {
    csrf.disable();
})
.build();
}
```

1.2.5 Spring Security 验证码登录流程分析

- 1、访问 <http://localhost:8080/>
- 2、被 spring security 的 filter 过滤器拦截（里面有 15 个 Filter）；
- 3、由于没有登录过，所以 spring security 就跳转到自定义的登录页 login.html；
- 4、我们在登录页输入账号、密码、验证码 去提交登录；
- 5、CaptchaFilter（我们写的）拦截登录请求，验证一下验证码对不对；

6、验证码正确,就执行下一个 Filter,调用 UsernamePasswordAuthenticationFilter(Spring Security 框架的) 接收账号和密码

7、 UserDetailsService.loadUserByUsername() (我们覆盖该方法) --> userMapper (mybatis) --> 查数据库 --> 返回 userDetails (框架的);

8、把 userDetails 返回给 (框架) 进行用户状态检查和密码比较;

this.additionalAuthenticationChecks(user,

(UsernamePasswordAuthenticationToken)authentication);

DaoAuthenticationProvider 类的 additionalAuthenticationChecks(..)方法里面

this.passwordEncoder.matches(presentedPassword, userDetails.getPassword())进行密码匹配, 匹配上了, 则认证 (登录) 成功, 否则认证失败;

1.2.6 Spring Security 密码加密和密码匹配

对于用户密码的保护, 通常都会进行加密然后存放在数据库中; (基本常识)

目前密码加密 MD5 和 BCrypt 比较流行, Spring Security 默认是采用 BCrypt;

Spring Security 密码加密接口: PasswordEncoder;

我们对 123 字符串加密三次, 然后匹配三次, 看看效果;

```
public class TestPasswordEncoder {
    public static void main(String[] args) {
        BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
        String encode1 = passwordEncoder.encode("123");
        System.out.println(encode1);
        String encode2 = passwordEncoder.encode("123");
        System.out.println(encode2);
        String encode3 = passwordEncoder.encode("123");
        System.out.println(encode3);
        // 查看加密后是否匹配
    }
}
```

```
System.out.println(passwordEncoder.matches("123", encode1));
System.out.println(passwordEncoder.matches("123", encode2));
System.out.println(passwordEncoder.matches("123", encode3));
}
}
```

查看控制台发现特点是：相同的字符串加密之后的结果都不一样，但是比较的时候是一样的：

```
$2a$10$DRirerHn6phXKvVZEKvcAO4nUuCnBochSXRbWetDtttzEOpK0VMmC
$2a$10$rfoSkKEsrXf7URiv1oDrUu26QBvDwSmSSelDsXA4wiLd9Lxn.eZwq
$2a$10$DJSAJE5QBVhe5ez0io62seB9U1ymrvahkAiyFcJ48pWZYrdB5zt3y
true
true
true
```

1.2.7 Spring Security BCrypt 密码加密和密码匹配原理

1.2.7.1 BCrypt 加密原理

输入的明文密码比如 `aaa111`，通过随机加盐 salt (abcdefghijklmnopqrstuvwxyz.....22 位字符串)后再进行加密得到

密文密码 `xxx` (version+salt+hash)，然后存入数据库；

`bcrypt(aaa111 + 22 位随机的盐 abcxyz) = 密文`

1.2.7.2 密码匹配原理

系统在验证用户的密码时，需要从密文密码 `xxx` 中取出盐 salt (22 位)，然后与用户页面输入的 password (`aaa111`) 进行加密，把得到的结果与保存在数据库中的密文 `xxx` 进行比对，如果一致才算验证通过；

明文：`aaa111`

密文：`$2a$10$ 9z08lUjY.Htp4xdWLT7TzO wrz4MGz4V7tt1m/61HdebDqR2m7Oj52`

比较：`bcrypt (aaa111 + 9z08lUjY.Htp4xdWLT7TzO) --> 密文 == 上面这个密文`；

1.2.8 Spring Security 获取当前登录用户信息

添加获取当前用户信息的 Controller

```
@GetMapping("userInfo")
public Principal getUserInfo(Principal principal) {
    return principal;
}

