5.Java配置

在Spring 3.1中Spring框架增加了对Java配置的支持。自从Spring Security 3.2开始，Spring Security也开始支持Java配置以便于使用者能够不使用任何的XML文件进行更简单的配置。

如果你熟悉第六章、安全命名空间配置的话，那么你会发现Java配置和XML配置很相似。

Spring Security提供了许多简单应用程序去指导Spring Security的Java配置。

5.1 Web安全的Java配置

第一部是去创建我们自己的Spring Security Java配置。这个配置创建了一个名字为springSecurityFilterChain的Servlet过滤器，这个过滤器对你应用程序中所有的安全工作负责。最基础的配置例子如下：

```
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.*;
import org.springframework.security.config.annotation.authentication.builders.*;
import org.springframework.security.config.annotation.web.configuration.*;

@EnableWebSecurity
public class WebSecurityConfig implements WebMvcConfigurer
{
    @Bean
    public UserDetailsService userDetailsService() throws Exception
    {
        InMemoryUserDetailsManager manager = new InmemoryUserDetailsManager();

manager.createUser(User.withDefaultPassworderEncoder().username("user").password("password").roles("U
SER").build());
        return manager;
    }
}
```

虽然这个配置不是很多，但是它确实做了很多事情。下面对这些事情进行了一个总结：

- 对你应用中的每个URL进行需要权限的配置
- 为你生成一个登陆表单
- 允许用户使用表单基本认证，提供用户名和密码进行认证
- 允许用户去登出
- CSRF 攻击预防
- 会话固定保护
- 安全头集成 - 强制安全传输技术
    - [X-Content-Type-Options](https://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx)集成
    - 缓存控制
    - [X-XSS-Protection](https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/compatibility/dd565647(v=vs.85))集成
    - 防止点击劫持的X-Frame-Options集成
- 继承了下面的Servlet API 方法
    - [HttpServletRequest#getRemoteUser()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getRemoteUser( ))
    - [HttpServletRequest.html#getUserPrincipal()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getUserPrincipal( ))
    - [HttpServletRequest.html#isUserInRole(java.lang.String)](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#isUserInRole(java.lang.String))
    - [HttpServletRequest.html#login(java.lang.String, java.lang.String)](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#login(java.lang.String,%20java.lang.String))
    - [HttpServletRequest.html#logout()](https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#logout( ))

5.1.1 AbstractSecurityWebApplicationInitializer

下一步是去注册springSecurityFilterChain。这项工作能够在Servlet 3.0+环境中使用Spring的 WebApplicationInitializer 支持进行Java配置。Spring Security也提供了一个基类AbstractSecurityWebApplicationInitializer去保证springSecurityFilterChain能够被注册。我们使用AbstractSecurityWebApplicationInitializer取决于是否我们已经使用了Spring或者是否Spring Security仅仅是我们程序中的一个Spring组件。

- 5.1.2节 介绍了如果你没有使用Spring如何使用AbstractSecurityWebApplicationInitializer
- 5.1.3节 介绍了如果你使用了Spring 如何使用AbstractSecurityWebApplicationInitializer

5.1.2 AbstractSecurityWebApplicationInitializer 不使用Spring

如果你没有使用Spring或者Spring MVC ,您需要将WebSecurityConfig传递给超类，以确保获取配置。下面给出一个例子：

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

public class SecurityWbeApplicaitonInitalizer extends AbstractSecurityWebApplicationInitializer
{
        public SecurityWebApplicationInitializer()
        {
                super(WebSecurityConfig.class);
        }
}
```

这个SecurityWebApplicationInitializer将会做如下的事情：

- 为你应用程序中每个URL自动注册springSecurityFilterChain
- 添加一个加载 WebSecurityConfig的ContextLoaderListener。

5.1.3 使用Spring MVC时使用 AbstractSecurityWebApplicationInitializer

如果我们在自己的应用程序中正在使用Spring，那么我们可能已经有了一个WebApplicationInitializer，它可以加载我们的Spring配置。如果我们使用以前的配置，那么可能会出现错误。我们应该使用已经存在的ApplicationContext注册Spring Security。例如，如果我们正在使用Spring MVC，那么我们的SecurityWebApplicationInitializer应该按照如下进行配置：

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

public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer
{
}
```

