SpringSecurity 学习札记：

1. 核心组件
2. SecurityContextHolder：一个功能对象，存储着security的上下文（SecurityContex），主要是和用户身份相关的一些信息（principal）。默认是绑定在ThreadLocal，也就是说和身份相关的请求处理完以后线程被清理，这个也就没有了。

也可以更改这个绑定策略：

SecurityContextHolder.MODE\_GLOBAL：所有线程公用同一个

SecurityContextHolder.MODE\_THREADLOCAL:默认的。一个线程一个context

SecurityContextHolder.MODE\_INHERITABLETHREADLOCAL：安全线程开启的子线程中共享context

有两种更改策略的方式：a.设置系统参数属性；b.通过SecurityContextHolder的静态方法设置

1. SecurityContext：SS中的安全上下文，主要作用是设置/获取Authentication对象
2. Authentication：一个接口。存储着用户身份相关信息

* getAuthorities()-获取用户拥有的权限/角色，一般由 AuthenticationManager设置进去
* getPrincipal()-获取身份验证的表示，一般是principal 代表了用户是谁
* getCredentials()-获取安全凭证。一般是密码
* getDetails()-一些额外的有关用户的信息，如IP地址，证书编号
* isAuthenticated/setAuthenticated 这个用户是否验证过。

1. userDetails: 一般principal都可以转换成一个UerDetails。

他代表了principal的概念，同时根据不同应用做出了扩展。他是实际应用中的用户和SS中需要的用户信息的一个适配器。一般可以通过SecurityContextHolder获得userDetails，然后转型成应用中的实际用户对象(employee,customer,user etc….)。

1. UserDetailsService：接口，用于获取userDetails

* UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

如果验证成功了，SS会用userDeatils构建Authentication对象，并保存到SecurityContext。JdbcDaoImpl、InMemoryDaoImpl都是UserDetailsService的实现类

UserDetailsService他一般是在DAO层上，用来获取实际应用中的“用户”。他只用来提供信息，不进行实践的验证操作。验证由AuthenticationManager完成（应用中定制AuthenticationProvider注册给AuthenticationManager）

1. GrantedAuthority：用户拥有的一组角色/权限。他一般代表着应用级别的角色，不要把domain级别的权限设置给他。

Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

**if** (principal **instanceof** UserDetails) {

String username = ((UserDetails)principal).getUsername();

} **else** {

String username = principal.toString();

}

1. 身份验证
2. **整体流程**：（假设用户登录成功）
3. 用户通过名称/密码登录系统--系统将username和password组装成一个UsernamePasswordAuthenticationToken（他是Authentication接口的实例）。
4. 系统验证身份 -- 将上面的token实例交给AuthenticationManager验证。
5. 获取用户信息 – AuthenticationManager根据token中的信息和验证成功后获取的信息重新建立一个Authentication对象，并返回。
6. 根据用户信息，建立安全上下文 – 将AuthenticationManager返回的对象 设置到SecurityContextHolder中

SecurityContextHolder.getContext().setAuthentication(…​)

1. ExceptionTranslationFilter：SS的Filter。用来检测SS抛出的异常（一般由AbstractSecurityInterceptor抛出）.他负责分发异常，不负责具体异常处理。

* 如果是AuthenticationException，他启动一个AuthenticationEntryPoint处理
* 如果是AccessDeniedException，他先判断用户有没有经过身份验证。如果是匿名用户，则启动一个AuthenticationEntryPoint。如果确实没有权限则委托AccessDeniedHandler进行下一步处理。
* 有一个属性requestCache：负责记录用户当前访问的资源，如果用户验明身份，可以通过他跳转到想要访问的资源。默认实现是HttpSessionRequestCache

1. AuthenticationEntryPoint：SS的登录点。他负责系统发现用户想访问一个资源，但是还没经过验证时，系统到哪里对用户进行具体的身份验证。
2. AuthenticationManager：SS身份验证的核心接口。

管理验证是否接受的核心接口 只有一个方法

Authentication authenticate(Authentication authentication) throws AuthenticationException

