1.简单Demo

1.1 引入依赖

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

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

1.2 编写测试controller

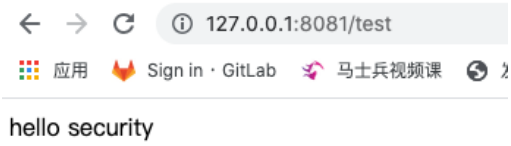
```java
@RestController
public class Democontroller {

   @RequestMapping("/test")
    public String demo(){
       return "hello security";
   }
}
```

1.3 验证

密码放在启动的目录中，默认用户名user

```
Using generated security password: fb053f82-e794-43f8-9253-6d539581f888
```

登陆结果：



hello security

2.用户认证登陆

2.1 设置登录的用户名密码

2.1.1 配置文件设置

```yaml
spring:
  security:
    user:
      name: user
      password: user
```

2.1.2 设置配置类

```java
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //对密码加密
        PasswordEncoder passwordEncoder =new BCryptPasswordEncoder();
        String encode = passwordEncoder.encode("123");
        auth.inMemoryAuthentication().withUser("user").password(encode).roles("admin");
    }

    @Bean
```

```
    //创建实现类，否则会报There is no PasswordEncoder mapped for the id "null"
    PasswordEncoder init(){
        return new BCryptPasswordEncoder();
    }
}
```

2.1.3 实现自定义类

如果在配置文件配置没有找到或者没有实现配置类，就会去查找UserDetailService接口

1.创建配置类，设置使用哪个userDetailService实现类

```
@Configuration
public class CustomSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private MyUserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        //设置用到的UserDetailsService实现类
        auth.userDetailsService(userDetailsService).passwordEncoder(init());
    }

    @Bean
    //创建实现类，否则会报There is no PasswordEncoder mapped for the id "null"
    PasswordEncoder init(){
        return new BCryptPasswordEncoder();
    }
}
```

2.编写实现类，返回user对象，user对象有用户名密码和操作权限

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {
        //设置权限
        List<GrantedAuthority> grantedAuthorities = AuthorityUtils.commaSeparatedStringToA
        //返回User
        return new User("user",new BCryptPasswordEncoder().encode("456"),grantedAuthoritie
    }
}
```

2.2 通过数据库完成验证登陆

2.2.1 引入依赖

```
<!-- 通过数据库实现认证登陆-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <!-- 这里必须增加版本号，否则引入不进来-->
    <version>3.0.5</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

2.2.2 创建数据库和表

创建数据库spring_security_demo，创建表users

```
create table users(
id bigint primary key auto_increment,
username varchar(20) unique not null, password varchar(100)
);
insert into users values(1,'user1','123');
insert into users values(2,'user2','456');
```

2.2.3 创建表对应的实体类

```
@Data
public class Users {
    private Integer id;
    private String username;
    private String password;
}
```

2.2.4 增加mapper

```
@Repository
public interface UsersMapper extends BaseMapper<Users> {
}
```

2.2.5 实现数据库逻辑

```
@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private UsersMapper usersMapper;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExceptio
        //调用userMapper的方法，根据用户名查询数据库
        QueryWrapper<Users>  wrapper  = new  QueryWrapper();
        wrapper.eq("username",username);
         Users user = usersMapper.selectOne(wrapper);
         if (user == null){
             throw new UsernameNotFoundException("用户找不到");
         }
         List<GrantedAuthority> grantedAuthorities = AuthorityUtils.commaSeparatedStringTo
         return new User(username,new BCryptPasswordEncoder().encode(user.getPassword()),g
    }
}
```

2.2.6 增加mapper扫描

```
@SpringBootApplication
@MapperScan("com.dudu.security.mapper")
```

```
public class SecurityApplication {

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

}
```

2.2.7 与数据库连接

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/spring_security_demo
    username: root
    password: root
```

3. 设置自定义页面

3.1 设置配置类

```
@Configuration
public class CustomSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private MyUserDetailsService userDetailsService;

