保护Web应用

1 Spring Security

Spring Security是为基于Spring的应员工程序提供声明式安全保护的安全性框架。提供了完整的安全性解决方案，它能够古在Web请求级别和方法调用级别处理身份认证和授权。

因为基于Spring框架，所以Spring Security充分利用了依赖注入和面向切面的技术。

1.1 Spring Security的模块

模块	描述
ACL	支持通过访问控制列表(access control list)为域对象提供安全性
切面(Aspects)	一个很小的模块，当使用Spring Security注解时，会使用基于AspectJ的切面，而不是员工标准的Spring AOP
CAS客户端	提供域Jasig的中心认证服务(Central Authentication Service，CAS)进行集成的功能
配置	包含通过XML和Java配置Spring Security的功能支持
核心(Core)	提供Spring Security基本库
加密	提供了加密和密码编码的功能
LDAP	提供基于LDAP进行认证
OpenID	支持使用OpenID进行集中式认证
Remoting	提供了对Spring Remotin的支持
标签库(Tag Library)	Spring Security 的JSP标签库
Web	提供了Spring Security基于Filter的Web安全性支持

应用程序路径下需要包含Core和Configuration这两个模块

1.2 过滤Web请求

Spring Security借助 一系列的Servlet Filter提供各种安全性功能。DelegatingFilterProxy是一个特殊的ServletFilter，它本身工作不多。只是将工作委托给一个javax.servlet.Filter实现类，这个实现类作为一个bean注册在Spring应用上下文中。

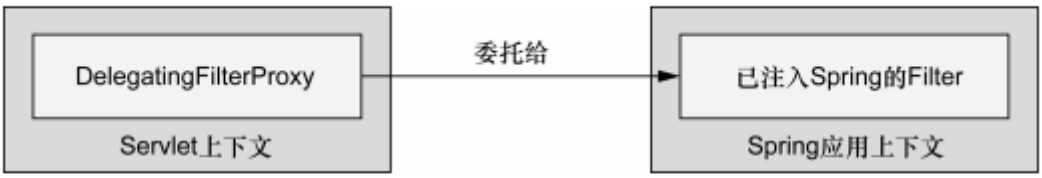

图9.1 DelegatingFilterProxy把Filter的处理逻辑委托给Spring应用上下文中所定义的一个代理Filter bean

```
import org.springframework.security.web.context.*;

public class SecurityWebInitializer extends
AbstractSecurityWebApplicationInitializer {

}
```

AbstractSecurityWebApplicationInitializer 实现了WebApplicationInitializer，因此Spring会发现它，并用它在Web容器中注册DelegatingFilterProxy。尽管可以重载它的appendFilters()或insertFilters()，方法来注册自己选择的Filter，但是要注册DelegatingFilterProxy，不需要重载任何方法。**DelegatingFilterProxy会拦截发往应用中的请求，并将请求委托个ID为springSecurityFilterChain bean。**

springSecurityFilterChain 本身是一个特殊的Filter，也被称为FilterChainProxy。它可以连接任意一个或多个其他的Filter。Spring Security依赖一系列的Servlet Filter来提供不同的安全性。但是我们不需要知道这些细节。因为不需要显式声明springSecurityFilterChain 以及其他所连接在一起的其他Filter。当启动Web安全性的时候，会自动创建这些Filter。

1.3 编写安全性配置

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

@EnableWebSecurity将会启用Web安全共功能。但它本身没什么用处，Spring Security必须配置在一个实现了WebSecurityConfigurer的bean或者扩展WebSecurityConfigurerAdapter

通过重载WebSecurityConfigurerAdapter中的一个或多个方法来配置Web安全性

方法	描述
configure(WebSecurity)	通过重载，配置Spring Security的Filter链
configure(HttpSecurity)	通过重载，配置如何通过拦截器保护请求
configure(AuthenticationManagerBuilder)	通过重载，配置user-data服务

```
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .anyRequest().authenticated()
      .and()
    .formLogin().and()
    .httpBasic();
}
```

