10.核心服务

现在我们对Spring Security的基本结构和核心类已经有了一个很深入的了解，让我们更深的了解下它的核心类和他们的实现，尤其是AuthenticationManager、UserDetailsService、AccessDecisionManager。这些文件会在本文档的其余部分定期出现，因此了解它们如何配置以及如何操作非常重要。

10.1 AuthenticationManager,ProviderManager和AuthenticationProvider

AuthenticationManager是一个接口，实现能供我们自己选择，但是它是如何工作的呢？如果我们要去检验多个认证数据库或者是一个不同认证服务的组合，例如一个数据库和一个LDAP服务器，会是怎样呢？

在Spring Security中它的默认实现是ProviderManager，它不是自己去处理这些认证请求，而是依赖一个配置好的AuthenticationProvider列表，他们中的每个都会按顺序被查询来处理认证问题。每个provider会抛出一个异常或者返回一个被填充的Authentication对象。还记得我们的好朋友（我是真不想翻译这个）,UserDetails和UserDetailsService么？如果你不记得，那么请看前一章回顾一下。最普通的验证用户认证请求的方法就是加载相应的UserDetails然后去检验这个被加载的密码和用户输入的密码是否相同。这也是DaoAuthenticationProvider所采用的方法。被加载的UserDetails对象-和特别是它包含的GrantedAuthority-当填充完整的Authentication对象时会被使用，这个Authentication对象将会在成功认证并且存储到SecurityContext时返回。

如果你使用了命名空间，那么一个ProviderManager实例将会在内部被创建和扩展，并且你可以使用命名空间认证提供器元素来为其添加provider。在这种情况下，你可以在你的应用程序上下文中不声明一个ProviderManagerbean。然而，如果你没有使用命名空间配置，那么你就要去声明它：

```
<bean id="authenticationManager"
class="org.springframework.security.authentication.ProviderManager">
    <constructor-arg>
        <list>
            <ref local="daoAuthenticationProvider"/>
            <ref local="anonymousAuthenticationProvider"/>
            <ref local="ldapAuthenticationProvider"/>
        </list>
    </constructor-arg>
</bean>
```

上面的例子中我们使用了3个provider。他们按照显示的顺序被尝试（这个功能靠使用一个列表实现的），每个provider都能够去尝试认证，或者靠返回null来跳过认证。如果所有的实现都返回了null，ProviderManager将会抛出一个ProviderNotFoundException异常。如果你对provider链感兴趣，请查看ProviderManager的JavaDOC文档。

对于认证机制，例如一个web表单登录处理过滤器通过一个ProviderManager进行注入并且调用其处理认证请求。你需要的提供器有时可能与你的认证机制是可交互的，在其他时候都依赖于特定的认证机制。例如，DaoAuthenticationProvider和LdapAuthenticationProvider和任何提交一个简单用户名/密码认证请求的机制都是兼容的。并且对于基于变淡的登录或者HTTP Basic认证也一样。另一方面，一些认证机制创建了一个认证请求对象，这种认证对象只能被单一一种AuthenticationProvider接受。就像JA-SIG CAS一样，它使用一个叫做服务单的概念，所以只有CasAuthenticationProvider能进行验证工作。你没必要关心这个，因为

如果你忘记注册一个合适的provider，当尝试进行验证的时候，你仅仅会收到一个`ProviderNotFoundException`。

10.1.1 擦除身份验证成功时的凭证

默认情况下（从Spring Security 3.1之后），`ProviderManager`将会尝试清除认证成功后返回的`Authentication`对象的所有敏感信息。这可以防止密码等信息被保留超过必要的时间。

当你使用一个缓存缓存user对象时会对提升一个无状态应用的性能产生问题。如果`Authentication`包含了一个在缓存中的引用（例如一个`UserDetails`实例）并且这个引用的凭证被移除了，那么它将不能继续进行验证工作。如果您使用缓存，则需要考虑这一点。一个明显的解决方案是先创建对象的副本，无论是在缓存实现中还是在创建返回的`Authentication`对象的`AuthenticationProvider`中。你可以在`ProviderManager`中通过`eraseCredentialsAfterAuthentication`属性中禁用这个功能。详细信息请查看JavaDOC文档。