@GetMapping("userInfo2")
public Object getUserInfo2() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    return authentication;
}
```

测试访问

<http://localhost:8080/userInfo>

<http://localhost:8080/userInfo2>



```
{
  "authorities": [],
  "details": {
    "remoteAddress": "0:0:0:0:0:0:0:1",
    "sessionId": "3DA54F3A82B1700127D1FE8D1D0F43C3"
  },
  "authenticated": true,
  "principal": {
    "password": null,
    "username": "user",
    "authorities": [],
    "accountNonExpired": true,
    "accountNonLocked": true,
    "credentialsNonExpired": true,
    "enabled": true
  },
  "credentials": null,
  "name": "user"
}
```

Principal

Authentication

UsernamePasswordAuthenticationToken

1.3 SpringSecurity 框架权限管理

1.3.1 基于角色的权限管理

用户（登录了的） --> 给用户配置角色 --> 给角色配置能访问的资源

1.3.1.1 基于角色的权限控制

- (1)、需要有一个用户；（从数据库查询用户）
- (2)、给用户配置角色；（从数据库查询用户的角色）
- (3)、给角色配置能访问的资源（这一步采用切面拦截，使用的是注解）

RBAC (基于角色的访问控制) role base access control

1.3.1.2 权限拦截注解

`@PreAuthorize` 在方法调用前进行权限检查; (常用)

`@PostAuthorize` 在方法调用后进行权限检查; (很少用)

上面的两个注解如果要使用的话必须加上

`@EnableMethodSecurity(prePostEnabled = true)`

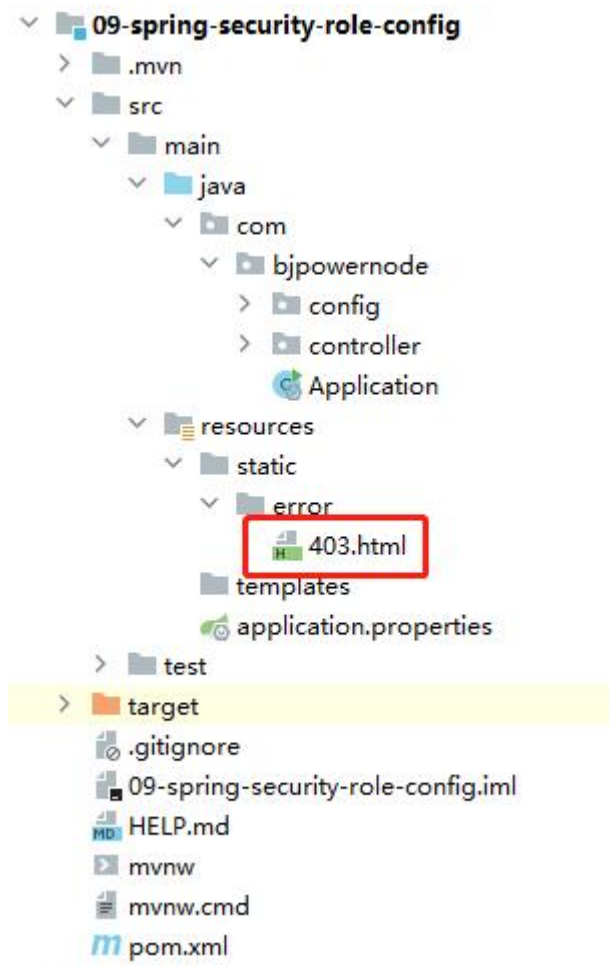
不加注解的，都可以访问，加了注解的，要有对应权限才可以访问

```
@PreAuthorize("hasRole('dev')")
@RequestMapping("/query")
public String query() {
    return "query";
}

@PreAuthorize("hasRole('dev')")
@RequestMapping("/add")
```

```
public String add() {  
    return "add";  
}  
  
@PreAuthorize("hasRole('admin')")  
@RequestMapping("/del")  
public String del() {  
    return "del";  
}  
  
@PreAuthorize("hasRole('admin')")  
@RequestMapping("/update")  
public String update() {  
    return "update";  
}  
  