上述的配置仅仅为你应用程序中的每个URL注册了springSecurityFilterChain，在这之后我们要确保WebSecurityConfig已经被加载到已经存在的ApplicationInitializer中。例如，如果你使用的是Spring MVC ，那么该配置可以被添加到getRootConfigClasses()中，代码如下：

```
public class MvcWebApplicationInitializer extends
AbstractAnnotationConfigDiespatcherServletInitializer
{
        @Override
        protected Class<?>[] getRootConfigClasses()
        {
                return Class[]{WebSecurityConfig.class};
        }
}
```

5.2 HttpSecurity

迄今为止，我们的WebSecurityConfig仅包含有关如何验证用户的信息。那么Spring怎么知道我们想要所有用户进行验证呢？Spring Security怎么知道我们想要它支持基本表单验证呢？这是因为`WebSecurityConfigurerAdapter`在`configure(HttpSecurity)`中提供了一个默认的设置，如下：

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

默认的设置进行了如下配置：

- 确保应用中所有的请求都只有被验证的用户才能访问。
- 允许用户去通过基本表单登录进行验证。
- 允许用户采用HTTP Basic进行验证

你可能会觉得这段代码和如下的XML命名空间配置非常相似：

```
<http>
    <intercept-url pattern="/**" access="authenticated"/>
    <form-login />
    <http-basic />
</http>
```

Java配置中的and()方法相当于一个闭合的XML标签，这种方式允许我们去继续配置父标签。如果你读了代码你也会这么想。我想去配置被授权的请求、配置表单登录和HTTP Basic验证。

然而，Java配置有不同的默认URL和参数。当创建用户登录页是要注意。这个结果是我们的URL更像RESTful风格。另外，我们用Spring Security帮助保护[信息泄漏](https://www.owasp.org/index.php/Information_Leak_(information_disclosure))是不是很明显的，例如：（我也不知道这文档在说啥鬼，不过他真是这么写的。）

5.3 Java配置和表单登录

你可能想知道登录表单在哪，我们什么时候填过这些信息，因为我们并没有写过HTML文件或者JSP文件。如果Spring Security的配置没有为登录页面明确设置一个URL，Spring security会自动生成一个，基于需要的用于处理提交登录的一般值，然后在注册等一系列操作后，转到默认目标URL。

自动生成的页面对于快速启动项目非常方便，但是大多数应用程序想要提供自己的登录页面，那么，我们按照如下方式更新配置：

```
protected void configure(HttpSecurity http)throws Exception
{
    http
        .authorizeRequests()
            .anyRequest().authenticated()
            .and()
        .formLogin()
            .loginPage("/login")
            .permitAll();
}
```

- loginPage()函数指定了特定的登录页面的地址。
- 我们必须赋予所有用户都能访问登录页面的权限，即unauthenticated权限，formLogin().permitAll()方法帮助我们做到这点。

和我们的现在的配置使用的一个简单的使用JSP实现的登录页面如下：

下面的登录页面表现了我们现在的配置。我们可以建档更新我们的配置，如果默认配置不符合我们的需要的话。

```
<c:url value="/login" var="loginUrl"/>
<form action="<img src="https://latex.codecogs.com/gif.latex?
{loginUrl}&amp;quot;%20method=&amp;quot;post&amp;quot;&amp;gt;%20%20%20%20%20%20%201
&amp;lt;c:if%20test=&amp;quot;"/>{param.error != null}">        2
        <p>
                Invalid username and password.
        </p>
</c:if>
<c:if test="<img src="https://latex.codecogs.com/gif.latex?
{param.logout%20!=%20null}&amp;quot;&amp;gt;%20%20%20%20%20%20%203
        &amp;lt;p&amp;gt;
                You%20have%20been%20logged%20out.
        &amp;lt;&amp;#x2F;p&amp;gt;
&amp;lt;&amp;#x2F;c:if&amp;gt;
&amp;lt;p&amp;gt;
        &amp;lt;label%20for=&amp;quot;username&amp;quot;&amp;gt;Username&amp;lt;&amp;#x2F;label&amp;gt;
        &amp;lt;input%20type=&amp;quot;text&amp;quot;%20id=&amp;quot;username&amp;quot;%20name=&amp;quot;username&amp;quot;&amp;#x2F;
&amp;gt;    4
&amp;lt;&amp;#x2F;p&amp;gt;
&amp;lt;p&amp;gt;
        &amp;lt;label%20for=&amp;quot;password&amp;quot;&amp;gt;Password&amp;lt;&amp;#x2F;label&amp;gt;
        &amp;lt;input%20type=&amp;quot;password&amp;quot;%20id=&amp;quot;password&amp;quot;%20name=&amp;quot;password&amp;quot;&amp;#
x2F;&amp;gt;        5
&amp;lt;&amp;#x2F;p&amp;gt;
&amp;lt;input%20type=&amp;quot;hidden&amp;quot;%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%206
        name=&amp;quot;"/>{_csrf.parameterName}"
        value="${_csrf.token}"/>
<button type="submit" class="btn">Log in</button>
</form>
```

