# Spring Boot核心-企业级开发-Spring Security

## Spring Security 快速入门

### 什么是Spring Security

Sprirg s~curity 是专门针对基于Spring 的项目的安全框架，充分利用了依赖注入和AOP来实现安全的功能。

在早期的Spring Security 版本，使用Spring Security 需要使用大量的XML 配置，而本节将全部基于Java 配置来实现Spring Security 的功能。

安全框架有两个重要的概念，即认证（ Authentication ）和授权（ Authorization ）。认证即确认用户可以访问当前系统：授权即确定用户在当前系统下所拥有的功能权限，本节将围绕认证和授权展开。

### Spring Security 的配置

（1）DelegatingFilterProxy

Spring Security 为我们提供了一个多个过滤器来实现所有安全的功能，我们只需注册一个特殊的DelegatingFilterProxy 过滤器到WebApplicationlnitializer 即可。而在实际使用中，我们只需让自己的Initializer 类继承AbstractSecurityWebApplicationlnitializer 抽象类即可。AbstractSecurityWebApplicationinitializer 实现了WebApplicationinitializer 接口，并通过onStartup 方法调用。它为我们注册了DelegatingFilterProxy 。insertSpringSecurityFilterChain 源码如下：

|  |
| --- |
| **private void** insertSpringSecurityFilterChain(ServletContext servletContext) {  String filterName = ***DEFAULT\_FILTER\_NAME***;  DelegatingFilterProxy springSecurityFilterChain = **new** DelegatingFilterProxy(  filterName);  String contextAttribute = getWebApplicationContextAttribute();  **if** (contextAttribute != **null**) {  springSecurityFilterChain.setContextAttribute(contextAttribute);  }  registerFilter(servletContext, **true**, filterName, springSecurityFilterChain); } |

所以我们只需用以下代码即可开启Spring Security 的过滤器支持：

|  |
| --- |
| **public class** AppInitializerSample **extends** AbstractSecurityWebApplicationInitializer { } |

（2）配置

Spring Security 的配置和Spring MVC 的配置类似，只需在一个配置类上注解@EnableWebSecurity ，并让这个类继承WebSecurityConfigAdapter即可，我们可与重写configure 方法来配置相关的安全配置。

|  |
| --- |
| @Configuration **public class** WebSecurityConfigSample **extends** WebSecurityConfigurerAdapter {*//1* @Bean  UserDetailsService customUserService() { *//2* **return new** CustomUserService();  }   @Override  **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  **super**.configure(auth);  }   @Override  **public void** configure(WebSecurity web) **throws** Exception {  **super**.configure(web);  }   @Override  **protected void** configure(HttpSecurity http) **throws** Exception {  **super**.configure(http);  } } |

### 用户认证

认证需要我们有一套用户数据的来源， 而授权则是对于某个用户有相应的角色权限。在Spring Security 但我们通过重写方法来实现定制。

|  |
| --- |
| **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {} |

( 1 ）内存中的用户

使用AuthenticationManagerBuilder 的inMemoryAuthentication 方法即可添加在内存中的用户，并可给用户指定角色权限。

|  |
| --- |
| @Override **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {   auth.inMemoryAuthentication()  .withUser(**"wjc"**).password(**"123456"**).roles(**"ROLE\_ADMIN"**)  .and()  .withUser(**"wisely"**).password(**"123456"**).roles(**"ROLE\_USER"**); } |

(2) JDBC 中的用户

JDBC 中的用户直接指定dataSource 即可。

|  |
| --- |
| @Autowired **private** DataSource **dataSource**;  @Override **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  auth.jdbcAuthentication().dataSource(**dataSource**); } |