@PreAuthorize("hasRole('admin')")  
@RequestMapping("/admin/index")  
public String admin() {  
    return "admin";  
}
```

1.3.1.3 自定义无权限页面



1.3.2 基于资源的权限管理

资源是什么？资源就是我们的 controller 的 http 接口；

- (1)、需要有一个用户；（从数据库查询用户）
- (2)、给用户配置权限标识符（权限 code, 权限代码）；（从数据库查询用户的权限标识符）
- (3)、给每个权限标识符配置能访问的资源；（这一步采用切面拦截，使用的是注解）

1.4 Spring Security 前后端分离登录认证

1.4.1 Spring Security 返回 JSON

在前面的例子中，我们是返回到 Thymeleaf 页面，但如果是前后端分离开发，是不能返回一个页面的，而应该是返回一个 JSON；

Vue + springboot/spring security/mybatis (Nginx + Tomcat) 无法共享 session

Thymeleaf + springboot/spring security/mybatis (Tomcat) 共享 session

- 1、建项目
- 2、看依赖
- 3、配文件
- 4、写代码
- 5、去运行

前端：Vue 页面我们此案例先不写，使用 Apifox 工具代替；

1.4.1.1 跨域问题

- 1、协议不同会跨域 <https://localhost:8080> <http://localhost:8080>
- 2、端口不同会跨域： <http://localhost:10492> <http://localhost:8080>
- 3、域名不同会跨域： <http://bjpowernode.com> <http://baidu.com>

三个里面有任何一个不同，都是跨域，跨域是浏览器不允许的，浏览器是为了安全，不允许你跨域访问；

1.4.1.2 Axios 发送请求的返回值

axios 发送异步请求，返回的 response 对象中有 6 个字段，其中 data 字段就是我们后端返回的数据，其他字段一般在项目开发中很少使用：

```
config : {transitional: {...}, adapter: Array(2), transformRequest: Array(1), transformResponse: Array(1), timeout: 0, ...}
data : "OK"
headers : AxiosHeaders {cache-control: 'no-cache, no-store, max-age=0, must-revalidate', content-length: '2', expires: '0', pragma: 'no-cache'}
request : XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, ...}
status : 200
statusText : ""
```

1.4.1.3 Js 中三个等于符号

三个等于符号是：

```
1、数据类型要相同；
2、数据值也要相同；
let age1 = "18"; //string
let age2 = 18; //int
```

```
age1 === age2 --> false
age1 == age2 --> true
```

后端：登录接口、退出接口、获取登录人信息接口

(1) 登录接口：/user/login (spring security 框架提供的，我们不需要写 Controller)

① username、password、_csrf(我们拿不到，后端要禁用)

(2) 退出接口：(spring security 框架提供的，我们不需要写 Controller)

//任何请求都需要认证(登录)后才能访问，但是 spring security 提供的地址是不会受下面的代码控制，比如 /user/login、/user/logout 都不会被拦截