1. 一个对/login的POST方法将会被用来认证用户
2. 如果查询参数error存在，证明认证执行了并且失败了。
3. 如果查询参数logout存在，证明用户成功登出了。
4. username的值必须作为HTTP的名为"username"的属性的值。
5. password的值必须作为HTTP的名为"password"的属性的值。
6. 我们可以看第19.4.3节，包括CSRF Token去学习更多关于第19章，CSRF章节的内容。

5.4 授权请求

我们的例子现在仅仅只要求用户被认证并且在我们的应用程序中，每个被认证的用户都能访问每个URL。我们能指定特定的需求为了我们的URL通过`http.authorizeRequests()`方法添加子标签。例如：

```
protected void configure(HttpSecurity http)throws Exception
{
    http
        .authorizeRequests()1
            .antMatchers("/resources/**","/signup","/about").permitAll()2
            .antMatchers("/admin/**").hasRole("ADMIN")3
            .antMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")4
            .antRequest().authecticated()5
            .and
        //...
        .formLogin();
}
```

1. http.authorizeRequests方法的多个字方法的每个匹配器都被按照声明顺序进行考虑。
2. 我们指定多个用户能访问的URL模式。特别是，任何url能访问一个请求，如果这个请求的URL以"/resources/"开始，或者这个URL是"/signup"或者"/about"。
3. 任意的以"/admin/"开头的URL相匹配的请求都被限制只能是有"ROLE_ADMIN"角色的用户才能访问。你会注意到因为我们调用了hasROle方法，我们没必要指定"ROLE_"前缀。
4. 任意的以"/db/"开头的URL相匹配的请求都被限制只能是同时有"ROLE_ADMIN"和"ROLE_DBA"角色的用户才能访问。你会注意到因为我们调用了hasROle表达式，我们没必要指定"ROLE_"前缀。
5. 在之前的表达式都没有匹配的任何URL都只有在用户被验证才能访问。

5.5 处理登出

当使用`WebSecurityConfigurerAdapter`，登出功能将会自动应用。默认的URL是"/logout"，登出用户通过如下过程：

- 使HTTP Session无效
- 清除配置的RememberMe 验证
- 清除SecurityContextHolder信息
- 重定向到/login?logout

和登录功能配置很相同，然和你也可以根据自己的登出需求配置特定的参数：

```
protected void configure(HttpSecurity http) throws Exception
{
    http
        .logout()1
            .logoutUrl("/my/logout")2
            .logoutSeccessUrl("/my/index")3
            .logoutSuccessHandler("logoutSuccessHandler")4
            .invalidateHttpSession(true)5
            .addLogoutHandler(logoutHandler)6
            .deleteCookies(cookieNamesToClear)7
            .and()
        ...
}
```

1. 提供登出支持，当使用WebSecurityConfigurerAdapter时会自动应用。
2. 触发登出的的URL，默认是/logout。如果CSRF保护是启动的（默认），那么这个请求务必是POST请求。关于更多信息，等查看JavaDOC文档。
3. 在登出完成后重定向到的URL。默认是/login?logout。关于更多信息，等查看JavaDOC文档
4. 让你制定一个自定义的LogoutSuccessHandler。如果这个被指定，那么logoutSuccessUrl()将会被忽略。关于更多信息，请查看JavaDOC文档。
5. 指定是否在登出一段时间后使HTTP Session失效。默认是真。配置下面的SecurityContextLogoutHandler。关于更多信息，请查看JavaDOC文档
6. 添加一个LogoutHandler。默认情况下，SecurityContextLogoutHandler被作为最后一个LogoutHandler添加。
7. 允许指定要在注销成功时删除的cookie的名称这是明确添加CookieClearingLogoutHandler的捷径。

