

Spring Security

Spring Security是spring家族中一个安全管理框架，相比其他安全框架Shiro,提供更丰富的功能，社区资源丰富
中大型项目使用Spring Security来作安全框架，小项目使用shiro;相比较Spring Security，shiro上手简单。

一般web应用需要进行认证和授权

认证：验证当前访问系统的是不是本系统的用户，并且要确认具体是哪个用户

授权：判断登录后的用户是否有权限进行某个操作

认证和授权是SpringSecurity作为安全框架核心功能。

快速入门

快速搭建一个简单springboot工程

设置父工程，添加依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.14</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>cn.edu.wfit</groupId>
  <artifactId>spring-security-demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-security-demo</name>
  <description>spring-security-demo</description>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>
```

```

        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>

```

创建启动类

```

@SpringBootApplication
public class SpringSeceurityDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringSeceurityDemoApplication.class, args);
    }

}

```

创建控制层controller

```
package cn.edu.wfit.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello(){
        return "hello";
    }
}
```

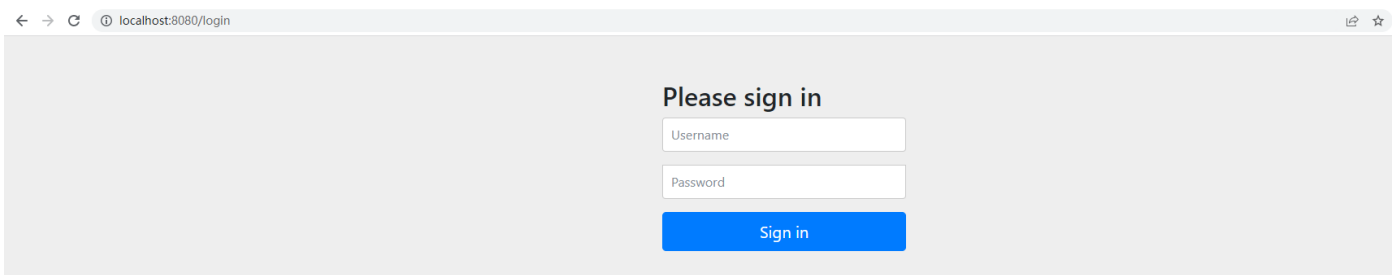


hello

引入spring security

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

经过测试，发现一般只要配置了SpringSecurity之后，即pom导入配置后，只要一访问控制器的接口，都会被拦截，自动跳转到SpringSecurity自定义的登录界面，界面如下：



认证

认证是我们网站的第一步,用户需要登录之后才能进入

Security自定义的账号是user,密码则是由控制台生成,输入账号和密码即可登录成功，并跳转到一开始输入要访问的页面

```

2023-08-24 06:18:10.299 WARN 9800 --- [main] .s.s.UserDetailsServiceAutoConfiguration :
Using generated security password: abdad86d-4e60-4313-b5ab-a3dc0ce282f1

This generated password is for development use only. Your security configuration must be updated before running your application in production.

2023-08-24 06:18:10.398 INFO 9800 --- [main] o.s.s.web.DefaultSecurityFilterChain : Will secure any request with [org.springframework.se
2023-08-24 06:18:10.438 INFO 9800 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context
2023-08-24 06:18:10.448 INFO 9800 --- [main] c.e.wfit.SpringSecurityDemoApplication : Started SpringSecurityDemoApplication in 1.831 seco
2023-08-24 06:18:19.752 INFO 9800 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherSer
2023-08-24 06:18:19.752 INFO 9800 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2023-08-24 06:18:19.753 INFO 9800 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms

```

基于内存验证

首先我们来看看最简单的基于内存的配置,也就是我们直接以代码的形式配置我们网站的用户和密码,配置方式非常简单,只需要在Security配置类中注册一个Bean即可:

```

@Configuration
@EnableWebSecurity // 开启WebSecurity相关功能
public class SecurityConfig {
    @Bean //UserDetailsService 就是获取用户信息的服务
    public UserDetailsService userDetailsService(){
        // 每一个UserDetails就代表一个用户信息,其中包含用户的用户名和密码以及角色
        UserDetails user=
        User.withDefaultPasswordEncoder().username("wfit").password("123").roles("USER").build(
        );
        return new InMemoryUserDetailsManager(user);
    }
    // 创建一个基于内存的用户信息管理器作为UserDetailsService
}

```

这样,我们的网站就成功用上了更加安全的SpringSecurity框架了。实际上这种方式为明文密码,这样存储密码并不安全。

我们应该使用加密算法将密码加密,最后将用户提供的密码以同样的方式加密后与密文进行比较。用户提供的密码属于隐私信息,如果直接明文存储并不好,而且如果数据库内容被窃取,那么所有用户的密码将全部泄漏,这绝对是不能容忍的结果,我们需要一种既能隐藏用户密码也能完成认证的机制,而Hash处理就是一种很好的解决方案,通过将用户的密码进行Hash值计算,得出来的结果一般是单向的,无法还原为原文,如果需要验证是否与此密码一致,那么需要以同样的方式加密再比较两个Hash值是否一致,这样就很好的保证了用户密码的安全性。

因此,我们在配置用户信息的时候,可以使用官方提供的BCrypt加密工具:

```

@Configuration
@EnableWebSecurity // 开启WebSecurity相关功能
public class SecurityConfig {
    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
    @Bean //UserDetailsService 就是获取用户信息的服务
    public UserDetailsService userDetailsService(PasswordEncoder encoder){
        // 每一个UserDetails就代表一个用户信息,其中包含用户的用户名和密码以及角色
    }
}

```

```

        UserDetails user=
        User.withUsername("wfit").password(encoder.encode("123")).roles("USER").build();
        System.out.println(encoder.encode("123"));
        return new InMemoryUserDetailsManager(user);
    }
}

```

这样，我们存储的密码就是更加安全的密码了：

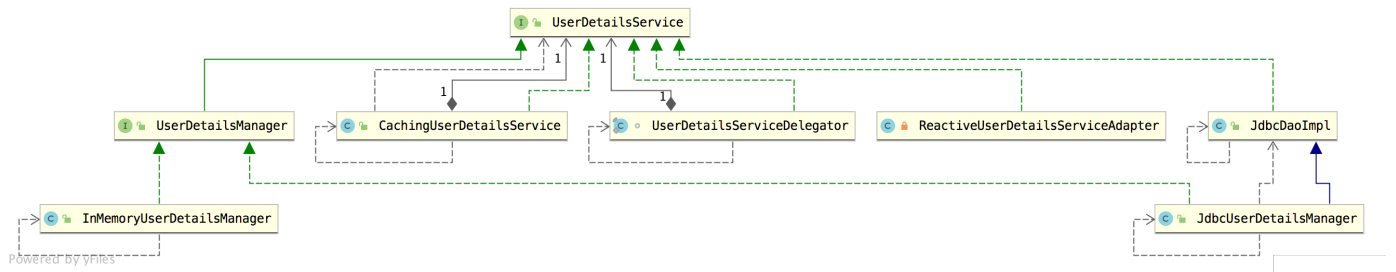
```

2023-08-24 09:12:08.744 INFO 1699 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 736 ms
$2a$10$HY.Y.ZLoSNaHAZ70Pm40sKehxr4W0putg0eLtF9t0vvbNDdm1VKy5C
2023-08-24 09:12:09.059 INFO 1699 --- [main] o.s.s.web.DefaultSecurityFilterChain : Will secure any request with [org.springframework.security.web.
2023-08-24 09:12:09.303 INFO 1699 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''

```

基于数据库认证

前面我们已经实现直接认证的方式，但是实际项目中往往都是将用户信息存储在数据库中的，那么如何将其连接到数据库，通过查询数据库中的用户信息来进行用户登录呢？



可以看到，在几个能直接使用的实现类中，除了 `InMemoryUserDetailsManager` 之外，还有一个 `JdbcUserDetailsManager`，使用 `JdbcUserDetailsManager` 可以让我们通过 JDBC 的方式将数据库和 Spring Security 连接起来。

`JdbcUserDetailsManager` 自己提供了一个数据库模型,保存位置如下：

`org.springframework.security/core/userdetails/jdbc/users.ddl`

存储的脚本内容如下：