```
authorizeHttpRequests.anyRequest().authenticated();
```

流程：

1、 <http://localhost:8080/user/login>,

Config 类中配置了 `formLogin.loginProcessingUrl("/user/login");` //登录处理 url，也就是登录提交到哪个地址上

2、 UsernamePasswordAuthenticationFilter 接收账号和密码；

3、 调用 `UserServiceImpl.loadUserByUsername(String username)` 查询数据库；

4、 返回 UserDetails（我们已经改成 TUser 对象）

5、 框架中检查 TUser 对象的 4 个状态（实际上只检查 3 个，密码的状态不检查）

6、 框架中比较密码，如果密码匹配就登录成功，否则失败；

7、 登录成功默认是跳转到上一个请求地址（我们此案例上一个请求地址是空的，我们一上来就是直接请求 `/user/login`，在这之前，我们没有请求任何后端地址，所以上一个地址是空的，上一个地址是空的，那么就默认跳转到斜杠/项目根路径）；

8、 登录失败，那么就跳转到 `/login?error`（该地址就是跳转到框架内部生成的登录页）

9、 `/user/login` 登录成功后，我们不要跳转到默认路径或页面，因为是前后端分离开始，登录成功后，应该给前端返回一个 json，告诉前端登录结果，前端自己进行页面跳转；那么后端通过一个 handler 返回 json 告诉前端结果；

10、 `/user/logout` 登录失败后，我们不要跳转到默认路径或页面，因为是前后端分离开始，登录成功后，应该给前端返回一个 json，告诉前端登录结果，前端自己进行页面跳转；那么

后端通过一个 handler 返回 json 告诉前端结果;

11、接下来, 用户登录成功了, 但是我们访问/welcome 接口, 你发现/welcome 接口仍然提示你需要登录, 为什么? 因为服务器端已经不记录 session 了; 那怎么解决该问题? 就是我们下面要学习的 jwt 解决该问题;

1.4.1.4 登录成功的处理器

AuthenticationSuccessHandler

上面的分析步骤的第 6 步, 变成跳转到 AuthenticationSuccessHandler, 执行该 handler 的方法, 返回 json 数据;

1.4.1.5 登录失败的处理器

AuthenticationFailureHandler

1.4.1.6 退出成功的处理器

LogoutSuccessHandler

LogoutHandler

凡是 Spring Security 框架提供的处理接口 (/login, /logout) , 是不会进行登录拦截;

1.4.1.7 配置处理器

```
@Configuration
public class SecurityConfig {
    @Resource
    private UserAuthenticationSuccessHandler userAuthenticationSuccessHandler;
    @Resource
```

```
private UserAuthenticationFailureHandler userAuthenticationFailureHandler;

@Resource
private UserLogoutSuccessHandler userLogoutSuccessHandler;

@Bean
public SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {
    // 给一个表单登陆 就是我们的登录页面, 登录成功或者失败后走我们的 url
    return httpSecurity.formLogin( formLogin -> {
        formLogin.successHandler(userAuthenticationSuccessHandler)
            .failureHandler(userAuthenticationFailureHandler);
    })
    .logout( logout -> {
        logout.logoutSuccessHandler(userLogoutSuccessHandler);
    })
    .authorizeRequests( authorizer -> {
        authorizer.anyRequest()
            .authenticated();
    })
    .build();
}
```

1.4.2 Spring Security 前后端分离无会话状态

```
//禁用 session、cookie 机制（因为我们是前后端分离项目的开发）
.sessionManagement( sessionManagement -> {
    sessionManagement.sessionCreationPolicy(SessionCreationPolicy.STATELESS);
//无状态策略
})
```

没有 session、前端 cookie 中也不会存储 sessionid；那么这样的话，用户状态怎么保持呢？

需要使用我们下面介绍的 jwt 解决该问题；

不分离的：Tomcat 【thymeleaf <----> (controller、scurity) 】

前后端分离：Nginx 【Vue】 <----jwt----> Tomcat 【 (controller、scurity) 】

1.4.3 JWT (JSON Web Token)

1.4.3.1 JWT 是什么？

JSON Web Tokens are an open, industry standard **RFC 7519** method for representing claims securely between two parties.

JWT (JSON Web Token) 是一种开放的行业标准 (RFC 7519)，用于安全地双方之间传输信息，常用于各方之间传输信息，特别是在身份认证领域使用非常广泛；

官网：<https://jwt.io/>

1.4.3.2 JWT 的数据结构

JWT 的数据结构像下面这样：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

它是一个很长的字符串，中间用点 (.) 分隔成三个部分；

注意，JWT 内部是没有换行的，这里只是为了便于展示，将它写成了几行；

JWT 的三个部分依次是：

- Header (头部)
- Payload (负载) 这里面可以携带业务数据 (比如一些参数)
- Signature (签名)

写成一行，就是下面的样子：

Header.Payload.Signature

JWT TOKEN



下面分别介绍一下这三个部分：

1.4.3.2.1 Header

Header 部分原文是一个 JSON 对象，描述 JWT 的元数据，通常如下：

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

其中 alg 属性表示签名的算法 (algorithm) ， 默认是 HMAC SHA256 (写成 HS256) ；

typ 属性表示这个令牌 (token) 的类型 (type) ， JWT 令牌统一写为 JWT；

最后，将上面的 JSON 对象使用 [Base64URL 算法](#)转成字符串，就得到 Header 部分；

1.4.3.2.2 Payload

Payload 部分原文也是一个 JSON 对象，用来存放实际需要传递的数据，JWT 定义了 7 个官

方字段供选用：

iss (issuer)：签发人
exp (expiration time)：过期时间
sub (subject)：主题
aud (audience)：受众
nbf (Not Before)：生效时间
iat (Issued At)：签发时间
jti (JWT ID)：编号

但是我们可以不使用官方的字段，我们可以使用任何字段来传递数据，比如：