10.1.2 DaoAuthenticationProvider

Spring Security实现的最简单的`AuthenticationProvider`是`DaoAuthenticationProvider`，它也是被框架最早支持的其中之一。它利用一个`UserDetailsService`（作为一个DAO）来查询用户名、密码和`GrantedAuthority`。它仅仅通过比较被封装在`UsernamePasswordAuthenticationToken`中的被提交的密码和`UserDetailsService`中的密码进行认证用户。例子如下：

```
<bean id="daoAuthenticationProvider"
class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
<property name="userDetailsService" ref="inMemoryDaoImpl"/>
<property name="passwordEncoder" ref="passwordEncoder"/>
</bean>
```

`PasswordEncoder`是可选的。一个`PasswordEncoder`提供了对当前的`UserDetailsService`返回的`UserDetails`对象的密码的编码和解码操作。具体请看下面章节。

10.2 UserDetailsService 实现

前面已经提到过，大多数的认证provider都使用了`UserDetails`和`UserrDetailsService`接口。回想下`UserDetailsService`要求的方法：

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

返回的`UserDetails`保证了用户名、密码、获取的授权以及账户是否禁用等认证信息是非空的。大多数的认证provider使用一个`UserDetailsService`，即使在认证决策中用户名和密码是没必要的。返回的`UserDetails`对象仅仅被用来完善`GrantedAuthority`信息，因为其他系统（像LDAP或者X.509或者CAS等）可能执行了认证验证凭证的职责。

被给定的`UserDetailsService`实现是很简单的，采用一个特定的持久化策略去检索用户的认证信息是很简单的。事实上，Spring Security已经包含了很多有用的基本实现，我们将在下面看到他们。

10.2.1 In-Memory 认证

创建并使用一个自定义UserDetailsService实现，它从一个特定的持久化引擎中提取数据，是非常简单的，而且许多应用也没有要求这么复杂。如果你正在构造一个应用原型或者仅仅刚开始集成Spring Security，你没必要去在配置数据库或写UserDetailsService实现上花费太多时间。对于这种情况，简单的基于内存的认证是更适合的。

```
<user-service id="userDetailsService">
<user name="jimi" password="{noop}jimispassword"
authorities="ROLE_USER,ROLE,ADMIN" />
<user name="bob" password="{noop}bobspassword" authorities="ROLE_USER" />
</user-service>
```

他也支持使用内部属性文件：

```
<user-service id="userDetailsService" properties="users.properties" />
```

属性文件应该为如下格式：

```
username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]
```

例如

```
jimi=jimispassowrd,ROLE_USER,ROLE_ADMIN,enabled
bob=bobspassword,ROLE_USER,enabled
```

10.2.2 JdbcDaoImpl

Spring Security也提供了一个UserDetailsService用于从一个JDBC数据源中获取认证信息。内部使用的是Spring JDBC，所以它避免了处理各类ORM，仅仅存储user细节。如果你的应用使用了一个ORM工具，你可能更想去写一个自定义UserDetailsService去重用你已经创建的映射文件。我们来看一下JdbcDaoImpl的配置文件：

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
<property name="url" value="jdbc:hsqldb:hsql://localhost:9001">
<property name="username" value="sa" />
<property name="password" value=""/>
</bean>

<bean id="userDetailsService"
class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
```