登出配置也能通过XML命名空间进行配置。请查看文档中的登出元素章节查询更多细节。

一般来说，为了定制注销功能，你可以添加`LogoutHandler`和/或`LogoutSuccessHandler`实现。对于很多常见场景，当使用流API时，这些处理器会被使用。

5.5.1 LogoutHandler

一般来说，`LogoutHandler`实现就是处理注销操作的类。它用于处理必要的清除操作。他们应该补抛出异常。多种多样的实现已经被提供了：

- PersistentTokenBasedRememberMeServieces
- TokenBasedRememberMeService
- CookieClearingLogoutHandler
- CsrfLogoutHandler

- SecurityContextLogoutHandler

关于更多细节，请查看第18.4章 "Remember-Me接口和实现"。

代替直接提供LogoutHandler实现，流API提供了一个断点，用于覆盖`deleteCookies()`方法指定登出成功时删除的cookie名称，以编写特定的`LogoutHandler`。与添加CookieClearingLogoutHandler相比，这是一条捷径。

5.5.2 LogoutSuccessfulHandler

当`LogoutFilter`进行了一个成功的注销时就会调用`LogoutSuccessHandler`,这个对象去处理登出后要重定向到哪个特定的目的URL。但是注意这个接口和`LogoutHandler`是相同的，但是可能会引起异常。

下面是被提供的实现：

- SimpleUrlLogoutSuccessHandler
- HttpStatusReturningLogoutSuccessHandler

像上面提到的，你没必要直接指定`SimpleUrlLogoutSuccessHandler`。代替的是，流API提供了一个断点去设置`logoutSuccessUrl()`。这将会设置`SimpleUrlLogoutSuccessHandler`。提供的URL将在注销操作完成后被转到。默认是`/login?logout`。

`HttpStatusReturningLogoutSuccessHandler`更适用于REST API场景。代替成功后跳转到特定URL，`LogoutSuccessHandler`允许你去提供一个要被返回的HTTP状态码，默认是200。

5.5.3 和注销有关的更多文档

- 注销处理
- 登出测试
- HttpServletRequest.logout()
- 18.4节 "Remember-Me 接口和实现"
- 在CSRF说明中的注销
- Section Single Logout (CAS 协议)
- XML命名空间配置文档中的登出元素章节。

5.8 验证

我们现在只看了下基本验证设置。让我们看看一些更高级的验证设置。

5.8.1 内存验证

我们已经看到了一个简单的对单一用户的内存验证配置，下面给出一个配置多个用户的例子：

```
@Bean
public UserDetailsService userDetailsService() throws Exception
{
    UserBuilder users = User.withDefaultPasswordEncoder;
    InMemoryUserDetailsManager manager = new InMemoryUserDetailManager();
    manager.createUser(users.username("user").password("password").roles("USER").build());
    manager.createUser(users.username("admin").password("password").roles("USER","ADMIN").build());
    return manager;
}
```

5.8.2 JDBC 验证

你可以发现这个更新支持基于JDBC的验证。下面的例子要求在你的应用程序中已经定义了一个`DataSource`，这个jdbc-javaconfig例子提供了一个完整的使用基于JDBC验证的例子。

```
@Autowired
private DataSource dataSource;
```

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)throws Exception
{
    UserBuilder users = User.   withDefaultPasswordEncoder();
    auth
        .jdbcAuthentication()
            .dataSource(dataSource)
            .withDefaultSchema()
            .withUser(users.username("user").password("password").roles("USER"))
            .withUser(users.username("admin").password("password").roles("USER","ADMIN"));
}
```

5.8.3 LDAP 验证

你能看到这个更新同时支持基于LDAP的验证。ldap-javaconfig例子提供了一个完整的采用基于LDAP验证的例子。

```
@Autowired
private DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)throws Exception
{
    auth
        .ldapAuthentication()
            .userDnPatterns("uid={0},ou=people")
            .groupSearchBase("ou=groups")
}
```

上面例子使用下面的LDIF和一个嵌入式的Apache DS LDAP实体。

user.ldif.