```

create table users(username varchar(50) not null primary key,password varchar(500) not
null,enabled boolean not null);
create table authorities (username varchar(50) not null,authority varchar(50) not
null,constraint fk_authorities_users foreign key(username) references users(username));
create unique index ix_auth_username on authorities (username,authority);

```

执行完 SQL 脚本后，我们可以看到一共创建了两张表：users 和 authorities

- users 表中保存用户的基本信息，包括用户名、用户密码以及账户是否可用。
- authorities 中保存了用户的角色。
- authorities 和 users 通过 username 关联起来。

配置完成后，接下来，我们将上篇文章中通过 `InMemoryUserDetailsManager` 提供的用户数据用 `JdbcUserDetailsManager` 代替掉

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.14</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>cn.edu.wfit</groupId>
  <artifactId>securitydemo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>securitydemo</name>
  <description>securitydemo</description>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.baomidou</groupId>
      <artifactId>mybatis-plus-boot-starter</artifactId>
      <version>3.5.3.1</version>
    </dependency>
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid-spring-boot-starter</artifactId>
      <version>1.2.18</version>
    </dependency>
    <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
```

```

        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>

```

application.yml

```

spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mybatisplus_db?serverTimezone=UTC
    username: root
    password: root
  # mybatis-plus日志控制台输出
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    banner: off # 关闭mybatisplus启动图标

```

SecurityConfig

```

@Configuration
@EnableWebSecurity // 开启WebSecurity相关功能
public class SecurityConfig {
    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
    @Bean //UserDetailsService 就是获取用户信息的服务
    public UserDetailsService userDetailsService(DataSource dataSource, PasswordEncoder encoder){
        JdbcUserDetailsManager manager = new JdbcUserDetailsManager(dataSource);
        // 仅首次启动的时候创建一个新的用户用于测试
        if(!manager.userExists("wfit")){
            manager.createUser(User.withUsername("wfit").password(encoder.encode("1234")).roles("USER").build());
        }
        return manager;
    }
}

```

这段配置的含义如下：

- 首先构建一个 JdbcUserDetailsManager 实例。
- 给 JdbcUserDetailsManager 实例添加一个 DataSource 对象。
- 调用 userExists 方法判断用户是否存在，如果不存在，就创建一个新的用户出来（因为每次项目启动时这段代码都会执行，所以加一个判断，避免重复创建用户）。
- 用户的创建方法和我们之前 InMemoryUserDetailsManager 中的创建方法基本一致。

这里的 createUser 或者 userExists 方法其实都是调用写好的 SQL 去判断的

这种方式虽然能够直接使用数据库，但是存在一定的局限性，只适合快速搭建Demo使用，不适合实际生产环境，接下来我们看看如何实现自定义验证：

自定义验证

我们的数据库一般不会像SpringSecurity默认的那样进行设计，都是采用自定义的表结构，上面的两种方式就很难进行验证了，此时我们需要编写自定义的验证，来应对各种变化的情况。

既然需要自定义，那么我们就需要自行实现UserDetailsService接口

```
@Service
public class AuthorizeService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return null;
    }
}
```

现在我们需要去实现这个loadUserByUsername方法，表示在验证的时候通过自定义的方式，根据给定的用户名查询用户，并封装为UserDetails对象返回，然后由SpringSecurity将我们返回的对象与用户登录的信息进行核验，基本流程实际上跟之前一样，只是现在由我们自己来提供用户查询

UsersMapper

```
public interface UsersMapper extends BaseMapper<Users> {
    @Select("select * from users where username= #{username}")
    Users findUsersByName(String username);
}
```

Users 实体类

```
@Data
public class Users {
    private String username;
    private String password;
    private int enabled;
}
```

AuthorizeService

```

@Service
public class AuthorizeService implements UserDetailsService {
    @Autowired
    private UsersMapper usersMapper;
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Users account = usersMapper.findUsersByName(username);
        if(account==null){
            throw new UsernameNotFoundException("没有该用户");
        }
        return
User.withUsername(account.getUsername()).password(account.getPassword()).roles("USER").
build();
    }
}

```

SecurityConfig

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Configuration
@EnableWebSecurity // 开启WebSecurity相关功能
public class SecurityConfig {
    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
    /*    @Bean //UserDetailsService 就是获取用户信息的服务
    public UserDetailsService userDetailsService(DataSource dataSource,PasswordEncoder
encoder){
        JdbcUserDetailsManager manager = new JdbcUserDetailsManager(dataSource);
        // 仅首次启动的时候创建一个新的用户用于测试
    }
    */
}

```