这个简单的默认配置指定了该如何保护HTTP请求，以及客户端认证用户的方案。通过调用authorizeRequests()和anyRequest().authenticated()就会要求所有进入应用的HTTP请求都要进行认证。它也配置Spring Security支持基于表单的登录以及HTTP Basic方式的认证。

同时，因为我们没有重载configure(AuthenticationManagerBuilder)方法，所以没有用户存储支撑认证过程。没有用户存储，实际上就等于没有用户。所以，在这里所有的请求都需要认证，但是没有人能够登录成功。

为了让Spring Security满足我们应用的需求，还需要再添加一点配置。具体来讲，我们需要：

- 配置用户存储；
- 指定哪些请求需要认证，哪些请求不需要认证，以及所需要的权限；
- 提供一个自定义的登录页面，替代原来简单的默认登录页。

除了Spring Security的这些功能，我们可能还希望基于安全限制，有选择性地在Web视图上显示特定的内容。

2 选择查询用户信息的服务

2.1 使用基于内存的用户存储

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.inMemoryAuthentication()
                .withUser("user").password("password").roles("USER").and()
                .withUser("admin").password("passward").roles("USER", "ADMIN");
    }
}
```

通过inMemoryAuthentication方法，可以启用配置并任意填充基于内存的用户存储

withUser()方法为内存用户存储添加新的用户，并提供多个进一步配置用户的方法，包括设置用户密码的passward()方法以及给用户授予一个或多个角色权限的roles方法。

roles()方法是authorities()方法的简写形式，roles()方法所给定值都会添加一个”ROLE_“前缀，并将其作为权限授予给用户。

等同于

```
auth.inMemoryAuthentication()
        .withUser("user").password("password").authorities("ROLE_USER").and()
        .withUser("admin").password("passward").authorities("ROLE_USER",
"ROLE_ADMIN");
```

表9.3 配置用户详细信息的方法

方　法	描　述
accountExpired(boolean)	定义账号是否已经过期
accountLocked(boolean)	定义账号是否已经锁定
and()	用来连接配置
authorities(GrantedAuthority...)	授予某个用户一项或多项权限
authorities(List<? extends GrantedAuthority>)	授予某个用户一项或多项权限
authorities(String...)	授予某个用户一项或多项权限
credentialsExpired(boolean)	定义凭证是否已经过期
disabled(boolean)	定义账号是否已被禁用
password(String)	定义用户的密码
roles(String...)	授予某个用户一项或多项角色

2.2 基于数据库表进行认证

```
@Configuration
@EnableWebSecurity
class JdbcSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private DataSource dataSource;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.jdbcAuthentication().dataSource(dataSource);
    }
}
```

必须配置的是一个DataSource，这样的化，就可以访问关系型数据库

2.2.1 重写默认的用户查询功能

```
public static final String DEF_USERS_BY_USERNAME_QUERY =
        "select username,password,enable 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,ga.authority from groups g, group_name gm,
group_authorities ga where gm.username = ? and g.id = ga.group_id and g.id =
gm.group.id";
```

如果能够在数据库中定义和填充满足这些查询的表，那么基本上就不需要在做什么额外的事情。同时如果希望在查询上由更多的控制。可以按照如下方式配置：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
    .usersByUsernameQuery("select username, password, true from Spitter where
username=?")
    .authoritiesByUsernameQuery("select username, 'ROLE_USER' from Spitter where
username = ?");
}
```

2.2.2 使用转码后的密码

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.jdbcAuthentication().dataSource(dataSource)
    .usersByUsernameQuery("select username, password, true from Spitter where
username=?")
    .authoritiesByUsernameQuery("select username, 'ROLE_USER' from Spitter where
username = ?")
    .passwordEncoder(new StandardPasswordEncoder("53cr3t"));
}
```

Spring Security的加密模块包含了BCryptPasswordEncoder、StandardPasswordEncoder和NoOpPasswordEncoder。当内置实现无法满足的时候，可以提供自定义实现。PasswordEncoder接口。

2.3 基于LDAP进行认证