```
dn: ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=springframework,dc=org
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=admin,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Rod Johnson
sn: Johnson
uid: admin
userPassword: password

dn: uid=user,ou=people,dc=springframework,dc=org
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Dianne Emu
sn: Emu
```

```
uid: user
userPassword: password

dn: cn=user,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: user
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
uniqueMember: uid=user,ou=people,dc=springframework,dc=org

dn: cn=admin,ou=groups,dc=springframework,dc=org
objectclass: top
objectclass: groupOfNames
cn: admin
uniqueMember: uid=admin,ou=people,dc=springframework,dc=org
```

5.8.4 AuthenticationProvider

你可以通过将自定义AuthenticationProvider作为一个bean进行定义来定义自定义身份验证。例如，下面将会自定义身份验证通过上述方式。其中SpringAuthenticationProvider实现了AuthenticationProvider。

这仅在AuthenticationManagerBuilder未被填充时才使用。

```
@Bean
public SpringAuthenticationProvider springAuthenticationProvider()
{
    return new SpringAuthenticationProvider();
}
```

5.8.5 UserDetailService

你可以定义一个UserDetailsService作为一个bean来实现自定义验证。例如下面的例子就采用上述方法，其中`SpringDataUserDetailsService`实现了`UserDetaisService`：

仅仅当AuthenticationManagerBuilder未被填充并且没有AuthenticationProviderBean被定义时才使用。

```
@Bean
public SpringDataUserDetailsService springDataUserDetailsService()
{
    return new SpringDataUserDetailsService();
}
```

你也可以自定义密码如何被加密，靠将一个`PasswordEncoder`作为一个bean。例如，如果你要使用bcrypt算法加密密码，定义如下：

```
@Bean
public BCryptPasswordEncoder passwordEncoder()
{
    return BCryptPasswordEncoder();
}
```

5.9 多样的HTTPSecurity

我们可以配置多样的HttpSecurity实例，就像我们使用`<http>`标签一样。 重要的是去继承`WebSecurityConfigurationAdapter`多次。例如，下面的代码提供了一个不同的配置关于以`/api/`开头的URL：

```
@EnableWebSecurity
public class MultiHttpSecurityConfig
{
    @Bean          1
    public UserDetails userDetailsService() throws Exception
    {
        UserBuilder users = User.withDefaultPasswordEncoder();
        InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
        manager.createUser(users.username("user").password("passoword").roles("USER").build());

manager.createUser(users.username("admin").password("password").roles("USER","ADMIN").build());
        return manager;
    }

    @Configuration
    @Order(1)        2
    public static class ApiWebSecurityConfigurationAdapter extends WebSecurityConfigurerAdapter
    {
        protected void configure(HttpSecurity http)throws Exception
        {
            http
                .antMatcher("/api/**")3
                .authorizeRequests
                    .anyRequest().hasRole("ADMIN")
                    .and()
                .httpBasic();
        }
    }

    @Configuration    4
    public static class FormLoginWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter
    {
        @Override
        protected void configure(HttpSecurity http)throws Exception
        {
            http
                .authorizeRequests()
                    .anyRequest().authenticated()
                    .and()
                .formLogin();
        }
    }

}
```

1. 像平常一样配置验证
2. 创建包含@Order的WebSecurityConfigurerAdapter去指定哪个WebSecurityConfigurerAdapter应该被第一个考虑。
3. http.antMatcher指明这个HttpSecurity将会仅仅适用于以`/api/`开头的URL。
4. 创建两一个WebSecurityConfigurerAdapter实例。如果这个URL不以`/api/`开头，这个配置将会采用。这个配置被考虑在`ApiWebSecurityConfigurationAdapter`之后因为`@Order`的值大于1,（`@Order`默认值为最后）。

5.10 方法安全

从版本2.0之后，Spring Security已提供在你的服务层方法上添加安全控制的支持。它提供了对于JSR-250注解和框架原始的`@Secured`注解的支持。从3.0版本开始你也可以使用新的基于语法的注解.您可以将安全性应用于单个bean，使用intercept-methods元素来装饰bean声明，或者可以使用AspectJ样式切入点在整个服务层中保护多个bean。

5.10.1 EnableGlobalMethodSecurity

我们使用`@EnableGlobalMethodSecurity`注解在任何`@Configuration`实例上来启动基于注解的安全。例如，下面的例子启动了Spring Security的`@Secured`注解。