如果验证成功，返回完全的Authentication对象

如果失败，抛出具体的异常

1. ProviderManager：AuthenticationManager的默认实现类

* 其内部是有一组AuthenticationProvider 集合，首先他遍历这些集合，轮训每一个provider是否支持对现有用户信息的验证。如果支持，就用这个具体的authticationProvider校验身份。（只是用第一个支持的provider，之后的就不轮训了）
* 验证后，如果抛异常ProviderManager负责截获，如果返回了一个result（含有全信息的Authentication对象）。则用这个result构建全信息的authentication对象。
* 他有一个boolean属性：eraseCredentialsAfterAuthentication.即验证成功会是否擦除登录时用户输入的凭证信息。以免后续泄露造成安全问题。具体的逻辑是存放这些信息的实体类，如user类，xxxtoken类。一般会扩展CredentialsContainer接口，表明这个类含凭证信息。然后实现这个接口的eraseCredentials，置空其中的关键信息，如密码。

需要注意的是，如果你把这样一个user缓存起来，以便后续再验证时候使用，当·eraseCredentialsAfterAuthentication为true是，第一次验证成功了就把密码等信息擦除了，以后就不能再用它来进行身份验证了。有个变通的方法是，可以在验证前copy一份这个对象，这样就不会清cache里的的。也可以直接把这个属性设为false

1. AuthenticationProvider 标准接口，负责具体验证。

* provider.supports(Authentication.getClass());--每一个AuthenticationProvider实现类 都有support方法，表名自己支持什么样的Authentication对象验证
* result = provider.authenticate(authentication)；每一个AuthenticationProvider实现类 都有authenticate方法用于验证

有很多他的实现类如DaoAuthenticationProvider，LdapAuthenticationProvider CasAuthenticationProvider。他们有的可以协同工作，有的需要自己的逻辑。但是都是按照统一的流程进行。

1. DaoAuthenticationProvider他是一个重要的AuthenticationProvider 实现类

一般的应用（把用户信息存在数据库里，通过输入用户名、密码进行登录的）。都是用这个provider进行身份验证的。

他内部使用UserDetailsService当做数据访问对象（DAO）来查询实际应用中的username password 用户拥有的权限等信息。

内部的验证逻辑就是把用户输入的信息UsernamePasswordAuthenticationToken和他用useDetailsService查到的信息做判断。

有两个属性：

1. userDetailsServer：作为他内部的一个dao。如inMemoryImpl，JdbcImpl。一般这个需要我们自己实现，以符合具体的应用内部逻辑
2. PasswordEncoder：密码编码器。用来把明文密码加密的。  
   关于Password Encoding。SS说了一大堆，就是推荐这个类：

org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder

\*关于密码补充一些基本常识：

* 首先是encode（编码）：把明文密码变成我们看不到的一个串 一般由两种格式：Base64和HEX。
* 其次是Hash（散列算法）：他是把明文密码变成一种加密的长度相等的串。一般有三种MD5，SHA126,SHA256,SHA512,Bcrype.

Hash是不可逆的，而encode是可以互相转换的。

Hash虽说不可逆，但是可以通过穷举试出原始值。所以就有了salt的感念，就是在密码中加入一些干扰字符。Password = password +salt 然后对password做hash。存储好这个盐。这样每一次都是用户输入password，系统给出盐，拼合后做指定次数的hash操作，然后和系统里的已经散列化密码做对比。从而增加破解难度。

存密码一般是先做对密码做几次hash 然后再编码:MD5(MD5(“pasword”)).HEX()

* 加密、解密：如对称式加密/解密算法 AES Blowfish.这个一般就不涉及了
* Bcrypt好像能随机生成salt，并攒在passowrd串里，这样就不用我们在数据库里维护盐这个字段了。反正现在这个是默认passwordEncoder用他这个就行了。

1. 访问控制
2. SecurityContextPersistenceFilter SS的filter：

想象一个常见，一个网站，用户应只需登录一次，即可方法所有他能看到的资源，而不是每一次方法，都需要重新登录，这就需要网站能够记住用户信息。

在SS中SecurityContextPersistenceFilter负责缓存这些principal信息。他默认是把SecurityContext放到HTTPSession中，他负责从session中还原和清除SecurityContextHolder。