```

        if(!manager.userExists("wfit")){

            manager.createUser(User.withUsername("wfit").password(encoder.encode("1234")).roles("U
SER").build());
        }
        return manager;
    }*/

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
        return http
            // 以下是验证请求拦截和放行配置
            .authorizeHttpRequests(auth -> {
                auth.antMatchers("/test").permitAll(); // 不拦截指定的页面
                auth.anyRequest().authenticated(); // 将所有请求全部拦截，一律需要验证
            })
            // 以下是表单登录相关配置
            .formLogin(conf->{
                conf.successHandler(this::onAuthenticationSuccess);// 登录成功执行的方法

                conf.failureHandler(this::onAuthenticationFailure);// 登录失败执行的方法

                conf.loginProcessingUrl("/dologin");// 表单提交的地址，可以自定义
                conf.permitAll();//将登录相关的地址放行，否则未登录的用户连登录界面都进不去
                //用户名和密码的表单字段名称，不过默认就是这个，可以不配置，除非有特殊需求
                conf.usernameParameter("username");
                conf.passwordParameter("password");
            })
            .csrf(AbstractHttpConfigurer::disable)
            .build();
    }

    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse
response, Authentication authentication) throws IOException, ServletException {
        response.setContentType("application/json;charset=UTF-8");
        response.getWriter().write("登录成功");
    }

    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse
response, AuthenticationException exception) throws IOException, ServletException {
        response.setContentType("application/json;charset=UTF-8");
        response.getWriter().write("登录失败");
    }
}

```

不要忘记扫描mapper包

```

@SpringBootApplication
@MapperScan("cn.edu.wfit.mapper")
public class SpringSeceurityDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringSeceurityDemoApplication.class, args);
    }

}

```

这样我们就通过自定义的方式实现了数据信息查询，并完成用户登录操作。

授权

用户登录后，可能会根据用户当前身份进行角色划分，比如我们最常用的QQ，一个QQ群里面，有群主，管理员和普通群成员三种角色，其中群主具有最高权限，群主可以管理整个群的任何板块，并具有解散和升级群的资格，而管理员只有群主的一部分权限，只能用于日常管理，普通群成员则只能进行最基本的聊天操作。

对于我们来说，用户的一个操作实际上就是在访问我们提供的接口，比如登录，就需要调用/login接口，退出登录就需要调用/logout接口，因此，从我们开发者的角度来说，决定用户能否使用某个功能，只需要决定用户是否能够访问对应的接口即可。

我们可以大致向下面这样进行划分：

- 群组：/login /logout /chat /edit /delete /upgrade
- 管理员：/login /logout /chat /edit
- 普通群成员：/login /logout /chat

也就是说，我们需要做的就是指定哪些请求可以由哪些用户发起。

SpringSecurity为我们提供了两种授权方式：

- 基于权限的授权：只要拥有某权限的用户，就可以访问某个路径。
- 基于角色的授权：根据用户属于哪个角色来决定是否可以访问某个路径。

两者只是概念上的不同，实际上使用起来效果差不多。这里我们先来演示以角色方式来进行授权。

基于角色授权

现在我们希望创建两个角色，普通用户和管理员，普通用户只能访问index页面，而管理员可以访问任何页面。

首先我们需要对数据库中的角色表进行一些修改，添加一个用户角色字段，并创建一个新的用户，Test用户的角色为user，而Admin用户的角色为admin

接着我们需要配置SpringSecurity，决定哪些角色可以访问哪些页面：

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
    return http
        // 以下是验证请求拦截和放行配置
        .authorizeHttpRequests(auth -> {
            auth.antMatchers("/test").permitAll(); // 不拦截指定的页面
        })
}