```
<property name="dataSource" ref="dataSource" />
</bean>
```

你可以更改上面的`DriverManagerDataSource`属性来换用不同的关系型数据库管理系统。你也可以使用一个从JNDI或者其他Spring配置中获取的全局数据源。

权限组

默认情况下，`JdbcDaoImpl`会为单个用户加载权限，并假定权限直接映射到用户（请参阅数据库架构附录）。另一种方法是将权限分为组并将组分配给用户。有些人更喜欢这种方法作为管理用户权利的手段。关于如何启动分组权限，请查看`JdbcDaoImpl`的JavaDoc文档。分组语法也被包含在附录中。

10.3 密码加密

Spring Security的`PasswordEncoder`接口被用来将原有密码进行单向转化以允许密码被安全的存储。鉴于`PasswordEncoder`是一种单向转换，当密码转换需要两种方式（即存储用于向数据库进行身份验证的凭证）时，并不打算这样做。通常，`PasswordEncoder`用于存储需要在验证时与用户提供的密码进行比较的密码。

10.3.1 密码历史

通过这么多年，一个标准的存储密码的机制已经形成。在开始，密码采用明文存储。密码被认为是安全的，因为数据存储密码被保存在所需的凭据中以便访问它。但是，恶意用户可以通过SQL注入等攻击找到方法来获取用户名和密码并进行大量"数据转储"。随着越来越用户凭证变成公共的，安全专家了解到我们需要去做更多去保护用户密码。

开发者之后被鼓励去对密码进行SHA-256加密后进行存储。当一个用户进行验证时，被哈希的密码将会和键入密码的哈希值进行比较。这意味着系统仅仅需要存储一个单向哈希后的密码。如果发生了违规，那么只有密码的单向散列被暴露。由于哈希是一种方法，并且在计算上很难猜测给定散列的密码，所以在系统中找出每个密码是不值得的。为了去打败这些新系统的恶意使用者决定去创建新的查询表，我们称其为Rainbow 表;他们不是每次都在猜测每个密码，而是一次计算密码并将其存储在查找表中。

为了去缓和这个Rainbow 表的影响，开发者被鼓励使用加盐的密码。代替仅仅使用密码作为哈希函数的输入，一些打乱的byte（作为盐）将会在每个用户密码存储中发挥作用。这个盐和用户密码将会通过哈希函数进而创建一个特殊的哈希值。盐将会以明文的形式与用户密码一起存储。然后，当用户试图进行认证时，被哈希的密码将会和通过存储的盐和输入的密码进行加密后的哈希值进行比较。这独特的盐似的Rainbow表不再有作用，因为每个盐和密码的组合都不同。

在现在，我们已经了解到密码哈希（例如SHA-256）已经不再安全了。原因是现代的硬件已经能达到每秒执行百万次哈希运算。这意味着我们能轻易攻破个人密码。

现在鼓励开发人员利用自适应单向函数来存储密码。使用自适应的单向函数进行密码验证是资源密集的（i.e. CPU，主存等）。自适应的单向函数允许配置一个工作因素，他能够随着硬件变好而成长。建议将"工作因素"调整为在您的系统上验证密码需要大约1秒钟的时间。这种权衡是为了让攻击者难以破解密码，也不会给你自己的系统带来过大的负担。Spring Security已经尝试去提供一个工作因素的好的开始点，但是用户被鼓励去自定义这个工作因素，因为对于不同系统之间它的表现是不同的。例如一个单项可适应的函数应该包括bcrypt，PBKDF2，scrypt，Argon2。

因为适应性单项函数是资源密集的，验证每个请求的用户名和密码将显着降低应用程序的性能。Spring Security（或者是任何其他库）都不能取加速密码验证，因为这个验证是资源密集的。用户被鼓励使用短期凭证

代替长期凭证，短期凭证能被更快的验证而且不会在安全上有损失。

10.3.2 DelegatingPasswordEncoder

在Spring Security 5.0之前默认的`PasswordEncoder`是`NoOpPasswordEncoder`，它要求明文密码。通过阅读上面的密码历史章节，你可能期望默认的`PasswordEncoder`是像`BCryptPasswordEncoder`这类。然而，这里忽视了3个真实世界中的问题：

- 许多应用使用的是老的密码编码方式，很难去更改
- 最好的密码存储时间可能会再次更改
- Spring Security作为一个框架不能频繁做出巨大改变

代替的是Spring Security使用`DelegatingPasswordEncoder`解决所有的问题：

- 确保使用当前密码存储建议对密码进行编码
- 允许验证现代和传统格式的密码
- 允许将来升级编码

你可以使用`PasswordEncoderFactories`轻易的创建一个`DelegatingPasswordEncoder`实例。

```
PasswordEncoder passwordEncoder =
PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