```
@Configuration
@EnableWebSecurity
class LadpSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.ldapAuthentication()
                .userSearchBase("ou=people")
                .userSearchFilter("{uid={0}}")
                .groupSearchBase("ou=groups")
                .groupSearchFilter("member={0}");
    }
}
```

userSearchBase属性为查找用户提供了基础查询。同样，groupSearchBase为查找组制定了基础查询。声明用户应该在名为people的组织单元下搜索而不是从根开始。而组应该在名为groups的组织单元下搜索。

基于LDAP进行认证的默认策略是进行绑定操作，直接通过LDAP服务器认证用户。

另一种可选的方式是进行比对操作，涉及将输入的密码发送到LDAP目录上，并要求服务器将这个密码和用户的密码进行比对。英雌比对实在LDAP服务器内完成的，实际的密码能够保持私密。

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.ldapAuthentication()
            .userSearchBase("ou=people")
            .userSearchFilter("{uid={0}}")
            .groupSearchBase("ou=groups")
            .groupSearchFilter("member={0}")
            .passwordCompare()
}
```

默认情况下，在登陆表单中提供的密码将会与用户的LDAP条目中的userPasword属性进行对比时。如果密码被保存在不同属性中，可以通过passwordAttribute方法来声明密码属性的名称

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.ldapAuthentication()
            .userSearchBase("ou=people")
            .userSearchFilter("{uid={0}}")
            .groupSearchBase("ou=groups")
            .groupSearchFilter("member={0}")
            .passwordCompare()
            .passwordEncoder(new Md5PasswordEncoder())
            .passwordAttribute("passcode");
}
```

2.3.2 引用原册灰姑娘LDAP服务器

默认情况下，Spring Security的LDAP认证假设LDAP服务器监听本地的33389段门口。

```
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.ldapAuthentication()
                .userSearchBase("ou=people")
                .userSearchFilter("{uid={0}}")
                .groupSearchBase("ou=groups")
                .groupSearchFilter("member={0}")
                .contextSource().url("ldap://habuma.com.389/dc=habuma,dc=com")
        ;
    }
```

2.3.4 配置嵌入式的LDAP服务器

```
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.ldapAuthentication()
                .userSearchBase("ou=people")
                .userSearchFilter("{uid={0}}")
                .groupSearchBase("ou=groups")
                .groupSearchFilter("member={0}")
                .contextSource()
                .root("dc=hahuma,dc=com")
        ;
    }
```

当LDAP服务器启动时，他会尝试在类路径下巡查哦LDIF文件来加载数据。LDIF(LDAP Data InterChange Format， LDAP数据交换格式)是以文本文件展现LDAP数据的标准方式，每条记录可以有一行或多行，每项包含一个名值对，记录之间通过空行分割。同时也可以调用ldif()方法指定加载哪个LDIF文件

```
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        auth.ldapAuthentication()
                .userSearchBase("ou=people")
                .userSearchFilter("{uid={0}}")
                .groupSearchBase("ou=groups")
                .groupSearchFilter("member={0}")
                .contextSource()
                    .root("dc=hahuma,dc=com")
                    .ldif("classpath:users.ldif")
        ;
    }
```

```
dn: ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
dn: ou=people,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people
dn: uid=habuma,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Craig Walls
sn: Walls
uid: habuma
userPassword: password
dn: uid=jsmith,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John Smith
sn: Smith
uid: jsmith
userPassword: password
dn: cn=spittr,ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: groupOfNames
cn: spittr
member: uid=habuma,ou=people,dc=habuma,dc=com
```

2.4 配置自定义的用户服务

需要提供一个自定义的UserDetailsService接口实现

```
public class SpitterUserService implements UserDetailsService {
    private final SpitterRepository spitterRepository;

    public SpitterUserService(SpitterRepository spitterRepository) {
        this.spitterRepository = spitterRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Spitter spitter = spitterRepository.findByUsername(username);
        if(spitter != null)
        {
            List<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(new SimpleGrantedAuthority("ROLE_SPITTER"));
            return new
User(spitter.getUsername(),spitter.getPassword(),authorities);
        }

        throw  new UsernameNotFoundException("User " + username + "not found.");
    }
}
```