    @Override
    protected void configure(HttpSecurity security) throws Exception {
       security.formLogin()
               //自定义用户登陆页面,需要认证的页面都会跳转到这个页面
                .loginPage("/login.html")
                //登陆页面表单提交到什么地址，这个地址不认证
                .loginProcessingUrl("/user/login")
                //登陆成功，跳转路径
                .defaultSuccessUrl("/sucess").permitAll()
                //设置哪些路径不需要认证可以直接访问
                .and().authorizeRequests().antMatchers("/","/test/hello","user/login").per
                 //其他任何请求都要过验证
                .anyRequest().authenticated()
                //关闭csrf防护
                .and().csrf().disable();
    }
}
```

3.2 设置controller

```
@RestController
public class Democontroller {
    @Autowired
    private MyUserDetailsService service;

   @RequestMapping("/sucess")
    public String sucess(){
       return "sucess";
```

```
    }
    @RequestMapping("/user/login")
    public void login(Users users){
        service.loadUserByUsername(users.getUsername());
    }
    @RequestMapping("/test/hello")
    public String hello(){
        return "hello";
    }
}
```

3.3 设置登陆页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
   <form action="/user/login" method="post">
       <!-- name必须是username和password，否则spring security得不到用户名和密码 -->
        用户名：<input type="text" name="username"><br/>
        密码：<input type="text" name="password"><br/>
        <input type="submit" value="login">
    </form>
</body>
</html>
```

4. 基于角色或权限的访问控制

4.1 基于权限访问控制：hasAuthority方法

只有相应权限的人才能访问某个路径，只针对一个权限

```
@Override
protected void configure(HttpSecurity security) throws Exception {
   security.formLogin()
           //自定义用户登陆页面,需要认证的页面都会跳转到这个页面
           .loginPage("/login.html")
           //登陆页面表单提交到什么地址，这个地址不认证
           .loginProcessingUrl("/user/login")
           //登陆成功，跳转路径
           .defaultSuccessUrl("/sucess").permitAll()
            //设置哪些路径不需要认证可以直接访问
            .and().authorizeRequests().antMatchers("/","/test/hello","user/login").permitA
            //只有具有admis权限才能访问这个路径
            .antMatchers("/test/auth").hasAuthority("admins")
            //其他任何请求都要过验证
            .anyRequest().authenticated()
            //关闭csrf防护
            .and().csrf().disable();
}
```

成功的话，返回auth

```
@RequestMapping("/test/auth")
public String auth(Users users){
    return "auth";
}
```

访问失败

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Sep 20 09:23:46 CST 2021
There was an unexpected error (type=Forbidden, status=403).
Forbidden

4.2 基于权限访问控制：hasAnyAuthority方法

如果有其中任意一个权限的人都可以访问这个路径

```
.antMatchers("/test/auth").hasAnyAuthority("role,manager")
```

4.3 基于角色访问控制：hasRole方法

如果用户是给定角色就允许访问，否则返回403

```
//3.hasRole  具有role权限或者manager访问这个路径  查看源码可知权限需要：ROLE_开头
 .antMatchers("/test/auth").hasRole("sale")
```

4.4 基于角色访问控制：hasAnyRole方法

如果用户是给定多个角色之一就允许访问

```
//4.hasAnyRole  具有role权限或者manager访问这个路径  查看源码可知权限需要：ROLE_开头
.antMatchers("/test/auth").hasAnyRole("sale","leader")
```

5.设置403页面

```
@Override
    protected void configure(HttpSecurity security) throws Exception {
        //没有权限跳转到的页面
        security.exceptionHandling().accessDeniedPage("/unauth.html");
        ......
    }
```

6.注解使用

6.1 @Secured

作用：用户只有具有某个角色才能访问方法

1.使用注解前需要开启注解,可以放在配置类上也可以放在启动类上

```
@EnableGlobalMethodSecurity(securedEnabled = true)
```

2.在controller层使用注解，角色前增加ROLE_开头前缀

```
    @RequestMapping("/update")
    @Secured({"ROLE_sale","ROLE_manager"})
    public String update(Users users){
        return "hello update";
    }
```

6.2 @PreAuthorize

作用：在方法之前进行权限验证

1.使用注解前需要开启注解,可以放在配置类上也可以放在启动类上

```
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)
```