```
{
  "number": "1234567890",
  "name": "cat",
}
```

```
"phone": "13700000000"
}
```

这个 JSON 对象也要使用 [Base64URL 算法](#)转成字符串;

注意, Base64URL 算法不是加密算法, 它是[编码算法](#), 是可以解码出原文的, 也就是 JWT 负载中的数据任何人都可以解码得到原文 (不安全), 所以不要把私密信息 (密码, 验证码等) 放在这个部分; (虽然可以解码出来, 但是我们把密码比如加密之后再放在负载里面, 也是没有问题, 是安全的)

```
{
  Id : 10285
  Name : 'cat'
  Passowrd: '$2a$10$QcgTWQSZ11b6BDIPjsUDT0sR9BTS1e.LUvTY.3RirFyR0.5PBfEM0'
}
```

1.4.3.2.3 Signature

Signature 部分是对前两部分的签名, [防止数据篡改](#);

首先, 需要指定一个密钥 (secret), 这个密钥只有服务器才知道, 不能泄露给用户, 然后, 使用 Header 里面指定的签名算法 (默认是 HMAC SHA256), 按照下面的公式产生签名;

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

算出签名以后, 把 Header、Payload、Signature 三个部分拼成一个字符串, 每个部分之间用"点" (.) 分隔, 就可以返回给用户;

```
base64UrlEncode(header) . base64UrlEncode(payload) . HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload), secret)
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJudW1iZXIiOiIxMjMONTY3ODkwIiwibmFtZSI6ImNhZCIsInBob251IjoiaMTM3MDAwMDAwMDAifQ.cGLcaxSy19dgiL_hK5wv1Pbgj8PGHFVXEWRdVW01N4
```

1.4.3.3 JWT 的使用

```
<!-- 添加 jwt 的依赖 -->
```

```
<dependency>
```

```
    <groupId>com.auth0</groupId>
```

```
    <artifactId>java-jwt</artifactId>
```

```
    <version>4.4.0</version>
```

```
</dependency>
```

```
<!-- hutool-jjwt -->
```

```
<dependency>
```

```
    <groupId>cn.hutool</groupId>
```

```
    <artifactId>hutool-jwt</artifactId>
```

```
    <version>5.8.32</version>
```

```
</dependency>
```

```
/**
```

```
 * 生成 JWT
```

```
 */
```

```
public String createToken(String userJson) {
```

```
    //组装头数据
```

```
    Map<String, Object> header = new HashMap<>();
```

```
    header.put("alg", "HS256");
```

```
    header.put("typ", "JWT");
```

```
    return JWT.create()
```

```
        //头
```

```
        .withHeader(header)
```

```
        //自定义数据
```

```
        .withClaim("user", userJson)
```

```
        //签名算法
```

```
        .sign(Algorithm.HMAC256(secret));
```

```
}
```

```
/**
```

```
 * 验证 JWT
```

```
 * @param token 要验证的 jwt 的字符串
```

```
 */
```

```
public Boolean verifyToken(String token) {
```

```
    try {
```

```
        // 使用密钥创建一个解析对象
```

```
        JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256(secret)).build();
```

```
        //验证 JWT
```



```
        jwtVerifier.verify(token);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}

