授权码模式：

资源服务器认证方式：

认证服务器暴露一个/auth/token\_key接口，提供公钥地址

Apm作为资源服务器在启动的时候取得公钥，在通讯的过程中，资源服务器用私钥进行加密，apm通过公钥解密。

APM启动时会调用/oauth/token\_key接口，获取token加密算法和签名密钥，拿到这些信息后初始化tokenstore, 完成启动后客户端带token的请求到APM后，APM会通过过滤器(AccessTokenFilter)进行token校验

[http://127.0.0.1:38067/oauth/authorize?client\_id=mecm-fe&response\_type=code&redirect\_uri=http://127.0.0.1:38092/login&scope=all](http://127.0.0.1:38067/oauth/authorize?client_id=mecm-fe&response_type=code&redirect_uri=http:/127.0.0.1:38092&scope=all)

认证流程

1. 获取认证类型为code,如果不是该类型会报异常Unsupported response types
2. 判断ClientId 是否为空，为空返回A client id must be provided异常
3. 根据principal判断oauth是否已经是Authenticated状态，如果是已认证状态，会跳到下一步，获取ClientDetails信息，这里走的是内存模式
4. 获取redirect\_uri参数

OAuth2.0协议在Spring Security中的应用

自带一个login页面 /login

一、自定义登录逻辑

1. 要实现自己的登录验证逻辑，需要实现UserDetailsService接口
2. Spring Security 自己实现了User类，实现了该接口。
3. Spring Security 提供了PasswordEncoder 接口，如果要自定义密码验证的逻辑需要重新实现该接口，官方推荐使用BCrtyptPasswordEncoder
4. 自己实现UserDetailsServiceImpl implements UserDetailsService{

内部重写loadUserByUsername方法();

|  |
| --- |
| if(StringUtils.isEmpty(username) || !"admin".equals(username)){  throw new UsernameNotFoundException("User Not Exists!");  }  //查询数据库，将查询到的权限信息放入user中然后返回  String password = passwordEncoder.encode("123");  return new User("admin",password,  AuthorityUtils.commaSeparatedStringToAuthorityList("ADMIN,NORMAL")); |

}

内部逻辑：

1. 查询数据库是否存在该用户,不存在抛出UserNotFound 异常
2. 存在用户，查询解析用户名中的密码（加密后的密码）

以上逻辑会修改默认的登录逻辑，从而实现用户及权限信息自定义。

二、自定义登录页面

1. 要使用自己的登录页面需要继承 WebSecurityConfigurerAdapter类
2. 重写里面的configure(http:HttpSecurity) 方法。

http.formLogin().loginPage(“./login.html”)

会自动加载resources.static.login.html，作为自定义的登录页面进行登录。

1. 配置了2中的loginPage后，认证的逻辑会失效，需要添加如下配置。
2. http.authroizeRequests().anyRequest().authenticated();

也就是添加了一个拦截器，所有的请求都需要认证。

1. 需要将login 这个路径的请求剔除认证，不然会死循环一直反复跳转到登录界面
2. http.antMathcers(“/login.html”).permitAll().authorizeRequsts().anyRequest().anthenticated();
3. 此时登录的逻辑并不会执行到上面重写的UserDetailsServiceImpl.loadUserByUsername() 方法上去，需要在配置中加上登录的逻辑认证请求的连接，在配置了以后才会生效。
4. http.formLogin().loginProcessingUrl(“/login”).loginPage(“/login.html”)
5. 此处配置的登录处理的url地址为前端点击登录按钮后请求的后端接口，也就是后端登录逻辑实际的验证接口。
6. 因为实现了UserDetailsService 接口以后，默认的/login 请求会实际执行UserDetailsServiceImpl 中的代码逻辑。
7. 还需要添加关闭csrf的配置
8. http.csrf().disable();
9. 如果要配置登录成功后跳转的页面，可以配置如下
10. http.formLogin().loginProcessingUrl(“/login”).loginPage(“/login.html”).successForwardUrl(“/xxx.html”);
11. 该路径可以自定义，但是该请求必须要是一个POST请求，可以写一个controller 进行跳转。
12. 同理可以添加错误的跳转逻辑，及页面，这里需要注意，失败跳转的页面会被拦截器拦截，需要修改配置，将失败错误的路径也放开拦截。
13. **这里因为需要完全按照security中的实现，所以登录页面中，登录提交的方式必须要是POST请求，同时，登录传递的参数也必须得是username 和 password。否则会失败！**
14. 上面具体的实现逻辑是在UsernamePasswordAuthenticationFilter 过滤器中实现。
15. 这里可以通过配置来修改 http.formLogin().usernameParameter(“自定义的用户变量”).passwordParameter(“自定义的密码变量”)
16. 自定义跳转页面
17. 由于默认的successForwardUrl 需要用POST做一个跳转，所以如果是前后端分离的项目，就没办法做页面的跳转，此时需要修改配置来做支持。
18. 目前有一个ForwardAuthenticationSuccessHandler 的处理器，用来处理登录成功后的业务逻辑。
19. 它实现了security中的AuthenticationSuccessHandler, 要想自定义跳转，需要我们自己动手实现这个handler，来做跳转，之后就可以配置了。
20. SuccessRedirectAuthenticationHandler implements AuthenticationSuccessHandler{