不过这看上去很奇怪，其实这里的Spring Security 是默认了你的数据库结构的。通过jdbcAuthentication 的源码，我们可以看出在JdbcDaolmpl 中定义了默认的用户及角色权限获取的SQL 语句：

|  |
| --- |
| **public static final** String ***DEF\_USERS\_BY\_USERNAME\_QUERY*** = **"select username,password,enabled "** + **"from users "** + **"where username = ?"**; **public static final** String ***DEF\_AUTHORITIES\_BY\_USERNAME\_QUERY*** = **"select username,authority "** + **"from authorities "** + **"where username = ?"**; **public static final** String ***DEF\_GROUP\_AUTHORITIES\_BY\_USERNAME\_QUERY*** = **"select g.id, g.group\_name, ga.authority "** + **"from groups g, group\_members gm, group\_authorities ga "** + **"where gm.username = ? "** + **"and g.id = ga.group\_id "** + **"and g.id = gm.group\_id"**; |

我们可以自定义我们的查询用户和权限的SQL 语句

|  |
| --- |
| @Override **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  auth.jdbcAuthentication().dataSource(**dataSource**)  .usersByUsernameQuery(**"select username,password,true "** + **"from t\_users "** + **"where username = ?"**)  .authoritiesByUsernameQuery(**"select username,authority "** + **"from t\_authorities "** + **"where username = ?"**); } |

（3）通用的用户

上面的两种用户和权限的获取方式只限于内存或者JDBC ，我们的数据访问方式可以是多种各样的，可以是非关系型数据库，也可以是我们常用的JPA 等。这时我们需要自定义实现UserDetailsService 接口。上面的内存中用户及JDBC 用户就是User Details Service 的实现，定义如下：

|  |
| --- |
| @Configuration **public class** CustomUserServiceSample **implements** UserDetailsService {  @Autowired  SysUserRepository **userRepository**;   @Override  **public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {  SysUser user = **userRepository**.findByUsername(username);  List<GrantedAuthority> authorities = **new** ArrayList<>();  authorities.add(**new** SimpleGrantedAuthority(**"ROLE\_ADMIN"**));  **return new** User(user.getUsername(), user.getPassword(), authorities);  } } |

说明： SysUser 是我们系统的用户领域对象类， User 来自于org.springframework.security. core. userdetails.User。除此之外，我们还需要注册这个CustomU serService ，代码如下：

|  |
| --- |
| @Bean UserDetailsService customUserService() {  **return new** CustomUserService(); }  @Override **protected void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  auth.userDetailsService(customUserService()); } |

（4）请求授权

Spring Security 是通过重写

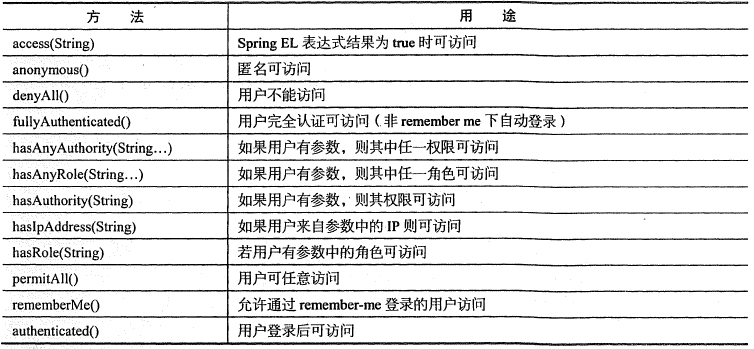
|  |
| --- |
| **protected void** configure(HttpSecurity http) **throws** Exception{} |

方法来实现请求拦截的。Spring Security 使用以下匹配器来匹配请求路径：

* antMatchers ：使用Ant 风格的路径匹配。
* regexMatchers ：使用正则表达式匹配路径。

anyRequest：匹配所有请求路径。

在匹配了请求路径后，需要针对当前用户的信息对请求路径进行安全处理， Spring Security提供了的安全处理方法，如下图所示。



代码示例

|  |
| --- |
| @Override **protected void** configure(HttpSecurity http) **throws** Exception {  http.authorizeRequests()  .antMatchers(**"/admin/\*\*"**).hasRole(**"ROLE\_ADMIN"**)  .antMatchers(**"/user/\*\*"**).hasAnyRole(**"ROLE\_ADMIN"**, **"ROLE\_USER"**)  .anyRequest().authenticated(); } |