```
@EnableGlobalMethodSecurity(securedEnabled = true)
public class MethodSecurityConfig
{
    //...
}
```

添加一个方法（在一个类或一个接口里）将会限制这个方法的访问。Spring Security的本地标记支持为一个方法定义一系列属性。这些方会被发送到AccessDecisionManager然后进行实际上的判断：

```
public interfacce BankService
{
    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account readAccount(Long id);

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Account[] findAccounts();

    @Secured("ROLE_TELLER")
    public Account post(Account account,double amount);
}
```

JSR-250类型的注解可以通过如下配置得到支持：

```
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig
{
    //...
}
```

这些都是基于标准的，允许应用简单的基于角色的约束，但是没有Spring Security的本地注释的强大功能。为了使用新的基于表达式的语法，你应该使用：

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig
{
    //...
}
```

并且与之相对应的Java代码如下：

```
public interface BankService
{
    @PreAuthorize("isAnonymous()")
    public Account readAccount(Long id);

    @PreAuthorize("isAnonymous()")
    public Account[] findAccounts();

    @PreAuthorize("hasAuthority('ROLE_TELLER')")
    public Account post(Account account,double amount);
}
```

5.10.2 GlobalMethodSecurityConfiguration

有时您可能需要执行比@EnableGlobalMethodSecurity批注允许的操作更复杂的操作。对于这种情况，你可以去继承GlobalMethodSecurityConfiguration确保@EnableGlobalMehtodSecurity注解能在你的子类中表示。例如你要去提供一个自定义的MethodSecurityExpressionHandler,你可以使用下面的配置;

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration
{
    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler()
    {
        // ... 创造并返回用户自定义的MethodSecurityExpressionHandler ...
        return expressionHandler;
    }
}
```

对于更多的关于能够被覆盖的方法的相关信息，请参考GlobalMethodSecurityConfiguration的JavaDoc文档。

5.10.3 EnableReactiveMethodSecurity

Spring Security支持使用ReactiveSecurityContextHolder设置的Reactor's Context进行方法安全。例如，例如，这将演示如何检索当前登录的用户的消息。

对于这个方法的返回类型一定要是一个org.rectivestreams.Publisher（i.e. Mono/Flux）。这是整合Rector's Context所必要的。

```
Authentication authentication = new TestingAuthenticationToken("user","password","ROLE_USER");

Mono<String> messageByUsername =
RectiveSecurityContextHolder.getContext().map(SecurityContext::getAuthentication)
.map(Authentication::getName)
.flatMap(this::findMessageByUsername)
//在一个WebFlux应用中，这个'subscriberContext'是使用'ReactorContextWebFilter'自动配置的
.subscriberContext(RectiveSecurityContextHolder.withAuthentication(authentication));
```

StepVerifier.create(messageByUsername) .expectNext("Hi user") .verifyComplete();

this::findMessageByUsername被定义为：

```
Mono<String> findMessageByUsername(String username)
{
    return Mono.juest("Hi"+username);
}
```

当你在应用程序中使用方法安全时下面是最小的安全配置

```
@EnableReactiveMethodSecurity
public class Security
{
    @Bean
    public MapRectiveUserDetailsService userDetailsService()
    {
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
        UserDetails rob = userBuilder.username("rob").password("rob").roles("USER").build();
        UserDetails admin =
userBuilder.username("admin").password("admin").roles("USER","ADMIN").build();
    return MapReactiveUserDetailsService(rob,admin);
    }
}
```

考虑下面的类：

```
@Component
public class HelloWorldMessageService
{
    @PreAuthorize("hasRole('ADMIN')")
    public Mono<String> findMessage()
    {
        return Mono.just("Hello World!");
    }
}
```

通过上述配置的组合，`@PreAuthorize("hasRole('ADMIN')")`将会确保`findByMessage`方法只会被带有`ADMIN`角色的用户使用。请注意，标准方法安全性中的任何表达式都适用于`@EnableReactiveMethodSecurity`。然而，现在我们只支持返回类型为`Boolean`或`boolean`的表达式，这意味着表达式不能被阻止。

当整合第5.6节，“WebFlux Security”，Reactor Context将会靠Spring Security 根据被验证的用户自动建立。