```
@Configuration
@EnableWebSecurity
class UserDetailsSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private SpitterRepository spitterRepository;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
        //自定义权限认证
        auth.userDetailsService(new SpitterUserService(spitterRepository));
    }
}
```

此外，还可以修改Spitter，让其实现UserDetails。这个样，loadUserByUsername()就可以直接返回Spitter对象，而不必再将它的值赋值到User对象中。

3 拦截请求

对每个请求进行细粒度安全控制的关键在于重载configure(HttpSecurity)方法，

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/").authenticated()//对/请求需要进行认证
        .antMatchers("/spitter/me").authenticated()//对"/spitter/me"请求需要进行认
证
        .antMatchers(HttpMethod.POST, "/spittles").authenticated()
        .anyRequest().permitAll();//其他的请求都允许，不需要认证和任何权限
}
```

authorizeRequests()调用该方法所返回的对象的方法来配置请求的细节。同时也可以再对一个antMather方法的调用中指定多个路径：

.antMatchers("/spitters/**"，”spittles/mine“).authenticated()。antMatchers方法所使用的路径可能会包含Ant风格的通配符，而regexMatchers方法则能够几首正则表达式来定义请求。

.regexMatchers("/spitters/.*").authenticated()

- authenticated() 要求在执行请求的时候，必须已经登陆了应用。如果用户没有认证，Spring Security的Filter将会捕获该请求，并将用户重定向到应用的登陆页面
- permitAll() 允许请求没有任何的安全限制

用来定义如何保护路径的配置方法

方　法	能够做什么
access(String)	如果给定的SpEL表达式计算结果为true，就允许访问
anonymous()	允许匿名用户访问
authenticated()	允许认证过的用户访问
denyAll()	无条件拒绝所有访问
fullyAuthenticated()	如果用户是完整认证的话(不是通过Remember-me功能认证的)，就允许访问
hasAnyAuthority(String...)	如果用户具备给定权限中的某一个的话，就允许访问
hasAnyRole(String...)	如果用户具备给定角色中的某一个的话，就允许访问
hasAuthority(String)	如果用户具备给定权限的话，就允许访问
hasIpAddress(String)	如果请求来自给定IP地址的话，就允许访问
hasRole(String)	如果用户具备给定角色的话，就允许访问
not()	对其他访问方法的结果求反
permitAll()	无条件允许访问
rememberMe()	如果用户是通过Remember-me功能认证的，就允许访问

可以将任意数量的antMathers、regexMatchers和anyRequest连接起来，这些规则会按照贵的的顺序发挥作用。更重要的一点是将作为具体的请求路径放在前面，而最不具体的路径(anyRequest)放在后面。如果不这样做，那么不具体的路径配置就会被府改为更具体的路径配置

3.1 使用Spring表达式进行安全保护

上表中的方法，不能再相同的路径上同时使用多个限制方法。因此需要借助于SpEL

```
.antMatchers("/spitter/me").access("hasRole('ROLE_SPITTER')")
```

表9.5 Spring Security通过一些安全性相关的表达式扩展了Spring表达式语言

安全表达式	计算结果
authentication	用户的认证对象
denyAll	结果始终为false
hasAnyRole(list of roles)	如果用户被授予了列表中任意的指定角色，结果为true
hasRole(role)	如果用户被授予了指定的角色，结果为true
hasIpAddress(IP Address)	如果请求来自指定IP的话，结果为true
isAnonymous()	如果当前用户为匿名用户，结果为true
isAuthenticated()	如果当前用户进行了认证的话，结果为true
isFullyAuthenticated()	如果当前用户进行了完整认证的话（不是通过Remember-me功能进行的认证），结果为true
isRememberMe()	如果当前用户是通过Remember-me自动认证的，结果为true
permitAll	结果始终为true
principal	用户的principal对象