```


方法

```
auth.antMatchers("/index").hasAnyRole("user", "admin");
auth.anyRequest().hasRole("admin");
// auth.anyRequest().authenticated(); // 将所有请求全部拦截，一律需要验证
})
// 以下是表单登录相关配置
.formLogin(conf->{
    conf.successHandler(this::onAuthenticationSuccess); // 登录成功执行的方法

    conf.failureHandler(this::onAuthenticationFailure); // 登录失败执行的方法

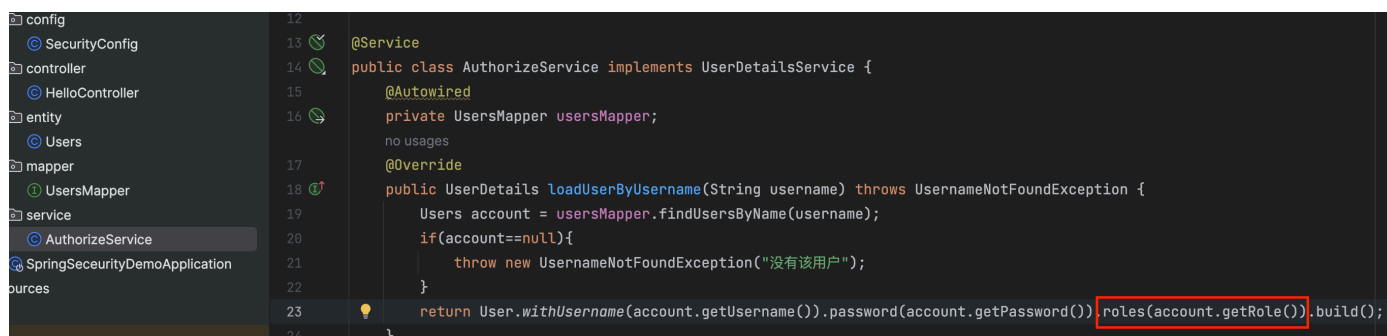
    conf.loginProcessingUrl("/dologin"); // 表单提交的地址，可以自定义
    conf.permitAll(); // 将登录相关的地址放行，否则未登录的用户连登录界面都进不去
    // 用户名和密码的表单字段名称，不过默认就是这个，可以不配置，除非有特殊需求
    conf.usernameParameter("username");
    conf.passwordParameter("password");
})
.csrf(AbstractHttpConfigurer::disable)
.build();
}
```

实体类



```
3 import lombok.Data;
4 import lombok.experimental.PackagePrivate;
5
6 5 usages
7 @Data
8 public class Users {
9     private String username;
10    private String password;
11    private int enabled;
12    private String role;
13 }
```

AuthorizeService



```
12
13 @Service
14 public class AuthorizeService implements UserDetailsService {
15     @Autowired
16     private UsersMapper usersMapper;
17     no usages
18     @Override
19     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
20         Users account = usersMapper.findUsersByName(username);
21         if(account==null){
22             throw new UsernameNotFoundException("没有该用户");
23         }
24         return User.withUsername(account.getUsername()).password(account.getPassword()).roles(account.getRole()).build();
25     }
26 }
```

使用注解权限判断

除了直接配置以外，我们还可以以注解的形式直接配置，首先需要在配置类上开启：

```
@Configuration
@EnableWebSecurity // 开启WebSecurity相关功能
@EnableMethodSecurity // 开启方法安全校验
public class SecurityConfig {
    ...
}
```

现在我们就可以在我们想要进行权限校验的方法上添加注解了：

```
@RestController
public class HelloController {
    @PreAuthorize("hasRole('user')") // 直接使用hasRole方法判断是否包含某个角色
    @GetMapping("/index")
    public String index(){
        return "index";
    }
}
```

通过添加 @PreAuthorize注解，在执行之前判断权限，如果没有对应的权限或是对应的角色，将无法访问页面。

除了Controller以外，只要是由Spring管理的Bean都可以使用注解形式来控制权限，我们可以在任意方法上添加这个注解，只要不具备指定的访问权限，那么就无法执行方法并且会返回403

项目实战中前后端分离项目

SecurityConfig我们一般会这样配置

```
@Configuration
@EnableWebSecurity // 开启WebSecurity相关功能
@EnableMethodSecurity // 开启方法安全校验
public class SecurityConfig {
    @Autowired
    private JwtSecurityFilter jwtSecurityFilter;
    @Autowired
    private JwtService jwtService;
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
        return http
            // 以下是验证请求拦截和放行配置
            .authorizeHttpRequests(auth -> {
                auth.antMatchers("/test").permitAll(); // 不拦截指定的页面
                auth.antMatchers("/index").hasAnyRole("user", "admin");
                // auth.anyRequest().hasRole("admin");
                auth.anyRequest().authenticated(); // 将所有请求全部拦截，一律需要验证
            })
    }
```