2.在controller层使用注解

```
    @RequestMapping("/update")
    //可以是hasAnyAuthority、hasAuthority、hasAnyRole、hasRole
    @PreAuthorize("hasAnyAuthority('admins')")
    public String update(Users users){
        return "hello update";
    }
```

6.3 @PostAuthorize

作用：在方法之后进行权限验证，适合验证有返回值的权限

1.使用注解前需要开启注解,可以放在配置类上也可以放在启动类上

```
@EnableGlobalMethodSecurity(securedEnabled = true,prePostEnabled = true)
```

2.在controller层使用注解

```
    @RequestMapping("/update")
    //可以是hasAnyAuthority、hasAuthority、hasAnyRole、hasRole
    @PostAuthorize("hasAnyAuthority('admins')")
    public String update(Users users){
        return "hello update";
    }
```

6.4 @PostFilter

作用：方法返回的数据过滤

```
@RequestMapping("/getALl")
@PostAuthorize("hasAnyAuthority('admins')")
//返回数据中 只返回username是admin1的数据
@PostFilter("filterObject.username =='admin1'")
public List<Users> getALl(){
    ArrayList<Users> list =new ArrayList<>();
    list.add(new Users(11,"admin1","666"));
    list.add(new Users(21,"admin2","888"));
    //返回结果 [{"id":11,"username":"admin1","password":"666"}]
     return list;
}
```

6.5 @PreFilter

作用：传入方法的数据进行过滤

```
@RequestMapping("/getTestPreFilter")
@PostAuthorize("hasAnyAuthority('admins')")
//对传入参数进行过滤
@PreFilter(value="filterObject.id%2 ==0")
public List<Users> getTestPreFilter(@RequestBody List<Users> list){
   list.forEach(t->{
       System.out.println(t.getId() + "\t" + t.getUsername());
   });
```

```
    return list;
}
```

7. 用户注销操作

7.1 配置类增加退出设置

```
//退出设置,logoutUrl("/logout"), 不用写logout logoutSuccessUrl: 退出之后跳转的url
security.logout().logoutUrl("/logout").logoutSuccessUrl("/test/logout").permitAll();
```