可选择的是，你可以创建属于你自己的自定义实例。例如：

```
String idForEncode = "bcrypt";
Map encoders = new HashMap<>();
encoders.put(idForEncode,new BCryptPasswordEncoder());
encoders.put("noop",NoOpPasswordEncoder.getInstance());
encoders.put("pbkdf2",new Pbkdf2PasswordEncoder());
encoders.put("scrypt",new SCryptPasswordEncoder());
encoders.put("sha256",new StandardPasswordEncoder());

PasswordEncoder passwordEncoder = new
DelegatingPasswordEncoder(idForEncode,encoders);
```

密码存储格式

一般的密码格式为

```
{id}encodedPassword
```

`id`是一个标识符，这个标识符标识哪个`PassswordEncoder`应该被使用，并且`encodedPassword`是被选择的`PasswordEncoder`的原始被编码的密码。`id`一定要在密码的开始，并且使用`{}`包裹。如果`id`不能被找到，那么`id`将会为空。例如下面的密码是使用不同`id`的编码密码，所有的原始密码都是“password”：

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG 1
{noop}password 2
{pbkdf2}5d923b44a6d129f3ddf3e3c8d29412723dcbde72445e8ef6bf3b508fbf17fa4ed4d6b99ca7
63d8dc 3
{scrypt}$e0801$8bWJaSu2IKSn9Z9kM+TPXfOc/9bdYSrN1oD9qfVThWEwdRTnO7re7Ei+fUZRJ68k9lT
yuTeUp4of4g24hHnazw==$OAOec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=  4
```

第一个密码的前缀是{bcrypt}，编码后的密码是$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG，当进行匹配的时候，会被代理到`BCryptPasswordEncoder`

第二个密码前缀是{noop}，当进行匹配时，会被代理到`NoOpPasswordEncoder`

第三个密码前缀是{pbkdf2}，当进行匹配时，会被代理到`Pbkdf2PasswordEncoder`

第四个密码前缀是{scrypt}，当金星匹配时，会被代理到`SCryptPasswordEncoder`

第五个密码前缀是`sha256`，当进行匹配时，会被代理到`StandardPasswordEncoder`

一些用户可能会担心存储格式是为潜在的黑客提供的。不用担心这个问题，因为密码的存储不依赖于算法。此外，就算没有前缀，攻击者也很容易找到具体的密码是从哪里开始的。例如，BCrypt密码经常以`$2a$`开始。

密码加密

`idForEncode`传给构造器，用来决定哪个`PasswordEncoder`将会被用来加密密码。在按照上面的方法构造的`DelegatingPasswordEncoder`，意味着前缀为`{bcrypt}`的加密的密码将会被代理到`BCryptPasswordEncoder`。最后结果如下：

```
{bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

密码匹配

匹配是基于`{id}`被做的，与`id`相对应的`PasswordEncoder`将会被提供在构造器里。在密码存储格式中的例子提供了一个如何工作的很好的例子。默认情况下，将会返回`matches(CharSequence,String)`的结果或者如果`id`没有匹配，则抛出一个`IllegalArgumentException`。这个表现能够被自定义化靠使用`DelegatingPasswordEncoder.setDefaultPasswordEncoderForMatches(PasswordEncoder)`。

靠使用`id`我们能匹配任何密码编码，但是我们应该使用最新的密码编码方式来对密码进行编码。这是很重要的，因为不像加密，密码哈希被设计用来没有一个简单的方式去恢复密码明文。因为无法恢复明文，所以去迁移密码就是很难的。虽然用户迁移`NoOpPasswordEncoder`非常简单，但我们选择默认包含它以简化入门体验。

开始使用

如果你正在做一个demo或者一个简单的例子，那么花费时间去哈希用户密码是非常费时费力的。这里有很方便的机制去使它更简单，但是这依旧不适用于生产环境。

```
User user = User.withDefaultPasswordEncoder()
.username("user")
.password("password")
.roles("user")
.build();
System.out.println(user.getPassword());
// {bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
```

如果你想创建多个用户，你可以重用这个构建器。

```
UserBuilder users = User.withDefaultPasswordEncoder();
User user = users
.username("user")
.password("password")
.roles("USER")
.build();
User admin = users
.username("admin")
.password("password")
.roles("USER","ADMIN")
.build();
```