Spring Security表达式不在局限于基于用户的权限进行访问限制。

```
.antMatchers("/spitter/me").access("hasRole('ROLE_SPITTER') and
hasIpAddress('127.00.1')")
```

3.2 强制通道的安全

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/").authenticated()
        .antMatchers("/spitter/me").authenticated()
        .antMatchers(HttpMethod.POST, "/spittles").authenticated()
        .antMatchers("/spitter/me").access("hasRole('ROLE_SPITTER') and
hasIpAddress('127.00.1')")
        .anyRequest().permitAll()
        .and()
        .requiresChannel()
        .antMatchers("/spitter/form").requiresSecure()
    ;
}
```

不论何时，只要是对"/spitter/form"的请求，Spring、 Security都视为需要安全通道并自动将请求重定向到HTTPS上。与之相反，有些页面并不需要通过HTTPS传送。可以使用requiresInsecure()替代requiresSecure

3.3 防止跨站请求伪造

跨站请求伪造(cross-site requestforgery CSRF)。如果以恶站点欺骗用户提交起高球到其他服务器，就会发生CSRF攻击

Spring Security通过一个同步token的方式来实现CSRF防护。他将会拦截状态变化的请求(GET、HEAD、OPTIONS和TRACE的请求)并检查CSRFtoken。如果请求终不包含CSRF token，或者toketn不能与服务端的token匹配，请求将会失败，并抛出CsrfException。意味着，所有的表单必须在一个"_csrf"域中提交token，而且这个token必须要与服务器段计算存储的token一致。这样提交表单的时候，才能进行匹配

好消息是，Spring Security已经简化了将token放到请求的属性中这一任务。如果你使用Thymeleaf作为页面模板的话，只要<form>标签的action属性添加了Thymeleaf命名空间前缀，那么就会自动生成一个"_csrf"隐藏域：

```
<form method="POST" th:action="@{/spittles}">
  ...
</form>
```

如果使用JSP作为页面模板的话，我们要做的事情非常类似：

```
<input type="hidden"
       name="${_csrf.parameterName}"
       value="${_csrf.token}" />
```

更好的功能是，如果使用Spring的表单绑定标签的话，<sf:form>标签会自动为我们添加隐藏的CSRF token标签。

处理CSRF的另外一种方式就是根本不去处理它。我们可以在配置中通过调用csrf().disable()禁用Spring Security的CSRF防护功能，如下所示：

程序清单9.6 我们可以禁用Spring Security的CSRF防护功能

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
    ...
    .csrf()
      .disable();      <-- 禁用 CSRF 防护功能
}
```

需要提醒的是，禁用CSRF防护功能通常来讲并不是一个好主意。如果这样做的话，那么应用就会面临CSRF攻击的风险。只有在深思熟虑之后，才能使用程序清单9.6中的配置。

4 认证用户

formLogin启用默认的登录页

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .formLogin()
            .and()
            .authorizeRequests()
            .antMatchers("/").authenticated()
            .antMatchers(HttpMethod.POST, "/spittles").authenticated()
```

```
            .antMatchers("/spitter/me").access("hasRole('ROLE_SPITTER') and
hasIpAddress('127.00.1')")
            .anyRequest().permitAll()
            .and()
            .requiresChannel()
            .antMatchers("/spitter/form").requiresSecure()
    ;
}
```

4.1 添加自定义的登录页

4.2 启用HTTP Basic认证

HTTP Basic认证会直接通过HTTP请求本省，对要访问应用程序的用户进行认证。

启用的化只需要再configure()方法所出啊如的HttpSecurity对象上掉员工HttpBasic即可，还可以调用realmName方法指定域

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .formLogin()
        .loginPage("/login")
        .and()
        .httpBasic()
        .realmName("Spittr");
        }
```