* 有的应用把并发请求放到一个session中，这样的话他们将共享同一个SecurityContext，尽管SS用了ThreadLocal，但是SecurityContextPersistenceFilter从session中获取的是同一个实例对象，这样的话如果你在一个线程中临时性的修改了context中的authtication对象，那么这种修改会波及到他的线程。针对这种使用情况：可定制SecurityContextPersistenceFilter的行为，让他每一个请求都new一个SecurityContext。或者你可以在你想临时改动context内容的时候调用SecurityContextHolder.createEmptyContext()，返回一个新实例，从而不影响其他的请求

1. **AccessDecisionManager**：SS中访问控制的核心接口

* 有一个decide() 方法，决定是否有权方法。如果没有，则抛出异常

需要三个参数：

1. Authentication：有用户登录填入的身份信息构建的对象
2. Object object：用户要访问的资源
3. Collection<ConfigAttribute> configAttributes：方法该资源需要的角色/权限信息

SS中可能同时存在多个AccessDecisionManager的实现类，AbstractSecurityInterceptor会根据不同的访问资源、权限信息作出判断到底使用哪一个AccessDecisionManager的实现类进行判断

* boolean supports(ConfigAttribute attribute)

根据资源需要的权限判断是否使用这个AccessDecisionManager

* boolean supports(Class<?> clazz)

根据被访问的资源决定否使用这个AccessDecisionManager

1. **AbstractSecurityInterceptor：**

在具体的应用中受保护的资源（secure object）可以是web request也可以是业务方法

SS提供Filter来控制web request安全，通过AOP来控制业务层方法访问的安全（可以使用Spring aop或者AspectJ）.不同类型的受保护资源都有其不同的interceptor类用来实现访问控制。他们功能继承了一个抽象类AbstractSecurityInterceptor。他提供了一个访问控制流程：

1. 查询资源所需的权限（"configuration attributes"）
2. 将资源、所需权限、和用户信息交给**AccessDecisionManager**来决定是否能访问
3. （可选）变更用户信息（Authentication） -RunAs的情况
4. 允许资源访问（如果有权限）
5. 如果配置了AfterInvocationManager ，当方法调用正常返回时调用它，如果调用出现异常，则不执行他。
6. **ConfigAttribute**  访问受保护资源的所需的权限信息。可以理解成一个参数接口。

SS中一般是一个含有特殊意义的字符。但实际应用中他可以是任何东西。他的作用就是传递个AccessDecisionManager做判断。所以他是什么要看AcceessDecisionManager的具体实现需要什么样参数。

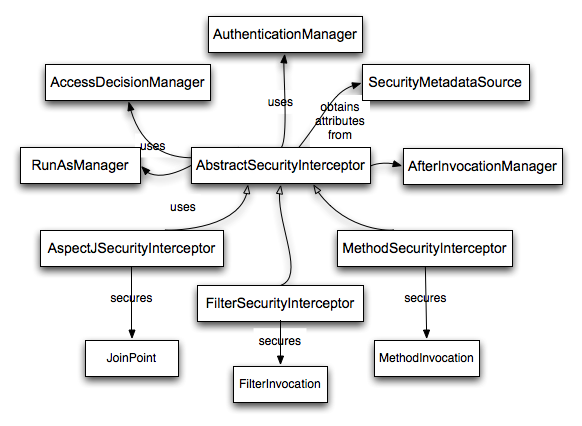
1. **SecurityMetadataSource**可以理解为存放**ConfigAttribute** 的仓库。**AbstractSecurityInterceptor**用他来查找具体资源所需要的权限。
2. **RunAsManager ：**

极少数情况，如果用户调用了一个方法，验证也成功了，但是这个方法内部需要以其他身份去访问另一个方法，这个时候需要**AbstractSecurityInterceptor**临时将SecurityContex中Authentication的替换为另一个身份（**RunAsManager**提供的另一个Authentication），待访问执行完成后再变会原有的Authentication