虽然被哈希的密码被存取了，但是密码依旧在内存和源码中被暴露了。因此，这种方式对生产环境来说是不安全的。对于产品，你应该考虑去在内部哈希你的密码。

故障排除

当你在密码存储格式章节中，当你存储一个密码，但是他没有id描述时会出现如下错误。

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id
"null"
at
org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdP
asswordEncoder.matches(DelegatingPasswordEncoder.java:233)
at
org.springframework.security.crypto.password.DelegatingPasswordEncoder.matches(Del
egatingPasswordEncoder.java:196)
```

最简单去解决这个错误的方法是切换到编码你密码的`PasswordEncoder`。这个方法要求你去弄清你的密码是如何存储的，并且提供正确的`PasswordEncoder`。如果你使用Spring Security 4.2.x版迁移，你可以通过暴露`NoOpPasswordEncoder` bean恢复到以前的行为。例如，如果你使用Java配置的话，你可以如下配置：

但是`NoOpPasswordEncoder`是不安全的。你应该去使用`DelegatingPasswordEncoder`去支持密码编码安全。

```
@Bean
public static NoOpPasswordEncoder passwordEncoder()
{
    return NoOpPasswordEncoder.getInstance();
}
```

如果你使用的XML配置，你应该使用一个`PasswordEncoder`的id去暴露它。

```
<b:bean id="passwordEncoder"
class="org.springframework.security.crypto.password.NoOpPasswordEncoder" factory-
method="getInstance"/>
```

你可以将你的密码都标上正确的前缀id并且使用`DelegatingPasswordEncoder`。例如，如果你使用的是BCrypt，你可以将你的密码从： $2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG 换为： {bcrypt}$2a$10$dXJ3SW6G7P50lGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG 对于映射的完整列表，你可以参考PassworEncoderFactory的Java文档。

10.3.3 BCryptPasswordEncoder

`BCryptPasswordEncoder`实现使用了被广泛支持的bcrypt算法去对密码进行哈希。为了能让它更能抵制密码破解，bcrypt故意放慢了速度。就像其它的适应性单项方法一样，他应该在你的系统上被调节为大约1秒认证要给密码。

```
// Create an encoder with strength 16
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(16);
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

10.3.4 Pbkdf2PasswordEncoder

`Pbkdf2PasswordEncoder`实现使用了PBKDF2算法去对密码进行哈希。为了去客服密码破解，PBKDF2也被故意放慢了速度。就像其他的适应性单项方法，他被调整在大约1秒认证一个密码。当FIPS认证被要求时这个算法是个很好的选择。

```
// Create an encoder with all the defaults
Pbkdf2PasswordEncoder encoder = new Pbkdf2PasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

10.3.5 SCryptPasswordEncoder

SCryptPasswordEncoder实现使用scrypt算法去对密码进行哈希。为了去防止密码破解，scrypt算法被通过自定义硬件来放慢速度，这个算法需要大量内存。像其他的适应性单向方法一样，应该1秒验证一个密码：

```
// Create an encoder with all the defaults
SCryptPasswordEncoder encoder = new SCryptPasswordEncoder();
String result = encoder.encode("myPassword");
assertTrue(encoder.matches("myPassword", result));
```

10.3.6 其他的PasswordEncoders

有大量的其他PasswordEncoder实现完全为了向后兼容而存在。他们都被弃用，表明他们不再被认为是安全的。但是，由于难以迁移现有的遗留系统，因此没有计划将其删除。

10.4 Jackson 支持

Spring Security 已经为persisting Spring Security的相关类添加了Jackson支持。当与分布式session工作时，会提高Spring Security的相关类的同步性能。

为乐趣使用它，要注册JacksonJacksonModules.getModules(ClassLoader)作为Jackson 模块。

```
ObjectMapper mapper = new ObjectMapper();
ClassLoader loader = getClass().getClassLoader();
List<Module> modules = SecurityJackson2Modules.getModules(loader);
mapper.registerModules(modules);

// ... use ObjectMapper as normally ...
SecurityContext context = new SecurityContextImpl();
// ...
String json = mapper.writeValueAsString(context);
```