4.3 启用Remember-me功能

Remember-me功能，只要登录过一次，应用就会记住，当再次回到应用功德时候就不需要登陆了

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .formLogin()
            .loginPage("/login")
            .and()
            .rememberMe()
            .tokenRepository(new InMemoryTokenRepositoryImpl())
            .tokenValiditySeconds(2419200)
            .key("spittrKey")
            .and()
            .httpBasic()
            .realmName("Spittr")
            .and()
            .authorizeRequests()
            .anyRequest().permitAll()
            .and()
            .requiresChannel()
            .antMatchers("/spitter/form").requiresSecure()
    ;
}
```

默认情况下这个官鞡你是通过再cookie中存储一个token完成的，这个token最多两周有效，有效时间可以指定。

存储在cookie中的token包含用户名。密码、过期时间和一个私钥--在写入cookie前都进行MD5哈希，默认情况下，私钥名为SpringSecured，可以设置为其他名。

4.4 退出

退出功能是通过Servlet容器中共的Filter实现的，这个Filter会拦截针对”/logout“的请求，因此，为应用添加退出功能只需添加如下连接即可(Thymeleaf代码片段)

```
<a th:href="@{/logout}">Logout</a>/a>
```

当用户点击这个连接的时候，会发起对"logout"的请求，这个请求会被Spring Security的LogoutFilter所处理。用户退出应用，所有的Remember-me token都会被清除掉。在退出后，用户浏览器将会从定向到”/login?logout“，从而允许用户进行再次登录。

还可以通过设置重定向到其他页面

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .formLogin()
        .loginPage("/login")
        .and()
        .logout()
        .logoutSuccessUrl("/")
```

重写默认的LogoutFilter拦截路径

```
.logout()
.logoutSuccessUrl("/")
.logoutUrl("/signout")
```

5 保护视图

Spring Security本身提供了一个JSP标签库，而Thymeleaf通过特定的方言实现了与Spring Security的集成

5.1 使用Spring Security的JSP标签库

Spring Security通过JSP标签库在驶入曾上支持安全性

JSP标签	作用
security:accesscontrollist	如果用户通过访问控制列表授予了指定的权限，那么全然该标签体中的内容
security:authentication	渲染当前用户认证独享的详细信息
security:authorize	如果用户被授予了特定的授权或者SpEL表达式的结果为true，那么渲染该标签体中的内容

5.2 使用Thymeleaf的Spring Security方言

Thymeleaf的安全方言提供了与Spring Security标签库相对应的属性

属性	作用
sec:authentication	渲染认证对象的属性。类似于Spring Security的sec:authentication/JSP标签
sec:authorize	基于表达式的计算结果，条件性的渲染内容。类似于Spring Security的sec:authorize/JSP标签
sec:authoirze-acl	基于表达式的计算结果，条件性的渲染内容。类似于Spring Security的security:accesscontrollist/JSP标签
sec:authorize-expr	sec:authorize属性的别名
sec:authorize-url	基于给定URL路径相关的安全规则，条件性的渲染内容。类似于Spring Security的sec:authorize/JSP标签使用url属性时的场景

为了使用安全方言，需要确保Thymeleaf Extras Spring Security位于应用的路径下。然后还需要再配置中是哟共SpringTemplateEngine来注册SpringSecurityDialect

```
@Bean
public ISpringTemplateEngine springTemplateEngine(ITemplateResolver
templateResolver) {
    SpringTemplateEngine springTemplateEngine = new SpringTemplateEngine();
    springTemplateEngine.addTemplateResolver(templateResolver);
    springTemplateEngine.addDialect(new SpringSecurityDialect());
    return springTemplateEngine;
}
```

标准的Thymeleaf方法依旧与之前一样，使用th前缀，安全方言则设置为使用sec前缀

```
<div sec:authorize="isAuthenticated()">
    Hello <span sec:authentication="name">someone</span>
</div>
```

sec:authorize属性会接受一个SpEl表达式。如果表达式的计算结果为true，那么元素的主体内容就会渲染。isAuthenticated，只有用户进行了认证才会渲染

标签的主题内容

```
<span sec:authorize-url="/admin">
    <br/><a th:href="@{/admin}">Admin</a>
</span>
```

如果用户有权限访问"/admin"，那么到管理页面的连接就会渲染，否则不渲染。