法
法

```
// 以下是表单登录相关配置
.formLogin(conf->{
    conf.successHandler(this::onAuthenticationSuccess); // 登录成功执行的方法

    conf.failureHandler(this::onAuthenticationFailure); // 登录失败执行的方法

    conf.loginProcessingUrl("/dologin"); // 表单提交的地址, 可以自定义
    conf.permitAll(); // 将登录相关的地址放行, 否则未登录的用户连登录界面都进不去
    // 用户名和密码的表单字段名称, 不过默认就是这个, 可以不配置, 除非有特殊需求
    conf.usernameParameter("username");
    conf.passwordParameter("password");
})
.sessionManagement(config->config.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // 禁用session
.headers(conf->{
    conf.frameOptions().disable();
    conf.cacheControl().disable(); // 禁用缓存
})
.exceptionHandling(config->config
    // 已登录但没有权限的情况下访问需要角色权限的接口
    .accessDeniedHandler(this::onAccessDeniedHandler)
    // 没有登录的情况下访问需要登录的接口
    .authenticationEntryPoint(this::commence))
.cors(config->config.configurationSource(this.corsConfigurationSource()))
.csrf(AbstractHttpConfigurer::disable) // 禁用csrf
.addFilterBefore(jwtSecurityFilter,
UsernamePasswordAuthenticationFilter.class)
.build();
}

public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration cors = new CorsConfiguration();
    cors.addAllowedOrigin("*");
    cors.addAllowedHeader("*");
    cors.addAllowedMethod("*");
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", cors);
    return source;
}

// 登录失败
void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse
response, AuthenticationException exception) throws IOException {
    response.setContentType("application/json;charset=UTF-8");
    exception.printStackTrace();
    String str = JSON.toJSONString(JsonResult.failure(400, exception.getMessage()));
    response.getWriter().write(str);
}

// 登录成功
```

```

    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse
response, Authentication authentication) throws IOException, ServletException {
        LoginUser principal = (LoginUser)authentication.getPrincipal();
        Long id = principal.getUser().getId();
        String token = jwtService.generateToken("wfit:" + id);
        response.setContentType("application/json;charset=UTF-8");
        String str = JSON.toJSONString(JsonResult.ok(token));
        response.getWriter().write(str);
    }

    public void onAccessDeniedHandler(HttpServletRequest request, HttpServletResponse
response, AccessDeniedException accessDeniedException) throws IOException {
        response.setContentType("application/json;charset=UTF-8");
        accessDeniedException.printStackTrace();
        String str =
JSON.toJSONString(JsonResult.failure(403,accessDeniedException.getMessage()));
        response.getWriter().write(str);
    }

    public void commence(HttpServletRequest request, HttpServletResponse response,
AuthenticationException authException) throws IOException {
        response.setContentType("application/json;charset=UTF-8");
        //      authException.printStackTrace();
        String str =
JSON.toJSONString(JsonResult.failure(400,authException.getMessage()));
        response.getWriter().write(str);
    }
    @Bean
    public PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}

```

LoginUser

```

@Data
public class LoginUser implements UserDetails , Serializable {

    /**
     * 自定义用户对象
     */
    private User user;
    private String[] permission;
    public LoginUser(User user){
        this.user = user;
    }
    public LoginUser(User user,String[] Permission){
        this.user = user;
        this.permission = Permission;
    }
}

```



```
}  
  
public LoginUser(){}  
  
/*  
 * 权限信息  
 * */  
@Override  
@JsonIgnore  
public Collection<? extends GrantedAuthority> getAuthorities() {  
    return AuthorityUtils.createAuthorityList(this.getPermission());  
}  
  
/*  
 * 密码  
 * */  
@Override  
@JsonIgnore  
public String getPassword() {  
    return user.getPassword();  
}  
  
/*  
 * 用户名  
 * */  
@Override  
@JsonIgnore  
public String getUsername() {  
    return user.getName();  
}  
  
/*  
 * 表示判断账户是否过期  
 * */  
@Override  
@JsonIgnore  
public boolean isAccountNonExpired() {  
    return true;  
}  
  
/*  
 * 表示判断账户是否被锁定  
 * */  
@Override  
@JsonIgnore  
public boolean isAccountNonLocked() {  
    return true;  
}
```

```

    /*
     * 表示凭证{密码}是否过期
     */
    @Override
    @JsonIgnore
    public boolean isCredentialsNonExpired() {
        return true;
    }

    /*
     * 是否可用
     */
    @Override
    @JsonIgnore
    public boolean isEnabled() {
        return true;
    }
}