/**
 * 解析 JWT 的数据
 */
public String parseToken(String token) {
    try {
        // 使用密钥创建一个解析对象
        JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256(secret)).build();
        // 验证 JWT
        DecodedJWT decodedJWT = jwtVerifier.verify(token);
        Claim user = decodedJWT.getClaim("user");
        return user.asString();
    } catch (TokenExpiredException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}
```

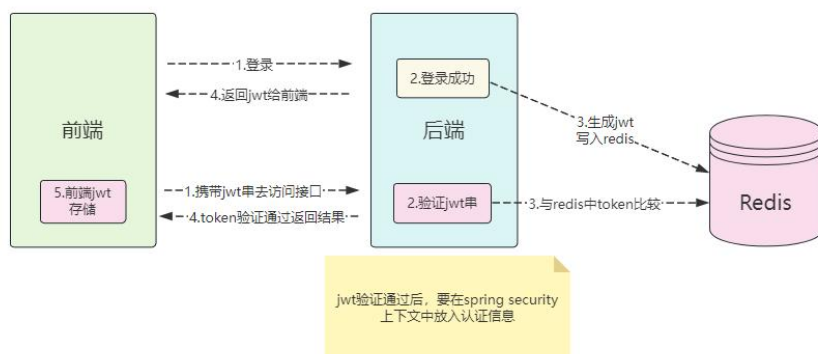
1.4.3.4 Spring Security+JWT 实现前后端分离的登录认证

解决之前的问题：

- 1、请求/user/login 接口并且登录成功了；
- 2、然后访问/welcome 接口，此时又提示我们需要登录；

Nginx (html) --> axios 发送请求 --> Tomcat (spring boot web 项目)

原因：前后端分离，无法使用 cookie 中的 jsessionId 和后端 session 保持登录状态，另外后端也禁用了 session，没有保持登录状态，导致登录之后，访问其他接口的时候，又提示需要登录；



1、#bind 127.0.0.1 -::1 （默认是没有注释掉，我们应该把它注释掉，前面加个#号）

2、protected-mode no （默认是 yes，我们把它改成 no，不保护）

3、requirepass 123456 （默认 redis 没有密码，我们加个密码）

4、daemonize yes （默认是 no，我们把它改成 yes，后台启动）

启动 redis，进入/opt/tool/redis-7.2.3/src 下，执行：`./redis-server ./redis.conf`

redis 启动有一个警告：

96489:C 08 Jul 2024 14:59:03.796 # WARNING Memory overcommit must be enabled! Without it, a background save or replication may fail under low memory condition. Being disabled, it can also cause failures without low memory condition, see <https://github.com/jemalloc/jemalloc/issues/1328>. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.

查看防火墙状态

`systemctl status firewalld`

关闭防火墙（临时关闭，重启 centos 后，防火墙又打开了）

`systemctl stop firewalld`

关闭防火墙，把防火墙置为不可用（也就是永久关闭，重启 centos 后，防火墙也是关闭的）

`systemctl disable firewalld`

MySQL (jdbc --> mybatis 框架)

Redis (Jedis --> spring data redis 框架)

Spring-Data-Redis 框架 (spring 家族下的一个框架)

- 1、建项目
- 2、加依赖 (spring data redis 依赖)
- 3、配文件
- 4、写代码 (直接注入 RedisTemplate 这个 Bean)
- 5、去运行

Spring-Data-Redis 在操作 redis 的时候, 默认 key 和 value 都是采用 [jdk 序列化](#)之后再写入 redis 的;

KEY: `\xAC\xED\x00\x05t\x00\x013`

VALUE:

`\xAC\xED\x00\x05t\x02\x0CeyJ0eXAIoiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MywibG9naW5BY3QiOiJ6aGFuZ3FpIiwibG9naW5Qd2QiOiIkMmEkMTAkbnQ5Q1pVViFlQ2xrV05INng2N1BMT244Sko5UHU2S05SbmZHQm1JandsUU9kcGhRRTBLV2EiLCJuYW1lIjoi5byg55CqliwicGhvbmUiOiIxMzYyMzYyMzIzIiwiaW1haWwiOiJ6aGFuZ3FpQHFxLmNvbSIsImFjY291bnROb0V4cGlyZWQ5IjoiJEsImNyZWVlbnRpbWxzTm9FeHBpcmVkJjoxLCJhY2NvdW50Tm9Mb2NrZWQ5IjoiJEsImFjY291bnRFbmFibGVkIjoxLCJjcmVhdGVUaW1lIjoxNjc3NzI4MjU0MDAwLCJlZGl0VGltZSI6MTY4NDc3MjQ2MjAwMCwibGFzdExvZ2luVGltZSI6MTcwMjMwMDAwODAwMH0.B1NLw5pZX3WeDCN3H8qDCs9LHnzlgkZjKlYdBfwAcyw`

其实虽然看起来像乱码一样, 但是你去读取 redis 的时候, 拿到的值依然是正常的; 所以这个像乱码一样的效果, 我们可以不处理, 是没有任何问题的; 但是我们平时开发维修项目的时候, 阅读起来不太方便, 所以还是建议你处理一下;

1.4.3.5 前端浏览器存储 Token

sessionStorage 浏览器对象, 在 js 中可以直接使用; (会话存储) 从安全角色考虑, 建议使用 sessionStorage;
localStorage 浏览器对象, 在 js 中可以直接使用; (本地存储)

它们两者的区别:

sessionStorage 只在一个页面有效（有效范围很窄），换一个页面就失效了，就读不到你放的这个 token 数据了

localStorage 在整个浏览器都有效（有效范围很广），重启浏览器也有效，都能拿到你放的这个 token 数据；