```
@EnableWebFluxSecurity
@EnableRectiveMethodSecurity
public class SecurityConfig
{
    @Bean
    SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) throws Exception
    {
        return http
            //证明方法的安全性
            //最佳做法是同时使用这两种方法进行纵深防御
            .authroizeExchange()
                .anyExchange().permitAll()
                .and()
            .httpBasic().and()
            .build();
    }

    @Bean
    MapReactiveUserDetailsService userDetailsService()
    {
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
        UserDetails rob = UserBuilder.username("rob").password("rob").roles("USER").build();
        UserDetails
admin=userBuilder.username("admin").password("admin").roles("USER","ADMIN").build();
        return new MapReactiveUserDetailsService(rob,admin);
    }
}
```

你可以在hellowebflux-method中看到一个完整的例子。

5.11 后处理配置的对象

Spring Security的Java配置没有暴露它配置的每个对象的每个属性。这是对大多数用户配置工作的简化。然而，如果每个属性都被暴露，用户可以用标准bean配置。 虽然有很多原因不直接暴露每个属性，用户可能需要更多高级配置选项。为了解决这个问题，Spring Security引入了ObjectPostProcessor的概念，该概念可用于修改或替换由Java配置创建的许多Object实例。例如，如果你想在`FilterSecurityIntercepter`上配置`filterSecurityPublishAuthorizationSuccess`属性，你可以按照如下方法做：

```
@Override
protected void configure(HttpSecurity http)throws Exception
```

```
{
    http
        .authorizeRequests()
            .anyRequest().authenticated()
            .withObjectPostProcessor(new ObjectPostProcessor<FilterSecurityInterceptor>()
            {
                public <O extends FilterSecurityInterceptor> O postProcess(O fsi)
                {
                    fsi.setPublishAutorizationSuccess(true);
                    return fsi;
                }
            })
}
```

5.12 自定义DSLs

在Spring Security中你能提供你自己的DSL。例如，你可能写了如下的东西：

```
public class MyCustomDsl extends AbstractHttpConfigurer<MyCustomDsl,HttpSecurity>
{
    private boolean flag;

    @Override
    public void init(H http)throws Exception
    {
        //任何添加另一个配置的方法必须在init方法中被执行
        http.csrf().disable();
    }

    @Override
    public void configure(H http) throws Exception
    {
        ApplicationContext context = http.getSharedObject(ApplicationContext.class);

        // here we lookup from the ApplicationContext. You can also just create a new instance.
        MyFilter myFilter = context.getBean(MyFilter.class);
        myFilter.setFlag(flag);
        http.addFilterBefore(myFilter, UsernamePasswordAuthenticationFilter.class);
    }

    public MyCustomDsl flag(boolean value)
    {
        this.flag = value;
        return this;
    }

    public static MyCustomDsl customDsl()
    {
        return new MyCustomDsl();
    }
}
```

这实际上是如何实现像HttpSecurity.authorizeRequests（）这样的方法的。

自定义的DSL按照如下方法被使用：

```
@EnableWebSecurity
public class Config extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(HttpSecurity http)throws Exception
```

```
    {
        http
            .apply(customDsl())
                .flag(true)
                .and()
            ...;
    }
}
```

代码按照如下顺序执行：

- “Config”中的代码被执行
- “MyCustomDsl”的init方法被执行
- “MyCustomDsl”的配置方法被执行

如果你愿意，你可以让WebSecurityConfiguerAdapter默认使用SpringFactories添加MyCustomDsl。例如，你想在classpath上创建一个名字为`META-INF/spring.factores`的资源：

META-INF/spring.factories.

```
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer =
sample.MyCustomDsl
```

希望禁用默认的用户可以明确地这样做。

```
@EnableWebSecurity
public class Config extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception
    {
        http
                .apply(customDsl()).disable()
                ...;
    }
}
```

关于5.6节及其5.7节内容，都是关于Spring Boot，WebFlux Security以及Spring Security如何与其他框架共同使用的，我们后来再补上。