```

JsonResult

```

@Data
public class JsonResult implements Serializable {

    // 响应业务状态
    private Integer code;

    // 响应消息
    private String msg;

    // 响应中的数据
    private Object data;

    public static JsonResult build(Integer code, String msg, Object data) {
        return new JsonResult(code, msg, data);
    }

    public static JsonResult failure(Integer code, String msg) {
        return new JsonResult(code, msg, null);
    }

    public static JsonResult ok(Object data) {
        return new JsonResult(data);
    }

    public static JsonResult ok() {
        return new JsonResult(null);
    }

    public JsonResult() {

```

```

    }

    public static JsonResult build(Integer code, String msg) {
        return new JsonResult(code, msg, null);
    }

    public JsonResult(Integer code, String msg, Object data) {
        this.code = code;
        this.msg = msg;
        this.data = data;
    }

    public JsonResult(Object data) {
        this.code = 200;
        this.msg = "OK";
        this.data = data;
    }
}

```

JwtService

```

@Service
public class JwtService {

    @Value("${application.security.jwt.secret-key}")
    private String secretKey;
    @Value("${application.security.jwt.expiration}")
    private long jwtExpiration;
    @Value("${application.security.jwt.refresh-token.expiration}")
    private long refreshExpiration;

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    public String generateToken(String userDetails) {
        return generateToken(new HashMap<>(), userDetails);
    }

    public String generateToken(
        Map<String, Object> extraClaims,
        String userDetails
    ) {

```

```

        return buildToken(extraClaims, userDetails, jwtExpiration);
    }

    public String generateRefreshToken(
        String userDetails
    ) {
        return buildToken(new HashMap<>(), userDetails, refreshExpiration);
    }

    private String buildToken(
        Map<String, Object> extraClaims,
        String userDetails,
        long expiration
    ) {
        return Jwts
            .builder()
            .setClaims(extraClaims)
            // .setSubject(userDetails.getUsername())
            .setSubject(userDetails)
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(getSignInKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    public boolean isTokenValid(String token, String userDetails) {
        final String username = extractUsername(token);
        // return (username.equals(userDetails.getUsername())) &&
        !isTokenExpired(token);
        return (username.equals(userDetails)) && !isTokenExpired(token);
    }

    public boolean isTokenValid(String token) {
        return !isTokenExpired(token);
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    private Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    private Claims extractAllClaims(String token) {
        return Jwts
            .parserBuilder()
            .setSigningKey(getSignInKey())
            .build()
            .parseClaimsJws(token)

```

```

        .getBody();
    }

    private Key getSignInKey() {
        byte[] keyBytes = Decoders.BASE64.decode(secretKey);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}

```

JwtSecurityFilter

```

@Component
public class JwtSecurityFilter extends OncePerRequestFilter {
    @Autowired
    private JwtService jwtService;
    @Autowired
    private RedisCache redisCache;
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException, IOException {
        if (request.getServletPath().contains("/users/login")) {
            filterChain.doFilter(request, response);
            return;
        }
        // 获取请求头中的token
        String authHeader = request.getHeader("Authorization");
        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }
        String jwt = authHeader.substring(7);
        String userId = jwtService.extractUsername(jwt);

        LoginUser loginUser = redisCache.getCacheObject(userId);
        //      System.out.println(loginUser);
        if (loginUser != null) {
            //      List<GrantedAuthority> authorities =
AuthorityUtils.createAuthorityList(loginUser.getPermission());
            // 存入 SecurityContextHolder 参数一: 用户信息 参数二: 参数三: 权限认证
            UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(loginUser, null, loginUser.getAuthorities());
            authToken.setDetails( new
WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
        filterChain.doFilter(request, response);
    }
}

```