说明

1. 通过authorizeRequests 方法来开始请求权限自己宜。
2. 请求匹配/admin\*\*，只有拥有ROLE ADMIN 角色的用户可以访问。
3. 请求匹配/user/\*\*，拥有ROLE\_ADMIN 或ROLE\_USER 角色的用户都可访问。
4. 其余所有的请求都需要认证后（登录后）才可访问。

也可以重写

|  |
| --- |
| **protected void** configure(HttpSecurity http) **throws** Exception{} |

方法来定制我们的登录行为。

|  |
| --- |
| @Override **protected void** configure(HttpSecurity http) **throws** Exception {  http.formLogin()  .loginPage(**"/login"**)  .defaultSuccessUrl(**"/index"**)  .failureUrl(**"/login?error"**)  .permitAll()  .and()  .rememberMe()  .tokenValiditySeconds(**1000**)  .key(**"key"**)  .and()  .logout()  .logoutUrl(**"/custom-logout"**)  .logoutSuccessUrl(**"/logout-success"**)  .permitAll(); } |

1. 通过formLogin 方法定制登录操作。
2. 使用loginPage 方法定制登录页面的访问地址。
3. defaultSuccessUrl 指定登录成功后转向的页面。
4. failure Uri 指定登录失败后转向的页面。
5. remember Me 开启cookie 存储用户信息。
6. token ValiditySeconds 指定cookie 有效期为1209600 秒，即2 个星期。
7. key 指定cookie 中的私钥。
8. 使用logout 方法定制注销行为。
9. logoutUrl 指定注销的URL 路径。
10. logoutSuccessUrl 指定注销成功后转向的页面。

## Spring Boot 的支持

Spring Boot 针对Spring Security 的自动配置在org.springframework.boot.autoconfigure.security 包中。主要通过SecurityAutoConfiguration 和SecurityProperties 来完成配置。Security AutoConfiguration 导入了SpringBootWebSecurityConfiguration 中的配置。在SpringBootWebSecurityConfiguratioh 配置中，我们获得如下的自动配置：

1. 自动配置了一个内存中的用户，账号为user，密码在程序启动时出现。
2. 忽略/css/\*\*、/js/\*\*、/images/\*\*和/\*\*/favicon.ico 等静态文件的拦截。
3. 自动配置的securityFilterChainRegistration 的Bean。

Security Properties 使用以“ security ”为前缀的属性配置Spring Security 相关的配置，包含：

|  |
| --- |
| *# ----------------------------------------*  *# SECURITY PROPERTIES*  *# ----------------------------------------*  *# SECURITY (*[SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.6.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java" \t "_top))  security.basic.authorize-mode=role *# Security authorize mode to apply.*  security.basic.enabled=true *# Enable basic authentication.*  security.basic.path=/\*\* *# Comma-separated list of paths to secure.*  security.basic.realm=Spring *# HTTP basic realm name.*  security.enable-csrf=false *# Enable Cross Site Request Forgery support.*  security.filter-order=0 *# Security filter chain order.*  security.filter-dispatcher-types=ASYNC, FORWARD, INCLUDE, REQUEST *# Security filter chain dispatcher types.*  security.headers.cache=true *# Enable cache control HTTP headers.*  security.headers.content-security-policy= *# Value for content security policy header.*  security.headers.content-security-policy-mode=default *# Content security policy mode.*  security.headers.content-type=true *# Enable "X-Content-Type-Options" header.*  security.headers.frame=true *# Enable "X-Frame-Options" header.*  security.headers.hsts=all *# HTTP Strict Transport Security (HSTS) mode (none, domain, all).*  security.headers.xss=true *# Enable cross site scripting (XSS) protection.*  security.ignored= *# Comma-separated list of paths to exclude from the default secured paths.*  security.require-ssl=false *# Enable secure channel for all requests.*  security.sessions=stateless *# Session creation policy (always, never, if\_required, stateless).*  security.user.name=user *# Default user name.*  security.user.password= *# Password for the default user name. A random password is logged on startup by default.*  security.user.role=USER *# Granted roles for the default user name.* |

Spring Boot 为我们做了如此多的配置，当我们需要自己扩展的配置时，只需配置类继承WebSecurityConfigurerAdapter 类即可。