1. **AfterInvocationManager：**

当一个方法执行完，程序又返回到过滤器链中的**AbstractSecurityInterceptor，**他会执行**afterInvocation**，即复原**SecurityContext**（如有必要）**，**然后执行**AfterInvocationManager**（如果配置了）。在**AfterInvocationManager**中，我们可以修改方法返回的对象（个人认为是可以做一些过滤等等）。

**AbstractSecurityInterceptor相关联的对象：**



1. 测试
2. **初始化**：

*@RunWith(SpringJUnit4ClassRunner.class)*

*@ContextConfiguration*

**public** **class** WithMockUserTests {

和一般的SpringTest一模一样.SS底层使用了WithSecurityContextTestExecutionListener确保在conext载入前将SecurityContextHolder暴露到context，测试结束后，清空holder。

如果你只想测SS，不需要Spring上下文。可替换@ContextConfiguration为@SecurityExecutionListeners

1. **@WithMockUser：**模拟用户身份

使用UsernamePasswordAuthenticationToken作为Authtication实现

Principal是SS中的一个User类

1. **@WithUserDetails** 模拟用户身份

使用UsernamePasswordAuthenticationToken作为Authtication实现

Principal是SS中的一个我们自己定制的类

可以通过吧UserDetailsService0当做一个bean，暴露到上下文中，从而帮助获取具体的UserDetails实现

1. **但是principal是用户定制的一个类**
2. **@WithSecurityContext** 他是一个用于标注注解的注解。

如果要完全定制SecurityContext中的内容。可以用到它

1. 自定义一个注解，给他标注上@WithSecurityContext

@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory.class)

public @interface WithMockCustomUser

1. 实现一个WithSecurityContextFactory接口，实现接口方法：

createSecurityContext,用来填充对应的SecurityContext

自己实现的SecurityContextFactory可以当做普通的Spring bean里使用，也可以给facotory注入属性

**public** **class** WithMockCustomUserSecurityContextFactory

**implements** WithSecurityContextFactory<WithMockCustomUser> {

*@Override*

**public** SecurityContext createSecurityContext(WithMockCustomUser customUser) {

SecurityContext context = SecurityContextHolder.createEmptyContext();

CustomUserDetails principal =

**new** CustomUserDetails(customUser.name(), customUser.username());

Authentication auth =

**new** UsernamePasswordAuthenticationToken(principal, "password", principal.getAuthorities());

context.setAuthentication(auth);

**return** context;

}

}

**final** **class** WithUserDetailsSecurityContextFactory

**implements** WithSecurityContextFactory<WithUserDetails> {

**private** UserDetailsService userDetailsService;

*@Autowired*

**public** WithUserDetailsSecurityContextFactory(UserDetailsService userDetailsService) {

**this**.userDetailsService = userDetailsService;

}

**public** SecurityContext createSecurityContext(WithUserDetails withUser) {

String username = withUser.value();

Assert.hasLength(username, "value() must be non empty String");

UserDetails principal = userDetailsService.loadUserByUsername(username);

Authentication authentication = **new** UsernamePasswordAuthenticationToken(principal, principal.getPassword(), principal.getAuthorities());

SecurityContext context = SecurityContextHolder.createEmptyContext();

context.setAuthentication(authentication);

**return** context;

}

}

1. 本地化：

SS支持异常信息的本地化，但不支持调试信息本地化。

SS实现了Spring的MessageSourceAware接口，期待Application初始化的时候注入message resolver用来解析messages.properties.SS的core.jar中内含了一些messages.properties。你也可以自己定制，如

<bean id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<property name="basename" value="classpath:org/springframework/security/messages"/>

</bean>

SS需要Spring提供的localization支持，以查找具体使用哪个properties文件。所以要保证request中的local信息存储在spring的org.springframework.context.i18n.LocaleContextHolder中，在SpringMVC里，

DispatcherServlet负责存储local信息，但是SS的Filter早于servlet执行，所以要提前将local信息设置到LocaleContextHolder中。可以自己写一个Filter干这件事，或者使用Spring提供的RequestContextFilter。

注意！在WEB.XML定义中，一定要把这个Filter放到SS的之前