重写onAuthenticationSuccess(request,response,authentication){

Url = 要跳转的url;

Response.sendRedirect(url);

}

}

1. 去掉successForwardUrl 配置，改成

successHandler(new SuccessRedirectAuthenticationHandler(“http://xxxxx.com/home”)

1. antMatchers语法
2. antMatchers可以匹配多个url，做允许访问或者拒绝。
3. URL匹配规则如下：
   1. ？：匹配一个字符
   2. \* ：匹配0个或多个字符
   3. \*\* ：匹配0个或多个目录
4. 例如：
   1. .antMatchers(“/js/\*\*”).permitAll()
   2. .antMatchers(“/js/\*.js”).permitAll()
5. mvcMatchers语法
6. .mvcMatchers(“/demo”).servletPath(“/xxx”).permitAll()

通过这个配置可以配置servlet前缀的所有请求

对应配置中spring.mvc.servlet.path=/xxx

1. 内置控制访问方法
   1. permitAll、denyAll、anoymous、authenticated、fullyAuthenticated、rememberMe
2. 角色的权限判断
   1. hasAuthority(“admin”)
   2. 判断用户是否拥有admin权限
   3. 当一个页面或者路径需要配置某个权限才能访问时，添加配置
   4. .antMatchers(“/test.html”).hasAuthorized(“admin”).anyRequest().anthenticated();
   5. hasAnyAuthority(“admin”,”test”).anyRequest().anthenticated();
   6. 同时支持以注解的形式进行配置

|  |
| --- |
| @PreAuthorize("hasRole('MECM\_TENANT') || hasRole('MECM\_ADMIN')" || hasAuthority(“ADMIN”)) |

* 1. 角色写法

.antMatchers(“/test.html”).hasRole(“ADMIN”).anyRequest().anthenticated();

* 1. 可以**指定IP地址进行权限**控制：
  2. .antMatchers(“/test.html”).hasIPAddress(“127.0.0.1”).anyRequest().anthenticated();

1. 对403页面进行自定义设置
   1. Security对403的处理设置了一个AccessDeniedHandler
   2. 想要自定义可以直接冲洗定义这个接口即可
   3. 在handler文件夹中新建 CustomAccessDeniedHandler类实现AccessDeniedHandler接口
   4. 实现其中的handle方法

|  |
| --- |
| response.setStatus(HttpServletResponse.SC\_FORBIDDEN);  response.setHeader("Content-Type","application/json;charset=utf-8");  PrintWriter writer = response.getWriter();  writer.write("{\"status\":\"Access Denied!\",\"msg\":\"请求被拒绝\"}");  writer.flush();  writer.close(); |

* 1. 然后设置exceptionHandler 即可
  2. http.exceptionHandling().accessDeniedHandler(new CustomAccessDeniedHandler());

1. permitAll() rememberMe、denyAll等控制方法的底层都是实现了access 方法
2. 通过写自定义方法，设置对某URL的权限
   1. 自定义一个业务方法，将request中的URI与权限池进行校验，**此处取巧**

|  |
| --- |
| public class RequestWithUrlServiceImpl implements RequestWithUrlService {  @Override  public boolean accessRequstUrl(HttpServletRequest request, Authentication authentication) {  Object obj = authentication.getPrincipal();  if (obj instanceof UserDetails){  UserDetails user = (UserDetails) obj;  Collection<? extends GrantedAuthority> authorities = user.getAuthorities();  return authorities.contains(request.getRequestURL());  }  return false;  }  } |

* 1. 在配置中添加对此类的配置

|  |
| --- |
| .anyRequest()  .access("@requestWithUrlServiceImpl.accessRequstUrl(request,authentication)"); |

最后需要在自定义的UserDetailsServiceImpl.loadUserByUsername中加入该URL的配置

|  |
| --- |
| return new User("admin",password,AuthorityUtils.commaSeparatedStringToAuthorityList("ADMIN,NORMAL,/security/v1/service")); |

1. 注解的方式在方法上卡权限
2. @Secured(“ROLE\_ADMIN”)
3. 该注解可以在类、方法上使用
4. 可以使要访问的接口需要满足指定的角色才可以访问。
5. 在使用该注解之前需要在启动类上开启该注解

@EnableGlobalMethodSecurity(securedEnabled = true)

1. @PreAuthorize / @PostAuthorize
2. 这两个注解都是方法、类级别的注解
3. 前者是在进入方法之前进行判断，后者是在方法运行完之后进行权限的判断
4. 在使用前需要在启动类上开启该注解

@EnableGlobalMethodSecurity(prePostEnabled= true)

1. 允许ROLE\_ 开头
2. @PreAuthorize(“hasRole(‘ROLE\_abc’)”)
3. REMEMBER ME 功能
   1. 前端代码里传值必须为remember-me，如果要用别的需要在配置中进行修改
   2. 因为勾选后程序会自动帮助我们登录，所以会记住我们的用户名和密码生成的token
   3. 同时需要缓存到数据库中，也可以在内存中
   4. 需要添加mysql 及 mybatis的依赖项

|  |
| --- |
| <!-- Mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.5.6</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.28</version>  </dependency> |