```
window.sessionStorage.setItem("loginToken", response.data.info)
```

```
window.localStorage.setItem("loginToken", response.data.info)
```

1.4.3.5.1 创建 JwtVerifyFilter

```
@Component
public class TokenFilter extends OncePerRequestFilter {

    @Resource
    private RedisManager redisManager;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
    FilterChain filterChain) throws ServletException, IOException {
        String token = request.getHeader("Authorization"); //业界的默契
        // 校验 token
        if (!StringUtils.hasText(token)) { //null、“”、空格都不行
            ResponseUtils.writeJson(response, R.FAIL(CodeEnum.LOGIN_TOKEN_EMPTY));
            return;
        }
        // 校验 token
        if (!JwtUtils.verifyToken(token)) {
            ResponseUtils.writeJson(response, R.FAIL(CodeEnum.LOGIN_TOKEN_ILLEGAL));
            return;
        }
        // 校验 token
        String userJSON = JwtUtils.parseToken(token);
        TUser user = JSONUtil.toBean(userJSON, TUser.class);
        Integer userId = user.getId();
        String redisToken = (String)redisManager.getValue(String.valueOf(userId));
        if (!token.equals(redisToken)) {
            ResponseUtils.writeJson(response, R.FAIL(CodeEnum.LOGIN_TOKEN_ERR));
            return;
        }
    }
}
```

```
// 校验通过，
// 1、把用户信息放入 Spring Security 框架的上下文中（Context）
UsernamePasswordAuthenticationToken authenticationToken
    = new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
SecurityContextHolder.getContext().setAuthentication(authenticationToken);