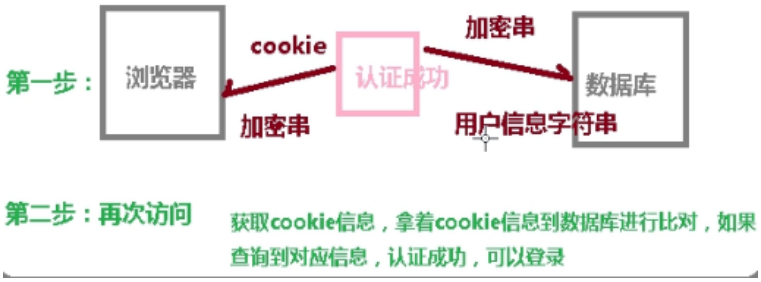
8.基于数据库实现自动登录

8.1 实现原理

1.用户验证成功之后，分别向浏览器cookie和数据库中写入加密串

2.用户再次来访问，携带cookie中加密串的信息去数据中查找，查找到结果登录成功，查找不到重新登录

3.指定时间内免登录是设置cookie的有效时间



8.2 具体实现

1.修改自定义配置类

```
@Configuration
public class RememberMeConfig extends WebSecurityConfigurerAdapter {

    //注入数据源
    @Autowired
    private DataSource dataSource;
    @Autowired
    private MyUserDetailsService userDetailsService;

     @Bean
     public PersistentTokenRepository persistentTokenRepository(){
         JdbcTokenRepositoryImpl jdbcTokenRepository = new JdbcTokenRepositoryImpl();
         //设置数据源
         jdbcTokenRepository.setDataSource(dataSource);
         //自动执行创建表操作，第二次启动就要注释掉 否则会重复
//         jdbcTokenRepository.setCreateTableOnStartup(true);
         return  jdbcTokenRepository;
     }

     @Override
     protected void configure(HttpSecurity security) throws Exception {
         //退出设置,logoutUrl("/logout"), 不用写logout logoutSuccessUrl: 退出之后跳转的url
         security.logout().logoutUrl("/logout").logoutSuccessUrl("/test/logout").permitAll
         //没有权限跳转到的页面
         security.exceptionHandling().accessDeniedPage("/unauth.html");
```

```
        security.formLogin()
                //自定义用户登陆页面,需要认证的页面都会跳转到这个页面
                .loginPage("/login.html")
                //登陆页面表单提交到什么地址，这个地址不认证
                .loginProcessingUrl("/user/login")
                //登陆成功，跳转路径
                .defaultSuccessUrl("/sucess.html").permitAll()
                //设置哪些路径不需要认证可以直接访问
                .and().authorizeRequests().antMatchers("/","user/login").permitAll()
                .antMatchers("/test/auth").hasAnyRole("sale1","leader")
                //其他任何请求都要过验证
                .anyRequest().authenticated()
                //实现自动登陆功能
                // tokenRepository：操作数据库对象
                // tokenValiditySeconds：60 设置有效时长，单位是秒
                .and().rememberMe().tokenRepository(persistentTokenRepository()).tokenVal
                .userDetailsService(userDetailsService)
                //关闭csrf防护
                .and().csrf().disable();
    }

    @Bean
        //创建实现类，否则会报There is no PasswordEncoder mapped for the id "null"
    PasswordEncoder init(){
        return new BCryptPasswordEncoder();
    }
```

2.修改登陆页面

```
<body>
    <form action="/user/login" method="post">
        <!-- name必须是username和password，否则spring security得不到用户名和密码 -->
        用户名：<input type="text" name="username"><br/>
        密码：<input type="text" name="password"><br/>
        <!-- name必须是remember-me      -->
        <input type="checkbox" name="remember-me"><br/> 自动登陆
        <input type="submit" value="login">
    </form>
</body>
```

3.登陆之后，cookies中增加一条remember-me记录

Response Cookies

Name	Value
remember-me	M2NmZFZjZ1glMkJMOVN5SW…

数据库增加一条user1的记录

username	series	token	last_used
user1	3cfdVcgX	9FXwMh8	2021-09-21 (

9. csrf跨站请求伪造

9.1 概念

跨站请求攻击，简单地说，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作(如发邮件，发消息，甚至财产操作如转账和购买商品)。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去运行。 这利用了 web 中用户身份验证的一个漏洞:简单的身份验证只能保证请求发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的。

用户登陆网站A验证通过，信息存储在cookies中，网站B可以获取cookies全部内容。

从 Spring Security 4.0 开始，默认情况下会启用 CSRF 保护，以防止 CSRF 攻击应用程序，Spring Security CSRF 会针对 PATCH，POST，PUT 和 DELETE 方法进行防护

```
private static final class DefaultRequiresCsrfMatcher implements RequestMatcher {
    private final HashSet<String> allowedMethods;

    private DefaultRequiresCsrfMatcher() {
        //不对get方法防护
        this.allowedMethods = new HashSet(Arrays.asList("GET", "HEAD", "TRACE", "OPTIONS")
    }

    public boolean matches(HttpServletRequest request) {
         return !this.allowedMethods.contains(request.getMethod());
     }
}
```

9.2 原理

1.生成 csrfToken 保存到 HttpSession 或者 Cookie 中。

```
public final class CsrfFilter extends OncePerRequestFilter {
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse respon
                                    throws ServletException, IOException {
    //首先从session或者cookie中拿token
    CsrfToken csrfToken = this.tokenRepository.loadToken(request);
      .....
    }
}
```

2.请求到来时，从请求中提取csrfToken，和保存的csrfToken做比较，进而判断当前请求是否合法。主要通过 CsrfFilter 过滤器来完成。

```
public final class CsrfFilter extends OncePerRequestFilter {
    //从cookies中获得csrfToken
    request.setAttribute(csrfToken.getParameterName(), csrfToken);
    if (!this.requireCsrfProtectionMatcher.matches(request)) {
        filterChain.doFilter(request, response);
    } else {
    }
```

```
8 }
```