// 放行，让下面的 Filter 继续执行
filterChain.doFilter(request, response);
}
```

1.4.3.5.2 去测试

- 1、使用 wangwu/123456 登录得到 token;
- 2、使用 [apifox 工具](#)（[html 代码](#)）把 token 放在请求头中请求各个接口;

1.4.3.5.3 测试后的问题解决

后端 springboot 项目重启，然后再执行以上测试的第 2 步（再访问/welcome 接口），我们发现，使用第 1 步生成的 token，依然可以直接不用登录就可以访问各个接口；

正常情况下，重启项目后，原来 web 项目的 session 会失效，那么 jwt 应该也要失效；

为什么？因为 JWT 无状态，重启项目后，jwt 并没有失效，依然可以访问后端的接口；

原因是，你重启后端 springboot 项目后，前端 sessionStorage 中 token 没有失效，后端 redis 中的 token 也没有失效

解决办法：

- 1、把 jwt 存入 redis 中并设置一个过期时间，到期后 jwt 自动失效；（30 分钟失效）
- 2、实现一个退出功能，用户点击退出登录，让 jwt 失效；（用户如果不点击退出）

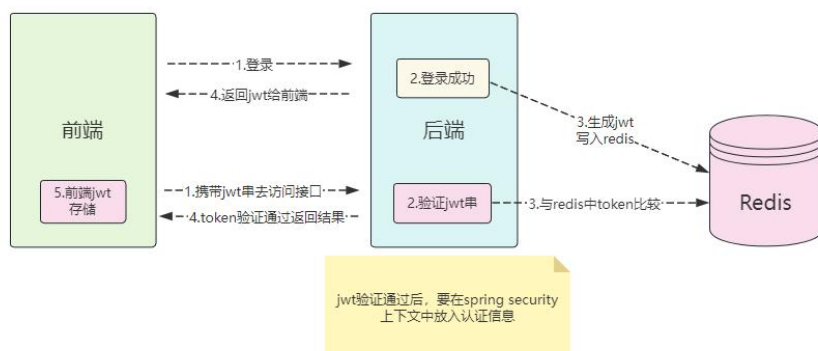
3、服务关闭/重启，删除 redis 的所有 jwt;

```
@Component
public class ShutdownListener implements ApplicationListener<ContextClosedEvent> {

    @Resource
    private RedisManager redisManager;

    @Override
    public void onApplicationEvent(ContextClosedEvent event) {
        // 这里写你想要在应用关闭时执行的业务逻辑
        System.out.println("Application is shutting down...");
        redisManager.removeAllToken();
    }
}
```

1.4.3.5.4 Spring Security +Jwt 前后端分离登录流程梳理



1、通过工具（apifox）或者 Vue（html）的 axios 发送请求访问登录接口：

http://localhost:8080/user/login, 传上账号密码参数;

2、spring security 框架处理/user/login 这个登录请求，具体处理是

UsernamePasswordAuthenticationFilter 类接收账号密码，然后调用 UserServiceImpl 中的方法：

- 3、loadUserByUsername(String username) --> 查询数据库，返回实现了 UserDetails 接口的 TUser 对象，然后回到 spring security 框架中验证 4 个状态值和比较密码，状态值都是 true，密码也匹配，那么就登录成功；
- 4、登录成功了就会调用登录成功的 AppAuthenticationSuccessHandler，该 handler 生成 jwt，然后 jwt 写入 redis/mysql，然后把 jwt 返回到前端；如果是登录失败就调用 AppAuthenticationFailureHandler，该 handler 就返回 R 失败的 json 信息对象；
- 5、前端拿到 jwt 后，把 jwt 要存储在前端（sessionStorage、localStorage），后续在请求后端的每一个接口时，都会在请求头 header 中带上这个 jwt；
- 6、后端接口接收到前端的请求时，首先都会被 jwt 的验证过滤器 JwtVerifyFilter 拦截，拦截里面会验证 jwt 是否合法（是否是空、有没有篡改，和 redis/mysql 是否相等），验证未通过就直接给前端返回一个 R 对象的 json，验证通过了，把 spring security 上下文中设置用户认证信息，表示该 jwt 的用户是登录过的，接下来就可以访问具体的后端 controller 接口了，接口里面执行具体的业务，然后 controller 接口返回 json 给前端，前端进行数据显示；
- 7、如果项目重启了，那么之前登录的 jwt 都要失效，在项目重启时，使用 spring 的事件监听，把之前登录的 jwt 全部从 redis/mysql 中删除；
- 8、补充：如果访问退出接口，那就是访问 /logout 接口，这个接口是 spring security 框架提供的，我们不需要写 controller，退出的具体操作逻辑是 spring security 自己实现的（内部把 spring security context 上下文的登录认证信息 Authentication 清除了），退出成功后会调用 AppLogoutSuccessHandler 这个 handler，在 handler 中我们要把 Redis 中的登录 token 删除，然后再返回一个 R 对象的 json 告诉前端退出成功就可以了；

1.5 SpringSecurity 前后端分离权限管理

无权限时，回调 AuthenticationSuccessHandler;